1.1 ФАЙЛОВАЯ СИСТЕМА

Все файлы, с которыми могут манипулировать пользователи, располагаются в файловой системе, представляющей собой дерево, промежуточные вершины которого соответствуют каталогам, а листья – файлам и пустым каталогам. Примерная структура файловой системы ОС UNIX показана на рисунке 1.1. Реально на каждом логическом диске располагается отдельная иерархия каталогов и файлов. Для получения общего дерева в динамике используется "монтирование" отдельных иерархий к фиксированной корневой файловой системе.

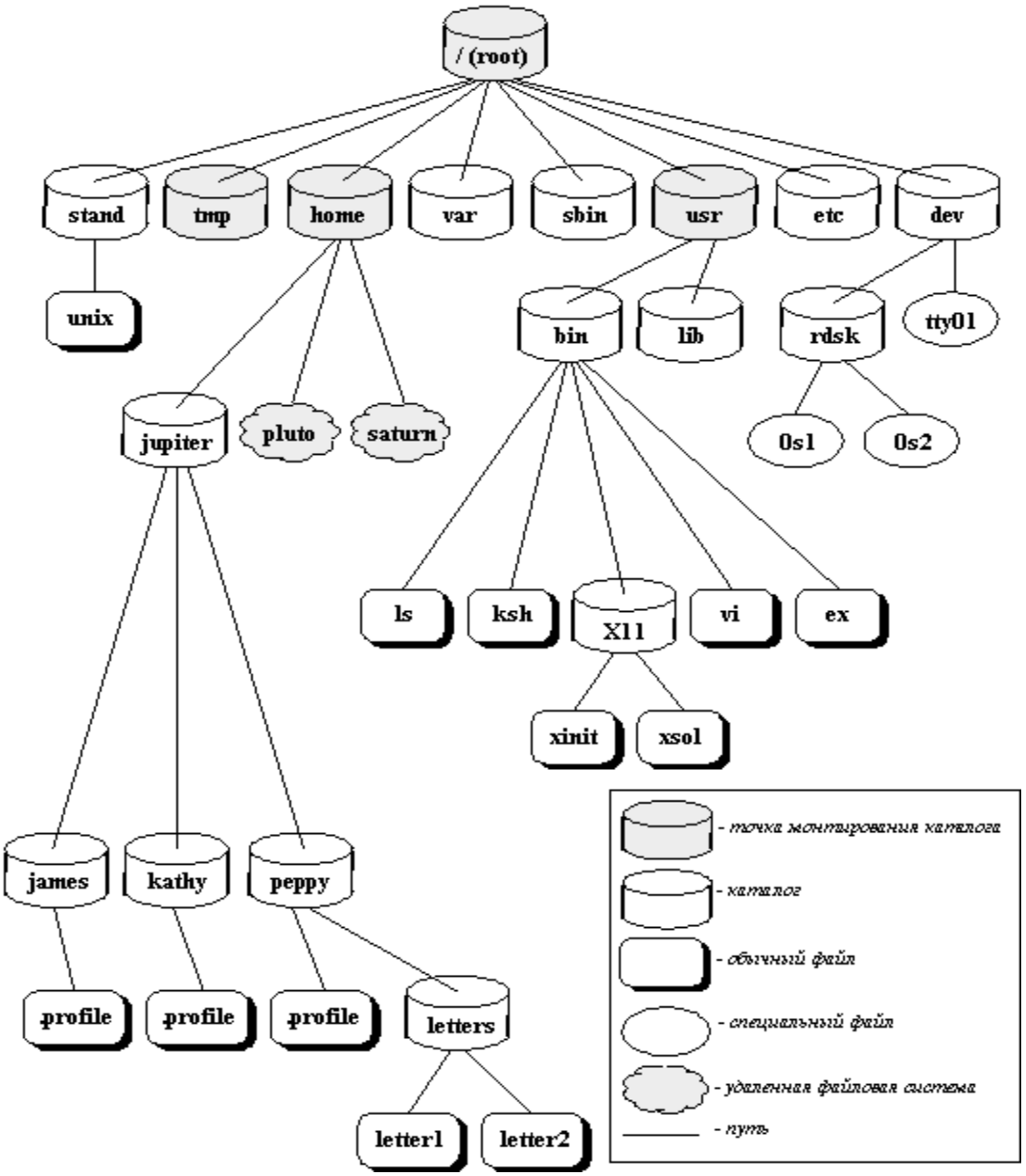

Рис. 1.1. Структура каталогов файловой системы

Каждый каталог и файл файловой системы имеет уникальное полное имя (`full pathname`), отсчитанное от корневого каталога обозначаемого "/". Относительным именем файла (`relative pathname`) называется имя, задающее путь к файлу от текущего каталога.

В каждом каталоге содержатся два специальных имени, имя ".", именующее сам этот каталог, и имя "..", именующее "родительский" каталог данного каталога.

UNIX поддерживает многочисленные утилиты, позволяющие работать с файловой системой и доступные как команды командного интерпретатора. В таблице 1.1 приведены наиболее употребимые их них.

Таблица 1.1

Команда	**Описание**
`cp имя1 имя2`	копирование файла имя1 в файл имя2
`rm имя1`	уничтожение файла имя1
`mv имя1 имя2`	переименование файла имя1 в файл имя2
`mkdir имя`	создание нового каталога имя
`rmdir имя`	уничтожение каталога имя
`ls имя`	выдача содержимого каталога имя
`cat имя`	выдача на экран содержимого файла имя
`chown имя режим`	изменение режима доступа к файлу

Ниже приведено описание основных элементов иерархии файловой системы UNIX.

`/bin` (т.е. двоичные или выполняемые файлы) – каталог, содержащий системные программы.

`/dev` – файлы в этом каталоге представляют сбой специальные файлы, т.е. драйверы устройств.

`/etc` – каталог, содержащий всевозможные файлов конфигурации. Например, `/etc/passwd` (файл паролей), `/etc/rc` (командный файл инициализации) и т.д.

`/sbin` – каталог предназначен для хранения важных системных двоичных файлов, используемых системным администратором.

`/home` – каталог содержит домашние каталоги пользователей.

`/lib` – содержит образы разделяемых библиотек. Эти файлы содержат код, который могут использовать многие программы.

`/proc` – это "виртуальная файловая система", в которой файлы хранятся в памяти, а не на диске. Они связаны с различными процессами, происходящими в системе, и позволяют получить информацию о том, что делают программы и процессы в указанное время.

`/tmp` − каталог для временных файлов.

/var – содержит каталоги, которые часто меняются в размере или имеют тенденцию быстро расти. Например, `/var/adm` – содержит файлы системного администрирования, фиксирующие ошибки и проблемы, возникающие в системе, `/var/spool` – содержит файлы, которые предварительно формируются для других программ.

/usr – каталог, содержащий наиболее важные и полезные (но не необходимые) программы и файлы конфигурации, используемые системой. Каталог включает следующие подкаталоги:

- `/usr/bin` – дополнительные программы UNIX.
- `/usr/etc` – аналогично `/etc`, содержит всевозможные системные программы и конфигурационные файлы, не существенные для системы но полезные
- `/usr/include` – содержит include-файлы для компилятора Си.
- `/usr/lib` – содержит библиотеки эквивалентные файлам из /lib.
- `/usr/local` – содержит различные программы и файлы, несущественные для системы, но полезные при работе с ней.
- `/usr/man` –содержит MAN-страницы
- `/usr/src` – содержит исходные коды для различных программ вашей системы.

1.1.1 СТРУКТУРА ФАЙЛОВОЙ СИСТЕМЫ

Файловая система обычно размещается на дисках или других устройствах внешней памяти, имеющих блочную структуру. Кроме блоков, сохраняющих каталоги и файлы, во внешней памяти поддерживается еще несколько служебных областей.

В мире UNIX существует несколько разных видов файловых систем со своей структурой внешней памяти. Наиболее известны традиционная файловая система UNIX System V (`s5`) и файловая система семейства UNIX BSD (`ufs`). Файловая система `s5` состоит из четырех секций (рисунок 1.2,a). В файловой системе `ufs` на логическом диске (разделе реального диска) находится последовательность секций файловой системы (рисунок 1.2,b).

Назначение областей диска следующее:

- `Boot`-блок содержит программу раскрутки, которая служит для первоначального запуска ОС UNIX.
- Суперблок – содержит список свободных блоков и свободные i-узлы (information nodes - информационные узлы). В файловых системах `ufs` для повышения устойчивости поддерживается несколько копий суперблока (по одной копии на группу цилиндров). Каждая копия су-

перблока имеет размер 8196 байт, и только одна копия суперблока используется при монтировании файловой системы (в случае повреждения используется резервная).

- Блок группы цилиндров – содержит число i-узлов, специфицированных в списке i-узлов для данной группы цилиндров, и число блоков данных, которые связаны с этими i-узлами. Размер блока группы цилиндров зависит от размера файловой системы. Для повышения эффективности файловая система `ufs` старается размещать i-узлы и блоки данных в одной и той же группе цилиндров.
- Список i-узлов – содержит список i-узлов, соответствующих файлам данной файловой системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных i-узлов. В i-узле хранится информация, описывающая файл: режимы доступа к файлу, время создания и последней модификации, идентификатор пользователя и идентификатор группы создателя файла, описание блочной структуры файла и т.д.
- Блоки данных – в этой части файловой системы хранятся реальные данные файлов. В случае файловой системы `ufs` все блоки данных одного файла пытаются разместить в одной группе цилиндров. Размер блока данных определяется при форматировании файловой системы и может быть установлен в 512, 1024, 2048, 4096 или 8192 байтов.

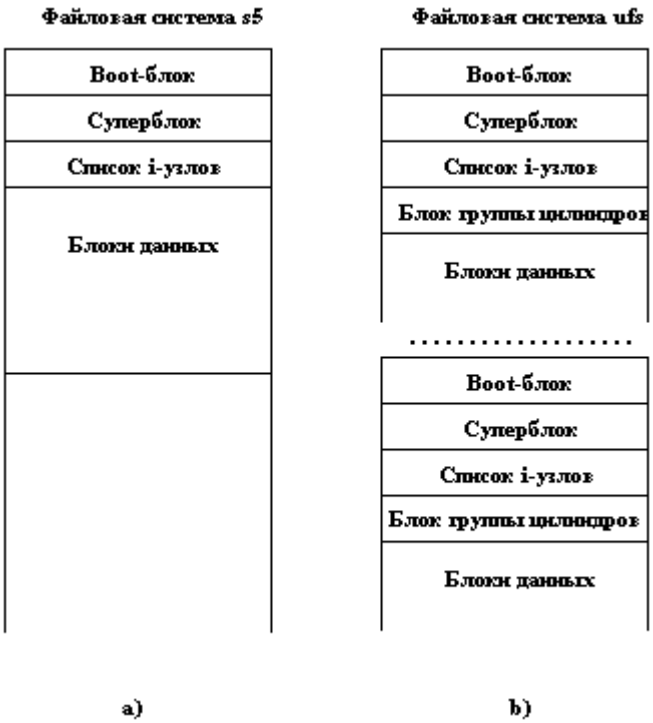

Рис. 1.2. Структура внешней памяти файловых систем s5 и ufs

1.1.2

1.1.3 МОНТИРУЕМЫЕ ФАЙЛОВЫЕ СИСТЕМЫ

Файлы любой файловой системы становятся доступными только после "монтирования" этой файловой системы. Файлы "не смонтированной" файловой системы не являются видимыми операционной системой.

Для монтирования файловой системы используется системный вызов `mount`. Монтирование состоит в следующем. В имеющемся к моменту монтирования дереве каталогов и файлов должен иметься листовой узел – пустой каталог (точка монтирования). Во время выполнения системного вызова `mount` корневой каталог монтируемой файловой системы совмещается с точкой монтирования, в результате чего образуется новая иерархия с полными именами каталогов и файлов.

Смонтированная файловая система впоследствии может быть отсоединена от общей иерархии с использованием системного вызова `umount`. Для успешного выполнения этого системного вызова требуется, чтобы отсоединяемая файловая система к этому моменту не находилась в использовании. Корневая файловая система всегда является смонтированной, и к ней не применим системный вызов `umount`.

Отдельная файловая система обычно располагается на логическом диске. Новая файловая система образуется на диске после форматирования с использованием утилиты `mkfs`. Вновь созданная файловая система инициализируется в состояние, соответствующее наличию всего лишь одного пустого корневого каталога.

1.1.4 ИНТЕРФЕЙС С ФАЙЛОВОЙ СИСТЕМОЙ

Ядро ОС UNIX поддерживает для работы с файлами несколько системных вызовов. Среди них наиболее важными являются:

```
open, creat, read, write, lseek, close
```

Файл в системных вызовах, обеспечивающих реальный доступ к данным, идентифицируется своим дескриптором (целым значением). Дескриптор файла выдается системными вызовами `open` (открыть файл) и `creat` (создать файл). Основным параметром операций открытия и создания файла является полное или относительное имя файла. Кроме того, при открытии файла указывается также режим открытия (только чтение, только запись, запись и чтение и т.д.) и характеристика, определяющая возможности доступа к файлу:

```
open(pathname, oflag [,mode])
```

Открытый файл может использоваться для чтения и записи последовательностей байтов. Для этого поддерживаются два системных вызова:

```
read(fd, buffer, count)
write(fd, buffer, count)
```

где `fd` – дескриптор файла, `buffer` – указатель символьного массива и `count` – число байтов, которые должны быть прочитаны из файла или в него записаны. Результат функции `read` или `write` - целое число, которое совпадает со значением `count`, если операция заканчивается успешно, равно нулю при достижении конца файла и отрицательно при возникновении ошибок.

В каждом открытом файле существует текущая позиция. При открытии файл позиционируется на первый байт. После выполнения системного вызова `read` (или `write`) указатель чтения/записи файла смещается в следующую позицию. Для явного позиционирования файла служит системный вызов

```
lseek(fd, offset, origin)
```

где `fd` - дескриптор файла, `offset` задает значение относительного смещения указателя чтения/записи, а параметр `origin` указывает, относительно какой позиции должно применяться смещение (0 –относительно начала файла, 1 – относительно текущей позиции файла, 2 – относительно конца файла).

1.2 РАЗНОВИДНОСТИ ФАЙЛОВ

В ОС UNIX понятие файла является универсальной абстракцией, позволяющей работать с обычными файлами, содержащимися на устройствах внешней памяти; с устройствами, отличающимися от устройств внешней памяти; с информацией, динамически генерируемой другими процессами и т.д. Для поддержки этих возможностей единообразным способом файловые системы ОС UNIX поддерживают несколько типов файлов, наиболее существенные из которых рассмотрены ниже.

1.2.1 ОБЫЧНЫЕ ФАЙЛЫ

Обычные (или регулярные) файлы реально представляют собой набор блоков на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию, так и произвольную двоичную информацию. Файловая система не предписывает обычным файлам какую-либо структуру, обеспечивая на уровне пользователей представление обычного файла как последовательности байтов. Используя базовые системные вызовы ввода/вывода, пользователи могут как угодно структурировать файлы.

Для некоторых файлов, которые должны интерпретироваться компонентами самой операционной системы, UNIX поддерживает фиксированную структуру. Наиболее важным примером таких файлов являются объектные и исполняемые файлы. Структура этих файлов поддерживается компиляторами, редакторами связей и загрузчиком (но для файловой системы такие файлы ничем не отличаются от обычных).

1.2.2 ФАЙЛЫ-КАТАЛОГИ

Каталоги представляют собой особый вид файлов, которые хранятся во внешней памяти подобно обычным файлам, но структура которых поддерживается самой файловой системой. Каталоги позволяют организовывать иерархическую структуру файловой системы.

Фактически, каталог – это таблица, каждый элемент которой состоит из двух полей: номера i-узла данного файла в его файловой системе и имени файла, которое связано с этим номером (этот файл может быть и каталогом). Если просмотреть содержимое текущего рабочего каталога с помощью команды `ls -ai`, то можно получить, например, следующий вывод:

```
inode File
number      name
---------------
33          .
122         ..
54          first_file
65          second_file
65          second_again
77          dir2
```

Этот вывод демонстрирует, что в любом каталоге содержатся два стандартных имени – "." и "..". Имени "." сопоставляется i-узел 33, соответствующий текущему каталогу, а имени ".." – i-узел, соответствующий "родительскому" каталогу. Файлы с именами `"first_file"` и `"second_file"` - это разные файлы с номерами i-узлов 54 и 65 соответственно. Файл `"second_again"` представляет пример так называемой жесткой ссылки: он имеет другое имя, но реально описывается тем же i-узлом 65, что и файл `"second_file"`. Последний элемент каталога описывает некоторый другой каталог с именем `"dir2"`.

Для каталога возможно выполнение только специального набора системных вызовов, приведенных в таблице 1.2

Таблица 1.2

Команда	**Описание**
mkdir	создание нового каталога
rmdir	удаление пустого каталога
getdents	вывод содержимого указанного каталога

Непосредственная запись в файл-каталог недопустима. Запись в файлы-каталоги производится неявно при создании и уничтожении файлов и каталогов. Чтение из файла-каталога при наличии соответствующих прав возможно (пример - стандартная утилита `ls`, которая использует системный вызовом `getdents`).

1.2.3 СПЕЦИАЛЬНЫЕ ФАЙЛЫ

Специальные файлы не хранят данные. Они обеспечивают механизм отображения физических внешних устройств в имена файлов файловой системы. Каждому устройству, поддерживаемому системой, соответствует, по меньшей мере, один специальный файл. Специальные файлы создаются при

выполнении системного вызова mknod, каждому специальному файлу соответствует порция программного обеспечения, называемая драйвером соответствующего устройства. При выполнении чтения или записи по отношению к специальному файлу, производится вызов соответствующего драйвера, программный код которого отвечает за передачу данных между процессом пользователя и соответствующим физическим устройством.

Имена специальных файлов можно использовать практически везде, где можно использовать имена обычных файлов. Например, команда

```
cp myfile /tmp/kuz
```

перепишет файл с именем myfile в подкаталог kuz рабочего каталога. В то же время, команда

```
cp myfile /dev/console
```

выдаст содержимое файла myfile на системную консоль.

Различаются два типа специальных файлов - блочные и символьные. Блочные специальные файлы ассоциируются с такими внешними устройствами, обмен с которыми производится блоками байтов данных, размером 512, 1024, 4096 или 8192 байтов. Типичным примером подобных устройств являются магнитные диски. Файловые системы всегда находятся на блочных устройствах, так что в команде mount обязательно указывается некоторое блочное устройство.

Символьные специальные файлы ассоциируются с внешними устройствами, которые не обязательно требуют обмена блоками данных равного размера. Примерами таких устройств являются терминалы (в том числе, системная консоль), последовательные устройства, некоторые виды магнитных лент.

При обмене данными с блочным устройством система буферизует данные во внутреннем системном кеше. Через определенные интервалы времени система "выталкивает" данные из буфера в устройство. Недостатком такого обмена является то, что при аварийной остановке компьютера (например, при внезапном выключении электрического питания) содержимое системного кеша может быть утрачено, в результате внешние блочные файлы могут оказаться в рассогласованном состоянии. Указанная ошибка может быть исправлена системой, но не всегда без потери информации.

Обмены с символьными специальными файлами производятся без использования системной буферизации.

1.2.4 СВЯЗЫВАНИЕ ФАЙЛОВ С РАЗНЫМИ ИМЕНАМИ

Файловая система ОС UNIX обеспечивает возможность связывания одного и того же файла с разными именами. Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл) командного интерпретатора. Например, выполняемый файл традиционного текстового ре-

дактора ОС UNIX vi обычно может вызываться под именами ex, edit, vi, view и vedit.

Можно узнать имена всех связей данного файла с помощью команды ncheck, если указать в числе ее параметров номер i-узла интересующего файла. Например, чтобы узнать все имена, под которыми возможен вызов редактора vi, можно выполнить следующую последовательность команд (третий аргумент команды ncheck представляет собой имя специального файла, ассоциированного с файловой системой /usr):

```
ls -i /usr/bin/vi
результат:
372 /usr/bin/vi

ncheck -i 372 /dev/dsk/sc0d0s5
результат:
/dev/dsk/sc0d0s5:
372 /usr/bin/edit
372 /usr/bin/ex
372 /usr/bin/vedit
372 /usr/bin/vi
372 /usr/bin/view
```

Ранее в большинстве версий ОС UNIX поддерживались только так называемые "жесткие" связи, означающие, что в соответствующем каталоге имени связи сопоставлялось имя i-узла соответствующего файла. Новые жесткие связи могут создаваться с помощью системного вызова link. При выполнении этого системного вызова создается новый элемент каталога с тем же номером i-узла, что и ранее существовавший файл.

В поздних версиях ОС UNIX появились "символические связи". Символическая связь создается с помощью системного вызова symblink. При выполнении этого системного вызова в соответствующем каталоге создается элемент, в котором имени связи сопоставляется некоторое имя файла (этот файл даже не обязан существовать к моменту создания символической связи). Для символической связи создается отдельный i-узел и отводится отдельный блок данных для хранения потенциально длинного имени файла.

Для работы с символьными связями поддерживаются три специальных системных вызова, приведенные в таблице 1.3.

Таблица 1.3

Команда	**Описание**
readlink	читает имя файла, связанного с именуемой символической связью (это имя может соответствовать реальному файлу, специальному файлу, жесткой ссылке или вообще ничему); имя хранится в блоке данных, связанном с данной символической ссылкой
lstat	аналогичен системному вызову stat (получить информацию о файле), но относится к символической ссылке
lchowm	аналогичен системному вызову chown, но используется для смены пользователя и группы символической ссылки

1.2.5 ИМЕНОВАННЫЕ ПРОГРАММНЫЕ КАНАЛЫ

Программный канал (pipe) - это одно из наиболее традиционных средств межпроцессных взаимодействий в ОС UNIX.

Основной принцип работы программного канала состоит в буферизации байтового вывода одного процесса и обеспечении возможности чтения содержимого программного канала другим процессом в режиме FIFO. В любом случае интерфейс программного канала совпадает с интерфейсом файла (т.е. используются те же самые системные вызовы `read` и `write`).

Однако различаются два вида программных каналов – неименованные и именованные. Неименованный программный канал создается процессом-предком, наследуется процессами-потомками, и обеспечивает тем самым возможность связи в иерархии порожденных процессов. Интерфейс неименованного программного канала совпадает с интерфейсом файла, но, поскольку такие каналы не имеют имени, им не поставлен в соответствие элемент каталога в файловой системе.

Именованному программному каналу обязательно соответствует элемент некоторого каталога и собственный i-узел, т.е. именованный программный канал выглядит как обычный файл, но не содержащий никаких данных до тех пор, пока некоторый процесс не выполнит в него запись. После того, как некоторый другой процесс прочитает записанные в канал байты, этот файл снова становится пустым. Именованные программные каналы могут использоваться для связи любых процессов. Интерфейс именованного программного канала практически полностью совпадает с интерфейсом обычного файла, но их поведение несколько отличается.

1.2.6 ФАЙЛЫ, ОТОБРАЖАЕМЫЕ В ВИРТУАЛЬНУЮ ПАМЯТЬ

В современных версиях ОС UNIX имеется возможность отображать обычные файлы в виртуальную память процесса с последующей работой с содержимым файла не с помощью системных вызовов `read`, `write`, `lseek`, а с помощью обычных операций чтения из памяти и записи в память.

Для отображения файла в виртуальную память, после открытия файла выполняется системный вызов `mmap`, действие которого состоит в том, что создается сегмент разделяемой памяти, ассоциированный с открытым файлом, и автоматически подключается к виртуальной памяти процесса. После этого процесс может читать из нового сегмента и писать в него (реально чтение и запись будут осуществляться в/из файла). При закрытии файла соответствующий сегмент автоматически отключается от виртуальной памяти процесса и уничтожается, если только файл не подключен к виртуальной памяти некоторого другого процесса.

Несколько процессов могут одновременно открыть один и тот же файл и подключить его к своей виртуальной памяти системным вызовом `mmap`. То-

гда любые изменения, производимые путем записи в соответствующий сегмент разделяемой памяти, будут сразу видны другим процессам.

1.2.7 СИНХРОНИЗАЦИЯ ПРИ ПАРАЛЛЕЛЬНОМ ДОСТУПЕ К ФАЙЛАМ

В ранних версиях ОС средства синхронизации отсутствовали и решение этой проблемы возлагались на процессы.

В поздних версиях ОС UNIX был введен дополнительный системный вызов `fcntl`, обеспечивающий средства синхронизации. С помощью этого системного вызова можно установить монопольную или совместную блокировку файла целиком или блокировать указанный диапазон байтов внутри файла. Допускаются два варианта синхронизации: с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено, и без ожидания, когда процесс немедленно оповещается об удовлетворении требования блокировки или о невозможности ее удовлетворения в данный момент времени. При этом установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами-потомками.

1.3 ПРИНЦИПЫ ЗАЩИТЫ

Поскольку ОС UNIX разрабатывалась как многопользовательская система, в ней всегда была актуальна проблема авторизации доступа различных пользователей к файлам файловой системы. Под авторизацией доступа понимаются действия системы, которые допускают или не допускают доступ данного пользователя к данному файлу в зависимости от прав доступа пользователя и ограничений доступа, установленных для файла.

1.3.1 ИДЕНТИФИКАТОРЫ ПОЛЬЗОВАТЕЛЯ И ГРУППЫ ПОЛЬЗОВАТЕЛЕЙ

С каждым выполняемым процессом в ОС UNIX связываются реальный идентификатор пользователя (`real user ID`), действующий идентификатор пользователя (`effective user ID`) и сохраненный идентификатор пользователя (`saved user ID`). Все эти идентификаторы устанавливаются с помощью системного вызова `setuid`, который можно выполнять только в режиме суперпользователя. Аналогично, с каждым процессом связываются три идентификатора группы пользователей – `real group ID`, `effective group ID` и `saved group ID`. Эти идентификаторы устанавливаются привилегированным системным вызовом `setgid`.

При входе пользователя в систему программа `login` проверяет пользователя, образует новый процесс и запускает в нем требуемый для данного пользователя `shell`. Но перед этим `login` устанавливает для вновь создан-

ного процесса идентификаторы пользователя и группы, используя для этого информацию, хранящуюся в файлах `/etc/passwd` и `/etc/group`. После того, как с процессом связаны идентификаторы пользователя и группы, для этого процесса начинают действовать ограничения для доступа к файлам. Процесс может получить доступ к файлу или выполнить его только в том случае, если хранящиеся при файле ограничения доступа позволяют это сделать. Связанные с процессом идентификаторы передаются дочерним процессам, распространяя на них те же ограничения. Однако в некоторых случаях процесс может изменить свои права с помощью системных вызовов `setuid` и `setgid`, а иногда система может изменить права доступа процесса автоматически.

Пример. В файл `/etc/passwd` запрещена запись всем, кроме суперпользователя (суперпользователь может писать в любой файл). Этот файл, помимо прочего, содержит пароли пользователей и каждому пользователю разрешается изменять свой пароль. Имеется специальная программа `/bin/passwd`, изменяющая пароли. Однако пользователь не может сделать это даже с помощью этой программы, поскольку запись в файл `/etc/passwd` запрещена. В системе UNIX эта проблема разрешается следующим образом. При выполняемом файле может быть указано, что при его запуске должны устанавливаться идентификаторы пользователя и/или группы. Если пользователь запрашивает выполнение такой программы, то для соответствующего процесса устанавливаются идентификатор пользователя, соответствующий идентификатору владельца выполняемого файла и/или идентификатор группы этого владельца. В частности, при запуске программы `/bin/passwd` процесс получит идентификатор суперпользователя, и программа сможет произвести запись в файл `/etc/passwd`.

И для идентификатора пользователя, и для идентификатора группы реальный ID является истинным идентификатором, а действующий ID - идентификатором текущего выполнения. Если текущий идентификатор пользователя соответствует суперпользователю, то этот идентификатор и идентификатор группы могут быть переустановлены в любое значение системными вызовами `setuid` и `setgid`. Если же текущий идентификатор пользователя отличается от идентификатора суперпользователя, то выполнение системных вызовов `setuid` и `setgid` приводит к замене текущего идентификатора истинным идентификатором (пользователя или группы соответственно).

1.3.2 ЗАЩИТА ФАЙЛОВ

В UNIX любой процесс может получить доступ к некоторому файлу в том и только в том случае, если права доступа, описанные при файле, соответствуют возможностям данного процесса.

Защита файлов в ОС UNIX основывается на трех фактах:

- Во-первых, с любым процессом, создающим файл, ассоциирован некоторый идентификатор пользователя (UID), который в дальнейшем можно трактовать как идентификатор владельца вновь созданного файла.

- Во-вторых, с каждый процессом, пытающимся получить некоторый доступ к файлу, связана пара идентификаторов - текущие идентификаторы пользователя и его группы.
- В-третьих, каждому файлу однозначно соответствует его описатель - i-узел.

На последнем факте стоит остановиться более подробно. Любому i-узлу файловой системы всегда однозначно соответствует один и только один файл. I-узел содержит достаточно много разнообразной информации (большая ее часть доступна через системные вызовы `stat` и `fstat`), среди которой находится часть, позволяющая файловой системе оценить правомощность доступа данного процесса к данному файлу в требуемом режиме.

Общие принципы защиты одинаковы для всех существующих вариантов системы. Информация i-узла включает UID и GID текущего владельца файла (устанавливается при создании, но могут быть изменены вызовами `chown` и `chgrp`). Кроме того, в i-узле файла хранится шкала, в которой отмечено, что может делать с файлом его владелец, что могут делать с файлом пользователи, входящие в ту же группу, что и владелец, и что могут делать с файлом остальные пользователи. В таблице 1.4 приведена типичная шкала ограничений, содержащаяся в i-узле файла

Таблица 1.4

Шкала ограничений в восьмеричном виде	**Описание**
04000	Устанавливать идентификатор пользователя-владельца при выполнении файла.
020n0	При n = 7, 5, 3 или 1 устанавливать идентификатор группы владельца при выполнении файла. При n = 6, 4, 2 или 0 разрешается блокирование диапазонов адресов файла.
01000	Сохранять в области подкачки образ кодового сегмента выполняемого файла после конца его выполнения.
00400	Владельцу файла разрешено чтение файла.
00200	Владелец файла может дополнять или модифицировать файл.
00100	Владелец файла может его исполнять, если файл - исполняемый, или производить в нем поиск, если это файл-каталог.
00040	Все пользователи группы владельца могут читать файл.
00020	Все пользователи группы владельца могут дополнять или модифицировать файл.
00010	Все пользователи группы владельца могут исполнять файл, если файл - исполняемый, или производить в нем поиск, если это файл-каталог.
00004	Все пользователи могут читать файл.
00002	Все пользователи могут дополнять или модифицировать файл.
00001	Все пользователи могут исполнять файл, если файл - исполняемый, или производить в нем поиск, если это файл-каталог.

1.4 УПРАВЛЕНИЕ УСТРОЙСТВАМИ

Управление внешними устройствами – это одна из важнейших функций любой операционной системы. Система должна обеспечивать эффективный и

удобный доступ к периферийным устройствам, а также обеспечивать возможность унифицированной разработки программного обеспечения для вновь подключаемых внешних устройств.

Как уже было отмечено в UNIX это решается, используя универсальную абстракцию файла, т.е. внешним устройствам в UNIX ставятся в соответствие специальные файлы, что позволяет естественным образом работать в одном и том же интерфейсе с любым файлом или внешним устройством.

1.4.1 ДРАЙВЕРЫ УСТРОЙСТВ

Как и в любой ОС, в UNIX драйвер устройства – это многовходовой программный модуль со своими статическими данными, который умеет инициировать работу с устройством.

Драйверы делятся на символьные, блочные и потоковые драйверы.

Символьные драйверы являются простейшими в ОС UNIX и предназначаются для обслуживания устройств, ориентированных на прием или выдачу произвольной последовательностей байтов. Такие драйверы используют минимальный набор стандартных функций ядра UNIX, которые главным образом заключаются в возможности взять данные из виртуального пространства пользовательского процесса и/или поместить данные в такое виртуальное пространство.

Блочные драйверы - более сложные. Они работают с использованием возможностей системной буферизации блочных обменов ядра ОС UNIX. В число функций такого драйвера входит включение соответствующего блока данных в систему буферов ядра ОС UNIX и/или взятие содержимого буферной области в случае необходимости.

Наиболее сложной организацией отличаются **потоковые драйверы.** Фактически, такой драйвер представляет собой конвейер модулей, обеспечивающий многоступенчатую обработку запросов пользователя. Потоковые драйверы в среде ОС UNIX в основном предназначены для реализации доступа к сетевым устройствам, которые должны работать в соответствии с многоуровневыми сетевыми протоколами.

В ОС UNIX возможны два способа включения драйвера в состав ядра ОС. Первый способ состоит в полном включении драйвера в состав ядра на стадии генерации системы (т.е. драйвер является частью ядра системы). Второй способ позволяет обойтись минимальным количеством статических объявлений на стадии генерации ядра. Во втором случае в любой момент работы системы такой драйвер может быть динамически загружен в ядро системы. После появления (статического или динамического) в ядре ОС UNIX драйверы всех разновидностей функционируют одинаково.

1.4.2 ВНЕШНИЙ И ВНУТРЕННИЙ ИНТЕРФЕЙСЫ УСТРОЙСТВ

Независимо от типа файла (обычный файл, каталог, связь или специальный файл) пользовательский процесс может работать с файлом через стандартный интерфейс, включающий системные вызовы `open`, `close`, `read` и `write`. Ядро само распознает, нужно ли обратиться к его стандартным функциям или вызвать подпрограмму драйвера устройства (см. рисунок 1.3).

К рисунку: С каждым специальным файлом в системе связаны старший (`major`) и младший (`minor`) номера. После того, как (по содержанию i-узла) файловая система распознает, что данный файл является специальным, ядро ОС UNIX использует старший номер специального файла как индекс в конфигурационной таблице драйверов устройств. Поддерживаются две раздельные таблицы для символьных и блочных специальных файлов (или соответствующих драйверов). Для блочных драйверов используется системная таблица `bdevsw`, а для символьных – `cdevsw`. В обоих случаях элементом таблицы является структура (в терминах языка программирования Си), элементы которой содержат указатели на подпрограммы соответствующего драйвера. Допускается реализация драйверов, которые одновременно могут обрабатывать и блочный, и символьный ввод/вывод. В этом случае для драйвера будут существовать и элемент таблицы `bdevsw`, и таблицы `cdevsw`. Младший номер специального файла передается в качестве параметра соответствующей функции драйвера, обычно младший номер используется в качестве номера устройства, т.е. один драйвер как программная единица может управлять несколькими физическими устройствами.

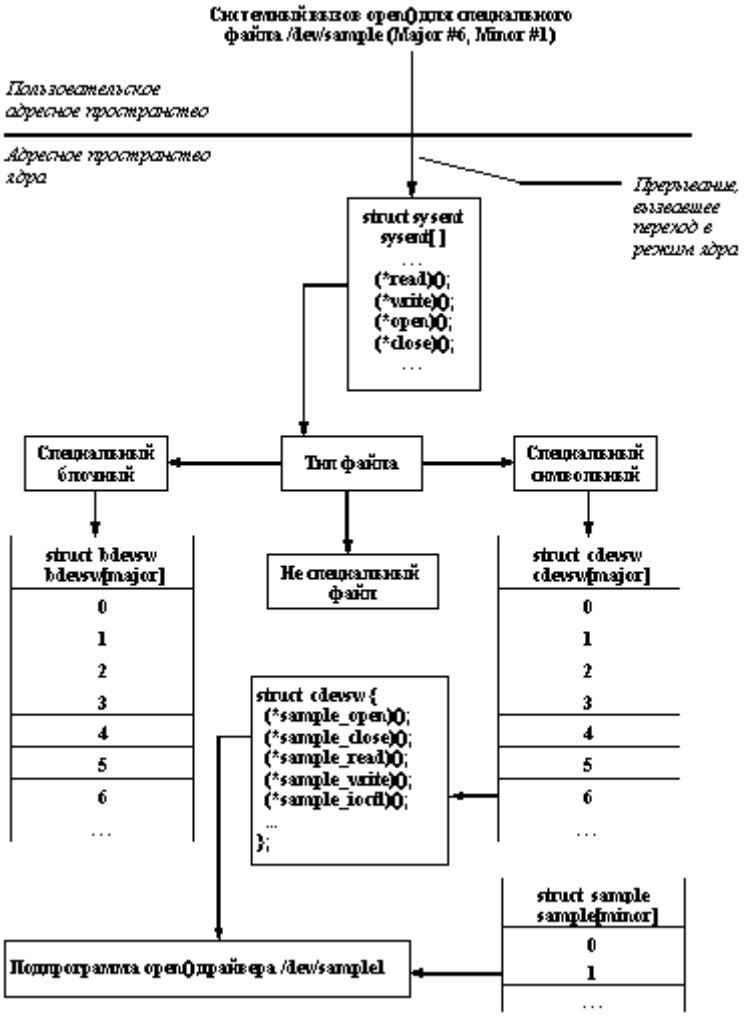

Рис. 1.3. Логическое представление открытия специального символьного файла