**Лекция 1. Основные понятия операционных сред, систем и оболочек**

Классификация программного обеспечения:

* операционные системы и сервисные программы;
* инструментальные языки и системы программирования;
* прикладные системы.

Операционная система — это комплекс управляющих и обрабатывающих программ, который, с одной стороны, выступает как интерфейс между аппаратной частью компьютера и пользователем с его задачами, а с другой — предназначен для наиболее эффективного использования ресурсов вычислительной системы и организации надежных вычислений.

**Основные функции операционных систем:**

* Прием от пользователя или от оператора системы заданий или команд, сформулированных на соответствующем языке в виде команд оператора или указаний с помощью соответствующего манипулятора (например, с помощью мыши), и их обработка.
* Прием и исполнение программных запросов на запуск, приостановку, остановку других программ.
* Загрузка в оперативную память подлежащих исполнению программ.
* Инициация программы — передача данной конкретной программе управления, в результате чего процессор приступает к ее выполнению.
* Идентификация всех программ и данных.
* Обеспечение режима мультипрограммирования, то есть выполнение двух или более программ на одном процессоре, создающее видимость их одновременного исполнения.
* Организация и управление всеми операциями ввода/вывода.
* Исполнение режима жестких ограничений на время ответа в режиме реального времени (характерно для ОС реального времени).
* Распределение памяти и организация виртуальной памяти.
* Планирование и диспетчеризация задач в соответствии с заданной стратегией и дисциплиной обслуживания.
* Организация механизмов обмена сообщениями и данными между выполняющимися программами.
* Защита одной программы от влияния другой и обеспечение сохранности данных.
* Предоставление услуг в случае частичного сбоя системы.
* Обеспечение работы систем программирования, с помощью которых пользователи создают свои программы.
* Обеспечение работы систем управления базами данных (СУБД).
* Обеспечение работы систем управления файлами (СУФ).

Операционная система, выполняя функции управления вычислительными процессами в вычислительной системе, распределяет ресурсы вычислительной системы между различными вычислительными процессами и образует программную среду, в которой выполняются прикладные программы пользователей, называемую операционной средой.

Операционная среда — это набор соответствующих интерфейсов, необходимых программам и пользователям для обращения к ОС с целью получения определенных сервисов.

Для удобства взаимодействия с ОС могут использоваться дополнительные интерфейсные оболочки.

Их основное назначение — расширение возможностей по управлению ОС и изменение встроенных в систему возможностей под конкретные требования пользователя.

Операционная среда определяется программным интерфейсом API (Application Program Interface), включающим в себя управление процессами, памятью и вводом/выводом.

В составе ОС присутствуют сервисные программы (утилиты ОС).

Это специальные системные программы, с помощью которых можно как обслуживать саму операционную систему, так и подготавливать для работы носители данных, выполнять перекодирование данных, осуществлять оптимизацию размещения данных на носителе и производить некоторые другие работы, связанные с обслуживанием вычислительной системы.

Инструментальные языки и системы программирования представляются, прежде всего, такими компонентами, как транслятор с соответствующего языка, библиотеки подпрограмм, редакторы, компоновщики и отладчики.

Прикладными системами является ПО, ориентированное на автоматизацию конкретных видов деятельности, например обучение определенным предметам, проектирование электронных изделий или строительных сооружений, анализ электрокардиограмм, проведение финансовых расчетов и многое другое. Кроме того, прикладные системы могут также обеспечивать автоматизацию таких общих функций, присущих многим видам деятельности, как формирование и печать различных документов, хранение и выдача справок и т.п.

**Классификация операционных систем**

В зависимости от особенностей используемого алгоритма управления процессором выделяют следующие типы операционных систем:

* многозадачные и однозадачные,
* многопользовательские
* однопользовательские;
* многопроцессорные и однопроцессорные системы;
* системы, поддерживающие и не поддерживающие распараллеливания вычислений в рамках одной задачи.

Главное отличие многопользовательских систем от однопользовательских — наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей.

Многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой:

* Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам.
* Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

Классификация многозадачных ОС с учетом критериев эффективности:

* системы пакетной обработки (OC EC)
* системы разделения времени (UNIX, Windows) - системы реального времени (QNX)

**Основные принципы построения операционных систем**

* Принцип модульности
* Принцип функциональной избирательности
* Принцип генерируемости ОС
* Принцип функциональной избыточности
* Принцип виртуализации
* Принцип независимости программ от внешних устройств
* Принцип совместимости
* Принцип открытости и наращиваемости
* Принцип мобильности (переносимости)
* Принцип обеспечения безопасности вычислений

**Принципы построения ядра операционных систем**

По основному архитектурному принципу ОС разделяются на микроядерные и монолитные.

Основой модульных и переносимых расширений является микроядро — минимальная стержневая часть операционной системы.

Микроядро является маленьким, передающим сообщения модулем системного программного обеспечения, работающим в наиболее приоритетном состоянии компьютера и поддерживающим остальную часть операционной системы, рассматриваемую как набор серверных приложений.

Микроядро включает только те функции, которые требуются для определения набора абстрактных сред обработки прикладных программ и организации совместной работы приложений при обеспечении сервисов и взаимодействия клиентами и серверами.

В результате микроядро обеспечивает только пять различных типов сервисов:

1. управление виртуальной памятью;
2. управление задания и потоки;
3. поддержку межпроцессных коммуникаций IPC (Inter- Process Communication);
4. управление поддержкой ввода/вывода и прерываниями;
5. поддержку сервисов набора хоста и процессора.

Наиболее ярким представителем микроядерных ОС является ОС реального времени QNX (Queue Nicks, с англ. очередь точных моментов времени, или очередь зарубок).

* Микроядро QNX поддерживает только планирование и диспетчеризацию процессов, взаимодействие процессов, обработку прерываний и сетевые службы нижнего уровня.
* Микроядро обеспечивает всего лишь около двух десятков системных вызовов, но благодаря этому оно может быть целиком размещено во внутреннем кэше даже таких процессоров, как Intel 486.
* Разные версии этой ОС имели и различные объемы ядер — от 8 до 46 Кбайт.

Монолитные ОС являются прямой противоположностью микроядерным ОС.

В монолитной ОС несмотря на ее возможную сильную структуризацию, достаточно трудно удалить какой-либо уровень многоуровневой модульной структуры.

Добавление новых функций и изменение существующих для монолитных ОС требует глубокого знания всей архитектуры ОС и чрезвычайно больших усилий.

Поэтому более современный подход к проектированию ОС, который может быть условно назван «клиент-серверной технологией», позволяет в большем количестве и с меньшими трудозатратами реализовать перечисленные выше принципы проектирования ОС.

**Принципы построения интерфейсов операционных систем**

ОС всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами.

Под интерфейсами ОС здесь и далее следует понимать специальные интерфейсы системного и прикладного программирования, предназначенные для выполнения следующих задач:

* управление процессами;
* управление памятью;
* управление вводом/выводом.

GUI (Graphical User Interface)

Управление GUI — частный случай задачи управления вводом/выводом, не являющийся частью ядра операционной системы, хотя в ряде случаев разработчики ОС относят функции GUI к основному системному API.

API (Application Program Interface)  
API – интерфейс прикладного программирования

RTL (Run Time Lubrary)

RTL включает в себя те стандартные подпрограммы, которые система программирования подставляет на этапе компиляции. В общем случае RTL включает в себя не только модули из системы программирования, но и модули самой ОС.

**Общее представление об интерфейсе прикладного программирования**

Интерфейс прикладного программирования API (Application Program Interface) можно разделить на следующие направления:

* API как интерфейс высокого уровня, принадлежащий к библиотекам RTL;
* API прикладных и системных программ, входящих в поставку операционной системы;
* прочие API.

Интерфейс прикладного программирования предназначен для использования прикладными программами системных ресурсов операционной системы и реализуемых ею функций.

API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС.

API представляет собой набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы с целевой вычислительной системой.

Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа, порождаемая системой программирования на основании кода исходной программы, созданного разработчиком, а также объектных модулей и библиотек, входящих в состав системы программирования.

**Функции API на различных уровнях реализации**

Существует несколько вариантов реализации API:

* реализация на уровне ОС;
* реализация на уровне системы программирования;
* реализация на уровне внешней библиотеки процедур и функций.

Возможности API можно оценивать со следующих позиций:

* эффективностью выполнения функций API, характеризуемой скоростью выполнения функций и объемом вычислительных ресурсов, требующихся для их выполнения;
* широтой предоставляемых возможностей;
* степенью зависимости прикладной программы от архитектуры целевой вычислительной системы.

При реализации функций API на уровне ОС за их выполнение ответственность несет ОС.

Объектный код, выполняющий функции, либо непосредственно входит в состав ОС (или даже ядра ОС), либо поставляется в составе динамически загружаемых библиотек, разработанных для данной ОС.

Система программирования ответственна только за организацию интерфейса для вызова этого кода.

В таком варианте результирующая программа обращается непосредственно к ОС. Поэтому достигается наибольшая эффективность выполнения функций API по сравнению со всеми другими вариантами реализации API.

Недостатком организации API по такой схеме является практически полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы. Программа, созданная для одной архитектуры вычислительной системы, не сможет исполняться на вычислительной системе другой архитектуры даже после того, как ее объектный код будет полностью перестроен. Чаще всего система программирования не сможет выполнить перестроение исходного кода для новой архитектуры вычислительной системы, поскольку многие функции API, ориентированные на определенную ОС, будут в новой архитектуре просто отсутствовать.

Если функции API реализуются на уровне системы программирования, они предоставляются пользователю в виде библиотеки функций соответствующего языка программирования.

Обычно речь идет о библиотеке времени исполнения RTL.

Система программирования предоставляет пользователю библиотеку соответствующего языка программирования и обеспечивает подключение к результирующей программе объектного кода, ответственного за выполнение этих функций.

Очевидно, что эффективность функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям ОС. Это происходит вследствие того, что для выполнения многих функций API библиотека RTL языка программирования должна все равно выполнять обращения к функциям ОС. Наличие всех необходимых вызовов и обращений к функциям ОС в объектном коде RTL обеспечивает система программирования.

Однако переносимость исходного кода программы в таком варианте будет самой высокой, поскольку синтаксис и семантика всех функций будут строго регламентированы в стандарте соответствующего языка программирования. Они зависят от языка и не зависят от архитектуры целевой вычислительной системы. Поэтому для выполнения прикладной программы на новой архитектуре вычислительной системы достаточно заново построить код результирующей программы с помощью соответствующей системы программирования.

Реализация функций API с помощью внешних библиотек осуществляется посредством предоставления пользователю данных функций в виде библиотеки процедур и функций, созданной сторонним разработчиком.

Причем разработчиком такой библиотеки может выступать тот же самый производитель.

Система программирования ответственна только за подключение объектного кода библиотеки к результирующей программе.

Причем внешняя библиотека может быть и динамически загружаемой, т.е. загружаемой во время выполнения программы.

С точки зрения эффективности выполнения этот метод реализации API имеет самые низкие результаты, поскольку внешняя библиотека обращается как к функциям ОС, так и к функциям RTL языка программирования.

Только при очень высоком качестве внешней библиотеки ее эффективность становится сравнимой с библиотекой RTL.

Если говорить о переносимости исходного кода, то здесь требование только одно — используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Тогда удается достигнуть переносимости. Это возможно, если используемая библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

**Платформенно-независимый интерфейс POSIX**

Платформенно-независимый системный интерфейс для компьютерного окружения POSIX (Portable Operating System Interface for Computer Environments) — это стандарт IEEE, описывающий системные интерфейсы для открытых ОС, в том числе оболочки, утилиты и инструментарии.

IEEE (Institute of Electrical and Electronics Engineers) — институт инженеров по электротехнике и радиоэлектронике.

Стандарт базируется на UNIX-системах, но допускает реализацию и в других ОС.

**Лекция 2. Теория построения операционных систем**

**Процессы**

Понятие «вычислительный процесс» (или просто — «процесс») является одним из основных при рассмотрении операционных систем. Под процессом понимают выполняемую программу, включающую текущие значения счетчика команд, регистров и переменных. Впервые концепция процесса была реализована в операционных системах 1960-х годов (MULTICS, 1965; THE, 1968; RC4000, 1970).

По принципу выполнения различают последовательные процессы и параллельные. Последовательный процесс, иногда называемый «задачей», — это выполнение отдельной программы с ее данными на последовательном процессоре. В концепции, которая получила наибольшее распространение в 70-е годы, под задачей (task) понимается совокупность связанных между собой и образующих единое целое программных модулей и данных, требующая ресурсов вычислительной системы для своей реализации. В последующие годы задачей стали называть единицу работы, для выполнения которой предоставляется центральный процессор.

**Ресурсы**

Определение концепции процесса преследует цель выработать механизмы распределения и управления ресурсами. Понятие ресурса вычислительного процесса при рассмотрении операционных систем является не менее важным. Термин «ресурс» обычно применяется по отношению к неоднократно используемым, относительно стабильным и «дефицитным» объектам, которые запрашиваются, используются и освобождаются процессами в период их активности. Другими словами, ресурсом является любой объект, который может распределяться внутри системы

**Классификация ресурсов**

Ресурсы могут быть разделяемыми, когда несколько процессов могут их использовать одновременно (в один и тот же момент времени) или параллельно (в течение некоторого интервала времени процессы используют ресурс попеременно), а могут быть и неделимыми

При разработке первых систем программирования под понятием «ресурсы» понимали процессорное время, память, каналы ввода/вывода и периферийные устройства

В настоящее время понятие ресурса превратилось в абстрактную структуру с целым рядом атрибутов, характеризующих способы доступа к этой структуре и ее физическое представление в системе.

**Мультипрограммирование**

Мультипрограммирование — это режим обработки данных, при котором на одном процессоре попеременно выполняется несколько программ. При мультипрограммировании повышается пропускная способность системы, но время выполнения отдельного процесса никогда не превышает времени выполнения данного процесса в однопрограммном режиме. Всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождающегося ресурса.

**Дескриптор процесса**

В общем случае дескриптор процесса содержит следующую информацию:

* идентификатор процесса PID (Process Identificator);
* тип (или класс) процесса, который определяет для супервизора некоторые правила предоставления ресурсов;
* приоритет процесса, в соответствии с которым супервизор предоставляет ресурсы. В рамках одного класса процессов в первую очередь обслуживаются более приоритетные процессы; •переменную состояния, которая определяет, в каком состоянии находится процесс (готов к работе, в состоянии выполнения, ожидание устройства ввода/вывода и т. д.);
* защищенную область памяти (или адрес такой зоны), в которой хранятся текущие значения регистров процессора, если процесс прерывается, не закончив работы. Эта информация называется контекстом задачи;
* сведения о ресурсах, которыми процесс владеет и/или имеет право пользоваться (указатели на открытые файлы, информация о незавершенных операциях ввода/вывода и т. п.);
* место (или его адрес) для организации общения с другими процессами;
* параметры времени запуска (момент времени, когда процесс должен активизироваться, и периодичность этой процедуры); в случае отсутствия системы управления файлами — адрес задачи на диске в ее исходном состоянии и адрес на диске, куда она выгружается из оперативной памяти, если ее вытесняет другая (для диск-резидентных задач, которые постоянно находятся во внешней памяти на системном магнитном диске и загружаются в оперативную память только на время выполнения).

**Потоки (треды).**

Понятие процесса было введено при реализации мультипрограммного режима работы вычислительной техники. В свое время различали термины «мультизадачность» и «мультипрограммирование». Для реализации «мультизадачности» в ее исходном толковании необходимо было тоже ввести соответствующую сущность. Такой сущностью и стали так называемые легковесные процессы, или, как их теперь преимущественно называют, — потоки или треды (нити).

Главное свойство многопоточности, — это возможность параллельно выполнять несколько видов операций в одной прикладной программе. Параллельные вычисления (а следовательно, и более эффективное использование ресурсов центрального процессора, и меньшее суммарное время выполнения задач) теперь уже часто реализуется на уровне тредов, и программа, оформленная в виде нескольких тредов в рамках одного процесса, может быть выполнена быстрее за счет параллельного выполнения ее отдельных частей.

Сущность «поток» была введена для того, чтобы с помощью именно этих единиц распределять процессорное время между возможными работами. Сущность «процесс» предполагает, что при диспетчеризации нужно учитывать все ресурсы, закрепленные за ним. А при манипулировании тредами можно менять только контекст задачи (образ ее текущего состояния) при переключении с одной задачи на другую в рамках одного процесса. Все остальные вычислительные ресурсы при этом не затрагиваются. Каждый процесс всегда состоит, по крайней мере, из одного потока, и только если имеется внутренний параллелизм, программист может «расщепить» один тред на несколько параллельных.

Каждый тред выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Треды, как и процессы, могут порождать треды- потомки, поскольку любой процесс состоит по крайней мере из одного треда. Подобно традиционным процессам (то есть процессам, состоящим из одного треда), каждый тред может находиться в одном из активных состояний. Пока один тред заблокирован (или просто находится в очереди готовых к исполнению задач), другой тред того же процесса может выполняться. Треды разделяют процессорное время так же, как это делают обычные процессы, в соответствии с различными вариантами диспетчеризации.

Так как все треды имеют одно и то же виртуальное адресное пространство своего процесса, они разделяют одни и те же глобальные переменные. Поскольку любой тред может иметь доступ к любому виртуальному адресу, один тред может использовать стек другого треда. Между потоками нет полной защиты, так как это, во-первых, невозможно, а во-вторых, не нужно.

Кроме разделения адресного пространства, все треды совместно используют также набор открытых файлов, общие устройства, выделенные процессу, имеют одни и те же наборы сигналов, семафоры и т.п. Собственными у тредов являются программный счетчик, стек, рабочие регистры процессора, потоки-потомки, состояние.

**Советы по использованию потоков при создании приложений:**

1. В случае использования однопроцессорной системы множество параллельных потоков зачастую не ускоряет работу приложения, поскольку в каждый отдельно взятый промежуток времени возможно выполнение только одного потока. Кроме того, чем больше у вас потоков, тем больше нагрузка на систему вследствие переключения между ними. Если ваш проект имеет более двух постоянно работающих потоков, которые требуют частого ввода/вывода, то в этом случае применение режима мультизадачности не сделает программу быстрее.
2. Прежде всего необходимо определить цели создания потока. Поток, осуществляющий обработку, может мешать быстрому реагированию системы на запросы ввода/вывода. Потоки позволяют программе отзываться на просьбы пользователя и устройств, но при этом сильно загружают процессор. Потоки позволяют компьютеру одновременно обслуживать множество устройств, и созданный вами поток, отвечающий за обработку специфического устройства, в качестве минимума может потребовать столько времени, сколько системе необходимо для обработки запросов всех устройств.
3. Потокам можно назначить определенный приоритет для того, чтобы наименее значимые процессы выполнялись в фоновом режиме. Это путь честного разделения ресурсов процессора. Однако необходимо осознать тот факт, что процессор один на всех, а потоков много. Если в вашей программе главная процедура передает нечто для обработки в низкоприоритетный поток, то сама программа становится просто неуправляемой.
4. Потоки эффективно работают, когда они независимы. Но они начинают работать непродуктивно, если достаточно часто возникает необходимость в процессах синхронизации для доступа к общим ресурсам. Блокировка и критические секции отнюдь не увеличивают скорость работы системы, хотя без использования этих механизмов взаимодействующие вычисления организовывать нельзя.
5. Помните, что память виртуальна. Механизм виртуальной памяти следит за тем, какая часть виртуального адресного пространства должна находиться в оперативной памяти, а какая должна быть сброшена в файл подкачки. Потоки усложняют ситуацию, если они обращаются в одно и то же время к разным адресам виртуального адресного пространства приложения. Это значительно увеличивает нагрузку на систему, особенно при небольшом объеме кэш-памяти.
6. Всякий раз, когда какой-либо из потоков пытается воспользоваться общим ресурсом вычислительного процесса, которому он принадлежит, вы обязаны тем или иным образом легализовать и защитить свою деятельность. Хорошим средством для этого являются критические секции, семафоры и очереди сообщений. Если вы протестировали свое приложение и не обнаружили ошибок синхронизации, то это еще не значит, что их там нет. Пользователь может создать самые непредсказуемые ситуации. Это очень ответственный момент в разработке многопоточных приложений.
7. Не возлагайте на поток несколько функций. Сложные функциональные отношения затрудняют понимание общей структуры приложения, его алгоритм. Чем проще и однозначнее каждая из рассматриваемых ситуаций, тем больше вероятность исключения ошибок.

**Прерывания.**

Прерывания представляют собой механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора.

Таким образом, прерывание — это принудительная передача управления от выполняемой программы к системе, а через нее — к соответствующей программе обработки прерывания, происходящая при возникновении определенного события.

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

Механизм обработки прерываний независимо от архитектуры вычислительной системы включает следующие шаги:

1. установление факта прерывания (прием сигнала на прерывание) и идентификация прерывания (в операционных системах иногда осуществляется повторно на шаге 4);
2. запоминание состояния прерванного процесса, определяемое, прежде всего, значением счетчика команд, содержимым регистров процессора и может включать также спецификацию режима (например, режим пользовательский или привилегированный) и другую информацию;
3. аппаратная передача управления подпрограмме обработки прерывания. В простейшем случае в счетчик команд заносится начальный адрес подпрограммы обработки прерываний, а в соответствующие регистры — информация из слова состояния. В более развитых процессорах, например в i80286 и последующих 32- битовых микропроцессорах, начиная с i80386, осуществляется достаточно сложная процедура определения начального адреса соответствующей подпрограммы обработки прерывания и не менее сложная процедура инициализации рабочих регистров процессора;
4. сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью действий аппаратуры. В некоторых вычислительных системах предусматривается запоминание довольно большого объема информации о состоянии прерванного процесса;
5. обработка прерывания. Эта работа может быть выполнена той же подпрограммой, которой было передано управление на шаге 3, но в ОС чаще всего она реализуется путем последующего вызова соответствующей подпрограммы;
6. восстановление информации, относящейся к прерванному процессу (шаг, обратный шагу 4).
7. Возврат в прерванную программу.

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

Прерывания, возникающие при работе вычислительной системы, можно разделить на два основных класса: внешние (их иногда называют асинхронными) и внутренние (синхронные).

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

* прерывания от таймера;
* прерывания от внешних устройств (по вводу/выводу); прерывания по нарушению питания;
* прерывания с пульта оператора вычислительной системы;
* прерывания от другого процессора или другой вычислительной системы.

Внутренние прерывания вызываются событиями, которые связаны с работой процессора и являются синхронными с его операциями. Примерами могут служить следующие запросы на прерывания:

* при нарушении адресации (в адресной части выполняемой команды указан запрещенный или несуществующий адрес, обращение к отсутствующему сегменту или странице при организации механизмов виртуальной памяти);
* при наличии в поле кода операции незадействованной двоичной комбинации;
* при делении на нуль;
* при переполнении или исчезновении порядка;
* при обнаружении ошибок четности, ошибок в работе различных устройств аппаратуры средствами контроля.

Существуют собственно программные прерывания. Такие прерывания происходят по соответствующей команде прерывания, то есть по этой команде процессор осуществляет практически те же действия, что и при обычных внутренних прерываниях. Данный механизм был специально введен для того, чтобы переключение на системные программные модули происходило не просто как переход в подпрограмму, а точно таким же образом, как и обычное прерывание. Этим обеспечивается автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд.

Сигналы, вызывающие прерывания, формируются вне процессора или в самом процессоре; они могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому типу прерывания.

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы. Процессор может обладать средствами защиты от прерываний: отключение системы прерываний, маскирование (запрет) отдельных сигналов прерывания. Программное управление этими средствами посредством специальных команд позволяет операционной системе регулировать обработку сигналов прерывания, заставляя процессор обрабатывать их сразу по поступлению, откладывать их обработку на некоторое время или полностью игнорировать.

Программное управление специальными регистрами маски (маскирование сигналов прерывания) позволяет реализовать различные дисциплины обслуживания:

* с относительными приоритетами, то есть обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса обслуживается запрос с наивысшим приоритетом. Для организации такой дисциплины необходимо в программе обслуживания данного запроса наложить маски на все остальные сигналы прерывания или просто отключить систему прерываний;
* с абсолютными приоритетами, то есть всегда обслуживается прерывание с наивысшим приоритетом. Для реализации этого режима необходимо на время обработки прерывания замаскировать все запросы с более низким приоритетом. При этом возможно многоуровневое прерывание, то есть прерывание программ обработки прерываний. Число уровней прерывания в этом режиме изменяется и зависит от приоритета запроса;
* по принципу стека, то есть запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Для этого необходимо не накладывать маски ни на один сигнал прерывания и не выключать систему прерываний.

**Лекция 3. Теория построения операционных систем**

**Планирование процессов и диспетчеризация задач**

Операционная система выполняет следующие основные функции, связанные с управлением задачами:

* создание и удаление задач;
* планирование процессов и диспетчеризация задач;
* синхронизация задач, обеспечение их средствами коммуникаций.

Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач.

Задача может породить новую задачу. При этом между процессами появляются «родственные» отношения. Порождающая задача называется «предком», «родителем», а порожденная — «потомком», «сыном» или «дочерней задачей».

«Предок» может приостановить или удалить свою дочернюю задачу, тогда как «потомок» не может управлять «предком».

Основная цель планирования вычислительного процесса заключается в распределении времени процессора или нескольких процессоров между выполняющимися заданиями пользователей таким образом, чтобы удовлетворить требованиям, предъявляемым пользователями к вычислительной системе

|  |  |
| --- | --- |
| Вид планирования | Выполняемые функции |
| Долгосрочное | Решение о добавлении задания (процесса) в пул выполняемых в системе |
| Среднесрочное | Решение о добавлении процесса к числу процессов, полностью или частично размещенных в основной памяти |
| Краткосрочное | Решение о том, какой из доступных процессов (потоков) будет выполняться процессором |
| Планирование ввода-вывода | Решение о том, какой из запросов процессов (потоков) на операцию ввода-вывода будет выполняться свободным устройством ввода- вывода |

Выделяют два типа планирования: статическое и динамическое планирование.

При динамическом планировании решения принимаются во время работы системы на основе анализа текущей ситуации, не используя никаких предложений о мультипрограммной смеси.

Организация работ по выполнению задач динамического планирования получила название диспетчеризации.

Другой тип планирования статический может быть использован только в специализированных системах с заданным набором задач, например, в управляющих вычислительных системах или системах реального времени. В этом случае статический планировщик принимает решение не во время работы системы, а заранее.

Основное отличие между долгосрочным и краткосрочным планировщиками заключается в частоте запуска: краткосрочный планировщик, например, может запускаться каждые 30 или 100 мс; долгосрочный — один раз за несколько минут или чаще.

С помощью долгосрочного планировщика определяется, какой из процессов, находящихся во входной очереди, должен быть переведен в очередь готовых процессов в случае освобождения ресурсов памяти и выбираются процессы из входной очереди с целью создания неоднородной мультипрограммной смеси. Это означает, что в очереди готовых к выполнению процессов должны находиться (в разной пропорции) как процессы, ориентированные на ввод/вывод, так и процессы, ориентированные на преимущественную работу с центральным процессором.

Функция краткосрочного планировщика состоит в определении конкретных задач из задач находящихся в очереди готовых к выполнению, которые должны быть переданы на исполнение. В большинстве современных операционных систем долгосрочный планировщик отсутствует.

В основе определения дисциплины диспетчеризации лежит выбор стратегии планирования. Стратегия планирования определяет, какие именно процессы могут быть направлены на выполнение для достижения целей, поставленных перед данными процессами.

Среди стратегий можно выделить следующие:

* обеспечения соответствия порядка окончания вычислений (вычислительных процессов) последовательности, в которой они были приняты к исполнению;   
  оказание предпочтения более коротким процессам;
* предоставление всем пользователям (процессам пользователей) одинаковых услуг, в том числе и одинакового времени ожидания.

Долгосрочное планирование заключается в подборе таких вычислительных процессов, которые бы меньше всего конкурировали между собой за ресурсы вычислительной системы.

Когда говорят о стратегии планирования, всегда имеют в виду понятие процесса, а не понятие задачи, поскольку процесс может состоять из нескольких потоков (задач).

Когда говорят о диспетчеризации, то всегда в явном или неявном виде имеют в виду понятие задачи (потока). Если ОС не поддерживает механизм тредов, то можно заменять понятие задачи понятием процесса.

Приоритеты имеют следующие свойства:

* приоритет, присвоенный задаче, может являться величиной постоянной (статический приоритет);
* приоритет задачи может изменяться в процессе ее решения (динамический приоритет).

**Дисциплина диспетчеризации FCFS**

Самой простой в реализации является дисциплина FCFS (first come — first served), согласно которой задачи обслуживаются в порядке очереди, то есть в порядке их появления. Те задачи, которые были заблокированы в процессе работы (попали в какое-либо из состояний ожидания, например, из-за операций ввода/вывода), после перехода в состояние готовности ставятся в эту очередь перед теми задачами, которые еще не выполнялись.

Дисциплина обслуживания SJN (Shortest Job Next) предполагает, что следующим будет выполняться кратчайшее задание. Для ее реализации необходимо, чтобы для каждого задания была известна оценка в потребностях машинного времени.

Одним из наиболее известных был, в частности, язык управления заданиями JCL (Job Control Language). Пользователи вынуждены были указывать предполагаемое время выполнения задания.

Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. Задания, которые в процессе своего исполнения были временно заблокированы (например, ожидали завершения операций ввода/вывода), вновь попадают в конец очереди готовых к выполнению наравне с вновь поступающими. Это приводит к тому, что задания, которым требуется очень немного времени для своего завершения, вынуждены ожидать процессор наравне с длительными работами, что не всегда целесообразно.

Для устранения этого недостатка и была предложена дисциплина SRT (Shortest Remaining Time), в соответствии с которой следующим будет выполняться задание, требующее наименьшего времени для своего завершения.

**Дисциплина обслуживания RR**

Дисциплина обслуживания RR предполагает, что каждая задача получает процессорное время порциями (квантами времени). После окончания кванта времени q задача снимается с процессора, и процессорное время передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению

Приоритетные дисциплины диспетчеризации:

* приоритетной многозадачностью;
* адаптивная многозадачность;
* спорадической многозадачности.

**Вытесняющаяся и не вытесняющаяся многозадачность**

Диспетчеризация без перераспределения процессорного времени, то есть невытесняющая многозадачность (non-preemptive multitasking) — это такой способ диспетчеризации процессов, при котором активный процесс выполняется до тех пор, пока он сам, что называется «по собственной инициативе», не отдаст управление диспетчеру задач для выбора из очереди другого, готового к выполнению процесса или треда. Дисциплины обслуживания FCFS, SJN, SRT относятся к невытесняющим.

Диспетчеризация с перераспределением процессорного времени между задачами, то есть вытесняющая многозадачность (preemptive multitasking) — это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается диспетчером задач, а не самой активной задачей.

При вытесняющей многозадачности механизм диспетчеризации задач целиком сосредоточен в операционной системе, и программист может писать свое приложение, не заботясь о том, как оно будет выполняться параллельно с другими задачами.

Дисциплина RR и многие другие, построенные на ее основе, относятся к вытесняющим.

**Качество диспетчеризации**

Гарантировать обслуживание можно следующими тремя способами:

1. выделять минимальную долю процессорного времени некоторому классу процессов, если, по крайней мере, один из них готов к исполнению. Например, можно отводить 20 % от каждых 10 мс процессам реального времени, 40 % от каждых 2 с — интерактивным процессам и 10 % от каждых 5 мин — пакетным (фоновым) процессам;
2. выделять минимальную долю процессорного времени некоторому конкретному процессу, если он готов к выполнению;
3. выделять столько процессорного времени некоторому процессу, чтобы он мог выполнить свои вычисления к определенному сроку.

Для сравнения алгоритмов диспетчеризации обычно используются следующие показатели:

* загрузка (использование) центрального процессора. В большинстве персональных систем средняя загрузка процессора не превышает 2–3 %, доходя в моменты выполнения сложных вычислений и до 100 %. В реальных системах, где компьютеры выполняют очень много работы, например в серверах, загрузка процессора колеблется в пределах 15–40 % для легко загруженного процессора и до 90–100 % — для сильно загруженного процессора;
* пропускная способность (CPU throughput) процессора, измеряемая количеством процессов, выполняемых в единицу времени;
* время оборота (turnaround time). Для некоторых процессов важным критерием является полное время выполнения, то есть интервал от момента появления процесса во входной очереди до момента его завершения. Это время названо временем оборота. Оно включает время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания в очередях к оборудованию, время выполнения в процессоре и время ввода/вывода;
* время ожидания (waiting time) — суммарное время нахождения процесса в очереди готовых процессов;
* время отклика (response time). Для интерактивных программ важным показателем является время отклика или время, прошедшее от момента попадания процесса во входную очередь до момента первого обращения к терминалу.

**Лекция 4. Введение в системы реального времени**

**Параллельные системы**

К параллельным системам относятся:

* Системы реального времени (СРВ) и встроенные системы (специального назначения).
* Обычные и распределенные операционные системы (их компоненты распределены по нескольким компьютерам).
* Системы управления базами данных и системы обработки транзакций.
* Распределенные сервисы прикладного уровня.

**Определения систем реального времени**

Система называется системой реального времени, если правильность ее функционирования зависит не только от логической корректности вычислений, но и от времени, за которое эти вычисления производятся.

Реальное время (программное обеспечение): относится к системе или режиму работы, в котором вычисления проводятся в течение времени, определяемого внешним процессом, с целью управления или мониторинга внешнего процесса по результатам этих вычислений.

Системы реального времени – это системы, которые предсказуемо (в смысле времени реакции) реагируют на непредсказуемые (по времени появления) внешние события.

Два типа систем реального времени:

* Системы с жесткими временными характеристиками — системы жесткого реального времени.
* Системы с нежесткими временными характеристиками — системы мягкого реального времени.

Также СРВ можно разделить на системы специализированные и универсальные:

* Специализированной СРВ называется система, где конкретные временные требования определены. Такая система должна быть специально спроектирована для удовлетворения этих требований.
* Универсальная СРВ должна уметь выполнять произвольные (заранее не определенные) временные задачи без применения специальной техники. Разработка таких систем, безусловно, является самой сложной задачей, хотя обычно, требования, предъявляемые к таким системам, мягче, чем требования для специализированных систем.

Если СРВ строится как программный комплекс, то, в общем, виде она может быть представлена как комбинация трех компонентов:

* Прикладное программное обеспечение.
* Операционная система реального времени (ОСРВ).
* Аппаратное обеспечение.

Встраиваемые системы (Embedded system) можно определить, как программное и аппаратное обеспечение, составляющее компоненты другой, большей системы и работающее без вмешательства человека.

**Области применения и вычислительные платформы СРВ**

Основные области применения СРВ:

Военная и космическая области:

* бортовое и встраиваемое оборудование;
* радары, системы измерения и управления;
* цифровые видеосистемы, симуляторы;
* ракеты, системы определения местоположения и привязки к местности.

Промышленность:

* автоматические системы управления производством;
* автоматические системы управления технологическими процессами;

Автомобилестроение:

* симуляторы, системы управления мотором, автоматическое сцепление ...

Энергетика:

* сбор информации, управление данными и оборудованием ...

Телекоммуникации:

* коммуникационное оборудование, сетевые коммутаторы, телефонные станции...

Банковское оборудование:

* банкоматы...

Товары широкого применения:

* мобильные телефоны;
* цифровое телевидение: мультимедиа, видеосервисы, цифровые телевизионные декодеры ...
* компьютерное и офисное оборудование.

**Области применения и  
вычислительные платформы СРВ**

Вычислительные установки, на которых используются СРВ, можно разделить на следующие платформы:

* «Обычные» компьютеры;
* Промышленные компьютеры;
* Встраиваемые системы.

**Диспетчеризация потоков**

Требования к ОСРВ:

1. ОСРВ должна реализовывать возможность многозадачности, причем с поддержкой вытесняющей приоритетной методики диспетчеризации.
2. ОС должна иметь достаточно большое (определяется масштабом задачи) количество приоритетов. Рекомендуемым значением является 128 уровней.
3. Наличие в ОС механизмов синхронизации доступа к разделяемым ресурсам.

**Инверсия приоритетов**

Инверсия приоритетов возникает, когда два потока, высоко приоритетный (В) и низкоприоритетный (Н) разделяют некий общий ресурс (Р). Предположим, также что в системе присутствует третий поток, приоритет которого находится между приоритетами В и Н. Назовем его средним (С).

Если поток В переходит в состояние готовности, когда активен поток Н, и Н заблокировал ресурс Р, то поток В вытеснит поток Н, и Р останется заблокирован. Когда В понадобится ресурс Р, то он сам перейдет в заблокированное состояние. Если в состоянии готовности находится только поток Н, то ничего страшного не произойдет, Н освободит заблокированный ресурс и будет вытеснен потоком В.

Но если на момент блокирования потока В в состоянии готовности находится поток С, приоритет которого выше чем у Н, то активным станет именно он, а Н опять будет вытеснен, и получит управление только после того, как С закончит свою работу.

Подобная задержка вполне может привести к тому, что критическое время обслуживания потока В будет пропущено. Если В — это поток жесткого реального времени, то подобная ситуация недопустима.

**Наследование приоритетов**

Суть метода наследования приоритетов заключается в наследовании низкоприоритетным потоком, захватившим ресурс, приоритета от высокоприоритетного потока, которому этот ресурс нужен. В описанном примере это означает следующее. Если Н блокировал ресурс Р, который нужен В, то при блокировании В его приоритет присваивается потоку Н, и, таким образом, он не может быть вытеснен потоком, с меньшим чем у В приоритетом. После того, как поток Н разблокирует ресурс Р, его приоритет понижается до исходного значения и он вытесняется потоком В.

**Протокол предельного приоритета**

Протокол Предельного Приоритета (Priority Ceiling Protocol). Метод заключается в добавлении к стандартным свойствам объектов синхронизации параметра, определяемого максимальным приоритетом потока, которые к объекту обращаются. Если такой параметр установлен, то приоритет любого потока, обращающегося к данному объекту синхронизации, будет увеличен до указанного уровня, и, таким образом, не сможет быть вытеснен никаким потоком, который может нуждаться в заблокированном им ресурсе. После разблокирования ресурса, приоритет потока понижается до начального уровня. Таким образом, получается нечто вроде предварительного наследования приоритетов.

**Временные характеристики ОС**

Время реакции операционной системы при любых вариантах загрузки должно оставаться постоянным. Независимость поведения системы и ее времен реакций от количества текущих задач.

**Лекция 5. Введение в ОС QNX и микроядро QNX  
Общее представление ОС реального времени QNX**

QNX была разработана, что называется «с нуля», канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США

* Предсказуемость, означающая ее применимость к задачам жесткого реального времени.
* Масштабируемость и эффективность, достигаемые оптимальным использованием ресурсов и означающие ее применимость для встроенных (embedded) систем;
* Расширяемость и надежность одновременно, поскольку написанный вами драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы.
* Быстрый сетевой протокол FLEET, прозрачный для обмена сообщениями, автоматически обеспечивающий отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа.
* Компактная графическая подсистема Photon, построенная на тех же принципах модульности, что и сама ОС, позволяет получить полнофункциональный GUI (расширенный Motif), работающий вместе с POSIX- совместимой ОС всего в 4Мбайт памяти, начиная с i80386 процессора.

**Архитектура ядра системы QNX**

В системе QNX ядро является действительно ядром. Прежде всего, как и подобает ядру операционной системы реального времени, оно имеет небольшой размер — менее 8 Кбайт. На ядро системы QNX возложено выполнение двух основных функций:

* передача сообщений (ядро реализует передачу всех сообщений между всеми процессами во всей системе) - (IPC — Inter-Process Communication);
* планирование (планировщик является частью ядра и подключается каждый раз, когда процесс меняет свое состояние в результате появления сообщения или прерывания).

Ядро системы QNX поддерживает также еще две функции:

* сетевое взаимодействие нижнего уровня. Ядро передает все сообщения, предназначенные процессам на другом узле;
* первичную обработку прерываний. Все прерывания и сбои аппаратного обеспечения сначала обрабатываются в ядре, а затем передаются соответствующему драйверу или системному администратору.

Все функции, выполняемые операционной системой QNX, за исключением функций ядра, реализуются стандартными процессами. В типичной конфигурации системы QNX имеются следующие системные процессы:

* Администратор процессов (Proc);
* Администратор файловой системы (Fsys);
* Администратор устройств (Dev);
* Сетевой администратор (Net).

**Связь между процессами**

1. Связь между процессами посредством сообщений
2. Связь между процессами посредством proxy
3. Связь между процессами посредством сигналов

**Связь между процессами посредством сигналов**

Связь посредством сигналов представляет собой традиционную форму асинхронного взаимодействия, используемую в различных операционных системах.

В системе QNX поддерживается большой набор POSIX-совместимых сигналов, специальные QNX- сигналы, а так же исторически сложившиеся сигналы, используемые в некоторых версиях системы UNIX.

Сигнал выдается процессу при наступлении некоторого заранее определенного для данного сигнала события. Процесс может выдать сигнал самому себе.

Если вы хотите сгенерировать сигнал из интерпретатора Shell, используйте утилиты kill() или slay().

Если вы хотите сгенерировать сигнал из процесса, используйте утилиты kill() или raise().

В зависимости от того, каким образом был определен способ обработки сигнала, возможны три варианта его приема:

1. Если процессу не предписано выполнять каких-либо специальных действия по обработке сигнала, то по умолчанию поступление сигнала прекращает выполнение процесса;
2. Процесс может проигнорировать сигнал. В этом случае выдача сигнала не влияет на работу процесса (обратите внимание на то, что сигналы SIGCONT, SIGKILL и SIGSTOP не могут быть проигнорированы при обычных условиях);
3. Процесс может иметь обработчик сигнала, которому передается управление при поступлении сигнала. В этом случае говорят, что процесс может "ловить" сигнал. Фактически такой процесс выполняет обработку программного прерывания. Данные с сигналом не передаются.

Для задания способа обработки сигнала следует воспользоваться функцией ANSI C signal() или функцией POSIX sigaction().

Функция sigaction() предоставляет больше возможностей по управлению средой обработки сигнала.

Существует важная взаимосвязь между сигналами и сообщениями. Если при генерации сигнала ваш процесс окажется SEND-блокированным или RECEIVE-блокированным (причем имеется обработчик сигналов), то будут выполняться следующие действия:

* процесс разблокировывается;
* выполняется обработка сигнала;
* функции Send() или Receive() возвращают управление с кодом ошибки.

Если процесс был SEND-блокированным, то проблемы не возникает, так как получатель не получит сообщение. Но если процесс был REPLY-блокированным, то неизвестно, было, обработано отправленное сообщение или нет, а, следовательно, неизвестно, нужно ли еще раз выдавать Send().

**Планирование процессов**

Планировщик ядра запускается в следующих случаях:

* после разблокировки процесса;
* по истечении временного кванта для выполняющегося процесса;
* после выгрузки выполняющегося процесса.

В системе QNX каждому процессу присваивается приоритет.

Планировщик выбирает для выполнения процессы, находящиеся в состоянии ГОТОВ, в соответствии с их приоритетами. (Центральный процессор может использовать только процесс, находящийся в состоянии ГОТОВ.)

Для выполнения выбирается процесс, имеющий наивысший приоритет.

Процессам присваиваются приоритеты в диапазоне от 0 (низший) до 31 (высший).

По умолчанию процесс наследует приоритет от породившего его процесса; обычно он равен 10 для приложений, запускаемых из интерпретатора Shell.

Если вы хотите определить приоритет процесса, используйте функцию getprio().

Если вы хотите задать приоритет процессу, используйте функцию setprio().

Для удовлетворения потребностей разных приложений в системе QNX реализованы три метода планирования:

* планирование по принципу простой очереди (первым пришел —первым обслужен);
* круговой метод планирования;
* спорадическое (ранее адаптивное) планирование.

Если вы хотите определить метод планирования для процесса, используйте функцию getscheduler().

Если вы хотите установить метод планирования для процесса, используйте функцию setscheduler().

**Первичная обработка прерываний**

Как бы нам этого не хотелось, но компьютер не может иметь бесконечное быстродействие.

В системе реального времени крайне важно, использовать все циклы работы центрального процессора.

Также важно минимизировать интервал времени между возникновением внешнего события и фактическим началом выполнения программы, реализующей ответную реакцию на это событие.

Это время называется задержкой или временем ожидания (latency).

Задержка планирования — это время между завершением работы обработчика прерываний и началом выполнения первой команды управляющего процесса. Обычно это интервал времени, который требуется для сохранения контекста процесса, выполняющегося в данный момент времени, и восстановления контекста управляющего процесса. Несмотря на то, что это время больше задержки прерывания, оно также остается небольшим в системе QNX.

**Вложенные прерывания**

Поскольку архитектура микрокомпьютера позволяет присваивать аппаратным прерываниям приоритеты, то высокоуровневые прерывания могут вытеснять низкоуровневые. Этот механизм полностью поддерживается в системе QNX.

**Лекция 6. Управление ресурсами в OC QNX**

Управление ресурсами ЭВМ — одна из главных функций любой ОС.

Администратор ресурсов — это прикладная серверная программа, принимающая QNX- сообщения от других программ и, при необходимости, взаимодействующая с аппаратурой.

Работу администратора ресурсов легко представить в виде схемы:

1. Выполняется инициализация интерфейса сообщений, при этом создается канал, по которому клиенты могут посылать свои сообщения администратору ресурсов.
2. Регистрируется путевое имя (т. е. зона ответственности) в пространстве имен администратора процессов.
3. Запускается бесконечный цикл по приему сообщений от клиентов.
4. С помощью операторов switch/case выполняется переключение на нужный обработчик для каждого типа сообщения.

**Файловые системы в QNX**

Файловые системы, поддерживаемые в QNX, можно классифицировать следующим образом:

Образная файловая система (image filesystem) — простая файловая система «только для чтения», состоящая из модуля procnto и других файлов, включенных в загрузочный образ QNX.

RAM — плоская «файловая система», которую автоматически поддерживает администратор процессов. Файловая система RAM основана на использовании ОЗУ и позволяет выполнять операции чтения/записи из каталога /dev/shmem.

Блочные файловые системы — традиционные файловые системы, обеспечивающие поддержку блок- ориентированных устройств типа жестких дисков и дисководов CD- ROM. К ним относятся файловые системы QNX4, DOS, ext2 и CD-ROM.

* файловая система QNX4 (fs-gnx4.so) — высокопроизводительная файловая система, сохранившая формат и структуру дисков ОС QNX4, однако усовершенствованная для повышения надежности, производительности и совместимости со стандартом POSIX;
* файловая система DOS (fs-dos.so) обеспечивает прозрачный доступ к локальным разделам FAT (12, 16, 32), при этом файловая система конвертирует POSIX-примитивы работы с диском в соответствующие DOS-команды. Если эквивалентную операцию выполнить нельзя (например, создать символьную ссылку), то возвращается ошибка;
* файловая система CD-ROM
* (fs-cd.so) обеспечивает прозрачный доступ к файлам на компакт-дисках формата ISO 9660 и его расширений (Rock Ridge, Joliet, Kodak Photo CD и Audio);
* файловая система Ext2 (fs-ext2.so) обеспечивает прозрачный доступ из среды QNX к Linux-разделам жесткого диска как 0, так и 1 версии.

Flash — не блокориентированные файловые системы, разрабатываемые специально для устройств флэш-памяти.

Network — файловые системы, обеспечивающие доступ к файловым системам на других ЭВМ. К ним относятся файловые системы NFS и CIFS (SMB).

файловая система NFS (Network File System) обеспечивает клиентской рабочей станции прозрачный доступ через сеть к файлам независимо от операционных систем, используемых файл- серверами. NFS использует механизм удаленного вызова процедур (RPC) и работает поверх TCP/IP.

файловая система CIFS (Common Internet File System) обеспечивает клиентским станциям прозрачный доступ к сетям Windows, а также к UNIX-системам с запущенным сервером SMB. Работает поверх TCP/IP.

Virtual — особые файловые системы, обеспечивающие специфические функциональные возможности при работе с другими файловыми системами:

* пакетная файловая система, обеспечивающая привычное представление выделенных файлов и каталогов для клиента.
* Inflator, администратор ресурсов, зона ответственности которого устанавливается ближе к корню файловой системы, предназначенный для динамического разжимания файлов, сжатых утилитой deflate.

**Инсталляционные пакеты и их репозитарии**

Создавая ОС QNX Neutrino, разработчики думали и о том, каким образом обеспечить простой и эффективный механизм поставки и инсталляции программного обеспечения для QNX.

В настоящее время очень распространенным средством доставки ПО от разработчика к пользователю стал Интернет.

Поэтому было принято решение, что файлы, входящие в состав программного продукта для QNX, следует помещать в архив формата TGZ, т. е. все файлы компонуются в один утилитой tar, и этот файл сжимается утилитой gzip.

Такой архив и есть QNX-пакет — файл, имеющий расширение qpk.

**Символьные устройства ввода/вывода**

Символьными устройствами ввода/вывода называют такие устройства, которые передают или принимают последовательность байт один за другим, в отличие от блок-ориентированных устройств. Имена администраторов символьных устройств ввода/вывода имеют вид devc-\*.

Обычно в системе имеются следующие символьные устройства:

* консольные устройства (или текстовые консоли);
* последовательные порты;
* параллельные порты;
* псевдотериналы (ptys).

Режимы ввода при работе устройств:

* поточный или «сырого ввода» (raw) — наиболее производительный режим, однако io- char не выполняет никаких редактирований принимаемых данных;
* редактируемый (edited) — режим, при котором io-char может выполнять операции редактирования над каждым символом строки. После окончания редактирования строки она становится доступной для обработки прикладным процессом (обычно, после ввода символа возврата каретки CR).

Такой режим часто именуют каноническим.

**Сетевая подсистема QNX**

В QNX реализованы средства поддержки двух протоколов — TCP/IP, являющегося промышленным стандартом, и Qnet, «родного» протокола QNX, реализующего концепцию «прозрачной сети».

Компьютеры, объединенные в сеть Qnet, фактически представляют собой виртуальную многопроцессорную суперЭВМ.

|  |  |
| --- | --- |
| Fault-Tolerant Networking | QNX может одновременно использовать несколько физических сетей. При выходе из строя любой из них данные будут автоматически перенаправлены «на лету» через другую сеть. |
| Load-Balancing on the Fly | При наличии нескольких физических соединений QNX автоматически распараллеливает передачу пакетов по соответствующим сетям. |
| Efficient Performance | Специальные драйверы, разрабатываемые фирмой QSSL для широкого спектра оборудования, позволяют с максимальной эффективностью использовать сетевое оборудование. |
| Extensable Architecture | Любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов. |
| TransparentDistributed Processing | Благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложения для того, чтобы они могли взаимодействовать через сеть. |

**Технология Jump Gate**

Для обеспечения прозрачности в графической оболочке Photon существует механизм, получивший название Jump Gate Technology.

Технология Jamp Gate основана на использовании серверного процесса phrelay, который передает клиентским программам информацию о графическом изображении в Photon.

**Обзор ОСРВ  
Стандарты на ОСРВ**

**Стандарт SCEPTRE**

Стандарт SCEPTRE (Standardisation du Cœur des Exécutifs des Produits Temps Réel Européens) — европейский стандарт на основы систем реального времени (sceptre по-французски означает «скипетр») разрабатывался в 1980-90 годы. За время его создания появились новые концепции в ОСРВ, не все из которых успели найти отражение в стандарте. В стандарте объединены усилия инженеров и исследователей в разработке групп спецификаций для промышленных приложений, даны определения и описания набора методов и подходов, используемых в ОСРВ.

Стандарт SCEPTRE определяет семь основных целей, которые должны преследовать ОСРВ:

* Адекватность поставленной задаче.
* Безопасность (система должна быть максимально устойчивой к аппаратным и программным сбоям).
* Минимальная стоимость.
* Максимальная производительность.
* Переносимость (возможность реализовывать систему на другом типе процессора, адекватном поставленной задаче).
* Адаптивность (способность системы приспосабливаться к новому управляемому ею оборудованию и задачам).
* Модульность (система должна состоять из достаточно независимых компонент, из которых можно построить систему, достаточную для решения поставленной задачи на имеющемся оборудовании).

Весь сервис, предоставляемый операционной системой, разделен в стандарте на следующие группы:

* Коммуникации (межпроцессорное взаимодействие);
* Синхронизация процессов;
* Контроль и планирование задач;
* Управление памятью;
* Управление прерываниями и оборудованием ввода/вывода;
* Высокоуровневый интерфейс ввода/вывода и управление периферийными устройствами;
* Управление файлами;
* Управление транзакциями (сообщениями и передачами данных);
* Обработка ошибок и исключений;
* Управление временем.

**Стандарт POSIX 1003.1b**

Стандарт POSIX (Portable Operating System Interface) 1003.1b, ранее существовавший под рабочим именем 1003.4 и разработанный IEEE (Institute of Electrical and Electronics Engineers), определяет расширение стандарта POSIX 1001 на операционные системы класса Unix, который позволяет использовать последние в качестве ОСРВ. Большинство приложений Unix могут быть перенесены в такие системы, поскольку стандарт POSIX 1003.1b обеспечивает единый с системами Unix программный интерфейс (API).

Стандарт POSIX 1003 состоит из следующих частей:

* POSIX 1003.1 — определяет стандарт на основные компоненты операционной системы, API для процессов, файловой системы, устройств и т.д.
* POSIX 1003.1b — определяет стандарт на основные расширения «реального времени».
* POSIX 1003.1c — определяет стандарт на задачи (threads).
* POSIX 1003.1d — определяет стандарт на дополнительные расширения «реального времени». Такие как, например, поддержка обработчиков прерываний.
* POSIX 1003.2 — определяет стандарт на основные утилиты.

Стандарту POSIX 1003 с расширением 1003.1b удовлетворяют такие системы, как Lynx, VxWorks, QNX. Некоторые системы, например, CHORUS, обеспечивают поддержку стандарта 1003.1b при загрузке поставляемого программного обеспечения, т.е. имеют как бы два типа API (оригинальный собственный и стандартный).

**Категории ОСРВ**

Операционные системы реального времени по способу разработки программного обеспечения делят на следующие категории:

* Self-Hosted ОСРВ
* Host/Target ОСРВ

Self-Hosted ОСРВ — это системы, в которых пользователи могут разрабатывать приложения, работая в самой ОСРВ. Обычно это предполагает, что ОСРВ поддерживает файловую систему, средства ввода-вывода, пользовательский интерфейс, имеет компиляторы, отладчик, средства анализа программ, текстовые редакторы, работающие под управлением ОСРВ.

**Категории ОСРВ  
Self-Hosted ОСРВ**

Достоинством таких систем является более простой и наглядный механизм создания и запуска приложений, которые работают на той же машине, что и пользователь.

Недостатком является то, что промышленному компьютеру во время его реальной эксплуатации часто вообще не требуется пользовательский интерфейс и возможность запуска тяжеловесных программ вроде компилятора. Следовательно, большинство возможностей ОСРВ не используются и только зря занимают память и другие ресурсы компьютера.

Обычно self-hosted ОСРВ применяются на «обычных» компьютерах промышленного исполнения.

**Категории ОСРВ. Host/Target ОСРВ**

Host/Target ОСРВ — это системы, в которых операционная система и компьютер, на котором разрабатываются приложения (host), и операционная система и компьютер, на котором запускаются приложения (target), различны. Связь между компьютерами осуществляется с помощью последовательно соединения (COM-порта), Ethernet, общей шины VME или compact PCI. В качестве host системы обычно выступает компьютер под управлением Unix или Windows NT, в качестве target системы выступает промышленный или встраиваемый компьютер под управлением ОСРВ.

Достоинством таких систем является использование всех ресурсов «обычной» системы для создания приложений и уменьшение размеров ОСРВ за счет включения только нужных приложению компонент.

Недостатком является относительная сложность программных компонент: кросс-компилятора, удаленного загрузчика и отладчика и т.д.

В настоящее время, рост мощности промышленных компьютеров позволяет использовать self-hosted ОСРВ на большем числе вычислительных систем. Хотя, с другой стороны, увеличивается распространение встраиваемых систем (в разнообразном промышленном и бытовом оборудовании), расширяет сферу применения host/target систем, поскольку при больших объемах выпуска цена системы является определяющим фактором.

**Деление ОСРВ по происхождению**

В зависимости от происхождения ОСРВ разделяют на следующие группы:

* Обычные ОС.
* Собственно ОСРВ.
* Специализированные ОСРВ.

Обычные ОС, используемые в качестве ОСРВ. Часто к обычным ОС добавляют дополнительные модули, реализующие поддержку специфического оборудования, а также планирования задач и обработку прерываний в соответствие с требованиями к ОСРВ и сглаживающие невозможность прервать ядро системы. Все системы относятся к разряду self- hosted.

Собственно ОСРВ. Бывают как self- hosted, так и host/target, некоторые ОСРВ поддерживают обе модели.

Специализированные ОСРВ. Это ОСРВ разрабатываемые для конкретного микроконтроллера его производителем. Часто не являются полноценными ОС, а представляют единый модуль с приложением и обеспечивают только необходимый минимум функций. Все такие системы относятся к категории host/target.

Системы на основе Linux (свободно распространяемой версии Unix). Она получила значительное распространение на настольных компьютерах благодаря своей бесплатности и качества.

В настоящий момент времени система Linux Может работать помимо процессоров Intel 80x86 на процессорах Alpha, SPARC, PowerPC, ARM, Motorola 68xxx, MIPS. Открытость ее исходных текстов позволяет реализовывать на ее основе специализированные системы и обеспечивать поддержку нового оборудования.

**Примеры обычны ОС**

Приспособление системы Linux к требованиям реального времени происходит по следующим направлениям:

Поддержка стандартов POSIX, касающихся реального времени. Стандарт POSIX 1003.1c (работа с задачами) уже поддержан, Стандарт POSIX 1003.1b (работа расширения реального времени) поддержан лишь частично: реализованы механизмы управления памятью и механизмы планирования задач.

Поддержка специального оборудования, важнейшим из которых является шина VME. Уже существует поддержка моста VME-PCI. Также для системы реального времени важным является повышение разрешения таймера системы.

Реализация механизма preemption (приоритетное прерывание обслуживания) для ядра системы. Этот механизм является необходимым для того, чтобы систему можно было назвать системой реального времени, но он также является очень сложным для реализации. Linux надолго запрещает прерывания при входе в ядро системы и не является preemptive.

Существует несколько проектов реализации preemption для ядра Linux. По способу решения этой задачи их можно разделить на две группы:

1. Механизм preemption реализуется путем переписывания ядра системы. На этом пути можно достичь самых качественных результатов, но на данный момент времени значительных успехов в этом плане нет по следующим причинам:

* слишком большой объем работы, связанный с большим объемом ядра;
* слишком высокая скорость изменения ядра, причем изменения вносятся, не учитывая интересы реального времени.

1. Механизм preemption реализуется путем написания микроядра, отвечающего за диспетчеризацию прерываний и задач. Ядро Linux работает как задача с низким приоритетом. Само ядро лишь незначительно изменено, для предотвращения блокирования им аппаратных прерываний. Задачи в такой системе разделены на две группы:

* процессы, работающие под управлением только микроядра (не использующие функции ядра Linux);
* процессы, работающие под управлением Linux (обычные приложения), а также задачи, работающие под управлением микроядра, но использующие функции Linux. Это процессы неудовлетворяющие требованиям реального времени, поскольку могут быть блокированы ядром Linux.

Недостатком такого подхода является необходимость реализации микроядра, обеспечивающего функционирование процессов реального времени. Например, если процесс реального времени хочет работать с коммуникационным портом, то драйвер этого порта надо перенести из ядра Linux в микроядро. Наиболее законченной реализацией этого подхода является проект RT-Linux.

Система RT-Linux является свободно распространяемой, разрабатываемой энтузиастами в ряде университетов мира. Создана в New Mexico Institute of Mining and Technology (USA).

RT-Linux представляет собой простейшее микроядро, отвечающее за создание и планирование задач, обеспечение их взаимодействия и диспетчеризацию прерываний. Реализован простейший приоритетный механизм планирования и единственный механизм взаимодействия — очередь сообщений FIFO. Ядро Linux работает как самая низкоприоритетная задача. В само ядро внесены исправления — макроопределения, отвечающие за запрещения/разрешения прерываний, заменены на соответствующие функции микроядра. Задачи Linux не могут прервать ядро Linux, задачи же микроядра могут.

Такая структура накладывает некоторые ограничения на задачи реального времени. Они не могут легко использовать различные драйвера Linux, не имеют доступ к сети и т.д. Зато они могут обмениваться данными со стандартами задачами Linux.

Простые очереди FIFO реализованы для обмена данными между процессами реального времени и процессами Linux.

Типичное приложение состоит из двух частей - задачи реального времени, непосредственно работающей с аппаратурой и обыкновенно задачи Linux, которая выполняет остальные операции, такие как сохранение данных на диск, пересылка их по сети, работа с пользователем (GUI) и т.д.

Самый короткий период для периодически вызываемых задач реального времени в RT-Linux на Pentium 120 - менее 150 мкс. Задачи, вызываемые по прерыванию могут иметь намного меньший период.

Ядро реального времени не защищает от перегрузок. Если одна из задач реального времени полностью утилизирует процессор, ядро Linux, имея самый низкий приоритет, не получит управления и система повиснет.

Задачи реального времени запускаются в адресном пространстве ядра и с привилегиями ядра и могут быть реализованы, например, при помощи модулей Linux.

Минимальный размер системы для записи в ПЗУ (без X-Windows) — 2.7 Мб.

**Лекция 7. Программная модель**

Существует два способа представления сигналов для хранения и передачи и обработки информации:

* Аналоговый
* Цифровой

**Единицы информации**

* Бит - имеет значение 0 (низкий сигнал) или 1 (высокий сигнал).
* Байт – состоит из 8 бит (28=256 значений).
* Слово – 2 байта (16 бит).
* Двойное слово – 4 байта (32 бита).
* Параграф – 16 байт.
* 1 Кб – 1024 байта (210 значений).
* Страница – 4 Кб.
* 1 Мб – 1024 Кб (220 значений).
* 1 Гб – 1024 Мб (230 значений).
* 1 Тб – 1024 Гб (240 значений).

**Архитектуры микропроцессоров семейства Intel**

Архитектура ЭВМ – это абстрактное представление ЭВМ, которое отражает ее архитектуру, схемотехническую и логическую организацию. Понятие архитектуры ЭВМ является комплексным и включает в себя:

* структурную схему ЭВМ;
* средства и способы доступа к элементам структурной схемы ЭВМ;
* организацию и разрядность интерфейсов ЭВМ;
* набор и доступность регистров;
* организацию и способы адресации памяти;
* способы представления и форматы данных ЭВМ;
* набор машинных команд ЭВМ;
* форматы машинных команд;
* обработку нештатных ситуаций(прерываний).

Все современные ЭВМ обладают некоторыми общими и индивидуальными свойствами архитектуры. К числу общих архитектурных свойств и принципов можно отнести:

* Принцип хранимой программы. Согласно ему, код программы и ее данные находятся в одном адресном пространстве в оперативной памяти.
* Принцип микропрограммирования. В состав процессора входит блок микропрограммного управления. Этот блок для каждой машинной команды имеет набор действий-сигналов, которые нужно сгенерировать для физического выполнения требуемой машинной команды.
* Линейное пространство памяти – совокупность ячеек памяти, которым последовательно присваиваются номера (адреса) 0, 1, 2, ...
* Последовательное выполнение программ. Процессор выбирает из памяти команды строго последовательно. Для изменения прямолинейного ходя выполнения программ необходимо использовать специальные команды (условного и безусловного перехода).
* Процессор не различает команды и данные, поэтому важно в программе всегда четко разделять пространство данных и команд.
* Безразличие к целевому назначению данных. Машине все равно, какую логическую нагрузку несут обрабатываемые ею данные.

Перечень индивидуальных свойств и принципов микропроцессоров весьма велик, наиболее значимыми являются:

* Суперскалярная архитектура. Важным элементом архитектуры, появившимся в i486, стал конвейер – специальное устройство, при котором выполнение команд в микропроцессоре разбивается на несколько этапов. В i486 они следующие:

1. выборка команд из кэш-памяти или оперативной;
2. декодирование команды;
3. генерация адреса;
4. выполнение операции с помощью АЛУ (арифметико-логическое устройство);
5. запись результата.

* Преимуществом такого подхода является то, что очередная команда после ее выборки попадает в блок декодирования. Таким образом, блок выборки свободен и может выбрать следующую команду. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Микропроцессоры, имеющие один конвейер, называются скалярными, а два и более – суперскалярными.
* Раздельное кэширование кода и данных. Кэширование – это способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в так называемой «кэш-памяти первого уровня», находящейся внутри микропроцессора.
* Предсказание правильного адреса перехода. Под переходом понимают запланированное алгоритмом изменение последовательного характера выполнения программы. Типичная программа на каждые 6-8 команд содержит одну команду перехода. Последствия этого предсказать не трудно: при наличии конвейера через каждые 6-8 команд его нужно очищать и заполнять заново в соответствии с адресом перехода. Все преимущества конвейеризации теряются. Поэтом в архитектуру Pentium был введен блок предсказания переходов. Вероятность правильного предсказания перехода составляет около 80%.

Режимы работы процессора. Процессоры семейства Intel х86, начиная с 80286, имеют несколько режимов работы:

* Реальный режим работы. Предназначен для совместимости с младшими моделями процессоров.
* Защищенный режим. Основной режим работы процессоров. Именно в нем доступны все особенности новых моделей процессоров такие, как многозадачность, защита программ пользователей, возможность заботы с большим объемом памяти, виртуальная память и т.д.
* Режим системного управления (SMM). В этом режиме доступны дополнительные возможности процессора.
* Режим Virtual-86. Этот режим схож с реальным режимом, однако может быть включен только в защищенном режим. В этом режиме возможно выполнение нескольких приложений реального режима.

«Нереальный» режим – это неофициальный режим, который поддерживают все 32-битные микропроцессоры. Он поддерживает адресацию к 4 Гбайтам памяти. В этом режиме команды исполняются также как и в реальном режиме с использованием дополнительных сегментных регистров.

**Архитектуры микропроцессоров семейства Intel / AMD64**

В 64-разрядных микропроцессорах архитектуры AMD64 введен дополнительный режим Long Mode («расширенный режим»), включающий два подрежима:

* 64-разрядный режим (позволяет 64-разрядной ОС выполнять 64-разрядное ПО);
* режим совместимости (позволяет 64-разрядной ОС выполнять 32-разрядное ПО).

**Архитектуры микропроцессоров семейства Intel**

Адресация ячеек памяти в реальном режиме. В «наследство» от процессоров 8086/88 достался своеобразный способ задания адреса ячейки памяти в виде указателя «seg:offset», состоящего из двух слов: сегмента (seg — segment) и смещения (offset). Такая запись предполагает вычисление полного адреса по формуле:

Addr = 16 \* seg + offset.

Любая выполняющаяся программа получает в свое распоряжение определенный набор ресурсов микропроцессора. Эти ресурсы необходимы для выполнения и хранения в памяти команд программы, данных и информации о текущем состоянии программы и микропроцессора. Набор этих ресурсов представляет собой программную модель микропроцессора.

**Программную модель микропроцессора Intel составляют:**

* пространство адресуемой памяти (для Pentium III — до 2^36- 1 байт);
* набор регистров для хранения данных общего назначения. Регистры этой группы используются для хранения данных и адресов;
* набор сегментных регистров. Регистры этой группы используются для хранения адресов сегментов в памяти;
* набор регистров состояния и управления (это регистры, которые содержат информацию о состоянии микропроцессора, исполняемой программы и позволяют изменить это состояние;
* системные регистры — это регистры для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели микропроцессора;
* набор регистров устройства вычислений с плавающей точкой (сопроцессора). Регистры этой группы предназначены для написания программ, использующих тип данных с плавающей точкой;
* набор регистров целочисленного MMX-расширения, отображенных на регистры сопроцессора (впервые появились в архитектуре микропроцессора Pentium ММХ);
* набор регистров MMX-расширения с плавающей точкой (впервые появились в архитектуре микропроцессора Pentium III);
* программный стек. Это специальная информационная структура, работа с которой предусмотрена на уровне машинных команд.

**Назначение регистров общего назначения.**

* eax/ax/ah/al (Accumulator register) — аккумулятор. Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;
* ebx/bx/bh/bl (Base register) — базовый регистр. Применяется для хранения базового адреса некоторого объекта в памяти;
* ecx/cx/ch/cl (Count register) — регистр-счетчик. Применяется в командах, про- изводящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла loop, кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра есх/сх;
* edx/dx/dh/dl (Data register) — регистр данных. Так же как и регистр eax/ax/ah/ al, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.
* esi/si (Source Index register) — индекс источника. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;
* edi/di (Destination Index register) — индекс приемника (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.
* esp/sp (Stack Pointer register) — регистр указателя стека. Содержит указатель вершины стека в текущем сегменте стека;
* ebp/bp (Base Pointer register) — регистр указателя базы кадра стека. Предназначен для организации произвольного доступа к данным внутри стека.

**Назначение сегментных регистров.**

* Сегмент кода. Содержит команды программы. Для доступа к этому сегменту служит регистр cs (code segment register) — сегментный регистр кода;
* Сегмент данных. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр ds (data segment register) — сегментный регистр данных;
* Сегмент стека. Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по следующему принципу: последний записанный в эту область элемент выбирается первым. Для доступа к этому сегменту служит регистр ss (stack segment register) — сегментный регистр стека, содержащий адрес сегмента стека;
* Дополнительный сегмент данных. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

**Назначение регистров состояния и управления.**

* eflags/flags (flag register) — регистр флагов. Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру flags для i8086.
* eip/ip (Instruction Pointer register) — указатель команд. Регистр eip/ip имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра eip/ip.

**Состав регистра флагов по группам:**

* 8 флагов состояния. Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра eflags отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм;
* 1 флаг управления. Обозначается как df (Directory Flag). Он находится в десятом бите регистра eflags и используется цепочечными командами. Значение флага df определяет направление поэлементной обработки в этих операциях: от начала строки к концу (df = 0) либо, наоборот, от конца строки к ее началу (df = 1). Для работы с флагом df существуют специальные команды сld (снять флаг df) и std (установить флаг df). Применение этих команд позволяет привести флаг df в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;
* 5 системных флагов, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.
* Системные регистры микропроцессора.
* Регистры управления (4 шт.);
* Регистры системны адресов (4 шт.); ν Регистры отладки (8 шт.).

**Регистры управления**

В группу регистров управления входят 4 регистра: cr0, cr1, cr2, cr3 (Эти регистры предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0).

* Регистр cr0 содержит системные флаги, управляющие режимами работы микропроцессора и отражающие его состояние глобально, независимо от конкретных выполняющихся задач;
* Регистр cr1 не доступен;
* Регистр cr2 используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти. В такой ситуации в микропроцессоре возникает исключительная ситуация с номером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр cr2. Имея эту информацию, обработчик исключения 14 определяет нужную страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы;
* Регистр cr3 также используется при страничной организации памяти. Это так называемый регистр каталога страниц первого уровня. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайт.

**Назначение системных флагов регистра cr0:**

* pe (Protect Enable), бит 0 — разрешение защищенного режима работы. Состояние этого флага показывает, в каком из двух режимов — реальном (pe=0) или защищенном (pe=1) — работает микропроцессор в данный момент времени.
* mp (Math Present), бит 1 — наличие сопроцессора. Всегда 1.
* ts (Task Switched), бит 3 — переключение задач. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи.
* am (Aligment Mask), бит 18 — маска выравнивания. Этот бит разрешает (am=1) или запрещает (am=0) контроль выравнивания.
* cd (Cache Disable), бит 30, — запрещение кэш-памяти. С помощью этого бита можно запретить (cd=1) или разрешить (cd=0) использование внутренней кэш-памяти (кэш- памяти первого уровня).
* pg (PaGing), бит 31, — разрешение (pg=1) или запрещение (pg=0) страничного преобразования. Флаг используется при страничной модели организации памяти.

Регистры системных адресов. Эти регистры еще называют регистрами управления памятью. Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора.

* регистр таблицы глобальных дескрипторов gdtr (Global Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый базовый адрес глобальной дескрипторной таблицы GDT и 16-битовое значение предела, представляющее собой размер в байтах таблицы GDT;
* регистр таблицы локальных дескрипторов ldtr (Local Descriptor Table Register) имеющего размер 16 бит и содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;
* регистр таблицы дескрипторов прерываний idtr (Interrupt Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый базовый адрес дескрипторной таблицы прерываний IDT и 16-битовое значение предела, представляющее собой размер в байтах таблицы IDT;
* 16-битовый регистр задачи tr (Task Register), который подобно регистру ldtr, содержит селектор, то есть указатель на дескриптор в таблице GDT. Этот дескриптор описывает текущий сегмент состояния задачи (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.
* Регистры отладки. Это группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486.
* Регистры dr0, dr1, dr2, dr3 имеют разрядность 32 бит и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах dr0...dr3, и при совпадении генерируется исключение отладки с номером 1.
* Регистры dr4, dr5 не используются.
* Регистр dr6 называется регистром состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1.
* Регистр dr7 называется регистром управления отладкой. В нем для каждого из четырех регистров контрольных точек отладки имеются поля, с помощью которых можно уточнить следующие условия, при которых следует сгенерировать прерывание.

**Структура программы на ассемблере**

Предложения ассемблера бывают четырех типов:

* команды или инструкции, представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд микропроцессора;
* макрокоманды–оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями;
* директивы, являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении;
* строки комментариев, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

**Формат предложений ассемблера:**

* Оператор директивы[;текст комментария]
* Оператор команды [;текст комментария]
* Оператор макрокоманды[;текст комментария]

**Формат директив:**

[Имя] директива [операнд1...операндN][;комментарий]

**Формат команд и макрокоманд**

[Имя метки:] КОП [операнд1...операндN][;комментарий]

Микропроцессор имеет шесть сегментных регистров, посредством которых может одновременно работать:

* с одним сегментом кода;
* с одним сегментом стека;
* с одним сегментом данных;
* с тремя дополнительными сегментами данных.

**Описание сегмента на ассемблере:**

Имя сегмента SEGMENT [тип выравнивания] [тип комбинирования] [класс сегмента] [тип размера сегмента] ...  
содержимое сегмента  
...  
Имя сегмента ENDS

Атрибут выравнивания сегмента (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в процессорах i80х86 выполняется быстрее. Допустимые значения этого атрибута следующие:

* BYTE – выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;
* WORD – сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);
* DWORD – сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита, равны 0 (выравнивание на границу двойного слова);
* PARA – сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);
* PAGE – сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу 256-байтной страницы);
* MEMPAGE – сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей 4-Кбайтной страницы памяти).

Атрибут комбинирования сегментов (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. Значениями атрибута комбинирования сегмента могут быть:

* PRIVATE – сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
* PUBLIC – заставляет компоновщик соединить все сегменты с одинаковыми именами. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды и данные, будут вычисляться относительно начала этого нового сегмента;
* COMMON – располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
* AT xxxx – располагает сегмент по абсолютному адресу параграфа (параграф – объем памяти, кратный 16, поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0);
* STACK – определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра ss.

Атрибут класса сегмента (тип класса) – это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при собирании программы из сегментов нескольких модулей. Компоновщик объединяет в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента).

Атрибут размера сегмента. Для процессоров i80386 и выше сегменты могут быть 16 или 32-разрядными. Это влияет, прежде всего, на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:

* USE16 – это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;
* USE32 – сегмент будет 32-разрядным. При формировании физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Для простых программ, содержащих по одному сегменту для кода, данных и стека в трансляторы MASM и TASM ввели возможность использования упрощенных директив сегментации.

Для этого, совместно с упрощенными директивами сегментации, стали использовать директиву указания модели памяти MODEL, которая частично стала управлять размещением сегментов и выполнять функции директивы ASSUME.

Описанные стандартные и упрощенные директивы сегментации не исключают друг друга.

Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

Создание СОМ-программ. Отличие EXE-программ от COM-программ.

* Размер программы. EXE-программа может иметь любой размер, в то время как COM-файл ограничен размером одного сегмента и не превышает 64К. COM-файл всегда меньше, чем соответствующий EXE-файл; одна из причин этого - отсутствие в COM-файле 512-байтового начального блока EXE-файла.
* Сегмент стека. В EXE-программе определяется сегмент стека, в то время как COM-программа генерирует стек автоматически. Таким образом, при создании ассемблерной программы, которая будет преобразована в COM-файл, стек должен быть опущен.
* Сегмент данных. В EXE программе обычно определяется сегмент данных, а регистр DS инициализируется адресом этого сегмента. В COM-программе все данные должны быть определены в сегменте кода. Ниже будет показан простой способ решения этого вопроса.
* Инициализация. EXE-программа записывает нулевое слово в стек и инициализирует регистр DS. Так как COM-программа не имеет ни стека, ни сегмента данных, то эти шаги отсутствуют. • Когда COM-программа начинает работать, все сегментные регистры содержат адрес префикса программного сегмента (PSP), – 256-байтового блока, который резервируется операционной системой DOS непосредственно перед COM или EXE программой в памяти. Так как адресация начинается с шест. смещения 100 от начала PSP, то в программе после оператора SEGMENT кодируется директива ORG 100H.
* Обработка. Для программ в EXE и COM форматах выполняется ассемблирование для получения OBJ-файла, и компоновка для получения EXE-файла. Если программа создается для выполнения как EXE-файл, то ее уже можно выполнить. Если же программа создается для выполнения как COM-файл, то компоновщиком будет выдано сообщение:  
  Warning: No STACK Segment (Предупреждение: Сегмент стека не определен)

При программировании на языке ассемблера используются данные следующих типов:

* Непосредственные данные, представляющие собой числовые или символьные значения, являющиеся частью команды. Непосредственные данные формируются программистом в процессе написания программы для конкретной команды ассемблера.
* Данные простого типа, описываемые с помощью ограниченного набора директив резервирования памяти, позволяющих выполнить самые элементарные операции по размещению и инициализации числовой и символьной информации.

Эти два типа данных являются элементарными, или базовыми; работа с ними поддерживается на уровне системы команд микропроцессора. Используя данные этих типов, можно формализовать и запрограммировать практически любую задачу.

* Данные сложного типа, которые были введены в язык ассемблера с целью облегчения разработки программ. Сложные типы данных строятся на основе базовых типов. Введение сложных типов данных позволяет несколько сгладить различия между языками высокого уровня и ассемблером. У программиста появляется возможность сочетания преимуществ языка ассемблера и языков высокого уровня (в направлении абстракции данных), что в конечном итоге **повышает эффективность конечной программы.**

**Простые типы данных:**

db – резервирование памяти для данных размером 1 байт dw – резервирование памяти для данных размером 2 байт dd – резервирование памяти для данных размером 4 байт df – резервирование памяти для данных размером 6 байт dp – резервирование памяти для данных размером 6 байт dq – резервирование памяти для данных размером 8 байт dt – резервирование памяти для данных размером 10 байт

TASM поддерживает следующие сложные типы данных:

* массивы;
* структуры;
* объединения;
* записи.

**Массивы. Описание и инициализация массива в программе.**

При необходимости использовать массив в программе его нужно моделировать одним из следующих способов:

1. mas dd 1,2,3,4,5 ; массив из 5 элементов. Размер каждого элемента 4 байта

2. mas dw 5 dup (0) ; массив из 5 нулевых элементов. Размер каждого элемента 2 байта

3. Используя директивы label и rept. mas\_b label byte  
mas\_w label word  
rept 4

dw 0f1f0h endm

**Массивы. Доступ к элементам массива.**

база + (индекс \* размер элемента)

**Структуры**

Для использования структур в программе необходимо выполнить три действия:

* Задать шаблон структуры. По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
* Определить экземпляр структуры. Этот этап подразумевает инициализацию конкретной переменной заранее определенной (с помощью шаблона) структурой.
* Организовать обращение к элементам структуры.

**Структуры. Описание шаблона структуры:**

имя\_структуры STRUC <описание полей>

имя\_структуры ENDS

Здесь <описание полей> представляет собой последовательность директив описания данных db, dw, dd, dq и dt.

**Объединения.**

Объединение — тип данных, позволяющий трактовать одну и ту же область памяти как имеющую разные типы и имена.

Описание объединений в программе напоминает описание структур, то есть сначала описывается шаблон, в котором с помощью директив описания данных перечисляются имена и типы полей:

имя\_объединения UNION <описание полей>

имя\_объединения ENDS

Отличие объединений от структур состоит, в частности, в том, что при определении переменной типа объединения память выделяется в соответствии с размером максимального элемента.

Обращение к элементам объединения происходит по их именам, но при этом нужно, конечно, помнить о том, что все поля в объединении накладываются друг на друга.

Одновременная работа с элементами объединения исключена. В качестве элементов объединения можно использовать и структуры.

**Записи**

Запись — структурный тип данных, состоящий из фиксированного числа элементов длиной от одного до нескольких бит.

Описание шаблона записи:  
имя\_записи RECORD <описание элементов>

При описании шаблона память не выделяется, так как это всего лишь информация для транслятора ассемблера о структуре записи.

**Записи. Работа с записями**.

Для установки значения некоторого поля записи используется команда setfield Использование различных способов адресации.

**Регистровая адресация**

Способ применим ко всем программно-адресуемым регистрам процессора. Операнд при этом находится в регистре.

push DS ;сохранить содержимое DS в стеке.

**Использование различных способов адресации**. **Непосредственная адресация.**

Операнд может быть числом, кодом ASCII, адресом, иметь символьное обозначение.

mov AX, 0101h ;16-ричное число помещается в регистр AX; mov DL, ‘1’ ;код ASCII символа ‘1’ загружается в DL.

**Использование различных способов адресации. Прямая адресация.**

В команде указывается либо символическое обозначение ячейки памяти, над содержимым которой должна выполниться операция, либо адрес непосредственно можно указать в качестве операнда. Во втором случае нужно предварительно настроить какой-либо сегментный регистр на начало нужного участка памяти, дальше через символ «:» в квадратных скобках указывается смещение в регистре.

mov DL, memx ;байт памяти с названием memx засылается в регистр DL; mov DL,ES:[2] ;в регистр DL засылается байт памяти с абсолютным адресом, начало сегмента памяти находится в ES, смещение в сегменте – в квадратных скобках.

**Использование различных способов адресации. Базовая адресация.**

В команде указывается относительный адрес ячейки памяти, над содержимым которой должна выполниться операция, Этот адрес находится в регистре, обозначение которого заключается в квадратные скобки, или вычисляется сложением содержимого двух регистров. Используются регистры:

BX (базовый регистр), в этом случае адрес сегмента данных, если не задан явно, предположительно хранится в DS (регистр сегмента данных);  
BP (регистр – указатель базы), в этом случае в качестве сегментного регистра используется регистр SS (регистр сегмента стека).

* mov DL, ES:[BX] ;данные из ячейки памяти, сегмент которой задан в ES, а смещение – в BX, помещаются в регистр DL;
* mov DL, [BX] ;команда аналогична предыдущей, за исключением того, что сегментный адрес ячейки памяти находится в DS.

**Использование различных способов адресации. Индексная адресация.**

Аналогична базовой.  
Используются регистры:

SI (регистр – индекс источника) и DI (регистр – индекс приемника). В этом случае адрес сегмента данных, если не задан явно, хранится в DS (регистр сегмента данных).  
mov AL, [DI] ;сегментный адрес ячейки памяти находится в DS.

**Использование различных способов адресации. Базовая адресация памяти со смещением**

Относительный адрес операнда определяется суммой содержимого регистра (BX, BP) и указанного в команде числа – смещения. При использовании регистра BP в качестве сегментного регистра по умолчанию подразумевается регистр SS. При использовании BX – регистр DS.

mas db 1,2,3,4,5  
mov BX, offset mas  
mov DL, 2[BX]  
массива вычисляется как сумма содержимого BX и смещения, заданного константой 2 или  
mov DL,[BX+2]  
или  
mov DL,[BX]+2

**Индексная адресация памяти со смещением.**

Аналогична базовой адресации со смещением.  
Используются регистры:

SI (регистр – индекс источника) и DI (регистр – индекс приемника). В этом случае адрес сегмента данных, если не задан явно, хранится в DS (регистр сегмента данных).  
mov DL, 2[SI] ;Относительный адрес нужного элемента памяти вычисляется как сумма содержимого SI и смещения, заданного константой 2 или  
mov DL,[SI+2] или mov DL,[SI]+2

**Использование различных способов адресации. Базово-индексная адресация памяти.**

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров. Допускается использование следующих пар: [BX][SI], [BX][DI], [BP][SI], [BP][DI].

Если в качестве базового регистра используется BX, то в качестве сегментного подразумевается DS. Если в качестве базового регистра – BP, то в качестве сегментного – SS. При необходимости сегментный регистр можно указать явно.

mov BX, [BP][SI]; в BX засылается слово из стека (сегментный адрес в SS), смещение вычисляется как сумма содержимого BP и SI.

**Использование различных способов адресации. Базово-индексная адресация со смещением.**

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров и дополнительного смещения. При адресации используются те же пары регистров, что и при базово- индексной адресации. Также если в качестве базового регистра используется BX , то в качестве сегментного подразумевается DS. Если в качестве базового регистра – BP, то в качестве сегментного – SS. При необходимости сегментный регистр можно указать явно.

mov mas[BX][SI],10 ;Число 10 засылается в ячейку памяти, сегментный адрес которой хранится в DS, а смещение равно сумме содержимого BX и SI и смещения ячейки mas.