УТВЕРЖДЕН

КПДА.00002-01 32-ЛУ

ЗАЩИЩЕННАЯ ОПЕРАЦИОННАЯ СИСТЕМА РЕАЛЬНОГО ВРЕМЕНИ КПДА.00002-01

Руководство системного программиста

КПДА.00002-01 32

Листов 181

2022

АННОТАЦИЯ

Данный документ представляет собой руководство системного программиста программного комплекса «Защищенная операционная система реального времени КПДА.00002-01» и адресован как программистам (системным и прикладным), так и системным администраторам.

В документе рассматриваются базовые вопросы применения операционной системы. Вопросы, связанные с особенностями защищенной версии и защитой информации от несанкционированного доступа рассматриваются в документе «Защищенная операционная система реального времени КПДА.00002-01. Руководство администратора безопасности» (КПДА.00002-01 92).

Документ включает следующие разделы: «Общие сведения об изделии», «Построение целевых систем» и «Проверка и восстановление целевых систем».

В разделе «Общие сведения об изделии» приведены назначение и функции изделия, а также сведения о технических средствах, обеспечивающих его функционирование.

В разделе «Построение целевых систем» приведено описание действий по настройке программы на условия конкретного применения (настройка на состав технических средств, выбор функций и др.).

В разделе «Проверка и восстановление целевых систем» приведено описание способов проверки, позволяющих дать общее заключение о работоспособности программы, и приведены методы, позволяющие восстанавливать поврежденную систему.

СОДЕРЖАНИЕ

1. ОБЩИЕ СВЕДЕНИЯ ОБ ИЗДЕЛИИ 5
1.1. Архитектура и функции микроядра 5
1.1.1. Средства организации взаимодействия между процессами 6
1.1.2. IPC посредством сообщений 6
1.1.3. IPC посредством прокси 16
1.1.4. IPC посредством сигналов 18
1.1.5. Организация взаимодействия между процессами в сети 21
1.1.6. Синхронизация процессов с использованием семафоров 25
1.1.7. Механизмы и методы диспетчеризации процессов 26
1.1.8. Механизмы работы в реальном времени 32
1.2. Администратор процессов 37
1.2.1. Общие сведения 37
1.2.2. Описание жизненного цикла процесса 39
1.2.3. Состояния процесса 40
1.2.4. Символьные имена процессов 43
1.2.5. Механизмы использования Таймеров 44
1.2.6. Обработчики прерываний 45
1.3. Администратор устройств 47
1.3.1. Назначение Администратора устройств 47
1.3.2. Механизмы обслуживания устройств 47
1.3.3. Режим редактируемого ввода 48
1.3.4. Режим необрабатываемого ввода 50
1.3.5. Использование драйверов устройств 51
1.3.6. Управление устройствами 53
1.3.7. Управление консолью 53
1.3.8. Управление последовательными устройствами 54
1.3.9. Управление параллельными устройствами 55
1.3.10. Обеспечение производительности подсистемы устройств 55
1.4. Администратор сети 57
1.4.1. Общие сведения 57
1.4.2. Назначение Администратора сети 57
1.4.3. Интерфейс между системными модулями 58
1.4.4. Сетевые драйверы 61
1.4.5. Идентификаторы узла и сети 61
1.4.6. Порядок выбора сети 63
1.5. Администратор файловой системы 66
1.5.1. Общие сведения 66
1.5.2. Регулярные файлы и каталоги 67
1.5.3. Ссылки и индексные дескрипторы (inodes) 70
1.5.4. Символические ссылки 72
1.5.5. Программные каналы (pipes) и FIFO 73
1.5.6. Файлы FIFO 74
1.5.7. Обеспечение производительности файловой системы 74
1.5.8. Обеспечение надежности файловой системы 76
1.5.9. Работа с дисками 78
1.5.10. Ключевые компоненты раздела КПДА.00002-01 82
1.5.11. Администратор файловой системы DOS 83
1.5.12. Файловая система CD-ROM 84
1.5.13. Файловая система флэш 85
1.6. Пространство имен ввода/вывода 87

1.6.1. Общие сведения ... 87
1.6.2. Разборка имен путей ... 87
1.6.3. Пространство дескрипторов файлов ... 93

2. ПОСТРОЕНИЕ ЦЕЛЕВЫХ СИСТЕМ ... 97

2.1. Установка инструментальной системы ... 97
2.2. Построение образа ОС ... 98
2.2.1. Общие замечания ... 98
2.2.2. Порядок создания файла построения ... 98
2.2.3. Формат файла построения ... 98
2.2.4. Хранение исполняемых файлов ... 100
2.2.5. Выбор компонентов для образа ... 100
2.3. Файловая система среды исполнения ... 105
2.3.1. Общие рекомендации ... 105
2.3.2. Разделение пространства имен путей ... 105
2.3.3. Установка файловой системы DOS ... 110
2.4. Подключение символьных устройств ... 119
2.5. Порядок создания сети ... 120
2.5.1. Общие замечания ... 120
2.5.2. Планирование сети ... 122
2.5.3. Выбор конфигурации сервера начальной загрузки ... 127
2.5.4. Загрузка узла из сети КПДА.00002-01 ... 135
2.5.5. Загрузка узла со своего собственного жесткого диска ... 136
2.5.6. Диагностика сети ... 137
2.6. Настройка учетных записей пользователей ... 138
2.6.1. Запуск сеанса пользователя ... 138
2.6.2. Управление доступом пользователя к ресурсам ... 140
2.6.3. Файл регистрации ... 153
2.6.4. Дополнительные файлы ... 155
2.7. Установка терминалов ... 156
2.7.1. Поиск характеристик терминалов ... 156
2.7.2 Установка типа терминала ... 156
2.7.3 База данных termcap ... 157
2.7.4 База данных terminfo ... 158

3. ПРОВЕРКА И ВОССТАНОВЛЕНИЕ ИЗДЕЛИЯ ... 160

3.1. Создание резервных копий ... 160
3.1.1. Общие замечания ... 160
3.1.2. Когда создавать резервные копии ... 160
3.1.3. Форматы резервных копий ... 160
3.1.4. Носители резервных копий ... 162
3.1.5. Сжатие данных ... 166
3.1.6. Примеры создания архивов ... 166
3.2. Восстановление дисков и файлов ... 169
3.2.1. Общие рекомендации ... 169
3.2.2. Краткий обзор структуры раздела КПДА.00002-01 ... 170
3.2.3. Утилиты сопровождения файловой системы ... 171
3.2.4. Процедуры восстановления диска ... 172
3.2.5. Порядок действий в случае, если система не загружается ... 175
3.2.6. Восстановление потерянных файлов и каталогов ... 180

1. ОБЩИЕ СВЕДЕНИЯ ОБ ИЗДЕЛИИ

Основу изделия составляет микроядро и четыре системных модуля:

1) Администратор процессов;

2) Администратор устройств;

3) Администратор сети;

4) Администратор файловой системы.

1.1. Архитектура и функции микроядра

Структура микроядра представлена на рисунке 1.

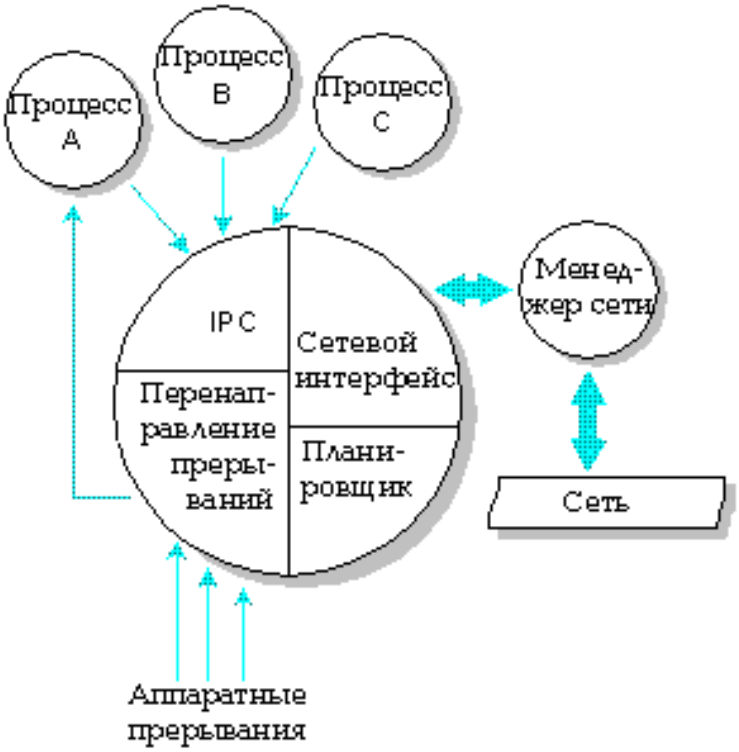


Рис. 1. Структура микроядра.

Микроядро изделия отвечает за выполнение следующих функций:

1) поддержка трех механизмов IPC (Inter Process Communication - межпроцессное взаимодействие): сообщения, прокси и сигналы;

2) обеспечение сетевого интерфейса низкого уровня. Микроядро осуществляет доставку всех сообщений, предназначенных для процессов, выполняющихся на других узлах сети;

3) диспетчеризация процессов. Входящий в состав микроядра планировщик решает, какому из готовых к выполнению процессов должно быть передано управление;

4) первичная обработка прерываний. Все аппаратные прерывания и исключения сначала обрабатываются микроядром, а затем передаются соответствующему драйверу или другому системному процессу.

1.1.1. Средства организации взаимодействия между процессами

Микроядро поддерживает три формы IPC: сообщения, прокси и сигналы.

Сообщения – это основополагающая форма IPC в изделии. Они обеспечивают синхронную связь между взаимодействующими процессами.

Прокси - это особый вид сообщения. Они больше всего подходят для извещения о наступлении какого-либо события, когда процессу, посылающему сообщение, не требуется вступать в диалог с получателем.

Сигналы - это традиционная форма IPC. Они используются для асинхронной связи между процессами.

1.1.2. IPC посредством сообщений

Сообщения – это пакеты байт, которые синхронно передаются от одного процесса к другому. При этом микроядро не анализирует содержание сообщения. Передаваемые данные понятны только отправителю и получателю и никому более.

Для непосредственной связи друг с другом взаимодействующие процессы используют следующие функции языка программирования Си:

1) Send( ) – посылка сообщений;

2) Receive() – получение сообщений;

3) Reply() – ответ процессу, пославшему сообщение.

Эти функции могут быть использованы как локально, т.е. для связи между процессами, выполняющимися на одном компьютере, так и в пределах сети, т.е. для связи между процессами, выполняющимися на разных узлах («Узел» - это отдельная ЭВМ, работающая под управлением изделия).

Следует заметить, однако, что не всегда возникает необходимость использовать функции Send(), Receive() и Reply() в явном виде. Библиотека функций языка Си в операционной системе КПДА.00002-01 построена на основе использования сообщений – в результате, когда процесс использует стандартные механизмы передачи данных (такие, как, например, программный канал – pipe), он косвенным образом использует передачу сообщений.

На рисунке 2 изображена последовательность событий, имеющих место, когда два процесса (процесс А и процесс В) используют функции Send(), Receive() и Reply() для связи друг с другом:

1. Процесс А посылает сообщение процессу В, вызвав функцию Send(), которая передает соответствующий запрос ядру. В этот момент времени процесс А переходит в SEND-блокированное состояние и остается в этом состоянии до тех пор, пока процесс В не вызовет функцию Receive() для получения сообщения. Процесс В вызывает Receive() и получает сообщение от процесса А. При этом состояние процесса А изменяется на REPLY-блокирован. Процесс В при вызове функции Receive() в данном случае не блокируется, т.к. к этому моменту его уже ожидало сообщение от процесса А.

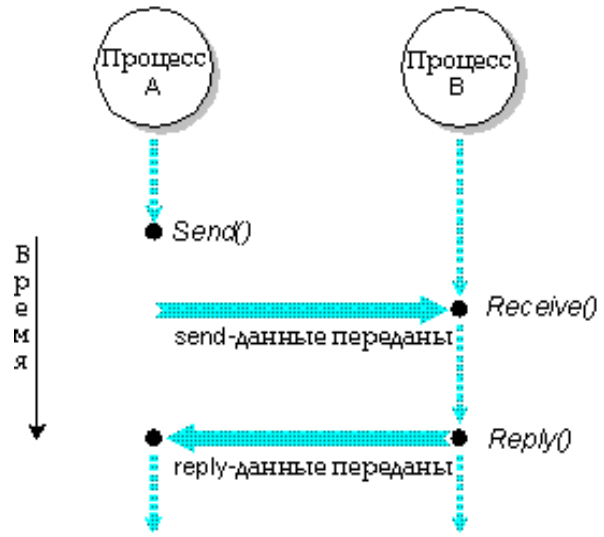


Рис. 2. Процесс А посылает сообщение процессу В, который получает его, обрабатывает и посылает ответ.

2. Если процесс B вызвал Receive() до того, как ему было послано сообщение, то он попадает в состояние RECEIVE-блокировки до получения сообщения. В этом случае процесс-отправитель сообщения немедленно после посылки сообщения попадает в состояние REPLY-блокирован.

3. Процесс B выполняет обработку полученного от процесса A сообщения и затем вызывает функцию Reply(). Ответное сообщение передается процессу A, который переходит в состояние готовности к выполнению. Вызов Reply() не блокирует процесс B, который также готов к выполнению. Какой из этих процессов будет выполняться, зависит от их приоритетов.

Использование сообщений обеспечивает выполнение следующих функций:

1) синхронизацию процессов;

2) блокирование.

Синхронизация процессов

Передача сообщений не только позволяет процессам обмениваться данными, но и предоставляет механизм синхронизации выполнения нескольких взаимодействующих процессов.

На рисунке 2 показано, что после того как процесс A вызвал функцию Send(), он не может продолжать выполнение до тех пор, пока не получит ответ на посланное сообщение. Это гарантирует, что выполняемая процессом B по запросу процесса A обработка данных будет завершена прежде, чем процесс A продолжит выполнение. Более того, после вызова процессом B запроса на получение данных Receive(), он не может продолжать выполнение до тех пор, пока не получит следующее сообщение.

Более подробно диспетчеризация процессов описана в разделе «Диспетчеризация процессов» далее в этой главе.

Блокирование

Если процессу не разрешается продолжать выполнение, т.к. он должен ожидать окончания определенной стадии протокола передачи сообщения, то процесс называется *блокированным*.

Возможные блокированные состояния процессов приведены в таблице 1.1. Граф переходов между состояниями процесса представлен на рис. 3.

Таблица 1.1. Состояния процессов при передаче сообщений

Условие блокирования	**Состояние процесса**
Процесс выполнил запрос Send(), и отправленное им сообщение еще не получено процессом-получателем	SEND-блокирован
Процесс выполнил запрос Send(), и отправленное им сообщение получено процессом-получателем, но ответ еще не выдан	REPLY-блокирован
Процесс выполнил запрос Receive(), но еще не получил сообщение	RECEIVE-блокирован

Замечание: Для получения информации о всех возможных состояниях процесса смотри соответствующий раздел руководства.

Использование функций Send(), Receive() и Reply()

Для подробного рассмотрения вызовов функций Send(), Receive() и Reply() использован пример, приведенный на рисунке 2 (передача сообщения от процесса A к процессу B).

Функция Send()

Предположим, что процесс A выдает запрос на передачу сообщения процессу B. Это выполняется посредством вызова функции Send():

```
Send ( pid, smsg, rmsg, smsg_len, rmsg_len );
```

При вызове функции Send() используются следующие аргументы:

1) pid – идентификатор процесса (*p*rocess *ID*), которому предназначается сообщение (т.е. процесса B); этот идентификатор используется для обращения к процессу со стороны операционной системы и других процессов;

2) smsg – буфер сообщения (т.е. сообщение, подлежащее посылке);

3) rmsg – буфер ответа (будет содержать ответ от процесса B)

4) smsg_len – длина посылаемого сообщения в байтах;

5) rmsg_len – максимальная длина ответа, который может принять процесс A, в байтах.

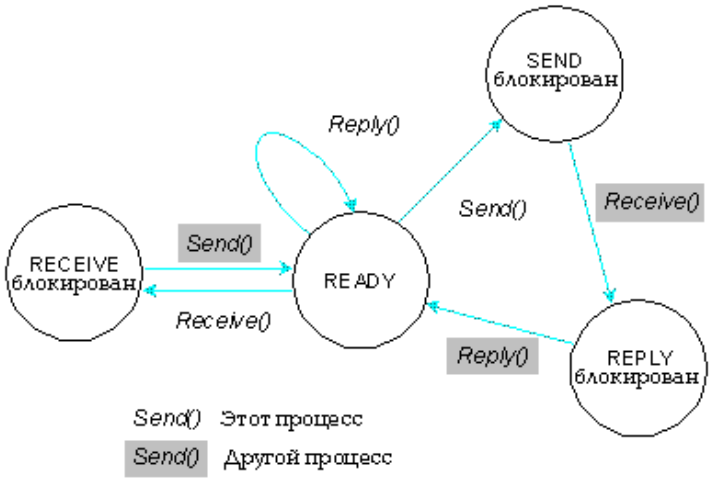


Рис. 3. Изменение состояния процессов в типичном случае передачи сообщения.

Обратите внимание, что будет передано не более smsg_len байт и не более чем rmsg_len байт будет получено в качестве ответа – это гарантирует, что не произойдет случайного переполнения буферов.

Функция Receive()

Вызвав запрос Receive(), процесс B может получить сообщение, направленное ему процессом A:

```
pid = Receive( 0, msg, msg_len )
```

Вызов функции Receive() содержит следующие аргументы:

1) pid – идентификатор процесса, который послал сообщение (т.е. процесс A);

2) 0 (ноль) – означает, что процесс B желает принять сообщение от любого процесса;

3) msg – буфер, куда будет помещено принимаемое сообщение;

4) msg_len – максимальный размер данных, которые будут помещены в буфер приема, в байтах.

Если значения аргументов smsg_len в вызове функции Send() и msg_len в вызове функции Receive() отличаются, друг от друга, то наименьшее из них определяет размер данных, которые будут переданы.

Функция Reply()

После успешного получения сообщения от процесса A, процесс B должен ответить процессу A, вызвав функцию Reply():

```
Reply( pid, reply, reply_len );
```

Вызов функции Reply() содержит следующие аргументы:

1) pid – идентификатор процесса, которому предназначается ответ (т.е. процесс A);

2) reply – буфер, содержащий ответ;

3) reply_len – длина данных, передаваемых в качестве ответа, в байтах.

Если значения аргументов reply_len при вызове функции Reply() и rmsg_len при вызове функции Send() отличаются друг от друга, то наименьшее из них определяет размер передаваемых данных.

Reply-управляемый обмен сообщениями

Рассмотренный пример обмена сообщениями иллюстрирует наиболее распространенный случай использования сообщений - случай, когда для процесса, выполняющего функции сервера, нормальным является состояние RECEIVE-блокирован в ожидании каких-либо запросов клиента. Это называется *send-управляемый обмен сообщениями*: процесс-клиент инициирует действие, посылая сообщения, ответ сервера на сообщение завершает действие.

Существует и другая модель обмена сообщениями, не столь распространенная, как рассмотренная выше, хотя в некоторых ситуациях она предпочтительнее: *reply-управляемый обмен* сообщениями, при котором действие инициируется вызовом функции Reply(). В этом случае процесс-"работник" посылает серверу сообщение, говорящее о том, что он готов к работе. Сервер не отвечает сразу, а "запоминает", что работник готов выполнить его задание. В последствии сервер может инициировать действие, ответив ожидающему задание работнику. Процесс-работник выполнит задание и завершит действие, послав серверу сообщение, содержащее результаты своей работы.

Особенности разработки программ, использующих передачу сообщений

При разработке программ, использующих передачу сообщений, необходимо иметь в виду следующее:

- сообщение сохраняется в теле посылающего процесса до тех пор, пока принимающий процесс не будет готов получить его. Сообщение не копируется в микроядро. Это безопасно, т.к. посылающий процесс SEND-блокирован и не может неумышленно изменить содержимое сообщения;

- при вызове функции Reply() данные копируются из отвечающего процесса в REPLY-блокированный процесс как одна неразрывная операция. Вызов функции Reply() не блокирует процесс - REPLY-блокированный процесс перестает быть блокированным после того, как в него скопированы данные;

- при посылке сообщения процессу не требуется знать состояние того процесса, которому это сообщение адресовано. Если получатель не готов к приему сообщения в момент его отправки, то посылающий сообщение процесс просто становится SEND-блокированным;

- если необходимо, процесс может посылать сообщение нулевой длины, ответ нулевой длины, либо и то и другое;

- с точки зрения разработчика, использование Send() для передачи процессу-серверу запроса на получение какой-либо услуги равнозначно вызову библиотечной подпрограммы, предоставляющей ту же самую услугу. В обоих случаях программа сначала подготавливает определенные данные, а затем вызывает либо функцию Send(), либо библиотечную подпрограмму, после чего ожидает, когда они закончат выполнение. В обоих случаях код, реализующий запрашиваемую услугу, выполняется между двумя четко определенными точками - Receive() и Reply() для процесса-сервера, входом в подпрограмму и командой return для вызова библиотечной подпрограммы. Когда вызываемая сервисная программа завершена, исполняемая программа "знает", куда помещены результаты, и может приступить к проверке возвращенного кода ошибки, обработке результатов и так далее.

Несмотря на кажущуюся простоту, вызов функции Send() делает больше, чем простой вызов библиотечной подпрограммы. Функция Send() может прозрачно для вызывающей программы передать запрос на другой узел сети, где и будет в действительности выполняться обслуживающий запрос код. При этом также может быть задействована параллельная обработка данных без издержек на создание нового процесса. Процесс-сервер может сразу, как только станет возможным, вызвать функцию Reply(), позволяя тем самым запрашивавшему процессу возобновить выполнение и, в то же время, продолжить выполнение самому.

Возможна ситуация, когда сообщения от нескольких процессов одновременно ожидают своей обработки одним и тем же принимающим процессом. Как правило, принимающий процесс получает сообщение в том же порядке, в каком они были посланы другими процессами; однако принимающий процесс может установить очередность получения сообщений на основе приоритетов посылающих процессов.

Существуют также следующие дополнительные возможности по передаче сообщений:

условный прием сообщения;

1) чтение или запись части сообщения;
2) составные сообщения (сообщения, состоящие из нескольких частей).

Условный прием сообщения (рис. 4)

Обычно, когда необходимо, чтобы процесс принял сообщение, используется функция Receive(). Это обычный способ получения сообщений, который пригоден в большинстве случаев.

Однако возможна ситуация, когда необходимо, чтобы процесс мог определить, имеются ли ожидающие приема сообщения, не попадая при этом в состояние RECEIVE-блокирован в случае их отсутствия. Например, процессу требуется опрашивать работающее с высокой скоростью устройство, которое не способно генерировать прерывание, и в то же время он должен отвечать на сообщения от других процессов. В этом случае процесс может использовать функцию Creceive(),

которая либо прочитает ожидающее приема сообщение, либо немедленно вернет управление процессу в случае отсутствия ожидающих приема сообщений

Замечание: Следует по возможности избегать использования функции Creceive(), т.к. она позволяет процессу непрерывно выполняться с неизменяющимся уровнем приоритета.

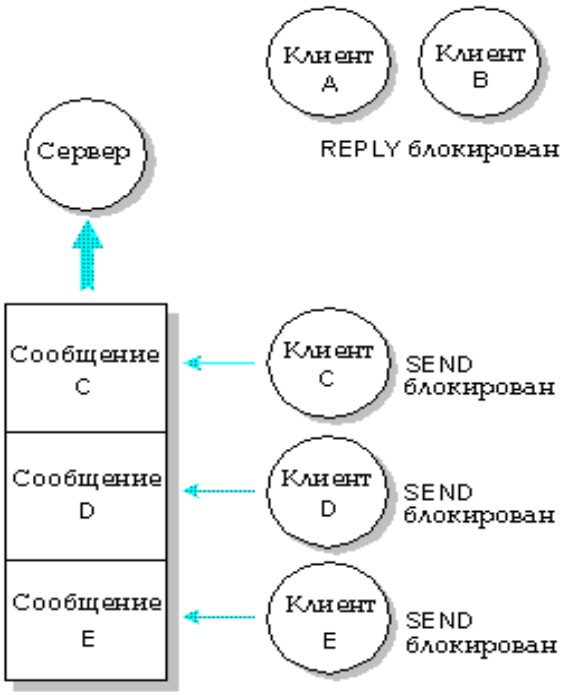


Рис. 4 Сервер получил сообщение от клиента A и клиента B (но еще не ответил им). Сообщения от клиентов C, D, E еще не получены.

Чтение или запись части сообщения

Иногда желательно читать или записывать сообщения по частям с тем, чтобы использовать уже выделенный для сообщения буфер вместо выделения отдельного рабочего буфера.

Например, администратор ввода/вывода может принимать данные в виде сообщений, которые состоят из заголовка фиксированной длины и следующих за ним данных переменной длины. В заголовке указывается размер данных (от 0 до 64 Кбайт). В этом случае администратор ввода/вывода может сначала принять только заголовок сообщения, а затем использовать функцию Readmsg() для чтения данных переменной длины непосредственно в соответствующий буфер вывода. Если размер данных превышает размер буфера, то администратор может неоднократно вызывать

функцию Readmsg() по мере освобождения буфера вывода. Аналогичным образом, функция Writemsg() может быть использована для поэтапного копирования данных в выделенный для ответного сообщения буфер непосредственно в теле процесса, пославшего сообщение, уменьшая, таким образом, потребность администратора ввода/вывода в выделении внутренних буферов.

Составные сообщения (Рис. 5)

До сих пор сообщения рассматривались как непрерывную последовательность байт. Однако сообщения часто состоят из двух или более отдельных частей. Например, сообщение может иметь заголовок фиксированной длины, за которым следуют данные переменной длины. Для того чтобы избежать копирования частей такого сообщения во временные промежуточные буферы при передаче или приеме, может быть использовано составное сообщение, состоящее из двух или более отдельных буферов сообщений. Благодаря этой возможности администраторы такие системные компоненты, обеспечивающие ввод/вывод, как Dev и Fsys, достигают своей высокой производительности.

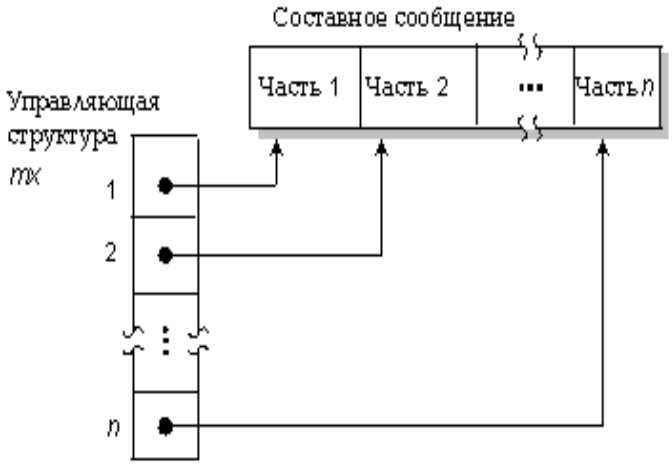


Рис. 5. Составные сообщения могут быть описаны с помощью специальной mx структуры. Микроядро объединяет части такого сообщения в единый непрерывный поток данных.

Следующие функции позволяют обрабатывать составные сообщения:

Creceivemx();

Readmsgmx();

Receivemx();

Replymx();

Sendmx();

Writemsgmx().

Зарезервированные коды сообщений

В изделии все сообщения начинаются с 16-ти битного слова, называемого *кодом сообщения*. Заметим, однако, что это не является обязательным для вас требованием при написании собственных программ. Изделие использует коды сообщений в диапазонах, указанных в таблице 1.2.

Таблица 1.2. Зарезервированный диапазоны кодов

Зарезервированный диапазон кодов сообщения	**Назначение**
0x0000 - 0x00FF	Сообщения Администратора процессов
0x0100 - 0x01FF	Сообщения ввода/вывода (общие для всех серверов ввода/вывода)
0x0200 - 0x02FF	Сообщения Администратора файловой системы
0x0300 - 0x03FF	Сообщения Администратора устройств
0x0400 - 0x04FF	Сообщения Администратора сети
0x0500 - 0x0FFF	Зарезервированы для будущих системных компонентов КПДА.00002-01

1.1.3. IPC посредством прокси

Прокси - это форма неблокирующего сообщения, наиболее подходящая для извещения о наступлении события, когда посылающий процесс не нуждается в диалоге с получателем. Единственная функция прокси состоит в посылке фиксированного сообщения определенному процессу, который является владельцем прокси. Подобно сообщениям, прокси могут быть использованы в пределах всей сети.

Используя прокси, процесс или обработчик прерывания может послать сообщение другому процессу, не блокируясь и не ожидая ответа.

Примеры использования прокси:

1) процесс желает известить другой процесс о наступлении какого-либо события, но при этом не может позволить себе посылку сообщения (в этом случае он оставался бы блокированным до тех пор, пока получатель не вызовет Receive() и Reply());

2) процесс хочет послать данные другому процессу, но при этом ему не требуется ни ответа, ни какого-либо другого подтверждения того, что адресат (получатель) получил сообщение;

3) обработчик прерывания хочет известить процесс о поступлении новых данных.

Для создания прокси используется функция языка Си. Любой другой процесс или обработчик прерывания, которому известен идентификатор прокси, может воспользоваться функцией языка Си Trigger() для того, чтобы заставить прокси передать заранее заданное сообщение. Запрос Trigger() обрабатывается Микроядром.

Прокси может быть "запущено" неоднократно - каждый раз при этом оно посылает сообщение. Прокси может накапливать очередь длиной до 65535 сообщений.

Пример использования прокси приведен на рис. 6.

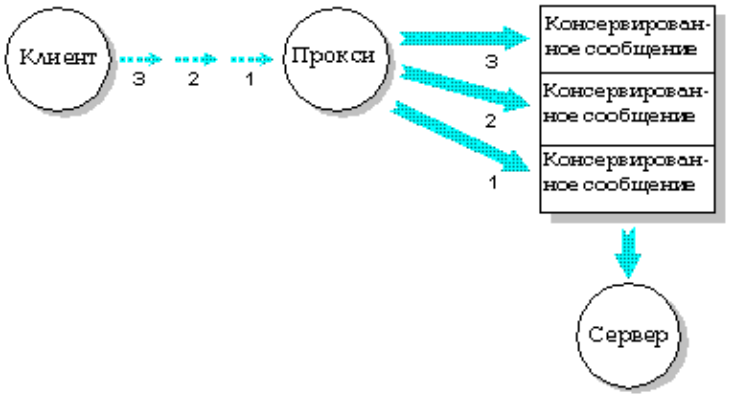


Рис. 6. Процесс-клиент запускает прокси 3 раза, в результате чего сервер получает 3 «консервированных» сообщения от прокси.

1.1.4. IPC посредством сигналов

Сигналы являются традиционным способом асинхронной связи, которая используется в различных операционных системах.

Механизм порождения сигнала

Считается, что сигнал доставлен процессу тогда, когда выполняется определенное в процессе для данного сигнала действие. Процесс может посылать сигнал самому себе.

Таблица 1.3. Инструменты порождения сигналов

Задача	**Инструмент**
Породить сигнал из командной строки	Утилиты: kill и slay
Породить сигнал внутри процесса	Функции Си: kill() и raise()

Получение сигналов

Процесс может принять сигнал одним из трех способов, в зависимости от того, как в процессе определена обработка сигналов:

1) если процесс не определил никаких специальных мер по обработке сигнала, то выполняется предусмотренное для сигнала действие по умолчанию - обычно таким действием по умолчанию является завершение работы процесса;

2) процесс может игнорировать сигнал. Если процесс игнорирует сигнал, то сигнал не оказывает на процесс никакого воздействия (учтите, что SIGCONT, SIGSTOP и SIGKILL не могут быть игнорированы в обычных обстоятельствах);

3) процесс может предусмотреть обработчик сигнала - функцию, которая будет вызываться при приеме сигнала. Если процесс содержит обработчик для какого-либо сигнала, говорят, что процесс может "поймать" этот сигнал. Любой процесс, который "ловит" сигнал, фактически получает особый вид программного прерывания. Никакие данные с сигналом не передаются.

В промежутке времени между моментом, когда сигнал порожден, и моментом, когда он доставлен, сигнал называется ожидающим. Для процесса ожидающими одновременно могут быть несколько различных сигналов. Сигналы доставляются процессу, когда планировщик ядра делает этот процесс готовым к выполнению.

Процесс не должен делать никаких предположений относительно порядка, в котором будут доставлены ожидающие сигналы.

Управление обработкой сигналов

Чтобы задать желаемый способ обработки для каждого из сигналов, можно использовать функции языка Си signal() стандарта ANSI или sigaction() стандарта POSIX.

Функция sigaction() предоставляет большие возможности по управлению обработкой сигналов.

Способ обработки сигнала можно изменить в любой момент времени. Если указать, что сигнал данного типа должен игнорироваться, то все ждущие сигналы такого типа будут немедленно отброшены.

Обработчики сигналов

Вызов обработчика сигнала аналогичен программному прерыванию. Он выполняется асинхронно по отношению к остальной части процесса. Таким образом, существует вероятность того, что обработчик сигнала будет вызван во время выполнения любой из имеющихся в программе функций (включая библиотечные функции).

Если в программе не предусмотрен возврат из обработчика сигнала, то могут быть использованы функции siglongjmp() либо longjmp() языка Си. Функция siglongjmp() предпочтительнее. При использовании функции longjmp() сигнал остается блокированным.

Блокирование сигналов

При возникновении необходимости временно запретить получение сигнала, не изменяя метод его обработки, используется набор функций, позволяющих блокировать получение сигналов. Блокированный сигнал остается ожидающим; после разблокирования он будет доставлен исполняемой программе.

Пока программа выполняет обработчик сигнала для определенного типа сигнала, микроядро автоматически блокирует этот сигнал. Это означает, что нет необходимости заботиться о вложенных вызовах обработчика сигнала. Каждый вызов обработчика сигнала - это неделимая операция по отношению к доставке

следующих сигналов этого типа. Если процесс выполняет нормальный возврат из обработчика, сигнал автоматически разблокируется.

Замечание: Реализация обработчиков сигналов в некоторых UNIX-системах отличается тем, что при получении сигналов они не блокируют его, а устанавливают действие по умолчанию. В результате некоторые UNIX приложения вызывают функцию signal() изнутри обработчика сигналов, чтобы подготовить обработчик к следующему вызову. Такой способ имеет два недостатка. Во-первых, если следующий сигнал поступает, когда программа уже выполняет обработчик сигнала, но еще не вызвала функцию signal(), то программа может быть завершена. Во-вторых, если сигнал поступает сразу после вызова функции signal() в обработчике, то возможен рекурсивный вход в обработчик сигнала. Изделие поддерживает блокирование сигналов и, таким образом, не страдает ни от одного из этих недостатков. При этом не нужно вызывать функцию signal() изнутри обработчика сигналов. Если необходимо покинуть обработчик через дальний переход (long jump), следует использовать функцию siglongjmp().

Сигналы и сообщения

Между сигналами и сообщениями существует взаимодействие. Если процесс SEND-блокирован или RECEIVE-блокирован в момент получения сигнала, и предусмотрен обработчик сигнала, то выполняются следующие действия:

а) процесс разблокируется;

б) выполняется обработка сигнала;

в) функция Send() или Receive() возвращает код ошибки.

Если процесс был SEND-блокирован в момент получения сигнала, то проблемы не возникает, потому что получатель еще не получил сообщение. Но если процесс был RECEIVE-блокирован, то неизвестно, было ли обработано посланное им сообщение и, следовательно, не известно, следует ли повторять Send().

Процесс, играющий роль сервера (т.е. получающий сообщения), может запросить, чтобы он получал извещение всякий раз, когда его процесс-клиент получает сигнал, находясь в состоянии REPLY-блокирован. В этом случае клиент становится SIGNAL-блокированным с ожидающим сигналом. Сервер получает

специальные сообщения, описывающие тип сигнала. Сервер может затем выбрать один из следующих вариантов:

закончить выполнение запроса от клиента нормальным образом: клиент (процесс-отправитель) будет уведомлен, что его сообщение было обработано должным образом,

ИЛИ

освободить используемые ресурсы и возвратить код ошибки, указывающий, что процесс был разблокирован сигналом: клиент получит четкий признак ошибки.

Когда сервер отвечает процессу, который был SIGNAL-блокирован, то сигнал выдается непосредственно после возврата из функции Send(), вызванной клиентом (процессом-отправителем).

1.1.5. Организация взаимодействия между процессами в сети

Виртуальные каналы

Приложение, выполняющееся в среде изделия, может взаимодействовать с приложениями на другом компьютере сети так же, как с приложением на том же самом компьютере. С точки зрения приложения нет различия между локальным и удаленным ресурсом.

Такая степень прозрачности становится возможна благодаря *виртуальным каналам* (от английского термина virtual circuit, сокращенно VC). VC – это пути, которые Администратор сети предоставляет для передачи сообщений, прокси и сигналов по сети. VC способствуют эффективному использованию сети КПДА.00002-01 в целом по нескольким причинам:

1) Когда создается VC, он получает возможность обрабатывать сообщения вплоть до определенного размера; это означает, что пользователю можно предварительно выделить ресурс для обработки сообщения. Тем не менее, если необходимо послать сообщение, превышающее заданный максимальный размер, VC автоматически увеличивает размер буфера.

2) Если два процесса, располагающиеся на различных узлах сети, используют для связи друг с другом более чем один VC, то такие VC будут разделяемыми, так как между процессами будет *реально* существовать только один виртуальный канал.

Такая ситуация обычно имеет место, когда процесс использует несколько файлов на удаленной файловой системе.

3) Если процесс присоединяется к существующему разделяемому VC и запрашивает буфер большего размера, чем тот, который используется, то размер буфера автоматически увеличивается.

4) Когда процесс завершает выполнение, все связанные с ним VC автоматически освобождаются.

Виртуальные процессы

Процесс-отправитель отвечает за подготовку VC между собой и тем процессом, с которым он хочет установить связь. С этой целью процесс-отправитель обычно вызывает специальную функцию. Кроме создания VC, эта функция также создает на каждом конце канала виртуальный идентификатор процесса, сокращенно VID. Для каждого из процессов, VID на противоположном конце виртуального канала является идентификатором удаленного процесса, с которым он устанавливает связь. Процессы поддерживают связь друг с другом посредством этих VID.

Например, на рисунке 7 изображен виртуальный канал, который соединяет PID 1 и PID 2. На узле 20 - где находится PID 1 - VID представляет PID 2. На узле 40 - где находится PID 2 - VID представляет PID 1. И PID 1 и PID 2 могут обращаться к VID на своем узле так же, как к любому другому локальному процессу, посылая сообщения, принимая сообщения, поставляя сигналы, ожидая и т.д. Так, например, PID 1 может послать сообщение VID на своем конце, и этот VID передаст сообщение по сети к VID, представляющему PID 1 на другом конце. Этот VID затем отправит сообщение к PID 2.

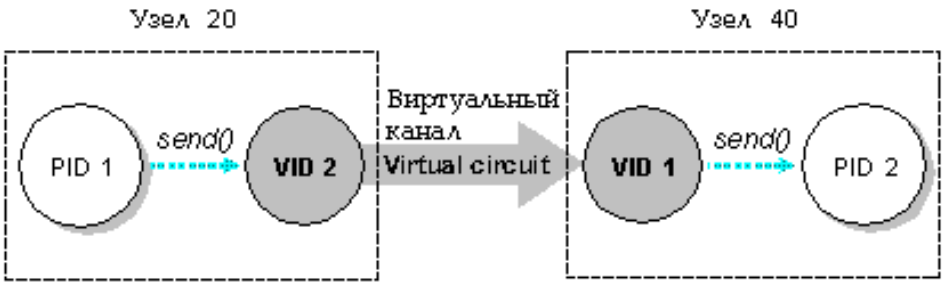


Рис. 7. Связь по сети реализуется посредством VC. Когда PID 1 посылает данные VID 2, посланный запрос передается по VC, в результате VID 1 направляет данные PID 2.

Каждый VID поддерживает соединение, которое имеет следующие атрибуты:

1) локальный pid;

2) удаленный pid;

3) удаленный nid (ID узла);

4) удаленный vid.

Если приложение хочет получить доступ к ресурсу ввода/вывода на другом узле сети, то VC создается библиотечной функцией open() от имени приложения. Приложение не принимает непосредственного участия в создании или использовании VC. Аналогичным образом, когда приложение определяет местонахождение сервера с помощью специальной функции, то VC автоматически создается от имени приложения. Для приложения VC просто является PID сервера.

Виртуальные прокси (Рис. 8)

Виртуальное прокси позволяет посылать прокси с удаленного узла, подобно тому, как виртуальный канал позволяет процессу обмениваться сообщениями с удаленным узлом.

В отличие от виртуального канала, который связывает вместе два процесса, виртуальный прокси может быть послан любым процессом на удаленном узле.

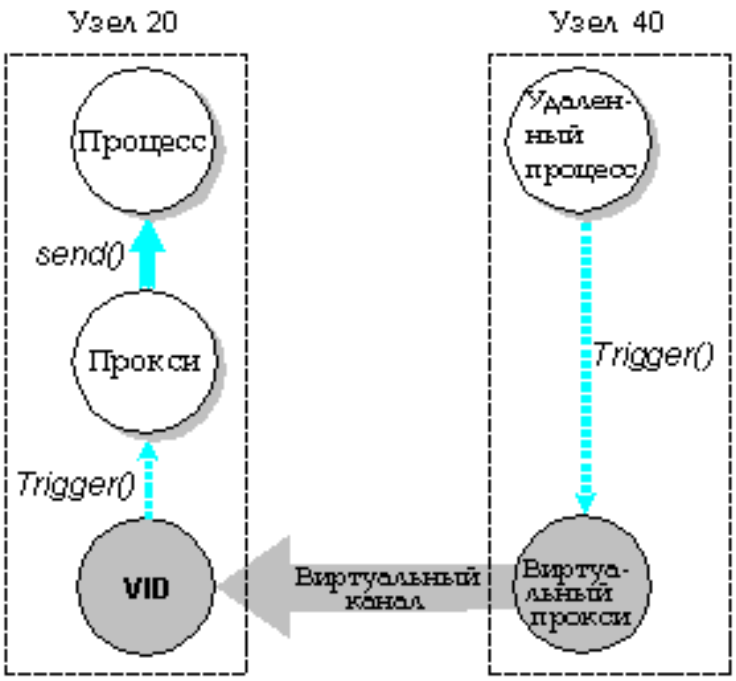


Рис. 8. На удаленном узле создается виртуальный прокси, который ссылается на локальный прокси.

Виртуальные прокси создаются специальной функцией, которой в качестве аргументов передаются узел (nid_t) и прокси (pid_t)

Следует иметь в виду, что на вызывающем узле виртуальный канал создается автоматически.

Отключение виртуальных каналов

Процесс может утратить возможность связи по установленному VC в результате различных причин:

- произошло отключение питания компьютера, на котором выполнялся процесс;
- был отсоединен сетевой кабель компьютера;
- удаленный процесс, с которым была установлена связь, завершил выполнение.

Любая из этих причин может помешать передаче сообщений по VC. Необходимо выявлять такие ситуации с тем, чтобы приложения могли предпринять корректирующие действия, либо корректно завершить свое выполнение. Если это не будет сделано, то ценные ресурсы могут оставаться без необходимости постоянно занятыми.

Администратор процессов на каждом узле проверяет целостность VC на своем узле. Он делает это следующим образом:

Каждый раз, когда происходит успешная передача по VC, обновляется временная метка, связанная с данным VC, в соответствии со временем последней операции.

Через определенные промежутки времени Администратор процессов проверяет каждый VC. Если не было зафиксировано никаких операций, то Администратор процессов посылает проверочный пакет Администратору процессов на другом конце VC.

Если ответ не получен или выявлен сбой, то VC помечается как неисправный. Затем делается определенное количество попыток восстановить соединение.

Если эти попытки не дают результата, то VC демонтируется; все процессы, блокированные VC, переходят в состояние READY (готов). Процесс получает код ошибки, возвращаемой ранее вызванной функцией связи.

Для установки параметров, связанных с данной проверкой целостности VC, необходимо использовать утилиту netpoll.

1.1.6. Синхронизация процессов с использованием семафоров

Другой распространенной формой синхронизации процессов являются семафоры. Операции "сигнализации" (sem_post()) и "ожидания" (sem_wait()) позволяют управлять выполнением процессов, переводя их в состояние ожидания либо выводя из него. Операция сигнализации увеличивает значение семафора на единицу, а операция ожидания уменьшает его на единицу.

При положительном значении семафора, при выполнении операции ожидания, процесс не блокируется. В противном случае операция ожидания блокирует процесс до тех пор, пока какой-либо другой процесс не выполнит операцию сигнализации. Допускается выполнение операции сигнализации один или более раз перед выполнением операции ожидания - это позволит одному или нескольким процессам выполнить операцию ожидания, не блокируясь.

Важное отличие между семафорами и другими примитивами синхронизации заключается в том, что семафоры "асинхронно безопасны" и могут использоваться обработчиками сигналов. Если требуется, чтобы обработчик сигнала выводил процесс из состояния ожидания, то семафоры являются правильным выбором.

1.1.7. Механизмы и методы диспетчеризации процессов

Планировщик микроядра принимает решение по диспетчеризации в следующих случаях:

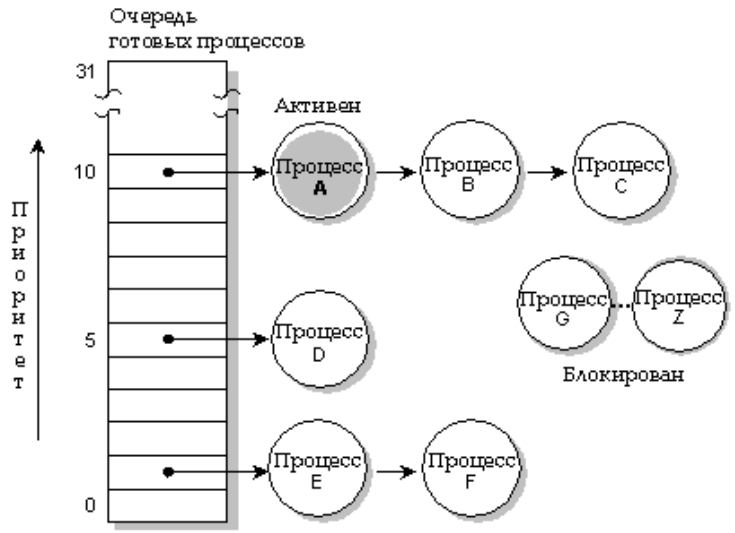
1) после разблокировки процесса;

2) по истечении временного интервала (кванта) для выполняющегося процесса;

3) в случае прерывания текущего процесса.

Каждому из процессов присваивается некоторый приоритет. Планировщик выбирает для выполнения следующий процесс, находящийся в состоянии READY, в соответствии с его приоритетом. (Программа в состоянии READY - это программа, которая способна использовать ЦП). Для выполнения выбирается процесс с наивысшим приоритетом. На рисунке 9 пояснен механизм действия приоритетов.

Приоритеты, присваиваемые процессам, находятся в диапазоне от 0 (наименьший) до 31 (наивысший). Уровень приоритета по умолчанию для создаваемого процесса наследуется от его родителя; для приложений, запускаемых



командным процессором, приоритет обычно равен 10.

Рис. 9. Приоритеты процессов.

На рисунке 9 показана ситуация, когда в очереди шесть процессов (A-F), готовых к выполнению и находящихся в состоянии READY. Остальные процессы (G-Z) блокированы. В данный момент выполняется процесс A. Процессы A, B и C

имеют наивысший приоритет, поэтому будут разделять центральный процессор в соответствии с алгоритмом диспетчеризации для выполняемого процесса.

Для определения или установления приоритетов процессов следует использовать функции языка Си:

1) Для определения приоритета процесса – getprio();
2) Для установки процессу приоритета – setprio().

Методы (дисциплины) диспетчеризации

В КПДА.00002-01 существуют три дисциплины диспетчеризации:

1) FIFO;

2) карусельный;

3) адаптивный.

Каждый процесс в системе может выполняться, используя любой из этих методов. Они действуют применительно к каждому отдельному процессу, а не ко всем процессам на узле.

Эти дисциплины диспетчеризации применимы, только когда два или более процесса с одинаковым приоритетом находятся в состоянии READY (т.е. эти процессы непосредственно конкурируют друг с другом). Если процесс с более высоким приоритетом переходит в состояние READY, то он немедленно вытесняет все процессы с более низким приоритетом.

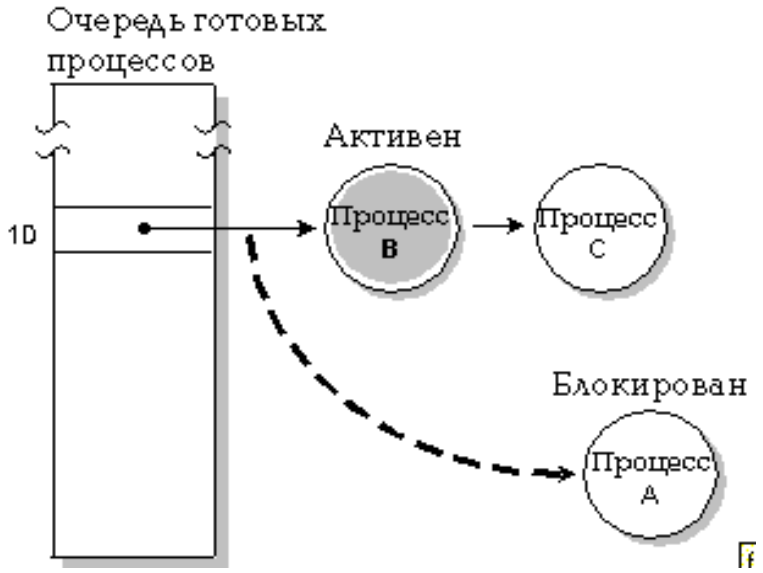


Рис. 10. Процесс А блокируется, процесс В выполняется.

На рисунке 10 три процесса с одинаковым приоритетом находятся в состоянии READY. Если процесс А блокируется, то выполняется процесс В.

Дисциплина диспетчеризации процесса наследуется от его родительского процесса (определить его из программы можно, используя функцию getsceduler() языка Си). Процесс имеет возможность изменить свою дисциплину диспетчеризации с помощью функции языка Си setscheduler().

FIFO диспетчеризация

При FIFO диспетчеризации процесс продолжает выполнение, пока не наступит момент, когда он:

1) либо добровольно уступает управление (т.е. выполняет любой вызов ядра);

2) либо вытесняется процессом с более высоким приоритетом.

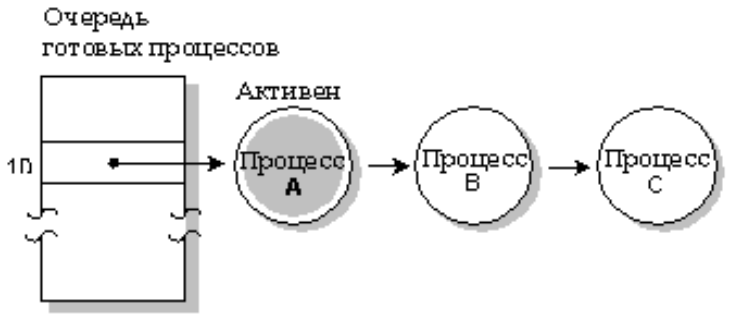


Рис. 11. FIFO диспетчеризация. Процесс А выполняется до тех пор, пока не блокируется.

Два процесса, которые выполняются с одним и тем же приоритетом (рис. 11), могут использовать метод FIFO для организации взаимоисключающего доступа к разделяемому (т.е. совместно используемому) ресурсу. Ни один из них не будет вытеснен другим во время своего выполнения. Так, например, если они совместно используют сегмент памяти, то каждый из этих двух процессов может обновлять данные в этом сегменте, не прибегая к использованию какого-либо способа синхронизации (например, семафора).

Карусельная диспетчеризация

При карусельной диспетчеризации (рис. 12) процесс продолжает выполнение, пока не наступит момент, когда он:

1) добровольно уступает управление (т.е. блокируется);

2) вытесняется процессом с более высоким приоритетом;

3) использовал свой квант времени (timeslice).

Квант времени – это интервал времени, выделяемый каждому процессу. После того, как процесс использовал свой квант времени, управление передается следующему процессу, который находится в состоянии READY и имеет такой же уровень приоритета. Квант времени равен 50 миллисекундам.

Замечание: За исключением квантования времени, карусельная диспетчеризация идентична FIFO-диспетчеризации.

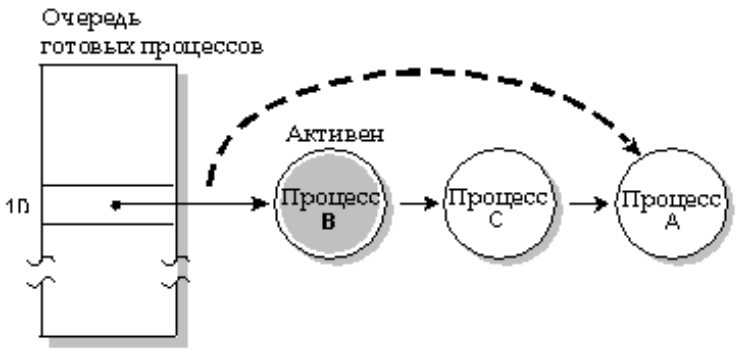


Рис. 12. Карусельная диспетчеризация.

На рисунке 12 показана карусельная диспетчеризация. Процесс A выполняется до тех пор, пока он не использовал свой квант времени; затем выполняется следующий процесс, находящийся в состоянии READY (процесс B).

Адаптивная диспетчеризация

При адаптивной диспетчеризации (рис. 13) процесс ведет себя следующим образом:

а) если процесс использовал свой квант времени (т.е. он не блокировался), то его приоритет уменьшается на 1. Это получило название *снижение приоритета* (priority decay). "Пониженный" процесс не будет продолжать "снижаться", даже если он

использовал еще один квант времени и не блокировался - он снизится только на один уровень ниже своего исходного приоритета;

б) если процесс блокируется, то ему возвращается первоначальное значение приоритета. Можно использовать адаптивную диспетчеризацию в тех случаях, когда процессы, производящие интенсивные вычисления, выполняются в фоновом режиме одновременно с интерактивной работой пользователей.

Адаптивная диспетчеризация дает производящим интенсивные вычисления процессам достаточный доступ к ЦП и в то же время сохраняет быстрый интерактивный отклик для других процессов.

Адаптивная диспетчеризация является методом диспетчеризации, использующимся по умолчанию для программ, запускаемых командным интерпретатором.

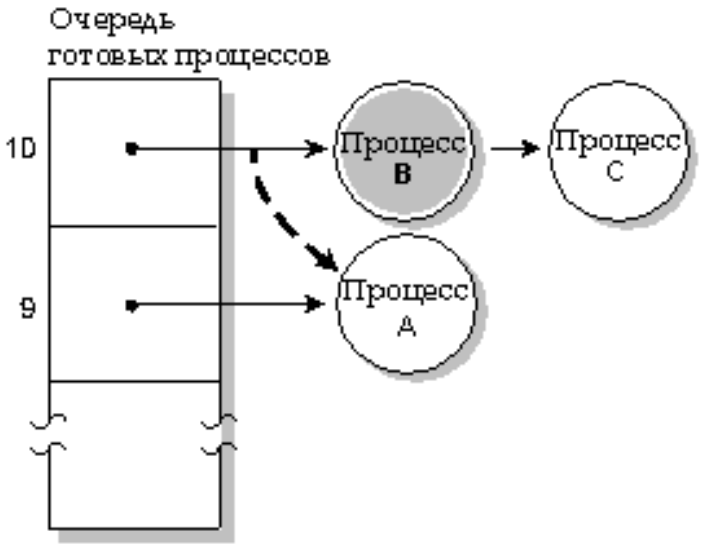


Рис. 13. Адаптивная диспетчеризация.

На рисунке 13 показана следующая ситуация: процесс А использовал свой квант времени; его приоритет снизился на 1. Выполняется следующий процесс в состоянии READY (процесс В).

Клиент - управляемый приоритет

В КПДА.00002-01 обмен данными между процессами в большинстве случаев организован с использованием модели *клиент/сервер*. *Серверы* выполняют

некоторые сервисные функции, а *клиенты*, посылая сообщение серверу, запрашивают эти услуги. Как правило, серверы более надежны и жизнеспособны, чем клиенты.

Количество клиентов обычно больше, чем серверов. Как следствие этого, принято запускать сервер с приоритетом более высоким, чем у любого из его клиентов. Может использоваться любой из трех рассмотренных выше методов диспетчеризации, но, наверное, самым распространенным является карусельный.

Если клиент с низким уровнем приоритета посылает сообщение серверу, то его запрос выполняется под более высоким уровнем приоритета, тем, который имеется у сервера. Это косвенным образом повышает приоритет клиента, т.к. именно его запрос заставил сервер выполняться.

Пока для выполнения запроса серверу достаточно короткого промежутка времени, проблем обычно не возникает. Если же выполнение запроса занимает у сервера более продолжительное время, то клиент с низким приоритетом может неблагоприятно повлиять на выполнение других процессов, приоритеты которых выше, чем у клиента, но ниже, чем у сервера.

Чтобы решить эту дилемму, сервер может поставить свой приоритет в зависимость от приоритета того клиента, чей запрос он выполняет. Когда сервер получает сообщение, приоритет будет установлен таким же, как у клиента. Обратите внимание, что меняется только приоритет сервера – его дисциплина диспетчеризации остается неизменной. Если во время выполнения запроса сервер получает другое сообщение, и приоритет нового клиента выше, чем у сервера, то приоритет сервера повышается. Фактически, новый клиент "заряжает" сервер до своего уровня приоритета, позволяя ему закончить выполнение текущего запроса и приступить к обработке запроса нового клиента. Если этого не делать, то фактически приоритет нового клиента понизится, пока он блокирован на сервере с низким приоритетом.

Если для сервера задается клиент-управляемый приоритет, то следует также запросить доставку сообщений в порядке убывания приоритетов (в противоположность хронологическому).

1.1.8. Механизмы работы в реальном времени

Для систем реального времени жизненно важно, чтобы такты работы ЦП расходовались наиболее эффективно. Также очень важно свести к минимуму время, которое проходит с момента наступления внешнего события до начала выполнения кода программы, ответственной за обработку данного события. Это время называется *задержкой*.

В КПДА.00002-01 встречается несколько видов задержек:

а) задержка прерывания;

б) задержка диспетчеризации.

Задержка прерывания (рис. 14)

Задержка прерывания – это интервал времени между аппаратным прерыванием и выполнением первой команды программного обработчика прерывания. КПДА.00002-01 практически все время оставляет прерывания полностью разрешенными, поэтому задержка прерывания, как правило, не существенна. Однако некоторые критические фрагменты кода требуют, чтобы прерывания были временно запрещены. Максимальная продолжительность такого запрета обычно определяет худший случай задержки прерывания – в КПДА.00002-01 это очень небольшая величина.

Следующая диаграмма иллюстрирует случай, когда аппаратное прерывание обрабатывается в установленном обработчиком прерывании. Обработчик прерывания либо просто выполнит команду возврата, либо при возврате запустит прокси.

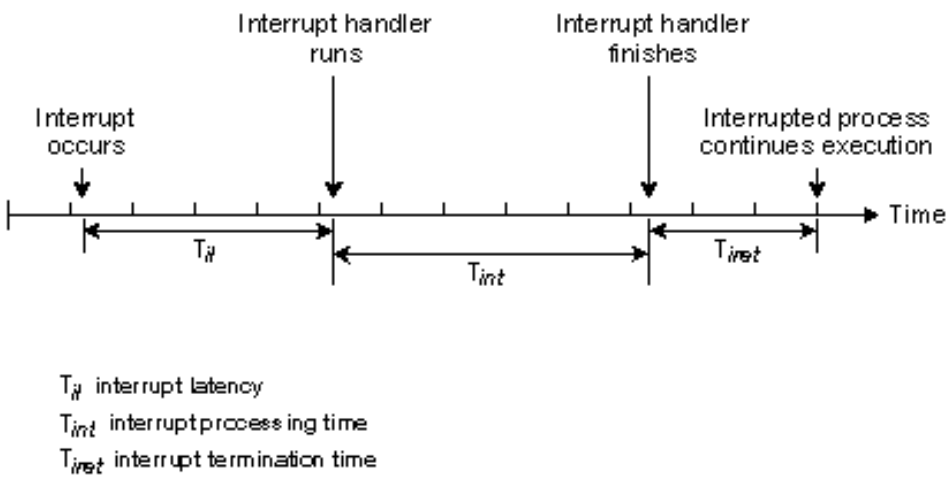


Рис.14. Обработчик прерывания просто завершается

Задержка прерывания (T*il*) на приведенной выше диаграмме отражает *минимальную* задержку - случай, когда прерывания были полностью разрешены в момент прерывания. В худшем случае задержка прерывания составит это время *плюс* время, в течение которого КПДА.00002-01 или выполняющийся процесс запрещает прерывания ЦП.

Типичные задержки прерывания (T*il)* для различных процессоров представлены в таблице 1.4.

Таблица 1.4. Задержки прерываний

Задержка прерывания (T*il*)	**Процессор**
3.3 микросекунды	166 МГц Pentium
4.4 микросекунды	100 МГц Pentium
5.6 микросекунды	100 МГц 486DX4
22.5 микросекунды	33 МГц 386EX

Задержка диспетчеризации

В некоторых случаях обработчик аппаратного прерывания низкого уровня должен передать управление процессу более высокого уровня. В этом случае обработчик прерывания перед выполнением команды "возврат" запускает прокси.

Здесь имеет место второй вид задержки - *задержка диспетчеризации*, - с которой также надо считаться.

Задержка диспетчеризации – это время между завершением работы обработчика прерывания и выполнением первой команды процесса-драйвера (рис.15). Это время, необходимое для сохранения контекста, выполняющегося в данный момент времени процесса, и восстановления контекста требуемого драйвера. В КПДА.00002-01 это время также невелико, хотя и больше задержки прерывания.

Важно отметить, что обработка *большинства* прерываний завершается без запуска прокси. В большинстве случаев обработчик прерывания сам может выполнить все необходимые действия. Запуск прокси, чтобы "разбудить" *драйвер*, процесс более высокого уровня, происходит только при наступлении важного события. Например, обработчик прерывания для драйвера последовательного порта при каждом прерывании "регистр передачи свободен" будет передавать один байт данных и запустит процесс более высокого уровня (Dev) только тогда, когда выходной буфер, наконец, опустеет.

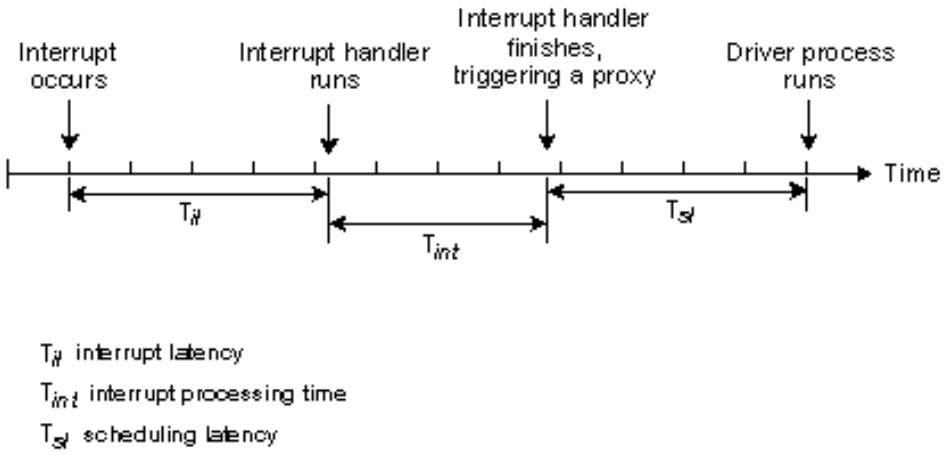


Рис. 15. Обработчик прерывания завершает работу и запускает прокси.

Типичные значения задержки диспетчеризации (T*sl)* для различных процессоров представлены в таблице 1.5.

Таблица 1.5. Задержки диспетчеризации.

Задержка диспетчеризации (T*sl*):	**Процессор:**

Задержка диспетчеризации (T*sl*):	**Процессор:**
4.7 микросекунды	166 МГц Pentium
6.7 микросекунды	100 МГц Pentium
11.1 микросекунды	100 МГц 486DX4
74.2 микросекунды	33 МГц 386EX

Организация стека прерываний

В изделии поддерживается заложенный в архитектуре x86 механизм назначения приоритетов аппаратным прерываниям. Это позволяет прерываниям с более высоким приоритетом вытеснять прерывания с меньшим приоритетом.

Выше были рассмотрены простейшие - и наиболее типичные - ситуации, когда происходит только одно прерывание. Практически такой же расчет времени справедлив для прерывания с наивысшим приоритетом. При рассмотрении наихудшего случая для прерывания с низким приоритетом необходимо учитывать время обработки всех прерываний более высокого уровня, т.к. прерывание с более высоким приоритетом вытеснит прерывание с меньшим приоритетом.

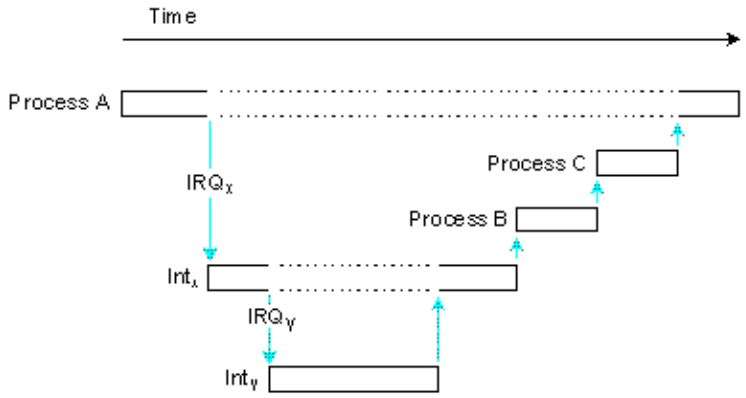


Рис. 16. Вытеснение прерываний

На рисунке 16 изображена следующая ситуация. Выполняется процесс A. Прерывание IRQ*x* вызывает выполнение обработчика прерывания Int*x*, который вытесняется IRQ*y* и его обработчиком Int*y*. Int*y* запускает прокси, вызывающее

выполнение процесса B, а Int*x* запускает прокси, вызывающее выполнение процесса C.

1.2. Администратор процессов

1.2.1. Общие сведения

1.2.1.1. Назначение Администратора процессов

Администратор процессов взаимодействует с микроядром для обеспечения выполнения функций, составляющих сущность операционной системы. Хотя он и использует то же адресное пространство, что и микроядро, но Администратор процессов выполняется как истинный процесс. И он, как и все остальные процессы, подвергается диспетчеризации со стороны микроядра и использует предоставляемые микроядром примитивы передачи сообщений для связи с другими процессами в системе.

Администратор процессов отвечает за создание новых процессов в системе и за управление основными ресурсами, связанными с процессом. Все эти услуги предоставляются посредством сообщений. Так, например, если процесс хочет породить новый процесс, он делает это, посылая сообщение с указанием атрибутов создаваемого процесса. Т.к. сообщения передаются по сети, то процесс может создаваться на другом узле сети посылкой сообщения Администратору процессов нужного узла.

1.2.1.2. Примитивы создания процессов

Изделие поддерживают три примитива создания процесса:

1) fork();

2) exec();

3) spawn().

Примитив fork()

Примитив fork() создает новый процесс, который является точной копией вызвавшего его процесса. Новый процесс использует тот же самый код, что и породивший его процесс, и наследует копию всех данных родительского процесса.

Примитив exec()

Примитив exec() заменяет вызвавший процесс новым. После успешного вызова exec() возврата не происходит, т.к. образ вызывающего процесса заменяется образом нового процесса. Обычной практикой в POSIX-системах для создания нового

процесса - без удаления вызывающего процесса - является сначала вызов fork(), а затем вызов exec() из порожденного процесса.

Примитив spawn()

Примитив spawn() создает новый процесс как потомок вызывающего процесса. С его помощью можно избежать вызовов fork() и exec(), используя более быстрый и эффективный способ создания новых процессов. В отличие от fork() и exec(), которые по своей природе выполняются на том же самом узле, что и вызывающий процесс, примитив spawn() может создавать процессы на любом узле сети.

1.2.1.3. Наследование

Когда с помощью одного из трех рассмотренных выше примитивов задается новый процесс, он наследует многое из своего окружения от родителя. Сведения о наследовании атрибутов сведены в таблицу 1.6.

Таблица 1.6. Наследование параметров.

Наследуемый параметр	***fork()***	***exec()***	***spawn()***
Идентификатор процесса	нет	да	нет
Открытые файлы	да	по выбору*	по выбору
Блокировка файлов	нет	да	нет
Ожидающие сигналы	нет	да	нет
Маска сигналов	да	по выбору	по выбору
Игнорируемые сигналы	да	по выбору	по выбору
Обработчики сигналов	да	нет	нет
Переменные окружения	да	по выбору	по выбору
Идентификатор сеанса	да	да	по выбору
Группа процесса	да	да	по выбору
Реальные UID, GID	да	да	да
Эффективные UID, GID	да	по выбору	по выбору
Текущий рабочий каталог	да	по выбору	по выбору
Маска создания файлов	да	да	да
Приоритет	да	по выбору	по выбору
Алгоритм диспетчеризации	да	по выбору	по выбору
Виртуальные каналы	нет	нет	нет

Наследуемый параметр	***fork()***	***exec()***	***spawn()***
Символьные имена	нет	нет	нет
таймеры реального времени	нет	нет	нет

Примечание. *по выбору: вызывающий процесс может по необходимости выбрать – да или нет.

1.2.2. Описание жизненного цикла процесса

Процесс проходит через четыре стадии:

1) Создание;

2) Загрузка;

3) Выполнение;

4) Завершение.

Создание нового процесса состоит из присвоения новому процессу идентификатора (ID) процесса и подготовки информации, которая определяет окружение нового процесса. Большая часть этой информации наследуется от родительского процесса.

Загрузка образа процесса производится *нитью загрузчика*. Код загрузчика находится в Администраторе процессов, но нить выполняется под ID нового процесса. Это позволяет Администратору процессов обрабатывать и другие запросы во время загрузки программы.

После того, как код программы загружен, процесс готов к выполнению; он начинает конкурировать с остальными процессами за ресурсы ЦП. Все процессы выполняются параллельно со своими родителями. Кроме того, смерть родительского процесса *не* означает автоматическую смерть его дочерних процессов.

Процесс завершается одним из двух способов:

1) процесс получает сигнал, который вызывает завершение процесса;

2) процесс вызывает функцию Си exit(), либо в явном виде, либо в качестве действия по умолчанию при возврате из функции main().

Завершение включает две стадии:

Выполняется *нить завершения* в Администраторе процессов. Этот "заслуживающий доверия" код расположен в Администраторе процессов, но нить выполняется с ID завершающегося процесса. Эта нить закрывает все открытые файловые дескрипторы и освобождает следующие ресурсы:

- все виртуальные каналы, принадлежащие процессу;
- всю память, выделенную процессу;
- все символьные имена;
- любые старшие номера устройств (только администраторы ввода/вывода);
- любые обработчики прерывания;
- любые прокси;
- любые таймеры.

После того, как нить завершения выполнена, извещение о завершении процесса посылается родительскому процессу (эта фаза выполняется внутри Администратора процессов).

Если родительский процесс не вызвал wait() или waitpid(), то дочерний процесс становится "зомби" и не будет завершен, пока родительский процесс не вызовет wait() или не завершит выполнение. (Если необходимо, чтобы процесс *ждал* смерти дочерних процессов, то необходимо либо установить _SPAWN_NOZOMBIE флаг, либо установить действие SIG_IGN для сигнала SIGCHLD посредством функции signal(). Таким образом, дочерние процессы не будут становиться зомби после смерти.)

Родительский процесс может ждать смерти дочернего процесса, запущенного на удаленном узле. Если родитель процесса-зомби умирает, то зомби освобождается.

Если запущена утилита dumper и процесс завершается в результате получения сигнала, то генерируется дамп памяти. Этот дамп можно исследовать с помощью отладчика.

1.2.3. Состояния процесса

Процесс всегда находится в одном из следующих состояний (Рис. 17):

1) READY (готов) - процесс способен использовать ЦП (т.е. он не ждет наступления какого-либо события).

2) BLOCKED (блокирован) - процесс в одном из следующих блокированных состояний:

- SEND-блокирован;
- RECEIVE-блокирован;
- REPLY-блокирован;
- SIGNAL-блокирован;
- SEMAPHORE-блокирован.

3) HELD (приостановлен) - процесс получил сигнал SIGSTOP. В этом состоянии процесс не имеет права использовать ЦП; выйти из состояния HELD процесс может только в результате получения сигнала SIGCONT, или завершив свою работу после получения другого сигнала.

4) WAIT (блокирован) - процесс выполнил вызов функции wait() или waitpid() для ожидания сообщения о завершении выполнения дочернего процесса.

5) DEAD (мертв) - процесс завершил выполнение, но не может послать сообщения об этом родительскому процессу, т.к. родительский процесс не вызвал wait() или waitpid(). Когда процесс находится в состоянии DEAD, ранее занимаемая им память уже освобождена. Процесс в состоянии DEAD также называют зомби.

Возможные переходы процесса из одного состояния в другое показаны на рисунке 17:

1) Процесс посылает сообщение.

2) Процесс-адресат получает сообщение.

3) Процесс-адресат отвечает на сообщение.

4) Процесс ожидает сообщения.

5) Процесс получает сообщение.

6) Сигнал разблокирует процесс.

7) Сигнал пытается разблокировать процесс; адресат ранее запросил извещение о сигнале путем посылки сообщения.

8) Процесс-адресат получает сообщение о сигнале.

9) Процесс ожидает смерти дочернего процесса.

10) Дочерний процесс умирает, либо сигнал разблокирует процесс.

11) Процесс получает SIGSTOP.

12) Процесс получает SIGCONT.

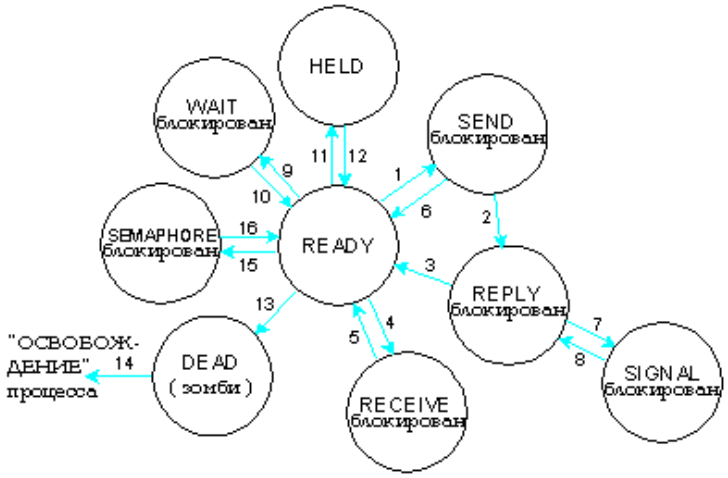


Рис. 17. Возможные состояния процесса в КПДА.00002-01.

13) Процесс умирает.

14) Родительский процесс вызывает функцию wait() или waitpid(), завершает ее выполнение, либо ранее уже завершил выполнение.

15) Процесс вызывает функцию semwait() для семафора с неположительным значением.

16) Другой процесс вызывает функцию sempost(), или приходит немаскированный сигнал.

1.2.3.1. Определение состояний процесса

Для определения состояния конкретного процесса из командного интерпретатора (Shell), используется утилиты ps и sin.

Для определения состояния операционной системы в целом из командного интерпретатора (Shell) используется утилита sin.

Утилита ps определена стандартом POSIX; ее использование в командных файлах *может быть* переносимым в другие системы. Утилита sin, напротив, уникальна для КПДА.00002-01: она дает полезную информацию, специфическую для изделия, которую невозможно получить, используя утилиту ps.

1.2.4. Символьные имена процессов

Изделие ориентировано на то, что для него будут разрабатываться приложения, которые разбиты на несколько взаимодействующих процессов. Такое приложение, которое существует как группа взаимодействующих процессов, имеет большие возможности распараллеливания и может быть распределено по сети для повышения производительности системы в целом.

Однако разделение приложений на взаимодействующие процессы требует ряда специальных соображений. Чтобы взаимодействующие процессы могли установить связь между собой, они должны иметь возможность определить идентификаторы друг друга. Например, имеется сервер базы данных, который может обслуживать произвольное количество клиентов. Клиенты могут запускаться и прекращать работу в любое время, однако сервер всегда остается доступным.

Процессы-клиенты узнают идентификатор процесса-сервера базы данных с тем, чтобы послать ему сообщение, следующим образом.

В изделии процессы могут присваивать себе символьные имена. В контексте одного узла процесса могут зарегистрировать эти имена у Администратора процессов на том узле, на котором они выполняются. Остальные процессы могут затем запросить у Администратора процессов идентификатор процесса, ассоциированный с определенным именем.

Ситуация становится более сложной, если имеется сеть, в которой сервер обслуживает клиентов, находящихся на различных узлах. В изделии предусмотрена поддержка как глобальных, так и локальных имен. Глобальные имена определены для всей сети, в то время как локальные имена определены только для того узла, на котором они зарегистрированы. Глобальные имена начинаются с косой черты (/). Например:

kpda/fsys	локальное имя
company/xyz	локальное имя
/company/xyz	глобальное имя

Рекомендуется использовать уникальные имена (например, название организации) в качестве префикса для всех регистрируемых имен с тем, чтобы избежать конфликтов имен между программами разных производителей.

Для использования глобальных имен хотя бы на одном узле сети должен быть запущен процесс, известный как *определитель имен* (утилита nameloc). Этот процесс ведет учет всех зарегистрированных глобальных имен.

В каждый момент времени в сети могут быть запущены до десяти определителей имен. Каждый определитель хранит идентичные копии всех активных глобальных имен. Такая избыточность гарантирует нормальное функционирование сети, даже если один или несколько узлов, обеспечивающих определение имен, выйдут из строя одновременно.

1.2.5. Механизмы использования Таймеров

1.2.5.1. Исчисление времени

В изделии исчисление времени основывается на системном таймере, поддерживаемом операционной системой. Таймер содержит текущее значение Координированного Всемирного Времени (UTC) относительно 0 часов, 0 минут, 0 секунд, 1 января 1970 года. Для определения местного времени функции исчисления времени используют переменную окружения TZ.

1.2.5.2. Простые средства отсчета времени

Командные файлы и процессы могут делать паузу на определенный интервал времени, используя простые средства отсчета времени. В командных файлах для этой цели используется утилита sleep; в процессах используется функция Си sleep(). Также может использоваться функция delay(), которой в качестве аргумента передается длина интервала времени в миллисекундах.

1.2.5.3. Более сложные средства отсчета времени

Кроме того, процесс может создавать таймеры, устанавливать их на определенный интервал времени и удалять таймеры.

Создание таймеров

Процесс может создать один или больше таймеров. Эти таймеры могут быть любых типов и в любом сочетании с учетом конфигурируемого ограничения общего числа таймеров, поддерживаемых операционной системой (такое ограничение

задается опциями Proc32). Для создания таймера используется функция timer_create().

Установка таймеров

При установке таймеров можно использовать один из двух типов интервалов времени:

1) абсолютный - время относительно 0 часов, 0 минут, 0 секунд, 1 января 1970 года;

2) относительный - время относительно текущего показания часов.

Также можно создать таймер, периодически срабатывающий с заданным интервалом времени. Например, можно установить таймер таким образом, чтобы он сработал первый раз завтра в 9 часов утра и после этого срабатывал каждые 5 минут.

Можно установить временной интервал для существующего таймера. Результат будет зависеть от типа интервала времени:

- для абсолютного таймера новый интервал заменяет текущий интервал времени;

- для относительного таймера новый интервал добавляется к любому остающемуся интервалу времени.

Установка таймера осуществляется с использованием функции timer_settime().

Удаление таймера

Для удаления таймера используется функция Си timer_delete().

Установка разрешения таймера

Возможна установка разрешения для таймера с помощью утилиты ticksize.

Чтение таймера

Для чтения оставшегося от таймера интервала или для проверки, не был ли таймер удален, используется функция Си timer_gettime().

1.2.6. Обработчики прерываний

Обработчики прерываний обслуживают систему аппаратных прерываний компьютера - они реагируют на аппаратные прерывания и осуществляют передачу данных между компьютером и внешними устройствами на низком уровне.

Обработчики прерываний физически являются частью стандартного процесса изделия (например, драйвера), но они всегда выполняются асинхронно по отношению к процессу, в который они включены.

Обработчик прерываний реализует следующие функции:

- получает управление посредством дальнего вызова, а не непосредственно от прерывания (это может быть написано на Си, а не на ассемблере);

- выполняется в контексте процесса, в который он включен, таким образом имеется доступ ко всем глобальным переменным этого процесса;

- выполняется при разрешенных прерываниях, но вытесняется только в случае, если происходит прерывание с более высоким приоритетом;

- не должен непосредственно управлять контроллером прерывания 8259 (об этом заботится операционная система);

- должен быть как можно короче.

Несколько процессов могут обрабатывать одно и то же прерывание, если это поддерживается аппаратно. Когда происходит физическое прерывание, все обработчики прерываний будут по очереди получать управление. Не должно делаться никаких предположений относительно порядка, в котором будут вызываться обработчики прерываний, разделяющие одно и то же прерывание.

1.2.6.1. Обработчики прерываний таймера

Можно установить обработчик прерывания непосредственно для системного таймера таким образом, что обработчик будет вызываться по каждому прерыванию таймера. Чтобы установить период, необходимо использовать утилиту ticksize.

Можно также установить обработчик прерывания для отмасштабированного прерывания таймера, которое будет происходить каждые 50 миллисекунд, независимо от *tick size*. Эти таймеры предлагают низкоуровневую альтернативу POSIX 1003.4 таймерам.

1.3. Администратор устройств

1.3.1. Назначение Администратора устройств

Администратор устройств (Dev) является интерфейсом между процессами и *терминальными устройствами*. Эти терминальные устройства располагаются в пространстве имен ввода/вывода с именами, начинающимися с /dev. Например, консольное устройство будет иметь имя:

```
/dev/con1
```

1.3.2. Механизмы обслуживания устройств

Программы получают доступ к терминальным устройствам, используя стандартные функции open(), close(), read() и write(). Для процесса терминальное (оконечное) устройство представляется двунаправленным потоком байт, который может считываться и записываться процессом.

Администратор устройств регулирует поток данных между приложением и устройством. Dev выполняет некоторую обработку этих данных в соответствии с параметрами *управляющей структуры терминала* (называемой *termios*), которая существует для каждого устройства. Пользователи могут запрашивать и/или изменять эти параметры с помощью утилиты stty; программы могут использовать функции tcgetattr() и tcsetattr().

Параметры структуры *termios* управляют функциями низкого уровня, такими как:

- параметры линии (включая скорость передачи, контроль четности, стоп-биты, биты данных);
- эхо-вывод символов;
- редактирование строки ввода;
- распознавание и реакция на команду "Break" и зависания;
- программное и аппаратное управление потоком данных;
- преобразование выводимых символов.

Администратор устройств также предоставляет набор дополнительных сервисов, доступных процессам для работы с терминальным устройством. В таблице 1.7 приведены некоторые из задач, решаемых процессами с помощью этих сервисов.

Таблица 1.7. Задачи, решаемые сервисами Dev

Задача	**Функция Си**
Чтение с контролем времени	dev_read() или read() и tcsetattr()
Получение асинхронного извещения о доступных данных	dev_arm()
Ожидание полного завершения операции вывода на одном или более устройствах ввода	tcdrain()
Отправка команды Break по каналу связи	tcsendbreak()
Разрыв соединения	tcdropline()
Вставка входных данных	dev_insert_chars()
Выполнение неблокирующихся чтения и записи	open() и fcntl() (O_NONBLOCK mode)

1.3.3. Режим редактируемого ввода

Наиболее важный режим работы устройства управляется битом ICANON в управляющей структуре *termios*. Если этот управляющий бит установлен, то Администратор устройств выполняет функции редактирования строки для принимаемых символов.

Таким образом, только когда строка "введена" - обычно, когда получен символ перевода каретки (CR), - данные становятся доступными для прикладных процессов. Такой режим работы называется *редактируемым* (от английского *edited)*. Этот режим еще называют *canonical* (каноническим) и иногда *cooked* (приготовительным).

Большинство неполноэкранных приложений выполняются в редактируемом режиме. Типичным примером является командный интерпретатор (Shell).

Таблица 1.8 показывает, как Dev обрабатывает некоторые специальные управляющие символы, если соответствующие параметры установлены в структуре *termios*.

Таблица 1.8. Редактирование ввода администратором Dev

Действие Администратора устройств (Dev)	**Полученный символ**
Сдвиг курсора на один символ влево	LEFT
Сдвиг курсора на один символ вправо	RIGHT
Сдвиг курсора в начало строки	HOME
Сдвиг курсора в конец строки	END
Удаление символа слева от курсора	ERASE
Удаление символа в текущей позиции курсора	DEL
Удаление всей строки ввода	KILL
Стирание текущей строки и восстановление предыдущей	UP
Стирание текущей строки и восстановление следующей	DOWN
Переключение между режимами вставки и замены	INS

Символы редактирования строки отличаются для различных терминалов. При запуске драйвера консоли всегда определен полный набор редактирующих символов.

Если терминал подключен к ЭВМ через последовательный канал, то необходимо установить редактирующие символы, которые будут использоваться для данного конкретного терминала. Для этого можно использовать утилиту stty. Например, если терминал VT100 подключен к последовательному порту (/dev/ser1), то следующая команда извлечет соответствующие редактирующие символы из базы данных *terminfo* и применит их к /dev/ser1:

```
stty term=vt100 </dev/ser1
```

1.3.4. Режим необрабатываемого ввода

Когда бит ICANON не установлен, то устройство находится в *необрабатываемом* ("*сыром*", английский термин *raw*) режиме. В этом режиме не производится никакое редактирование ввода, а все получаемые данные немедленно становятся доступными для процессов КПДА.00002-01.

Полноэкранные программы и коммуникационные программы являются примерами приложений, которые переводят устройство в *сырой* режим.

При чтении из *сырого* устройства приложение может задать условия, при которых будет удовлетворен запрос на ввод данных. Критерии, используемые при приеме *сырых* данных, базируются на двух параметрах структуры *termios*: MIN и TIME. Приложение может определить еще один дополнительный параметр при вызове функции dev_read(). Этот параметр, TIMEOUT, используется при написании протоколов или приложений реального времени. Для функции read() TIMEOUT всегда 0.

Когда процесс запрашивает ввод n байт данных, эти три параметра определяют, когда должен быть удовлетворен запрос:

Таблица 1.9. Параметры для установки времени обработки запросов.

MIN	**TIME**	**TIMEOUT**	**Описание**
0	0	0	Вернуть немедленно столько байт, сколько доступно в данный момент (но не более n байт)
M	0	0	Вернуть не более n байт только тогда, когда доступны, по меньшей мере, M байт
0	T	0	Вернуть не более n байт только тогда, когда доступен хотя бы один байт либо истекло T x .1 секунд.
M	T	0	Вернуть не более n байт только тогда, либо когда будут доступны M байт, либо будет принят хотя бы один байт и интервал между любыми последовательно принятыми символами превысил T x .1 секунд
0	0	t	Зарезервировано.

MIN	TIME	TIMEOUT	Описание
M	0	t	Вернуть не более n байт только тогда, когда доступны M байт либо истекло t x .1 секунд
0	T	t	Зарезервировано
M	T	t	Вернуть не более n байт только тогда, когда доступны M байт, либо истекло t x .1 секунд и не был получен ни один символ, либо был принят хотя бы один байт и интервал между любыми последовательно принятыми символами превысил T x .1 секунд

На рисунке 18 показана типичная подсистема устройств в КПДА.00002-01.

1.3.5. Использование драйверов устройств

Администратор устройств (Dev) организует обмен данными между устройствами и прикладными программами. Аппаратный интерфейс управляется индивидуальными процессами - *драйверами*. Dev для каждого из *устройств* обменивается данными с *драйверами* через очереди в разделяемой памяти.

Замечание: Использование разделяемой памяти требует, чтобы Dev и драйверы находились на одном и том же физическом компьютере, преимуществом же является повышенная производительность.

Для каждого *терминального устройства* используются три очереди. Каждая очередь реализована на основе механизма "первый вошел - первый вышел". Каждой очереди также соответствует своя управляющая структура.

Принимаемые данные помещаются драйвером в очередь «сырого» ввода, и Dev извлекает их, только когда прикладные программы запрашивают данные. Обработчики прерываний внутри драйверов обычно вызывают проверенную библиотечную процедуру внутри Dev для добавления данных в эту очередь - это обеспечивает единообразный порядок ввода и существенно минимизирует ответственность драйвера.

Dev помещает выводимые данные в очередь вывода; данные извлекаются драйвером по мере того, как символы физически передаются устройству. Dev вызывает проверенную процедуру внутри драйвера каждый раз, когда добавляются

новые данные, таким образом он "подталкивает" драйвер к работе (в случае, если он не был занят). Благодаря использованию очередей вывода Dev реализует *буферизованную запись* (write-behind) для всех *терминальных устройств*. Только когда буферы вывода заполнены, Dev вызывает блокирование процесса при записи.

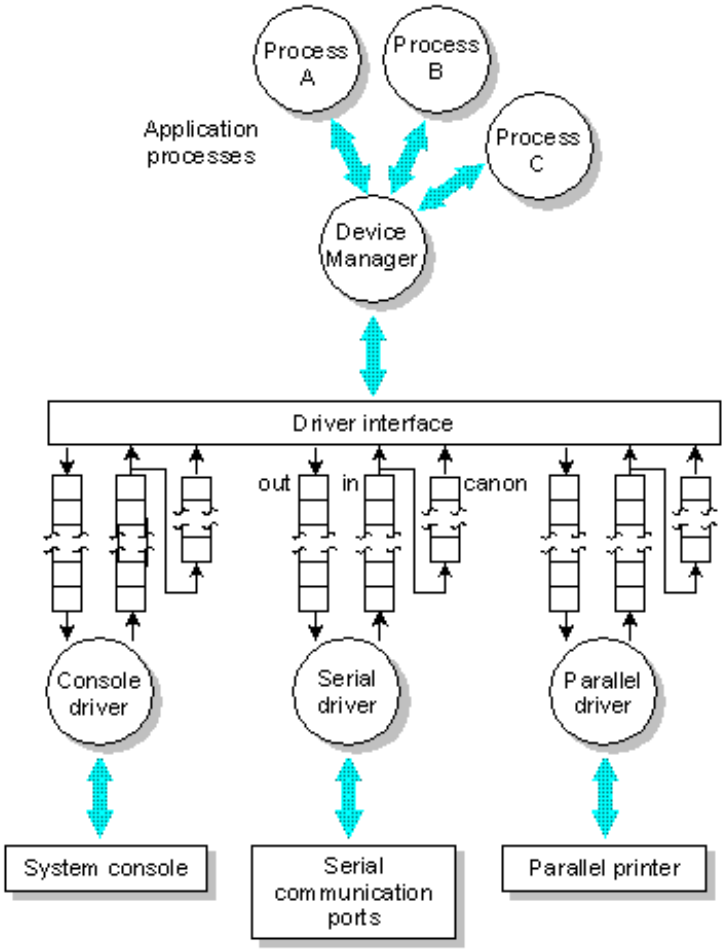


Рис. 18. Подсистема устройств в КПДА.00002-01.

Редактируемая очередь полностью управляется Dev и используется при вводе данных в *редактируемом* режиме. Размер этой очереди определяет максимальный размер строки редактируемого ввода для конкретного устройства.

Размеры всех этих очередей могут конфигурироваться системным администратором; единственное ограничение состоит в том, что общий суммарный размер всех трех очередей не может превышать 64K. Значений по умолчанию обычно более чем достаточно для большинства аппаратных конфигураций, но их можно "настраивать" либо для уменьшения общей потребности системы в памяти, либо в случае нестандартных аппаратных конфигураций.

1.3.6. Управление устройствами

Драйверы устройств добавляют получаемые данные в очередь «сырого» ввода или извлекают и передают данные из очереди вывода. Dev решает, когда вывод должен быть приостановлен, должно ли быть "эхо" принимаемых данных и т.д.

Для обеспечения хорошей реакции на входные события, Dev должен выполняться с достаточно высоким приоритетом. Dev обычно мало загружен работой, поэтому он редко снижает общую производительность системы. Сами драйверы, как и любые другие процессы в данной ОСРВ, могут выполняться с различными приоритетами в зависимости от характера обслуживаемых устройств.

Управление устройствами на низком уровне выполняется через дальний вызов входной точки *ioctl* внутри каждого драйвера. Общий набор *ioctl* команд поддерживается непосредственно Dev. Специфические для устройства *ioctl* команды могут быть посланы процессами драйверу через функцию Си.

1.3.7. Управление консолью

Системные консоли поддерживаются драйвером Dev.con. Экран, адаптер дисплея и системная клавиатура - вместе называются *консолью*.

Операционная система обеспечивает параллельное выполнение множества сессий на консолях посредством *виртуальных консолей*. Драйвер Dev.con обычно поддерживает более одного набора очередей ввода/вывода к Dev, которые доступны пользовательским процессам как множество *терминальных устройств* с именами вида /dev/con1, /dev/con2 и т.д. С точки зрения приложения, это "настоящие" доступные консоли.

За исключением специальных, поддерживаемых на аппаратном уровне, многомониторных конфигураций у ЭВМ имеется только один *физический* экран и клавиатура, и поэтому только *одна* из этих виртуальных консолей действительно показывается в каждый момент времени. Клавиатура "подключена" к той виртуальной консоли, которая видима в текущий момент.

В дополнение к реализации стандартного терминала, драйвер консоли также обеспечивает набор специфических для консоли функций, которые позволяют приложениям непосредственно взаимодействовать с драйвером консоли путем

обмена сообщениями. Связь устанавливается вызовом функции Си console_open(). После того как связь установлена, процесс имеет доступ к ряду возможностей, представленных в таблице 1.10.

Таблица 1.10. Функции работы с консолью

Необходимая процессу возможность	**Функция Си**
Читать непосредственно с экрана консоли	console_read()
Писать непосредственно на экран консоли	console_write()
Получать асинхронное извещение о важных событиях (например, изменение положения курсора, размера дисплея, переключение консоли и т.д.)	console_arm()
Управлять размерами консоли	console_size()
Переключать видимую консоль	console_active()

Драйвер консоли выполняется как обычный процесс операционной системы. Вводимые с клавиатуры символы помещаются обработчиком прерывания клавиатуры непосредственно в очередь ввода. Выводимые данные отображаются Dev.con в то время, когда он выполняется как процесс.

1.3.8. Управление последовательными устройствами

Последовательные каналы связи обслуживаются драйвером Dev.ser. Этот драйвер может обслуживать более одного физического канала; он обеспечивает терминальные устройства с именами вида /dev/ser1, /dev/ser2 и т.д.

При запуске Dev.ser можно задать аргументы командной строки, которые определяют, какие - и сколько - последовательные порты установлены. Чтобы увидеть доступные последовательные порты, используется утилита ls:

```
ls /dev/ser*
```

Dev.ser является примером управляемого прерываниями сервера ввода/вывода. После инициализации оборудования сам процесс переходит в состояние ожидания ("засыпает"). Прерывания помещают входные данные непосредственно в очередь ввода. Первый выводимый символ передается устройству, когда Dev "подталкивает"

драйвер. Последующие символы передаются при обработке соответствующего прерывания.

1.3.9. Управление параллельными устройствами

Параллельные порты принтера обслуживаются драйвером Dev.par. При запуске Dev.par можно задать аргумент командной строки, который определяет, какой последовательный порт установлен. Чтобы увидеть, доступен ли последовательный порт, используется утилита ls:

```
ls /dev/par*
```

Dev.par является только выводящим драйвером, поэтому у него нет очередей ввода. Можно конфигурировать размер буфера вывода с помощью аргументов командной строки при запуске Dev.par. Буфер вывода может быть сделан очень большим, если необходимо обеспечить программный буфер печати.

Dev.par является примером сервера ввода/вывода, не использующим прерывания. Этот процесс нормально находится в состоянии RECEIVE-блокирован, ожидая появления данных в очереди вывода и "толчка" от Dev. Когда доступны данные для вывода на печать, Dev.par выполняется в цикле работа-ожидание (с относительно низким приоритетом), ожидая, когда символы будут приняты принтером. Такой низкоприоритетный цикл работа-ожидание не влияет на общую производительность системы и, в то же время, достигает максимально возможную пропускную способность к параллельному устройству.

1.3.10. Обеспечение производительности подсистемы устройств

Поток событий внутри подсистемы устройств сконструирован так, что минимизируется избыточность и достигается максимальная пропускная способность, когда устройство работает в «*сыром*» режиме. С этой целью используются следующие правила:

1) обработчики прерываний помещают принятые данные непосредственно в очереди в памяти. Только при наличии ждущего запроса на чтение и при условии, что этот запрос может быть удовлетворен, обработчик прерывания инициирует выполнение Dev. Во всех остальных случаях выполняется просто возврат из

прерывания. Более того, если Dev уже выполняется, то никакой диспетчеризации не производится;

2) когда запрос на чтение удовлетворен, Dev копирует данные в приложение непосредственно из буфера «сырого» ввода. Как результат, данные копируются только один раз.

Эти правила обеспечивают модель ввода, реализованную в изделии.

1.4. Администратор сети

1.4.1. Общие сведения

Администратор сети (Net) предоставляет пользователям изделия прозрачное расширение возможностей механизма передачи сообщений. Взаимодействуя непосредственно с микроядром, Администратор сети усиливает механизм IPC на основе обмена сообщениями, передавая сообщения на удаленные компьютеры. Кроме того, Администратор сети обеспечивает:

1) повышенную пропускную способность за счет балансировки нагрузки;

2) отказоустойчивость за счет резервирования соединений;

3) функции моста между сетями КПДА.00002-01.

1.4.2. Назначение Администратора сети

Администратор сети отвечает за распространение примитивов передачи сообщений, поддерживаемых микроядром, в пределах локальной сети. Стандартные примитивы передачи сообщений используются без изменения для связи с удаленным компьютером. Другими словами, не существует особых "сетевых" Send(), Receive() или Reply().

Использование Администратора сети обладает рядом особенностей.

Администратор сети не должен быть обязательно встроен в образ операционной системы. Он может быть запущен и остановлен в любое время, чтобы обеспечить или удалить сетевые возможности передачи сообщений.

При запуске Администратор сети регистрируется у Администратора процессов и микроядра, что приводит к активизации внутри последних соответствующего кода, взаимодействующего с Администратором сети. Это означает, что передача сообщений по сети и создание удаленных процессов не является слоем, добавляемым поверх операционной системы, а интегрированы в самую сердцевину примитивов передачи сообщений и управления процессами.

Такая глубокая интеграция на самом нижнем уровне обеспечивает изделию сетевую прозрачность и делает его полностью распределенной операционной системой. Поскольку приложения запрашивают и получают доступ ко всем обслуживающим программам посредством сообщений, а Администратор сети

обеспечивает прозрачное прохождение сообщений по сети, то несколько ЭВМ под управлением изделия функционируют вместе как единый логический компьютер.

Поскольку Администратор сети и микроядро отделены друг от друга, микроядро получает независимость от сетевого оборудования, а компьютеры, не подключенные к сети, могут выиграть за счет меньшего объема кода.

1.4.3. Интерфейс между системными модулями

Микроядро и Администратор процессов взаимодействуют с Администратором сети через специальную неблокирующую очередь в памяти. Эта очередь представляет собой список передач, которые должен выполнить Администратор сети. Элементы очереди содержат всю информацию о конкретной операции (например, Send(), Reply(), создание виртуального канала (VC), передача удаленного сигнала и т.д.).

Другим ресурсом операционной системы, используемым для обеспечения прозрачной передачи сообщений, является *буфер виртуального канала*. Выделяемый при создании VC процессом, буфер виртуального канала хранит данные до завершения операции передачи сообщения на другой узел.

Ниже приведены диаграммы, иллюстрирующие процесс посылки и приема удаленных сообщений.

1.4.3.1. Механизм передачи сообщения на удаленный узел

В случае вызова Send() или Reply() к удаленному узлу, имеют место следующие события:

1) процесс вызывает Send() или Reply(), и Микроядро копирует данные из области данных процесса в буфер виртуального канала;

2) микроядро добавляет в очередь Администратора сети элемент, содержащий идентификаторы отправителя, удаленного получателя и указатель на данные в буфере виртуального канала. Если перед этим очередь была пуста, то Администратор сети получает прокси, извещающее о том, что появилась новая работа;

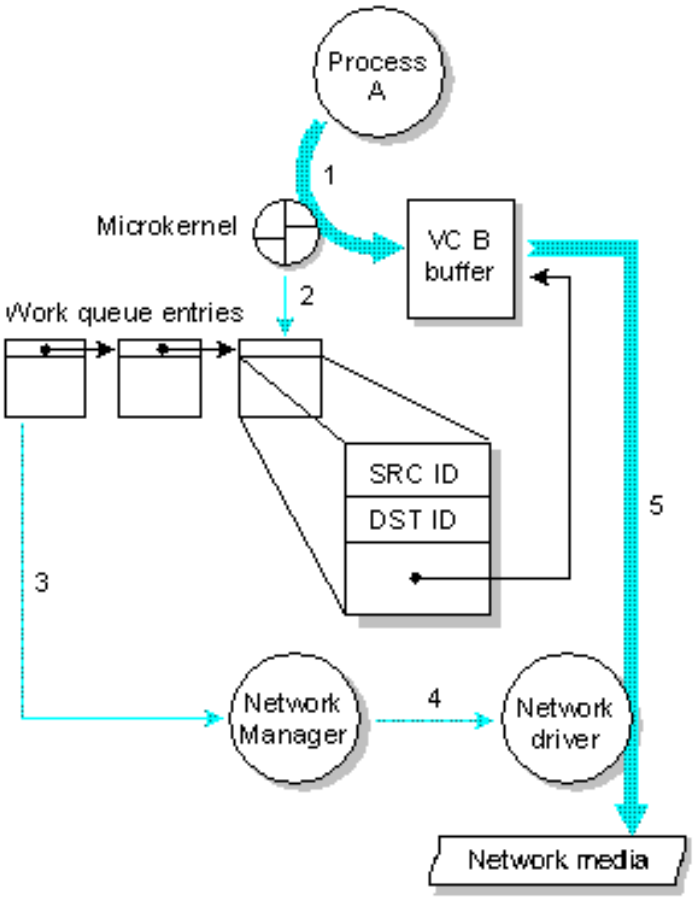


Рис. 19. Процесс вызывает Send() или Reply() к удаленному узлу.

3) администратор сети извлекает элемент из очереди;

4) администратор сети посылает элемент соответствующему сетевому драйверу;

5) администратор сети начинает передачу данных по сети и отвечает за доставку.

В случае распространения сигнала или создания VC, Администратор процессов, а не микроядро, добавляет управляющий пакет в очередь. Как и в предыдущем случае, Администратор сети передаст пакет по назначению.

1.4.3.2. Механизм получения сообщения с удаленного узла

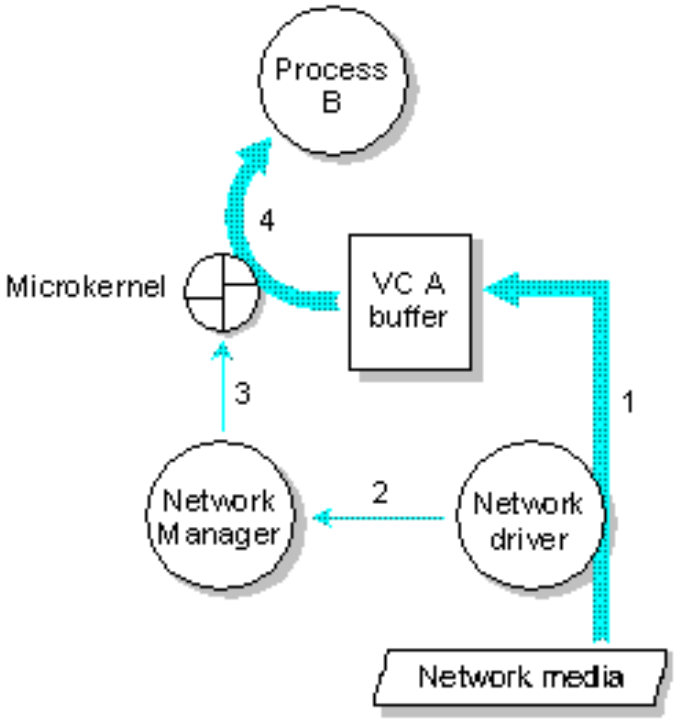


Рис. 20. Процесс получает удаленный Send() или Reply().

Если удаленный узел послал сообщение, как описано выше, то на принимающем узле выполняются следующие события:

1) сетевой драйвер помещает поступившие по сети данные в соответствующий буфер виртуального канала;

2) сетевой драйвер информирует Администратор сети о том, что прием завершен;

3) администратор сети использует частный вызов микроядра, извещая его о том, что прием завершен;

4) микроядро копирует данные из буфера виртуального канала в буфер процесса (при условии, что он RECEIVE- или REPLY-блокирован на этом виртуальном канале).

Любые управляющие пакеты, которые получает Администратор сети, немедленно переправляются Администратору процессов через стандартный примитив Send(). Эти управляющие пакеты используются для распространения сигналов и создания виртуальных каналов.

1.4.4. Сетевые драйверы

Подобно Администратору файловой системы и Администратору устройств, Администратор сети не содержит аппаратно-зависимого кода. Эта функциональность обеспечивается драйверами сетевых плат. Администратор сети может поддерживать одновременно несколько сетевых драйверов. Каждый драйвер обычно обслуживает одну сетевую плату. Можно устанавливать драйверы/платы одного и того же типа или различных типов - например, два драйвера/платы Ethernet или, например, драйвер/плату Ethernet и драйвер/плату Arcnet.

Интерфейс между Администратором сети и драйверами реализован через очереди в разделяемой памяти. Драйвер определяет протокол, подходящий для данного сетевого носителя.

Драйвер отвечает за формирование пакетов, организацию последовательности и в случае, когда требуется гарантированная доставка данных удаленному узлу, повторную передачу. Такая архитектура позволяет изделию легко поддерживать новое сетевое оборудование и протоколы за счет написания или модификации только сетевого драйвера.

1.4.5. Идентификаторы узла и сети

Каждый узел в локальной сети идентифицируется двумя числами:

1) физическим идентификатором (ID) узла (или идентификаторами, если узел имеет более одной сетевой платы);

2) комбинацией логического ID узла и логического ID сети.

1.4.5.1. Физический ID узла

Физический ID узла определяется оборудованием. Сетевые платы обмениваются данными друг с другом, указывая физический ID удаленного узла, с которым должна быть установлена связь. В случае сети Ethernet или Token Ring ID – это большое число, которым неудобно оперировать людям и утилитам. Например, каждая плата Ethernet или Token Ring имеет уникальный 48-битный физический ID узла в соответствии со стандартом IEEE 802. Платы Arcnet, напротив, имеют всего лишь 8-битный ID.

Использование физического ID узла имеет существенный недостаток: при соединении некоторых сетей может возникнуть конфликт адресов (особенно в случае Arcnet), или формат адресов может радикально отличаться.

1.4.5.2. Логический ID узла

Для устранения названных выше проблем, возникающих при использовании физических ID узлов, каждому узлу присваивается *логический ID узла*. Все процессы изделия оперируют логическими ID узлов. Физические ID узлов скрыты от процессов, выполняющихся в изделии.

Логические ID позволяют утилитам, которые опрашивают сеть, использовать простой цикл, в котором логический ID узла изменяется от 1 до количества узлов.

Соответствие между логическими и физическими ID узлов устанавливается Администратором сети. Администратор сети, когда поручает драйверу передать данные на другой узел, передает ему физический ID этого узла.

Логические ID узлов обычно присваиваются по порядку, начиная с 1. Например, узел, имеющий плату Ethernet, может получить логический ID 2, который будет соответствовать физическому ID узла 00:00:c0:46:93:30.

Логические ID узлов должны быть уникальны для всех узлов во *всех* соединенных сетях КПДА.00002-01 для того, чтобы сделать возможным функционирование мостов.

1.4.5.3. Логический ID сети

ID сети идентифицирует конкретную *логическую сеть*. Логическая сеть – это любое оборудование, которое позволяет сетевому драйверу непосредственно взаимодействовать с сетевым драйвером на другом узле (например, последовательный порт или адаптер Ethernet).

На рис. 21 узел 7 имеет две сетевые платы, которые позволяют ему иметь доступ к узлам в логических сетях 1 и 2. Узлы 8 и 9 имеют по три платы, подключающие их к сетям 1, 2 и 3. Все логические ID узлов уникальны для всех трех логических узлов.

Замечание: Логические ID узлов сети назначаются системным администратором.

На рисунке 21 представлена организация несколько физических сетей за счет логических сетей.

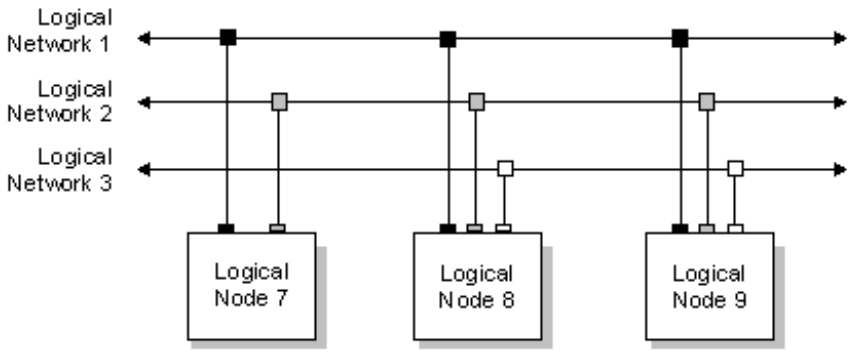


Рис. 21. Пример организации нескольких физических сетей за счет логических сетей.

1.4.6. Порядок выбора сети

В случае, когда узлы соединены более чем одной логической сетью, Администратор сети может выбирать, какую из сетей использовать для передачи к удаленному узлу. Например, на приведенном выше рисунке узел 7 может передавать данному узлу 8, используя либо сеть 1, либо сеть 2. Выбор сети зависит от ряда факторов.

1.4.6.1. Распределение нагрузки

Пропускная способность сети в целом определяется совокупностью скорости компьютера и скорости передачи данных по сети. Если компьютер может выдавать данные быстрее, чем сеть может их передавать, то сеть будет ограничивать пропускную способность.

Например, два компьютера Pentium, соединенные сетью 10BASE-T Ethernet, будут ограничены 1.1 миллионом байт в секунду, то есть скоростью передачи данных, обеспечиваемой сетевым оборудованием. Однако если поместить по две платы Ethernet в каждый из компьютеров и соединить их отдельными кабелями, то Администратор сети сможет передавать данные по обеим сетям одновременно. При большой нагрузке это обеспечит увеличение пропускной способности в два раза по сравнению с одной сетью.

Администратор сети будет пытаться сбалансировать нагрузку, выбирая сетевой драйвер. В рассмотренном выше примере, если производится передача с узла 7 на

узел 8 по сети 1 и другая передача на узел 8 инициируется на узле 7, то сеть 2 будет *автоматически* выбрана для передачи данных.

1.4.6.2. Обеспечение отказоустойчивости

Когда узлы соединены двумя и более сетями, то существует больше чем один возможный путь для связи. В случае отказа платы в одной из сетей, когда связь по этой сети невозможна, Администратор сети автоматически перенаправит все данные через другую сеть. Это происходит без какого-либо вмешательства со стороны прикладных программ и обеспечивает прозрачную отказоустойчивость сети. Если кабели различных сетей проложены раздельно, то компьютеры также защищены от случайного обрыва кабеля.

Это позволяет также проектировать системы - "тандемы", в которых две машины соединены высокоскоростной сетью для нормальной работы и другой, более дешевой и медленной сетью (например, по последовательному каналу), которая служит резервом. В случае отказа первой сети соединение не оборвется, хотя пропускная способность, конечно же, снизится.

1.4.6.3. Организация мостов между сетями КПДА.00002-01

Администратор сети позволяет любому узлу играть роль моста между двумя отдельными сетями КПДА.00002-01, базирующимися на стандарте IEEE 802.

Так как КПДА.00002-01 использует одинаковый формат пакетов и протокол на всех IEEE 802 сетях, то с помощью изделия можно создавать мосты между сетями Ethernet, Token Ring и FDDI.

Замечание: для сетей Arcnet нельзя создавать мосты.

Рассмотрим следующую диаграмму, где одной сети принадлежат узлы 17 и 18, а другой - узлы 18 и 19.

Узлы 17 и 18 находятся в одной сети, поэтому они могут общаться друг с другом напрямую. То же справедливо для узлов 18 и 19. Но как могут общаться узлы 17 и 19?

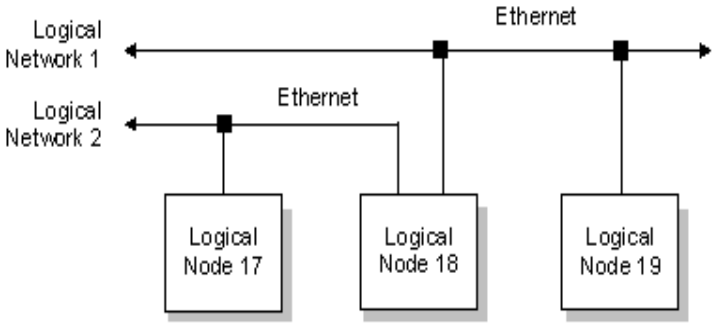


Рис. 22. Организация моста между двумя IEEE 802 сетями КПДА.00002-01.

Так как *обе* локальные сети базируются на IEEE 802, узел 18 автоматически перенаправляет пакеты, позволяя узлам 17 и 18 создать виртуальный канал. Хотя они и не подключены к одной и той же локальной сети, узлы 17 и 19, тем не менее, могут общаться друг с другом.

1.5. Администратор файловой системы

1.5.1. Общие сведения

Администратор файловой системы (Fsys) обеспечивает стандартизованные способы сохранения данных на дисках и доступа к ним. Fsys отвечает за обработку всех запросов на открытие, закрытие, чтение и запись файлов.

1.5.1.1. Характеристики файла

В изделии *файл* - это объект, в который может производиться запись, из которого может производиться чтение, либо и то и другое. Fsys поддерживает следующие типы файлов:

- регулярные файлы - состоят из последовательности байт с произвольным доступом и не имеют предопределенной структуры;

- каталоги - содержат информацию, необходимую для поиска регулярных файлов; также содержат статус и атрибуты для каждого регулярного файла;

- символические связи - содержат путь к файлу или каталогу, к которым перенаправляются все запросы; символические связи часто используются для предоставления множества путей к одному файлу;

- программные каналы (pipes) и FIFO - служат как каналы ввода/вывода между взаимодействующими процессами;

- блок-ориентированные файлы - относятся к устройствам, таким, как диски, ленты и разделы дисков. Доступ к этим файлам обычно осуществляется таким образом, что аппаратные характеристики устройств скрыты от приложений.

Блок-ориентированные файлы обслуживаются Администратором устройств.

1.5.1.2. Метки даты и времени

Fsys поддерживает четыре различных метки времени для каждого файла. Это:

- дата последнего доступа (чтения);

- дата последней записи;

- дата последней модификации;

- дата создания (уникальна для КПДА.00002-01).

1.5.1.3. Организация доступа к файлам

Доступ к регулярным файлам и каталогам управляется битами режима, хранящимися в *inode* (индексном дескрипторе) файла. Более подробно inode описан в секции «Связи и индексные дескрипторы (inodes)». Эти биты разрешают чтение, запись и выполнение в зависимости от ID пользователя и группы. При этом пользователи делятся на три категории:

- владелец файла;
- члены группы, к которой принадлежит владелец;
- остальные.

Процесс может выполняться с ID пользователя или ID группы файла, а не родительского процесса. Механизм, который позволяет это, называется *setuid* (установить ID пользователя) и *setgid* (установить ID группы).

1.5.2. Регулярные файлы и каталоги

1.5.2.1. Регулярные файлы

В изделии *регулярный файл* рассматривается как последовательность байт с возможностью произвольного доступа и не имеющую другой предопределенной внутренней структуры. Прикладные программы сами несут ответственность за понимание структуры и содержания конкретного регулярного файла.

Регулярные файлы составляют большинство файлов в *файловых системах*. Файловые системы поддерживаются Администратором файловой системы и реализованы на базе блок-ориентированных файлов, соответствующих разделам диска (разделы описаны в подразделе «Работа с дисками»).

1.5.2.2. Каталоги

Каталог - это файл, который содержит *элементы каталога*. Каждый элемент каталога увязывает *имя файла* с файлом. Имя файла - это символьное имя, которое позволяет идентифицировать файл и работать с ним. Файл может быть идентифицирован несколькими именами (см. подразделы «Ссылки и индексные дескрипторы (inodes)» и «Символические ссылки»).

Следующая диаграмма показывает, как производится поиск файла с именем /usr/bill/file2.

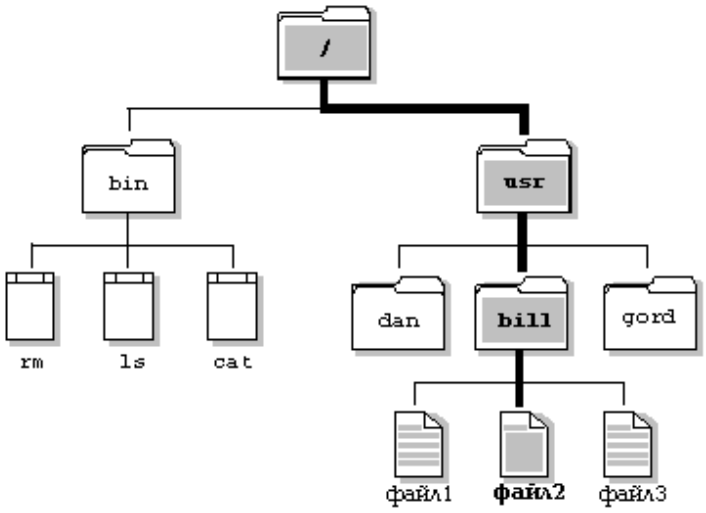


Рис. 23. Путь в структуре каталога КПДА.00002-01 к файлу /usr/bill/file2.

Операции с каталогами

Каталог ведет себя во многом как стандартный файл,

Администратор файловой системы накладывает некоторые ограничения на операции, которые можно производить с каталогом. В частности, нельзя открыть каталог для записи либо создать связь для каталога с помощью функции Си link().

Чтение элементов каталога

Для чтения элементов каталога можно использовать набор функций Си, определенных POSIX, которые обеспечивают независимый от ОС доступ к элементам каталога. Эти функции включают:

opendir()

readdir()

rewinddir()

closedir()

Так как каталоги в изделии – это просто файлы, содержащие "известную" информацию, можно также читать элементы каталога непосредственно функциями Си open() и read(). Однако эта техника не переносима - формат элементов каталога отличается в различных операционных системах.

1.5.2.3. Экстенты

Регулярные файлы и файлы каталога хранятся как последовательность *экстентов*. Экстент – это непрерывная последовательность блоков на диске.

Где хранятся экстенты

Файлы, которые состоят только из одного экстента, хранят информацию об экстенте в элементе каталога. Но, если файл состоит более чем из одного экстента, информация о расположении экстентов хранится в одном или более *связных блоках экстентов* (связные - имеющие прямые/обратные указатели). Каждый блок экстентов может содержать информацию не более чем о 60 экстентах (схему взаимосвязи записей каталога, экстентов и блок-экстентов см. на рис. 24).

Увеличение файлов

Когда Администратору файловой системы необходимо увеличить файл, он сначала пытается увеличить последний экстент, хотя бы даже на один блок. Но если последний экстент не может быть дополнен, то для расширения файла выделяется новый экстент.

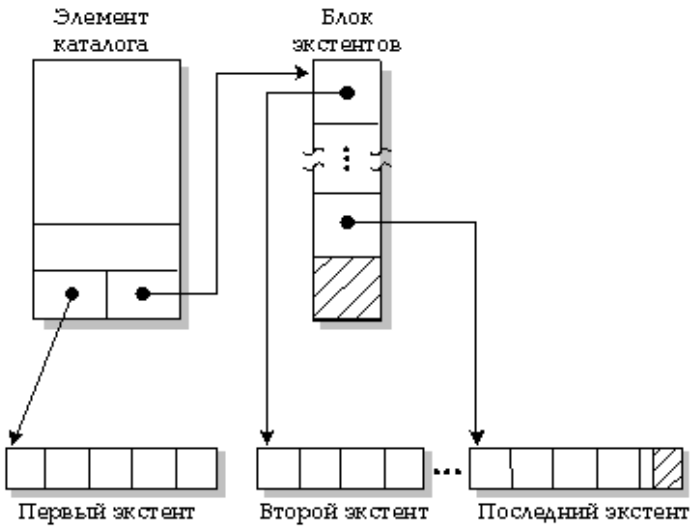


Рис. 24. Файл, состоящий из множества непрерывных областей на диске, называемых в изделии экстентами.

Для выделения новых экстентов Администратор файловой системы использует метод "первого попадания". Специальная таблица в Администраторе файловой системы содержит сведения обо всех блоках, описанных в файле /.bitmap (этот файл описан в подразделе «Ключевые компоненты раздела КПДА.00002-01»). Для

каждого блока указывается размер соответствующего ему свободного экстента. Администратор файловой системы выбирает из таблицы первый достаточно большой экстент.

1.5.3. Ссылки и индексные дескрипторы (inodes)

Файл в изделии может обозначаться более чем одним именем. Каждое имя файла называется *ссылкой*. В действительности существует два вида ссылок: жесткие ссылки, или просто "ссылки", и символические ссылки. Символические ссылки описаны в следующем подразделе.

Для поддержки ссылок каждого файла, имя файла отделяется от остальной информации, описывающей файл. Эта информация хранится в структуре, называемой *inode* (индексным дескриптором).

Если файл имеет только одну ссылку (т.е. одно имя), то блок inode хранится в элементе каталога для этого файла. Но если файл имеет более чем одну ссылку, то inode хранится как запись в специальном файле /.inodes, а элемент каталога для файла содержит указатель на запись inode.

Следует учитывать, что можно создать ссылку для файла, только если файл и связь находятся в одной и той же файловой системе.

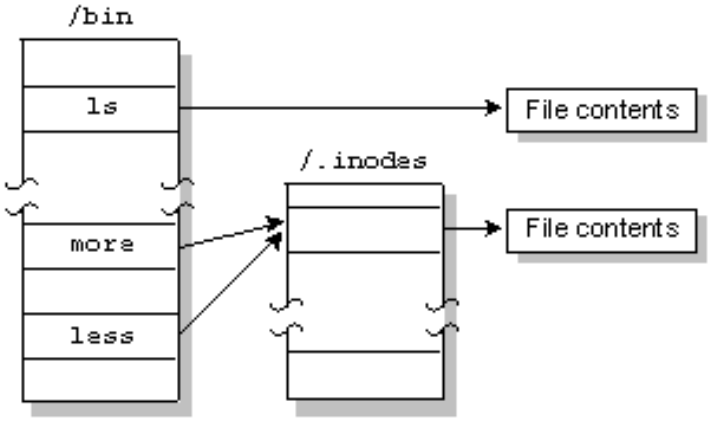


Рис. 25. Один и тот же файл обозначен двумя ссылками с именами "more" и "less".

Существует еще две ситуации, в которых для файла создается запись в файле /.inodes:

1) если имя файла длиннее чем 16 символов, то информация inode хранится в файле /.inodes, оставляя место в элементе каталога для 48-символьного имени файла;

2) если файл имел более одной связи и все связи, кроме одной, были удалены, то за файлом сохраняется отдельная запись в файле /.inodes. Это сделано, чтобы избежать поиска элемента каталога, указывающего на inode (элементы inode не имеют обратных связей с элементами каталога).

Для создания жесткой ссылки из программы используется функция ln(), а для создания жесткой ссылки из командного интерпретатора – утилита ln.

Удаление ссылок

При создании файла, для него устанавливается *счетчик ссылок*, равный единице. По мере добавления ссылок этот счетчик увеличивается; при удалении ссылки счетчик ссылок уменьшается. Файл не удаляется с диска до того, как счетчик ссылок станет равным нулю *и* все программы, использующие этот файл, закроют его. Это позволяет использовать открытый файл даже после того, как у него удалены все ссылки.

Для удаления жесткой ссылки из программы используется функция remove() или функция unlink(), а для удаления жесткой ссылки из командного интерпретатора – утилита rm.

Ссылки каталога

Нельзя создавать жесткие ссылки для каталога. Однако каждый каталог имеет две жестко определенные ссылки:

. ("точка")

.. ("точка - точка")

Имя файла "точка" соответствует текущему каталогу; "точка - точка" соответствует каталогу, *предшествующему* текущему каталогу.

Следует заметить, что "точка - точка" для каталога "/" - это просто "/", - невозможно подняться выше.

1.5.4. Символические ссылки

Символическая ссылка – это особый файл, который содержит в качестве данных имя пути. Когда символическая ссылка используется в запросе ввода/вывода - например, open(), - обозначение ссылки в имени пути заменяется ее "данными". Символическая ссылка является гибким средством для перенаправления пути и часто используется для создания множества путей к одному и тому же файлу. В отличие от жестких ссылок, символические ссылки могут выходить за пределы файловой системы и также являться ссылками для каталогов.

В следующем примере каталоги //1/usr/fred и //2/usr/barney являются ссылками на один и тот же каталог, хотя они находятся в различных файловых системах, и даже на различных узлах. Это не может быть сделано с использованием жестких связей:

//1/usr/fred --> //2/usr/barney

Заметьте, что символическая ссылка и адресуемый каталог не обязаны иметь одно и то же имя. В большинстве случаев символические ссылки используются для привязки одного каталога к другому. Однако они также могут быть использованы для файлов, как в этом примере:

//1/usr/eric/src/test.c --> //1/usr/src/game.c

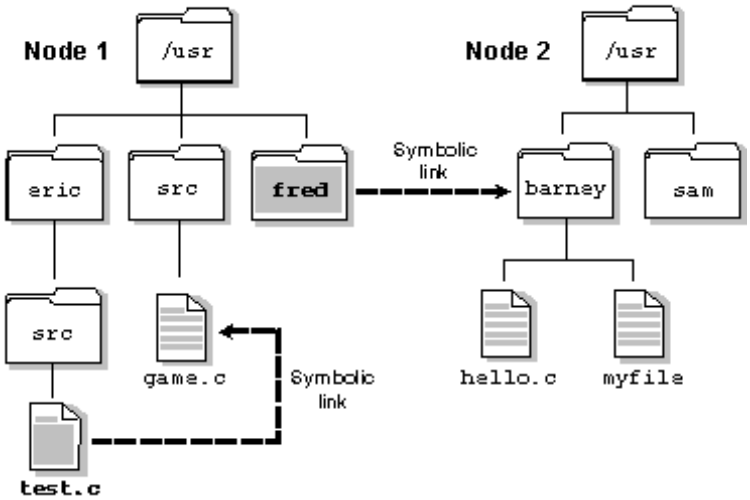


Рис. 26. Ссылки каталогов

Некоторые функции оперируют непосредственно с символическими ссылками. Для этих функций замена обозначения связи в пути на ее содержимое не

производится. К этим функциям относятся unlink() (которая удаляет символическую ссылку), lstat() и readlink().

Создается символическая ссылка с помощью утилиты ln, вызванной с опцией «-s». Удаляется символическая ссылка с помощью утилиты rm. Выяснить, является ли файл символической ссылкой, можно, используя утилиту ls с опцией «-l».

Примечание. Удаление символической ссылки действует только на ссылку, а не на адресуемый объект

Так как символические ссылки могут указывать на каталоги, то неверная конфигурация может привести к проблемам, таким, как циклические ссылки. Чтобы защититься от циклических ссылок, система накладывает ограничения на количество переходов; этот предел определен как SYMLOOP_MAX в заголовочном файле <limits.h>.

1.5.5. Программные каналы (pipes) и FIFO

1.5.5.1. Программные каналы (pipes)

Программный канал - это неименованный файл, который служит как канал ввода/вывода между двумя или более взаимодействующими процессами - один процесс пишет в программный канал, другой читает из программного канала. Администратор файловой системы обеспечивает буферизацию данных. Размер буфера определен как PIPE_BUF в файле <limits.h>. Программный канал удаляется после того, как закрыты оба его конца.

Программные каналы обычно используются, когда два процесса хотят выполняться параллельно, с однонаправленной передачей данных от одного процесса к другому. Если требуется двунаправленная передача данных, то должны использоваться сообщения.

Типичное применение программного канала состоит в соединении вывода одной программы с вводом другой программы. Это соединение часто производится командным интерпретатором (Shell).

Например, следующая команда:

ls | more

направляет стандартный вывод от утилиты ls через программный канал в стандартный ввод утилиты more.

Для создания программного канала из командного интерпретатора используется символ «|», а для создания программного канала непосредственно из приложения – функция pipe() или функция popen().

Замечание: На бездисковых рабочих станциях можно запустить Администратор программных каналов (Pipe) вместо Администратора файловой системы, когда требуются только программные каналы. Администратор программных каналов оптимизирован для канального (конвейерного) ввода/вывода и может обеспечить большую пропускную способность, чем Администратор файловой системы.

1.5.6. Файлы FIFO

Файлы FIFO - это по существу то же самое, что и программные каналы, за исключением того, что FIFO являются именованными постоянными файлами, которые хранятся в каталогах файловой системы.

В командном интерпретаторе для создания FIFO используется утилита mkfifo, для удаления – rm. В приложении для создания FIFO используется функция mkfifo(), для удаления – функция remove() или unlink().

1.5.7. Обеспечение производительности файловой системы

Администратор файловой системы обладает следующими свойствами, повышающими производительность доступа к диску:

- лифтовый поиск;
- кэш-буфер;
- многопотоковая обработка;
- клиент-управляемый приоритет;
- временные файлы;
- псевдодиски.

1.5.7.1. Лифтовый поиск

Лифтовый поиск минимизирует общие затраты времени на позиционирование магнитной головки при чтении или записи на диск. Запросы ввода/вывода

упорядочиваются таким образом, чтобы все они могли быть выполнены за один проход магнитной головки, от самого младшего к самому старшему адресу на диске.

Лифтовый поиск также имеет усовершенствование, обеспечивающее выполнение мультисекторного ввода/вывода там, где это возможно.

1.5.7.2. Кэш-буфер

Кэш-буфер – это интеллектуальный буфер между Администратором файловой системы и драйвером диска. Кэш-буфер хранит блоки файлов с целью минимизации количества обращений Администратора файловой системы к диску. По умолчанию размер кэша определяется общим объемом оперативной памяти, но можно задать другое значение через опцию при запуске Fsys.

Операции чтения являются синхронными. Операции записи, напротив, обычно являются асинхронными. Когда данные попадают в кэш, Администратор файловой системы отвечает клиенту (функцией Reply()), извещая его, что данные записаны. Затем выполняется запись данных на диск с максимально-возможной скоростью, обычно не позднее, чем через пять секунд.

Приложения могут изменять механизм записи применительно к конкретным файлам. Например, приложение, работающее с базой данных, может потребовать, чтобы запись в определенный файл производилась синхронно. Это обеспечит высокий уровень целостности файла в случае аппаратного или программного сбоя.

1.5.7.3. Многопотоковая обработка

Администратор файловой системы является многопотоковым процессом, и, таким образом, он может обрабатывать несколько запросов ввода/вывода одновременно. Это позволяет Администратору файловой системы в полной мере реализовать возможности параллельной обработки, так как он в состоянии:

- получить одновременный доступ к нескольким устройствам;
- удовлетворить запросы ввода/вывода из кэш-буфера, в то время как выполняются другие запросы ввода/вывода, осуществляющие доступ к диску.

1.5.7.4. Клиент-управляемый приоритет

Приоритет Администратора файловой системы может определяться приоритетом процесса, пославшего ему запрос. В этом случае, когда Администратор

файловой системы получает сообщение, его приоритет устанавливается равным приоритету процесса, пославшего сообщение. Более подробно об этом изложено в подразделе «Диспетчеризация процессов» раздела "Микроядро".

1.5.7.5. Временные файлы

В изделии предусмотрена возможность открытия временных файлов, т.е. файлов, которые записываются и затем читаются в течение короткого промежутка времени. Для таких файлов Администратор файловой системы старается хранить блоки данных в кэш-буфере и производит запись блоков на диск только в случае крайней необходимости.

1.5.7.6. Псевдодиски

Администратор файловой системы содержит встроенную возможность поддержки электронного диска, позволяющую использовать до 8 Мегабайт оперативной памяти в качестве псевдодиска. Так как Администратор файловой системы использует механизм составных сообщений, то данные из псевдодиска копируются непосредственно в приложение.

Администратор файловой системы в состоянии обойти кэширование, так как псевдодиск является встроенным, а не реализован как отдельный драйвер.

Так как псевдодиски исключают аппаратные задержки и не используют кэширование данных, поэтому они обеспечивают больший детерминизм при выполнении операций ввода/вывода по сравнению с жесткими дисками.

1.5.8. Обеспечение надежности файловой системы

Высокая производительность файловой системы обеспечивается несколькими способами.

В то время как большая часть данных помещается в кэш-буфер и записывается с небольшой задержкой, критические для файловой системы данные записываются немедленно. Обновления каталогов, индексных дескрипторов (inodes), блоков экстентов и битовой карты производятся без задержки для того, чтобы гарантировать целостность структуры файловой системы на диске (т.е. исключить внутреннее несоответствие данных на диске).

Иногда все перечисленные выше структуры данных должны быть обновлены. Например, если перемещается файл в каталог, последний экстент которого заполнен, то каталог должен вырасти. В таких случаях порядок операций подобран таким образом, что даже если произойдет катастрофический сбой в момент, когда операция еще не полностью завершена (например, отключение питания), то файловая система после перезапуска все же сохранит работоспособность. В худшем случае, некоторые блоки будут выделены, но не использованы. Исправить подобную ситуацию можно, запустив утилиту chkfsys.

1.5.8.1. Восстановление файловой системы

Даже самые надежные системы не застрахованы от аварийных ситуаций, таких как:

- появление плохих блоков на диске в результате бросков или провалов напряжения;

- неумелый или злонамеренный пользователь, имеющий доступ к привилегиям администратора системы, выполнил инициализацию файловой системы (утилитой dinit);

- некорректная программа (в особенности, выполняющаяся не в среде КПДА.00002-01) игнорирует информацию о разделах диска и перезаписывает часть раздела КПДА.00002-01.

Чтобы после таких ситуаций можно было восстановить как можно больше файлов, на диск записываются уникальные "сигнатуры" для автоматической идентификации и восстановления критических частей файловой системы. Файл с индексными дескрипторами (/.inodes), так же как и каждый каталог, и блок экстентов, все содержат уникальные образцы данных, которые могут быть использованы утилитой chkfsys для восстановления серьезно поврежденной файловой системы.

Более подробная информация о восстановлении файловой системы содержится в описании утилиты chkfsys (Описание программы, часть 2, КПДА.00002-01 13 01-2).

1.5.9. Работа с дисками

Администратор файловой системы управляет *блок*-ориентированными файлами. Эти файлы определяют диски и разделы дисков.

1.5.9.1. Диски и дисковые подсистемы

В изделии каждый физический диск на компьютере считается блок-ориентированным файлом. Блок-ориентированный файл диск рассматривается файловой системой изделия как последовательность блоков размером по 512 байт, независимо от размера диска. Блоки нумеруются, начиная с первого блока на диске (блок 1).

Так как каждый диск является блок-ориентированным файлом, он может быть открыт как одно целое для низкоуровневого доступа с использованием стандартных POSIX Си функций, таких как open(), read(), write() и close(). На уровне блок-ориентированного файла, который определяет целый диск, изделие не делает никаких предположений о каких-либо структурах данных, которые могут существовать на диске.

ЭВМ под управлением данной ОСРВ может иметь одну или несколько дисковых подсистем. Каждая дисковая подсистема состоит из контроллера и одного или более дисков. При запуске драйвера устройства должна запускаться своя дисковая подсистема, управляемая Администратором файловой системы.

1.5.9.2. Разделы диска

Изделие соответствует промышленному стандарту де-факто, который позволяет различным операционным системам разделять один и тот же физический диск. В соответствии с этим стандартом, таблица разделов может определять до четырех первичных разделов на диске. Таблица хранится в первом блоке диска.

Каждый раздел должен иметь "тип", узнаваемый операционной системой, которая должна использовать этот раздел. Список, представленный в таблице 1.11, содержит используемые типы разделов.

Таблица 1.11. Типы разделов диска.

Тип	**Раздел**

Тип	Раздел
1	DOS (12-битная FAT)
4	DOS (16-битная FAT; разделы <32Мбайт)
5	Расширенный раздел DOS
6	DOS 4.0 (16-битная FAT; раздел >=32Мбайт)
7	OS/2 HPFS
11	DOS 32-битная FAT; разделы до 2047 Гбайт
12	То же, что тип 11, но использует LBA расширения прерывания Int 13h
14	То же, что тип 6, но использует LBA расширения прерывания Int 13h
15	То же, что тип 5, но использует LBA расширения прерывания Int 13h
77	POSIX раздел КПДА.00002-01
78	POSIX раздел КПДА.00002-01 (вторичный)
79	POSIX раздел КПДА.00002-01 (вторичный)
99	UNIX
131	Ext2 (Linux)

Если необходимо иметь несколько разделов КПДА.00002-01 на одном физическом диске, следует использовать тип 77 для первого раздела КПДА.00002-01, тип 78 для второго раздела КПДА.00002-01, и тип 79 для третьего. Можно использовать другие типы для второго и третьего разделов КПДА.00002-01, но 78 и 79 предпочтительнее. Чтобы пометить любой из этих разделов как загрузочный, следует использовать утилиту fdisk.

Во время загрузки, загрузчик изделия (опционально устанавливаемый утилитой fdisk) позволяет выбирать в таблице разделов в качестве загрузочного другой раздел, не являющийся загрузочным по умолчанию.

Можно использовать утилиту fdisk для создания, модификации или удаления разделов.

Так как изделие рассматривает каждый раздел на диске как блок-ориентированный файл, то это дает возможность доступа к следующим элементам:

- к диску в целом, игнорируя разделы, как к блок-ориентированному файлу;

- к отдельному разделу как к блок-ориентированному файлу; этот блок-ориентированный файл будет подмножеством блок-ориентированного файла, который определяет весь диск.

1.5.9.3. Определение блок-ориентированных файлов

Имена всех блок-ориентированных файлов помещаются в дерево префиксов того компьютера, на котором расположены блок-ориентированные файлы (дерево префиксов рассматривается в главе "Пространство имен ввода/вывода"). Когда запускается драйвер контроллера дисков, Администратор файловой системы автоматически регистрирует префиксы, которые определяют блок-ориентированный файл для каждого физического диска, подключенного к контроллеру. Утилита mount используется для того, чтобы зарегистрировать префикс для блок-ориентированного файла для каждого раздела на диске.

Пусть, например, имеется стандартный контроллер Western Digital, к которому подключены два диска. На одном диске необходимо смонтировать раздел DOS, раздел КПДА.00002-01 и раздел UNIX. На другом диске - раздел DOS и раздел КПДА.00002-01.

Администратор файловой системы определит блок-ориентированные файлы /dev/hd0 и /dev/hd1 для двух дисков, подключенных к контроллеру.

Затем, используя утилиту mount, можно определить блок-ориентированный файл для каждого раздела.

Например:

```
mount -p /dev/hd0 -p /dev/hd1
```

определит блок-ориентированные файлы, представленные в таблице 1.12

Таблица 1.12. Пример имен блок-ориентированных файлов

Раздел ОС	**Блок-ориентированный файл**

Раздел ОС	Блок-ориентированный файл
Раздел DOS на диске hd0	/dev/hd0t4
Раздел КПДА.00002-01 на диске hd0	/dev/hd0t77
Раздел UNIX на диске hd0	/dev/hd0t99
Раздел DOS на диске hd1	/dev/hd1t4
Раздел КПДА.00002-01 на диске hd1	/dev/hd1t77

Замечания.

1. Обозначение tn используется для обозначения разделов на диске, используемых определенными операционными системами. Например, раздел DOS обозначается t4, раздел FAT32 – это t11 и т.д.

2. Обозначение может содержать еще и номер контроллера (например, /dev/hd0.0t77), что определяется аргументами, заданными при запуске драйвера Fsys.eide (см. «Описание программы» КПДА.00002-01 13 01-2).

2.5.9.4. Монтирование файловой системы

Обычно файловая система КПДА.00002-01 монтируется на блок-ориентированном файле. Для этого используется утилита mount - она позволяет задать префикс, который идентифицирует файловую систему.

Например:

```
mount /dev/hd0t77 /
```

монтирует файловую систему с префиксом / на разделе, который определен блок-ориентированным файлом с именем hd0t77.

Замечание: Если диск разбит на разделы, то следует смонтировать блок-ориентированный файл для раздела КПДА.00002-01 (например /dev/hd0t77), а не основной блок-ориентированный файл, который определяет весь диск (например, /dev/hd0). Если попытаться смонтировать основной блок-ориентированный файл для всего диска, то при попытке доступа к файловой системе выдается сообщение "corrupt filesystem" (поврежденная файловая система).

1.5.9.5. Демонтирование файловой системы

Для демонтажа файловой системы, используется утилита umount. Так, например, следующая команда демонтирует файловую систему на первичном разделе КПДА.00002-01:

```
umount /dev/hd0t77
```

После того, как файловая система демонтирована, файлы в этом разделе уже не доступны пользователю.

1.5.10. Ключевые компоненты раздела КПДА.00002-01

В начале каждого раздела КПДА.00002-01 располагаются следующие ключевые компоненты:

- блок загрузчика;
- корневой блок;
- битовая карта;
- корневой каталог.

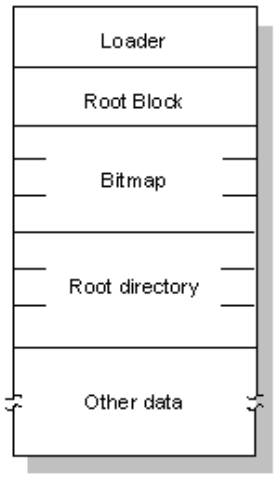


Рис. 27. Структура файловой системы КПДА.00002-01 внутри раздела диска.

Эти структуры создаются при инициализации файловой системы утилитой dinit.

1.5.10.1. Блок загрузчика

Это первый физический блок раздела диска. Этот блок содержит код, который загружается и затем исполняется BIOS компьютера для загрузки операционной системы из раздела. Если диск не разбит на разделы (например, гибкий диск), этот блок является первым физическим блоком на диске.

1.5.10.2 Корневой блок

Корневой блок имеет ту же структуру, что и обычный каталог. Он содержит информацию о четырех особых файлах:

- *корневой каталог* файловой системы (/)
- файл /.inodes
- файл /.boot
- файл /.altboot

Файлы /.boot и /.altboot содержат образы операционной системы, которые загружаются программой начальной загрузки КПДА.00002-01.

Обычно загрузчик изделия загружает образ ОС, хранящийся в файле /.boot. Но если файл /.altboot не пуст, то будет предложена опция загрузки образа, хранящегося в файле /.altboot.

1.5.10.3. Битовая карта

Для распределения пространства на диске КПДА.00002-01 использует *битовую карту*, хранящуюся в файле /.bitmap. Этот файл содержит битовую маску для всех блоков на диске, показывающую, какие блоки заняты. Каждому блоку соответствует один бит. Если значение бита 1, то соответствующий блок на диске занят.

1.5.10.4. Корневой каталог

Корневой каталог раздела ведет себя как обычный каталог, за двумя исключениями:

как ".", так и ".." являются связями к одной и той же inode информации, а именно, корневым каталогом inode в корневом блоке;

корневой каталог всегда содержит элементы для файлов /.bitmap, /.inodes, /.boot и /.altboot. Эти элементы предусмотрены для того, чтобы программы, которые выдают информацию о файловой системе, видели эти элементы как обычные файлы.

1.5.11. Администратор файловой системы DOS

Пространство имен ввода/вывода управляется через префиксы, которые направляют запросы соответствующим процессам-администраторам (подробнее см. п.п. 1.6). Одним из таких процессов является Администратор файловой системы DOS (Dosfsys). Dosfsys регистрирует префикс /dos и представляет диски DOS

внутри пространства имен Администратора процессов как "гостевые" файловые системы.

Dosfsys обеспечивает прозрачный доступ к дискам DOS, так что можно рассматривать файловые системы DOS как файловые системы КПДА.00002-01. Такая прозрачность позволяет процессам работать с файлами DOS без каких-либо специальных знаний или действий. Стандартные библиотечные функции ввода/вывода, такие как open(), close(), read() и write() работают с файлом в разделе DOS точно так же, как и с файлом в разделе КПДА.00002-01.

Например, чтобы копировать файл из раздела КПДА.00002-01 в раздел DOS, достаточно выполнить команду:

```
cp /usr/luc/file.dat /dos/c/file.dat
```

/dos/c - это путь к DOS диску C. Команда cp не содержит какого-либо специального кода для определения, находится ли копируемый файл на диске DOS. Другие команды работают с такой же прозрачностью (например, утилиты cd, ls и mkdir).

Если не существует эквивалента DOS для функции КПДА.00002-01, такой как mkfifo() или link(), то Dosfsys возвращает соответствующий код ошибки (т.е. errno).

Dosfsys работает как с гибкими дисками, так и с разделами жестких дисков. Весь требуемый доступ к диску на низком уровне производится через стандартные функции Администратора файловой системы. Поэтому, без доступа к коду низкого уровня, Dosfsys способен обеспечить прозрачный интерфейс между приложениями КПДА.00002-01 и файловой системой DOS.

1.5.12. Файловая система CD-ROM

Файловая система CD-ROM, Iso9660fsys, обеспечивает прозрачный доступ к носителям CD-ROM, таким образом можно работать с файловыми системами CD-ROM, как будто это файловые системы POSIX. Такая прозрачность обеспечивает процессам доступ к файлам на CD-ROM стандартными средствами.

Администратор Iso9660fsys реализует стандарт ISO 9660, включая расширения Rock Ridge. Этому стандарту соответствуют компакт-диски DOS и Windows. В дополнение к обычным файлам, Iso9660fsys также поддерживает аудио.

1.5.13. Файловая система флэш

Администратор файловой системы флэш Efsys.* был специально разработан для работы, как со встроенной, так и со сменной флэш-памятью. Файлы, записанные на сменные носители флэш (карты PC-Card), переносимы в другие системы, которые также поддерживают этот стандарт.

Администратор Efsys.* сочетает функции администратора файловой системы и драйвера устройства. Так как Efsys.* содержит драйвер устройства, то существуют отдельные версии Efsys.* для различных видов оборудования встраиваемых систем. Например:

`Efsys.explr2` для платы `Intel EXPLR2 (c RFA);`

`Efsys.elansc400` для платы `AMD Elan SC400;`

Efsys.pcmcia для смешанного использования файловых систем флэш и устройств PCMCIA.

1.5.13.1. Ограничения

Функциональность файловой системы ограничена используемыми устройствами памяти. Например, для устройств ROM файловая система доступна только для чтения.

Для устройств флэш-памяти существуют ограничения на запись файлов. Можно только дозаписывать файл. Кроме того, не обновляется время последнего доступа к файлу. В настоящий момент эти ограничения распространяются даже на SRAM устройства.

1.5.13.2. Восстановление свободного пространства

Администратор Efsys.* хранит каталоги и файлы, используя связные (связные - имеющие прямые/обратные указатели) списки структур данных, а не блоки фиксированного размера, как на диске, используемые в традиционных файловых системах с вращающимся носителем. При удалении каталога или файла, принадлежащие ему структуры данных помечаются как удаленные, но не стираются, чтобы избежать непрерывного стирания и перезаписи (и тем самым потерь времени на эти операции).

Со временем свободное место закончится, и Администратору файловой системы придется выполнить восстановление свободного пространства (иногда эту операцию называют еще "уборка мусора"). Во время этой процедуры Efsys.* восстанавливает свободное место, занимаемое удаленными файлами и каталогами. Для проведения этой операции Администратору файловой системы требуется хотя бы один свободный блок. Утилита mkffs автоматически резервирует для этой цели один блок при создании файловой системы.

1.5.13.3. Сжатие и распаковка

Администратор Efsys.* поддерживает распаковку, что увеличивает объем данных, который может храниться на носителе. Распаковка выполняется Администратором файловой системы для приложений прозрачно.

Для этого файлы должны быть перед запуском утилиты mkffs предварительно сжаты с помощью утилиты bpe. Если же копировать сжатый файл в уже созданную файловую систему флэш, то он останется сжатым и при чтении.

1.5.13.4. Доступ к файлам

Если запретить расширения POSIX, то владельцем файлов всегда будет считаться root, а биты режима всегда будут установлены в rwx. Команды chgrp, chmod и chown в этом случае не будут работать.

1.5.13.5. Монтирование файловой системы флэш

Монтирование файловой системы флэш может производиться только при инициализации разделов или при запуске Администратора файловой системы.

1.5.13.6. Доступ на низком уровне

При запуске Efsys.* он создает для каждого устройства памяти специальный файл в каталоге /dev. В системе с двумя устройствами памяти Efsys.* создаст файлы /dev/skt1 и /dev/skt2. Специальные устройства игнорируют разбиение на разделы, позволяя доступ к носителям на низком уровне.

Доступ к разделу, содержащему образ файловой системы, возможен только на низком уровне (как к "сырому" устройству). Для каждого такого раздела Efsys.* создает специальный файл вида /dev/sktXimgY, где X - это номер гнезда (socket), а Y - номер раздела на этом носителе.

1.6. Пространство имен ввода/вывода

1.6.1. Общие сведения

1.6.1.1. Пространство имен ввода/вывода

Механизмы ввода/вывода не встроены внутрь микроядра. Процессы, обслуживающие ввод/вывод, запускаются динамически во время работы системы. Так как файловая система является необязательным элементом изделия, то пространство путей (полных имен файлов и каталогов) не встроено внутрь файловой системы.

1.6.1.2 Префиксы и области полномочий

Пространство имен путей разделено на области полномочий. Любой процесс, выполняющий файл - ориентированное обслуживание ввода/вывода, должен зарегистрировать у Администратора процессов свой префикс, определяя часть пространства имен, которое он хочет контролировать (т.е. область своих полномочий). Эти префиксы составляют дерево префиксов, которое хранится в памяти на каждом компьютере под управлением изделия.

1.6.2. Разборка имен путей

1.6.2.1. Префиксы администратора ввода/вывода

Когда процесс открывает файл, то полное имя файла сопоставляется с деревом префиксов с тем, чтобы отправить запрос open() к соответствующему администратору ресурсов ввода/вывода. Например, Администратор устройств (Dev) обычно регистрирует префикс /dev. Если процесс вызывает open() с аргументом /dev/xxx, то совпадет префикс /dev и open() будет направлен к Dev (его владельцу).

Дерево префиксов может содержать частично перекрывающиеся области полномочий. В этом случае ищется наиболее длинное совпадение. Например, предположим, что есть три зарегистрированных префикса:

/	дисковая файловая система (Fsys)
/dev	система символьных устройств (Dev)
/dev/hd0	дисковый том без разделов (Fsys)

Администратор файловой системы зарегистрировал два префикса, один для смонтированной файловой системы КПДА.00002-01 (т.е. /) и один для блок-ориентированного файла, который представляет весь физический жесткий диск (т.е.

/dev/hd0). Администратор устройств зарегистрировал единственный префикс. Таблица 1.13 иллюстрирует правило наиболее длинного совпадения при разборе имен путей.

Таблица 1.13. Правила совпадения префикса

Имя пути	**Совпадает**	**Ответственный процесс**
/dev/con1	/dev	Dev
/dev/hd0	/dev/hd0	Fsys
/usr/dtdodge/test	/	Fsys

Дерево префиксов хранится как список префиксов, разделенных двоеточиями, следующим образом:

prefix=pid,unit:prefix=pid,unit:prefix=pid,unit

где pid - это ID процесса администратора ресурсов I/O, а unit - это один символ, используемый администратором для нумерации префиксов, которыми он владеет. В приведенном выше примере, если Fsys был бы процессом 3 и Dev был бы процессом 5, то дерево префиксов выглядело бы как:

/dev/hd0=3,a:/dev=5,a:/=3,e

Для просмотра дерева префиксов из командного интерпретатора используется утилита prefix.

1.6.2.2. Сетевой корень

Изделие поддерживает концепцию сетевого корня, которая позволяет сопоставлять имя пути с деревом префиксов на конкретном узле. Для обозначения узла имя пути начинается с *двух* косых черт, за которыми следует номер узла. Это также позволяет легко получать доступ к файлу и устройству, которые лежат за пределами пространства путей узла. Например, в типичной сети КПДА.00002-01 возможны следующие имена путей:

/dev/ser1	локальный последовательный порт
//10/dev/ser1	последовательный порт в узле 10
//0/dev/ser1	локальный последовательный порт
//20/usr/dtdodge/test	файл на узле 20

Замечание: //0 всегда указывает на локальный узел.

1.6.2.3. Сетевой корень по умолчанию

Когда программа выполняется на удаленном узле, обычно желательно, чтобы она рассматривала имена путей в контексте пространства имен своего узла. Например, эта команда:

//5 ls /, которая запускает утилиту ls на узле 5, возвратит то же самое, что и:

ls /, запускаемая на своем узле.

В обоих случаях "/" будет соотноситься с деревом префиксов на узле пользователя, а не на узле 5. В противном случае, можно представить себе беспорядок, который мог бы возникнуть, если бы префикс "/" рассматривался своим как для узла 5, так и для своего узла: выбирались бы одновременно файлы из разных файловых систем!

Чтобы правильно интерпретировать имена путей, каждый процесс имеет *сетевой корень по умолчанию*, который определяет, дерево префиксов какого узла будет использоваться для разбора любых имен путей, начинающихся с одиночной косой черты ("/"). Когда разбирается имя пути, начинающееся с одиночной косой черты, то оно предваряется *сетевым корнем по умолчанию*. Например, если процесс имеет сетевой корень по умолчанию //9, тогда:

/usr/home/luc

будет разбираться, как:

//9/usr/home/luc

что может быть интерпретировано, как "разобрать путь /usr/home/luc, используя дерево префиксов 9-го узла".

Сетевой корень по умолчанию наследуется процессами при их создании и соответствует локальному узлу, на котором запускается система. Например, допустим, что пользователь работает на узле 9, находясь в командном интерпретаторе, для которого сетевой корень по умолчанию установлен как узел 9 (весьма типичный случай). Если он выполнит команду:

ls /

команда унаследует сетевой корень по умолчанию //9, и в результате получится:

ls //9/

Подобно тому, если бы ввели команду:

//5 ls /

При запуске команды ls на узле 5 она по-прежнему унаследует сетевой корень по умолчанию //9, поэтому в результате опять получится ls //9/. В обоих случаях имя пути разбирается относительно одного и того же пространства имен.

Передача имен путей между процессами

Если запущено одновременно несколько процессов, они не обязательно имеют один и тот же сетевой корень по умолчанию, даже если они выполняются на одном и том же узле. К примеру, один из процессов мог унаследовать сетевой корень по умолчанию от родительского процесса на каком-либо другом узле сети, либо его сетевой корень по умолчанию мог быть в явном виде задан родительским процессом.

При передаче имени пути между процессами, чьи сетевые корни могут отличаться (например, передавая файл спулеру для печати), необходимо добавить сетевой корень в начало имени пути, прежде чем передать путь другому процессу. Если есть уверенность, что посылающий процесс и получатель имеют один и тот же сетевой корень по умолчанию (или имя пути уже начинается с //узел/), то можно опустить этот шаг.

1.6.2.4. Псевдонимы префиксов

Первая форма префиксов регистрируется администраторами ввода/вывода. Вторая форма префикса известна как *псевдоним префикса* - это простая подстановка строки вместо искомого префикса. Псевдоним префикса имеет форму:

prefix=строка-заменитель

Например, при работе на машине, у которой нет локальной файловой системы (поэтому нет и процесса, который бы отвечал за "/"). Однако, файловая система имеется на другом узле (допустим, 10), и желательно ее использовать как "/". Это можно сделать с помощью следующего псевдонима префикса:

/=//10/

Это приведет к тому, что ведущая косая черта (/) будет заменяться на префикс //10/. Например, путь /usr/dtdodge/test будет заменен следующим:

//10/usr/dtdodge/test

Это новое имя пути будет снова сравниваться с деревом префиксов; на этот раз будет использоваться уже дерево префиксов 10 узла, т.к. имя пути начинается с //10.

Это приведет к Администратору файловой системы на узле 10, которому и будет направлен запрос, например, open(). С помощью всего лишь нескольких символов этот псевдоним позволил обращаться к удаленной файловой системе как к локальной.

Для того чтобы выполнить перенаправление, не обязательно запускать локальную файловую систему. Дерево префиксов для бездисковой рабочей станции может выглядеть примерно так:

/dev=5,a:/=//10/

При таком дереве префиксов номера путей, начинающихся с /dev, будут направляться к локальному Администратору символьных устройств, в то время как запросы с любыми другими именами путей, будут направляться к удаленной файловой системе.

Создание специальных имен устройств

Псевдонимы можно использовать для создания специальных имен устройств. Например, если спулер печати запущен на узле 20, то можно создать для него локальный псевдоним следующим образом:

/dev/printer=//20/dev/spool

Любой запрос на открытие локального принтера /dev/printer будет направляться по сети к реальному спулеру. Аналогичным образом, если не существует локального дисковода для гибких дисков, псевдоним для удаленного дисковода на узле 20 может иметь вид:

/dev/fd0=//20/dev/fd0

В обоих рассмотренных выше случаях можно не использовать перенаправление с помощью псевдонима, а обращаться непосредственно к удаленному ресурсу следующим образом:

//20/dev/spool *ИЛИ* //20/dev/fd0

1.6.2.5. Относительные пути

Путь не обязательно должен начинаться с одной или двух косых черт. В таких случаях путь считается *относительным* по отношению к текущему рабочему каталогу. Изделие хранит текущий рабочий каталог в виде строки символов. Относительные пути преобразуются в полные сетевые имена пути добавлением текущего рабочего каталога в начало относительного пути.

Результат зависит от того, начинается ли текущий рабочий каталог с одинарной косой черты, либо с сетевого корня.

1.6.2.6. Текущий рабочий каталог

Если текущий рабочий каталог начинается с двойной косой черты (сетевого корня), он называется *определенным* и привязан к пространству имен указанного узла. В противном случае, если он начинается с одинарной косой черты, то в начало добавляется сетевой корень по умолчанию.

Например, эта команда:

cd //18/

является примером первой (определенной) формы и привязывает последующие разборы относительных путей к узлу 18, независимо от того, какое значение сетевого корня по умолчанию. Выполнив затем команду cd dev, система попадет в //18/dev.

С другой стороны команда:

cd /

является примером второй формы, когда сетевой корень по умолчанию влияет на разбор относительного пути.

Например, если сетевой корень по умолчанию //9, тогда, выполнив команду cd dev, система попадет в //9/dev. Так как текущий рабочий каталог не начинается с указания узла, то сетевой корень по умолчанию добавляется для получения полного сетевого пути.

При работе сетевые корни (//узел/) обычно *не* указываются, и все, что делается, работает внутри пространства имен узла (определяемого сетевым корнем по умолчанию).

Замечание о cd

В некоторых традиционных UNIX-подобных операционных системах команда cd (сменить директорию) модифицирует заданный путь, если он содержит символьные ссылки. Как результат, путь к новому текущему рабочему каталогу - который можно посмотреть командой pwd, может отличаться от того, который был задан в команде cd.

В изделии, однако, cd не изменяет путь - за исключением сокращенных ссылок "..". Например:

cd /usr/home/luc/test/../doc

установит текущий рабочий каталог /usr/home/luc/doc, даже если некоторые элементы этого пути являются символическими ссылками.

Более подробная информация о символических ссылках размещена в разделе «Администратор файловой системы».

Замечание: Для получения полного сетевого имени, следует использовать утилиту fullpath.

1.6.3. Пространство дескрипторов файлов

При открытии ресурса ввода/вывода функция open() возвращает целое число, называемое *дескриптор файла* (FD), которое используется затем, чтобы направлять все последующие запросы I/O к данному Администратору. (Внутри библиотечных процедур для направления опроса используется вызов ядра Sendfd().)

Пространство дескрипторов файлов, в отличие от пространства имен пути, полностью локально для каждого процесса. Администратор использует сочетание PID и FD, чтобы определить управляющую структуру, созданную предыдущим вызовом open(). Эта структура называется *блок управления файлом* (OCB от английского open control block) и хранится внутри администратора I/O.

Следующая диаграмма показывает, как администратор I/O находит соответствующий OCB для пары PID и FD.

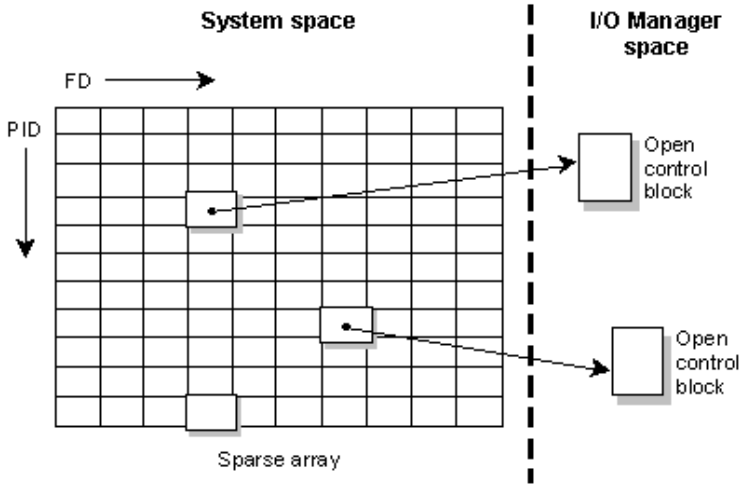


Рис. 28. PID и FD определяют OCB для Администратора ввода/вывода.

1.6.3.1. Блоки управления файлами

OCB содержит действующую информацию об открытом ресурсе. Например, файловая система хранит здесь текущий указатель на позицию в файле. Каждый вызов open() создает новый OCB. Таким образом, если процесс дважды открывает один и тот же файл, любые вызовы lseek(), использующие один FD, не повлияют на текущую позицию для другого FD. То же самое справедливо и для различных процессов, открывающих один и тот же файл. Следующая диаграмма показывает два процесса, один из которых открывает файл дважды, а другой открывает этот же файл один раз. При этом нет разделяемых FD.

Несколько дескрипторов файла в одном или нескольких процессах могут указывать на один и тот же OCB. Это может быть достигнуто двумя способами:

- процесс может использовать функции Си dup(), dup2() или fcntl() для создания дубликата дескриптора файла, который указывает на тот же самый OCB;

- при создании нового процесса посредством fork(), spawn() или exec(), все дескрипторы открытых файлов по умолчанию наследуются новым процессом; эти наследуемые дескрипторы указывают на те же самые OCB, что и соответствующие дескрипторы файла в родительском процессе.

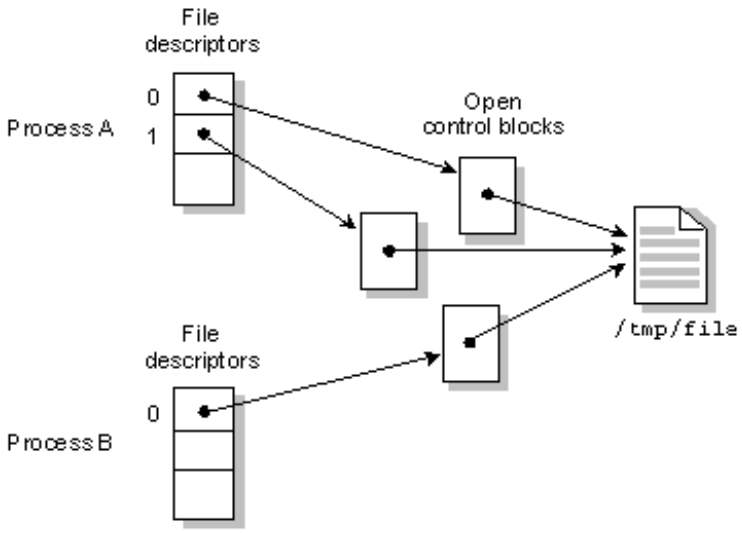


Рис. 29. Процесс А открывает файл /tmp/file дважды. Процесс В открывает тот же файл один раз.

Когда несколько FD указывают на один и тот же ОСВ, тогда любое изменение в состоянии ОСВ немедленно видимо для всех процессов, у которых есть дескрипторы файлов, указывающие на этот ОСВ.

Например, если один из процессов использует функцию lseek() для изменения текущей позиции, тогда чтение или запись начинается с новой позиции, независимо от того, какой из дескрипторов файла используется. Следующая диаграмма показывает два процесса, из которых один открывает файл дважды, а затем вызывает dup(), чтобы получить третий дескриптор. Затем он создает дочерний процесс, который наследует все открытые файлы.

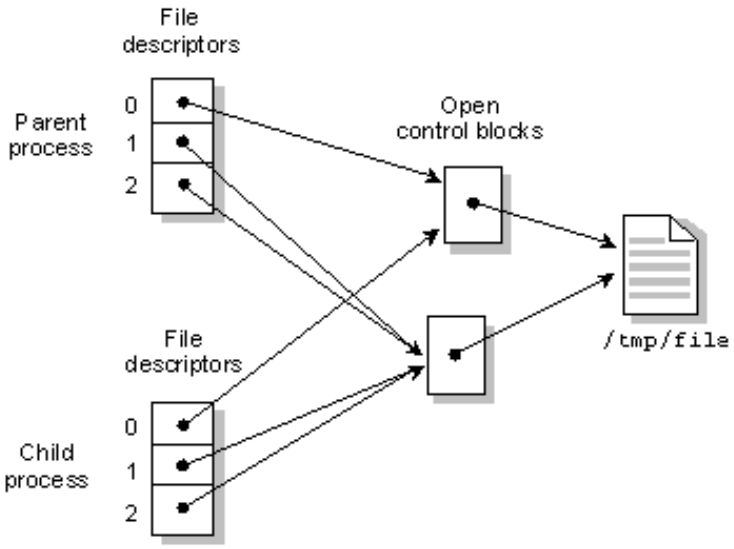


Рис. 30. Процесс открывает файл дважды, а затем получает третий FD посредством dup(). Его дочерний процесс унаследует все эти три дескриптора файла.

Наследование дескриптора файла может быть предотвращено при использовании spawn() или exec(), предварительно вызвав функцию fcntl() и установив флаг FD_CLOEXEC.

2. ПОСТРОЕНИЕ ЦЕЛЕВЫХ СИСТЕМ

2.1. Установка инструментальной системы

2.1.1. Общие указания

Установите комплект разработчика КПДА.00002-01, используя инсталляционный компакт-диск производства компании ООО «СВД ВС» и лицензионные ключи на устанавливаемое программное обеспечение. Затем с компакт-диска производства компании ООО «СВД ВС» установите комплект программных компонент изделия «Операционная система реального времени КПДА.00001-01», руководствуясь документом «Описание применения» соответствующего изделия.

2.2. Построение образа ОС

2.2.1. Общие замечания

При формировании образа ОС необходимо выбрать сервисные функции, которые нужно иметь доступными немедленно после загрузки образа, и включить в образ процессы, которые обеспечивают эти функции в построенной ОС. Этот образ создается утилитой buildkpda. Образ может быть загружен с диска программой-загрузчиком раздела КПДА.00002-01 или загружен через сеть утилитой netboot.

Замечание: Файлы для построения целевых систем находятся в каталоге $HOME/kpda. Для удобства в дальнейшем все пути будем указывать без добавления «$HOME/kpda» Например, если указывается каталог /boot, то подразумевается $HOME/kpda/boot и т.д.

2.2.2. Порядок создания файла построения

Утилита buildkpda выдает двоичный файл образа, содержащий несколько процессов, которые перечислены во входном текстовом файле, называемом файлом "построения". Файлы построения содержатся в каталоге /boot/build, а файлы образа содержатся в каталоге /boot/images.

Можно создать образ для любого узла, вызывая непосредственно buildkpda или используя утилиту make и Makefile в каталоге $HOME/kpda/boot.

Например, можно ввести любую из следующих команд, чтобы создать файл образа, именуемый hard.1, из образца файла построения hard.1:

```
cd /boot
buildkpda build/hard.1 images/hard.ata.1
```

или

```
cd /boot
make b=hard.ata.1
```

2.2.3. Формат файла построения

Каждая программа, которую необходимо включить в создаваемый образ, предусматривает использование трех строк в файле построения:

1) первая строка - путь к программе, которую необходимо включить;

2) вторая строка начинается с символа $, за ним - начальное значение (по умолчанию - 1) размера динамической области памяти ("кучи"), а за размером "кучи" следует команда;

3) третья строка должна быть пустой, она отделяет одну запись от другой.

Замечание: Соответствующие имена пути должны предполагать запуск в /boot.

Пример файла построения hard.ata.1:

```
sys/boot
$ 1 boot -v

sys/Proc32
$ 1 Proc32 -l 1

sys/Slib32
$ 1 Slib32

sys/Slib16
$ 1 Slib16

/bin/Fsys
$ 1 Fsys

/bin/Fsys.ata
$ 1 Fsys.ata

/bin/mount
$ 1 mount -p /dev/hd0 /dev/hd0t77 /

/bin/sinit
$ 1 sinit TERM=ansi
```

Установка размеров "кучи"

Число после символа $ - начальный размер "кучи". Значение "1" заставит размер "кучи" динамически расти до необходимой величины.

Установка приоритетов

Можно установить начальный приоритет процесса посредством задания вслед за значением размера "кучи" необязательного аргумента :nn - нужный приоритет,

задаваемый в диапазоне от 1 (самый низкий) до 30 (самый высокий). Например, следующая команда запустит драйвер гибких дисков (Fsys.floppy) с приоритетом 8:

/bin/Fsys.floppy

$ 1:8 Fsys.floppy

2.2.4. Хранение исполняемых файлов

Большинство исполняемых файлов системы КПДА.00002-01 хранится в каталоге /bin. Некоторые элементы пригодны для использования только как часть образа начальной загрузки, они хранятся в подкаталогах в каталоге /boot, см. таблицу 2.1.

Таблица 2.1. Расположение файлов образа

Файлы	**Расположение**
Системные исполняемые файлы	/bin
Makefile образа ОС	/boot
Файлы построения для создания образа (их читает утилита make)	/boot/build
Файл образа ОС	/boot/images
Процессы системы, необходимые во время начальной загрузки	/boot/sys

2.2.5. Выбор компонентов для образа

2.2.5.1. Общие замечания

Компоненты, которые включаются в образ ОС, определяются по нескольким факторам. Можно сгруппировать образы в три класса:

1) образы, загружаемые с диска;

2) образы, загружаемые из сети;

3) образы, загружаемые во встраиваемых системах.

Файл boot (/boot/sys/boot) всегда автоматически включается в первую строку сгенерированного образа, даже если он явно не определен. Если необходимо задать опции для sys/boot, нужно добавить строку "sys/boot" в файл построения как первый процесс.

Для образов, которые загружаются с диска или через сеть, можно запустить большинство процессов после загрузки, размещая их вызов в файле инициализации системы, который выполняется после загрузки образа (смотрите подраздел "Файл инициализации системы" в разделе «Базовая установка»). Это позволит сохранять образ загрузки небольшим (< 512K) и простым.

Замечание: Файл построения обычно не содержит Администратора устройств /bin/Dev. Dev и его драйверы обычно запускаются в файле инициализации системы после начальной загрузки системы. Поэтому если файл инициализации системы не выполнен, то Dev не будет запущен. В результате этого клавиатура и системная консоль не будут функционировать.

2.2.5.2. Обязательные компоненты

При построении образа, необходимо учитывать, что есть два обязательных процесса:

1) Администратор процессов с микроядром (/boot/sys/Proc32);

2) системная совместно-используемая (разделяемая) библиотека (/boot/sys/Slib32).

Чтобы запустить 16-битовые программы, нужно включить 16-битовую совместно-используемую библиотеку /boot/sys/Slib16.

2.2.5.3. Дисковые образы

Для загрузки с жесткого диска в образ необходимо включить:

1) два обязательных системных процесса (первый - Proc32, а второй - Slib32);

2) Администратор файловой системы (Fsys);

3) драйвер, необходимый для доступа к накопителю (Fsys.driver).

Замечание: Для получения дополнительной информации см. описание buildkpda, Fsys*, Proc32, Slib* и sinit в книге «Описание программы».

Использование файла Makefile для загрузки с диска

Файл Makefile для создания образов начальной загрузки (в каталоге /boot) содержит входные данные для создания настраиваемого образа загрузки с жесткого

диска. Если при построении образа ОС используется утилита make и при этом не вводятся новые значения параметров, будут использоваться значения, задаваемые по умолчанию.

Пример файла make, в котором создается образ загрузки с именем /boot/images/hard.1, используя файл построения с именем /boot/build/hard.1:

```
cd /boot
make b=hard.1
```

Необходимо помнить, что можно создавать образы либо статически, вызывая утилиту buildkpda, или динамически, выполняя утилиту netboot. Для получения более подробной информации необходимо смотреть описание этих утилит в книге «Описание программы».

Пример вызова buildkpda "вручную", в котором создается образ с именем /boot/images/ws32.ether1000 из файла построения с именем /boot/build/ws32.ether1000:

```
cd /boot
buildkpda build/ws images/ws32.ether1000
```

Копирование образа в /.boot

Построенный образ не станет образом новой начальной загрузки до тех пор, пока его не скопируют в файл /.boot. Однако, прежде чем это сделаеть, следует сохранить текущий файл /.boot, скопировав его в файл /.altboot:

```
cp /.boot /.altboot
```

Если по какой-либо причине новый образ не работает правильно, можно нажать Esc после подсказки во время процесса начальной загрузки и загрузить файл /.altboot вместо файла /.boot.

Если выбирается образ альтернативной начальной загрузки, то обычная проверка файла /etc/config/sysinit.узел заменяется проверкой файла /etc/config/altsysinit. При этом необходима гарантия того, что файл altsysinit содержит самую последнюю копию работающего файла sysinit:

```
cp /etc/config/sysinit.узел /etc/config/altsysinit
```

2.2.5.4. Сетевые образы

Для загрузки через сеть в состав образа нужно включить:

1) два обязательных системных процесса (первый - Proc32, а второй - Slib32);

2) Администратор сети (Net)

3) драйвер, необходимый для доступа к аппаратным средствам сети (Net.driver).

Файл построения образа для сетевой загрузки

Следующий файл построения (/boot/build/ws.ether1000) содержит маркеры параметра, которые унаследуют их значения из сетевого окружения:

```
sys/Proc32
$ 1 Proc32 -l $(lnode) -D

sys/Slib32
$ 1 Slib32

sys/Slib16
$ 1 Slib16

/bin/Net
$ 1 Net -m $(netmap)

/bin/Net.ether1000
$ 1 Net.ether1000

/bin/sinit
$ 1 sinit -r //$(bnode)/  TERM=ansi TZ=$(TZ)

sys/Debugger32
$ 1 Debug
```

Маркер $(lnode) в команде Proc32 берет в качестве своего значения логический ID узла загружающегося компьютера. Логический ID узла определяется утилитой netboot на основе адреса Ethernet загружающегося компьютера и записи в файле /etc/config/netmap и затем передается утилите buildkpda.

Маркер $(netmap) в команде Net сообщает Net первоначальную карту сети сервера начальной загрузки с тем, чтобы загружающийся компьютер смог начать сетевую связь с ним.

Маркер $(bnode) в команде sinit представляет логический ID узла сервера начальной загрузки. Команда устанавливает корень файловой системы (/) загружающегося компьютера в корень сервера загрузки.

Значение маркера $(TZ) наследуется от сервера начальной загрузки. Команда устанавливает информацию зоны времени загружающегося компьютера, чтобы она совпадала с текущей установкой переменной окружения TZ сервера начальной загрузки.

Замечания:

1. Если необходимо создать для узла предварительно построенный образ, то нельзя использовать маркеры, а вместо этого нужно жестко кодировать требуемые значения непосредственно в пользовательском файле построения. Для более полной информации следует ознакомиться с утилитой buildkpda в книге «Описание программы».

2. При загрузке через сеть, имеется вариант загрузки либо предварительно созданного образа, либо образа, создаваемого «на лету» - по ходу установки. При создании образа «на лету», что рекомендуется, не будете нужды в том, чтобы строить его вручную, как показано выше. Вариант определяется в файле /etc/config/netboot и документирован в описании утилиты netboot.

Когда утилита netboot вызывает buildkpda, чтобы построить образ «на лету», загрузочный модуль не записывается на диск.

2.2.5.5. Встраиваемые образы

Встраиваемые образы требуются для использования изделия во встроенных системах с центральным процессором. Изделие содержит компоненты, необходимые для поддержки различных плат центрального процессора и микросхем флэш-памяти. Эта поддержка включает загрузку из платы флэш-памяти и использование флэш-памяти как файловой системы. Разработку программного обеспечения, необходимого для поддержки дополнительных процессорных плат, выполняет ООО «СВД ВС».

2.3. Файловая система среды исполнения

2.3.1. Общие рекомендации

Каждая файловая система имеет свой собственный корневой каталог. Этот корень находится в верхней части иерархии каталогов, от него КПДА.00002-01 начинает поиск других каталогов и файлов. Типичное имя для корневого каталога файловой системы на жестком диске по умолчанию - слеш, наклонная черта вправо, (/).

Файловая система с корнем / может состоять из одной или более физических файловых систем, подсоединенных вместе. Физическая файловая система может быть на отдельном диске или разделе диска.

Если жесткий диск имеет более чем одну файловую систему, их имена могут быть следующими: /hd2, /hd3, /home2, /home3 и т.д.

Одна из физических файловых систем обычно назначается корнем (/), в то время как другие файловые системы монтируются как подкаталоги.

Дерево префиксов определяет принадлежность путей Администраторам ввода/вывода и определяет, к каким дискам и устройствам осуществляется доступ, когда процесс открывает файл. Во всех примерах этой главы дерево префиксов используется для разделения пространства имен.

2.3.2. Разделение пространства имен путей

Утилита mount может использоваться, чтобы монтировать разделы диска как блок-ориентированные файлы и чтобы монтировать блок-ориентированные файлы как файловые системы КПДА.00002-01. При монтировании блок-ориентированного файла как файловой системы, ее позиция в пространстве имен путей называется точкой монтирования.

Например, задан раздел КПДА.00002-01 (тип 77) на жестком диске 0 (/dev/hd0), следующая команда создаст блок-ориентированный файл /dev/hd0t77:

```
mount -p /dev/hd0
```

Следующая команда будет монтировать все разделы, обнаруженные на жестком диске 0, и установит раздел КПДА.00002-01 как корень:

```
mount -p /dev/hd0 /dev/hd0t77 /
```

Замечания:

1. Пользователь, имеющий разрешение записи на устройство или раздел, может монтировать или демонтировать это устройство.
2. Рекомендуется демонтировать съемные носители информации перед тем, как вынуть их из дисковода. Это поможет предотвратить необычные ошибки или возможную порчу файлов. Для дополнительной информации смотрите описание umount в документе «Описание программы».

Следующие примеры иллюстрируют, как пространство имен путей можно разбить на разделы. Рассмотрим эти конфигурации:

1) жесткий и гибкий диск;

2) два жестких диска;

3) несколько разделов КПДА.00002-01 на жестком диске;

4) локальные (местные) и удаленные жесткие диски.

В этих примерах принимается, что Fsys и соответствующие драйверы запущены, и что на жестком диске была выполнена команда mount -p.

2.3.2.1. Жесткий и гибкий диск

Жесткий диск монтируется как наклонная черта вправо - слеш (/), и формируется корень файловой системы. Гибкий диск монтируется как /fd0:

```
mount /dev/hd0t77 /
mount /dev/fd0 /fd0
```

Любое обращение с именем пути, начинающимся с /fd0, будет адресоваться к файловой системе КПДА.00002-01 на гибком диске. Например, показать все файлы на гибком диске можно следующей командой:

```
ls -aR /fd0
```

2.3.2.2. Два жестких диска (на одном узле)

Первый жесткий диск монтируется как слеш (/) и образует корень файловой системы. Второй жесткий диск монтируется как /home2:

```
mount /dev/hd0t77 /
mount /dev/hd1t77 /home2
```

Любое обращение с именем пути, начинающимся с /home2, будет адресоваться к файловой системе КПДА.00002-01 на втором жестком диске. Например, показать все файлы на втором жестком диске можно следующей командой:

```
ls -aR /home2
```

2.3.2.3. Несколько разделов КПДА.00002-01 на жестком диске

На одном жестком диске можно иметь до четырех разделов КПДА.00002-01. Первый раздел должен иметь тип 77 (смотрите утилиту fdisk в книге Описание программы). Остальным разделам должны быть назначены типы 78, 79 и 80. Например:

```
mount /dev/hd0t77 /
mount /dev/hd0t78 /home2
```

Любое обращение с именем пути, начинающимся с /home2, будет адресоваться к файловой системе КПДА.00002-01 во втором разделе жесткого диска. Например, показать все файлы во втором разделе жесткого диска можно следующей командой:

```
ls -aR /home2
```

2.3.2.4. Локальные и удаленные жесткие диски

В сети можно иметь диски с файловой системой КПДА.00002-01 на более чем одном компьютере. Можно сконфигурировать пространства имен так, чтобы эти файловые системы были:

независимыми;

первичными/вторичными;

связанными независимыми.

В следующих примерах локальные файловые системы ассоциируются с блок-ориентированными файлами (например, /dev/hd0t77), а удаленные файловые системы ассоциируются с отображениями префиксов имен путей. Это отображение префиксов переназначает запросы к удаленной файловой системе, которая будет ассоциирована с удаленным блок-ориентированным файлом.

Независимая

В этой конфигурации каждая машина рассматривается как независимая автономная файловая система. Для доступа к файлу на удаленной машине необходимо в начале имени пути указать номер узла удаленной машины. Например:

```
//10/etc/motd
```

Возможность указать файловую систему для конкретного узла будет работать всегда и являться наиболее общим механизмом для доступа к удаленным файлам.

Первичная/вторичная

В этой конфигурации одна файловая система рассматривается как первичная, а вторая файловая система монтируется как подкаталог первичной. Например, предположим, что узел 1 имеет первичную файловую систему, а узел 2 - вторичную, смонтированную как /home2. Узел 1 загружен с жесткого диска и имеет корень, установленный на его локальном диске утилитой mount, встроенной в образ операционной системы (с Fsys и драйвером). Его файл системной инициализации вызывает утилиту prefix, чтобы монтировать удаленную файловую систему следующим образом:

```
prefix -A /home2=//2/home2
```

Узел 2 загружается через сеть с узла 1 и имеет корень файловой системы, установленный /=//1/, используя утилиту sinit с опцией -r, встроенную в образ ОС. Его файл системной инициализации должен вызвать утилиту mount, чтобы монтировать локальную файловую систему следующим образом:

```
mount /dev/hd0t77 /home2
```

Администратор файловой системы (Fsys) и его драйвер могут быть включены в образ, но более предпочтительно запускать их из файла системной инициализации. Другими словами, узел 2 загружается, подобно простой бездисковой машине (рабочей станции), которая после загрузки запускает свою файловую систему.

Оба узла, 1 и 2 будут иметь доступ к файловой системе на узле 1 как /, а на узле 2 - как /home2.

Связанная независимая

В этой конфигурации каждая машина является независимой автономной файловой системой, однако, частично связанными вместе посредством утилиты prefix. Например, допустим, что файловая система на узле 1 имеет каталог /home1, а файловая система на узле 2 имеет каталог /home2. Можно отобразить каждый из каталогов home в область другой файловой системы следующим образом:

На узле 1:

```
prefix -A /home2=//2/home2
```

На узле 2:

```
prefix -A /home1=//1/home1
```

Кроме этой связи, каждая файловая система автономна со своими собственными копиями /bin и т.п. Преимущество - большая избыточность: если один отдел использует узел 2, а узел 1 отключается, то отдел, используя узел 2, может продолжать работу (кроме файлов из /home1, которые будут недоступны).

В качестве альтернативы, можно использовать символические ссылки, чтобы обращаться к файлам в другой файловой системе. Символическая ссылка создает специальный файл, который имеет имя пути как его данные. Чтобы создать символическую ссылку, задайте опцию -s для утилиты ln.

Допустим, на узле 1 в каталоге /home1 есть файл с именем file1. Следующая команда обеспечит отделу, использующему узел 2, доступ к file1 через символическую ссылку file2 в каталоге /home2.

```
ln -s /home1/file1 //2/home2/file2
```

Этим создается символическая ссылка //2/home2/file2, которая указывает на /home1/file1. Теперь можно отобразить на экране содержимое file1 из каталога /home2 на узле 2, используя утилиту less, следующим образом:

```
less file2
```

Удаление и повторная установка драйверов

Суперпользователь может удалять (обычно с помощью утилиты rm) блок-ориентированные устройства. Если все устройства, за которые отвечает драйвер, удаляются, то драйвер будет завершаться автоматически. Это позволяет пользователям с соответствующими привилегиями устанавливать, удалять и повторно устанавливать драйверы в работающей системе.

Замечание: Для получения информации о том, как изменять атрибуты доступа пользователя, смотри подраздел "Атрибуты доступа файлов" в разделе «Настройка учетных записей».

2.3.3. Установка файловой системы DOS

Администратор файловой системы Dosfsys обеспечивает доступ к файлам DOS и каталогам, которые находятся на дисках или разделах дисков с DOS. Dosfsys может поддерживать до восьми дисководов.

Имеется возможность создавать, читать, записывать и удалять файлы и каталоги на дисках DOS с использованием большинства программ и стандартных утилит изделия (таких как mkdir, ls и rmdir). Большинство утилит изделия будут работать с файлами DOS при условии, что структура файлов DOS соответствует функциональным требованиям утилиты.

Собственные Си-программы также смогут обрабатывать файлы DOS точно так же, как они обрабатывают файлы КПДА.00002-01, используя стандартные функции ввода/вывода, такие как open(), read(), write(), close(), seek() и т.л. Каталоги DOS представляются в формате КПДА.00002-01.

2.3.3.1. Режимы вызова

Dosfsys имеет четыре режима вызова:

```
Dosfsys [-S|-s][-e][-m][-t][dos_drive=kpda_drive[,R]]... &

Dosfsys -i [-n node] [dos_drive_path]...

Dosfsys -o [-n node]

Dosfsys -x [-n node]
```

Опция -i позволяет получать информацию о накопителях DOS, подключенных в настоящее время. Опция -o позволяет увидеть имена любых файлов, открытых в настоящее время на каждом устройстве. Опция -x завершает работу сервера Dosfsys.

Если не заданы -i, -o или -x, то Dosfsys будет запускаться и пытаться подключить указанные дисководы.

Для того чтобы запустить сервер Dosfsys или завершить его работу, необходимо войти в систему как суперпользователь (root).

2.3.3.2. Запуск Dosfsys

При запуске администратор Dosfsys выполняет следующие действия:

1) открывает определенный дисковод(ы);

2) регистрирует корневое имя DOS (/dos)

3) регистрирует имя kpda/dosfsys у локального Администратора процессов.

Если никакие опции не были определены, или если были заданы опции -S или -s, то Dosfsys просматривает каталог /dev для подключения допустимых дисководов DOS. Если в командной строке не задано иначе, то Dosfsys будет пытаться подключить до восьми дисководов.

Первичные разделы DOS (/dev/hd0t1, /dev/hd0t4, /dev/hd0t6 и т.п.) и расширенные разделы (/dev/hd0t1.1, /dev/hd0t1.2 и т.п.) монтируются, начиная с накопителя C (/dos/c).

Дисководы гибких дисков монтируются следующим образом:

```
/dev/fd0   -->  /dos/a

/dev/fd1   -->  /dos/b
```

2.3.3.3. Регистрация имен Администратором Dosfsys

Администратор файловой системы Dosfsys может подключить до восьми дисководов. Как упоминалось выше, Администратор Dosfsys зарегистрирует имя /dos как системный префикс. Он также сможет управлять каждым специфическим именем дисковода, таким как /dos/a, /dos/b и т.п. Эти имена не регистрируются в дереве системных префиксов, но содержатся непосредственно в Dosfsys. Это могло быть прозрачно для пользователя, если бы не то обстоятельство, что пользователь не может создавать файлы или каталоги в корне /dos.

2.3.3.4. Устройства DOS

Устройство DOS может быть одним из следующих:

1) раздел DOS на жестком диске;

2) гибкий диск (дискета);

3) образ раздела DOS или дискеты.

Для того чтобы создать образ дискеты DOS или раздела DOS, используйте утилиту cp. Например, для того, чтобы скопировать образ дискеты DOS на дисковод гибких дисков номер 0, можно использовать следующее:

```
cp /dev/fd0 /usr/kpda/dosa
```

и затем вызвать Dosfsys следующим образом:

```
Dosfsys a=/usr/kpda/dosa &
```

То же самое можно выполнить с разделом жесткого диска. Dosfsys будет обрабатывать эти образы так же, как будто это существующее устройство.

Для всех несъемных устройств Dosfsys при запуске считывает непосредственно блок параметров начальной загрузки DOS (Boot Parameter Block - BPB), а также часть таблицы распределения файлов (File Allocation Table - FAT). На съемных устройствах BPB и FAT будут считаны только тогда, когда устройство будет доступно.

Когда Dosfsys имеет открытое несъемное устройство, то устройство заблокировано для ЗАПИСИ так, что никакой другой процесс не сможет записывать на это устройство без взаимодействия с Dosfsys. Съемные устройства остаются открытыми и блокируются только во время доступа (например, во время чтения или записи на диск). Обратите внимание, что если опция R не задана, то все дисководы имеют доступ для ЧТЕНИЯ/ЗАПИСИ.

2.3.3.5. Поддержка версий DOS

Администратор Dosfsys поддерживает все форматы разделов, начиная с DOS версии 2.1 или выше, включая стандартные первичные разделы DOS, большие разделы DOS (DOS 4.0 и DOS 5.0 > 32 Мбайт) и расширенные разделы DOS (тип 5). Поддерживаются жесткие диски и гибкие диски размера 51/4" и 31/2".

Типы разделов DOS

Стандартные типы разделов жесткого диска DOS представлены в таблице 2.2.

Таблица 2.2. Типы разделов DOS

Тип раздела	Описание
1	первичный раздел DOS (12-битовые FAT)
4	первичный раздел DOS (16-битовые FAT; <= 32 Мбайт)
5	расширенный раздел DOS (DOS 3.3 или более поздней)
6	первичный раздел DOS (DOS 4.0 или более поздней; > 32 Мбайт)
11	32-битовые FAT DOS; разделы до 2047 Гбайт
12	то же самое, что и тип 11, но для увеличения объема используется Logical Block Address (LBA) и Int 13h
14	то же самое, что и тип 6, но для увеличения объема используется Logical Block Address (LBA) и Int 13h
15	то же самое, что и тип 5, но для увеличения объема используется Logical Block Address (LBA) и Int 13h

2.3.3.6 Текстовые файлы DOS

DOS использует структуру для текстовых файлов, которая отличается от структуры, используемой в КПДА.00002-01 (текстовыми файлами мы называем строковые файлы, содержащие строки ASCII-текста, разделяемые последовательностями разделителей строк). В DOS каждая строка текстового файла завершается последовательностью возврат каретки/перевод строки (CR/LF); в КПДА.00002-01 каждая строка завершается символом перевода строки (LF).

Администратор Dosfsys не преобразует эти файлы. Все файлы обрабатываются "как есть". Поэтому необходимо использовать утилиту textto или tr, чтобы преобразовывать текстовые файлы перед копированием их на диски или с дисков КПДА.00002-01 и DOS.

Следует обратить также внимание, что текстовые файлы, созданные некоторыми программами DOS, могут содержать символ SUB (^Z) как последний символ файла. Он тоже обрабатывается "как есть".

2.3.3.7. Двоичные файлы DOS

Поскольку Dosfsys не преобразует содержимое файлов, двоичные файлы могут быть скопированы из разделов КПДА.00002-01 в разделы DOS и обратно "как есть".

2.3.3.8. Преобразование символов и имен из КПДА.00002-01 в DOS

В DOS имена файла ограничиваются в соглашении "8.3" (КПДА.00002-01 не имеет такого ограничения). Кроме того, нельзя включить любой из представленных ниже символов в имя файла (несмотря на то, что они вполне допустимы в КПДА.00002-01):

/ \ [ ] : * | + = ; , ?

Нельзя использовать следующие имена файлов:

AUX, CLOCK$, COM1, COM2, COM3, COM4, CON, LPT1, LPT2, LPT3, NUL, PRN.

При попытке в DOS создать файл, который содержит один из недопустимых символов DOS или имеет недопустимое имя, вам будет отказано в доступе.

Так как под КПДА.00002-01 все имена файлов DOS и символы имен файлов разрешены проверки правильности для этих имен файлов не требуется.

DOS также преобразует все символы алфавита в верхний регистр, поэтому Dosfsys преобразует эти символы в верхний регистр при создании имени файла DOS; он преобразует имя файла в нижний регистр при возврате имени файла в приложение КПДА.00002-01.

Преобразование имен файлов КПДА.00002-01

Если при запуске -t задается Dosfsys, то сервер будет преобразовывать и/или усекать любые имена файлов КПДА.00002-01, которые не соответствуют соглашению о присваивании имен в DOS. Таблица 2.3. описывает, как преобразовываются символы имен файлов КПДА.00002-01 в имена DOS.

Таблица 2.3. Правила замены символов

Символ в имени файла КПДА.00002-01	**Действие Dosfsys**
Начинается с точки (.)	Изменять . на $
Содержит несколько точек	Изменять все точки, кроме последней, на $
Имеет префикс длиннее восьми символов	Усекать префикс до восьми символов
Имеет суффикс длиннее трех символов	Усекать суффикс до трех символов

Замечание: В версии Windows 95 фирмы Microsoft добавлена поддержка длинных имен файлов, которые не ограничиваются традиционным соглашением 8.3 DOS об именах файлов. Dosfsys может считывать, но не создавать, длинные имена файлов Windows 95. Чтобы Dosfsys распознавала длинные имена файлов Windows 95, при запуске Dosfsys задайте опцию -L.

Несколько примеров преобразования имен КПДА.00002-01 в имена DOS представлены в таблице 2.4.

Таблица 2.4. Преобразование имен из КПДА.00002-01 в DOS.

Имя файла КПДА.00002-01	**Имя файла DOS**
.profile	$profile
a.b.c.d	abc.d
longfilename	longfile

Имя файла КПДА.00002-01	Имя файла DOS
longfilename.extension	longfile.ext
a..b..c.def.g.h	a$$b$$c$.h

2.3.3.9. Метки тома DOS

DOS использует понятие метки тома, которое является действующим элементом каталога в корне файловой системы DOS. Чтобы различать метку тома и действующий файл DOS, Dosfsys ставит знак равенства (=) в качестве первого символа имени метки тома. Dosfsys трактует этот элемент каталога как файл нулевой длины только для чтения, чьи атрибуты доступа не могут быть изменены.

2.3.3.10. Преобразование атрибутов доступа DOS/КПДА.00002-01

DOS не поддерживает все биты атрибутов доступа, которые использует КПДА.00002-01. DOS имеет следующие биты атрибутов:

READ_ONLY - только для чтения

HIDDEN – скрытый

SYSTEM – системный

VOLUME_LABEL - метка тома

DIRECTORY – каталог

ARCHIVE – архивный

Преобразование битов атрибутов доступа КПДА.00002-01 в DOS

Dosfsys использует следующую логику соответствия, чтобы осуществить преобразование битов атрибутов КПДА.00002-01 в DOS:

если запись представляет собой каталог, то устанавливается бит файла DOS DIRECTORY;

если запись представляет собой файл, и если все биты WRITE выключены, то устанавливается бит DOS READ_ONLY.

Преобразование битов атрибутов доступа DOS в КПДА.00002-01

Чтобы осуществить преобразование битов атрибутов доступа DOS в КПДА.00002-01, используется следующая логика соответствия:

устанавливаются биты атрибутов доступа READ ("чтение") для пользователя, группы и для остальных;

если запись не является меткой тома и если запись не только для чтения, то устанавливаются биты атрибутов доступа WRITE ("запись") для пользователя, группы и для остальных;

если запись представляет собой каталог, то устанавливаются биты DIRECTORY ("каталог") и EXECUTE ("выполнить") для пользователя, группы и для остальных;

если запись представляет собой файл, устанавливается бит REGULAR FILE ("регулярный файл").

Замечание: Если файл записывается, то будет устанавливаться бит DOS ARCHIVE ("архивный").

Если Dosfsys запускается с опцией -e, все файлы и каталоги DOS будут иметь набор битов КПДА.00002-01 для группа/владелец/остальные.

2.3.3.11. Принадлежность файла

Хотя файловая система DOS не поддерживает идентификаторы пользователей и идентификаторы групп, Dosfsys не будет возвращать код ошибки при попытке изменить идентификатор группы или идентификатор пользователя утилитой chown или библиотечной функцией chown(). Ошибка не возвращается, потому что ряд утилит использует библиотечную функцию chown(), что может привести к большому количеству отображаемых сообщений об ошибках.

Все файлы под Dosfsys, принадлежащие суперпользователю (uid=0, group=0), доступны для всех.

2.3.3.12. Завершение Dosfsys

Опция -x прекращает действия сервера Dosfsys. Если опция-x задана, то никакие новые запросы функций open() не будут приниматься и сервер закончит работу, как только все активные файлы (то есть файлы все еще открытые) закроются.

2.3.3.13. Коды ошибок, возвращаемые Dosfsys

Если запрос, сделанный к Dosfsys, не поддерживается, то приложению, делающему запрос, будет возвращен код ошибки EOPNOTSUPP. Примеры запросов, не поддерживаемых Dosfsys:

LINK
BLOCK_READ
BLOCK_WRITE
MOUNT_PARTITION
MOUNT_RAMDISK
PIPE
DISK_GET_ENTRY
RECORD LOCKING
SYMBOLIC LINK

Если Dosfsys обнаруживает испорченную файловую систему, то будет возвращена ошибка EBADFSYS, которая указывает на необходимость выполнения утилиты CHKDSK под DOS, чтобы устранить проблему.

Структура файловой системы DOS такая, что размер корневого каталога фиксируется во время форматирования и не может изменяться. Если корневой каталог переполняется, то выдается ошибка (ENOSPC).

2.4. Подключение символьных устройств

ЭВМ обычно содержит одно или более символьных устройств. Все эти устройства управляются процессом Dev. Сначала должен быть запущен этот процесс (как root), затем можно запускать любые драйверы устройств:

```
/bin/Dev &
```

Как только Dev запущен, можно запустить один или более следующих драйверов устройств (как root):

/bin/Dev.con (драйвер консольного устройства);

/bin/Dev.par (драйвер параллельного печатающего устройства);

/bin/Dev.ser (драйвер последовательного устройства);

/bin/Dev.pty (драйвер псевдотерминала).

Каждый из этих драйверов более подробно описан в документе «Описание программы».

Замечание: Может возникнуть необходимость модификации команды Dev в в файле sysinit.узел с целью поддержки большего числа устройств, чем это принято по умолчанию. Например, если задать:

```
Dev [другие_опции] -n 64 &
```

то Dev будет поддерживать 64 терминальных устройства (виртуальных консолей, последовательных терминалов и псевдотерминальных устройств).

2.5. Порядок создания сети

2.5.1. Общие замечания

Изделие является сетевой распределенной операционной системой. Каждый компьютер под управлением изделия называется *узлом*. Одиночный компьютер можно рассматривать как сеть с одним узлом.

Администратор сети распознает широкий диапазон пакетов протоколов и передает их прозрачно соответствующим адресатам.

Замечание: Если необходимо обеспечить поддержку стека протоколов TCP/IP, то необходимо дополнительно установить соответствующий дополнительный программный продукт.

2.5.1.1. Логические ID узлов

Каждому узлу присваивается уникальный логический идентификатор (ID) узла, который представляет собой положительное целое число. Операционная система использует эти идентификаторы, чтобы взаимодействовать с другими узлами, связанными сетью.

Следует присваивать логические ID узлов в непрерывной последовательности, начиная с 1 (например, 1, 2, 3, 4, 5, ...). Для этого имеется две причины:

1) механизм лицензирования основывается на общем числе логических узлов в сети. Пятиузловая сетевая лицензия позволяет взаимодействовать узлам с 1-го по 5-й. Узел 6 будет изолироваться и игнорироваться, даже если один из узлов с 1-го по 5-й был опущен;

2) несколько важных утилит (например, nameloc) пытаются взаимодействовать с каждым узлом сети. Они делают это с помощью посылки сообщений, начиная с узла 1 и до узла n, где n - число установленных сетевых лицензий.

2.5.1.2. Логические ID сетей

Узлы связаны через одну или более физических сетей. Каждая сеть представляет собой отдельную коммуникационную связь. Сетям, подобно узлам, присваиваются логические ID сети, которые должны быть уникальными.

Единственная сеть (наиболее общий случай) по умолчанию имеет логический ID сети, равный 1. Если у вас есть вторая сеть, присвойте ей ID сети, равный 2.

2.5.1.3. Физические ID узлов

Хотя процессы работают с логическими ID узлов, сами сетевые платы взаимодействуют, используя физические ID узлов. Эти физические ID обычно содержатся непосредственно в самой сетевой плате. Формат физического ID зависит от типа сети (например, Ethernet, Arcnet, Token Ring) и невидим для большинства приложений. Физические ID при установке сети должны быть уникальными в пределах сети, но они не должны быть уникальными для всех сетей.

2.5.1.4. Серверы начальной загрузки и загружаемые узлы

Локальная сеть КПДА.00002-01, по существу, представляет собой сеть с равноправными узлами, которую устанавливают в два этапа:

1) выбирают конфигурацию сервера начальной загрузки;

2) выбирают конфигурацию загружаемых узлов.

Любой узел, имеющий жесткий диск, может загружаться со своего собственного жесткого диска. Сервер начальной загрузки - это просто узел, который обслуживает запросы загрузки от других узлов в сети. Обычно в качестве сервера загрузки выбирается узел, который редко перезагружается.

Когда узел выполняет начальную загрузку через сеть, то он получает образ ОС от сервера начальной загрузки. Этот образ начальной загрузки загружается в память выполняющей начальную загрузку машины, которая обычно наследует от загружаемого образа начальной загрузки следующие параметры:

1) время;

2) глобальный список символьных имен процессов (см. nameloc в документе «Описание программы»);

3) лицензионные возможности.

При необходимости, можно устанавливать сеть с несколькими серверами начальной загрузки, чтобы обеспечить отказоустойчивость (в случае, если

первичный сервер начальной загрузки станет неработоспособен) и распределить ресурсы начальной загрузки так, чтобы избежать образования узких мест.

Выполнение начальной загрузки сети

Сетевые платы в загружающемся узле выдают запросы начальной загрузки одним из двух способов:

1) посылают запрос начальной загрузки в заданный сервер начальной загрузки

или

2) широковещательно посылают запрос начальной загрузки ко всем узлам и ожидают, что сервер начальной загрузки ответит на запрос.

Плата Arcnet, имеющая энергонезависимое ОЗУ, в котором можно хранить ID узла сервера, использует точный метод начальной загрузки. Большинство других сетевых плат, таких как Ethernet и Token Ring, не имеют средств для хранения физического ID узла сервера, поэтому они должны использовать широковещательный метод.

1.9.1.5. Создание "мостов" между локальными сетями КПДА.00002-01

Любой узел сети КПДА.00002-01 может действовать как мост между двумя различными сетями КПДА.00002-01, если они обе являются сетями стандарта IEEE 802. Например, создание "моста" КПДА.00002-01 позволит присоединить сеть Ethernet к сети Token Ring или к сети FDDI. Это возможно, потому что изделие использует один и тот же формат пакета и протокол на всех сетях, основанных на стандарте IEEE 802. Для этого необходимо присоединить две различные сети к одному и тому же узлу. Узел может быть сервером начальной загрузки или просто загружающейся машиной.

Замечание: изделие не может создать "мост" к сети Arcnet.

2.5.2. Планирование сети

2.5.2.1. Выбор количества сетей

При установке небольшой сети, вероятно, потребуется только один сервер начальной загрузки (обычно узел 1). При установке большой сети - которая будет

охватывать несколько отделов - следует устанавливать сервер начальной загрузки для каждого отдела. Пользователи каждого отдела смогут иметь доступ к файлам (при условии разрешения доступа к файлам) в других отделах.

Такая установка создает отказоустойчивую среду - отказ в одном отделе не помешает другому отделу выполнить начальную загрузку. Для большого числа узлов это должно также уменьшить вероятность "узкого места" при начальной загрузке, когда служащие в офисе начинают свою работу с загрузки.

2.5.2.2. Несколько сетевых связей

Наличие больше чем одной сетевой связи на компьютере обеспечивает отказоустойчивость и увеличивает производительность.

Обмен информацией через сети

Узел может обмениваться информацией одновременно со многими сетями. Рассмотрите диаграмму, представленную на рис. 31, где четыре компьютера связываются через две сети.

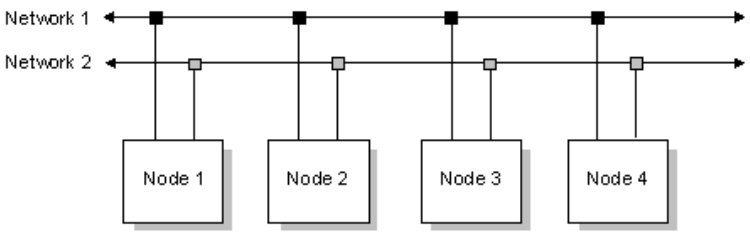


Рис. 31. Использование нескольких физических сетей.

В этой отказоустойчивой конфигурации, каждый из четырех узлов может напрямую обмениваться информацией с любым другим узлом через сеть 1 или сеть 2.

Сравните это с вариантом, представленным на рис. 32, где узлы 1 и 3 подключены к различным сетям, а узел 2 является общим для обеих сетей. Узлы 1 и 3 напрямую не соединяются друг с другом, но они оба могут обращаться к узлу 2.

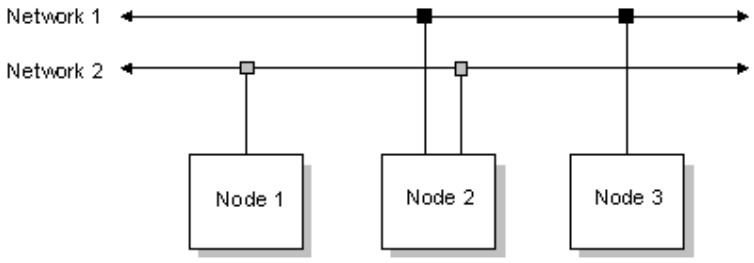


Рис. 32. Создание моста между сетями.

Если узел 2 был соединен с двумя сетями стандарта IEEE 802, то сетевой "мост" КПДА.00002-01 будет автоматически связывать две сети через узел 2.

2.5.2.3. Примеры построения сетей

На схеме, представленной на рис. 33, показана сеть, состоящая из пяти узлов. Один из узлов - сервер начальной загрузки, а три - узлы, которые будут загружаться и получать файлы с сервера начальной загрузки. Узел 5 загружается со своего собственного жесткого диска.

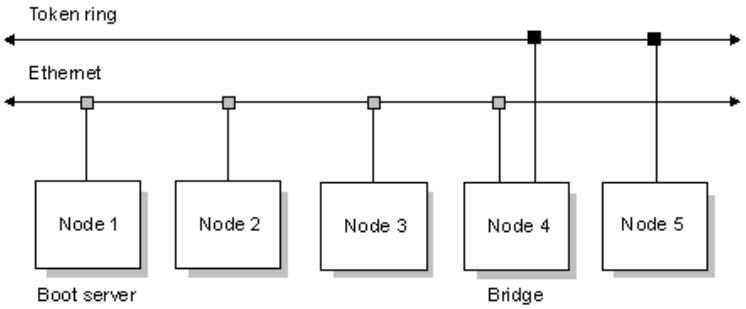


Рис. 33. Пример схемы сети.

Каждому узлу присваиваются логические ID; серверу начальной загрузки - узел 1. Узел 4 будет действовать как "мост" между сетью Ethernet (логический ID сети - 1) и сетью Token Ring (логический ID сети - 2).

Замечание: Невозможно производить начальную загрузку через "мост". В предыдущей диаграмме, например, узел 5 не может быть загружен с узла 1.

Сетевые платы и драйверы

Установка сети обычно включает установку сетевой платы в компьютер и ее соединение кабелем с другими компьютерами, которые имеют тот же самый тип сетевой платы.

Кроме сетевой платы, необходим поддерживающий ее сетевой драйвер. Имена драйверов начинаются с префикса "Net." (их описание см. в документе «Описание программы»).

Платы от различных поставщиков для одних и тех же сетевых сред могут потребовать свои собственные специальные драйверы. Но часто драйвер может поддерживать платы более чем одной компании, если платы поддерживают аналогичный аппаратный интерфейс.

Замечание: Для получения информации относительно аппаратной установки/конфигурации сетевой платы и физического соединения между сетевыми платами, смотрите документы, поставляемые с платой.

Определение физических ID узлов

Метод, используемый для определения физического ID узла сетевой платы зависит от типа имеющейся платы. Обычно производители помещают на упаковке сетевых адаптеров этикетку с адресом платы. Иногда физический ID узла сетевой платы задается переключателями или перемычками на плате.

Ethernet и Token Ring

Каждая плата Ethernet или Token Ring поставляется со своим собственным физическим ID узла, встроенным в плату. Этот ID, который является совершенно уникальным, имеет длину 48 битов и соответствует стандарту IEEE 802.

Для плат Ethernet можно найти значение ID где-нибудь на самой плате или на упаковке. Формат адреса ID может меняться. Например, все ID, указанные ниже, одинаковые:

```
0000c0 4a9330
0000c04a9330
00 00 c0 4a 93 30
00:00:c0:4a:93:30
0000 c04a 9330
```

Arcnet

Каждая плата Arcnet требует запрограммировать физический ID узла на плате. В зависимости от изготовителя, это делается посредством DIP-переключателя или записью в энергонезависимое ОЗУ с помощью интерфейсного меню во время начальной загрузки.

Физический ID имеет длину 8 битов. Нельзя использовать физический ID, равный 0, который зарезервирован. При выборе физического ID узла рекомендуется для единственной сети Arcnet установить физический ID таким же, как логический ID узла.

Присвоение логических ID узлам и сетям

Так как процессы используют логические ID, Администратор сети (Net) должен установить соответствие между логическими ID и физическими ID, используемыми аппаратными средствами, - карту сети. Эта карта сети задается в файле /etc/config/netmap. Каждая строка в этом файле определяет соответствие для одного узла: первыми в строке задаются логические ID узла и сети, за ними следует физический ID узла. Например, следующая строка устанавливает соответствие логического узла 8 в логической сети 1 48-битному физическому ID (MAC-адресу) 0000c4a9330:

```
8   1 0000c 4a9330
```

Можно разделить логический и физический ID символами пробела или табуляции. Логический ID - в десятичный, а физический ID - 16-ричный, если ему не предшествует "t", - в этом случае он также десятичный.

Например, в случае использования Arcnet более удобно представлять физический ID узла в десятичном виде:

```
15   1   t15
```

Физический ID не может превышать 48 битов. Требуемое количество битов определяется сетевым аппаратным обеспечением.

2.5.3. Выбор конфигурации сервера начальной загрузки

Если одиночный сервер установлен, можно устанавливать загружающиеся узлы. Если требуются дополнительные серверы начальной загрузки, следует обратиться к подразделу 1.9.7 «Несколько серверов начальной загрузки».

Замечание: Эта процедура предполагает, что базовая установка изделия на жестком диске ЭВМ уже выполнена. Следовательно, ЭВМ уже сконфигурирована для загрузки со своего жесткого диска

Превращение узла КПДА.00002-01 в сервер начальной загрузки включает в себя следующие шаги:

- установка сетевой платы (плат);
- установка ваших сетевых лицензий;
- запуск Администратора сети и сетевого драйвера (драйверов);
- запуск утилиты nameloc;
- запуск утилиты netboot;
- модификация файла sysinit.узел;
- модификация файла netmap;
- модификация файла netboot.

2.5.3.1. Шаг 1. Установка сетевой платы (плат)

Выключите питание ЭВМ, затем установите сетевую плату согласно инструкциям, прилагаемым к плате. Если возможно, определите физический ID платы. Когда закончите, загрузите компьютер.

Замечание: Если физический ID узла при установке платы не найден, или если его не было на плате, необходимо запустить (после запуска сетевых драйверов) утилиту netmap без каких-либо опций, и она отобразит физический ID узла.

Установка нескольких плат

При установке двух одинаковых плат в одной и той же машине, нельзя полагаться на автоматическое обнаружение по умолчанию портов ввода/вывода и т.п. Необходимо проверить уникальность установочных параметров – часть этой информации можно определить, используя опции командной строки в сетевом

драйвере. Платы следует сконфигурировать таким образом, чтобы они не имели аппаратных конфликтов.

2.5.3.2. Шаг 2. Установка сетевых лицензий

Для каждого узла сети нужна сетевая лицензия. Можно проверить, сколько лицензий было установлено, используя утилиту `licinfo`.

2.5.3.3. Шаг 3. Запуск Администратора сети и сетевого драйвера(ов)

При сообщении install, что данный узел должен быть сервером начальной загрузки, должны быть запущены соответствующие сетевые сервисные процессы. Этот шаг допускается пропускать.

Запуск Администратора сети и одного сетевого драйвера

Для доступа к логической сети 1 должны быть запущены Администратор сети (Net) и сетевой драйвер.

Имена для всех сетевых драйверов имеют шаблон Net.xxx, где xxx представляет собой соответствующий драйвер. Например, если есть плата Ethernet, совместимая с NE2000, то ввод будет похож на следующий:

```
Net &
Net.ether1000 &
```

Чтобы выяснить, какие сетевые драйверы доступны в вашей системе, нужно ввести команду:

```
ls /bin/Net.*
```

Запуск двух или более сетевых драйверов

Если установлено две сетевые платы, необходимо запустить драйвер для второй сети.

По умолчанию все сетевые драйверы связаны с логическим ID сети, равным 1. При создании нескольких сетей, необходимо задать опцию -l в командной строке запуска драйвера, чтобы иметь драйвер, связанный с сетью иной, чем логическая сеть 1.

Например, следующие три команды в файле sysinit.узел сервера начальной загрузки запустят Администратор сети, драйвер Ethernet для логической сети 1, а драйвер Token Ring для логической сети 2:

```
Net &
Net.ether1000 &
Net.tr164a -l 2 &
```

Нужно обратить внимание, что Администратор сети должен запускаться с опцией -d всякий раз, когда запускаются три или большее число сетевых драйверов.

Эта команда заставляет nettrap просканировать машину, на которой запущена, и найти сетевые платы:

```
nettrap
```

Замечание: Утилита nettrap считывает и записывает порты ввода/вывода и память. Это может вызвать неумышленные побочные эффекты в других аппаратных средствах, размещенных в проверяемой памяти и адресах ввода/вывода. Если система установлена для управления внешним оборудованием, необходимо отключить эти устройства перед выполнением nettrap.

Утилита nettrap также может быть использована для запуска Net вместе с соответствующими сетевыми драйверами (и netmap -f):

```
nettrap start
```

2.5.3.4. Шаг 4. Запуск утилиты nameloc

Утилита nameloc выполняется в фоновом режиме и обеспечивает в масштабах сети систему идентификации имен для всех процессов, выполняющихся под ОС. Для обеспечения работы лицензионных продуктов, утилита nameloc должна быть запущена, как минимум, на одной машине в сети, даже если это *единственная* машина в автономной "сети":

```
nameloc &
```

Запустившись, nameloc опрашивает каждый узел в сети для составления списка глобальных символических имен процессов. Для получения дополнительной информации смотрите описание утилиты nameloc в книге «*Описание программы*».

Замечание: ЭВМ, на которой выполняется программа nameloc, *должна* быть способна взаимодействовать с любой другой машиной в сети. Необходимо убедиться, что компьютер имеет *полный* файл netmap. Если nameloc выполняется на машине, не имеющей полного доступа в сеть, могут возникнуть различные проблемы – неверная информация о лицензиях, неполный список глобальных имен, неправильная работа сетевых утилит и т.п.

2.5.3.5. Шаг 5. Запуск утилиты netboot

Когда загружающийся узел запускается, его ПЗУ начальной загрузки посылает запрос о загрузке на сервер начальной загрузки или он пересылает запрос о загрузке на любой сервер начальной загрузки, сконфигурированный для ответа. Когда Администратор сети на сервере начальной загрузки получает запрос о загрузке, он передает этот запрос утилите netboot. Чтобы запустить netboot, необходимо выполнить команду:

```
netboot &
```

При ответе на запрос о начальной загрузке утилита netboot обращается к файлу /etc/config/netboot, чтобы определить, какой из файлов построения может использоваться, чтобы сгенерировать образ ОС для запрашивающего узла.

Замечание: Выполнение netboot с опцией **-v** ("многословный" режим) может быть полезно для поиска неисправностей. Многократное применение опции **v** увеличит уровень информированности - при этом можно направлять вывод в файл или вывести его на своей собственной консоли.

Можно запустить утилиту netboot с опцией **-a**, чтобы автоматизировать создание карты сети (смотри Шаг 7):

```
netboot -a &
```

В том случае, если файл /etc/config/netmap не содержит соответствующих установок для запрашивающего узла, то netboot автоматически запишет необходимую строку карты сети в файл /etc/config/netmap, используя ближайший доступный логический ID узла и логический ID сети, и запрашивающий узел подключится.

2.5.3.6. Шаг 6. Модификация файла sysinit.узел

Теперь, когда запущены все требуемые сетевые драйверы и сервисные процессы, узел может функционировать как сервер начальной загрузки. Если добавить команды запуска сети в файл sysinit.*узел* сервера начальной загрузки, они будут автоматически запускаться всякий раз, когда узел перезагружается.

Скопируйте файл sysinit.*узел* сервера начальной загрузки (например, sysinit.1) в altsysinit прежде, чем продолжить:

```
cp /etc/config/sysinit.1 /etc/config/altsysinit
```

Это сохранит альтернативную копию файла инициализации сервера начальной загрузки. Теперь необходимо добавить требуемые записи в /etc/config/sysinit.*узел*. Для сервера начальной загрузки требуются записи, похожие на следующую:

```
Net &
Net.ether1000 &
nameloc &
netboot &
```

2.5.3.7. Шаг 7. Модификация файла netmap

Файл /etc/config/netmap является по умолчанию файлом карты сети ID узлов и сетей, и используется утилитами netboot и netmap. Этот файл устанавливает соответствие физического ID узла, логического ID узла и логического ID сети для каждого узла.

Используемый файл, должен содержать логические ID узлов и сетей для сервера начальной загрузки и для всех сетевых узлов, взаимодействующих с сервером начальной загрузки. Соответствующие записи в файле выглядят примерно так, как показано в таблице 2.5.

Таблица 2.5. Структура записей в карте сети.

ID узла	**ID сети**	**Адрес сетевой платы**
1	1	*Физический_адрес_загрузочного_сервера*
2	1	*Физический_адрес_узла2*

ID узла	ID сети	Адрес сетевой платы
3	1	*Физический_адрес_узла3*
4	1	*Физический_адрес_узла4*

Нужно обратить внимание, что когда имеется единственная сеть, логический ID сети обычно равен "1". Все узлы сети должны иметь свои собственные уникальные логические ID узлов. Следует выбирать логический ID узла, равный 1, для первого (первичного) сервера начальной загрузки. Для узла, который взаимодействует с двумя или более сетями, один тот же логический ID узла будет иметь две записи, каждая из которых будет иметь свой уникальный логический сетевой идентификатор.

Файл netmap можно редактировать вручную, или, если запущена утилита netboot с опцией **-a**, то система будет автоматически записывать данные об узле в файл netmap, при включении любого узла. Узлу, который будет включен первым, будет присвоен следующий свободный логический ID узла, правильный логический ID сети и правильный физический адрес. Если задается опция **-A**, то новому узлу будет присваиваться наименьший свободный логический ID узла и правильный физический адрес.

Если имеются узлы, загружающиеся с их собственных жестких дисков, то можно сгенерировать модифицированный "главный" файл /etc/config/netmap на узле 1 следующим образом:

1. На всех машинах, за исключением узла 1, нужно создать локальный файл netmap, содержащий карту сети логических и физических ID для данного узла и узла 1.

2. На всех узлах, за исключением узла 1, надо выдать следующую команду, чтобы откорректировать сохраняемую в памяти карту сети на узле 1:

```
sin -n1
```

3. Чтобы сохранить из памяти карту сети на диске узла 1, нужно набрать на клавиатуре:

```
netmap > /etc/config/netmap
```

Чтобы перенести карту сети на другие узлы, можно скопировать главный файл с узла 1. Например, чтобы скопировать главный файл на узел 2, нужно набрать на клавиатуре:

```
cp //1/etc/config/netmap //2/etc/config/netmap
```

Карта сети для нескольких сетей

Для определения узла, имеющего более одной сетевой платы, необходимо добавить одну запись, чтобы соответствовать каждой сети. Логические сетевые ID должны быть уникальны. Например, если сеть имела пятый узел, соединенный с двумя различными сетями, надо добавить еще две записи в файл netmap (см. таблицу 2.6).

Таблица 2.6. Записи для 2 сетевых плат на одной ЭВМ

ID узла	ID сети	Адрес сетевой платы
5	1	*Физический_адрес_платы1_узла5*
5	2	*Физический_адрес_платы2_узла5*

Для получения дополнительной информации о файле netmap смотрите утилиту netmap в книге «Описание программы». Примеры, описывающие карту сети для нескольких сетей, найдете в подразделе «Примеры сетей» в этом разделе.

Замечание: Каждый раз при редактировании файла netmap следует также откорректировать сохраняемые в памяти сетевые таблицы:

```
netmap -f
```

2.5.3.8. Шаг 8. Модификация файла netboot

Когда сервер начальной загрузки получает запрос о начальной загрузке от узла, Администратор сети отправляет запрос вместе с физическим ID узла и логическим ID сети утилите `netboot`.

Имея эту информацию, утилита `netboot` обращается к файлу карты сети (/etc/config/netmap), чтобы определять логический идентификатор узла (загружаемой машины). Затем утилита обращается к файлу /etc/config/netboot, чтобы определить,

какой файл построения можно использовать, чтобы сгенерировать образ ОС для узла.

Файл /etc/config/netboot на сервере начальной загрузки может содержать строку, аналогичную следующей:

* f=build/ws.ether1000

Запись f= позволяет задавать файл построения для клиентского узла. Если некоторое количество узлов оборудовано одним и тем же видом сетевой платы, для этих узлов может быть использована одна запись.

Если сервер начальной загрузки будет обслуживать запросы о начальной загрузке от двух или более различных узлов и/или сетей, то можно задавать различные заказные файлы построения для некоторых из узлов.

Например, имеется восьмиузловая сеть Ethernet, совместимая с NE2000, и эти узлы загружают из netboot запись f=build/ws.ether1000. Если добавить еще три узла, которые будут загружаться из того же самого сервера, можно изменить /etc/config/netboot таким образом, что он будет выглядеть примерно так:

```
9     f=build/ether2100.node9
10    f=build/ws.tr16a
11    f=build/mybuild.node11
*     f=build/ws.ether1000
```

В вышеуказанном примере узлы 9, 10 и 11 используют заказные файлы построения, а все другие узлы используют по умолчанию сетевой файл построения ws.ether1000.

***Замечание*:** Позиция строки с * важна! Необходимо удостовериться, что строка, начинающаяся с *, является *последней* строкой в /etc/config/netboot, поскольку обработка файла прекратится, когда будет обнаружено первое соответствие.

Установка для широковещательной загрузки

При широковещательной загрузке с ПЗУ утилита netboot должна знать, в качестве какого сервера она выступает по отношению к каждому узлу (первичного,

вторичного или третичного). Утилита выясняет это, считывая необязательный параметр режима сервера в файле netboot.

Общий синтаксис записей netboot следующий:

логич_ID_узла f=*файл_постр.* | F=*файл_образа* [*режим_сервера*]

где *режим_сервера* определяет роль, которую утилита netboot должна играть по отношению к узлу. Если *режим_сервера* не задан, то netboot будет действовать для данного узла как первичный сервер начальной загрузки.

Замечание: Опция F=*файл_образа* позволяет указать имя предварительно построенного заказного образа ОС, который будет загружен в узел из сети. По умолчанию, КПДА.00002-01 строит образ "на лету", используя заданный файл построения. Дополнительную информацию о передаче образа ОС загружающемуся узлу, вместо именованного, читайте в подразделе «Загрузка узла из сети».

Возможные значения для *режим_сервера*:

A первичный сервер (по умолчанию)

B вторичный сервер

C третичный сервер

В следующем примере задается, что netboot должен быть первичным сервером начальной загрузки для всех узлов, за исключением узла 4, для которого он будет вторичным сервером начальной загрузки:

```
4   f=build/ws.tr16a       B
*   f=build/ws.ether1000   A
```

Дополнительную информацию о выполнения netboot на нескольких серверах начальной загрузки читайте в подразделе «Несколько серверов начальной загрузки».

2.5.4. Загрузка узла из сети КПДА.00002-01

Когда узел, загружаемый из сети КПДА.00002-01, включается, то сначала выполняется программа начальной загрузки изделия, находящаяся в ПЗУ на сетевой плате машины. При этом либо устанавливается связь с конкретным сервером

начальной загрузки, либо делается широковещательный запрос о начальной загрузке.

Процесс netboot на ответившем сервере начальной загрузки генерирует и загружает образ начальной загрузки в ОЗУ сетевой платы загружающегося узла. Затем запускается программа sinit и начнется запуск сервисных средств, определенных в файле sysinit.*узел* для данного узла.

Конфигурирование узла, загружаемого из сети КПДА.00002-01, включает следующие шаги:

- установка ПЗУ начальной загрузки КПДА.00002-01;
- установка сетевой платы;
- создание файла построения;
- модификация файла sysinit.*узел*;
- загрузка узла.

2.5.5. Загрузка узла со своего собственного жесткого диска

Сеть может включать узел, который будет загружаться со своего собственного жесткого диска. Если это так, то прежде чем загружать узел, нужно сделать следующее:

- установить сетевую плату, как это объяснено ранее;
- отредактировать файл построения узла;
- скопировать главный файл netmap на узел;
- установить или перенести лицензию на жесткий диск узла.

2.5.5.1. Редактирование файла построения узла

Первые записи в файле построения запускают Администратор процессов (например, sys/Proc32) и устанавливают опции, которые будет передаваться Proc32. Опция **-l** для Proc32 присваивает логический ID узла.

При установке изделия на жесткий диск данного узла, предполагалось, что логический ID узла равен 1. Следует изменить логический ID узла, задаваемый опцией **-l**, чтобы он соответствовал суффиксу *узел* файла sysinit.*узел* узла.

Для получения дополнительной информации относительно построения пользовательского образа для узла смотрите раздел "Сетевые образы" в разделе «Построение образа ОС».

2.5.5.2. Копирование файла netmap в узел

Машина, загружаемая со своего собственного жесткого диска, нуждается в файле netmap, содержащем карту сети, устанавливающей соответствие между логическими ID и физическими ID, для *всех* узлов, с которыми эта машина должна взаимодействовать.

Можно использовать модифицированный "главный" файл /etc/config/netmap сервера начальной загрузки, как это описано в Шаге 7 раздела "Выбор конфигурации сервера начальной загрузки".

2.5.5.3. Установка или перенос лицензий

Чтобы узел надлежащим образом загрузился с собственного жесткого диска, то в файле /.licenses должна иметься лицензия. При наличии лицензионного сертификата, можно добавить номер лицензии в файл /.licenses. При этом необходимо активизировать новые лицензии (используя license -r).

2.5.6. Диагностика сети

Администратор сети при выполнении поддерживает кольцевой буфер значимых сетевых событий. Можно выполнить утилиту netinfo, чтобы отобразить самый последний буфер. Вывод включает время события в сети, номер связанного с ним узла, числовой код события и краткое описание этого события. Дополнительная информация относительно этой утилиты содержится в книге «Описание программы».

2.6. Настройка учетных записей пользователей

В данном подразделе содержится описание штатных средств управления учетными записями пользователей (пользовательскими бюджетами). Рекомендуется дополнительно изучить документ «Руководство администратора безопасности».

2.6.1. Запуск сеанса пользователя

Утилита login управляет доступом пользователя ко всем ресурсам ОСРВ и системным функциям, когда пользователь делает попытку входа в систему. Пользователь с терминала /dev/ser1 может получить доступ в систему, введя правильные ответы для следующей команды:

```
on -t /dev/ser1 /bin/login
```

Вызываемая без опций, утилита login запрашивает у пользователя имя и пароль. Можно автоматизировать процесс входа в систему с помощью утилиты tinit. Чтобы tinit автоматически вызывала login при запуске системы, следует добавить следующую запись в файл sysinit.*узел* узла:

```
tinit -t /dev/ser1 &
```

2.6.1.1. Инициализация командного интерпретатора

После успешного входа пользователя в систему, на терминале запускается командный интерпретатор (shell). Запускаемый при входе в систему командный интерпретатор (например, /bin/sh) выполняет сначала файл /etc/profile. Посредством этого файла могут быть установлены локальные соглашения.

Затем выполняется специальный пользовательский файл установки начальных параметров, обычно это $HOME/.profile. В этот файл можно добавить любые нужные вам экспортируемые переменные окружения. Например, при необходимости сохранения настроек запускаемого при входе в систему командного интерпретатора всякий раз, когда процесс запускает новый командный

интерпретатор, нужно определить и экспортировать переменную окружения ENV в файле .profile. Например:

```
export ENV=$HOME/.kshrc
```

Эта запись гарантирует, что каждый раз при запуске нового командного интерпретатора будет выполняться файл $HOME/.kshrc. Чтобы отобразить значения всех экспортируемых переменных, напечатайте export -p.

2.6.1.2. Установка переменных окружения

Когда пользователь начинает сеанс, то начинает работу программа запускаемого при входе в систему командного интерпретатора (например, /bin/sh). Смотрите описание утилит login и sh в книге «Описание программы», если хотите получить дополнительную информацию о переменных командного интерпретатора и о функционировании по умолчанию.

Дополнительные установочные параметры берутся из пользовательского файла .profile, если он существует. Командный интерпретатор может автоматически добавить некоторые из требуемых значений в файл .profile пользователя. Например, файл .profile пользователя может содержать установочные параметры для следующих переменных окружения:

TERM - определяет тип терминала пользователя (перечень допустимых значений смотрите в разделе»Установка терминалов»)

TMPDIR - задает местоположение временного рабочего пространства, которое может быть использовано утилитами и программами (например, /tmp)

TZ - устанавливает зону времени (например, TZ=EST5EDT4)

Команда set может устанавливать (+) или не устанавливать (-) эти опции командного интерпретатора. Чтобы отобразить список переменных, которые могут быть определены для командного интерпретатора, нужно напечатать set после приглашения командного интерпретатора. Чтобы увидеть значение одной

переменной, необходимо ввести echo $*имя_переменной* после приглашения командного интерпретатора (например, echo $WORKDIR). Для получения дополнительной информации об этих командах командного интерпретатора смотрите утилиту sh в документе «Описание программы».

2.6.2. Управление доступом пользователя к ресурсам

Изделие предоставляет механизмы дискреционного разграничения доступа к ресурсам и критическим системным функциям. Эти механизмы базируются на способности системы определять конкретного пользователя.

Примечание. Защищенная версия изделия дополнительно предоставляет механизмы мандатного разграничения доступа к ресурсам (см. «Руководство администратора безопасности»).

Когда система загружается в первый раз (сразу после начальной установки), автоматически происходит регистрирация пользователя в ней как суперпользователя (root). Это имя пользователя имеет доступ к любой части системы и не требует пароля.

Чтобы предотвратить несанкционированный доступ пользователей в систему, для пользователя root следует задать пароль. Можно создать новую учетную запись пользователя, чтобы использовать ее впоследствии для ежедневной работы. Утилита passwd позволяет изменять пароль входа в систему; можно использовать утилиту, чтобы создавать новую учетную запись для любого пользователя.

2.6.2.1. Утилиты управления доступом

Утилиты управления доступом в КПДА.00002-01:

- login - вход в систему;
- su - временное получение прав другого пользователя;
- passwd - создание и ведение учетных записей и изменения паролей пользователей.

Утилита login

Утилита login может быть запущена с помощью tinit на всех устройствах tty. Утилита запрашивает имя пользователя и пароль и сверяет их с базой данных пользователей /etc/passwd перед предоставлением доступа к системе. Если комбинация имя пользователя/пароль неправильная, login выдает сообщение об этом и завершается. Если пользователь вводит правильную комбинацию, то login запускает командный интерпретатор и загружает окружение для данного пользователя.

Утилита su

Утилита su позволяет пользователю временно иметь привилегии другого пользователя. Если идентификатор (ID) пользователя не введен в командную строку при запуске su, утилита запрашивает пароль, который должен соответствовать паролю суперпользователя. Ввод правильной комбинации доступа заставляет su обратиться в базу данных пользователя /etc/passwd и запустить командный интерпретатор, который имеет все права и привилегии соответствующего пользователя. Выход из командного интерпретатора возвратит вас к своему настоящему ID пользователя.

Утилита passwd

Утилита passwd может использоваться для изменения пароля или ввода нового пользователя в систему. Любой пользователь может изменить свой собственный пароль, но только суперпользователь (root) имеет право создавать или изменять другие пароли. В обоих случаях утилита passwd блокирует доступ к файлу паролей пользователей /etc/passwd, чтобы предотвратить любые другие попытки доступа к файлу во время его модификации.

Пользователь, имеющий эффективный идентификатор 0 (т.е. root) может использовать утилиту passwd для создания новых учетных записей (бюджетов) пользователей, а так же для изменения пароля любого пользователя. Остальные

пользователи имею возможность использовать утилиту passwd только для изменения собственных паролей.

Перед началом создания ID пользователя, имеется возможность изменить режим работы passwd. Параметры установки в файле /etc/default/passwd определяют, какой локальный алгоритм будет осуществлять утилита passwd, например, обязательность паролей, диапазон изменений ID пользователя и т.п. Для получения дополнительной информации об опциях, которые можно задавать, смотрите утилиту passwd в документе «Описание программы».

Суперпользователь может добавить новую учетную запись, вызвав passwd с новым *именем пользователя*. Утилита запрашивает информацию, которая будет управлять окружением нового пользователя, включая права доступа. Учетная запись включает в себя ID группы, к которой принадлежит пользователь, и атрибуты доступа файла.

Например, чтобы создать новую учетную запись с именем пользователя jsmith, суперпользователь должен ввести:

```
passwd jsmith
```

По умолчанию, passwd будет затем запрашивать следующую информацию:

- ID пользователя;
- ID группы, к которой принадлежит пользователь;
- реальное имя пользователя;
- исходный (домашний) каталог пользователя;
- начальная команда (например, /bin/sh);
- начальный пароль.

Номер ID пользователя

Обычно учетные записи пользователя имеют номер ID пользователя >= 100. Номера ID пользователя ниже 100 часто используются процессами системы. Диапазон для ID пользователя устанавливается в файле /etc/default/passwd.

ID группы, к которой принадлежит пользователь

Если ID группы не существует, система выведет напоминание, что нужно добавить группу в файл /etc/group.

Файл /etc/group предоставляет привилегии всем пользователям, являющимся членами определенной группы. Например, отделу "Финансы" разрешен доступ к финансовым записям компании, а группе "Поддержка Заказчика" - нет. Можно установить ID группы "Финансы" с требуемыми правами доступа и затем назначать пользователей в этой группе согласно привилегиям, которых они требуют.

Файл /etc/group содержит список пользователей в группе. Каждая строка имеет следующий общий синтаксис:

имя_группы::*ID_группы*:*польз-ль1*, *польз-ль2*,..., *польз-льN*

Например, следующий ввод определяет группу "maestri" с ID группы "123" и членами "alvivaldi", "gfhandel", "gptelemann" и "jsbach":

```
maestri::123:alvivaldi, gfhandel, gptelemann, jsbach
```

Сначала следует добавлять группы, которые не имеют никаких пользователей, как в следующем примере:

```
docs::0:
techies::1:
support::2:
marketing::3:
finance::4:
```

Когда пользователь входит в систему, значение в базе данных паролей /etc/passwd определит ID группы, который был присвоен пользователю (если

имеется). Пользователям, которые принадлежат к этой группе, позволено подключаться к привилегиям этой группы всякий раз, когда они используют команду newgrp.

Исходный каталог, начальная команда и начальный пароль

Исходный каталог, начальная команда и начальный пароль - значения необязательные.Если эти значения не определены, то будут использоваться следующие значения по умолчанию:

исходный каталог - используется каталог /home/имя_пользователя (создается, если он не существует);

начальная команда - команда запуска оболочки по умолчанию (/bin/sh), выполняемая немедленно после входа пользователя в систему

пароль - учетная запись пользователя создается без пароля

2.6.2.2. Удаление учетной записи пользователя

Чтобы удалить учетную запись, необходимо удалить *все*, что указано ниже:

- запись в базе данных пользователя из файла /etc/passwd;
- запись с именем пользователя из соответствующей ID группы принадлежности в файле /etc/group;
- соответствующую закодированную запись пароля входа в систему из файла /etc/shadow;
- (необязательно) исходный каталог пользователя;
- (необязательно) "почтовый ящик" пользователя (то есть /usr/spool/mail/*имя_пользователя*).

2.6.2.3. Подробные сведения о ID пользователей и групп

После регистрации в системе пользователю присваиваются ID пользователя и ID группы.

Они известны как *реальный* ID пользователя и *реальный* ID группы.

ID пользователя должен быть уникален – два пользователя не должны совместно использовать одинаковые ID. Это правило соблюдается утилитой passwd, однако суперпользователь может обойти это правило, редактируя непосредственно файл пароля.

ID группы позволяет нескольким пользователям объединиться в группу. Этот механизм позволяет группе пользователей совместно использовать общие ресурсы, не делая их доступными для всех остальных пользователей. Для получения дополнительной информации о файле /etc/group и его содержимом смотрите утилиту newgrp в книге «Описание программы».

Эффективные идентификаторы пользователя и группы

Процессы имеют два класса ID пользователя и группы: *реальные* и *эффективные*. Когда процесс создается, то он автоматически наследует эти четыре ID от своего родителя (который является обычно запускаемым при входе в систему командным интерпретатором):

- реальный ID пользователя;
- реальный ID группы;
- эффективный ID пользователя;
- эффективный ID группы.

Эффективные ID пользователя и группы используются для проверки прав доступа. Процесс может изменить эффективные ID пользователя или группы, или оба ID так, чтобы они отличались от своих реальных аналогов. Обычно это делается для того, чтобы получить доступ к ресурсам, которые не доступны для реальных ID пользователя и группы.

Для изменения EUID и EGID процесса непосредственно из программы используются функции *seteuid()* и *setegid()* соответственно. Для достижения этой же цели из командного интерпретатора используются, соответственно, утилиты `su` и `newgrp`. Для изменения атрибутов файлов предназначены утилиты `chmod`, `chown` и `chgrp`.

Для получения большего количества информации о битах setuid и setgid, смотри подраздел «Атрибуты доступа файлов».

Утилита newgrp

Утилита newgrp запускает новый командный интерпретатор с различными реальными и эффективными ID группы. Если она вызывается без параметров, то newgrp подключает ID группы идентифицированного в базе данных пароля для текущего пользователя. База данных пароля управляет группами, к которым может подключиться конкретный пользователь.

2.6.2.4. Атрибуты доступа файлов

Файлы изделия обладают связанным набором атрибутов доступа, называемые *битами режима*. Эти биты режима, в сочетании с *владельцем* и *группой* файла, управляют доступом к файлу.

Когда процесс создает файл, эффективный ID пользователя и эффективный ID группы процесса определяют владельца и группу, которым будет принадлежать файл. Файлу также назначается набор битов режима (это описано в подразделе «Как устанавливаются биты режима»).

Имеются три класса битов режима:

- владелец;
- группа;
- остальные.

Когда процесс пробует обратиться к файлу, Администратор файловой системы обслуживает соответствующий класс битов режима, выполняя несколько сравнений в определенном порядке:

1) эффективный ID пользователя запрашивающего процесса сравнивается с владельцем файла. Если они совпадают, то далее рассматриваются биты режима *владелец*;

2) если сравнение с владельцем файла терпит неудачу, то эффективный ID группы запрашивающего процесса сравнивается с ID группы файла. Если ID этих групп соответствуют, то рассматриваются биты режима *группа*;

3) если и сравнение ID владельца, и сравнение ID группы терпят неудачу, то рассматриваются биты режима *остальные*.

Замечание. Суперпользователь root (uid 0) может обращаться к любому файлу для чтения/записи независимо от атрибутов доступа, и может выполнить любой файл, для которого установлен хотя бы один бит режима разрешения выполнения.

Различие воздействия атрибутов доступа на регулярные файлы и каталоги Следующая таблица показывает:

Таблица 2.7. Воздействие атрибутов доступа на файлы и каталоги

Атрибут	**Воздействие на файл**	**Воздействие на каталог**
r	Чтение файла	Проверка имен файлов в каталоге (например, с помощью ls)
w	Запись в файл	Создание, удаление и переименование файлов внутри каталога, включая подкаталоги
x	Выполнение файла	Поиск каталога. Это означает, что текущий рабочий каталог можно изменить на этот каталог. Кроме того, этот каталог можно включить (например, в *open()*, *stat()* и т.п.).

Замечание. Бит атрибутов доступа - "разрешение выполнения" имеет смысл только для каталогов и регулярных файлов; он не применяется к другим типам файла.

Просмотр атрибутов доступа

Для просмотра атрибутов доступа файла, владельца и группу можно использовать команду ls -l. Следующий вывод ls -l для файла alpha.c показывает, что владелец и любой пользователь из группы techies может читать и записывать в файл, в то время как остальные пользователи могут только читать из него:

```
-rw-rw-r--  1 mplanck   techies   8475 Apr 1  1997 alpha.c
```

В следующем примере вывод ls -ld для каталога /bin показывает, что владелец и члены группы могут читать, записывать и вести поиск в каталоге, в то время как остальные пользователи не имеют разрешения записи и, следовательно, не могут добавлять какие-либо новые файлы в каталог:

```
drwxrwxr-x  2 root     techies   2048 Sep 19  1990 bin
```

Следует обратить внимание, что остальные пользователи все же могут просматривать или выполнять файлы в /bin.

Допускается задавать для каталога разрешение на чтение при запрете на поиск в нем. Например, ls может работать (в зависимости от заданных опций), но пользователь не может обратиться к любому файлу в каталоге или выполнить команду cd в каталоге.

Как устанавливаются биты режима

Атрибуты доступа файла устанавливаются, как комбинация текущей маски umask создающего процесса и режима доступа, запрашиваемого функциями open()/creat().

Маска битов атрибутов доступа - umask - указывает, какие биты атрибутов доступа будут "сброшены", если процесс определяет по умолчанию атрибуты доступа при создании файла, каталога или очереди (смотрите описание утилиты umask в книге «*Описание программы»*).

Биты режима setuid и setgid

Чтобы выполнить определенные функции, пользователь должен иногда выполнить команду, на уровне привилегий более высоком, чем он имеет. Выполнять команду на более высоком уровне позволяют пользователю следующие два бита режима файла:

- setuid (установить ID пользователя) - например, необходимо, чтобы утилита passwd вела себя так, как будто она является root для того, чтобы можно было изменить файлы passwd и shadow в интересах пользователя, который обычно не имеет разрешения записи в эти файлы. (Биты режима для этой команды будут выглядеть следующим образом: -rwsrwxr-x);

- setgid (установить ID группы) - например, необходимо, чтобы команда mailx позволила пользователю временно "принадлежать" группе mail для того, чтобы разрешить пользователю записывать в файлы "почтовых ящиков" (mailboxes) других пользователей, которые обычно имеют разрешения: -rw-rw ---- имя_пользователя mail. (Биты режима для этой команды будут выглядеть следующим образом: -r-xr-sr-x).

Если эти биты установлены, то выполняемый файл будет выполняться с привилегиями его владельца и группы, а не с привилегиями процесса, который вызывает выполнение такого файла. Эти биты режима применяются только к выполняемым регулярным файлам.

Если файл загружается для выполнения и бит режима setuid установлен, то эффективным ID пользователя нового процесса становится таким, как у владельца

файла. Аналогично, если бит режима setgid установлен, то эффективным ID группы процесса становится таким, как у группы файла.

Для устранения потенциального изъяна защиты биты setuid и setgid обнуляются всякий раз, когда изменяется владелец или группа файла. Установка бита setuid защищает от записи пользователями выполняемого файла на диск и, таким образом, изменяя владельца файла на root, пользователь получает неограниченные системные полномочия.

Утилиты passwd, login, su и newgrp - все имеют владельца root и бит setuid; следовательно, эти программы запускаются с правами доступа суперпользователя.

Принято, что root – единственный пользователь, имеющий ID пользователя равный 0, который обеспечивает статус суперпользователя. При управлении правами доступа, пользователь должен гарантировать, что можно доверять установку setuid как root только надежным и совершенно надежным программам. Никаких специальных привилегий при установке setgid как root программа не получает.

Замечание. Так как все программы при установке setuid как root наследуют возможности суперпользователя, необходимо удостовериться, что они *не* имеют всеобщих разрешений записи, чтобы только суперпользователь был способен модифицировать программы.

2.6.2.5. База данных пароля

Эти файлы формируют общую базу данных пароля:

- /etc/passwd
- /etc/group
- /etc/shadow
- /etc/.pwlock

Таблица 2.8 иллюстрирует, какие значения должны иметь атрибуты доступа к этим файлам.

Таблица 2.8. Атрибуты доступа файлов базы данных паролей

Файл:	Владелец:	Группа:	Атрибуты доступа:
/etc/passwd	root	root	rw- r-- r--
/etc/group	root	root	rw- r-- r--
/etc/shadow	root	root	rw- --- ---
/etc/.pwlock	root	root	rw- r-- r--

Файл /etc/passwd

Файл /etc/passwd содержит набор строк следующего формата:

username:*haspw*:*userid*:*group*:*comment*:*homedir*:*shell*

где:

username – имя пользователя для входа в систему;

haspw – если пусто, то пользователь не имеет никакого пароля; если x, то пароль пользователя находится в файле /etc/shadow. Пустое значение или значение x - единственные разрешенные значения;

userid – числовой ID пользователя;

group – числовой ID группы;

comment – поле комментария свободного формата; не должно содержать ":";

homedir – исходный (домашний) каталог этого пользователя (по умолчанию - /);

shell – начальная команда (и аргументы), запускаемая после login (по умолчанию - /bin/sh).

Пример строки из файла /etc/passwd:

```
bubba:x:290:120:Bubba L. Jones:/home/bubba:/bin/sh
```

Файл /etc/group

Файл /etc/group содержит строки в следующем формате:

groupname::*group_ID*:[*user_ID*[,*user_ID*]...]

где:

groupname – имя группы;

group_ID – числовой ID группы;

user_ID – один или более ID пользователя, принадлежащих этой группе.

Пример строки из файла /etc/group:

```
techies::123:bubba,charlie,sue,jake
```

Файл /etc/shadow

Файл /etc/shadow содержит строки в следующем формате:

user_ID:*password*:0:0:0

где:

user_ID – имя пользователя;

password – зашифрованный пароль пользователя.

Пример строки из файла /etc/shadow:

```
bubba:SRjg51|weIJ[H:819388588:0:0
```

Файл /etc/.pwlock

Файл /etc/.pwlock создается утилитой passwd, чтобы указывать другим запускаемым утилитам passwd, что файл пароля в данный момент времени изменяется. Когда утилита passwd завершается, файл блокировки удаляется.

Файл /etc/passwd доступен для чтения любому пользователю. Это обеспечивает стандартными утилитами простой механизм поиска информации о пользователях. Поскольку этот файл читается, зашифрованный пароль здесь не хранится.

Зашифрованный пароль хранится в файле /etc/shadow, который читается только суперпользователем. Это должно предотвращать несанкционированные попытки дешифрования паролей. Для обеспечения безопасности сообщества пользователей, необходимо обеспечить поддержание этих разрешений.

Файла /etc/shadow не существует по умолчанию, т.к. учетные записи пользователей первоначально не имеют паролей.

Обратите внимание, что если какая-либо утилита пробует обратиться к базе данных пароля в то время, когда редактируется файл /etc/passwd, то это может закончиться разрушением базы данных пароля!

Внимание, при удалении пароля пользователя, необходимо также удалить соответствующую запись из файла /etc/shadow.

2.6.2.6. Файлы пароля по умолчанию

По умолчанию файл /etc/passwd, поставленный с системой КПДА.00002-01, содержит следующее:

```
root::0:0::/:/bin/sh
```

Файл /etc/group по умолчанию содержит следующее:

```
root::0:root
```

2.6.3. Файл регистрации

Утилита login обновляет системную учетную информацию, которая регистрируется в файле /etc/acclog. Если этот файл не существует, то вся учетная информация будет игнорироваться.

2.6.3.1. Включение регистрации

Для включения регистрации, необходимо создать пустой файл /etc/acclog. Его можно создать, используя утилиту touch:

```
touch /etc/acclog
```

```
chmod g=,o= /etc/acclog
```

Если этот файл создан, то учетная информация будет регистрироваться в нем.

Следует обратить внимание, что только суперпользователь (root) может создавать и изменять этот файл.

2.6.3.2. Очистка регистрационного файла

В процессе эксплуатации изделия файл /etc/acclog может увеличиваться, занимая значительное дисковое пространство, поэтому следует регулярно распечатывать или архивировать информацию из этого файла. Можно автоматизировать эту процедуру, используя утилиту cron (смотри документ «Описание программы»).

В следующем примере регистрационный файл переименовывается в файл, название которого содержит год и месяц, и создается новый пустой регистрационный файл:

```
mv /etc/acclog /etc/acclogs/9607

touch /etc/acclog

chmod g=,o= /etc/acclog
```

2.6.3.6. Сжатие файла регистрации

Так как файл регистрации содержит повторяющиеся данные, то можно использовать утилиту сжатия gzip, которая обеспечивает высокий коэффициент сжатия файла.

Замечание: Необходимо переименовывать файл *перед* его сжатием. Нельзя сжимать непосредственно файл /etc/acclog.

Пример рекомендуемой процедуры сжатия:

```
mv /etc/acclog /etc/logs/9607

touch /etc/acclog

chmod g=,o= /etc/acclog

gzip /etc/logs/9607
```

Следует учитывать, что в файл /etc/acclog могут добавлять свои регистрационные записи другие утилиты (возможно третьих фирм).

2.6.4. Дополнительные файлы

Для того чтобы компоненты изделия записывали события в регистрационный файл, нужно создать пустой регистрационный файл в соответствующем каталоге.

Файл /tmp/syslog

Некоторые утилиты (как правило, портированные из добных систем) регистрируют проблемы или непредвиденные события в файле /tmp/syslog.

Файл /usr/adm/syslog

Утилита logger добавляет строки в файл /usr/adm/syslog. Пока программа имеет доступ записи в этот файл, она может выполнять утилиту logger, как процесс setuid, принадлежащий adm:adm.

Файл /etc/nologin

Присутствие файла /etc/nologin запрещает кому-либо регистрацию в системе.

2.7. Установка терминалов

2.7.1. Поиск характеристик терминалов

Два вида файлов с характеристиками терминалов дают программистам свободу для написания терминально-независимых программ. Более старые полноэкранные программы чтобы использовать возможности терминала полагались на коды управления в файле с характеристиками терминала /etc/termcap. Более новые программы и утилиты запрашивают базу данных terminfo (в каталоге /usr/lib/terminfo).

2.7.2 Установка типа терминала

Переменная окружения TERM определяет тип терминала. Сначала программы запрашивают переменную окружения TERM, чтобы определить, с каким типом терминала они выполняются. После определения типа терминала программа находит характеристики терминала либо в файле termcap, либо в соответствующем файле terminfo.

Установка типа терминала обычно наследуется из окружения от sinit немедленно после начальной загрузки:

sinit -r //$(bnode)/ TERM=ansi

Как альтернативный вариант, тип терминала может определяться в пользовательском файле .profile.

Для стационарных терминалов, которые выполняют заранее известные прикладные программы, можно предварительно установить переменную окружения TERM при запуске приложения:

TERM=vt100 on -t /dev/ser1 заказная_прикладная_программа

Нельзя сделать это при входе в систему пользователей с помощью login и tinit. Если тип терминала известен, то можно заставить tinit устанавливать переменную окружения TERM перед запуском login:

tinit -c login -t /dev/ser1 TERM=ansi &

2.7.3 База данных termcap

Коды управления в файле termcap могут определять входные и выходные характеристики ряда типов терминалов, в том числе:

- универсальный терминал стандарта ANSI (ansi);
- консоль;
- ANSI (qansi, qansi-g, qansi-t, qansi-m или qansi-w);
- терминал;
- терминал с событиями "мыши";
- Windows;
- монохромный терминал или консоль;
- последовательный терминал;
- VT100 с Auto-margin (vt100);
- VT102 (vt102 или vt102-plus);
- Эмулятор терминала X-term (xterm, xterms, xterm-m или xterm-q).

Полный список типов терминала находится в /etc/termcap.

Каждый элемент в файле termcap начинается со строки псевдонимов терминала. Остальные строки с кодами управления представляет собой список терминальных характеристик.

Каждая строка характеристики может идентифицироваться именем из двух символов. В общем, строка описывает поведение терминала, когда пользователь нажимает клавишу, или визуальный эффект по команде программы. Если имеется более чем одна запись для одного и того же типа терминала, то используется первая запись.

2.7.4 База данных terminfo

Подкаталоги terminfo - /usr/lib/terminfo/... - содержат двоичные файлы. Каждый файл в базе данных terminfo содержит описание характеристик, связанное с одним типом терминала. Изделие поставляется с несколькими такими файлами, включая:

/usr/lib/terminfo/v/vt100 (VT100 терминал)

/usr/lib/terminfo/a/ansi (ANSI терминал)

Соответствующие исходные тексты для всех поддерживаемых типов терминала содержатся в /usr/lib/terminfo/terminfo.src.

Исходные тексты файлов termcap и terminfo схожи в том, что они оба содержат описания характеристик терминала. Однако, имена характеристик в файле terminfo могут иметь длину от двух до пяти символов. И так как файлы terminfo компилируются, то они загружаются и выполняются быстрее.

При необходимости преобразовать из terminfo в termcap (в случае, если имеются какие-то старые программы), утилита infocmp имеет опцию -C, которая генерирует запись termcap из существующего файла terminfo. Если имеется более чем одна запись для одного и того же терминала в файле terminfo, то используется последняя запись.

Если файла terminfo для данного типа терминала нет, можно либо найти соответствующий файл terminfo в любой системе UNIX, либо создать новый файл terminfo с помощью утилит infocmp и tic, либо заказать его у предприятия-поставщика изделия.

Создание файла terminfo

Чтобы просматривать или изменять информацию в существующем файле terminfo, необходимо сначала декомпилировать файл с помощью утилиты infocmp. Чтобы использовать измененный исходный файл, нужно скомпилировать его в двоичную форму, используя утилиту tic, компилятор для terminfo. Прикладные

программы могут извлекать характеристики, в которых они нуждаются для данного терминала, прямо из компилированной версии.

Замечание: Можно получить характеристики из библиотеки curses более высокого уровня.

3. ПРОВЕРКА И ВОССТАНОВЛЕНИЕ ИЗДЕЛИЯ

3.1. Создание резервных копий

3.1.1. Общие замечания

Создание копий данных необходимо для того, чтобы избежать потери данных при сбоях аппаратных средств, программного обеспечения или ошибках оператора, которые могут разрушить оригинал. Если данные важны, то следует регулярно выполнять процедуры резервного копирования, что позволит восстановить потерянную информацию с минимальными затратами времени и денег.

Можно копировать целые файловые системы или только их части. Пользователи могут решить копировать только свои собственные файлы, обычно на дискеты. Для того чтобы копировать большие части файловой системы с файлами, принадлежащими многим пользователям, необходимо разрешение на чтение этих файлов. Суперпользователь (root) имеет такие привилегии.

3.1.2. Когда создавать резервные копии

Данные следует копировать так часто, чтобы можно было восстановить данные, которые все еще являются текущими или на основе которых могут быть восстановлены текущие данные с минимальными затратами. В группе разработки программного обеспечения частота копирования может изменяться в пределах от дня до недели. При сохранении финансовых или кассовых данных, резервное копирование выполняется, как правило, ежедневно или даже дважды в день. Рекомендуется вести архив не на рабочей установке.

3.1.3. Форматы резервных копий

Изделие поддерживает ряд форматов резервных копий, которые могут быть распределены на две группы:

- архивы;

- регулярные файловые системы.

Архив состоит из одного или более файлов, объединенных в единый модуль с собственным каталогом содержимого. Архив может быть сохранен или в регулярном файле, или на низком уровне на блок-ориентированном устройстве, например дискете или ленте.

Сохранение в *регулярной файловой системе* заключается просто в копировании файлов. В этом случае адресатом должно быть устройство с установленной на нем файловой системой.

3.1.3.1. Архивные копии

Изделие поддерживает три основные утилиты архивирования:

- cpio;
- tar;
- pax.

Утилиты cpio и tar связаны с утилитой pax, которая может читать и записывать в обоих форматах, cpio и tar. Утилиты cpio и tar являются стандартом для систем UNIX. Утилита pax включает cpio так и tar, таким образом, она не поддерживает собственный архивный формат. По умолчанию pax будет использовать формат tar при создании архива.

Утилита pax определит, когда будет достигнут конец диска или ленты для тома, и подскажет, чтобы вставили следующий том, который нужно использовать для сохранения. Результат представляет собой резервную копию, распределенную по нескольким носителям (дискетам, лентам и т.п.).

Присваивание имен томам

Формат tar/cpio не присваивает метки тома носителю. Если перепутать носитель или вставить его не в нужном порядке, это приведет к восстановлению ошибочных данных.

Для того чтобы исключить эту возможность, в изделии предусмотрена утилита vol, которая метит каждый дисковый том последовательным номером и, следовательно, предотвращает установку носителей не в том порядке.

Замечание. Утилита vol может использоваться с дискетами или сменными дисками. Эта утилита не может использоваться с ленточными накопителями.

Утилита vol будет, по умолчанию, пропускать первый блок носителя. Это важно для дискет и кассетных дисков, которые содержат сигнатуру в первом блоке. Эта сигнатура содержит размер дискеты (360K, 1.2M и т.п.) и обеспечивает автоматическое перемонтирование сменного носителя файловой системы.

3.1.3.2. Создание резервных копий файловой системы

Существует возможность сохранения файловой системы посредством копирования файлов утилитами cp или cpio **-p**. Если носитель расположен на гибком диске, то утилита cp подскажет о необходимости установки дискеты, но при этом размер файлов не должен превышать емкость дискеты. Если желательно поддерживать копии на гибких дисках, то рекомендуется использовать одну из утилит, предназначенных для создания архивов.

3.1.4. Носители резервных копий

Выбор носителя резервных копий будет определяться доступным аппаратным обеспечением и его стоимостью. Имеется несколько вариантов выбора:

- гибкий диск;

- магнитная лента;
- сменный диск;
- жесткий диск.

Изделие имеет драйверы для нескольких типов SCSI-контроллеров. Эти драйверы определяют и поддерживают жесткие диски, накопители на магнитных лентах с последовательным доступом, устройства WORM (однократная запись/многократное чтение), дисководы CD-ROM и оптические дисководы.

3.1.4.1. Гибкие диски

Для того чтобы выполнять операции записи на дискету и чтения с нею необходимо убедиться, что запущен драйвер гибких дисков Fsys.floppy (см. описание Fsys.floppy в документе «Описание программы»).

Следующая командная строка запустит драйвер (предполагается, что процесс Fsys уже запущен):

```
Fsys.floppy &
```

По умолчанию, Fsys.floppy создает блок-ориентированное устройство для дисковода гибких дисков номер 0 (/dev/fd0). Если компьютер имеет более чем один дисковод для гибких дисков, то Fsys.floppy создает блок-ориентированное устройство для каждого накопителя. Например, дисковод номер 1 будет иметь имя /dev/fd1.

Архивирование данных на дискетах

При использовании для создания резервных копий архивной утилиты необходимо помнить, что утилита считывает и записывает данные, начиная прямо с первого блока. Утилита tar также пишет архивные данные в первый блок, поэтому первый блок на дискете будет содержать 512 байтов исходных данных. Это может

привести к некорректной работе драйвера Fsys.floppy, который на всех дискетах пытается интерпретировать первый блок как сигнатуру КПДА.00002-01.

Чтобы исключить непредсказуемое поведение при архивировании данных на гибкий диск или восстановлении архивированных данных из гибкого диска, перед началом копирования, следует запустить утилиту lockfd. Утилита lockfd заставляет драйвер Fsys.floppy зафиксировать специфический размер носителя и подавляет попытку драйвера читать информацию сигнатуры из первого блока при восстановлении данных из дискеты.

Копирование файлов на гибкий диск

Каждую неформатированную дискету прежде, чем начинать копировать на нее файлы, необходимо отформатировать и инициализировать. В следующем примере показано, как инициализировать гибкий диск высокой плотности $3^1/_2$":

```
fdformat -s 1.4m /dev/fd0

dinit /dev/fd0
```

Далее следует смонтировать файловую систему на блок-ориентированном устройстве:

```
mount /dev/fd0 /fd0
```

Теперь можно рассматривать дискету как файловую систему КПДА.00002-01, смонтированную как каталог /fd0. Для копирования файлов на дискету используйте утилиту cp.

3.1.4.2. Магнитная лента

Чтобы поддерживать сохранение и восстановление копий на лентах, следует удостовериться, что запущен необходимый драйвер. Обычно это один из SCSI-драйверов, описанных в документе «Описание программы».

Например, имеется SCSI-контроллер, совместимый с Adaptec 1540, и подключенный ленточный накопитель, то следующая команда запустит драйвер и зарегистрирует ленту с последовательным доступом как /dev/tape1:

```
Fsys.aha4scsi fsys -n 1=tape
```

Утилиты архивирования будут читать и записывать непосредственно на ленту блок-ориентированные файлы. Нельзя монтировать файловую систему с этим типом блок-ориентированного файла.

Когда драйвер AHA 4 получает запрос на чтение или запись, он начинает выполнение операции, начиная с текущей позиции ленты. Если на ленте необходимо начать записывать новую копию, то необходимо будет стереть данные с ленты и перемотать ленту на начало.

Поиск номера позиции на ленте и функции управления обеспечиваются утилитой tape, которая описана в книге *«Описание программы»*. Например, следующая команда перемотает и сотрет ленту, затем перемотает ленту на начало и подготовит ее к процедуре архивирования:

```
tape rewind erase
```

3.1.4.3. Сменные диски

В отличие от дискет и лент, жесткий сменный диск позволяет избегать использования утилит архивирования, таких как tar. Вместо этого можно будет использовать утилиту cp или pax -rw для копирования данных на файловую систему сменного диска. Это позволит быстро и легко восстанавливать одиночные файлы легко.

3.1.4.4. Жесткий диск

Можно установить в компьютере второй жесткий диск для копирования в режиме online. Как вариант, можно рассмотреть создание копий на жестком диске в

другой машине в сети. Однако, копирование через сеть медленнее, чем копирование на локальный жесткий диск. Если имеется дополнительный жесткий диск, то можно спланировать все операции копирования как задание для утилиты cron, которая выполнит их, например, в ночное время.

3.1.5. Сжатие данных

Можно использовать утилиту сжатия, чтобы уменьшить размер дискового пространства, требуемого для сохранения данных. Степень сжатия будет зависеть от характера сохраняемых данных. Некоторые базы данных, содержащие большое количество повторяемых данных, могут быть сжаты до 90%. Отдельные же данные могут быть сжаты менее чем на 10%.

Замечание: Хотя сжатие и сохраняет свободное пространство носителя, но имеет два побочных эффекта:

- сжатие данных требует значительного количества вычислений, что может замедлять процесс сохранения данных;
- невозможно восстановить сжатые данные в случае возникновения сбоя во время процесса сжатия данных или при хранении носителя. Есть вероятность, что один дефектный блок данных может повлиять на остальные данные (т.е. увеличивается вероятность потери всех данных в архивном файле).

Можно использовать утилиты gzip или freeze, чтобы сжать данные, и утилиты gunzip или melt, чтобы восстановить их. Обе утилиты работают как с потоком данных, так и с файлами. Способность этих утилит работать как фильтр позволяет присоединять их к одному из стандартных архиваторов через конвейер.

3.1.6. Примеры создания архивов

3.1.6.1. Сжатые данные в архиве на дискетах

Собрать файлы в каталоге /home в архивном формате tar, сжать архив и записать его на такое количество дискет, которое потребуется, проставляя порядковые номера на дискеты:

```
pax -w -x ustar /home | freeze | vol -w /dev/fd0
```

Считать данные с дискет, распаковать данные и восстановить файлы:

```
vol -r /dev/fd0 | melt | pax -r
```

3.1.6.2. UNIX-совместимые архивы на дискетах

Сохранить файлы в каталоге /home/dino в архивном формате tar для восстановления в системе UNIX:

```
pax -w -t/dev/fd0 -xustar /home/dino
```

Сохранить файлы в каталоге /home/dino в архивном формате cpio для восстановления в системе UNIX:

```
pax -w -xcpio -t/dev/fd0 /home/dino
```

Восстановить данные в архивном формате tar или cpio из другой системы UNIX и поместите все файлы в каталог /usr/unix:

```
pax -r -s -t/dev/fd0  ",/,/usr/unix/,"
```

3.1.6.3. Архив на ленте

Начать на ленте новый архив и сохраните все файлы на ленте:

```
tape rewind erase    (стереть данные на ленте и перемотать ее)
```

```
pax -w -t/dev/tp0 /
```

Добавить файлы, которые изменились позднее даты последней модификации файла lastsave, в конец существующей архивной ленты. После сохранения обновить время последней модификации lastsave:

```
tape forward

touch newsave

find / -newer lastsave | pax -w -t/dev/tp0

mv newsave lastsave
```

Восстановить все файлы с ленты в каталоге /home/dino:

```
tape rewind

pax -r "/home/dino/*" -t/dev/tp0
```

3.1.6.4. Сменные магнитные/оптические диски

Скопировать все файлы с файловой системы на узле 1 на файловую систему на узле 2:

```
cp -Rp //1/ //2/
```

В следующем примере на узле 2 очень большой оптический диск. Создание полной копии выполняется каждую пятницу, а создание частичной копии измененных файлов может выполняться каждый день недели:

```
cp -Rp //1/ //2/fri
cp -Rp -a date //1/ //2/mon
cp -Rp -a date //1/ //2/tue
.
.
```

Это также можно выполнять с помощью cpio -p.

3.2. Восстановление дисков и файлов

3.2.1. Общие рекомендации

Файловая система базируется на принципе обеспечения целостности и непротиворечивости файловой системы в целом в любой момент времени. В то время как большинство данных содержится в буферном кэше и записывается только после короткой задержки, критические данные файловой системы записываются немедленно. Изменения в каталогах, индексных дескрипторах (inode), блоках экстентов и битовых картах немедленно записываются на диск, чтобы гарантировать, что структура файловой системы на диске никогда не разрушится (то есть данные на диске никогда не будут внутренне противоречивыми).

Если происходит сбой, можно использовать следующие утилиты сопровождения и восстановления:

- fdisk
- dinit
- chkfsys
- dcheck
- zap
- spatch

Эти утилиты позволят определить, было ли какое-то повреждение в файлах, которые были открыты на запись в момент возникновения сбоя. Эти же самые утилиты могут исправить такие повреждения и, во многих случаях, полностью восстановить файловую систему.

Иногда повреждения могут быть более серьезными. Например, на жестком диске может появиться плохой блок в середине файла или, еще хуже, - в середине каталога или какого-то другого критического блока.

И в этом случае предоставляемые утилиты могут помочь определить степень такого повреждения. Может потребоваться перестроить файловую систему таким

образом, чтобы избежать поврежденных областей. В этом случае некоторые данные будут потеряны, но, приложив некоторые усилия, большую часть используемых данных можно восстановить.

3.2.2. Краткий обзор структуры раздела КПДА.00002-01

Файловая система КПДА.00002-01 может занимать целый диск (в случае дискет) или один из разделов жесткого диска. В пределах дискового раздела, файловая система КПДА.00002-01 содержит следующие компоненты:

- блок загрузчика;
- корневой блок;
- битовая карта;
- корневой каталог;
- другие данные.

Блок загрузчика

Блок загрузчика - первый блок раздела КПДА.00002-01. Он содержит программу начальной загрузки, которая загружает ОС КПДА.00002-01 в память.

Корневой блок

Корневой блок - второй блок раздела КПДА.00002-01. Он содержит элемент каталога для корня (/), inode-элементы для файла inode и поле меток.

Блоки битовой карты

За корневым блоком следует несколько блоков битовой карты. Они образуют битовую карту для раздела КПДА.00002-01. Каждому блоку раздела соответствует один бит, таким образом один блок битовой карты будет использоваться для каждых 4096 дисковых блоков (что соответствует 2 Мб дискового пространства).

Если значение бита нулевое, то соответствующий ему блок не использован. Неиспользуемые биты в конце последнего блока битовой карты (для которых нет соответствующих дисковых блоков) имеют значение единица.

Присвоение битов начинается с младшего бита байта 0 первого блока битовой карты, который соответствует блоку 1 раздела КПДА.00002-01.

Корневой каталог

Корневой каталог следует за блоками битовой карты. Корневой каталог является "нормальным" каталогом (смотрите ниже “Каталоги”) и первоначально создается утилитой dinit с объемом, достаточным для 32 элементов каталога (4 блока).

Корневой каталог (/) содержит элементы каталогов для различных специальных файлов, которые всегда существуют в файловой системе КПДА.00002-01 – их создает утилита dinit при инициализии файловой системы (см. таблицу 3.1).

Таблица 3.1. Элементы корневого каталога, создаваемые утилитой `dinit`

Файл:	**Описание:**
/.	Ссылка на каталог /
/..	Тоже ссылка на каталог /
/.bitmap	Файл "только для чтения", состоящий из блоков битовой карты.
/.inodes	Нормальный файл, занимающий, по крайней мере, один блок на дискете или ОЗУ-диске и 16 блоков на других дисках, /.inodes - это набор inode-элементов. Первый элемент резервируется и используется в качестве области сигнатуры. Первые байты файла .inode - "IamTHE.inodeFILE".
/.boot	Файл образа ОС, который будет загружаться в память при *стандартной* начальной загрузке. Этот файл будет иметь нулевую длину, если файла начальной загрузки не существует.
/.altboot	Файл образа ОС, который будет загружаться в память при *альтернативной* начальной загрузке. Этот файл имеет нулевую длину, если файла альтернативной начальной загрузки не существует.

3.2.3. Утилиты сопровождения файловой системы

Для работы с файловой системой предназначены следующие утилиты:

fdisk – создает и поддерживает *блок раздела* на жестком диске. Этот блок совместим с другими операционными системами и может быть поддержан версиями fdisk других ОС.

dinit – создание раздела КПДА.00002-01.

chkfsys – средство контроля логической целостности раздела КПДА.00002-01.

dcheck – средство контроля физической целостности раздела КПДА.00002-01.

zap – позволяет root удалять файлы или каталоги из файловой системы, не возвращая используемые блоки в свободный список.

spatch – позволяет просматривать диск на низком уровне и устранять некоторые неполадки. Иногда можно устранить кратковременные сбои диска, считывая и перезаписывая неисправный блок посредством утилиты spatch.

Подробное описание этих утилит содержится в «Описании программы».

3.2.4. Процедуры восстановления диска

3.2.4.1. Использование утилиты chkfsys

Утилита chkfsys – это главное средство для проверки и восстановления поврежденной файловой системы в операционной системе КПДА.00002-01. С помощью этой утилиты можно определить и исправить большинство небольших проблем, а также проверить целостность дисковой системы в целом.

Обычно для утилиты chkfsys требуется, чтобы проверяемый раздел файловой системы был не задействован и чтобы среди размещенных в нем файлов не было открытых. Должны быть завершены все процессы, которые открыли файлы или могут их открыть, пока выполняется chkfsys.

Чтобы выполнить chkfsys на точке монтирования, следует ввести:

```
chkfsys /
```

Утилита просматривает весь дисковый раздел от корня вниз, создавая внутреннюю копию битовой карты, и проверяет непротиворечивость всех файлов и каталогов, которые она находит в процессе.

После завершения обработки всех файлов chkfsys сравнит внутреннюю битовую карту с битовой картой на диске. Если они совпадают, то выполнение chkfsys завершается. При обнаружении несоответствия утилита chkfsys перепишет (с вашего согласия) битовую карту в соответствии с данными о файлах, которые она обнаружила и проверила.

Кроме того, проверяя распределение блоков (битовую карту), chkfsys пытается фиксировать любые обнаруженные проблемы. Например, chkfsys может:

- "освободить" файлы, которые записывались в момент краха системы;
- откорректировать размер файла в элементе каталога, чтобы он соответствовал реальным данным.

Когда выполнять chkfsys

Рекомендуется выполнять chkfsys как часть регулярных плановых процедур сопровождения вычислительной системы под управлением операционной системы КПДА.00002-01 – это позволит проверять целостность данных на диске. Например, возможен вариант выполнения chkfsys на ЭВМ при каждой загрузке. Автоматизированная проверка в файловой системе во время начальной загрузки гарантирует, что chkfsys будет пытаться устранить любые проблемы, которые она найдет во время просмотра. Чтобы автоматизировать этот процесс, добавьте chkfsys в файл sysinit.*узел* сервера.

Особенно важно выполнять chkfsys после системного отказа, сбоя по питанию или непредвиденной перезагрузки системы, чтобы определить поврежденные файлы. Утилита chkfsys проверяет флажок "чисто" ("clean") на диске, чтобы определить, была ли система в тот момент в устойчивом состоянии.

Флажок "чисто" сохраняется на диске и поддерживается системой. Флажок сбрасывается всякий раз, когда файл открывается для модификации, и устанавливается после того, как все открытые файлы закроются и все связанные с ними данные будут сброшены из кэша на диск. Когда флажок "чисто" установлен,

chkfsys считает, что файловая система не повреждена. Если chkfsys обнаруживает флажок очистки сброшенным, она пробует устранить проблему.

Утилита chkfsys поддерживает опцию **-u**, которая позволяет игнорировать флаг "чисто" и вызывает безусловное выполнение chkfsys. Потребность игнорировать флажок "чисто" может возникнуть, когда:

- dcheck обнаруживает дефектные блоки;
- файлы были преднамеренно стерты или удалены утилитой zap;
- необходимо выполнить общую проверку исправности.

Использование chkfsys в работающей системе

Утилита chkfsys обычно требует эксклюзивного использования файловой системы для обеспечения *всесторонней* проверки диска.

Замечание: Имеется некоторый риск при запуске утилиты chkfsys на работающей системе - и chkfsys, и файловая система читают и, возможно, записывают на диске одни и те же блоки. Кроме того, файловая система имеет внутренние кэшированные данные о файлах и каталогах, которые не могут быть откорректированы, когда chkfsys делает изменение. Но статические изменения в файлах или каталогах, которые не открыты Fsys в настоящее время, *вероятно*, не вызовут проблем.

Если выполняется приложение, которое не может прерваться, или не была выполнена chkfsys, потому что были открытые файлы, следует выполнить chkfsys с опцией **-f**:

```
chkfsys -f /dev/hd0t77
```

Это вызывает специальный режим "только для чтения" chkfsys. Он даст вам представление об общем состоянии файловой системы.

3.2.4.2 Восстановление дефектного блока в середине файла

На жестких дисках со временем могут появляться дефектные блоки. В некоторых случаях можно восстановить большую часть или даже все данные файла, содержащего дефектный блок.

Некоторые дефектные блоки являются результатом сбоев по питанию или низкого качества носителя жесткого диска. В этих случаях просто чтение и последующая перезапись блока "восстановит" блок на короткий период времени. Это позволит скопировать весь файл куда-нибудь еще, прежде чем блок испортится снова. Данная процедура, несомненно, не причинит вреда и часто стоит того, чтобы ее выполнили.

Для того чтобы проверить блоки в файле, используют утилиту spatch. При чтении дефектного блока spatch сообщает об ошибке, но в действительности, возможно, была прочитана часть "хороших" байтов из этого блока. Запись того же блока обратно часто достигает цели.

В то же самое время, spatch перезапишет правильную CRC (контрольную сумму избыточного циклического кода), что вновь сделает блок хорошим (но, возможно, с неправильными данными).

Можно затем скопировать весь файл куда-нибудь еще и потом утилитой zap удалить поврежденный файл. Для того чтобы завершить процедуру, необходимо пометить сбойный блок как дефектный (добавить его в файл /.bad_blks), затем восстановить остальные хорошие блоки, используя утилиту chkfsys.

Если эта процедура не поможет, следует использовать утилиту spatch, чтобы скопировать по возможности большую часть файла в другой файл, и затем утилитой zap удалить дефектный файл и выполнить chkfsys.

3.2.5. Порядок действий в случае, если система не загружается

Если ранее работавшее изделие внезапно прекратило работу и больше не загружается, то, возможно, произошло следующее:

- отказали аппаратные средства или были повреждены данные на жестком диске;
- изменен/переписан файл начальной загрузки или изменен файл системной инициализации – это два наиболее типичных случая.

Следующие шаги могут помочь идентифицировать проблему. Там, где это возможно, рекомендуются корректирующие действия.

Шаг 1 – Следует попробовать загрузиться из дискеты или через сеть

Если имеется сеть для загрузки, надо попытаться загрузить машину по сети. Если машина загружается, следует войти в систему как root, а затем запустить локальную файловую систему:

```
Fsys &
```

Если нет сети, следует загружаться с восстановительной дискеты (описанной ранее в этом разделе) или с загрузочной дискеты КПДА.00002-01, которая использовалась для установки системы на жесткий диск. В данном случае файловая система будет запущена, и осуществлена регистрирация как root.

Шаг 2 - Запустить драйвер жесткого диска

Теперь нужно запустить подходящий драйвер жесткого диска. Например, чтобы запустить драйвер для адаптера Adaptec 4 SCSI, следует напечатать:

```
Fsys.aha4scsi &
```

Если используется другой тип драйвера, необходимо ввести другое имя вместо этого.

Эта команда должна создать блок-ориентированный файл с именем /dev/hd0, который представляет весь жесткий диск.

Шаг 3 - Выполнить fdisk

Выполнение утилиты fdisk немедленно даст полезную информацию о состоянии жесткого диска.

Утилита fdisk может сообщить об одной из нескольких типов проблем, указанных в таблице 3.2.

Таблица 3.2. Проблемы, выявляемые утилитой fdisk

Проблема	**Вероятная причина**	**Действия**
Ошибка чтения блока 1.	Произошел отказ либо дискового контроллера, либо жесткого диска.	Если диск исправен, то, заменив плату контроллера, *можно* продолжить использование диска. В противном случае следует заменить жесткий диск, вновь установить КПДА.00002-01 и восстановить файлы с копии.
Неправильные дисковые параметры.	Аппаратные средства, вероятно, "потеряли" информацию об этом жестком диске. Наиболее вероятная причина состоит в том, что разрядилась батарея памяти CMOS.	Перезапустить процедуру аппаратной установки *setup* (или процедуру выбора программируемых опций на PS/2). Конечно, замена батареи более надежна.
Информация о дефектном разделе.	Если дисковый размер правильно определен утилитой fdisk, но информация о разделе некорректна, то это означает, что данные в блоке 1 физического диска были повреждены.	Использовать fdisk, чтобы вновь создать правильную информацию о разделе. Рекомендуется записывать или распечатывать правильную информации о разделе на случай, если когда-либо придется выполнять этот шаг.

Шаг 4 – Монтирование раздела и файловой системы

После проверки аппаратных средств (они работают, по крайней мере, для блока 1) и определения для изделия правильного раздела следует создать блок-ориентированный файл непосредственно для раздела КПДА.00002-01 и монтировать блок-ориентированный файл как файловую систему КПДА.00002-01:

```
mount -p /dev/hd0 /dev/hd0t77 /hd
```

3.2.5.1. Порядок действий в случае, если команда mount выполняется

Эта команда должна создать том, называемый /dev/hd0t77. В зависимости от состояния раздела КПДА.00002-01, mount может потерпеть неудачу. Если информация раздела правильная, проблем не должно возникать. Поскольку корень (/) уже существует (на дискете или на удаленном диске на сети), то монтируется локальный раздел жесткого диска как файловая система с именем /hd.

Теперь цель будет состоять в выполнении утилиты chkfsys, чтобы исследовать - и, возможно, исправить - файловую систему.

***Замечание*:** Если загрузка произведена с дискеты и неизвестно, есть ли какое-либо повреждение файловой системы на жестком диске (например, система не в состоянии загрузиться из-за простой ошибки в файле загрузки или файле инициализации системы), то можно изменить корневой префикс на раздел жесткого диска следующей командой, что возобновит нормальную работу системы:

```
/hd/bin/prefix -R /=/hd/
```

Если выполняется эта команда, можно пропустить остальную часть этого раздела.

3.2.5.2. Порядок действий в случае, если команда mount не выполняется

Если команда mount не выполняется, то, вероятно, повреждена первая часть раздела КПДА.00002-01 (так как Fsys не будет монтировать поврежденную, по его мнению, файловую систему).

В этом случае можно использовать утилиту dinit для записи достаточного количества хорошей информации на диск, чтобы удовлетворить Fsys:

```
dinit -hr /dev/hd0t77
```

Опция **-r** сообщает утилите dinit, чтобы она перезаписала:

- корневой блок;
- битовую карту (с распределением *всех* блоков);
- постоянные части корневого каталога.

Следует теперь повторно выполнить команду mount и еще раз поробовать создать точку монтирования для файловой системы КПДА.00002-01 с именем /hd.

После этого необходимо восстановить битовую карту с помощью утилиты chkfsys даже для хорошего раздела.

Шаг 5. Запуск chkfsys

По крайней мере часть файловой системы КПДА.00002-01 теперь должна быть доступна. Можно использовать утилиту chkfsys, чтобы проверить файловую систему и восстановить, по возможности, большую часть данных.

Если жесткий диск смонтирован как /hd (например, компьютер загружается с дискеты), введите:

```
/hd/bin/chkfsys /hd
```

Если жесткий диск смонтирован, как / (например, сетевая начальная загрузка), введите:

```
chkfsys /
```

В каждом случае следует взять на заметку все возникающие проблемы и позволить утилите chkfsys исправить все, что возможно. Дальнейшие действия зависят от результата выполнения утилиты chkfsys.

3.2.5.3. Порядок действий в случае неисправности диска

Если по некоторой причине диск полностью неисправен, следует обратиться к подразделу 1.14.7 «Восстановление потерянных файлов и каталогов». В некоторых

случаях, вероятно, придется вновь установить изделие и восстановить диск с резервных копий файлов.

3.2.5.4. Порядок действий, если файловая система не повреждена

Если файловая система не повреждена, однако компьютер еще отказывается загружаться с жесткого диска, то, вероятно, повреждено что-либо из следующего:

- загрузчик раздела в физическом блоке 1;
- загрузчик изделия в первом блоке раздела КПДА.00002-01.

Для перезаписи загрузчика раздела для диска, следует использовать fdisk:

```
fdisk /dev/hd0 loader
```

Для перезаписи загрузчика, следует использовать dinit:

```
dinit -b /dev/hd0t77
```

3.2.6. Восстановление потерянных файлов и каталогов

При обнаружении потери полностью или частично файлов или каталогов вследствие повреждения диска следует выполнить утилиту chkfsys. Если в случае ее выполнения определенные ключевые файлы или каталоги не были восстановлены, можно использовать утилиту spatch для восстановления некоторых или всех данных.

Перед этим необходимо сначала ознакомиться с организацией файловой системы КПДА.00002-01 и изучить описание утилиты spatch.

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (стра-ниц) в докум.	№ документа	Входящий № сопроводи-тельного докум. и дата	Под-пись	Дата
Изм.	Изме-ненных	Заменен-ных	Новых	аннули-рован-ных					