Часть шестая

Файловая система

Файл — это совокупность связанной информации, определённая его создателем. Файлы отображаются операционной системой на физических устройствах хранения данных. Файловая система описывает, как файлы отображаются на физических устройствах, а также как пользователи и программы получают к ним доступ и выполняют с ними операции.

Доступ к физическим хранилищам часто может быть медленным, поэтому файловые системы должны быть разработаны для эффективного доступа. Также могут быть важны другие требования, включая поддержку совместного доступа к файлам и удалённого доступа к ним.

13ГЛАВА

Файловая система Интерфейс

Для большинства пользователей файловая система является наиболее заметным аспектом операционной системы общего назначения. Она предоставляет механизм для онлайн-хранения и доступа к данным и программам операционной системы, а также всем пользователям компьютера. Файловая система состоит из двух отдельных частей: набора файлов, каждый из которых хранит связанные данные, и структуры каталогов, которая организует и предоставляет информацию обо всех файлах в системе. Большинство файловых систем размещаются на устройствах хранения данных, которые мы описали в главе 11 и продолжим обсуждать в следующей главе. В этой главе мы рассмотрим различные аспекты файлов и основные структуры каталогов. Мы также обсудим семантику совместного использования файлов несколькими процессами, пользователями и компьютерами. Наконец, мы обсудим способы защиты файлов, необходимые при наличии нескольких пользователей и необходимости контролировать, кто и как может получить доступ к файлам.

ЦЕЛИ ГЛАВЫ

• Объясните функции файловых систем. • Опишите интерфейсы файловых систем. • Обсудите компромиссы при проектировании файловых систем, включая методы доступа, совместное использование файлов. , блокировка файлов и структуры каталогов. • Изучите защиту файловой системы.

13.1 Концепция файла

Компьютеры могут хранить информацию на различных носителях, таких как энергонезависимые устройства (NVM) , жёсткие диски (HDD), магнитные ленты и оптические диски. Для удобства использования компьютерной системы операционная система обеспечивает единообразное логическое представление хранимой информации. Операционная система абстрагируется от физических свойств своих устройств хранения, чтобы определить логическую единицу хранения – fil . Файлы отображаются операционной системой на физических устройствах. Эти устройства хранения обычно энергонезависимы, поэтому их содержимое сохраняется между перезагрузками системы.

Файл — это именованный набор связанной информации, записанный во вторичном хранилище. С точки зрения пользователя, файл — это наименьший фрагмент логического вторичного хранилища; то есть данные не могут быть записаны во вторичное хранилище, если они не находятся в файле. Обычно файлы представляют собой программы (как исходные, так и объектные формы) и данные. Файлы данных могут быть числовыми, алфавитными, буквенно-цифровыми или двоичными. Файлы могут иметь свободную форму, например, текстовые, или быть жестко отформатированными. В общем случае, файл — это последовательность битов, байтов, строк или записей, значение которых определяется создателем и пользователем файла. Таким образом, понятие файла является чрезвычайно общим.

Поскольку файлы — это метод , используемый пользователями и приложениями для хранения и извлечения данных, а также их универсальность, их применение вышло за рамки первоначальной сферы применения. Например, UNIX, Linux и некоторые другие операционные системы предоставляют файловую систему proc , которая использует интерфейсы файловой системы для доступа к системной информации (например, к сведениям о процессах).

Информация в файле определяется его создателем. В файле может храниться множество различных типов информации: исходные или исполняемые программы, числовые или текстовые данные, фотографии, музыка, видео и так далее. Файл имеет определённую структуру, зависящую от его типа. Текстовый файл — это последовательность символов, организованных в строки (и, возможно, страницы). Исходный файл — это последовательность функций, каждая из которых организована в виде объявлений, за которыми следуют исполняемые операторы.
Исполняемый файл — это последовательность секций кода, которые загрузчик может загрузить в память и выполнить.

13.1.1 Атрибуты файла

Файлу присваивается имя для удобства пользователей, и он используется для обращения к нему по имени. Имя обычно представляет собой строку символов, например, example.c. В некоторых системах различаются заглавные и строчные буквы в именах, в то время как в других системах этого нет. Когда файлу присваивается имя, он становится независимым от процесса, пользователя и даже системы, которая его создала. Например, один пользователь может создать файл example.c, а другой может редактировать этот файл, указав его имя. Владелец файла может записать файл на USB- накопитель, отправить его как вложение по электронной почте или скопировать по сети, и он по-прежнему будет называться example.c в целевой системе. Если не используется метод общего доступа и синхронизации, эта вторая копия теперь независима от первой и может быть изменена отдельно.

Атрибуты файла различаются в зависимости от операционной системы, но обычно включают в себя следующее:

- Имя. Символическое имя файла – единственная информация, хранящаяся в памяти. читабельной форме.
- Идентификатор . Этот уникальный тег, обычно номер, идентифицирует файл в файловой системе; это нечитаемое для человека имя файла.
- Тип. Эта информация необходима для систем, поддерживающих разные типы. файлов.
- Местоположение. Эта информация представляет собой указатель на устройство и местоположение файла на этом устройстве.

- Размер. Текущий размер файла (в байтах, словах или блоках) и, возможно, максимально допустимый размер включен в этот атрибут.
- Защита. Информация о контроле доступа определяет, кто может читать, писать, выполнять и т. д.
- Временные метки и идентификация пользователя . Эта информация может храниться для даты создания, последнего изменения и последнего использования. Эти данные могут быть полезны для защиты, безопасности и мониторинга использования.

Некоторые новые файловые системы также поддерживают расширенные атрибуты файла, включая кодировку символов и функции безопасности, такие как контрольная сумма файла. На рисунке 13.1 показано окно информации о файле в macOS , в котором отображаются атрибуты файла.

Информация обо всех файлах хранится в структуре каталогов, которая находится на том же устройстве, что и сами файлы. Обычно запись каталога состоит из имени файла и его уникального идентификатора. Идентификатор, в свою очередь, определяет другие атрибуты файла. Запись этой информации для каждого файла может занять более килобайта. В системе с большим количеством файлов размер самого каталога может составлять мегабайты или гигабайты. Поскольку каталоги должны соответствовать изменчивости файлов, как и сами файлы, они должны храниться на устройстве и обычно переносятся в память по частям, по мере необходимости.

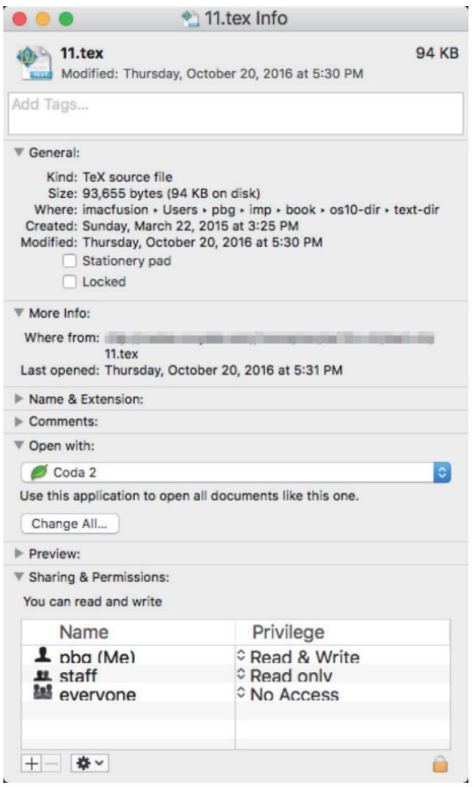


Рисунок 13.1 Окно информации о файле в macOS.

13.1.2 Операции с файлами

Файл — это абстрактный тип данных. Чтобы правильно определить файл, необходимо рассмотреть операции, которые можно выполнять над файлами. Операционная система может предоставлять системные вызовы для создания, записи, чтения, перемещения, удаления и усечения файлов. Давайте рассмотрим, что должна делать операционная система для выполнения каждой из этих семи основных операций с файлами. После этого будет легко понять, как можно реализовать другие подобные операции, например, переименование файла.

- Создание файла fil . Для создания файла необходимо два шага. Во-первых, необходимо найти место в файловой системе для файла. Мы обсудим, как выделить место для файла, в главе 14. Во-вторых, необходимо создать запись для нового файла в каталоге.

- Открытие файла fil . Вместо того, чтобы все файловые операции задавали имя файла, заставляя операционную систему оценивать имя, проверять права доступа и т. д., все операции, кроме создания и удаления, требуют предварительного вызова функции file open() . В случае успеха вызов open возвращает дескриптор файла, который используется в качестве аргумента в других вызовах.

- Запись файла fil . Для записи файла мы выполняем системный вызов, указывающий как дескриптор открытого файла, так и информацию, которую нужно записать в файл. Система должна хранить указатель записи на место в файле, куда должна быть выполнена следующая запись, если она последовательная. Указатель записи должен обновляться при каждой записи.

- Чтение файла fil . Для чтения из файла мы используем системный вызов, который указывает дескриптор файла и место (в памяти), где должен быть помещен следующий блок файла. Система также должна хранить указатель чтения на место в файле, откуда должно быть выполнено следующее чтение, если оно последовательное. После завершения чтения указатель чтения обновляется. Поскольку процесс обычно либо читает из файла, либо записывает в него, текущее положение операции можно сохранить как указатель current-file-positio для каждого процесса. Операции чтения и записи используют один и тот же указатель, что экономит место и снижает сложность системы.

- Изменение положения в файле. Указатель текущей позиции открытого файла перемещается на заданное значение. Изменение положения в файле не требует выполнения каких-либо операций ввода -вывода. Эта файловая операция также называется поиском в файле.

- Удаление файла . Чтобы удалить файл, мы ищем указанный файл в каталоге. Найдя соответствующую запись каталога, мы освобождаем всё файловое пространство, чтобы его можно было использовать другими файлами, и стираем или отмечаем запись каталога как свободную. Обратите внимание, что некоторые системы допускают жёсткие ссылки — несколько имён (записей каталога) для одного и того же файла. В этом случае фактическое содержимое файла не удаляется, пока не будет удалена последняя ссылка.

- Усечение файла fil . Пользователь может захотеть удалить содержимое файла, сохранив его атрибуты. Вместо того, чтобы заставлять пользователя удалять файл и создавать его заново, эта функция позволяет сохранить все атрибуты без изменений, за исключением длины файла. После этого длину файла можно сбросить до нуля, освободив при этом занимаемое им файловое пространство.

Эти семь базовых операций составляют минимальный набор необходимых файловых операций. Другие распространённые операции включают добавление новой информации в конец существующего файла и переименование существующего файла. Эти примитивные операции затем можно комбинировать для выполнения других файловых операций. Например, мы можем создать копию файла, создав новый файл, затем прочитав данные из старого и записав данные в новый. Также нам нужны операции, позволяющие пользователю получать и устанавливать различные атрибуты файла. Например, нам могут понадобиться операции, позволяющие пользователю определять статус файла, например, его длину, и устанавливать атрибуты файла, например, владельца.

Как уже упоминалось, большинство упомянутых файловых операций включают поиск записи, связанной с указанным файлом, в каталоге. Чтобы избежать этого постоянного поиска, многие системы требуют выполнения системного вызова open() перед первым использованием файла. Операционная система хранит таблицу, называемую таблицей open-fil, содержащую информацию обо всех открытых файлах. При запросе файловой операции файл указывается через индекс в этой таблице, поэтому поиск не требуется.
Когда файл больше не используется активно, он закрывается процессом, и операционная система удаляет его запись из таблицы открытых файлов, что может привести к снятию блокировок. create() и delete() — это системные вызовы, которые работают с закрытыми, а не с открытыми файлами.

Некоторые системы неявно открывают файл при первой ссылке на него.
Файл автоматически закрывается после завершения задания или программы, открывшей файл. Однако в большинстве систем требуется, чтобы программист явно открыл файл с помощью системного вызова open(), прежде чем этот файл можно будет использовать.
Операция open() принимает имя файла и выполняет поиск в каталоге, копируя запись каталога в таблицу открытых файлов. Вызов open() также может принимать информацию о режиме доступа — создание, только чтение, чтение-запись, только добавление и т. д.
Этот режим проверяется на соответствие разрешениям файла. Если режим запроса разрешён, файл открывается для процесса. Системный вызов open() обычно возвращает указатель на запись в таблице открытых файлов. Этот указатель, а не фактическое имя файла, используется во всех операциях ввода-вывода , что позволяет избежать дальнейшего поиска и упростить интерфейс системного вызова.

Реализация операций open() и close() усложняется в среде, где несколько процессов могут одновременно открывать файл. Это может произойти в системе, где несколько различных приложений одновременно открывают один и тот же файл. Обычно операционная система использует два уровня внутренних таблиц: таблицу для каждого процесса и общесистемную таблицу. Таблица для каждого процесса отслеживает все файлы, открытые процессом. В этой таблице хранится информация об использовании файла процессом. Например, здесь находится текущий указатель на файл для каждого файла. Также могут быть включены права доступа к файлу и учётная информация.

Каждая запись в таблице по процессам, в свою очередь, указывает на общесистемную таблицу открытых файлов. Эта общесистемная таблица содержит информацию, не зависящую от процесса, такую как расположение файла на диске, даты доступа и размер файла. После открытия файла одним процессом запись об этом файле добавляется в общесистемную таблицу. Когда другой процесс выполняет вызов open() , в таблицу открытых файлов этого процесса просто добавляется новая запись, указывающая на соответствующую запись в общесистемной таблице. Обычно в таблице открытых файлов для каждого файла также указывается счётчик открытых файлов, показывающий, сколько процессов открыли этот файл. Каждый вызов close() уменьшает этот счётчик открытых файлов, и когда счётчик открытых файлов достигает нуля, файл перестаёт использоваться, и запись о нём удаляется из таблицы открытых файлов.

БЛОКИРОВКА ФАЙЛОВ В JAVA

В API Java для получения блокировки требуется сначала получить FileChannel для блокируемого файла. Метод lock() FileChannel используется для получения блокировки.

API метода lock () :

Блокировка FileLock (длинное начало, длинный конец, логическое общее)

где begin и end — начальная и конечная позиции блокируемой области. Установка значения shared в true используется для разделяемых блокировок; установка значения shared в false устанавливает монопольную блокировку. Блокировка снимается вызовом метода release() объекта FileLock, возвращаемого операцией lock() .

Программа на рисунке 13.2 иллюстрирует блокировку файла в Java. Эта программа устанавливает две блокировки на файл file.txt. Блокировка первой половины файла является исключительной, а второй — разделяемой.

Подводя итог, можно сказать, что с открытым файлом связано несколько фрагментов информации.

- Указатель файла. В системах, где смещение файла не используется в системных вызовах read() и write() , система должна отслеживать последнее место чтения-записи как указатель текущей позиции файла. Этот указатель уникален для каждого процесса, работающего с файлом, и поэтому должен храниться отдельно от атрибутов файла на диске.

- Счётчик открытых файлов. По мере закрытия файлов операционная система должна повторно использовать записи в таблице открытых файлов, иначе в таблице может закончиться место. Файл может быть открыт несколькими процессами, и системе придётся дождаться закрытия последнего файла, прежде чем удалить запись из таблицы открытых файлов. Счётчик открытых файлов отслеживает количество открытий и закрытий и достигает нуля при последнем закрытии. После этого система может удалить запись. •

Расположение файла fil . Большинство файловых операций требуют от системы чтения или записи данных из файла. Информация, необходимая для поиска файла (где бы он ни находился, будь то на запоминающем устройстве, на файловом сервере в сети или на RAM- диске), хранится в памяти, поэтому системе не приходится считывать её из структуры каталогов для каждой операции.

- Права доступа. Каждый процесс открывает файл в определённом режиме доступа. Эта информация хранится в таблице каждого процесса, чтобы операционная система могла разрешать или запрещать последующие запросы ввода-вывода .

Некоторые операционные системы предоставляют возможности блокировки открытого файла (или его разделов). Блокировки файлов позволяют одному процессу заблокировать файл и запретить другим процессам доступ к нему. Блокировки файлов полезны для файлов, используемых несколькими процессами, например, для файла системного журнала, доступ к которому и его изменение могут получить несколько процессов в системе.

Блокировки файлов обеспечивают функциональность, аналогичную блокировкам чтения-записи, описанным в разделе 7.1.2. Совместная блокировка похожа на блокировку чтения тем, что несколько процессов могут одновременно получить блокировку. Исключительная блокировка ведёт себя как блокировка записи: только один процесс может получить такую блокировку одновременно. Важно отметить, что не

```
импорт java.io.*; импорт
java.nio.channels.*;

открытый класс LockingExample {
  public static final boolean EXCLUSIVE = false; public static final boolean
  SHARED = true;

  public static void main(String args[]) выдает IOException {
    FileLock sharedLock = null;
    FileLockexclusiveLock = ноль;


    попробуйте { RandomAccessFile raf = new RandomAccessFile("file.txt","rw");

      // получить канал для файла FileChannel ch =
      raf.getChannel();

      // это блокирует первую половину файла - эксклюзивно exclusiveLock = ch.lock(0,
      raf.length()/2, EXCLUSIVE);

      /** Теперь измените данные.          .  . */

      // снять блокировку
      эксклюзивнаяБлокировка.выпуск();

      // это блокирует вторую половину файла - общий sharedLock = ch.lock(raf.length()/
      2+1,raf.length(),SHARED);

      /** Теперь прочитаем данные.          . */

      // снять блокировку
      sharedLock.release();
    } поймать (java.io.IOException ioe) {
      System.err.println(ioe);
    }
    наконец
      { если (exclusiveLock != null)
          эксклюзивнаяБлокировка.выпуск();
      если (sharedLock != null)
          sharedLock.release();


} } }
```

Рисунок 13.2 Пример блокировки файла в Java.

все операционные системы предоставляют оба типа блокировок: некоторые системы предоставляют только исключительную блокировку файлов.

Кроме того, операционные системы могут предоставлять как обязательные , так и рекомендательные механизмы блокировки файлов. При обязательной блокировке, как только процесс получает исключительную блокировку, операционная система запрещает любому другому процессу

Доступ к заблокированному файлу. Например, предположим, что процесс устанавливает исключительную блокировку файла system.log. При попытке открыть system.log из другого процесса, например, текстового редактора, операционная система запретит доступ до тех пор, пока исключительная блокировка не будет снята. Если же блокировка рекомендательная, операционная система не будет препятствовать текстовому редактору получить доступ к sys-tem.log. Текстовый редактор должен быть написан таким образом, чтобы он вручную устанавливал блокировку перед доступом к файлу. Другими словами, если схема блокировки обязательна, операционная система обеспечивает целостность блокировки. В случае рекомендательной блокировки разработчики программного обеспечения должны обеспечить правильное получение и снятие блокировок. Как правило, операционные системы Windows используют обязательную блокировку, а системы UNIX — рекомендательную.

Использование блокировок файлов требует тех же мер предосторожности, что и обычная синхронизация процессов. Например, программисты, работающие в системах с обязательной блокировкой, должны соблюдать осторожность и устанавливать исключительные блокировки файлов только во время доступа к ним. В противном случае они помешают другим процессам получить доступ к файлу. Кроме того, необходимо принять меры, чтобы предотвратить взаимоблокировку двух или более процессов при попытке получения блокировок файлов.

13.1.3 Типы файлов

При проектировании файловой системы, а по сути, и всей операционной системы в целом, мы всегда учитываем, должна ли операционная система распознавать и поддерживать типы файлов. Если операционная система распознаёт тип файла, она может работать с ним разумным образом. Например, распространённая ошибка возникает, когда пользователь пытается вывести программу в виде двоичного объекта. Эта попытка обычно приводит к ошибке; однако она может быть успешной, если операционной системе известно, что файл представляет собой программу в виде двоичного объекта.

Распространенным способом реализации типов файлов является включение типа в имя файла. Имя разделяется на две части: имя и расширение, обычно разделённые точкой (рис. 13.3). Таким образом, пользователь и операционная система могут определить тип файла по одному только имени. Большинство операционных систем позволяют пользователям указывать имя файла в виде последовательности символов, за которой следует точка и расширение, состоящее из дополнительных символов. Примеры включают resume.docx, server.c и ReaderThread.cpp.

Система использует расширение для указания типа файла и типа операций, которые можно с ним выполнять. Например, можно выполнить только файл с расширением .com, .exe или .sh . Файлы .com и .exe представляют собой два вида двоичных исполняемых файлов, тогда как файл .sh представляет собой скрипт оболочки , содержащий команды операционной системе в формате ASCII . Прикладные программы также используют расширения для указания интересующих их типов файлов. Например, компиляторы Java ожидают, что исходные файлы будут иметь расширение .java , а текстовый процессор Microsoft Word ожидает, что его файлы будут иметь расширение .doc или .docx .
Эти расширения не всегда обязательны, поэтому пользователь может указать файл без расширения (чтобы сэкономить время на ввод), и приложение будет искать файл с указанным именем и ожидаемым расширением. Поскольку эти расширения не поддерживаются операционной системой, их можно считать «подсказками» для приложений, которые с ними работают.

Рассмотрим также операционную систему macOS. В этой системе каждый файл имеет тип, например, .app (для приложения). У каждого файла также есть атрибут создателя.

тип файла	обычное расширение	функция
исполняемый файл	exe, com, bin или нет	готовая к запуску программа на машинном языке,
объект	объект, о	скомпилированная, машина язык, не связанный
исходный код с, cc, java, perl, асм		исходный код в различных языки
партия	летучая мышь, ш	команды команде устный переводчик
разметка	xml, html, tex	текстовые данные, документы
текстовый процессор	xml, rtf, docx	различные текстовые процессоры форматы
библиотека	lib, a, so, dll	библиотеки процедур для программисты
распечатать или просмотреть	gif, pdf, jpg	ASCII или двоичный файл в формат для печати или просмотр
архив	rar, zip, tar	связанные файлы сгруппированы в один файл, иногда сжатый, для архивирования или хранение
мультимедиа	mpeg, mov, mp3, mp4, avi	двоичный файл, содержащий аудио или аудио/видео информация

Рисунок 13.3 Распространенные типы файлов.

содержащее имя программы, создавшей его. Этот атрибут устанавливается операционной системы во время вызова create() , поэтому его использование является обязательным и поддерживаемые системой. Например, файл, созданный текстовым редактором, Имя создателя файла – текстовый процессор. Когда пользователь открывает этот файл, дважды щелкнув мышью по значку, представляющему файл, текстовый процессор вызывается автоматически, и файл загружается, готовый к редактированию.

Система UNIX использует магическое число, хранящееся в начале некоторых двоичные файлы для указания типа данных в файле (например, формат файла изображения). Аналогично, он использует магическое число в начале текста файлы для указания типа файла (на каком языке оболочки написан скрипт) и так далее. (Более подробную информацию о магических числах и прочем компьютерном жаргоне см. см. http://www.catb.org/esr/jargon/.) Не все файлы имеют магические числа, поэтому Возможности системы не могут основываться исключительно на этой информации. UNIX не Также запишите имя создающей программы. UNIX допускает подсказки по расширению имени файла, но эти расширения не являются обязательными и не зависят от операционная система; они предназначены в основном для того, чтобы помочь пользователям определить, что Тип содержимого файла. Расширения могут быть использованы или проигнорированы. приложение, но это зависит от программиста приложения.

13.1.4 Структура файла

Типы файлов также могут использоваться для указания внутренней структуры файла. Источник и объектные файлы имеют структуры, которые соответствуют ожиданиям программ которые их читают. Кроме того, некоторые файлы должны соответствовать требуемой структуре,

распознаётся операционной системой. Например, операционная система требует, чтобы исполняемый файл имел определённую структуру, чтобы она могла определить, куда в памяти загрузить файл и где находится первая инструкция. Некоторые операционные системы расширяют эту идею, предлагая набор поддерживаемых системой файловых структур с наборами специальных операций для работы с файлами.
структуры.

Это подводит нас к одному из недостатков поддержки операционной системой нескольких файловых структур: это делает операционную систему громоздкой и громоздкой. Если операционная система определяет пять различных файловых структур, ей необходимо содержать код для их поддержки. Кроме того, может потребоваться определить каждый файл как один из поддерживаемых операционной системой типов. Когда новым приложениям требуется информация, структурированная способом, не поддерживаемым операционной системой, могут возникнуть серьёзные проблемы.

Например, предположим, что система поддерживает два типа файлов: текстовые файлы (состоящие из символов ASCII, разделённых символами возврата каретки и перевода строки) и исполняемые двоичные файлы. Если мы (как пользователи) захотим определить зашифрованный файл для защиты его содержимого от несанкционированного чтения, ни один из этих типов файлов может оказаться неподходящим. Зашифрованный файл представляет собой не текстовые строки ASCII , а (на первый взгляд) случайные биты. Хотя он может выглядеть как двоичный файл, он не является исполняемым. В результате нам, возможно, придётся обойти или неправильно использовать механизм типов файлов операционной системы или отказаться от нашей схемы шифрования.

Некоторые операционные системы устанавливают (и поддерживают) минимальное количество файловых структур. Этот подход принят в UNIX, Windows и других системах.
UNIX рассматривает каждый файл как последовательность 8-битных байтов; операционная система не интерпретирует эти биты. Эта схема обеспечивает максимальную гибкость, но ограниченную поддержку. Каждая прикладная программа должна включать собственный код для интерпретации входного файла в соответствии с его структурой. Однако все операционные системы должны поддерживать как минимум одну структуру — исполняемый файл — чтобы система могла загружать и запускать программы.

13.1.5 Внутренняя структура файла

На внутреннем уровне определение смещения внутри файла может быть сложным для операционной системы. Дисковые системы обычно имеют чётко определённый размер блока, определяемый размером сектора. Все операции ввода-вывода на диске выполняются блоками по одному блоку (физической записи), и все блоки имеют одинаковый размер. Маловероятно, что размер физической записи будет точно соответствовать длине желаемой логической записи. Длина логических записей может даже различаться. Упаковка нескольких логических записей в физические блоки — распространённое решение этой проблемы.

Например, операционная система UNIX определяет все файлы как потоки байтов. Каждый байт адресуется индивидуально по смещению от начала (или конца) файла. В этом случае размер логической записи составляет 1 байт. Файловая система автоматически упаковывает и распаковывает байты в физические дисковые блоки, например, по 512 байт на блок, по мере необходимости.

Размер логической записи, размер физического блока и метод упаковки определяют количество логических записей в каждом физическом блоке. Упаковка может выполняться как пользовательским приложением, так и операционной системой. В любом случае файл можно рассматривать как последовательность блоков. Все основные функции ввода-вывода работают с блоками. Преобразование логических записей в физические блоки — относительно простая программная задача.

Поскольку дисковое пространство всегда выделяется блоками, часть последнего блока каждого файла, как правило, тратится впустую. Например, если бы каждый блок был размером 512 байт, то файлу размером 1949 байт было бы выделено четыре блока (2048 байт); последние 99 байт были бы потрачены впустую. Потери, возникающие при хранении данных в блоках (а не в байтах), — это внутренняя фрагментация. Все файловые системы страдают от внутренней фрагментации: чем больше размер блока, тем сильнее внутренняя фрагментация.

13.2 Методы доступа

Файлы хранят информацию. При использовании к этой информации необходимо получить доступ и прочитать её в память компьютера. Доступ к информации в файле возможен несколькими способами. Некоторые системы предоставляют только один метод доступа к файлам. Другие (например, операционные системы для мэйнфреймов) поддерживают множество методов доступа, и выбор подходящего метода для конкретного приложения представляет собой серьёзную проблему проектирования.

13.2.1 Последовательный доступ

Самый простой метод доступа — последовательный. Информация в файле обрабатывается последовательно, одна запись за другой. Этот режим доступа, безусловно, самый распространённый; например, редакторы и компиляторы обычно обращаются к файлам именно таким образом.

Чтение и запись составляют большую часть операций с файлом. Операция чтения — read next() — считывает следующую часть файла и автоматически перемещает указатель файла, отслеживающий позицию ввода-вывода . Аналогично, операция записи — write next() — добавляет данные в конец файла и перемещает их в конец только что записанного материала (новый конец файла). Такой файл можно сбросить в начало, и в некоторых системах программа может пропускать n записей вперёд или назад для некоторого целого числа n — возможно, только для n = 1. Последовательный доступ, представленный на рисунке 13.4, основан на ленточной модели файла и работает как на устройствах с последовательным доступом, так и на устройствах с произвольным доступом.

13.2.2 Прямой доступ

Другой метод — прямой доступ (или относительный доступ). В этом случае файл состоит из логических записей фиксированной длины , что позволяет программам быстро читать и записывать записи в произвольном порядке. Метод прямого доступа основан на дисковой модели файла, поскольку диски обеспечивают произвольный доступ к любому блоку файла. При прямом доступе файл рассматривается как пронумерованная последовательность блоков или записей. Таким образом,

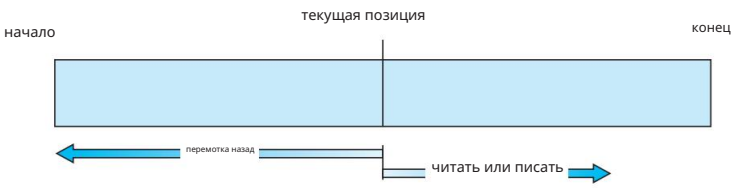


Рисунок 13.4 Файл последовательного доступа.

мы можем прочитать блок 14, затем прочитать блок 53, а затем записать блок 7. Для файла прямого доступа нет ограничений на порядок чтения или записи.

Файлы прямого доступа очень полезны для мгновенного доступа к большим объёмам информации. Базы данных часто относятся к этому типу. При поступлении запроса по определённой теме мы вычисляем, какой блок содержит ответ, а затем напрямую считываем этот блок, чтобы получить необходимую информацию.

В качестве простого примера: в системе бронирования авиабилетов мы можем хранить всю информацию о конкретном рейсе (например, рейсе 713) в блоке, идентифицируемом номером рейса. Таким образом, количество доступных мест на рейс 713 хранится в блоке 713 файла бронирования. Для хранения информации о более широком наборе данных, например, о людях, мы можем вычислить хеш-функцию для имён людей или выполнить поиск по небольшому индексу в памяти, чтобы определить блок для чтения и поиска.

Для метода прямого доступа файловые операции необходимо модифицировать, включив номер блока в качестве параметра. Таким образом, вместо read next( ) мы используем read(n), где n — номер блока, и write(n) , а не write next(). Альтернативный подход заключается в сохранении read next() и write next() , а также добавлении операции position file(n), где n — номер блока. Тогда для выполнения read(n) мы бы позиционировали file(n) , а затем read next().

Номер блока, предоставляемый пользователем операционной системе, обычно является относительным номером блока. Относительный номер блока – это индекс относительно начала файла. Таким образом, первый относительный номер блока файла имеет номер 0, следующий – 1 и так далее, даже если абсолютный адрес на диске может быть 14703 для первого блока и 3192 для второго. Использование относительных номеров блоков позволяет операционной системе определить, где следует разместить файл (это называется проблемой распределения, как мы обсудим в главе 14), и помогает предотвратить доступ пользователя к тем частям файловой системы, которые могут не быть частью его файла. В некоторых системах относительные номера блоков начинаются с 0, в других – с 1.

Как же тогда система удовлетворяет запрос на запись N в файле? Если предположить, что длина логической записи L, запрос на запись N преобразуется в запрос ввода-вывода на L байтов, начиная с позиции L (N) в файле (предполагая, что первая запись N = 0). Поскольку логические записи имеют фиксированный размер, их также легко читать, записывать и удалять.

Не все операционные системы поддерживают как последовательный, так и прямой доступ к файлам. Некоторые системы допускают только последовательный доступ к файлам, другие — только прямой. В некоторых системах требуется, чтобы файл был определён как последовательный или прямой при его создании. К такому файлу можно получить доступ только способом, соответствующим его объявлению. Мы можем легко имитировать последовательный доступ к файлу с прямым доступом, просто сохранив переменную cp , определяющую нашу текущую позицию, как показано на рисунке 13.5. Однако имитация файла с прямым доступом к файлу с последовательным доступом крайне неэффективна и громоздка.

13.2.3 Другие методы доступа

Другие методы доступа могут быть построены на основе метода прямого доступа. Эти методы обычно предполагают создание индекса для файла. Индекс , подобно указателю в конце книги, содержит указатели на различные блоки. Чтобы найти запись в файле, мы сначала выполняем поиск по индексу, а затем используем указатель для прямого доступа к файлу и нахождения нужной записи.

последовательный доступ	реализация для прямого доступа
перезагрузить	ср 0;
read_next	читать ср; ср ср 1;
написать_следующий	написать ср; ср ср 1;

Рисунок 13.5 Моделирование последовательного доступа к файлу прямого доступа.

Например, файл с розничными ценами может содержать универсальные коды товаров (UPC) и соответствующие цены. Каждая запись состоит из 10-значного UPC и 6-значной цены, что составляет 16-байтовую запись. Если на диске блок занимает 1024 байта, мы можем хранить 64 записи в каждом блоке. Файл из 120 000 записей займёт около 2000 блоков (2 миллиона байт). Сортируя файл по UPC, мы можем определить индекс, состоящий из первого UPC в каждом блоке. Этот индекс будет содержать 2000 записей по 10 цифр каждая, или 20 000 байт, и, таким образом, может храниться в памяти. Чтобы найти цену конкретного товара, мы можем выполнить двоичный поиск по индексу.

В результате этого поиска мы точно узнаём, какой блок содержит нужную запись, и получаем доступ к этому блоку. Такая структура позволяет нам выполнять поиск в большом файле, выполняя минимум операций ввода-вывода.

При работе с большими файлами сам индексный файл может стать слишком большим для хранения в памяти. Одним из решений является создание индекса для индексного файла. Первичный индексный файл содержит указатели на вторичные индексные файлы, которые указывают на фактические элементы данных.

Например, метод индексированного последовательного доступа (ISAM) IBM использует небольшой главный индекс, который указывает на дисковые блоки вторичного индекса. Блоки вторичного индекса указывают на сами блоки файла. Файл сортируется по заданному ключу. Чтобы найти нужный элемент, мы сначала выполняем двоичный поиск по главному индексу, который возвращает номер блока вторичного индекса. Этот блок считывается, и снова двоичный поиск используется для поиска блока, содержащего нужную запись. Наконец, этот блок ищется последовательно. Таким образом, любую запись можно найти по её ключу максимум за два чтения с прямым доступом. На рисунке 13.6 показана аналогичная ситуация, реализованная индексом OpenVMS и связанными файлами.

13.3 Структура каталога

Каталог можно рассматривать как таблицу символов, которая преобразует имена файлов в управляющие блоки. Если рассматривать этот подход, то сам каталог может быть организован различными способами. Организация должна позволять добавлять записи, удалять записи, искать именованную запись и выводить список всех записей в каталоге. В этом разделе мы рассмотрим несколько схем определения логической структуры системы каталогов.

При рассмотрении конкретной структуры каталога необходимо учитывать операции, которые будут выполняться над каталогом:

- Поиск файла fil . Нам необходимо иметь возможность поиска в структуре каталогов, чтобы найти запись для конкретного файла. Поскольку файлы имеют символические имена и подобные

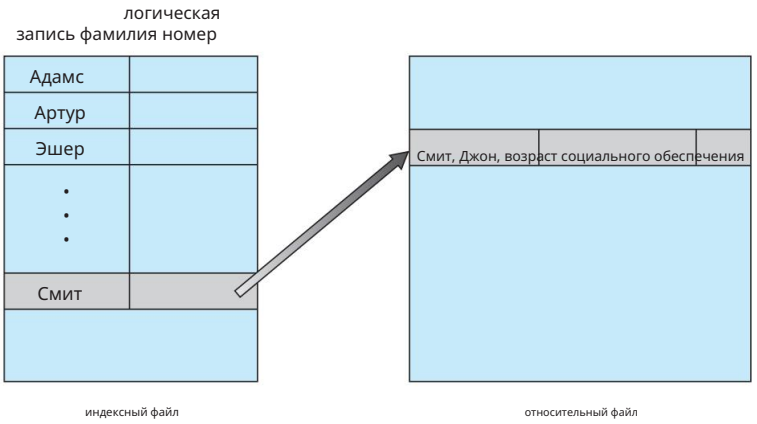


Рисунок 13.6 Пример индексных и относительных файлов.

Имена могут указывать на связь между файлами, нам может потребоваться найти все файлы, имена которых соответствуют определенному шаблону.

- Создайте файл fil . Необходимо создать новые файлы и добавить их в каталог.
- Удаление файла. Когда файл больше не нужен, мы хотим иметь возможность удалить его из каталога. Обратите внимание, что удаление оставляет дыру в структуре каталога, и файловая система может иметь метод дефрагментации каталога. структура.
- Вывести список каталога. Нам необходимо иметь возможность вывести список файлов в каталоге и содержимое записи каталога для каждого файла в списке.
- Переименование файла. Поскольку имя файла отражает его содержимое для пользователей, необходимо иметь возможность изменить имя при изменении содержимого или назначения файла. Переименование файла также может привести к изменению его положения в структуре каталогов.
- Обход файловой системы. Нам может потребоваться доступ к каждому каталогу и каждому файлу в структуре каталогов. Для обеспечения надёжности рекомендуется регулярно сохранять содержимое и структуру всей файловой системы. Часто для этого мы копируем все файлы на магнитную ленту, другие вторичные носители или по сети в другую систему или облако. Этот метод обеспечивает резервное копирование на случай сбоя системы. Кроме того, если файл больше не используется, его можно скопировать в резервную копию, а дисковое пространство, занимаемое этим файлом, освободить для повторного использования другим файлом.

В следующих разделах мы опишем наиболее распространенные схемы определения логической структуры каталога.

13.3.1 Одноуровневый каталог. Простейшая

структура каталогов — одноуровневый. Все файлы хранятся в одном каталоге, что упрощает поддержку и понимание (рис. 13.7).

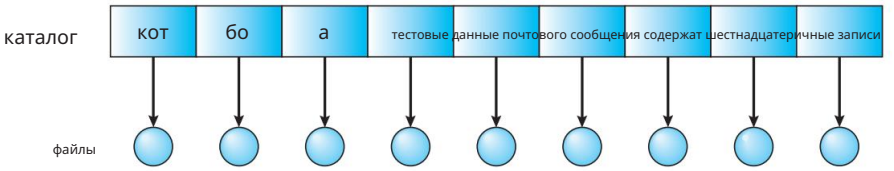


Рисунок 13.7 Одноуровневый каталог.

Однако одноуровневый каталог имеет существенные ограничения, когда
Количество файлов увеличивается или когда в системе больше одного пользователя. Поскольку все
Файлы находятся в одном каталоге, у них должны быть уникальные имена. Если два пользователя вызывают
В их файле данных test.txt нарушается правило уникальности имени. Например,
На одном из занятий по программированию 23 студента назвали программу своей второй
присваивание prog2.c; ещё 11 назвали его assign2.c. К счастью, большинство файлов
Системы поддерживают имена файлов длиной до 255 символов, поэтому их относительно легко
выберите уникальные имена файлов.

Даже одному пользователю одноуровневого каталога может быть сложно запомнить имена всех файлов по мере их увеличения. Это не редкость.

для пользователя иметь сотни файлов на одной компьютерной системе и равное количество
Количество дополнительных файлов в другой системе. Отслеживать такое количество файлов – задача не из лёгких.
сложная задача.

13.3.2 Двухуровневый каталог

Как мы видели, одноуровневый каталог часто приводит к путанице в именах файлов.
между разными пользователями. Стандартное решение — создать отдельный каталог.
для каждого пользователя.

В двухуровневой структуре каталогов каждый пользователь имеет свой собственный каталог файлов (UFD).
UFD имеют схожую структуру, но каждый содержит только файлы
один пользователь. Когда пользовательское задание запускается или пользователь входит в систему, главный сервер системы
Поиск в каталоге fil (MFD) выполняется. MFD индексируется по имени пользователя или учётной записи.
номер, и каждая запись указывает на УФД для этого пользователя (рисунок 13.8).

Когда пользователь обращается к определённому файлу, поиск выполняется только по его собственному UFD-файлу . Таким образом,
У разных пользователей могут быть файлы с одинаковыми именами, если все имена файлов
В пределах каждого USB-накопителя они уникальны. Чтобы создать файл для пользователя, операционная система
ищет только UFD этого пользователя , чтобы выяснить, есть ли другой файл с таким же именем

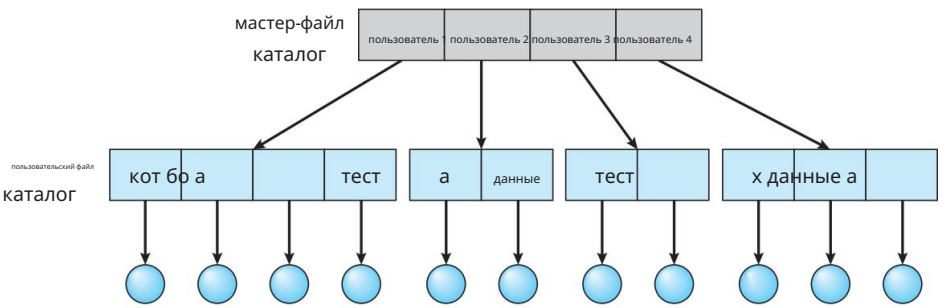


Рисунок 13.8 Двухуровневая структура каталога.

Существует. Чтобы удалить файл, операционная система ограничивает поиск локальным флэш-накопителем; таким образом, она не сможет случайно удалить файл другого пользователя с таким же именем.

Сами пользовательские каталоги должны создаваться и удаляться по мере необходимости. Специальная системная программа запускается с соответствующим именем пользователя и данными учётной записи. Программа создаёт новый USB-диск и добавляет запись о нём в MFD. Выполнение этой программы может быть ограничено системными администраторами. Выделение дискового пространства для пользовательских каталогов можно выполнить с помощью методов, описанных в главе 14 для самих файлов.

Хотя двухуровневая структура каталогов решает проблему конфликта имён, у неё всё ещё есть недостатки. Такая структура эффективно изолирует одного пользователя от другого. Изоляция является преимуществом, когда пользователи полностью независимы, но становится недостатком, когда пользователи хотят совместно работать над какой-либо задачей и получать доступ к файлам друг друга. Некоторые системы просто не позволяют другим пользователям получать доступ к локальным файлам пользователей.

Если доступ должен быть разрешён, один пользователь должен иметь возможность назвать файл в каталоге другого пользователя. Чтобы дать файлу уникальное имя в двухуровневом каталоге, необходимо указать как имя пользователя, так и имя файла. Двухуровневый каталог можно рассматривать как дерево или перевёрнутое дерево высоты 2. Корнем дерева является MFD . Его прямыми потомками являются UFD. Потомками UFD являются сами файлы. Файлы являются листьями дерева. Указание имени пользователя и имени файла определяет путь в дереве от корня (MFD) до листа (указанного файла). Таким образом, имя пользователя и имя файла определяют путь. Каждый файл в системе имеет путь. Чтобы дать файлу уникальное имя, пользователь должен знать путь к нужному файлу.

Например, если пользователь A хочет получить доступ к своему тестовому файлу test.txt, он может просто обратиться к test.txt. Однако для доступа к файлу test.txt пользователя B (с именем записи каталога userb) ему, возможно, придётся обратиться к /userb/test.txt. Каждая система имеет свой собственный синтаксис именования файлов в каталогах, отличных от каталога пользователя.

Для указания тома файла требуется дополнительный синтаксис. Например, в Windows том обозначается буквой, за которой следует двоеточие. Таким образом, спецификация файла может быть C: userb test. Некоторые системы идут ещё дальше и разделяют части спецификации, содержащие том, имя каталога и имя файла. В OpenVMS, например, файл login.com может быть указан как: u: [sst.crissmeyer]login.com;1, где u — имя тома, sst — имя каталога, crissmeyer — имя подкаталога, а 1 — номер версии. Другие системы, такие как UNIX и Linux, просто обрабатывают имя тома как часть имени каталога. Первое указанное имя — это имя тома, а остальные — имя каталога и файла. Например, /u/ pgalvin/test может указывать на том u, каталог pgalvin и файл test.

Особый случай такой ситуации возникает с системными файлами. Программы, предоставляемые как часть системы — загрузчики, ассемблеры, компиляторы, служебные процедуры, библиотеки и так далее — обычно определяются как файлы. Когда операционной системе даются соответствующие команды, эти файлы считываются загрузчиком и выполняются. Многие интерпретаторы команд просто воспринимают такую команду как имя файла для загрузки и выполнения. В системе каталогов, как мы определили выше, поиск этого имени файла будет производиться на текущем флэш-накопителе. Одним из решений было бы копирование системных файлов на каждый флэш-накопитель. Однако копирование всех системных файлов привело бы к потере огромного количества места. (Если системные файлы

требуется 5 МБ, то поддержка 12 пользователей потребует 5 × 12 = 60 МБ только для
копии системных файлов.)

Стандартное решение — немного усложнить процедуру поиска.
определен специальный пользовательский каталог, содержащий системные файлы (например, пользователь
0). Всякий раз, когда указывается имя файла для загрузки, операционная система сначала
Поиск на локальном USB-накопителе. Если файл найден, он используется. Если файл не найден, система
автоматически ищет специальный пользовательский каталог, содержащий системные файлы.
Последовательность каталогов, в которых выполняется поиск при указании имени файла, называется поиском.
Путь поиска можно расширить, включив в него неограниченный список каталогов.
для поиска по имени команды. Этот метод наиболее часто используется.
в UNIX и Windows. Системы также могут быть спроектированы таким образом, чтобы у каждого пользователя был свой
собственный путь поиска.

13.3.3 Древовидные каталоги

После того, как мы увидели, как просматривать двухуровневый каталог как двухуровневое дерево,
Естественным обобщением является расширение структуры каталога до дерева
Произвольная высота (рис. 13.9). Это обобщение позволяет пользователям создавать свои
собственные подкаталоги и соответствующим образом организовывать их файлы. Дерево – это самый
Общая структура каталогов. Дерево имеет корневой каталог, и каждый файл в нём
система имеет уникальное имя пути.

Каталог (или подкаталог) содержит набор файлов или подкаталогов. В
Во многих реализациях каталог — это просто еще один файл, но он рассматривается в
особым образом. Все каталоги имеют одинаковый внутренний формат. Один бит в каждом
Запись каталога определяет запись как файл (0) или как подкаталог (1). Специальный

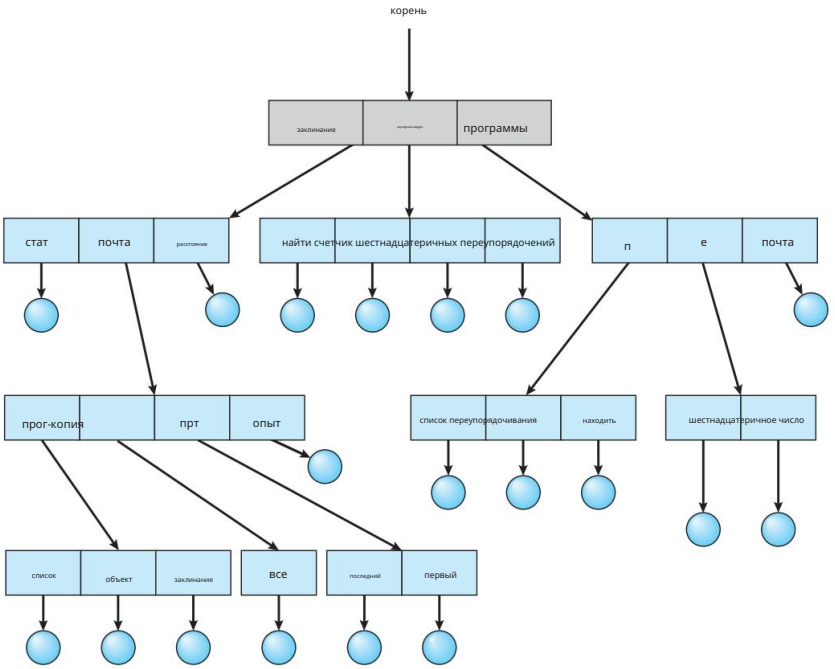


Рисунок 13.9 Древовидная структура каталогов.

Системные вызовы используются для создания и удаления каталогов. В этом случае операционная система (или код файловой системы) реализует другой формат файла — каталог.

При обычном использовании у каждого процесса есть текущий каталог. Текущий каталог должен содержать большинство файлов, представляющих интерес для процесса. При обращении к файлу поиск выполняется в текущем каталоге. Если требуется файл, которого нет в текущем каталоге, пользователь обычно должен либо указать путь к нему, либо изменить текущий каталог на каталог, содержащий этот файл.
Для смены каталогов можно использовать системный вызов, принимающий имя каталога в качестве параметра и использующий его для переопределения текущего каталога. Таким образом, пользователь может сменить текущий каталог в любое время. В других системах отслеживание текущего каталога и управление им возлагается на приложение (например, оболочку), поскольку у каждого процесса могут быть разные текущие каталоги.

Начальный текущий каталог оболочки входа пользователя назначается при запуске пользовательского задания или входе пользователя в систему. Операционная система ищет запись для этого пользователя в файле учёта (или в другом предопределённом месте) для учёта. В файле учёта содержится указатель на начальный каталог пользователя (или его имя). Этот указатель копируется в локальную переменную для этого пользователя, которая определяет начальный текущий каталог пользователя. Из этой оболочки могут быть запущены другие процессы. Текущим каталогом любого подпроцесса обычно является текущий каталог родительского процесса на момент его запуска.

Пути могут быть двух типов: абсолютные и относительные. В UNIX и Linux абсолютный путь начинается с корня (обозначается начальным символом «/») и продолжается до указанного файла, указывая имена каталогов на пути. Относительный путь определяет путь от текущего каталога. Например, в древовидной файловой системе, показанной на рисунке 13.9, если текущий каталог — /spell/mail, то относительный путь prt/first ссылается на тот же файл, что и абсолютный путь /spell/mail/prt/first.

Разрешение пользователю создавать собственные подкаталоги позволяет ему задавать определённую структуру для своих файлов. Эта структура может привести к созданию отдельных каталогов для файлов, связанных с разными темами (например, для хранения текста этой книги был создан подкаталог) или для различных форм информации. Например, каталог programs может содержать исходные коды программ; каталог bin может хранить все двоичные файлы. (Кстати, исполняемые файлы во многих системах назывались «двоичными», поэтому они хранились в каталоге bin .)

Интересное решение политики в древовидной структуре каталога касается того, как обрабатывать удаление каталога. Если каталог пуст, его запись в каталоге, содержащем его, можно просто удалить. Однако предположим, что удаляемый каталог не пуст, а содержит несколько файлов или подкаталогов. Можно выбрать один из двух подходов. Некоторые системы не удаляют каталог, если он не пуст. Таким образом, чтобы удалить каталог, пользователь должен сначала удалить все файлы в этом каталоге. Если существуют какие-либо подкаталоги, эта процедура должна быть применена к ним рекурсивно, чтобы их также можно было удалить. Такой подход может привести к значительному объему работы. Альтернативный подход, например, используемый командой UNIX rm , заключается в предоставлении опции: при запросе на удаление каталога все файлы и подкаталоги этого каталога также должны быть удалены. Любой из подходов довольно прост в реализации; выбор зависит от политики.

Последний вариант удобнее, но и опаснее, поскольку целую структуру каталогов можно удалить одной командой. Если эта команда

выдан по ошибке, потребуется восстановить большое количество файлов и каталогов (при условии существования резервной копии).

Благодаря древовидной структуре каталогов пользователям может быть предоставлен доступ не только к своим файлам, но и к файлам других пользователей. Например, пользователь B может получить доступ к файлу пользователя A, указав путь к нему. Пользователь B может указать как абсолютный, так и относительный путь. Кроме того, пользователь B может изменить свой текущий каталог на каталог пользователя A и получить доступ к файлу по его имени.

13.3.4 Каталоги ациклических графов

Рассмотрим двух программистов, работающих над совместным проектом. Файлы, связанные с этим проектом, можно хранить в подкаталоге, отделяя их от других проектов и файлов двух программистов. Но поскольку оба программиста в равной степени ответственны за проект, они оба хотят, чтобы подкаталог находился в их собственных каталогах. В этом случае общий подкаталог должен быть общим.

Общий каталог или файл существует в файловой системе в двух (или более) местах одновременно.
один раз.

Древовидная структура запрещает совместное использование файлов и каталогов. Ациклический граф , то есть граф без циклов, позволяет каталогам совместно использовать подкаталоги и файлы (рис. 13.10). Один и тот же файл или подкаталог может находиться в двух разных каталогах. Ациклический граф является естественным обобщением древовидной структуры каталогов.

Важно отметить, что общий файл (или каталог) — это не то же самое, что две копии файла. При наличии двух копий каждый программист может видеть копию, а не оригинал, но если один программист изменит файл, изменения не отобразятся в копии другого. При общем файле существует только один файл, поэтому любые изменения, внесённые одним пользователем, сразу видны другому. Общий доступ — это

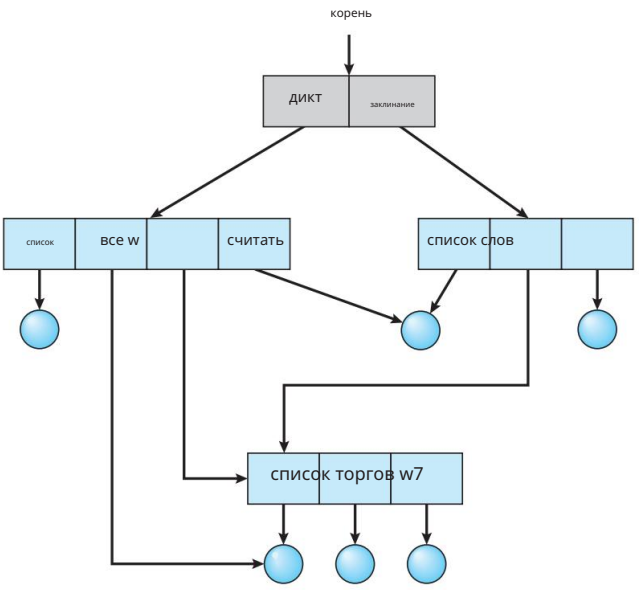


Рисунок 13.10 Структура каталога с ациклическим графом.

особенно важно для подкаталогов: новый файл, созданный одним человеком, автоматически появится во всех общих подкаталогах.

Когда люди работают в команде, все файлы, которыми они хотят поделиться, можно поместить в один каталог. Домашний каталог каждого члена команды может содержать этот каталог с общими файлами в качестве подкаталога. Даже в случае работы одного пользователя организация файлов пользователя может потребовать размещения некоторых файлов в разных подкаталогах. Например, программа, написанная для конкретного проекта, должна находиться как в каталоге всех программ, так и в каталоге этого проекта.

Общие файлы и подкаталоги можно реализовать несколькими способами. Распространенный способ, представленный в системах UNIX , — это создание новой записи каталога, называемой ссылкой. Ссылка фактически представляет собой указатель на другой файл или подкаталог. Например, ссылка может быть реализована как абсолютный или относительный путь. При указании ссылки на файл мы выполняем поиск в каталоге. Если запись каталога отмечена как ссылка, то имя реального файла включается в информацию о ссылке. Мы разрешаем ссылку, используя этот путь для поиска реального файла. Ссылки легко идентифицируются по формату в записи каталога (или по наличию специального типа в системах, поддерживающих типы) и фактически являются косвенными указателями. Операционная система игнорирует эти ссылки при обходе деревьев каталогов, чтобы сохранить ациклическую структуру системы.

Другой распространённый подход к реализации общих файлов — простое дублирование всей информации о них в обоих каталогах общего доступа. Таким образом, обе записи идентичны и равноправны. Рассмотрим разницу между этим подходом и созданием ссылки. Ссылка явно отличается от исходной записи каталога, поэтому они не равны. Однако дублирование записей каталога делает оригинал и копию неразличимыми. Основная проблема дублирования записей каталога — сохранение согласованности при изменении файла.

Структура каталогов на основе ациклического графа более гибкая, чем простая древовидная структура, но и более сложная. Необходимо тщательно продумать несколько проблем. Файл может иметь несколько имён абсолютных путей. Следовательно, разные имена файлов могут ссылаться на один и тот же файл. Эта ситуация аналогична проблеме с псевдонимами в языках программирования. Если мы пытаемся обойти всю файловую систему — найти файл, собрать статистику по всем файлам или скопировать все файлы в резервное хранилище — эта проблема становится существенной, поскольку мы не хотим обходить общие структуры более одного раза.

Другая проблема связана с удалением. Когда можно освободить и повторно использовать пространство, выделенное для общего файла? Один из вариантов — удалять файл каждый раз, когда кто-либо его удаляет, но это действие может оставить висячие указатели на уже несуществующий файл. Хуже того, если оставшиеся указатели файлов содержат реальные дисковые адреса, а пространство впоследствии используется другими файлами, эти висячие указатели могут указывать на середину других файлов.

В системе, где общий доступ реализован с помощью символических ссылок, такая ситуация несколько проще. Удаление ссылки не обязательно затрагивает исходный файл; удаляется только сама ссылка. Если удаляется сама запись файла, место для файла освобождается, оставляя ссылки висячими. Мы можем искать эти ссылки и удалять их, но если список связанных ссылок не хранится для каждого файла, такой поиск может быть затратным. В качестве альтернативы, мы можем оставить ссылки до тех пор, пока не будет предпринята попытка их использования. В этот момент мы можем определить, что файл с именем, указанным в ссылке, не существует, и может произойти сбой в разрешении имени ссылки; обращение обрабатывается так же, как и для любого другого недопустимого имени файла. (В этом случае разработчик системы должен тщательно продумать, что делать, если файл...

(удалён и создан другой файл с тем же именем, прежде чем будет использована символическая ссылка на исходный файл.) В случае UNIX при удалении файла остаются символические ссылки, и пользователю остаётся лишь определить, что исходный файл удалён или заменён. Microsoft Windows использует тот же подход.

Другой подход к удалению — сохранение файла до тех пор, пока все ссылки на него не будут удалены. Для реализации этого подхода необходим механизм определения того, что последняя ссылка на файл была удалена. Мы могли бы вести список всех ссылок на файл (записей каталога или символических ссылок). При создании ссылки или копии записи каталога в список ссылок на файл добавляется новая запись. При удалении ссылки или записи каталога её запись удаляется из списка. Файл удаляется, когда список ссылок на файл становится пустым.

Проблема этого подхода заключается в изменчивом и потенциально большом размере списка ссылок на файлы. Однако нам на самом деле не нужно хранить весь список — достаточно лишь подсчёта количества ссылок. Добавление новой ссылки или записи в каталог увеличивает счётчик ссылок. Удаление ссылки или записи уменьшает счётчик. Когда счётчик равен 0, файл можно удалить; ссылок на него больше не остаётся. Операционная система UNIX использует этот подход для несимволических ссылок (или жёстких ссылок), сохраняя счётчик ссылок в блоке информации о файле (или inode; см. раздел C.7.2). Фактически запрещая множественные ссылки на каталоги, мы поддерживаем структуру ациклического графа.

Чтобы избежать проблем, подобных тем, что только что обсуждались, некоторые системы просто не разрешайте общие каталоги или ссылки.

13.3.5 Общий каталог графиков

Серьёзной проблемой при использовании ациклической графовой структуры является обеспечение отсутствия циклов. Если начать с двухуровневого каталога и разрешить пользователям создавать подкаталоги, получится древовидная структура. Должно быть довольно легко увидеть, что простое добавление новых файлов и подкаталогов в существующий древовидный каталог сохраняет его структуру. Однако при добавлении ссылок древовидная структура разрушается, и в результате получается простая графовая структура (рис. 13.11).

Основным преимуществом ациклического графа является относительная простота алгоритмов его обхода и определения отсутствия ссылок на файл. Мы хотим избежать двойного обхода общих участков ациклического графа, главным образом из соображений производительности. Если мы только что выполнили поиск файла в основном общем подкаталоге, но не нашли его, нам нужно избежать повторного поиска в этом подкаталоге; повторный поиск будет пустой тратой времени.

Если в каталоге допускаются циклы, мы также хотим избежать повторного поиска любого компонента как из соображений корректности, так и производительности. Неудачно спроектированный алгоритм может привести к бесконечному циклу, который будет постоянно проходить по всему циклу и никогда не завершится. Одним из решений является произвольное ограничение количества каталогов, к которым будет осуществляться доступ во время поиска.

Аналогичная проблема возникает, когда мы пытаемся определить, когда файл можно удалить. В структурах каталогов с ациклическим графом значение 0 в счётчике ссылок означает, что ссылок на файл или каталог больше нет, и файл можно удалить. Однако при наличии циклов счётчик ссылок может быть отличен от 0, даже если ссылка на каталог или файл больше невозможна.

Эта аномалия возникает из-за возможности самоссылки (или цикличности) в структуре каталогов. В этом случае обычно требуется сборка мусора .

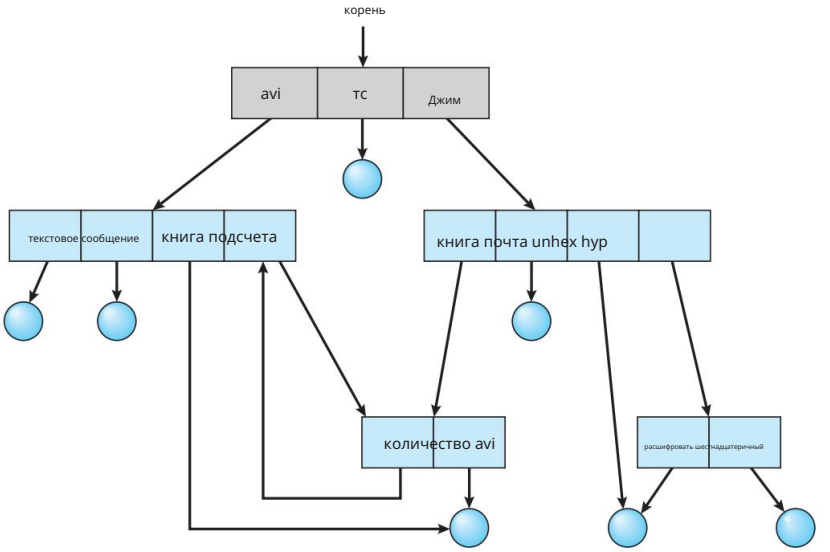


Рисунок 13.11 Общий каталог графов.

Схема определения момента удаления последней ссылки и возможности перераспределения дискового пространства. Сборка мусора включает в себя обход всей файловой системы и маркировку всего, к чему возможен доступ. Затем, во втором проходе, всё, что не отмечено, собирается в список свободного пространства. (Аналогичная процедура маркировки может использоваться для обеспечения того, чтобы обход или поиск охватил всё в файловой системе один и только один раз.) Однако сборка мусора для дисковой файловой системы чрезвычайно трудоёмка и поэтому применяется редко.

Сборка мусора необходима только из-за возможных циклов в графе.
Таким образом, работать со структурой ациклического графа гораздо проще. Сложность заключается в том, чтобы избежать циклов при добавлении новых ссылок в структуру. Как узнать, когда новая ссылка завершит цикл? Существуют алгоритмы для обнаружения циклов в графах, однако они требуют больших вычислительных затрат, особенно если граф хранится на диске. Более простой алгоритм в частном случае каталогов и ссылок заключается в обходе ссылок при обходе каталогов. Это позволяет избежать циклов и не влечет за собой дополнительных накладных расходов.

13.4 Защита

Когда информация хранится в компьютерной системе, мы хотим защитить ее от физических повреждений (вопрос надежности) и неправомерного доступа (вопрос защиты).

Надёжность обычно обеспечивается дубликатами файлов. Многие компьютеры оснащены системными программами, которые автоматически (или при участии оператора) регулярно (раз в день, неделю или месяц) копируют файлы с диска на ленту, чтобы сохранить копию на случай случайного повреждения файловой системы.
Файловые системы могут быть повреждены из-за проблем с оборудованием (например, ошибок чтения или записи), скачков напряжения или сбоев, поломок головок, грязи, экстремальных температур,

и вандализм. Файлы могут быть случайно удалены. Ошибки в программном обеспечении файловой системы также могут привести к потере содержимого файлов. Вопросы надёжности более подробно рассматривались в главе 11.

Защита может быть обеспечена различными способами. Для ноутбука с современной операционной системой мы можем обеспечить защиту, требуя аутентификации по имени пользователя и паролю для доступа к нему, шифруя дополнительное хранилище, чтобы даже открыв ноутбук и извлекши диск, было сложно получить доступ к его данным, и защищая сетевой доступ с помощью межсетевого экрана, чтобы во время использования компьютера было сложно взломать его через сетевое подключение. В многопользовательской системе даже для допустимого доступа к системе требуются более продвинутые механизмы, позволяющие только допустимому доступу к данным.

13.4.1 Типы доступа

Необходимость защиты файлов напрямую связана с возможностью доступа к ним. Системы, не разрешающие доступ к файлам других пользователей, не нуждаются в защите. Таким образом, мы могли бы обеспечить полную защиту, запретив доступ. В качестве альтернативы, мы могли бы предоставить свободный доступ без какой-либо защиты. Оба подхода слишком радикальны для широкого использования. Необходим контролируемый доступ.

Механизмы защиты обеспечивают контролируемый доступ, ограничивая типы доступа к файлам. Доступ разрешается или запрещается в зависимости от нескольких факторов, одним из которых является тип запрашиваемого доступа. Можно контролировать несколько различных типов операций:

- Читать. Читать из файла.
- Запись. Запись или перезапись файла.
- Выполнить. Загрузить файл в память и выполнить его. • Добавить. Записать новую информацию в конец файла. • Удалить. Удалить файл и освободить место для возможного повторного использования.
- Список. Перечислите имя и атрибуты файла.
- Изменение атрибутов. Изменение атрибутов файла.

Другие операции, такие как переименование, копирование и редактирование файла, также могут контролироваться. Однако во многих системах эти высокоуровневые функции могут быть реализованы системной программой, которая выполняет низкоуровневые системные вызовы. Защита обеспечивается только на нижнем уровне. Например, копирование файла может быть реализовано просто последовательностью запросов на чтение. В этом случае пользователь с доступом на чтение может также инициировать копирование, печать и т. д. файла.

Было предложено множество механизмов защиты. Каждый из них имеет свои преимущества и недостатки и должен соответствовать своему назначению. Например, небольшая компьютерная система, используемая лишь несколькими членами исследовательской группы, может не нуждаться в тех же типах защиты, что и большой корпоративный компьютер, используемый для исследований, финансов и управления персоналом. Мы рассмотрим некоторые подходы к защите в следующих разделах и представим более полное описание в главе 17.

13.4.2 Контроль доступа

Наиболее распространённый подход к решению проблемы защиты — сделать доступ зависящим от личности пользователя. Разным пользователям могут потребоваться разные типы доступа к файлу или каталогу. Наиболее общая схема реализации доступа, зависящего от личности, заключается в связывании с каждым файлом и каталогом списка управления доступом (ACL), в котором указаны имена пользователей и типы доступа, разрешённые для каждого пользователя.
Когда пользователь запрашивает доступ к определённому файлу, операционная система проверяет список доступа, связанный с этим файлом. Если этот пользователь указан в списке для запрашиваемого доступа, доступ предоставляется. В противном случае происходит нарушение защиты, и заданию пользователя отказывается в доступе к файлу.

Преимущество такого подхода заключается в возможности реализации сложных методов доступа. Основная проблема списков доступа — их длина. Если мы хотим разрешить всем читать файл, нам необходимо перечислить всех пользователей с правом на чтение. Этот подход имеет два нежелательных последствия:

- Составление такого списка может оказаться утомительной и неблагодарной задачей, особенно если мы заранее не знаем список пользователей в системе.
- Запись каталога, ранее имевшая фиксированный размер, теперь должна иметь переменный размер, что приводит к более сложному управлению пространством.

Эти проблемы можно решить, используя сокращенную версию списка доступа.

Чтобы сократить длину списка контроля доступа, многие системы распознают три классификации пользователей в связи с каждым файлом:

- Владелец. Владельцем является пользователь, создавший файл.
- Группа. Группа пользователей, совместно использующих файл и нуждающихся в одинаковом доступе, называется рабочей группой.
- Другие. Все остальные пользователи в системе.

Наиболее распространённый в настоящее время подход заключается в сочетании списков контроля доступа с более общей (и более простой в реализации) схемой контроля доступа на основе владельца, группы и вселенной, описанной выше. Например, Solaris использует три категории доступа по умолчанию, но позволяет добавлять списки контроля доступа к определённым файлам и каталогам, когда требуется более детальный контроль доступа.

Для иллюстрации рассмотрим Сару, которая пишет новую книгу. Она наняла трёх аспирантов (Джима, Дон и Джилл) для помощи в работе над проектом. Текст книги хранится в файле с именем book.tex. Защита этого файла следующая:

- Сара должна иметь возможность выполнять все операции с файлом.
- Джим, Дон и Джилл должны иметь возможность только читать и записывать файл; они не должно быть разрешено удалять файл.
- Все остальные пользователи должны иметь возможность читать файл, но не писать в нем. (Сара хочет, чтобы как можно больше людей прочитали текст, чтобы она могла получить обратную связь.)

РАЗРЕШЕНИЯ В СИСТЕМЕ UNIX

В системе UNIX защита каталогов и файлов реализована аналогично. С каждым файлом и каталогом связаны три поля: владелец, группа и юниверс, каждое из которых состоит из трёх битов: rwx, где r управляет доступом на чтение, w — доступом на запись, а x — выполнением. Таким образом, пользователь может просмотреть содержимое подкаталога, только если в соответствующем поле установлен бит r . Аналогично пользователь может изменить свой текущий каталог на другой текущий каталог (например, foo), только если в соответствующем поле установлен бит x , связанный с подкаталогом foo .

Пример листинга каталогов из среды UNIX показан ниже:

-рв-рв-р--	1 pbg	сотрудники	31200	3 сен 08:30 8	intro.ps
drwx------	5 pbg	сотрудники	512	июл 09.33 8	private/
drwxrwxr-x	2 pbg	сотрудники	512	июл 09:35 3	doc/
drwxrwx---	2 jwg	студенты	512	авг 14:13 24	student-proj/
-рв-р--р--	1 pbg	сотрудники	9423	фев 2017 24	program.c
-rwxr-xr-x	1 pbg	сотрудники	20471	фев 2017 31	program
drwx--x--x	4 tag	преподаватели	512	июл 10:31 29	lib/
drwx------	3 pbg	сотрудники	1024	авг 06:52 8 июл	mail/
drwxrwxrwx	3 pbg	сотрудники	512	09:35	test/

Первое поле описывает уровень защиты файла или каталога. Первый символ «d» обозначает подкаталог. Также отображаются количество ссылок на файл, имя владельца, имя группы, размер файла в байтах, дата последнего изменения и, наконец, имя файла (с необязательным расширением).

Для обеспечения такой защиты необходимо создать новую группу, например, text , с участниками Джим, Дон и Джилл. Имя группы text должно быть связано с файлом book.tex, а права доступа должны быть установлены в соответствии с описанной нами политикой.

Теперь представьте посетителя, которому Сара хочет предоставить временный доступ к Главе 1. Посетителя нельзя добавить в текстовую группу, поскольку это дало бы ему доступ ко всем главам. Поскольку файл может находиться только в одной группе, Сара не может добавить другую группу к Главе 1. Однако с добавлением функциональности списка контроля доступа посетителя можно добавить в список контроля доступа к Главе 1.

Для корректной работы этой схемы необходимо строго контролировать разрешения и списки доступа. Этот контроль может быть реализован несколькими способами. Например, в системе UNIX группы может создавать и изменять только руководитель объекта (или любой суперпользователь). Таким образом, управление осуществляется посредством взаимодействия с пользователем. Списки доступа подробно рассматриваются в разделе 17.6.2.

При более ограниченной классификации защиты для определения защиты достаточно всего трёх полей. Часто каждое поле представляет собой набор битов, каждый из которых либо разрешает, либо запрещает связанный с ним доступ. Например, в системе UNIX определены три поля по три бита каждое: rwx, где r управляет доступом на чтение, w — доступом на запись, а x — выполнением. Отдельное поле предназначено для

Для владельца файла, для группы файла и для всех остальных пользователей. В этой схеме для записи информации о защите требуется девять битов на файл. Таким образом, в нашем примере поля защиты для файла book.tex следующие: для владельца Sara установлены все биты; для группы text установлены биты r и w ; а для universe установлен только бит r .

Одна из сложностей при комбинировании подходов возникает в пользовательском интерфейсе. Пользователи должны иметь возможность определить, когда для файла установлены необязательные разрешения ACL . В примере Solaris к обычным разрешениям добавляется символ «+», например:

19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1

Для управления используется отдельный набор команд setfacl и getfacl.

Списки контроля доступа.

Пользователи Windows обычно управляют списками контроля доступа через графический интерфейс. На рисунке 13.12 показано окно управления правами доступа к файлу в файловой системе NTFS в Windows 7. В этом примере пользователю «гость» специально запрещён доступ к файлу ListPanel.java.

Другая сложность — назначение приоритета при конфликте разрешений и списков контроля доступа (ACL) . Например, если Уолтер входит в группу файла, которая имеет разрешение на чтение, но у файла есть ACL, предоставляющий Уолтеру разрешение на чтение и запись, следует ли разрешить или запретить запись Уолтеру? Solaris и другие операционные системы отдают приоритет спискам контроля доступа (ACL) (поскольку они более детальны и не назначаются по умолчанию). Это следует из общего правила, согласно которому приоритет должен быть отдан специфичности.

13.4.3 Другие подходы к защите

Другой подход к решению проблемы защиты — привязка пароля к каждому файлу. Подобно тому, как доступ к компьютерной системе часто контролируется паролем, доступ к каждому файлу можно контролировать таким же образом. Если пароли выбираются случайным образом и часто меняются, такая схема может быть эффективна для ограничения доступа к файлу. Однако использование паролей имеет ряд недостатков.

Во-первых, количество паролей, которые пользователю необходимо запомнить, может стать слишком большим, что сделает схему непрактичной. Во-вторых, если для всех файлов используется один пароль, то после его обнаружения все файлы становятся доступны; защита действует по принципу «всё или ничего». Некоторые системы позволяют пользователю связать пароль с подкаталогом, а не с отдельным файлом, чтобы решить эту проблему.

Чаще всего надежную защиту обеспечивает шифрование раздела или отдельных файлов, но ключевым фактором является управление паролями.

В многоуровневой структуре каталогов необходимо защищать не только отдельные файлы, но и наборы файлов в подкаталогах, то есть необходимо предусмотреть механизм защиты каталогов. Операции с каталогами, которые необходимо защитить, несколько отличаются от операций с файлами. Мы хотим контролировать создание и удаление файлов в каталоге. Кроме того, мы, вероятно, хотим контролировать, может ли пользователь определить наличие файла в каталоге.

Иногда знание о существовании и имени файла само по себе важно. Поэтому вывод содержимого каталога должен быть защищённой операцией.

Аналогично, если имя пути указывает на файл в каталоге, пользователю должен быть разрешён доступ как к каталогу, так и к файлу. В системах, где файлы могут иметь несколько имён путей (например, ациклические и общие графы), у пользователя могут быть разные права доступа к конкретному файлу в зависимости от используемого имени пути.

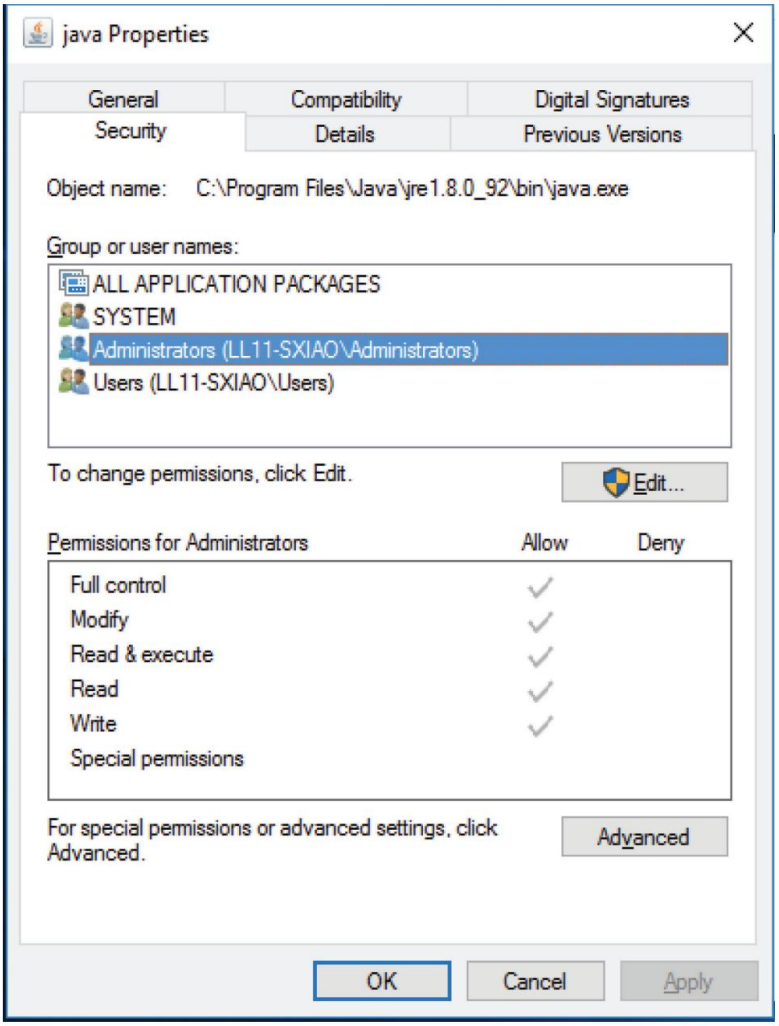


Рисунок 13.12 Управление списками контроля доступа Windows 10.

13.5 Файлы, отображенные в памяти

Есть еще один метод доступа к файлам, и он используется очень часто.
Рассмотрим последовательное чтение файла с диска с использованием стандартных системных вызовов open(), read() и write(). Каждое обращение к файлу требует системного вызова и обращения к диску. В качестве альтернативы можно использовать методы виртуальной памяти, описанные в главе 10, чтобы рассматривать файловый ввод-вывод как обычные операции доступа к памяти. Этот подход, известный как отображение файла в память, позволяет логически связать часть виртуального адресного пространства с файлом. Как мы увидим, это может привести к значительному повышению производительности.

13.5.1 Базовый механизм

Отображение файла в памяти осуществляется путём сопоставления блока диска со страницей (или страницами) памяти. Первоначальный доступ к файлу осуществляется по обычному запросу.

Подкачка страниц, приводящая к ошибке страницы. Однако при этом часть файла размером со страницу считывается из файловой системы в физическую страницу (некоторые системы могут одновременно считывать фрагменты памяти размером с страницу). Последующие операции чтения и записи в файл обрабатываются как обычные операции доступа к памяти. Манипулирование файлами через память, а не использование системных вызовов read() и write(), упрощает и ускоряет доступ к файлам и их использование.

Обратите внимание, что запись в файл, отображаемый в памяти, не обязательно является немедленной (синхронной) записью в файл на вторичном хранилище. Как правило, системы обновляют файл на основе изменений в образе памяти только при его закрытии. В условиях нехватки памяти системы вынуждены перемещать любые промежуточные изменения в пространство подкачки, чтобы не потерять их при освобождении памяти для других целей. При закрытии файла все данные, отображаемые в памяти, записываются обратно в файл на вторичном хранилище и удаляются из виртуальной памяти процесса.

Некоторые операционные системы обеспечивают отображение памяти только через определенный системный вызов и используют стандартные системные вызовы для выполнения всех остальных файловых операций ввода-вывода. Однако некоторые системы отображают файл в память независимо от того, был ли он указан как отображаемый в память. Возьмём в качестве примера Solaris. Если файл указан как отображаемый в память (с помощью системного вызова mmap()), Solaris отображает его в адресное пространство процесса. Если файл открывается и к нему осуществляется доступ с помощью обычных системных вызовов, таких как open(), read() и write(), Solaris всё равно отображает файл в память, однако файл отображается в адресное пространство ядра.
Независимо от того, как открыт файл, Solaris рассматривает все операции ввода-вывода файлов как отображенные в память, что позволяет осуществлять доступ к файлам через эффективную подсистему памяти и избегать накладных расходов на системные вызовы, вызванных каждым традиционным чтением (read()) и записью (write()).

Несколько процессов могут одновременно отображать один и тот же файл, обеспечивая совместное использование данных. Запись, выполняемая любым из процессов, изменяет данные в виртуальной памяти и видна всем остальным процессам, отображающим тот же раздел файла.
Учитывая наше предыдущее обсуждение виртуальной памяти, должно быть понятно, как реализуется совместное использование отображённых в память участков памяти: карта виртуальной памяти каждого процесса, использующего совместное использование, указывает на одну и ту же страницу физической памяти — страницу, содержащую копию дискового блока. Такое совместное использование памяти показано на рисунке 13.13. Системные вызовы, обеспечивающие отображение памяти, также могут поддерживать функцию копирования при записи, позволяя процессам совместно использовать файл в режиме только для чтения, но при этом иметь собственные копии любых изменяемых ими данных. Для координации доступа к общим данным участвующие процессы могут использовать один из механизмов взаимного исключения, описанных в главе 6.

Довольно часто общая память фактически реализуется с помощью файлов отображения памяти. В этом случае процессы могут взаимодействовать, используя общую память, отображая один и тот же файл в свои виртуальные адресные пространства. Отображённый в память файл служит областью общей памяти между взаимодействующими процессами (рис. 13.14). Мы уже видели это в разделе 3.5, где создаётся объект общей памяти POSIX , и каждый взаимодействующий процесс отображает этот объект в своё адресное пространство. В следующем разделе мы обсудим поддержку общей памяти с помощью отображённых в память файлов в Windows API .

13.5.2 Общая память в Windows API

Общая схема создания области общей памяти с использованием отображенных в память файлов в API Windows включает в себя сначала создание отображения файла для

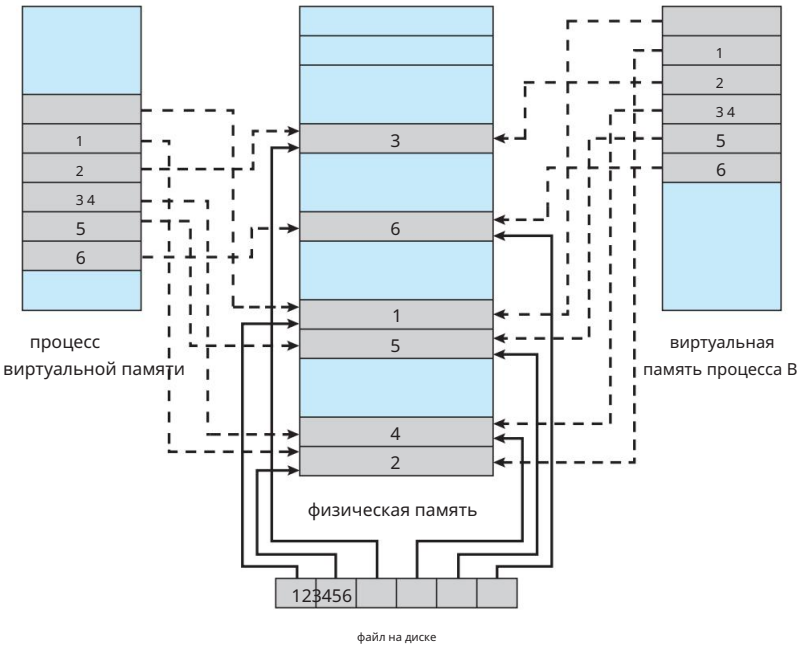


Рисунок 13.13 Файлы, отображенные в памяти.

Файл, который необходимо отобразить, и затем создать представление отображённого файла в виртуальном адресном пространстве процесса. Затем второй процесс может открыть и создать представление отображённого файла в своём виртуальном адресном пространстве. Отображённый файл представляет собой объект разделяемой памяти, который обеспечит взаимодействие между процессами.

Далее мы проиллюстрируем эти шаги более подробно. В этом примере процесс-производитель сначала создаёт объект общей памяти, используя функции отображения памяти, доступные в Windows API. Затем процесс-производитель записывает сообщение в общую память. После этого процесс-потребитель открывает отображение объекта общей памяти и считывает сообщение, записанное потребителем.

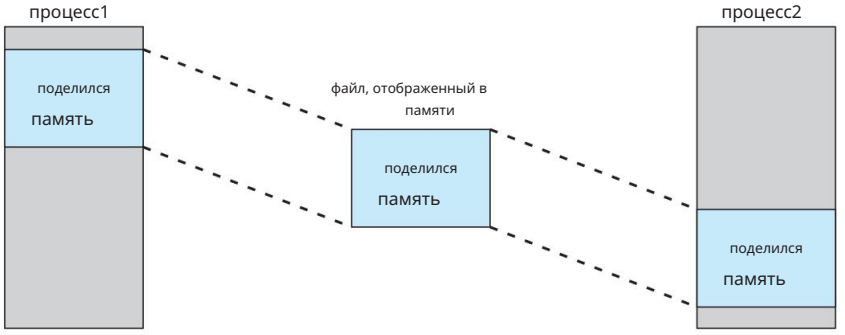


Рисунок 13.14 Общая память с использованием ввода-вывода, отображенного в памяти.

Чтобы создать файл, отображаемый в память, процесс сначала открывает файл, который необходимо отобразить, с помощью функции CreateFile() , которая возвращает дескриптор открытого файла. Затем процесс создаёт отображение этого дескриптора файла с помощью функции CreateFileMapping() . После завершения отображения файла процесс устанавливает представление отображаемого файла в своём виртуальном адресном пространстве с помощью функции MapViewOfFile() . Представление отображаемого файла представляет собой часть отображаемого файла в виртуальном адресном пространстве процесса — может быть отображен весь файл или только его часть. Эта последова

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    HANDLE hFile, hMapFile;
    LPVOID lpMapAddress;

    hFile = CreateFile("temp.txt", /* имя файла */
        GENERIC_READ | GENERIC_WRITE, /* доступ на чтение/запись */ 0, /* нет
        общего доступа к файлу */
        NULL, /* безопасность по умолчанию */
        ОТКРЫТЬ ВСЕГДА, /* открыть новый или существующий файл */
        АТРИБУТ ФАЙЛА ОБЫЧНЫЙ, /* обычные атрибуты файла */
        NULL); /* нет шаблона файла */

    hMapFile = CreateFileMapping(hFile, /* дескриптор файла */
        NULL, /* безопасность по умолчанию */
        PAGE_READWRITE, /* доступ на чтение/запись к отображенным страницам */
        0, /* отображение всего файла */
        0,
        TEXT("SharedObject")); /* именованный объект общей памяти */

    lpMapAddress = MapViewOfFile(hMapFile, /* дескриптор сопоставленного объекта */
        FILE_MAP_ALL_ACCESS, /* доступ на чтение/запись */ 0, /*
        отображенный вид всего файла */ 0, 0);



    /* запись в общую память */
    sprintf(lpMapAddress,"Shared memory message");

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hFile);
    CloseHandle(hMapFile);
}
```

Рисунок 13.15. Производитель записывает данные в общую память с использованием API Windows.

показано на рисунке 13.15. (Мы исключаем большую часть проверки ошибок для краткости кода.)

Вызов CreateFileMapping() создаёт именованный объект общей памяти с именем SharedObject. Процесс-потребитель будет взаимодействовать, используя этот сегмент общей памяти, создавая сопоставление с тем же именованным объектом.
Затем производитель создаёт представление отображённого в память файла в своём виртуальном адресном пространстве. Передавая последним трём параметрам значение 0, он указывает, что отображённое представление представляет собой весь файл. Вместо этого можно было бы передать значения, указывающие смещение и размер, создав таким образом представление, содержащее только часть файла. (Важно отметить, что при установке отображения всё отображение может быть загружено в память не полностью. Вместо этого отображённый файл может быть подкачиваемым по требованию, то есть страницы будут попадать в память только по мере доступа
Функция MapViewOfFile() возвращает указатель на объект общей памяти; таким образом, любой доступ к этой области памяти является доступом к файлу, отображённому в память. В этом случае процесс-производитель записывает сообщение «Сообщение общей памяти» в общую память.

Программа, иллюстрирующая, как процесс-потребитель устанавливает представление именованного объекта общей памяти, показана на рисунке 13.16. Эта программа

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    ОБРАБОТАТЬ hMapFile;
    LPVOID lpMapAddress;

    hMapFile = OpenFileMapping(FILE MAP ALL ACCESS, /* Доступ для чтения и записи */
        ЛОЖЬ, /* нет наследования */
        TEXT("SharedObject")); /* имя сопоставленного файлового объекта */

    lpMapAddress = MapViewOfFile(hMapFile, /* дескриптор сопоставленного объекта */
        FILE MAP ALL ACCESS, /* доступ на чтение/запись */ 0, /*
        отображенный вид всего файла */ 0, 0);



    /* чтение из общей памяти */
    printf("Прочитано сообщение %s", lpMapAddress);

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hMapFile);
}
```

Рисунок 13.16 Потребитель считывает данные из общей памяти с использованием API Windows.

Несколько проще, чем показано на рисунке 13.15, поскольку процессу необходимо лишь создать сопоставление с существующим именованным объектом общей памяти. Процесс-потребитель также должен создать представление сопоставленного файла, как это сделал процесс-производитель в программе на рисунке 13.15. Затем потребитель считывает из общей памяти сообщение «Сообщение общей памяти» , записанное процессом-производителем.

Наконец, оба процесса удаляют представление отображенного файла с помощью вызова UnmapViewOfFile(). В конце главы мы приводим упражнение по программированию, использующее общую память с отображением памяти в Windows API.

13.6 Резюме

- Файл — это абстрактный тип данных, определяемый и реализуемый операционной системой. Он представляет собой последовательность логических записей. Логическая запись может представлять собой байт, строку (фиксированной или переменной длины) или более сложный элемент данных. Операционная система может поддерживать различные типы записей отдельно или возложить эту поддержку на прикладную программу.
- Одной из основных задач операционной системы является отображение концепции логического файла на физические устройства хранения, такие как жёсткий диск или энергонезависимая память (NVM) . Поскольку размер физической записи на устройстве может не совпадать с размером логической записи, может потребоваться упорядочить логические записи в физические. Опять же, эта задача может быть реализована операционной системой или возложена на прикладную программу. •

В файловой системе полезно создавать каталоги для организации файлов. Одноуровневый каталог в многопользовательской системе вызывает проблемы с именами, поскольку каждый файл должен иметь уникальное имя. Двухуровневый каталог решает эту проблему, создавая отдельный каталог для файлов каждого пользователя.
В каталоге файлы перечислены по именам и содержат информацию об их расположении на диске, размере, типе, владельце, времени создания, времени последнего использования

и т. д. • Естественным обобщением двухуровневого каталога является древовидная структура. Древовидная структура каталога позволяет пользователю создавать подкаталоги для организации файлов. Ациклические графовые структуры каталогов позволяют пользователям совместно использовать подкаталоги и файлы, но усложняют поиск и удаление. Общая графовая структура обеспечивает полную гибкость в совместном использовании файлов и каталогов, но иногда требует сборки мусора для восстановления неиспользуемого дискового пространства.

- Удалённые файловые системы создают проблемы с надёжностью, производительностью и безопасностью. Распределённые информационные системы хранят информацию о пользователях, хостах и доступе, чтобы клиенты и серверы могли обмениваться информацией о состоянии для управления использованием и доступом.
- Поскольку файлы являются основным хранилищем информации в большинстве компьютерных систем, в многопользовательских системах необходима защита файлов. Доступ к файлам можно контролировать отдельно для каждого типа доступа: чтение, запись, выполнение, добавление, удаление, просмотр каталога и т. д. Защита файлов может быть обеспечена списками доступа, паролями и другими методами.

Практические упражнения

13.1 Некоторые системы автоматически удаляют все пользовательские файлы при выходе пользователя из системы или завершении задания, если только пользователь явно не запросил их сохранение. Другие системы сохраняют все файлы, пока пользователь явно не удалит их. Обсудите относительные достоинства каждого подхода.

13.2 Почему одни системы отслеживают тип файла, другие оставляют это на усмотрение пользователя, а третьи вообще не поддерживают поддержку нескольких типов файлов? Какая система «лучше»?

13.3 Аналогично, некоторые системы поддерживают множество типов структур данных файла, в то время как другие поддерживают только поток байтов. Каковы преимущества и недостатки каждого подхода?

13.4 Можете ли вы смоделировать многоуровневую структуру каталогов с одноуровневой структурой каталогов, в которой можно использовать имена произвольной длины? Если да, объясните, как это сделать, и сравните эту схему с многоуровневой схемой каталогов. Если нет, объясните, что мешает успешной симуляции. Как бы изменился ваш ответ, если бы имена файлов были ограничены семью символами?

13.5 Объясните назначение операций open() и close() .

13.6 В некоторых системах подкаталог может быть прочитан и записан пользователем. измененный пользователь, как и обычные файлы.

a. Опишите проблемы защиты, которые могут возникнуть.

б) Предложить схему решения каждой из этих проблем защиты. лемы.

13.7. Рассмотрим систему, поддерживающую 5000 пользователей. Предположим, вы хотите разрешить 4990 из них доступ к одному файлу.

a. Как бы вы определили эту схему защиты в UNIX?

б. Можете ли вы предложить другую схему защиты, которая может быть использована для этой цели более эффективно, чем схема, предоставляемая UNIX?

13.8 Исследователи предложили вместо списка контроля доступа, связанного с каждым файлом (определяющего, какие пользователи могут получить доступ к файлу и как), использовать список контроля пользователей, связанный с каждым пользователем (определяющий, к каким файлам пользователь может получить доступ и как). Обсудите сравнительные преимущества этих двух схем.

Дополнительное чтение

Многоуровневая структура каталогов была впервые реализована в системе MULTICS ([Organick (1972)]). В настоящее время большинство операционных систем реализуют многоуровневые структуры каталогов. К ним относятся Linux ([Love (2010)]), macOS ([Singh (2007)]), Solaris ([McDougall and Mauro (2007)]) и все версии Windows ([Russinovich et al. (2017)]).

Общее обсуждение файловых систем Solaris можно найти в руководстве Sun System Administration Guide: Devices and File Systems (http://docs.sun.com/app/). docs/doc/817-5093).

Сетевая файловая система (NFS), разработанная Sun Microsystems, позволяет Структуры каталогов, распространяемые по сетевым компьютерным системам. NFS Версия 4 описана в RFC3505 (http://www.ietf.org/rfc/rfc3530.txt).

Отличный источник значений компьютерного жаргона — http://www.catb.org/ esr/жаргон/.

Библиография

[Любовь (2010)] Р. Лав, Разработка ядра Linux, Третье издание, для разработчиков Библиотека (2010).

[Макдугалл и Мауро (2007)] Р. Макдугалл и Дж. Мауро, Solaris Internals, Второе издание, Prentice Hall (2007).

[Органик (1972)] EI Organick, Система Multics: исследование ее структуры, MIT Press (1972).

[Руссинович и др. (2017)] М. Руссинович, Д.А. Соломон и А. Ионеску, Вин- Внутреннее устройство dows – часть 1, седьмое издание, Microsoft Press (2017).

[Сингх (2007)] А. Сингх, Внутреннее устройство Mac OS X: системный подход, Эддисон-Уэсли (2007).

Глава 13 Упражнения

13.9. Рассмотрим файловую систему, в которой файл может быть удалён, а его дисковое пространство восстановлено, при этом ссылки на него сохраняются. Какие проблемы могут возникнуть при создании нового файла в той же области хранения или с тем же абсолютным путём? Как можно избежать этих проблем?

13.10 Таблица открытых файлов используется для хранения информации о файлах, которые в данный момент открыты. Должна ли операционная система поддерживать отдельную таблицу для каждого пользователя или только одну таблицу, содержащую ссылки на файлы, к которым в данный момент обращаются все пользователи? Если к одному и тому же файлу обращаются две разные программы или пользователи, должны ли быть отдельные записи в таблице открытых файлов? Объясните.

13.11 Каковы преимущества и недостатки предоставления обязательных блокировок вместо рекомендательных блокировок, применение которых остается на усмотрение пользователей?

13.12 Приведите примеры приложений, которые обычно получают доступ к файлам следующими методами:

- Последовательный
- Случайный

13.13 Некоторые системы автоматически открывают файл при первом обращении к нему и закрывают его после завершения задания. Обсудите преимущества и недостатки этой схемы по сравнению с более традиционной, где пользователю приходится открывать и закрывать файл вручную.

13.14 Если бы операционная система знала, что определенное приложение будет обращаться к данным файла последовательно, как она могла бы использовать эту информацию для повышения производительности?

13.15 Приведите пример приложения, которое могло бы выиграть от поддержки операционной системы для произвольного доступа к индексированным файлам.

13.16 Некоторые системы обеспечивают общий доступ к файлам, сохраняя одну копию файла. Другие системы поддерживают несколько копий, по одной для каждого пользователя, предоставляющего общий доступ к файлу. Обсудите относительные преимущества каждого подхода.

Файловая система Выполнение

Как мы видели в главе 13, файловая система обеспечивает механизм онлайнового хранения и доступа к содержимому файлов, включая данные и программы. Файловые системы обычно постоянно размещаются на вторичных носителях, предназначенных для хранения больших объёмов данных. В этой главе рассматриваются вопросы, связанные с хранением и доступом к файлам на наиболее распространённых вторичных носителях: жёстких дисках и энергонезависимых запоминающих устройствах. Мы рассмотрим способы структурирования использования файлов, выделения пространства для хранения, восстановления освобождённого пространства, отслеживания местоположения данных и взаимодействия других частей операционной системы с вторичными носителями. Вопросы производительности рассматриваются на протяжении всей главы.

Каждая операционная система общего назначения предоставляет несколько файловых систем. Кроме того, многие операционные системы позволяют администраторам или пользователям добавлять файловые системы. Почему так много? Файловые системы различаются по многим параметрам, включая функции, производительность, надёжность и цели проектирования, и разные файловые системы могут служить разным целям. Например, временная файловая система используется для быстрого хранения и извлечения временных файлов, в то время как файловая система вторичного хранилища по умолчанию (например, Linux ext4) жертвует производительностью ради надёжности и функционально
Как мы видели в ходе этого исследования операционных систем, существует множество вариантов и вариаций, что затрудняет их полное описание. В этой главе мы сосредоточимся на общих чертах.

ЦЕЛИ ГЛАВЫ

- Опишите детали реализации локальных файловых систем и структуры каталогов туры.
- Обсудить алгоритмы распределения блоков и освобождения блоков, а также компромиссы.
- Изучить вопросы эффективности и производительности файловой системы.
- Рассмотреть восстановление после сбоев файловой

системы. • Описать файловую систему WAFL в качестве конкретного примера.

14.1 Структура файловой системы

Диски обеспечивают большую часть вторичного хранилища, на котором поддерживаются файловые системы. Две характеристики делают их удобными для этой цели:

1. Диск можно перезаписать на месте: можно прочитать блок с диска, изменить его и записать обратно в тот же блок.
2. Диск может напрямую обращаться к любому содержащемуся на нём блоку информации. Таким образом, доступ к любому файлу, как последовательно, так и в случайном порядке, прост, а переключение с одного файла на другой требует от привода перемещения головок чтения-записи и ожидания вращения носителя.

Устройства энергонезависимой памяти (NVM) всё чаще используются для хранения файлов и, следовательно, в качестве места для размещения файловых систем. Они отличаются от жёстких дисков тем, что не могут быть перезаписаны на месте, и обладают другими характеристиками производительности. Структура дисков и устройств NVM подробно обсуждается в главе 11.

Для повышения эффективности ввода-вывода операции ввода-вывода между памятью и накопителем выполняются блоками . Каждый блок на жёстком диске состоит из одного или нескольких секторов. В зависимости от типа диска размер сектора обычно составляет 512 или 4096 байт. Устройства NVM обычно используют блоки по 4096 байт, а используемые методы передачи аналогичны используемым в дисководах.

Файловые системы обеспечивают эффективный и удобный доступ к устройству хранения данных, позволяя легко хранить, находить и извлекать данные. Разработка файловой системы ставит две совершенно разные задачи. Первая — определение того, как файловая система должна выглядеть для пользователя. Эта задача включает в себя определение файла и его атрибутов, разрешённых операций с файлом и структуры каталогов для организации файлов. Вторая — создание алгоритмов и структур данных для отображения логической файловой системы на физические вторичные устройства хранения данных.

Сама файловая система, как правило, состоит из множества различных уровней. Структура, показанная на рисунке 14.1, является примером многоуровневой архитектуры. Каждый уровень использует возможности нижних уровней для создания новых функций, доступных на верхних уровнях.

Уровень управления вводом-выводом состоит из драйверов устройств и обработчиков прерываний для передачи информации между основной памятью и дисковой системой. Драйвер устройства можно рассматривать как транслятор. Его входные данные состоят из высокоуровневых команд, таких как «извлечь блок 123». Выходные данные состоят из низкоуровневых, аппаратно-специфичных инструкций, используемых аппаратным контроллером, который связывает устройство ввода-вывода с остальной частью системы. Драйвер устройства обычно записывает определённые битовые последовательности в специальные области памяти контроллера ввода-вывода , чтобы сообщить контроллеру, с каким устройством следует работать и какие действия выполнять.

Подробная информация о драйверах устройств и инфраструктуре ввода-вывода приведена в главе 12.

Базовая файловая система (в Linux называемая «подсистемой блочного ввода -вывода ») должна лишь выдавать общие команды соответствующему драйверу устройства для чтения и записи блоков на устройстве хранения данных. Она выдаёт команды накопителю на основе логических адресов блоков. Она также отвечает за планирование запросов ввода-вывода . Этот уровень также управляет буферами памяти и кэшами, содержащими различные блоки файловой системы, каталогов и данных. Блок в буфере выделяется до того, как может быть выполнена передача блока запоминающего устройства. Когда буфер заполнен, диспетчер буферов должен найти дополнительную буферную память или освободить пространство, чтобы обеспечить...

Рисунок 14.1 Многоуровневая файловая система.

Запрошенный ввод-вывод для завершения. Кэши используются для хранения часто используемых метаданных файловой системы для повышения производительности, поэтому управление их содержимым критически важно для оптимальной производительности системы.

Модуль организации файлов знает о файлах и их логических блоках. Логические блоки каждого файла пронумерованы от 0 (или 1) до N. Модуль организации файлов также включает менеджер свободного пространства, который отслеживает нераспределенные блоки и предоставляет эти блоки модулю организации файлов по запросу.

Наконец, логическая файловая система управляет метаданными. Метаданные включают в себя всю структуру файловой системы, за исключением самих данных (или содержимого файлов). Логическая файловая система управляет структурой каталогов, предоставляя модулю организации файлов необходимую ему информацию, используя символическое имя файла. Она поддерживает структуру файла с помощью блоков управления файлами (FCB). Блок управления файлами (FCB) (инод в файловых системах UNIX) содержит информацию о файле, включая владельца, права доступа и расположение его содержимого. Логическая файловая система также отвечает за защиту, как описано в главах 13 и 17.

При использовании многоуровневой структуры для реализации файловой системы дублирование кода сводится к минимуму. Управление вводом-выводом , а иногда и базовый код файловой системы могут использоваться несколькими файловыми системами. Каждая файловая система может иметь собственную логическую файловую систему и модули организации файлов. К сожалению, многоуровневая структура может привести к увеличению накладных расходов операционной системы, что может привести к снижению производительности. Использование многоуровневой структуры, включая определение количества используемых слоев и функций каждого слоя, представляет собой серьёзную проблему при проектировании новых систем.

Сегодня используется множество файловых систем, и большинство операционных систем поддерживают более одной. Например, большинство компакт-дисков записаны в формате ISO 9660 — стандартном формате, согласованном производителями компакт-дисков . Помимо файловых систем для сменных носителей, каждая операционная система имеет одну или несколько файловых систем на дисках. UNIX использует файловую систему UNIX (UFS), основанную на

Berkeley Fast File System (FFS). Windows поддерживает форматы дисковых файловых систем FAT, FAT32 и NTFS (или файловую систему Windows NT), а также форматы файловых систем CD-ROM и DVD . Хотя Linux поддерживает более 130 различных файловых систем, стандартная файловая система Linux известна как расширенная файловая система, наиболее распространённые версии которой — ext3 и ext4. Существуют также распределённые файловые системы, в которых файловая система на сервере монтируется одним или несколькими клиентскими компьютерами по сети.

Исследования файловых систем продолжают оставаться активной областью разработки и внедрения операционных систем. Google создала собственную файловую систему для удовлетворения специфических потребностей компании в хранении и поиске данных, включая высокопроизводительный доступ множества клиентов к очень большому количеству дисков. Ещё один интересный проект — файловая система FUSE , которая обеспечивает гибкость в разработке и использовании файловых систем, реализуя и выполняя файловые системы на уровне пользователя, а не на уровне ядра. Используя FUSE, пользователь может добавить новую файловую систему в различные операционные системы и использовать её для управления своими файлами.

14.2 Операции файловой системы

Как было описано в разделе 13.1.2, операционные системы реализуют системные вызовы open() и close() для запросов процессов на доступ к содержимому файлов. В этом разделе мы подробно рассмотрим структуры и операции, используемые для реализации операций в файловой системе.

14.2.1 Обзор

Для реализации файловой системы используется несколько структур, хранящихся в памяти и на накопителе. Эти структуры различаются в зависимости от операционной системы и файловой системы, но некоторые общие принципы применимы.

На диске файловая система может содержать информацию о том, как загрузить операционную систему, общее количество блоков, количество и расположение свободных блоков, структуру каталогов и отдельных файлов. Многие из этих структур подробно описаны в оставшейся части этой главы. Здесь мы кратко опишем их:

- Блок управления загрузкой (для каждого тома) может содержать информацию, необходимую системе для загрузки операционной системы с этого тома. Если диск не содержит операционной системы, этот блок может быть пустым. Обычно это первый блок тома. В UFS он называется загрузочным блоком. В NTFS это загрузочный сектор раздела.
- Блок управления томом (для каждого тома) содержит сведения о томе, такие как количество блоков в томе, размер блоков, количество свободных блоков и указатели свободных блоков, а также количество свободных FCB и указатели FCB . В UFS это называется суперблоком. В NTFS он хранится в главной таблице файлов.
- Структура каталогов (для каждой файловой системы) используется для организации файлов. В UFS это включает имена файлов и связанные с ними номера инодов. В NTFS это хранится в главной таблице файлов.

- Блок FCB для каждого файла содержит множество сведений о нём. Он имеет уникальный идентификатор, позволяющий связать его с записью каталога. В NTFS эта информация хранится в главной таблице файлов, которая использует структуру реляционной базы данных, где каждому файлу соответствует отдельная строка.

Информация в памяти используется как для управления файловой системой, так и для повышения производительности посредством кэширования. Данные загружаются при монтировании, обновляются во время операций с файловой системой и удаляются при размонтировании. Могут быть включены различные типы структур.

- Таблица монтирования в памяти содержит информацию о каждом смонтированном объем.
- Кэш структуры каталогов в памяти хранит информацию о каталогах, к которым недавно осуществлялся доступ. (Для каталогов, в которых смонтированы тома, он может содержать указатель на таблицу томов.) • Системная таблица открытых файлов

содержит копию FCB каждого открытого файла, а также другую информацию.

- Таблица открытых файлов для каждого процесса содержит указатели на соответствующие записи в общесистемной таблице открытых файлов, а также другую информацию для всех файлов, открытых процессом.
- Буферы хранят блоки файловой системы при их чтении из файловой системы или записи в нее.

Для создания нового файла процесс обращается к логической файловой системе. Логическая файловая система знает формат структур каталогов. Для создания нового файла она выделяет новый блок FCB. (Или, если реализация файловой системы создаёт все блоки FCB при её создании, блок FCB выделяется из набора свободных блоков FCB.) Затем система считывает соответствующий каталог в память, обновляет его, добавляя новое имя файла и блок FCB, и записывает его обратно в файловую систему. Типичный блок FCB показан на рисунке 14.2.

Некоторые операционные системы, включая UNIX, обрабатывают каталог точно так же, как файл, имеющий поле «тип», указывающее, что это каталог. Другие операционные системы

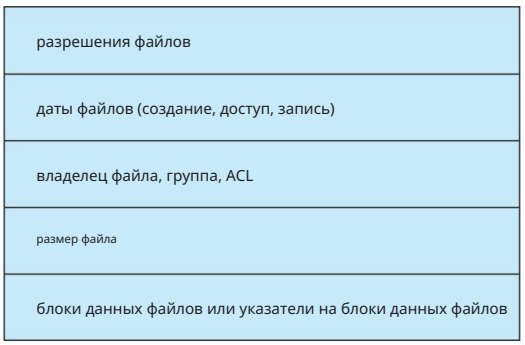


Рисунок 14.2 Типичный блок управления файлами.

В системах управления данными, включая Windows, реализованы отдельные системные вызовы для файлов и каталогов, и каталоги рассматриваются как сущности, отдельные от файлов. Независимо от более масштабных структурных проблем, логическая файловая система может вызвать модуль организации файлов для сопоставления операций ввода-вывода каталога с расположением блоков памяти, которые передаются базовой файловой системе и системе управления вводом-выводом .

14.2.2 Использование

Теперь, когда файл создан, его можно использовать для ввода-вывода. Однако сначала его нужно открыть. Вызов open() передает имя файла в логическую файловую систему. Системный вызов open() сначала выполняет поиск в общесистемной таблице открытых файлов, чтобы выяснить, не используется ли файл другим процессом. Если используется, создается запись в таблице открытых файлов для каждого процесса, указывающая на существующую общесистемную таблицу открытых файлов. Этот алгоритм может существенно сэкономить на накладных расходах. Если файл еще не открыт, в структуре каталогов выполняется поиск заданного имени файла. Части структуры каталогов обычно кэшируются в памяти для ускорения операций с каталогами. Как только файл найден, FCB копируется в общесистемную таблицу открытых файлов в памяти. Эта таблица не только хранит FCB , но и отслеживает количество процессов, которые открыли файл.

Затем в таблицу открытых файлов для каждого процесса добавляется запись, содержащая указатель на запись в общесистемной таблице открытых файлов и некоторые другие поля. Эти поля могут включать указатель на текущее местоположение в файле (для следующей операции read() или write()) и режим доступа, в котором открыт файл.

Вызов open() возвращает указатель на соответствующую запись в таблице файловой системы для каждого процесса. Все файловые операции затем выполняются через этот указатель. Имя файла может не быть частью таблицы открытых файлов, поскольку система не будет использовать его после того, как соответствующий блок FCB будет найден на диске. Однако его можно кэшировать для экономии времени при последующих открытиях того же файла. Имя записи может быть разным. В системах UNIX она называется дескриптором файла (fil descriptor), а в Windows — дескриптором файла (fil handle).

Когда процесс закрывает файл, запись из таблицы для каждого процесса удаляется, а общесистемный счётчик открытых файлов уменьшается. Когда все пользователи, открывшие файл, закрывают его, все обновлённые метаданные копируются обратно в структуру каталогов на диске, а запись из общесистемной таблицы открытых файлов удаляется.

Не следует упускать из виду аспекты кэширования структур файловой системы.
Большинство систем хранят в памяти всю информацию об открытом файле, за исключением самих блоков данных. Система BSD UNIX типична тем, что использует кэши везде, где возможно сохранение дискового ввода-вывода . Её средний процент попаданий в кэш, равный 85%, показывает, что эти методы вполне оправданы. Система BSD UNIX подробно описана в Приложении C.

Операционные структуры реализации файловой системы представлены на рисунке 14.3.

14.3 Реализация каталога

Выбор алгоритмов распределения и управления каталогами существенно влияет на эффективность, производительность и надежность файловой системы. В этом разделе мы обсудим компромиссы, связанные с выбором одного из этих алгоритмов.

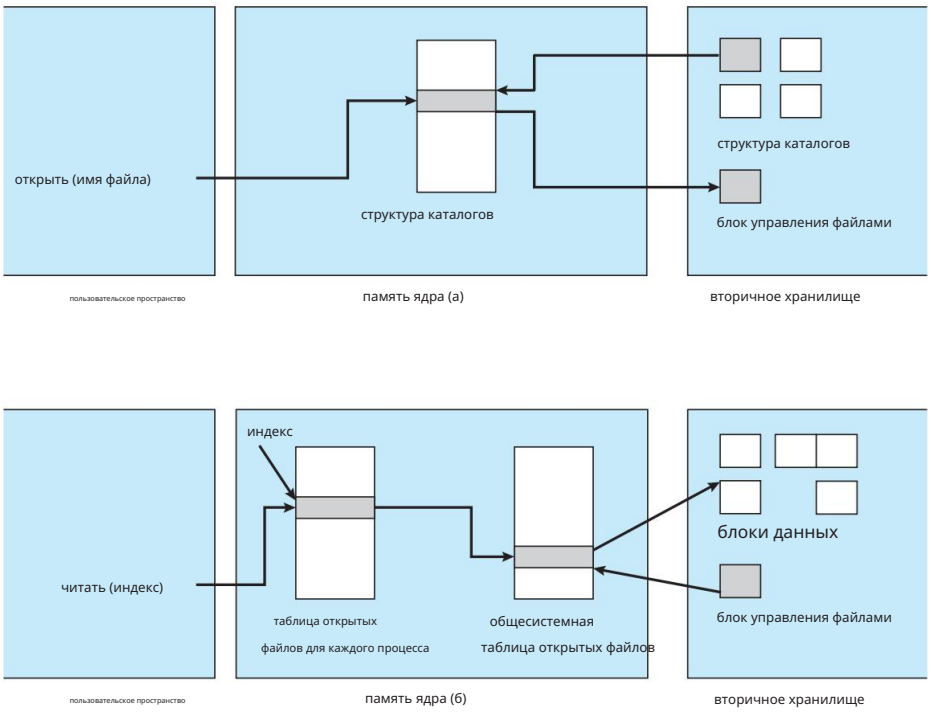


Рисунок 14.3 Структуры файловой системы в памяти. (а) Файл открыт. (b) Файл прочитан.

14.3.1 Линейный список

Простейший способ реализации каталога — использование линейного списка имён файлов с указателями на блоки данных. Этот метод прост в программировании, но требует много времени на выполнение. Чтобы создать новый файл, необходимо сначала выполнить поиск в каталоге, чтобы убедиться, что ни один существующий файл не имеет того же имени. Затем мы добавляем новую запись в конец каталога. Чтобы удалить файл, мы ищем в каталоге указанный файл и освобождаем выделенное для него место. Чтобы повторно использовать запись каталога, можно сделать одно из нескольких. Мы можем пометить запись как неиспользуемую (присвоив ей специальное имя, например, полностью пустое имя, присвоив ей недопустимый номер инода (например, 0) или добавив бит used –unused в каждую запись), или мы можем добавить её в список свободных записей каталога. Третий вариант — скопировать последнюю запись каталога в освободившееся место и уменьшить длину каталога. Связанный список также может использоваться для сокращения времени, необходимого для удаления файла.

Реальный недостаток линейного списка записей каталога заключается в том, что для поиска файла требуется линейный поиск. Информация каталога используется часто, и пользователи заметят это, если доступ к ней медленный. Фактически, многие операционные системы используют программный кэш для хранения последней использованной информации каталога. Попадание в кэш позволяет избежать необходимости постоянного повторного считывания информации из вторичного хранилища. Сортированный список позволяет выполнять двоичный поиск и сокращает среднее время поиска. Однако требование, чтобы список был отсортированным, может усложнить создание и удаление файлов, поскольку может потребоваться перемещение значительных объёмов данных.

Информация каталога для поддержания его в отсортированном виде. В этом случае может помочь более сложная древовидная структура данных, например, сбалансированное дерево. Преимущество отсортированного списка заключается в том, что отсортированный список каталога можно получить без отдельного этапа сортировки.

14.3.2 Хэш-таблица

Другая структура данных, используемая для каталога файлов, — это хэш-таблица. Здесь записи каталога хранятся в линейном списке, но также используется хэш-структура данных. Хэш-таблица принимает значение, вычисленное из имени файла, и возвращает указатель на имя файла в линейном списке. Таким образом, она может значительно сократить время поиска в каталоге.
Вставка и удаление также довольно просты, хотя необходимо предусмотреть некоторые коллизии — ситуации, в которых два имени файлов попадают в одно и то же место.

Основные трудности с хеш-таблицей заключаются в её, как правило, фиксированном размере и зависимости хеш-функции от этого размера. Например, предположим, что мы создаём хеш-таблицу с линейным зондированием, содержащую 64 записи. Хеш-функция преобразует имена файлов в целые числа от 0 до 63 (например, используя остаток от деления на 64). Если позже мы попытаемся создать 65-й файл, нам придётся увеличить хеш-таблицу каталога, скажем, до 128 записей. В результате нам потребуется новая хеш-функция, которая должна сопоставлять имена файлов с диапазоном от 0 до 127, и нам нужно реорганизовать существующие записи каталога, чтобы отразить их новые значения хеш-функции.

В качестве альтернативы можно использовать хэш-таблицу с цепочкой переполнений. Каждая запись хэша может представлять собой связанный список, а не отдельное значение, и мы можем разрешать коллизии, добавляя новую запись в связанный список. Поиск может несколько замедлиться, поскольку для поиска имени может потребоваться пошаговый просмотр связанного списка конфликтующих записей таблицы. Тем не менее, этот метод, вероятно, будет намного быстрее линейного поиска по всему каталогу.

14.4 Методы распределения

Прямой доступ к файлам во вторичном хранилище обеспечивает гибкость при реализации. Практически всегда на одном устройстве хранится множество файлов. Основная проблема заключается в том, как выделить место под эти файлы, чтобы эффективно использовать его и обеспечить быстрый доступ к файлам. Широко распространены три основных метода выделения пространства во вторичном хранилище: непрерывный, связанный и индексированный. Каждый метод имеет свои преимущества и недостатки. Хотя некоторые системы поддерживают все три, чаще всего система использует один метод для всех файлов в файловой системе одного типа.

14.4.1 Непрерывное распределение

Непрерывное размещение требует, чтобы каждый файл занимал набор смежных блоков на устройстве. Адреса устройств определяют линейный порядок на устройстве. При таком размещении, если к устройству обращается только одно задание, доступ к блоку b + 1 после блока b обычно не требует перемещения головки. Когда требуется перемещение головки (из последнего сектора одного цилиндра в первый сектор следующего цилиндра), головке достаточно переместиться только с одной дорожки на другую. Таким образом, для жёстких дисков количество обращений к диску, необходимое для доступа к смежным файлам, равно

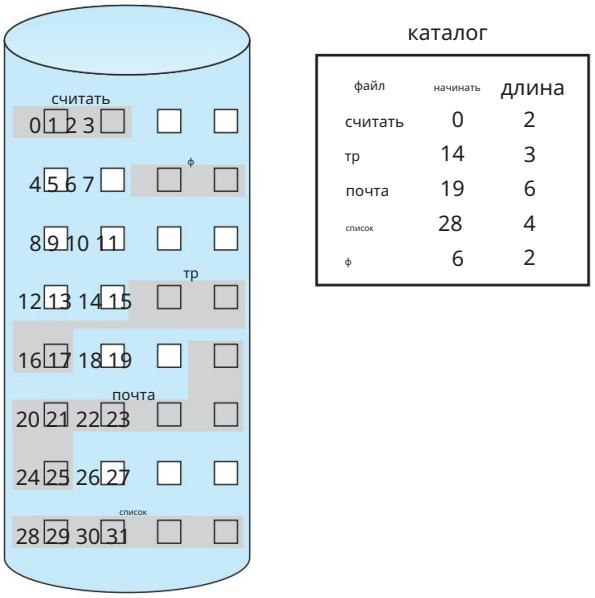


Рисунок 14.4 Непрерывное распределение дискового пространства.

минимально (предполагая, что блоки с близкими логическими адресами близки физически), так как время поиска, когда поиск наконец необходим.

Непрерывное размещение файла определяется адресом первого блока. и длина (в блоках) файла. Если файл имеет длину n блоков и начинается с местоположение b, то оно занимает блоки b, b + 1, b + 2, ..., b + n 1. Запись каталога для каждого файла указывается адрес начального блока и длина области выделено для этого файла (рис. 14.4). Непрерывное выделение памяти легко реализовать. но имеет ограничения и поэтому не используется в современных файловых системах.

Доступ к файлу, выделенному последовательно, прост. Для последовательного доступа доступа, файловая система запоминает адрес последнего использованного блока и, При необходимости считывает следующий блок. Для прямого доступа к блоку i файла, который начинается с блока b, мы можем сразу получить доступ к блоку b + i. Таким образом, оба последовательных и прямой доступ может поддерживаться непрерывным распределением.

Однако непрерывное выделение памяти имеет некоторые проблемы. Одна из них — поиск места для нового файла. Система, выбранная для управления свободным пространством, определяет как эта задача выполняется; эти системы управления обсуждаются в Раздел 14.5. Можно использовать любую систему управления, но некоторые из них медленнее, чем другие.

Проблему непрерывного распределения можно рассматривать как частное приложение общей проблемы динамического распределения памяти, обсуждаемой в разделе 9.2, которая включает в себя, как удовлетворить запрос размера n из списка свободных отверстий. Сначала Подгонка и наилучшая подгонка являются наиболее распространенными стратегиями, используемыми для выбора свободной лунки из набор доступных отверстий. Моделирование показало, что как первый подходящий, так и наилучший подходящий более эффективны, чем наихудшие варианты, с точки зрения использования времени и хранилища. Ни первый подходящий, ни наилучший подходящий не являются однозначно лучшими с точки зрения использования хранилища, но первый подгонка обычно происходит быстрее.

Все эти алгоритмы страдают от проблемы внешней фрагментации. файлы распределяются и удаляются, свободное пространство для хранения разбивается на небольшие части.

Внешняя фрагментация возникает всякий раз, когда свободное пространство разбивается на фрагменты. Она становится проблемой, когда самого большого непрерывного фрагмента недостаточно для выполнения запроса; хранилище фрагментируется на несколько фрагментов, ни один из которых не имеет достаточного размера для хранения данных. В зависимости от общего объёма дискового пространства и среднего размера файла внешняя фрагментация может представлять собой как незначительную, так и серьёзную проблему.

Одной из стратегий предотвращения потери значительных объемов дискового пространства из-за внешней фрагментации является копирование всей файловой системы на другое устройство. Затем исходное устройство полностью освобождается, создавая одно большое непрерывное свободное пространство. Затем мы копируем файлы обратно на исходное устройство, выделяя непрерывное пространство из этого большого свободного пространства. Эта схема эффективно сжимает всё свободное пространство в одно непрерывное пространство, решая проблему фрагментации.
Однако стоимость такого уплотнения — это время, которое может быть особенно высоким для больших устройств хранения данных. Уплотнение таких устройств может занимать часы и может потребоваться еженедельно. В некоторых системах эта функция должна выполняться в автономном режиме, с демонтированной файловой системой. В период простоя нормальная работа системы, как правило, невозможна, поэтому на рабочих машинах такого уплотнения всеми силами избегают. Большинство современных систем, которым требуется дефрагментация, могут выполнять её в режиме онлайн во время обычной работы системы, но при этом производительность может существенно снизиться.

Другая проблема, связанная с непрерывным выделением памяти, — определение необходимого для файла объёма пространства. При создании файла необходимо определить и выделить общий объём необходимого ему пространства. Как создатель (программа или пользователь) узнаёт размер создаваемого файла? В некоторых случаях это определение может быть довольно простым (например, копирование существующего файла). Однако в общем случае оценить размер выходного файла может быть сложно.

Если выделить файлу слишком мало места, мы можем обнаружить, что его невозможно расширить. Особенно при использовании стратегии распределения по оптимальному размеру, пространство по обе стороны файла может быть занято. Следовательно, мы не можем увеличить файл на месте.
Далее возможны два варианта. Во-первых, пользовательская программа может быть завершена с соответствующим сообщением об ошибке. После этого пользователю необходимо выделить больше места и запустить программу снова. Эти повторные запуски могут быть дорогостоящими. Чтобы избежать этого, пользователь обычно переоценивает объём необходимого пространства, что приводит к значительному его расходу впустую. Другой вариант — найти большее свободное пространство, скопировать содержимое файла в новое пространство и освободить предыдущее. Эту последовательность действий можно повторять до тех пор, пока есть свободное место, хотя это может занять много времени. Однако пользователю не нужно явно сообщать о происходящем; система продолжает работу, несмотря на проблему, хотя и всё медленнее.

Даже если общий объём необходимого файлу пространства известен заранее, предварительное выделение может быть неэффективным. Файлу, который будет медленно расти в течение длительного периода (месяцы или годы), необходимо выделить достаточно места для достижения его конечного размера, даже если большая часть этого пространства будет неиспользована в течение длительного времени. Поэтому файл имеет высокую степень внутренней фрагментации.

Чтобы минимизировать эти недостатки, операционная система может использовать модифицированную схему непрерывного выделения пространства. В этом случае изначально выделяется непрерывный фрагмент пространства. Затем, если этого объёма оказывается недостаточно, добавляется ещё один непрерывный фрагмент пространства, называемый экстентом . Расположение блоков файла затем записывается в виде местоположения и количества блоков, а также ссылки на первый блок следующего экстента. В некоторых системах владелец файла может задать размер экстента, но эта настройка приводит к неэффективности, если владелец указан неверно.

Фрагментация может по-прежнему представлять проблему, если экстенты слишком велики, а внешняя фрагментация может стать проблемой при выделении и освобождении экстентов разного размера. Коммерческая файловая система Symantec Veritas использует экстенты для оптимизации производительности. Veritas — это высокопроизводительная замена стандартной файловой системы UNIX UFS.

14.4.2 Связанное распределение

Связанное распределение решает все проблемы непрерывного распределения. При связанном распределении каждый файл представляет собой связанный список блоков памяти; блоки могут быть разбросаны по всему устройству. Каталог содержит указатель на первый и последний блоки файла. Например, файл из пяти блоков может начинаться с блока 9 и продолжаться блоком 16, затем блоком 1, затем блоком 10 и, наконец, блоком 25 (рис. 14.5). Каждый блок содержит указатель на следующий блок. Эти указатели недоступны пользователю. Таким образом, если размер каждого блока составляет 512 байт, а адрес блока (указатель) занимает 4 байта, то пользователь видит блоки по 508 байт.

Чтобы создать новый файл, мы просто создаём новую запись в каталоге. При связанном выделении памяти каждая запись каталога имеет указатель на первый блок файла. Этот указатель инициализируется значением null (значение указателя конца списка), что означает пустой файл. Поле размера также устанавливается равным 0. Запись в файл заставляет систему управления свободным пространством найти свободный блок, и этот новый блок записывается в конец файла и связывается с ним. Чтобы прочитать файл, мы просто считываем блоки, следуя указателям от блока к блоку. При связанном выделении памяти внешняя фрагментация отсутствует, и любой свободный блок в списке свободного пространства может быть использован для выполнения запроса. Размер файла не обязательно объявлять при его создании.
Файл может продолжать расти до тех пор, пока есть свободные блоки. Следовательно, нет необходимости сжимать дисковое пространство.

Однако у связанного выделения памяти есть недостатки. Главная проблема заключается в том, что его можно эффективно использовать только для файлов с последовательным доступом. Чтобы найти i-й

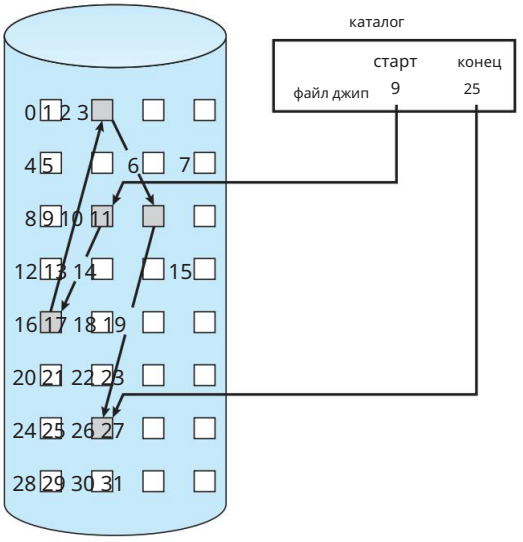


Рисунок 14.5 Связанное распределение дискового пространства.

Для доступа к блоку файла необходимо начать с его начала и следовать указателям, пока не доберёмся до i-го блока. Каждый доступ к указателю требует чтения с устройства хранения, а в некоторых случаях — поиска на жёстком диске . Следовательно, поддержка прямого доступа к файлам со связанным распределением памяти неэффективна.

Другим недостатком является объём, необходимый для указателей. Если указатель занимает 4 байта из 512-байтового блока, то 0,78% диска используется для указателей, а не для информации. Каждый файл занимает немного больше места, чем обычно.

Обычным решением этой проблемы является объединение блоков в группы, называемые кластерами, и выделение кластеров, а не блоков. Например, файловая система может определить кластер как четыре блока и работать на вторичном устройстве хранения только в виде кластерных единиц. В этом случае указатели используют гораздо меньший процент пространства файла. Этот метод позволяет упростить сопоставление логических блоков с физическими, но повышает пропускную способность жесткого диска (поскольку требуется меньше обращений к головкам диска) и уменьшает объем памяти, необходимый для выделения блоков и управления свободными списками. Цена такого подхода — увеличение внутренней фрагментации, поскольку при частичном заполнении кластера теряется больше места, чем при частичном заполнении блока. Кроме того, страдает производительность случайного ввода-вывода , поскольку запрос небольшого объема данных переносит большой объем данных. Кластеры могут использоваться для сокращения времени доступа к диску и для многих других алгоритмов, поэтому они используются в большинстве файловых систем.

Ещё одна проблема связанного размещения — надёжность. Вспомните, что файлы связаны между собой указателями, разбросанными по всему устройству, и представьте, что произойдёт, если указатель будет потерян или повреждён. Ошибка в программном обеспечении операционной системы или аппаратный сбой могут привести к выбору неверного указателя.
Эта ошибка, в свою очередь, может привести к связыванию со списком свободного пространства или с другим файлом. Одним из частичных решений является использование двусвязных списков, а другим — хранение имени файла и относительного номера блока в каждом блоке. Однако эти схемы требуют ещё больших накладных расходов для каждого файла.

Важным вариантом связанного распределения является использование таблицы распределения файлов (FAT). Этот простой, но эффективный метод распределения дискового пространства использовался операционной системой MS-DOS . Для размещения таблицы выделяется раздел в начале каждого тома. Таблица содержит одну запись для каждого блока и индексируется по номеру блока. FAT используется практически так же, как связанный список. Запись каталога содержит номер первого блока файла. Запись таблицы, индексированная этим номером блока, содержит номер следующего блока в файле. Эта цепочка продолжается до тех пор, пока не достигнет последнего блока, имеющего специальное значение конца файла в качестве записи таблицы. Неиспользуемый блок обозначается значением таблицы, равным 0. Выделение нового блока файлу заключается в простом поиске первой записи таблицы со значением 0 и замене предыдущего значения конца файла адресом нового блока. Затем 0 заменяется значением конца файла. Наглядным примером является структура FAT , показанная на рисунке 14.6 для файла, состоящего из дисковых блоков 217, 618 и 339.

Схема распределения FAT может привести к значительному количеству обращений к головке диска, если FAT не кэшируется. Головка диска должна переместиться в начало тома, чтобы прочитать FAT и найти местоположение нужного блока, а затем переместиться к местоположению самого блока. В худшем случае оба перемещения выполняются для каждого блока. Преимущество заключается в уменьшении времени произвольного доступа, поскольку головка диска может найти местоположение любого блока, считывая информацию из FAT .

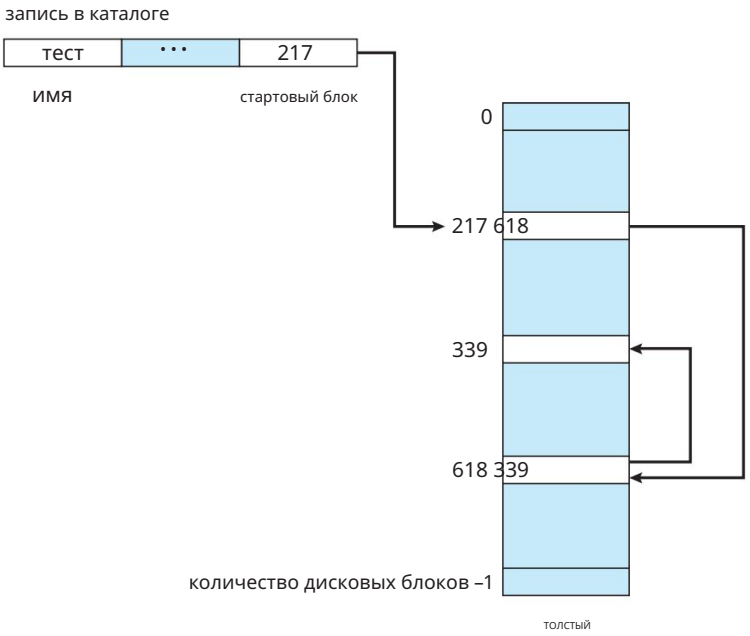


Рисунок 14.6 Таблица распределения файлов.

14.4.3 Индексированное распределение

Связанное выделение решает проблемы внешней фрагментации и объявления размера, характерные для непрерывного выделения. Однако без FAT связанное выделение не может обеспечить эффективный прямой доступ, поскольку указатели на блоки разбросаны по всему диску вместе с самими блоками и должны извлекаться по порядку. Индексированное выделение решает эту проблему, объединяя все указатели в одном месте: индексном блоке.

Каждый файл имеет свой индексный блок, представляющий собой массив адресов блоков хранения. Запись i-го индексного блока указывает на i-й блок файла.
Каталог содержит адрес индексного блока (рис. 14.7). Для поиска и чтения i-го блока мы используем указатель в записи i-го индексного блока. Эта схема аналогична схеме страничного доступа, описанной в разделе 9.3.

При создании файла все указатели в индексном блоке устанавливаются в значение NULL. При первой записи i-го блока блок извлекается из менеджера свободного пространства, и его адрес помещается в запись i-го индексного блока.

Индексированное выделение поддерживает прямой доступ, избегая внешней фрагментации, поскольку любой свободный блок на устройстве хранения может удовлетворить запрос на дополнительное пространство. Однако индексированное выделение страдает от непроизводительного расхода пространства. Накладные расходы на указатель индексного блока, как правило, превышают накладные расходы на указатель при связанном выделении. Рассмотрим распространённый случай: файл состоит всего из одного или двух блоков. При связанном выделении мы теряем пространство только в размере одного указателя на блок. При индексированном выделении необходимо выделить весь индексный блок, даже если только один или два указат

В связи с этим возникает вопрос о размере индексного блока. Каждый файл должен иметь индексный блок, поэтому мы хотим, чтобы он был как можно меньше.

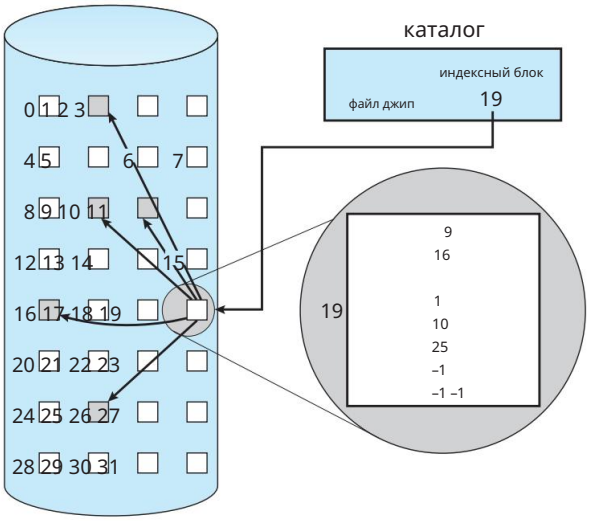


Рисунок 14.7 Индексированное распределение дискового пространства.

Это возможно. Однако, если индексный блок слишком мал, он не сможет содержать достаточно указателей для большого файла, и потребуется механизм для решения этой проблемы. Для этого используются следующие механизмы:

- Связанная схема. Индексный блок обычно представляет собой один блок хранения. Таким образом, его можно читать и записывать непосредственно сам по себе. Чтобы обеспечить работу с большими файлами, можно связать несколько индексных блоков. Например, индексный блок может содержать небольшой заголовок с именем файла и набором из первых 100 адресов блоков диска. Следующий адрес (последнее слово в индексном блоке) равен нулю (для небольшого файла) или является указателем на другой индексный блок (для большого файла).

- Многоуровневый индекс. Вариант связанного представления использует индексный блок первого уровня для указания на набор индексных блоков второго уровня, которые, в свою очередь, указывают на блоки файла. Для доступа к блоку операционная система использует индекс первого уровня для поиска индексного блока второго уровня, а затем использует этот блок для поиска нужного блока данных. Этот подход можно расширить до третьего или четвёртого уровня, в зависимости от желаемого максимального размера файла. При блоках по 4096 байт мы можем хранить 1024 четырёхбайтовых указателя в индексном блоке. Два уровня индексов позволяют хранить 1 048 576 блоков данных и файл размером до 4 ГБ.

- Комбинированная схема. Другой вариант, используемый в файловых системах UNIX, заключается в хранении первых, скажем, 15 указателей индексного блока в индексном узле файла. Первые 12 из этих указателей указывают на прямые блоки, то есть содержат адреса блоков, содержащих данные файла. Таким образом, данные для небольших файлов (не более 12 блоков) не требуют отдельного индексного блока. Если размер блока составляет 4 КБ, то возможен прямой доступ к данным объёмом до 48 КБ . Следующие три указателя указывают на косвенные блоки. Первый указывает на один косвенный блок, представляющий собой индексный блок, содержащий не данные, а адреса блоков, содержащих данные. Второй указывает на двойной косвенный блок, содержащий адрес блока, содержащего адреса блоков, содержащих указатели на блоки данных. Последний указатель содержит адрес тройного косвенного блока. (Инод UNIX показан на рисунке 14.8.)

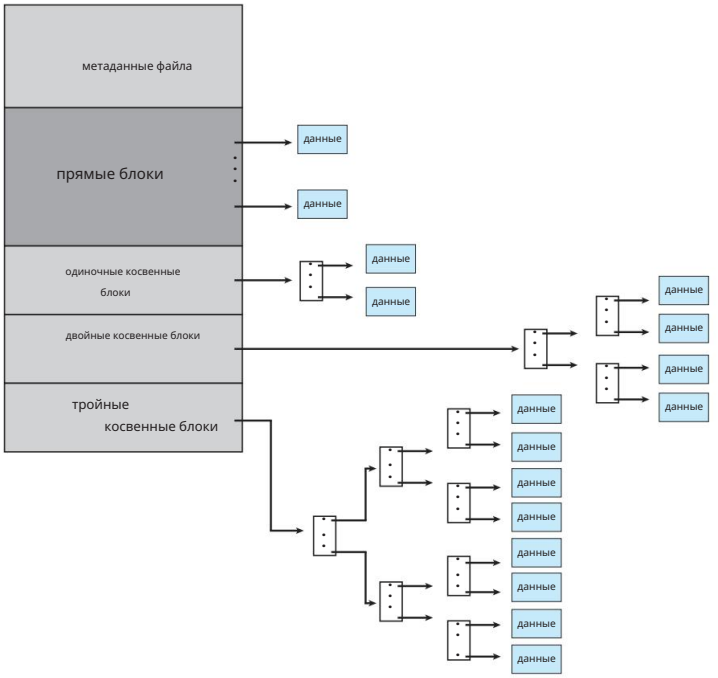


Рисунок 14.8. Узел UNIX.

При использовании этого метода количество блоков, которые можно выделить файлу, превышает объём пространства, адресуемого 4-байтовыми файловыми указателями, используемыми во многих операционных системах. 32-битный файловый указатель достигает всего 232 байт, или 4 ГБ. Многие реализации UNIX и Linux теперь поддерживают 64-битные файловые указатели, что позволяет файлам и файловым системам достигать размера в несколько эксбибайт.
Файловая система ZFS поддерживает 128-битные указатели файлов.

Схемы индексированного распределения страдают от тех же проблем с производительностью, что и схемы связанного распределения. В частности, блоки индекса могут кэшироваться в памяти, но блоки данных могут быть разбросаны по всему тому.

14.4.4 Производительность

Рассмотренные нами методы распределения различаются по эффективности хранения и времени доступа к блокам данных. Оба эти фактора являются важными критериями при выборе подходящего метода или методов для реализации в операционной системе.

Прежде чем выбрать метод распределения, необходимо определить, как будут использоваться системы. Система с преимущественно последовательным доступом не должна использовать тот же метод, что и система с преимущественно случайным доступом.

При любом типе доступа непрерывное выделение памяти требует только одного доступа для получения блока. Поскольку начальный адрес файла легко хранить в памяти, мы можем мгновенно вычислить адрес i-го (или следующего) блока и напрямую прочитать его.

При связанном выделении памяти мы также можем хранить адрес следующего блока в памяти и читать его напрямую. Этот метод подходит для последовательного доступа; однако при прямом доступе доступ к i-му блоку может потребовать i операций чтения. Это

проблема указывает, почему связанное выделение не следует использовать для приложений, требующих прямого доступа.

В результате некоторые системы поддерживают файлы с прямым доступом, используя непрерывное выделение памяти, а файлы с последовательным доступом — связанное выделение памяти. В этих системах тип доступа должен быть объявлен при создании файла.
Файл, созданный для последовательного доступа, будет связан и не может быть использован для прямого доступа. Файл, созданный для прямого доступа, будет непрерывным и может поддерживать как прямой, так и последовательный доступ, но его максимальная длина должна быть объявлена при создании. В этом случае операционная система должна иметь соответствующие структуры данных и алгоритмы для поддержки обоих методов выделения памяти. Файлы можно преобразовать из одного типа в другой, создав новый файл нужного типа, в который копируется содержимое старого файла. После этого старый файл можно удалить, а новый переименовать.

Индексированное выделение памяти сложнее. Если индексный блок уже находится в памяти, доступ к нему можно осуществить напрямую. Однако хранение индексного блока в памяти требует значительного объёма. Если этот объём памяти недоступен, может потребоваться сначала прочитать индексный блок, а затем нужный блок данных. Для двухуровневого индекса может потребоваться два чтения индексного блока.
Для очень большого файла доступ к блоку, расположенному ближе к концу файла, потребовал бы чтения всех индексных блоков, прежде чем будет прочитан нужный блок данных. Таким образом, производительность индексированного выделения памяти зависит от структуры индекса, размера файла и желаемого положения блока.

Некоторые системы комбинируют непрерывное и индексированное выделение памяти, используя непрерывное выделение памяти для небольших файлов (до трёх-четырёх блоков) и автоматически переключаясь на индексированное выделение памяти при увеличении размера файла. Поскольку большинство файлов небольшие, а непрерывное выделение памяти для них эффективно, средняя производительность может быть весьма высокой.

Используется множество других оптимизаций. Учитывая разницу между скоростью процессора и скоростью диска, вполне разумно добавить тысячи дополнительных инструкций в операционную систему, чтобы сэкономить всего несколько движений головки диска. Более того, эта разница со временем увеличивается, и теперь для оптимизации перемещения головки можно было бы использовать сотни тысяч инструкций.
менты.

В устройствах NVM поиск головки диска не производится, поэтому требуются другие алгоритмы и оптимизации. Использование устаревшего алгоритма, который тратит много процессорных циклов, пытаясь избежать несуществующего перемещения головки, было бы крайне неэффективно. Существующие файловые системы модифицируются и создаются новые для достижения максимальной производительности устройств хранения данных NVM . Эти разработки направлены на сокращение количества инструкций и общего пути между устройством хранения и доступом приложения к данным.

14.5 Управление свободным пространством

Поскольку объём хранилища ограничен, необходимо повторно использовать пространство, занимаемое удалёнными файлами, для новых файлов, если это возможно. (Оптические диски с однократной записью допускают только одну запись в любой сектор, поэтому повторное использование физически невозможно.) Для отслеживания свободного места на диске система ведёт список свободного места. В нём записываются все свободные блоки устройства — те, которые не выделены для какого-либо файла или каталога. Чтобы создать файл, мы ищем в списке свободного места необходимый объём и выделяем его.

Это пространство будет выделено новому файлу. Затем оно будет удалено из списка свободного пространства. При удалении файла его свободное пространство добавляется в список свободного пространства. Список свободного пространства, несмотря на название, не обязательно реализуется как список, как мы уже обсуждали. следующий.

14.5.1 Битовый вектор

Список свободного пространства часто реализуется в виде битовой карты или битового вектора. Каждый блок представлен одним битом. Если блок свободен, бит равен 1; если блок выделен, бит равен 0.

Например, рассмотрим диск, на котором блоки 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 и 27 свободны, а остальные блоки распределены. Битовая карта свободного пространства будет выглядеть следующим образом:

001111001111110001100000011100000 ...

Основным преимуществом этого подхода является его относительная простота и эффективность при поиске первого свободного блока или n последовательных свободных блоков на диске. Действительно, многие компьютеры предоставляют инструкции по битовой манипуляции, которые можно эффективно использовать для этой цели. Один из методов поиска первого свободного блока в системе, использующей битовый вектор для выделения памяти, заключается в последовательной проверке каждого слова в битовой карте на предмет того, не равно ли оно нулю, поскольку слово со значением 0 содержит только 0 битов и представляет собой набор выделенных блоков. Первое ненулевое слово сканируется на предмет первого бита 1, который указывает на местоположение первого свободного блока.
Расчет номера блока

(количество бит в слове) × (количество слов с 0-значением) + смещение первого 1 бита.

Опять же, мы видим, как аппаратные функции управляют функциональностью программного обеспечения. К сожалению, битовые векторы неэффективны, если весь вектор не хранится в основной памяти (и периодически не записывается на устройство, содержащее файловую систему, для нужд восстановления). Хранение его в основной памяти возможно для небольших устройств, но не обязательно для больших. Диску объёмом 1,3 ГБ с блоками по 512 байт потребуется битовая карта размером более 332 КБ для отслеживания свободных блоков, хотя кластеризация блоков в группы по четыре уменьшает это число примерно до 83 КБ на диск. Диску объёмом 1 ТБ с блоками по 4 КБ потребуется 32 МБ (240 / 212 = 228 бит = 225 байт = 25 МБ) для хранения своей битовой карты. Учитывая, что размер диска постоянно растёт, проблема с битовыми векторами также будет продолжать обостряться.

14.5.2 Связанный список

Другой подход к управлению свободным пространством заключается в объединении всех свободных блоков, сохранении указателя на первый свободный блок в специальном месте файловой системы и его кэшировании в памяти. Этот первый блок содержит указатель на следующий свободный блок и так далее. Вспомним наш предыдущий пример (раздел 14.5.1), в котором блоки 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 и 27 были свободны, а остальные блоки были выделены. В этой ситуации мы бы сохранили указатель на блок 2 как на первый свободный блок. Блок 2 содержал бы указатель на блок 3, который указывал бы на блок 4, который указывал бы на блок 5, который указывал бы на блок 8 и так далее (рисунок 14.9). Такая схема неэффективна; для обхода списка необходимо читать каждый блок, что требует значительного времени ввода-вывода на жёстк
К счастью, просмотр свободного списка — нечастое действие. Обычно

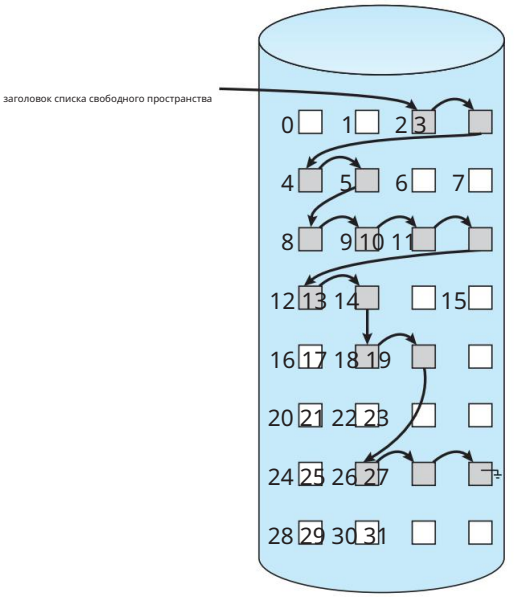


Рисунок 14.9 Связанный список свободного пространства на диске.

Операционной системе просто нужен свободный блок, чтобы выделить его файлу, поэтому используется первый блок из списка свободных блоков. Метод FAT включает учёт свободных блоков в структуру данных распределения. Отдельный метод не требуется.

14.5.3 Группировка.

Модификация метода свободных списков сохраняет адреса n свободных блоков в первом свободном блоке. Первые n 1 из этих блоков фактически свободны. Последний блок содержит адреса ещё n свободных блоков и т. д. Адреса большого количества свободных блоков теперь можно найти быстро, в отличие от ситуации, когда используется стандартный метод связанных списков.

14.5.4 Подсчет

Другой подход использует тот факт, что, как правило, несколько смежных блоков могут быть выделены или освобождены одновременно, особенно когда пространство выделяется с помощью алгоритма непрерывного выделения или посредством кластеризации.
Таким образом, вместо того, чтобы хранить список из n адресов свободных блоков, мы можем хранить адрес первого свободного блока и количество (n) свободных смежных блоков, следующих за первым. Каждая запись в списке свободного пространства тогда состоит из адреса устройства и счётчика. Хотя каждая запись занимает больше места, чем простой адрес диска, общий список получается короче, при условии, что счётчик обычно больше 1. Обратите внимание, что этот метод отслеживания свободного пространства аналогичен методу распределения блоков по экстентам. Эти записи можно хранить в сбалансированном дереве, а не в связанном списке, для эффективного поиска, вставки и удаления.

14.5.5 Космические карты

Файловая система ZFS от Oracle (применяемая в Solaris и некоторых других операционных системах) была разработана для работы с огромным количеством файлов, каталогов и даже файловых систем (в ZFS можно создавать иерархии файловых систем). В таких масштабах ввод-вывод метаданных может оказывать значительное влияние на производительность. Рассмотрим, например, что если список свободного пространства реализован в виде битовой карты, битовые карты должны изменяться как при выделении блоков, так и при их освобождении. Освобождение 1 ГБ данных на диске объёмом 1 ТБ может привести к обновлению тысяч блоков битовых карт, поскольку эти блоки данных могут быть разбросаны по всему диску. Очевидно, что структуры данных для такой системы могут быть большими и неэффективными.

Для управления свободным пространством ZFS использует комбинацию методов для управления размером структур данных и минимизации количества операций ввода-вывода, необходимых для управления этими структурами. Во-первых, ZFS создаёт метаслябы (метаблоки) для разделения пространства на устройстве на фрагменты управляемого размера. Один том может содержать сотни метаслябов. Каждому метаслябу соответствует своя карта пространства. ZFS использует алгоритм подсчёта для хранения информации о свободных блоках. Вместо записи структур подсчёта на диск, ZFS использует для их записи методы журналирования файловой системы.
Карта пространства представляет собой журнал всех действий с блоками (выделение и освобождение) в порядке времени, в формате подсчёта. Когда ZFS решает выделить или освободить пространство в метаслябе, она загружает соответствующую карту пространства в память в виде сбалансированной древовидной структуры (для высокой эффективности работы), индексированной по смещению, и воспроизводит журнал в этой структуре. Карта пространства в памяти, таким образом, является точным представлением выделенного и свободного пространства в метаслябе. ZFS также максимально уплотняет карту, объединяя смежные свободные блоки в одну запись.
Наконец, список свободного пространства обновляется на диске в рамках транзакционно-ориентированных операций ZFS. Во время фазы сбора и сортировки по-прежнему могут возникать запросы на блокировку, и ZFS удовлетворяет эти запросы из журнала. По сути, журнал плюс сбалансированное дерево составляют список свободного пространства.

14.5.6 Удаление неиспользуемых блоков. Жёстким

дискам и другим носителям, допускающим перезапись блоков для обновления, для управления свободным пространством нужен только список свободных блоков. При освобождении блоков не требуется специальной обработки. Освобождённый блок обычно сохраняет свои данные (но без указателей на него) до тех пор, пока они не будут перезаписаны при следующем выделении блока.

Устройства хранения данных, не допускающие перезапись, такие как устройства на основе флэш-памяти NVM , серьёзно страдают от применения этих же алгоритмов. Напомним из раздела 11.1.2, что такие устройства необходимо очистить, прежде чем на них снова можно будет записывать данные, причём эти операции очистки должны выполняться большими блоками (блоками, состоящими из страниц) и занимают относительно много времени по сравнению с чтением или записью.

Необходим новый механизм, позволяющий файловой системе сообщать устройству хранения данных о том, что страница свободна и может быть удалена (после того, как блок, содержащий эту страницу, полностью освободится). Этот механизм зависит от контроллера хранилища. Для накопителей, подключенных по интерфейсу ATA, это команда TRIM, а для накопителей на базе NVMe — команда unallocate . Независимо от конкретной команды контроллера, этот механизм сохраняет пространство хранилища доступным для записи. Без такой возможности устройство хранения данных заполняется и требует сборки мусора и удаления блоков, что приводит к снижению производительности ввода-вывода (так называемому «обрыву записи»).

Благодаря механизму TRIM и аналогичным возможностям этапы сбора мусора и стирания могут выполняться до того, как устройство почти заполнится, что позволяет устройству обеспечивать более стабильную производительность.

14.6 Эффективность и производительность

Теперь, когда мы обсудили различные варианты распределения блоков и управления каталогами, мы можем подробнее рассмотреть их влияние на производительность и эффективное использование хранилища. Диски, как правило, представляют собой серьёзное «узкое место» в производительности системы, поскольку являются самым медленным основным компонентом компьютера. Даже устройства NVM медленнее процессора и основной памяти, поэтому их производительность также необходимо оптимизировать. В этом разделе мы рассмотрим различные методы повышения эффективности и производительности вторичных хранилищ.

14.6.1 Эффективность

Эффективность использования пространства на устройстве хранения данных во многом зависит от используемых алгоритмов распределения и каталогизации. Например, в UNIX иноды заранее выделяются на томе. Даже на пустом диске часть пространства теряется из-за инодов.
Однако, предварительно выделяя иноды и распределяя их по всему тому, мы повышаем производительность файловой системы. Это повышение производительности достигается благодаря алгоритмам распределения и свободного пространства UNIX , которые стараются размещать блоки данных файла рядом с блоком инодов этого файла, чтобы сократить время поиска.

В качестве другого примера рассмотрим схему кластеризации, обсуждаемую в разделе 14.4, которая повышает производительность поиска и передачи файлов за счёт внутренней фрагментации. Чтобы уменьшить эту фрагментацию, BSD UNIX изменяет размер кластера по мере роста файла. Большие кластеры используются там, где они могут быть заполнены, а малые — для небольших файлов и последнего кластера файла. Эта система описана в Приложении C.

Типы данных, обычно хранящихся в каталоге (или иноде) файла, также требуют рассмотрения. Обычно регистрируется «дата последней записи», чтобы предоставить пользователю информацию и определить, требуется ли резервное копирование файла. Некоторые системы также хранят «дату последнего доступа», чтобы пользователь мог определить, когда файл был прочитан в последний раз. В результате хранения этой информации при каждом чтении файла необходимо записать данные в поле в структуре каталога.
Это означает, что блок должен быть прочитан в память, раздел изменён, а блок записан обратно на устройство, поскольку операции с вторичным хранилищем выполняются только блоками (или кластерами). Таким образом, при каждом открытии файла для чтения его FCB также должен быть прочитан и записан. Это требование может быть неэффективным для часто используемых файлов, поэтому при проектировании файловой системы необходимо взвесить его преимущества и потери производительности. Как правило, каждый элемент данных, связанный с файлом, необходимо учитывать с точки зрения его влияния на эффективность и производительность.

Рассмотрим, например, как на эффективность влияет размер указателей, используемых для доступа к данным. В большинстве операционных систем используются либо 32-битные, либо 64-битные указатели. Использование 32-битных указателей ограничивает размер файла 232 ГБ, или 4 ГБ. Использование 64-битных указателей позволяет хранить файлы очень больших размеров, но для хранения таких указателей требуется больше места. В результате методы распределения памяти и управления свободным пространством (связанные списки, индексы и т. д.) требуют больше места.

Одна из трудностей при выборе размера указателя — или, по сути, любого фиксированного размера выделения памяти в операционной системе — заключается в планировании последствий изменения технологий. Предположим, что IBM PC XT имел жёсткий диск объёмом 10 МБ и файловую систему MS-DOS FAT , которая могла поддерживать только 32 МБ. (Каждая запись FAT была 12 бит, указывающих на кластер размером 8 КБ .) По мере увеличения ёмкости дисков большие диски приходилось разбивать на разделы по 32 МБ, поскольку файловая система не могла отслеживать блоки размером более 32 МБ. По мере распространения жёстких дисков ёмкостью более 100 МБ структуры дисковых данных и алгоритмы в MS-DOS пришлось модифицировать для поддержки более крупных файловых систем. (Каждая запись FAT была расширена до 16 бит, а позже до 32 бит.) Первоначальные решения относительно файловой системы принимались из соображений эффективности; Однако с появлением MS-DOS версии 4 миллионы пользователей компьютеров столкнулись с неудобствами, когда им пришлось перейти на новую, более объёмную файловую систему. Файловая система ZFS в Solaris использует 128-битные указатели, которые теоретически не требуют расширения. (Минимальная масса устройства, способного хранить 2128 байт при использовании атомного уровня памяти, составила бы около 272 триллионов килограммов.)

В качестве другого примера рассмотрим эволюцию операционной системы Solaris. Изначально многие структуры данных имели фиксированную длину и выделялись при запуске системы. К таким структурам относятся таблица процессов и таблица открытых файлов.
Когда таблица процессов заполнялась, создавать новые процессы было невозможно. Когда таблица файлов заполнялась, файлы больше нельзя было открыть. Система переставала предоставлять услуги пользователям. Размеры таблиц можно было увеличить только перекомпиляцией ядра и перезагрузкой системы. В более поздних версиях Solaris (как и в современных ядрах Linux) почти все структуры ядра выделялись динамически, что устраняло эти искусственные ограничения производительности системы. Конечно, алгоритмы, работающие с этими таблицами, стали сложнее, а операционная система работала немного медленнее, поскольку ей приходилось динамически выделять и освобождать записи в таблицах; но это обычная плата за более общую функциональность.

14.6.2 Производительность

Даже после выбора базовых алгоритмов файловой системы производительность можно повысить несколькими способами. Как обсуждалось в главе 12, контроллеры устройств хранения данных включают локальную память для формирования встроенного кэша, объём которого достаточно велик для хранения целых дорожек или блоков одновременно. На жёстком диске после выполнения поиска дорожка считывается в кэш диска, начиная с сектора под головкой диска (что сокращает время задержки). Затем контроллер диска передаёт все запросы секторов операционной системе. После того, как блоки попадают из контроллера диска в основную память, операционная система может кэшировать их там.

В некоторых системах для буферного кэша выделяется отдельный раздел основной памяти , где блоки хранятся в предположении, что они будут использованы в ближайшее время. Другие системы кэшируют данные файлов с помощью страничного кэша. Страничный кэш использует методы виртуальной памяти для кэширования данных файлов в виде страниц, а не блоков, ориентированных на файловую систему. Кэширование данных файлов с использованием виртуальных адресов гораздо эффективнее, чем кэширование через физические дисковые блоки, поскольку доступ осуществляется через виртуальную память, а не через файловую систему. Некоторые системы, включая Solaris, Linux и Windows, используют страничное кэширование для кэширования как страниц процессов, так и данных файлов. Это называется унифицированной виртуальной памятью.

Некоторые версии UNIX и Linux предоставляют унифицированный буферный кеш. Чтобы проиллюстрировать преимущества унифицированного буферного кеширования, рассмотрим два варианта.

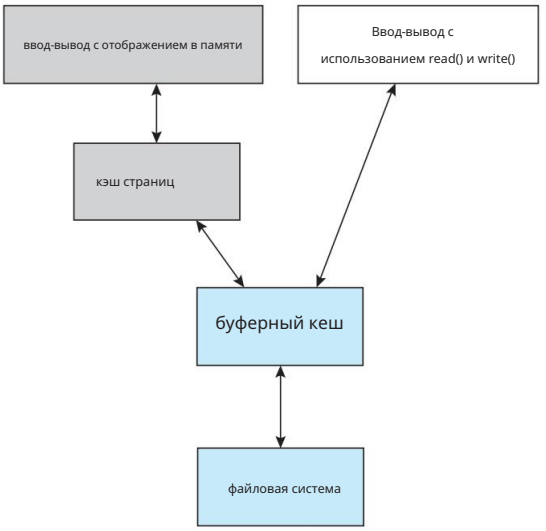


Рисунок 14.10 Ввод-вывод без единого буферного кэша.

для открытия файла и доступа к нему. Один из подходов — использование отображения памяти (раздел 13.5); второй — использование стандартных системных вызовов read() и write(). Без единого буферного кэша мы получаем ситуацию, аналогичную рис. 14.10. Здесь системные вызовы read() и write() проходят через буферный кэш.
Однако вызов отображения памяти требует использования двух кэшей — страничного кэша и буферного кэша. Отображение памяти происходит путем чтения блоков диска из файловой системы и сохранения их в буферном кэше. Поскольку система виртуальной памяти не взаимодействует с буферным кэшем, содержимое файла в буферном кэше должно быть скопировано в страничный кэш. Эта ситуация, известная как двойное кэширование, требует двойного кэширования данных файловой системы. Это не только приводит к нерациональному использованию памяти, но и к значительным затратам процессорного времени и циклов ввода-вывода из-за дополнительного перемещения данных в системной памяти. Кроме того, несоответствия между двумя кэшами могут привести к повреждению файлов. Напротив, при использовании унифицированного буферного кэша и отображение памяти, и системные вызовы read() и write() используют один и тот же страничный кэш. Это позволяет избежать двойного кэширования и позволяет системе виртуальной памяти управлять данными файловой системы. Унифицированный буферный кеш показан на рисунке 14.11.

Независимо от того, кэшируем ли мы блоки памяти или страницы (или и то, и другое), алгоритм LRU (Learning Recently Used , LRU) (Раздел 10.4.4) представляется разумным универсальным алгоритмом для замены блоков или страниц. Однако эволюция алгоритмов кэширования страниц в Solaris выявила сложность выбора алгоритма. Solaris позволяет процессам и страничному кэшу совместно использовать неиспользуемую память. Версии до Solaris 2.5.1 не делали различий между выделением страниц процессу и выделением их страничному кэшу. В результате система, выполняющая множество операций ввода-вывода, использовала большую часть доступной памяти для кэширования страниц. Из-за высокой скорости ввода-вывода сканер страниц (Раздел 10.10.3) при нехватке свободной памяти забирал страницы у процессов, а не из страничного кэша. В Solaris 2.6 и Solaris 7 опционально реализовывалась приоритетная подкачка, при которой сканер страниц отдавал приоритет обработке страниц по сравнению со страничным кэшем. Solaris 8 применял фиксированный лимит для обработки страниц и кэша страниц файловой системы, предотвращая

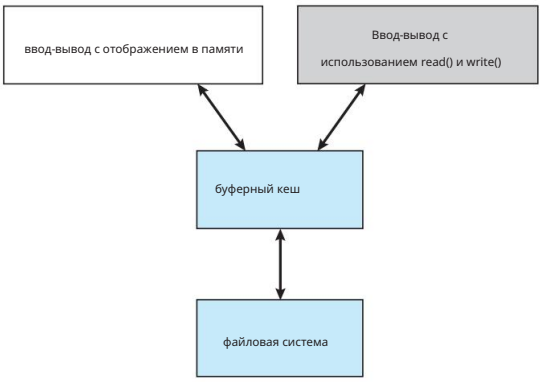


Рисунок 14.11 Ввод-вывод с использованием унифицированного буферного кеша.

не допуская вытеснения одного из них из памяти другим. В Solaris 9 и 10 алгоритмы снова были изменены для максимального использования памяти и минимизации пробуксовки.

Ещё одна проблема, которая может повлиять на производительность ввода-вывода , заключается в том, выполняется ли запись в файловую систему синхронно или асинхронно. Синхронная запись выполняется в порядке её получения подсистемой хранения, и запись не буферизуется. Таким образом, вызывающая процедура должна дождаться достижения данными диска, прежде чем сможет продолжить. При асинхронной записи данные сохраняются в кэше, и управление возвращается вызывающей программе. Большинство операций записи выполняются асинхронно. Однако запись метаданных, среди прочего, может быть синхронной. Операционные системы часто включают флаг в системный вызов open , позволяющий процессу запросить синхронную запись. Например, базы данных используют эту функцию для атомарных транзакций, чтобы гарантировать, что данные попадут в стабильное хранилище в требуемом порядке.

Некоторые системы оптимизируют кэш страниц, используя различные алгоритмы замены в зависимости от типа доступа к файлу. Файл, считываемый или записываемый последовательно, не должен заменять страницы в порядке LRU , поскольку последняя использованная страница будет использована последней, а возможно, и никогда больше. Вместо этого последовательный доступ можно оптимизировать с помощью методов, известных как «свободное чтение» (free-behind) и «упреждающее чтение» (read-ahead). «Свободное чтение» удаляет страницу из буфера сразу после запроса следующей. Предыдущие страницы, скорее всего, не будут использоваться повторно и не будут занимать место в буфере. При упреждающем чтении запрошенная страница и несколько последующих страниц считываются и кэшируются. Эти страницы, вероятно, будут запрошены после обработки текущей страницы. Извлечение этих данных с диска за один проход и их кэширование значительно экономят время. Можно подумать, что кэш дорожек на контроллере устранит необходимость в упреждающем чтении в многопрограммной системе. Однако из-за высокой задержки и накладных расходов, связанных с выполнением множества небольших передач из кэша дорожек в основную память, упреждающее чтение остаётся выгодным.

Кэш страниц, файловая система и драйверы устройств взаимодействуют между собой интересным образом. При записи небольших объёмов данных в файл страницы буферизуются в кэше, а драйвер устройства хранения сортирует свою выходную очередь в соответствии с адресом устройства. Эти два действия позволяют драйверу диска минимизировать количество обращений к головке диска. Если синхронная запись не требуется, процесс, записывающий данные на диск, просто записывает данные в кэш, а система асинхронно записывает их.

Данные на диск, когда это удобно. Пользовательский процесс видит очень быструю запись. Когда данные считываются из файла на диске, система блочного ввода-вывода выполняет некоторое опережающее чтение; однако запись гораздо более асинхронна, чем чтение. Таким образом, вывод на диск через файловую систему часто происходит быстрее, чем ввод для небольших передач, вопреки интуиции. Независимо от того, какой объем буферизации и кэширования доступен, большой, непрерывный ввод-вывод может переполнить емкость и в конечном итоге стать узким местом в производительности устройства. Рассмотрим запись большого файла фильма на жесткий диск. Если файл больше, чем страничный кэш (или часть страничного кэша, доступная процессу), то страничный кэш заполнится, и весь ввод-вывод будет выполняться на скорости диска. Современные жесткие диски читают быстрее, чем записывают, поэтому в этом случае аспекты производительности противоположны меньшей производительности ввода-вывода .

14.7 Восстановление

Файлы и каталоги хранятся как в основной памяти, так и на томе хранилища, поэтому необходимо соблюдать осторожность, чтобы сбой системы не привёл к потере данных или их несогласованности. Сбой системы может привести к несогласованности структур данных файловой системы на хранилище, таких как структуры каталогов, указатели свободных блоков и указатели свободных блоков FCB . Многие файловые системы применяют изменения к этим структурам на месте. Типичная операция, такая как создание файла, может включать множество структурных изменений в файловой системе на диске. Изменяются структуры каталогов, выделяются блоки FCB , выделяются блоки данных, а количество свободных блоков для всех этих блоков уменьшается. Эти изменения могут быть прерваны сбоем, что может привести к несогласованности структур. Например, количество свободных блоков FCB может указывать на то, что блок FCB был выделен, но структура каталога может не указывать на этот блок FCB. Усугубляет эту проблему кэширование, которое операционные системы используют для оптимизации производительности ввода-вывода . Некоторые изменения могут попадать непосредственно в хранилище, в то время как другие могут кэшироваться. Если кэшированные изменения не достигнут устройства хранения до возникновения сбоя, возможны дополнительные повреждения.

Помимо сбоев, файловую систему могут повредить ошибки в реализации файловой системы, контроллерах устройств и даже пользовательских приложениях. В файловых системах используются различные методы борьбы с повреждениями, зависящие от структур данных и алгоритмов. Далее мы рассмотрим эти проблемы.

14.7.1 Проверка целостности. Независимо от

причины повреждения, файловая система должна сначала обнаружить проблемы и исправить их. Для их обнаружения сканирование всех метаданных каждой файловой системы может подтвердить или опровергнуть целостность системы. К сожалению, такое сканирование может занять от нескольких минут до нескольких часов и должно выполняться при каждой загрузке системы. Альтернативно, файловая система может записывать свое состояние в метаданных файловой системы. В начале любого изменения метаданных устанавливается бит статуса, указывающий на то, что метаданные находятся в процессе изменения. Если все обновления метаданных завершены успешно, файловая система может сбросить этот бит. Однако, если бит статуса остаётся установленным, запускается проверка согласованности.

Проверка согласованности — системная программа, такая как fsck в UNIX — сравнивает данные в структуре каталога и другие метаданные с состоянием хранилища и пытается исправить любые обнаруженные несоответствия. Алгоритмы распределения и управления свободным пространством определяют, какие типы проблем будут обнаружены.

Может ли программа проверки обнаружить эти ошибки и оценить, насколько успешно она их исправит. Например, если используется связанное выделение памяти и существует ссылка из любого блока на следующий блок, то весь файл может быть восстановлен из блоков данных, а структура каталога может быть воссоздана заново. Напротив, потеря записи каталога в индексированной системе выделения памяти может иметь катастрофические последствия, поскольку блоки данных не имеют информации друг о друге. По этой причине некоторые файловые системы UNIX кэшируют записи каталога для чтения, но любая запись, приводящая к выделению пространства или другим изменениям метаданных, выполняется синхронно, до записи соответствующих блоков данных. Конечно, проблемы всё равно могут возникнуть, если синхронная запись прерывается сбоем. Некоторые устройства хранения данных NVM содержат батарею или суперконденсатор, обеспечивающий достаточное питание даже при отключении питания для записи данных из буферов устройства на носитель, чтобы данные не были потеряны. Но даже эти меры предосторожности не защищают от повреждения данных в результате сбоя.

14.7.2 Файловые системы с журнальной структурой

Специалисты по информатике часто обнаруживают, что алгоритмы и технологии, изначально применявшиеся в одной области, одинаково полезны и в других. Например, алгоритмы восстановления баз данных на основе журналирования. Эти алгоритмы журналирования успешно применяются для решения задач проверки согласованности. Полученные реализации известны как файловые системы , ориентированные на транзакции (или журналирование), основанные на журналировании .

Обратите внимание, что подход с проверкой согласованности, обсуждавшийся в предыдущем разделе, по сути, позволяет структурам разрушаться и восстанавливаться при восстановлении. Однако у этого подхода есть несколько проблем. Одна из них заключается в том, что несогласованность может быть неустранимой. Проверка согласованности может не восстановить структуры, что приведет к потере файлов и даже целых каталогов. Проверка согласованности может потребовать вмешательства человека для разрешения конфликтов, что неудобно, если человек отсутствует. Система может оставаться недоступной до тех пор, пока человек не даст ей указаний, как действовать дальше. Проверка согласованности также занимает системное и тактовое время. Для проверки терабайт данных могут потребоваться часы.

Решение этой проблемы заключается в применении методов восстановления на основе журналов к обновлениям метаданных файловой системы. Этот метод используется как в NTFS , так и в файловой системе Veritas и включён в последние версии UFS для Solaris. Более того, он теперь распространён во многих файловых системах, включая ext3, ext4 и ZFS.

По сути, все изменения метаданных последовательно записываются в журнал. Каждый набор операций для выполнения определенной задачи является транзакцией . После того, как изменения записаны в этот журнал, они считаются зафиксированными, и системный вызов может вернуться к пользовательскому процессу, позволяя ему продолжить выполнение. Тем временем эти записи журнала воспроизводятся в реальных структурах файловой системы. По мере внесения изменений указатель обновляется, чтобы указать, какие действия были завершены, а какие еще не завершены. Когда вся зафиксированная транзакция завершается, в журнале делается запись, указывающая на это. Файл журнала фактически представляет собой кольцевой буфер. Кольцевой буфер записывает данные в конец своего пространства, а затем продолжает запись с начала, перезаписывая старые значения по мере ее выполнения. Мы не хотим, чтобы буфер перезаписывал данные, которые еще не были сохранены, поэтому такой сценарий избегается. Журнал может находиться в отдельном разделе файловой системы или даже на отдельном устройстве хранения.

В случае сбоя системы файл журнала будет содержать ноль или более транзакций.
Все содержащиеся в нем транзакции не были завершены в файловой системе, хотя они были зафиксированы операционной системой, поэтому теперь их необходимо завершить.
Транзакции могут выполняться с указателя до завершения работы, сохраняя при этом согласованность структуры файловой системы. Единственная проблема возникает, если транзакция была прервана, то есть не была зафиксирована до сбоя системы. Любые изменения, внесённые в файловую систему такой транзакцией, должны быть отменены, что, опять же, сохраняет согласованность файловой системы.

Это восстановление — все, что нужно после сбоя, устраняя любые проблемы с проверкой согласованности.

Дополнительным преимуществом использования журналирования обновлений метаданных на диске является то, что эти обновления выполняются гораздо быстрее, чем при непосредственном применении к структурам данных на диске. Это объясняется преимуществом последовательного ввода-вывода в производительности по сравнению с произвольным вводом-выводом. Затратные синхронные случайные записи метаданных преобразуются в гораздо менее затратные синхронные последовательные записи в область журналирования файловой системы с журнальной структурой. Эти изменения, в свою очередь, воспроизводятся асинхронно посредством случайных записей в соответствующие структуры. В результате достигается значительный прирост производительности операций с метаданными, таких как создание и удаление файлов, на жестком диске .

14.7.3 Другие решения

Другая альтернатива проверке согласованности используется в файловой системе WAFL компании Network Appli-ance и файловой системе ZFS в Solaris . Эти системы никогда не перезаписывают блоки новыми данными. Вместо этого транзакция записывает все изменения данных и метаданных в новые блоки. После завершения транзакции структуры метаданных, указывающие на старые версии этих блоков, обновляются, чтобы указывать на новые блоки. Затем файловая система может удалить старые указатели и старые блоки, сделав их доступными для повторного использования. Если старые указатели и блоки сохраняются, создается снимок , представляющий собой представление файловой системы на определенный момент времени (до применения каких-либо обновлений после этого момента). Это решение не должно требовать проверки согласованности, если обновление указателей выполняется атомарно. Однако в WAFL есть средство проверки согласованности, поэтому некоторые сценарии сбоев все равно могут привести к повреждению метаданных. (Подробнее о файловой системе WAFL см. в разделе 14.8 .)

ZFS использует ещё более инновационный подход к обеспечению согласованности данных на диске. Как и WAFL, она никогда не перезаписывает блоки. Однако ZFS идёт дальше и обеспечивает контрольную сумму всех метаданных и блоков данных. Это решение (в сочетании с RAID) гарантирует постоянную корректность данных. Поэтому в ZFS нет средства проверки согласованности. (Подробнее о ZFS см. в разделе 11.8.6.)

14.7.4 Резервное копирование и восстановление

Устройства хранения данных иногда выходят из строя, и необходимо принять меры, чтобы данные, потерянные в результате такого сбоя, не были утеряны безвозвратно. Для этого можно использовать системные программы для резервного копирования данных с одного устройства хранения на другое, например, на магнитную ленту или другой вторичный накопитель. Восстановление данных после потери отдельного файла или всего устройства может заключаться в восстановлении данных из резервной копии.

Чтобы минимизировать необходимое копирование, можно использовать информацию из записи каталога каждого файла. Например, если программа резервного копирования знает, когда последний файл был

Резервное копирование файла было выполнено, и дата последней записи файла в каталог указывает на то, что файл не изменялся с этой даты, поэтому повторное копирование файла не требуется. Типичный график резервного копирования может выглядеть следующим образом:

- День 1. Копирование всех файлов из файловой системы на резервный носитель. Это называется полное резервное копирование.
- День 2. Скопируйте на другой носитель все файлы, измененные с первого дня. Это инкрементное резервное копирование.
- День 3. Скопируйте на другой носитель все файлы, измененные с дня 2.

...

- День N. Скопируйте на другой носитель все файлы, изменённые с дня N 1. Затем перейдите вернемся к первому дню.

Резервная копия нового цикла может быть записана поверх предыдущего набора или на новый набор резервных носителей.

Используя этот метод, мы можем восстановить всю файловую систему, начав восстановление с полной резервной копии и продолжая восстановлением каждой из инкрементных резервных копий.
Конечно, чем больше значение N, тем большее количество носителей необходимо прочитать для полного восстановления. Дополнительным преимуществом этого цикла резервного копирования является возможность восстановить любой файл, случайно удалённый во время цикла, извлекая его из резервной копии предыдущего дня.

Длительность цикла — это компромисс между объёмом необходимого резервного копирования и количеством дней, охватываемых восстановлением. Чтобы уменьшить количество лент, которые необходимо прочитать для восстановления, можно выполнить полное резервное копирование, а затем ежедневно создавать резервные копии всех файлов, изменившихся с момента создания полного резервного копирования. Таким образом, восстановление можно выполнить с помощью последнего инкрементного резервного копирования и полного резервного копирования, без необходимости создания других инкрементных резервных копий. Компромисс заключается в том, что каждый день будет изменяться больше файлов, поэтому каждое последующее инкрементное резервное копирование будет включать больше файлов и больше резервных носителей.

Пользователь может заметить, что определённый файл отсутствует или повреждён спустя долгое время после повреждения. Поэтому мы обычно планируем время от времени создавать полную резервную копию, которая будет храниться «вечно». Рекомендуется хранить эти постоянные резервные копии подальше от обычных, чтобы защитить их от опасностей, таких как пожар, который уничтожит компьютер и все резервные копии. В сериале «Мистер Робот» хакеры атаковали не только основные источники банковских данных, но и их резервные копии. Наличие нескольких резервных копий может быть хорошей идеей, если ваши данные важны.

14.8 Пример: файловая система WAFL

Поскольку операции ввода -вывода вторичного хранилища оказывают столь сильное влияние на производительность системы, проектирование и реализация файловой системы требуют пристального внимания со стороны разработчиков. Некоторые файловые системы являются универсальными, то есть обеспечивают приемлемую производительность и функциональность для широкого спектра размеров и типов файлов, а также объёмов ввода-вывода . Другие оптимизированы для определённых задач, стремясь обеспечить более высокую производительность в этих областях, чем универсальные файловые системы.

Файловые системы. Примером такой оптимизации является файловая система с возможностью записи в любом месте (WAFL) от NetApp, Inc. WAFL — это мощная и элегантная файловая система, оптимизированная для случайной записи.

WAFL используется исключительно на сетевых файловых серверах NetApp и предназначен для использования в качестве распределённой файловой системы. Он может предоставлять файлы клиентам по протоколам NFS, CIFS, iSCSI, FTP и http , хотя изначально разрабатывался только для NFS и CIFS. Когда множество клиентов используют эти протоколы для взаимодействия с файловым сервером, сервер может столкнуться с очень высокой потребностью в произвольном чтении и ещё большей потребностью в произвольной записи. Протоколы NFS и CIFS кэшируют данные, полученные при чтении, поэтому запись представляет наибольшую проблему для создателей файловых серверов.

WAFL используется на файловых серверах, которые включают кэш NVRAM для записи.

Разработчики WAFL воспользовались преимуществами работы на определенной архитектуре, чтобы оптимизировать файловую систему для случайного ввода-вывода с кэшем стабильного хранения. Простота использования — один из основополагающих принципов WAFL. Создатели также включили в него новую функцию создания снимков, которая создаёт несколько копий файловой системы, доступных только для чтения, в разные моменты времени, как мы увидим далее.

Эта файловая система похожа на Berkeley Fast File System, но имеет множество модификаций. Она основана на блоках и использует иноды для описания файлов. Каждый инод содержит 16 указателей на блоки (или косвенные блоки), принадлежащие файлу, описываемому инодом. Каждая файловая система имеет корневой инод. Все метаданные хранятся в файлах.
Все иноды находятся в одном файле, карта свободных блоков — в другом, а карта свободных инодов — в третьем, как показано на рисунке 14.12. Поскольку это стандартные файлы, блоки данных не ограничены по расположению и могут размещаться где угодно. При расширении файловой системы путём добавления дисков длина файлов метаданных автоматически увеличивается файловой системой.

Таким образом, файловая система WAFL представляет собой дерево блоков, основой которого является корневой инод. Чтобы сделать снимок, WAFL создаёт копию корневого инода. Любые последующие обновления файлов или метаданных относятся к новым блокам, а не перезаписывают существующие. Новый корневой инод указывает на метаданные и данные, изменённые в результате этих записей. При этом снимок (старый корневой инод) по-прежнему указывает на старые блоки, которые не были обновлены. Таким образом, он предоставляет доступ к файловой системе в том виде, в каком она была на момент создания снимка, и занимает при этом совсем немного места. По сути, дополнительное пространство, занимаемое снимком, состоит только из блоков, изменённых с момента создания снимка.

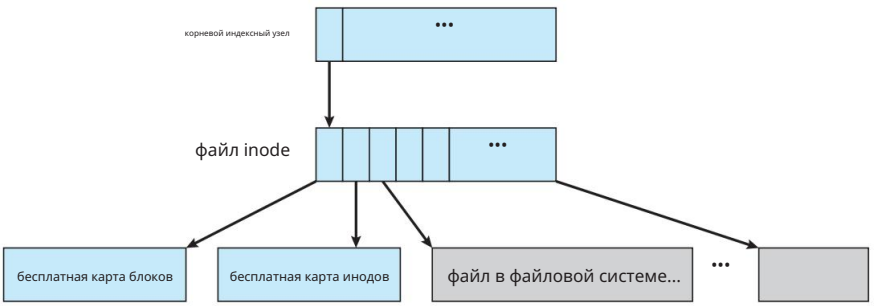
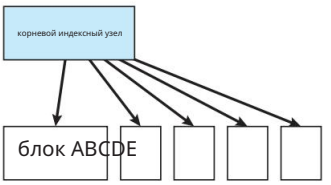


Рисунок 14.12 Структура файла WAFL.

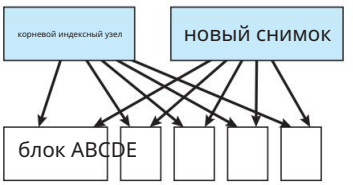
Важное отличие от более стандартных файловых систем заключается в том, что карта свободных блоков содержит более одного бита на блок. Это битовая карта, в которой бит установлен для каждого снимка, использующего этот блок. После удаления всех снимков, использовавших этот блок, битовая карта этого блока заполняется нулями, и блок может быть использован повторно. Используемые блоки никогда не перезаписываются, поэтому запись выполняется очень быстро, поскольку запись может выполняться в свободный блок, ближайший к текущему положению заголовка. В WAFL также реализовано множество других оптимизаций производительности.

Одновременно может существовать несколько снимков, например, каждый час суток и каждый день месяца. Пользователь, имеющий доступ к этим снимкам, может получить доступ к файлам в том состоянии, в котором они были на момент создания снимка. Функция снимков также полезна для резервного копирования, тестирования, управления версиями и т. д. Функция снимков WAFL очень эффективна, поскольку не требует даже копирования при записи каждого блока данных перед его изменением. Другие файловые системы предоставляют снимки, но зачастую менее эффективны. Снимки WAFL показаны на рисунке 14.13.

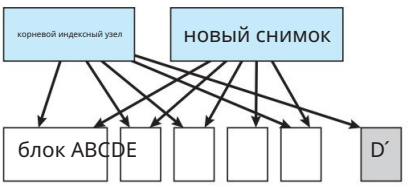
В новых версиях WAFL теперь можно создавать снимки состояния с возможностью чтения и записи, известные как клоны. Клоны также эффективны, используя те же методы, что и снимки состояния. В этом случае снимок состояния файловой системы, доступный только для чтения, фиксирует состояние файловой системы, а клон ссылается на этот снимок. Все записи в клон сохраняются в новых блоках, а указатели клона обновляются, чтобы ссылаться на новые блоки. Исходный снимок не был изменен и по-прежнему позволяет увидеть файловую систему как есть.



(а) Перед снимком.



(б) После снимка, до изменения каких-либо блоков.



(с) После того, как блок D изменился на D´.

Рисунок 14.13 Снимки в WAFL.

Файловая система Apple

В 2017 году компания Apple, Inc., выпустила новую файловую систему на замену своей 30-летней HFS+ . HFS+ была расширена для добавления множества новых функций, но, как обычно, это привело к увеличению сложности, увеличению количества строк кода и затруднению добавления новых функций. Разработка с нуля, с чистого листа, позволяет проектировать с использованием современных технологий и методологий, обеспечивая точный набор необходимых функций.

Файловая система Apple (APFS) — хороший пример такой архитектуры. Её цель — работать на всех современных устройствах Apple, от Apple Watch и iPhone до компьютеров Mac. Создание файловой системы, работающей в watchOS, I/O, tvOS и macOS , безусловно, представляет собой непростую задачу. APFS обладает богатым набором функций, включая 64-битные указатели, клонирование файлов и каталогов, моментальные снимки, совместное использование пространства, быстрое изменение размера каталогов, атомарные примитивы безопасного сохранения, копирование при записи, шифрование (с одним и несколькими ключами) и объединение операций ввода-вывода . Она поддерживает как NVM, так и HDD- хранилища.

Большинство этих функций мы уже обсудили, но есть несколько новых концепций, которые стоит изучить. Совместное использование пространства — это функция, подобная ZFS, в которой хранилище доступно в виде одного или нескольких больших свободных пространств (контейнеров), из которых файловые системы могут выделять ресурсы (что позволяет томам, отформатированным в APFS, увеличиваться и уменьшаться). Быстрое изменение размера каталога обеспечивает быстрый расчет и обновление используемого пространства. Атомарное безопасное сохранение — это примитив (доступный через API, а не через команды файловой системы), который выполняет переименование файлов, групп файлов и каталогов как отдельные атомарные операции. Объединение ввода -вывода — это оптимизация для устройств NVM , при которой несколько небольших операций записи объединяются в одну большую для оптимизации производительности записи.

Apple решила не внедрять RAID как часть новой APFS, положившись вместо этого на существующий механизм томов Apple RAID для программного RAID.
APFS также совместим с HFS+, что позволяет легко конвертировать существующие развертывания.

Это было до обновления клона. Клоны также могут быть использованы для замены исходной файловой системы; это подразумевает удаление всех старых указателей и любых связанных с ними старых блоков. Клоны полезны для тестирования и обновлений, поскольку исходная версия остаётся нетронутой, а клон удаляется после завершения теста или в случае сбоя обновления.

Другая функция, которая естественным образом вытекает из реализации файловой системы WAFL, — это репликация, дублирование и синхронизация набора данных по сети с другой системой. Сначала снимок файловой системы WAFL дублируется на другой системе. Когда на исходной системе делается еще один снимок, относительно легко обновить удаленную систему, просто отправив все блоки, содержащиеся в новом снимке. Эти блоки — те, которые изменились между моментами создания двух снимков. Удаленная система добавляет эти блоки в файловую систему и обновляет свои указатели, и новая система затем является дубликатом исходной системы на момент создания второго снимка. Повторение этого процесса поддерживает удаленную систему как почти актуальную копию первой системы. Такая репликация используется для аварийного восстановления. В случае разрушения первой системы большая часть ее данных доступна для использования на удаленной системе.

Наконец, обратите внимание, что файловая система ZFS поддерживает столь же эффективные моментальные снимки, клоны и репликацию, и эти функции становятся все более распространенными в различных файловых системах с течением времени.

14.9 Резюме

- Большинство файловых систем располагаются на вторичных носителях, предназначенных для постоянного хранения больших объёмов данных. Наиболее распространённым вторичным носителем является диск, но использование устройств NVM растёт.
- Устройства хранения данных сегментируются на разделы для управления использованием носителей и создания нескольких, возможно, различных, файловых систем на одном устройстве. Эти файловые системы монтируются в логическую архитектуру файловой системы, чтобы быть доступными для использования.
- Файловые системы часто реализуются в виде многоуровневой или модульной структуры. Нижние уровни отвечают за физические свойства устройств хранения данных и взаимодействие с ними. Верхние уровни отвечают за символические имена файлов и логические свойства файлов.
- Различным файлам в файловой системе можно выделить место на устройстве хранения тремя способами: посредством непрерывного, связанного или индексированного распределения. Непрерывное выделение может страдать от внешней фрагментации. Прямой доступ при связанном выделении очень неэффективен. Индексированное выделение может потребовать значительных накладных расходов на индексный блок. Эти алгоритмы можно оптимизировать различными способами. Непрерывное пространство можно расширить с помощью экстентов для повышения гибкости и снижения внешней фрагментации.

  Индексированное распределение может осуществляться кластерами из нескольких блоков для повышения пропускной способности и сокращения количества необходимых записей индекса. Индексирование в больших кластерах аналогично непрерывному распределению с экстентами. • Методы распределения

свободного пространства также влияют на эффективность использования дискового пространства, производительность файловой системы и надежность вторичного хранилища. Используемые методы включают битовые векторы и связанные списки. Оптимизации включают группировку, подсчёт и FAT , которая размещает связанный список в одной непрерывной области.

- Процедуры управления каталогами должны учитывать эффективность, производительность и надежность. Хеш-таблица — распространённый метод, поскольку он быстрый и эффективный. К сожалению, повреждение таблицы или сбой системы могут привести к несоответствию между информацией в каталоге и содержимым диска.
- Средство проверки согласованности может использоваться для восстановления поврежденных структур файловой системы. Инструменты резервного копирования операционной системы позволяют копировать данные на магнитную ленту или другие устройства хранения, что позволяет пользователю восстановить данные в случае потери данных или даже полного отказа устройства из-за аппаратного сбоя, ошибки операционной системы и
  или ошибка пользователя.
- Поскольку файловые системы играют основополагающую роль в работе системы, их производительность и надёжность имеют решающее значение. Такие методы, как структуры журналов и кэширование, помогают повысить производительность, а структуры журналов и RAID повышают надёжность. Файловая система WAFL — пример оптимизации производительности для соответствия определённой нагрузке ввода-вывода .

Практические упражнения

14.1. Рассмотрим файл, состоящий из 100 блоков. Предположим, что блок управления файлом (и индексный блок в случае индексированного выделения) уже находится в памяти. Рассчитайте количество операций дискового ввода-вывода , необходимых для стратегий непрерывного, связанного и индексированного (одноуровневого) выделения, если для одного блока выполняются следующие условия. В случае непрерывного выделения предположим, что в начале нет места для роста, но есть в конце. Также предположим, что информация о добавляемом блоке хранится в памяти.

а. Блок добавляется в начало.

б. Блок добавляется в середину.

в. Блок добавляется в конец.

г. Блок снимается с начала.

е. Блок удаляется из середины.

е. Блок удаляется с конца.

14.2 Почему битовая карта для распределения файлов должна храниться на запоминающем устройстве, а не в основной памяти?

14.3 Рассмотрим систему, поддерживающую стратегии непрерывного, связанного и индексированного размещения. Какие критерии следует использовать при выборе оптимальной стратегии для конкретного файла?

14.4 Одна из проблем непрерывного выделения памяти заключается в том, что пользователь должен заранее выделить достаточно места для каждого файла. Если файл превышает выделенное ему пространство, необходимо предпринять специальные действия. Одним из решений этой проблемы является определение структуры файла, состоящей из начальной непрерывной области заданного размера. Если эта область заполнена, операционная система автоматически определяет область переполнения, связанную с начальной непрерывной областью. Если область переполнения заполнена, выделяется другая область переполнения. Сравните эту реализацию файла со стандартными непрерывными и связанными реализациями.

14.5 Как кэши помогают повысить производительность? Почему системы не используют больше кэшей или не увеличивают их объём, если они так полезны?

14.6 Почему пользователю выгодно, чтобы операционная система динамически выделяла память для внутренних таблиц? Какие штрафы это наносит операционной системе?

Дополнительное чтение

Внутреннее устройство системы BSD UNIX подробно описано в [McKusick et al. (2015)]. Подробную информацию о файловых системах для Linux можно найти в [Love (2010)]. Файловая система Google описана в [Ghemawat et al. (2003)]. FUSE можно найти по адресу http://fuse.sourceforge.net.

Журнально-структурированная организация файлов, повышающая производительность и согласованность, обсуждается в работах [Rosenblum and Ousterhout (1991)], [Seltzer et al. (1993)] и [Seltzer et al. (1995)]. Журнально-структурированная организация сетевых файловых систем предлагается в работах [Hartman and Ousterhout (1995)] и [Thekkath et al. (1997)].

Исходный код ZFS для космических карт можно найти по адресу http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/ fs/zfs/space map.c.

Документацию ZFS можно найти по адресу http://www.opensolaris.org/os/community/ZFS/docs .

Файловая система NTFS описана в [Solomon (1998)], файловая система Ext3, используемая в Linux, описана в [Mauerer (2008)], а файловая система WAFL рассмотрена в [Hitz et al. (1995)].

Библиография

[Гемават и др. (2003)] С. Гемават, Х. Гобиофф и С.-Т. Леунг, «Файловая система Google», Труды симпозиума ACM по принципам операционных систем (2003).

[Хартман и Оустерхаут (1995)] Дж. Х. Хартман и Дж. К. Оустерхаут, «Сетевая файловая система с полосками зебры», ACM Transactions on Computer Systems, том 13, номер 3 (1995), страницы 274–310.

[Хитц и др. (1995)] Д. Хитц, Дж. Лау и М. Малкольм, «Проектирование файловой системы для «NFS File Server Appliance», Технический отчет, NetApp (1995).

[Любовь (2010)] Р. Лав, Разработка ядра Linux, Третье издание, для разработчиков Библиотека (2010).

[Мауэрер (2008)] В. Мауэрер, Профессиональная архитектура ядра Linux, Джон Уайли и сыновья (2008).

[МакКусик и др. (2015)] МК МакКусик, GV Neville-Neil и RNM Wat-son, Проектирование и реализация операционной системы FreeBSD UNIX – второе издание, Pearson (2015).

[Розенблюм и Оустерхаут (1991)] М. Розенблюм и Дж. К. Оустерхаут, «Проектирование и реализация файловой системы с журнальной структурой», Труды симпозиума ACM по принципам операционных систем (1991), страницы 1–15.

[Зельцер и др. (1993)] М.И. Сельцер, К. Бостик, М.К. МакКусик и К. Стаелин, «Реализация журналируемой файловой системы для UNIX», USENIX Winter (1993), страницы 307–326.

[Зельцер и др. (1995)] М.И. Сельцер, К.А. Смит, Х. Балакришнан, Дж. Чанг, С. Макмейнс и В.Н. Падманабхан, «Ведение журнала файловой системы против кластеризации: сравнение производительности», USENIX Winter (1995), страницы 249–264.

[Соломон (1998)] Д.А. Соломон, «Внутри Windows NT», второе издание, Microsoft Пресс (1998).

[Теккат и др. (1997)] CA Thekkath, T. Mann и EK Lee, «Frangipani: масштабируемая распределенная файловая система», Симпозиум по принципам операционных систем (1997), страницы 224–237.

Глава 14 Упражнения

14.7 Рассмотрим файловую систему, использующую модифицированную схему непрерывного распределения с поддержкой экстентов. Файл представляет собой набор экстентов, каждый из которых соответствует непрерывному набору блоков. Ключевым вопросом в таких системах является степень изменчивости размера экстентов.
Каковы преимущества и недостатки следующих схем?

а. Все экстенты имеют одинаковый размер, и этот размер предопределен. б. Экстенты могут быть любого размера и распределяются динамически. в. Экстенты могут иметь несколько фиксированных размеров, и эти размеры предопределены.
добыто.

14.8 Сравните производительность трех методов выделения дисковых блоков (непрерывных, связанных и индексированных) для последовательного и случайного доступа к файлам.

14.9 Каковы преимущества варианта связанного распределения, который использует FAT для объединения блоков файла?

14.10 Рассмотрим систему, в которой свободное пространство хранится в списке свободного пространства.

а. Предположим, что указатель на список свободного пространства утерян. Может ли система восстановить список свободного пространства? Объясните свой ответ.

б. Рассмотрим файловую систему, похожую на ту, что используется в UNIX с индексированным распределением. Сколько операций дискового ввода-вывода может потребоваться для чтения содержимого небольшого локального файла в каталоге /a/b/c? Предположим, что ни один из блоков диска в данный момент не кэшируется.

в) Предложите схему, которая гарантирует, что указатель никогда не будет потерян в результате сбоя памяти.

14.11 Некоторые файловые системы позволяют выделять дисковое пространство с разной степенью детализации. Например, файловая система может выделить 4 КБ дискового пространства как один блок размером 4 КБ или как восемь блоков по 512 байт. Как можно использовать эту гибкость для повышения производительности? Какие изменения необходимо внести в схему управления свободным пространством для поддержки этой функции?

14.12 Обсудите, как оптимизация производительности файловых систем может привести к трудностям в поддержании согласованности систем в случае сбоев компьютера.

14.13 Обсудите преимущества и недостатки поддержки ссылок на файлы, которые пересекают точки монтирования (то есть ссылка на файл ссылается на файл, который хранится в другом томе).

14.14 Рассмотрим файловую систему на диске, где логические и физические блоки имеют размер 512 байт. Предположим, что информация о каждом файле уже находится в памяти. Для каждой из трёх стратегий распределения (непрерывной, связанной и индексированной) ответьте на следующие вопросы:

а. Как в этой системе осуществляется сопоставление логических и физических адресов? (Для индексированного распределения предположим, что длина файла всегда меньше 512 блоков.)

б) Если мы сейчас находимся в логическом блоке 10 (последним блоком, к которому был получен доступ, был блок 10) и хотим получить доступ к логическому блоку 4, сколько физических блоков необходимо прочитать с диска?

14.15 Рассмотрим файловую систему, использующую иноды для представления файлов. Размер дисковых блоков составляет 8 КБ , а указатель на дисковый блок занимает 4 байта. В этой файловой системе имеется 12 прямых дисковых блоков, а также одинарные, двойные и тройные косвенные дисковые блоки. Каков максимальный размер файла, который может храниться в этой файловой системе?

14.16 Фрагментацию на устройстве хранения данных можно устранить с помощью уплотнения. У обычных дисковых устройств нет регистров перемещения или базовых регистров (например, тех, которые используются при уплотнении памяти), так как же можно перемещать файлы? Приведите три причины, по которым часто избегают уплотнения и перемещения файлов.

14.17 Объясните, почему регистрация обновлений метаданных обеспечивает восстановление файловой системы. tem после сбоя файловой системы.

14.18 Рассмотрим следующую схему резервного копирования: • День

1. Копируем на резервный носитель все файлы с диска. • День 2. Копируем на другой носитель все файлы, измененные с первого дня. • День 3. Копируем на другой носитель все файлы, измененные с первого дня.

Это отличается от расписания, приведенного в разделе 14.7.4, тем, что все последующие резервные копии копируют все файлы, измененные с момента первого полного резервного копирования. Каковы преимущества этой системы по сравнению с системой, описанной в разделе 14.7.4? В чём недостатки? Упрощаются или усложняются операции восстановления? Объясните свой ответ.

14.19 Обсудите преимущества и недостатки связывания с удаленными файловыми системами (хранящимися на файловых серверах) набора семантик отказов, отличных от тех, которые связаны с локальными файловыми системами.

14.20 Каковы последствия поддержки семантики согласованности UNIX? для совместного доступа к файлам, хранящимся в удаленных файловых системах?

15 ГЛАВА

Файловая система Внутренности

Как мы видели в главе 13, файловая система обеспечивает механизм онлайн-хранения и доступа к содержимому файлов, включая данные и программы. Эта глава посвящена, главным образом, внутренней структуре и функционированию файловых систем.
Мы подробно рассмотрим способы структурирования использования файлов, выделения пространства для хранения, восстановления освобожденного пространства, отслеживания местоположения данных и сопряжения других частей операционной системы с вторичным хранилищем.

ЦЕЛИ ГЛАВЫ

- Подробно изучить файловые системы и их реализацию.
- Изучить загрузку и общий доступ к файлам.
- Описать удаленные файловые системы на примере NFS .

15.1 Файловые системы

Конечно, ни один компьютер общего назначения не хранит только один файл. Обычно в компьютере хранятся тысячи, миллионы, а то и миллиарды файлов. Файлы хранятся на устройствах хранения данных с произвольным доступом, включая жёсткие диски, оптические диски и энергонезависимые устройства памяти.

Как вы видели в предыдущих главах, компьютерная система общего назначения может иметь несколько устройств хранения данных, которые можно разделить на разделы, содержащие тома, которые, в свою очередь, содержат файловые системы. В зависимости от менеджера томов, один том также может охватывать несколько разделов.
На рисунке 15.1 показана типичная организация файловой системы.

Компьютерные системы также могут иметь разное количество файловых систем, и сами файловые системы могут быть разных типов. Например, типичная система Solaris может иметь десятки файловых систем дюжины различных типов, как показано в списке файловых систем на рисунке 15.2.

В этой книге мы рассматриваем только файловые системы общего назначения. Однако стоит отметить, что существует множество специализированных файловых систем. Рассмотрим типы файловых систем из упомянутого выше примера Solaris:

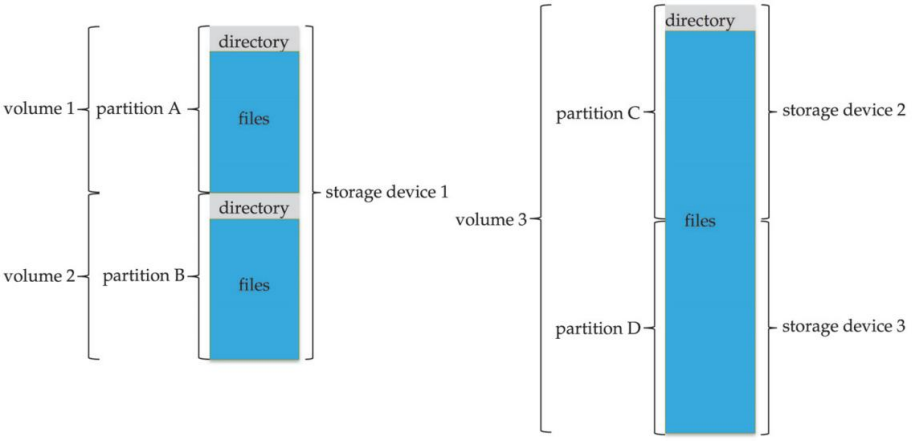


Рисунок 15.1 Типичная организация устройства хранения данных.

- tmpfs — «временная» файловая система, которая создается в энергозависимой основной памяти и ее содержимое стирается в случае перезагрузки или сбоя системы
- objfs — «виртуальная» файловая система (по сути, интерфейс к ядру, который выглядит как файловая система), которая предоставляет отладчикам доступ к символам ядра. • ctfs — виртуальная файловая система,

которая поддерживает информацию о «контрактах» для управления тем, какие процессы запускаются при загрузке системы и должны продолжать работать во время работы.

- lofs — файловая система с обратной связью, которая позволяет получать доступ к одной файловой системе вместо другой
- procfs — виртуальная файловая система, которая представляет информацию обо всех процессах в виде файловая система
- ufs, zfs — файловые системы общего назначения

Таким образом, файловые системы компьютеров могут быть обширными. Даже внутри файловой системы полезно разделять файлы на группы и управлять ими. Такая организация предполагает использование каталогов (см. раздел 14.3).

15.2 Монтирование файловой системы

Подобно тому, как файл необходимо открыть перед использованием, файловая система должна быть смонтирована, прежде чем станет доступна процессам в системе. В частности, структура каталогов может состоять из нескольких томов, содержащих файловые системы, которые необходимо смонтировать, чтобы сделать их доступными в пространстве имён файловой системы.

Процедура монтирования проста. Операционной системе передаются имя устройства и точка монтирования — место в файловой структуре, к которому будет подключена файловая система. Некоторые операционные системы требуют указания типа файловой системы, в то время как другие проверяют структуру устройства.

	уфс
/ /devices /	devfs
dev /	дев
system/contract /proc /	ctfs
etc/	процесс
mnttab /etc/svc/	mntfs
volatile /system/object /	tmpfs
lib/libc.so.1 /dev/fd /	objfs
var /tmp /var/run /	лофс
opt /zpbge /	фд
zpbge/	уфс
backup /	tmpfs
export/home /	tmpfs
var/	уфс
mail /var/	зфс
spool/mqueue /zpbg /	зфс
zpbg/zones	зфс
	зфс
	зфс
	зфс
	зфс

Рисунок 15.2 Файловые системы Solaris.

и определить тип файловой системы. Обычно точка монтирования — это пустая
Например, в системе UNIX файловая система, содержащая домашний каталог пользователя .
каталоги могут быть смонтированы как /home; тогда для доступа к структуре каталогов
В этой файловой системе мы могли бы предварять имена каталогов /home, например
в /home/jane. Монтирование этой файловой системы в /users приведет к
имя пути /users/jane, которое мы могли бы использовать для доступа к тому же каталогу.

Затем операционная система проверяет, что устройство содержит действительный файл.
Система делает это, запрашивая у драйвера устройства доступ к каталогу устройства.
и проверка того, что каталог имеет ожидаемый формат. Наконец, операционная
система отмечает в своей структуре каталогов, что файловая система смонтирована в
указанной точки монтирования. Эта схема позволяет операционной системе перемещаться
его структура каталогов, переключение между файловыми системами и даже файловые системы
различные типы, по мере необходимости.

Для иллюстрации монтирования файла рассмотрим файловую систему, изображенную на рисунке 15.3.
где треугольники представляют поддеревья интересующих нас каталогов. Рисунок
На рисунке 15.3(a) показана существующая файловая система, а на рисунке 15.3(b) показана несмонтированная файловая система.
Том, находящийся на /device/dsk. На этом этапе доступны только файлы на существующем
Доступ к файловой системе возможен. На рисунке 15.4 показаны результаты монтирования
Том, расположенный в /device/dsk над /users. Если том размонтирован,
файловая система восстанавливается до ситуации, показанной на рисунке 15.3.

Системы используют семантику для уточнения функциональности. Например, система
может запретить монтирование каталога, содержащего файлы; или может сделать

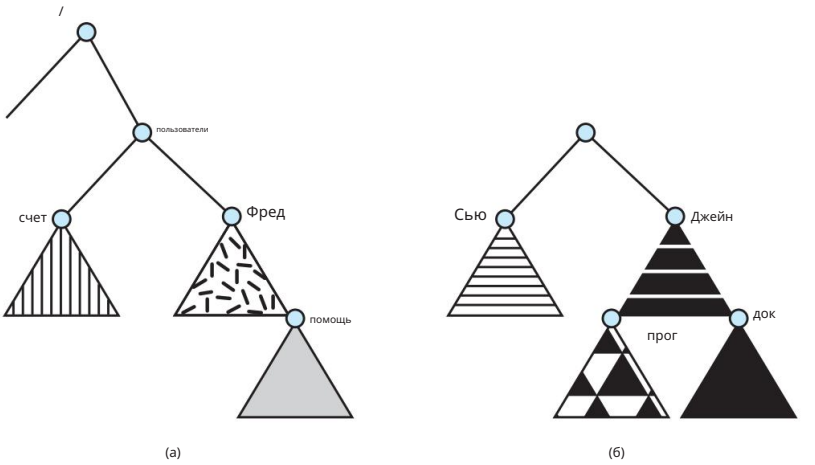


Рисунок 15.3 Файловая система. (а) Существующая система. (b) Несмонтированный том.

смонтированную файловую систему, доступную в этом каталоге, и скрывает существующие файлы в этом каталоге до тех пор, пока файловая система не будет размонтирована, что прекратит её использование и позволит получить доступ к исходным файлам в этом каталоге. Другой пример: система может разрешать многократное монтирование одной и той же файловой системы в разных точках монтирования или разрешать только одно монтирование для каждой файловой системы.

Рассмотрим действия операционной системы macOS. При первом обращении к диску (при загрузке или во время работы) операционная система macOS ищет файловую систему на устройстве.

Если файловая система найдена, она автоматически монтируется в каталог /Volumes , добавляя значок папки с именем файловой системы (сохранённым в каталоге устройства). Пользователь может щёлкнуть по этому значку и отобразить смонтированную файловую систему.

Семейство операционных систем Microsoft Windows поддерживает расширенную двухуровневую структуру каталогов, в которой устройствам и томам назначаются буквы дисков. Каждый том имеет общую графовую структуру каталогов, связанную с

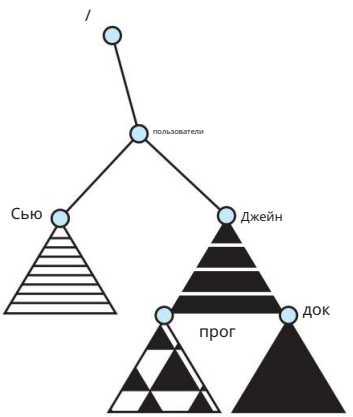


Рисунок 15.4 Том смонтирован в /users.

с буквой диска. Путь к конкретному файлу имеет вид : буква_диска : путь к файлу. Более поздние версии Windows позволяют монтировать файловую систему в любом месте дерева каталогов, как это делается в UNIX .
Операционные системы Windows автоматически обнаруживают все устройства и монтируют все найденные файловые системы при загрузке. В некоторых системах, например, UNIX, команды монтирования задаются явно. Файл конфигурации системы содержит список устройств и точек монтирования для автоматического монтирования при загрузке, но некоторые монтирования можно выполнить вручную.

Вопросы, касающиеся монтирования файловой системы, более подробно обсуждаются в разделе 15.3 и в разделе C.7.5.

15.3 Перегородки и монтаж

Структура диска может иметь множество вариантов в зависимости от операционной системы и программного обеспечения для управления томами. Диск может быть разделён на несколько разделов, а том может охватывать несколько разделов на нескольких дисках. Первая структура обсуждается здесь, а вторая, которую правильнее считать разновидностью RAID, рассматривается в разделе 11.8.

Каждый раздел может быть либо «сырым», то есть без файловой системы, либо «готовым», то есть с файловой системой. Сырой диск используется там, где файловая система не подходит.

Например, пространство подкачки UNIX может использовать раздел RAW, поскольку оно использует собственный формат на диске и не использует файловую систему. Аналогично, некоторые базы данных используют RAW-диск и форматируют данные в соответствии со своими потребностями.

RAW-диск также может содержать информацию, необходимую для дисковых RAID- систем, например, битовые карты, указывающие, какие блоки зеркалируются, а какие были изменены и требуют зеркалирования. Аналогично, RAW-диск может содержать мини-базу данных с информацией о конфигурации RAID , например, о том, какие диски входят в каждый RAID- набор. Использование RAW-диска обсуждается в разделе 11.5.1.

Если раздел содержит загружаемую файловую систему, то есть с правильно установленной и настроенной операционной системой, то разделу также необходима загрузочная информация, как описано в разделе 11.5.2. Эта информация имеет свой собственный формат, поскольку во время загрузки система не загружает код файловой системы и, следовательно, не может интерпретировать формат файловой системы. Загрузочная информация обычно представляет собой последовательную последовательность блоков, загружаемых в память в виде образа. Выполнение образа начинается с предопределенного места, например, с первого байта. Этот образ, загрузчик начальной загрузки, в свою очередь, имеет достаточно информации о структуре файловой системы, чтобы найти и загрузить ядро, а затем начать его выполнение.

Загрузчик может содержать не только инструкции по загрузке конкретной операционной системы. Например, многие системы поддерживают двойную загрузку, что позволяет устанавливать несколько операционных систем на одной системе. Как система определяет, какую из них загружать? Загрузчик, распознающий несколько файловых систем и несколько операционных систем, может занимать загрузочное пространство.
После загрузки он может загрузить одну из операционных систем, доступных на диске.
Диск может содержать несколько разделов, каждый из которых содержит файловую систему разного типа и операционную систему. Обратите внимание: если загрузчик не распознаёт определённый формат файловой системы, операционная система, хранящаяся в этой файловой системе, не может быть загружена. Это одна из причин, по которой только некоторые файловые системы поддерживаются в качестве корневых для любой операционной системы.

Корневой раздел , выбранный загрузчиком и содержащий ядро операционной системы, а иногда и другие системные файлы, монтируется во время загрузки. Другие тома могут монтироваться автоматически при загрузке или вручную позже, в зависимости от операционной системы. В рамках успешной операции монтирования операционная система проверяет, содержит ли устройство допустимую файловую систему. Для этого она запрашивает у драйвера устройства чтение каталога устройства и проверяет, имеет ли каталог ожидаемый формат. Если формат недопустим, необходимо проверить целостность раздела и, возможно, исправить её, как с участием пользователя, так и без него. Наконец, операционная система отмечает в своей таблице монтирования в памяти, что смонтирована файловая система, а также её тип. Детали этой функции зависят от операционной системы.

Системы Microsoft Windows монтируют каждый том в отдельное пространство имён, обозначенное буквой и двоеточием, как уже упоминалось ранее. Например, чтобы указать, что файловая система смонтирована в F:, операционная система помещает указатель на файловую систему в поле структуры устройства, соответствующее F:. Когда процесс указывает букву драйвера, операционная система находит соответствующий указатель файловой системы и просматривает структуру каталогов на этом устройстве, чтобы найти указанный файл или каталог. Более поздние версии Windows могут монтировать файловую систему в любую точку существующей структуры каталогов.

В UNIX файловые системы можно монтировать в любой каталог. Монтирование осуществляется путём установки флага в копии inode этого каталога в оперативной памяти.
Флаг указывает, что каталог является точкой монтирования. Поле указывает на запись в таблице монтирования, указывающую, какое устройство смонтировано в этом каталоге. Запись в таблице монтирования содержит указатель на суперблок файловой системы на этом устройстве. Эта схема позволяет операционной системе перемещаться по структуре каталогов, легко переключаясь между файловыми системами различных типов.

15.4 Обмен файлами

Возможность обмена файлами крайне важна для пользователей, стремящихся к совместной работе и сокращению усилий, необходимых для достижения вычислительных целей. Поэтому ориентированные на пользователя операционные системы должны учитывать необходимость обмена файлами, несмотря на присущие этому сложности.

В этом разделе мы рассмотрим дополнительные аспекты общего доступа к файлам. Начнём с обсуждения общих проблем, возникающих при совместном использовании файлов несколькими пользователями. После того, как нескольким пользователям разрешено обмениваться файлами, возникает проблема распространения этого доступа на несколько файловых систем, включая удалённые; мы также обсудим эту проблему. Наконец, мы рассмотрим, что делать с конфликтующими действиями, возникающими над общими файлами. Например, если несколько пользователей записывают данные в файл, следует ли разрешить все операции записи или операционная система должна защищать действия пользователей друг от друга?

15.4.1 Несколько пользователей

Когда операционная система обслуживает нескольких пользователей, вопросы совместного доступа к файлам, их именования и защиты становятся первостепенными. Учитывая структуру каталогов, позволяющую пользователям совместно использовать файлы, система должна выступать посредником в этом процессе. Система может либо разрешить пользователю доступ к файлам других пользователей,

по умолчанию или требовать, чтобы пользователь предоставил доступ к файлам. Эти вопросы управления доступом и защиты рассматриваются в разделе 13.4.

Для реализации общего доступа и защиты система должна поддерживать больше атрибутов файлов и каталогов, чем требуется в однопользовательской системе. Хотя для удовлетворения этого требования было использовано множество подходов, большинство систем в итоге перешли на концепции владельца (или пользователя) и группы файла (или каталога). Владелец — это пользователь, который может изменять атрибуты и предоставлять доступ, а также обладает наибольшим контролем над файлом. Атрибут «группа» определяет подмножество пользователей, которые могут совместно использовать доступ к файлу. Например, владелец файла в системе UNIX может выполнять все операции над файлом, в то время как члены группы файла могут выполнять один подмножество этих операций, а все остальные пользователи могут выполнять другой подмножество операций. Владелец файла определяет, какие именно операции могут выполнять члены группы и другие пользователи.

Идентификаторы владельца и группы данного файла (или каталога) хранятся вместе с другими атрибутами файла. Когда пользователь запрашивает операцию с файлом, идентификатор пользователя можно сравнить с атрибутом владельца, чтобы определить, является ли запрашивающий пользователь владельцем файла. Аналогичным образом можно сравнить идентификаторы групп . Результат показывает, какие разрешения применимы. Затем система применяет эти разрешения к запрошенной операции, разрешая или запрещая её.

Во многих системах имеется несколько локальных файловых систем, включая тома одного диска или несколько томов на нескольких подключенных дисках. В этих случаях проверка идентификаторов и сопоставление прав доступа выполняются просто после монтирования файловых систем. Но представьте себе внешний диск, который можно перемещать между системами. Что делать, если идентификаторы в системах различаются? Необходимо убедиться, что идентификаторы совпадают между системами при перемещении устройств между ними, или что при таком перемещении сбрасывается владелец файла. (Например, можно создать новый идентификатор пользователя и назначить этот идентификатор всем файлам на съемном диске , чтобы гарантировать, что ни один файл не будет случайно доступен существующим пользователям.)

15.5 Виртуальные файловые системы

Как мы уже видели, современные операционные системы должны одновременно поддерживать несколько типов файловых систем. Но как операционная система позволяет интегрировать несколько типов файловых систем в структуру каталогов? И как пользователи могут легко переключаться между типами файловых систем при навигации по файловому пространству? Теперь мы обсудим некоторые детали реализации.

Очевидный, но не самый оптимальный метод реализации нескольких типов файловых систем — это написание процедур для работы с каталогами и файлами для каждого типа. Однако вместо этого большинство операционных систем, включая UNIX, используют объектно-ориентированные методы для упрощения, организации и модульности реализации. Использование этих методов позволяет реализовывать в рамках одной структуры очень разные типы файловых систем, включая сетевые файловые системы, такие как NFS. Пользователи могут получать доступ к файлам, хранящимся в нескольких файловых системах на локальном диске или даже в файловых системах, доступных по сети.

Структуры данных и процедуры используются для изоляции базовой функциональности системных вызовов от деталей реализации. Таким образом, реализация файловой системы состоит из трёх основных уровней, как схематически показано на рисунке 15.5. Первый уровень — это интерфейс файловой системы, основанный на вызовах open(), read(), write() и close() , а также на файловых дескрипторах.

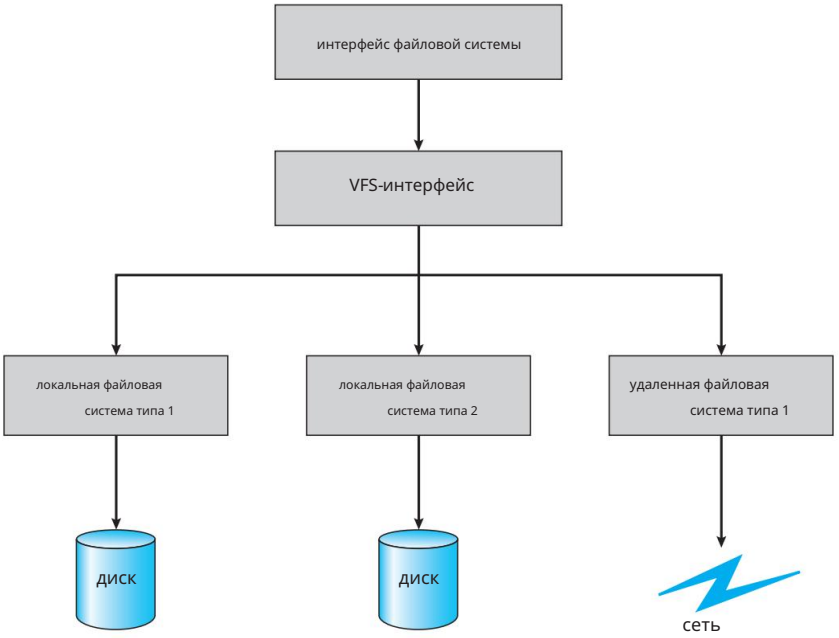


Рисунок 15.5 Схематическое изображение виртуальной файловой системы.

Второй уровень называется уровнем виртуальной файловой системы (VFS) . Уровень VFS выполняет две важные функции:

1. Он отделяет операции, характерные для файловой системы, от их реализации, определяя чистый интерфейс VFS . Несколько реализаций интерфейса VFS могут сосуществовать на одной машине, обеспечивая прозрачный доступ к различным типам файловых систем, смонтированных локально.

2. Предоставляет механизм уникального представления файла в сети. VFS основана на структуре представления файла, называемой vnode (виртуальный узел), которая содержит числовое обозначение уникального файла в сети.
   (Иноды UNIX уникальны только в пределах одной файловой системы.) Эта уникальность в масштабах всей сети необходима для поддержки сетевых файловых систем. Ядро поддерживает одну структуру vnode для каждого активного узла (файла или каталога).

Таким образом, VFS отличает локальные файлы от удаленных, а локальные файлы дополнительно различаются в соответствии с типами их файловой системы.

VFS активирует специфичные для файловой системы операции для обработки локальных запросов в соответствии с типами их файловой системы и вызывает процедуры протокола NFS (или другие процедуры протокола для других сетевых файловых систем) для удаленных запросов. Дескрипторы файлов формируются из соответствующих виртуальных узлов и передаются этим процедурам в качестве аргументов. Уровень, реализующий тип файловой системы или протокол удалённой файловой системы, является третьим уровнем архитектуры.

Давайте кратко рассмотрим архитектуру VFS в Linux. Четыре основных объекта Типы, определенные Linux VFS :

• Объект inode, представляющий отдельный файл. • Объект fil,
представляющий открытый файл. • Объект superblock,
представляющий всю файловую систему. • Объект dentry, представляющий
отдельную запись каталога.

Для каждого из этих четырёх типов объектов VFS определяет набор операций, которые могут быть реализованы. Каждый объект одного из этих типов содержит указатель на таблицу функций. В таблице функций перечислены адреса фактических функций, реализующих определённые операции для данного объекта. Например, сокращённый API для некоторых операций для объекта «файл» включает:

• int open(. . .) — Открыть файл. • int
close(. . .) — Закрыть уже открытый файл. • ssize t read(. . .)
— Чтение из файла.

• ssize t write(. . .) — Запись в файл.

• int mmap(. . .) — Отображение файла в памяти.

Для реализации каждой функции, указанной в определении объекта файла, требуется реализация объекта файла для конкретного типа файла. (Полное определение объекта файла указано в структуре файловых операций, которая находится в файле /usr/include/linux/fs.h.)

Таким образом, программный уровень VFS может выполнить операцию над одним из этих объектов, вызвав соответствующую функцию из таблицы функций объекта, не зная заранее, с каким именно объектом он работает. VFS не знает и не заботится о том, представляет ли индексный узел файл на диске, файл каталога или удалённый файл. Соответствующая функция для операции read() этого файла всегда будет находиться в одном и том же месте в таблице функций, и программный уровень VFS вызовет эту функцию, не заботясь о том, как именно считываются данные.

15.6 Удаленные файловые системы

С появлением сетей (глава 19) стало возможным взаимодействие между удалёнными компьютерами. Сетевое взаимодействие позволяет совместно использовать ресурсы, распределённые по всему кампусу или даже по всему миру. Один из очевидных ресурсов для совместного использования — это данные в виде файлов.

С развитием сетевых и файловых технологий методы удалённого обмена файлами изменились. Первый реализованный метод предполагает ручную передачу файлов между компьютерами с помощью программ, таких как FTP. Второй основной метод использует распределённую файловую систему (DFS), в которой удалённые каталоги видны с локального компьютера. В некотором смысле третий метод, Всемирная паутина, представляет собой возврат к первому. Для доступа к удалённым файлам требуется браузер, а для передачи файлов используются отдельные операции (по сути, оболочка для FTP) . Всё чаще для обмена файлами используются облачные вычисления (раздел 1.10.5).

FTP используется как для анонимного, так и для аутентифицированного доступа. Анонимный доступ позволяет пользователю передавать файлы без учётной записи на удалённой системе. Во Всемирной паутине анонимный обмен файлами используется практически исключительно. DFS предполагает гораздо более тесную интеграцию между машиной, получающей доступ к удалённым файлам, и машиной, предоставляющей эти файлы. Эта интеграция усложняет работу, как мы расскажем в этом разделе.

15.6.1 Модель клиент-сервер

Удалённые файловые системы позволяют компьютеру монтировать одну или несколько файловых систем с одной или нескольких удалённых машин. В этом случае машина, содержащая файлы, является сервером , а машина, запрашивающая доступ к файлам, — клиентом . Клиент-серверные отношения характерны для сетевых машин. Как правило, сервер объявляет, что ресурс доступен клиентам, и точно указывает, какой именно ресурс (в данном случае, какие файлы) и каким именно клиентам. Сервер может обслуживать несколько клиентов, а клиент может использовать несколько серверов, в зависимости от особенностей реализации данной клиент-серверной модели.

Сервер обычно определяет доступные файлы на уровне тома или каталога. Идентификация клиента сложнее. Клиент может быть указан по сетевому имени или другому идентификатору, например, IP- адресу, но эти имена могут быть подменены или имитированы. В результате подмены неавторизованному клиенту может быть предоставлен доступ к серверу. Более безопасные решения включают безопасную аутентификацию клиента с помощью зашифрованных ключей. К сожалению, обеспечение безопасности сопряжено со многими проблемами, включая обеспечение совместимости клиента и сервера (они должны использовать одни и те же алгоритмы шифрования) и безопасность обмена ключами (перехваченные ключи могут снова позволить несанкционированный доступ). Из-за сложности решения этих проблем чаще всего используются небезопасные методы аутентификации.

В случае UNIX и его сетевой файловой системы (NFS) аутентификация по умолчанию осуществляется через сетевую информацию клиента. В этой схеме идентификаторы пользователей на клиенте и сервере должны совпадать. В противном случае сервер не сможет определить права доступа к файлам. Рассмотрим пример пользователя с идентификатором 1000 на клиенте и 2000 на сервере. Запрос клиента к серверу на получение конкретного файла не будет обработан должным образом, поскольку сервер определит, имеет ли пользователь 1000 доступ к файлу, а не будет основываться на реальном идентификаторе пользователя 2000. Таким образом, доступ предоставляется или отклоняется на основе неверной информации аутентификации. Сервер должен доверять клиенту в предоставлении правильного идентификатора пользователя. Обратите внимание, что протоколы NFS допускают отношения «многие ко многим». То есть, множество серверов могут предоставлять файлы множеству клиентов. Фактически, одна и та же машина может быть как сервером для одних клиентов NFS , так и клиенто NFS- серверы.

После монтирования удаленной файловой системы запросы на файловые операции отправляются от имени пользователя по сети на сервер по протоколу DFS .
Обычно запрос на открытие файла отправляется вместе с идентификатором запрашивающего пользователя. Затем сервер применяет стандартные проверки доступа, чтобы определить, есть ли у пользователя учётные данные для доступа к файлу в запрошенном режиме. Запрос либо разрешён, либо отклонён. Если запрос разрешён, клиентскому приложению возвращается дескриптор файла, и приложение может выполнять чтение, запись и другие операции с файлом. Клиент закрывает файл после завершения доступа. Операционная система может применять семантику, аналогичную семантике локального монтирования файловой системы, или использовать другую семантику.

15.6.2 Распределенные информационные системы

Для упрощения управления клиент-серверными системами распределённые информационные системы, также известные как распределённые службы имён, обеспечивают унифицированный доступ к информации, необходимой для удалённых вычислений. Система доменных имён (DNS) обеспечивает преобразование имён хостов в сетевые адреса для всего Интернета.
До широкого распространения DNS файлы, содержащие одну и ту же информацию, пересылались по электронной почте или по FTP между всеми сетевыми хостами. Очевидно, что эта методология не была масштабируемой! Подробнее о DNS говорится в разделе 19.3.1.

Другие распределённые информационные системы предоставляют пространство для хранения имени пользователя/пароля/идентификатора пользователя / идентификатора группы для распределённого хранилища. В системах UNIX используется широкий спектр методов распределённой информации. Компания Sun Microsystems (ныне часть корпорации Oracle) представила систему «жёлтых страниц» (позднее переименованную в сетевую информационную службу, или NIS), и большинство предприятий отрасли переняли её использование. Она централизует хранение имён пользователей, имён хостов, информации о принтерах и т. К сожалению, он использует небезопасные методы аутентификации, включая отправку паролей пользователей в незашифрованном виде (открытым текстом) и идентификацию хостов по IP- адресу. NIS+ от Sun был гораздо более безопасной заменой NIS , но был гораздо сложнее и не получил широкого распространения.

В случае с общей файловой системой Интернета (CIFS) от Microsoft сетевая информация используется совместно с аутентификацией пользователя (имя пользователя и пароль) для создания сетевого имени пользователя, которое сервер использует для принятия решения о предоставлении или запрете доступа к запрашиваемой файловой системе. Для корректной аутентификации имена пользователей должны совпадать на всех компьютерах (как в случае с NFS). Microsoft использует Active Directory в качестве распределенной структуры именования, предоставляя пользователям единое пространство имен. После установки распределенная система именования используется всеми клиентами и серверами для аутентификации пользователей через версию протокола сетевой аутентификации Kerberos от Microsoft (https://web.mit.edu/kerberos/).

Отрасль переходит к использованию облегчённого протокола доступа к каталогам (LDAP) в качестве безопасного механизма распределённого именования. Active Directory, по сути, основана на LDAP. Oracle Solaris и большинство других основных операционных систем поддерживают LDAP и позволяют использовать его для аутентификации пользователей, а также для получения информации по всей системе, например, о доступности принтеров.
Предполагается, что организация может использовать один распределенный каталог LDAP для хранения всей информации о пользователях и ресурсах для всех компьютеров организации.
Результатом станет безопасный единый вход для пользователей, которым потребуется вводить свои аутентификационные данные один раз для доступа ко всем компьютерам организации. Это также упростит администрирование системы, объединив в одном месте информацию, которая в настоящее время разбросана по разным файлам в каждой системе или в различных распределенных информационных сервисах.

15.6.3 Виды отказов

Локальные файловые системы могут выйти из строя по разным причинам, включая сбой диска, содержащего файловую систему, повреждение структуры каталогов или другой информации управления диском (совместно именуемой метаданными), сбой контроллера диска, сбой кабеля и сбой хост-адаптера. Сбой пользователя или системного администратора также может привести к потере файлов или удалению целых каталогов или томов. Многие из этих сбоев приводят к сбою хоста и появлению сообщения об ошибке, и для устранения повреждения может потребоваться вмешательство человека.

Удалённые файловые системы имеют ещё больше видов отказов. Из-за сложности сетевых систем и необходимости взаимодействия между удалёнными машинами, на корректную работу удалённых файловых систем может повлиять гораздо больше проблем. В случае сетей сетевое соединение может прерываться между двумя хостами. Такие сбои могут быть вызваны отказом оборудования, его неправильной конфигурацией или проблемами реализации сети. Хотя некоторые сети обладают встроенной отказоустойчивостью, включая поддержку нескольких путей между хостами, многие сети её не имеют. Таким образом, любой единичный сбой может прервать выполнение команд DFS .

Представьте себе клиента, использующего удалённую файловую систему. У него открыты файлы с удалённого хоста; среди прочих действий он может выполнять поиск в каталогах для открытия файлов, чтение или запись данных в файлы, а также закрытие файлов. Теперь представьте себе разрыв сети, сбой сервера или даже его плановое выключение. Внезапно удалённая файловая система становится недоступной.
Такой сценарий довольно распространён, поэтому клиентской системе было бы нецелесообразно действовать так, как при потере локальной файловой системы. Вместо этого система может либо завершить все операции с потерянным сервером, либо отложить их до тех пор, пока сервер не станет снова доступен. Семантика сбоев определена и реализована как часть протокола удалённой файловой системы. Завершение всех операций может привести к потере данных и терпения пользователей. Таким образом, большинство протоколов DFS либо принудительно, либо разрешают отсрочку операций с файловой системой на удалённых хостах в надежде, что удалённый хост снова станет доступен.

Для реализации такого типа восстановления после сбоя определённая информация о состоянии может храниться как на клиенте, так и на сервере. Если и сервер, и клиент знают о своих текущих действиях и открытых файлах, то они могут без проблем восстановиться после сбоя. В ситуации, когда сервер выходит из строя, но должен распознать, что он удалённо смонтировал экспортированные файловые системы и открыл файлы, NFS версии 3 использует простой подход, реализуя DFS без сохранения состояния . По сути, он предполагает, что клиентский запрос на чтение или запись файла не был бы выполнен, если бы файловая система не была удалённо смонтирована, а файл ранее не был открыт. Протокол NFS передаёт всю информацию, необходимую для поиска соответствующего файла и выполнения запрошенной операции. Аналогичным образом, он не отслеживает, у каких клиентов смонтированы экспортированные тома, снова предполагая, что если поступает запрос, он должен быть легитимным. Хотя такой подход без сохранения состояния делает NFS устойчивой и довольно простой в реализации, он также делает её небезопасной. Например, NFS- сервер может разрешить поддельные запросы на чтение или запись . Эти проблемы устранены в отраслевом стандарте NFS версии 4, в котором NFS становится с отслеживанием состояния для повышения безопасности, производительности и функциональности.

15.7 Семантика согласованности

Семантика согласованности представляет собой важный критерий оценки любой файловой системы, поддерживающей общий доступ к файлам. Эта семантика определяет, как несколько пользователей системы могут одновременно получать доступ к общему файлу. В частности, она определяет, когда изменения данных, внесённые одним пользователем, будут видны другим пользователям. Эта семантика обычно реализуется в виде кода вместе с файловой системой.

Семантика согласованности напрямую связана с алгоритмами синхронизации процессов из главы 6. Однако сложные алгоритмы этой главы, как правило,

Не рекомендуется применять в случае файлового ввода-вывода из-за больших задержек и низкой скорости передачи данных на дисках и в сетях. Например, выполнение атомарной транзакции на удалённом диске может потребовать нескольких сетевых соединений, нескольких операций чтения и записи на диск, или и того, и другого. Системы, пытающиеся реализовать такой полный набор функций, как правило, работают неэффективно. Успешную реализацию сложной семантики совместного доступа можно найти в файловой системе Andrew.

В дальнейшем обсуждении мы предполагаем, что серия обращений к файлу (то есть чтение и запись), предпринимаемых пользователем в отношении одного и того же файла, всегда заключена между операциями open() и close() . Эта серия обращений между операциями open() и close() составляет сеанс fil. Для иллюстрации этой концепции мы приведём несколько ярких примеров семантики согласованности.

15.7.1 Семантика UNIX

Файловая система UNIX (глава 19) использует следующую семантику согласованности:

- Записи, сделанные пользователем в открытый файл, немедленно видны другим пользователям, которые откройте этот файл.
- Один из режимов общего доступа позволяет пользователям делиться указателем текущего местоположения в файле. Таким образом, перемещение указателя одним пользователем влияет на всех пользователей, предоставляющих доступ. В этом случае файл имеет единое изображение, которое чередуется при всех обращениях, независимо от их источника.

В семантике UNIX файл связан с одним физическим изображением, доступ к которому осуществляется как к эксклюзивному ресурсу. Конкуренция за это изображение приводит к задержкам в работе пользовательских процессов.

15.7.2 Семантика сеанса

Файловая система Andrew (OpenAFS) использует следующую семантику согласованности:

- Записи, сделанные пользователем в открытый файл, не видны сразу другим пользователям. у которых открыт один и тот же файл.
- После закрытия файла внесённые в него изменения видны только в последующих сеансах. В уже открытых экземплярах файла эти изменения не отражаются.

Согласно этой семантике, файл может быть временно связан с несколькими (возможно, разными) изображениями одновременно. Следовательно, нескольким пользователям разрешено одновременно выполнять чтение и запись своих изображений файла без задержек. Планирование практически не ограничивается.

доступы.

15.7.3 Семантика неизменяемых общих файлов

Уникальный подход — неизменяемые общие файлы. После того, как файл объявлен его создателем как общий, его нельзя изменить. Неизменяемый файл обладает двумя ключевыми свойствами: его имя нельзя использовать повторно, а его содержимое нельзя изменить.
Таким образом, имя неизменяемого файла означает, что его содержимое фиксировано. Реализация этой семантики в распределённой системе (глава 19) проста, поскольку совместное использование данных организовано (доступно только для чтения).

15.8 НФС

Сетевые файловые системы широко распространены. Они, как правило, интегрированы с общей структурой каталогов и интерфейсом клиентской системы. NFS — хороший пример широко распространённой и хорошо реализованной клиент-серверной сетевой файловой системы. Здесь мы используем её в качестве примера для изучения особенностей реализации сетевых файловых систем.

NFS — это одновременно реализация и спецификация программной системы для доступа к удалённым файлам по локальным (или даже глобальным) сетям. NFS является частью ONC+, которую поддерживают большинство поставщиков UNIX и некоторые операционные системы для ПК . Описанная здесь реализация является частью операционной системы Solaris, которая представляет собой модифицированную версию UNIX SVR4. Она использует протокол TCP или UDP/IP (в зависимости от соединительной сети). Спецификация и реализация тесно переплетены в нашем описании NFS. При необходимости подробного описания мы ссылаемся на реализацию Solaris; если описание носит общий характер, оно применимо и к спецификации.

Существует несколько версий NFS, последняя из которых — версия 4. Вот, мы описываем версию 3, которая используется наиболее часто.

15.8.1 Обзор

NFS рассматривает множество взаимосвязанных рабочих станций как набор независимых машин с независимыми файловыми системами. Цель — обеспечить прозрачный доступ к файлам между этими системами (по явному запросу).
Совместное использование основано на клиент-серверных отношениях. Машина может быть, и часто является, как клиентом, так и сервером. Совместное использование разрешено между любой парой машин. Чтобы обеспечить независимость машин, совместное использование удаленной файловой системы влияет только на клиентскую машину и ни на какую другую машину.

Чтобы удалённый каталог был доступен прозрачно с определённой машины, например, с M1, клиент этой машины должен сначала выполнить операцию монтирования. Семантика этой операции подразумевает монтирование удалённого каталога поверх каталога локальной файловой системы. После завершения операции монтирования смонтированный каталог становится неотъемлемой частью поддерева локальной файловой системы, заменяя поддерево, нисходящее от локального каталога. Локальный каталог становится именем корня вновь смонтированного каталога.

Указание удалённого каталога в качестве аргумента для операции монтирования не осуществляется прозрачно; необходимо указать местоположение (или имя хоста) удалённого каталога. Однако с этого момента пользователи на машине M1 смогут получать доступ к файлам в удалённом каталоге совершенно прозрачно.

Для иллюстрации монтирования файлов рассмотрим файловую систему, изображённую на рисунке 15.6, где треугольники представляют поддеревья интересующих нас каталогов. На рисунке показаны три независимые файловые системы на машинах с именами U, S1 и S2. В этот момент на каждой машине доступен только локальный доступ к файлам. На рисунке 15.7(a) показаны результаты монтирования S1:/usr/shared поверх U:/usr/local.
На этом рисунке показано, как пользователи на компьютере U видят свою файловую систему. После завершения монтирования они могут получить доступ к любому файлу в каталоге dir1 , используя префикс /usr/local/dir1. Исходный каталог /usr/local на этом компьютере больше не отображается.

При условии аккредитации прав доступа любая файловая система или любой каталог внутри файловой системы может быть удаленно смонтирован поверх любого локального каталога.

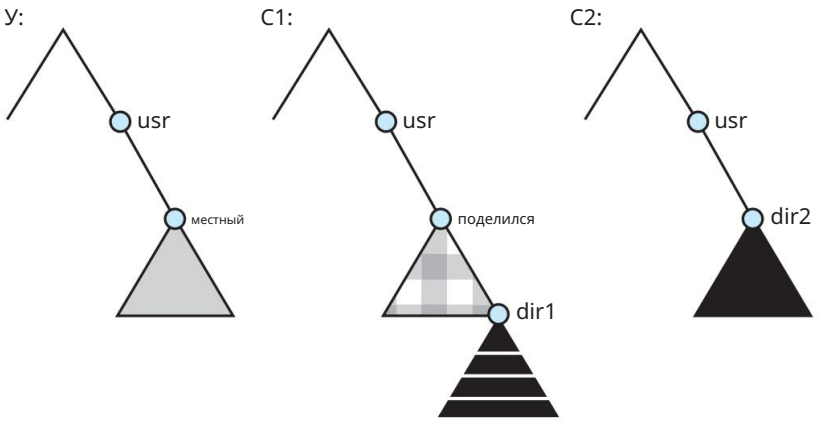


Рисунок 15.6 Три независимые файловые системы.

Бездисковые рабочие станции могут даже монтировать собственные корневые каталоги с серверов. В некоторых реализациях NFS также разрешено каскадное монтирование . То есть, файловая система может быть смонтирована поверх другой файловой системы, смонтированной удалённо, а не локально. На машину влияют только те монтирования, которые она сама инициировала. Монтирование удалённой файловой системы не даёт клиенту доступа к другим файловым системам, которые были случайно смонтированы поверх предыдущей. Таким образом, механизм монтирования не обладает свойством транзитивности.

На рисунке 15.7(b) показано каскадное монтирование. На рисунке показан результат монтирования каталога S2:/usr/dir2 к каталогу U:/usr/local/dir1, который уже удалённо смонтирован с каталога S1. Пользователи могут получить доступ к файлам в каталоге dir2 на диске U, используя префикс /usr/local/dir1. Если общая файловая система смонтирована над домашними каталогами пользователя на всех машинах в сети, пользователь может войти на любую рабочую станцию и получить доступ к своей домашней среде. Это свойство обеспечивает мобильность пользователя.

Одной из целей разработки NFS была работа в гетерогенной среде, состоящей из различных машин, операционных систем и сетевых архитектур.

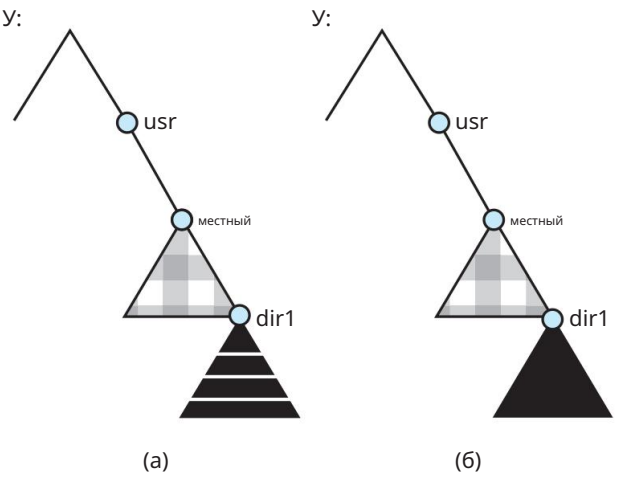


Рисунок 15.7 Монтирование в NFS. (a) Монтирования. (b) Каскадное монтирование.

Спецификация NFS независима от этих носителей. Эта независимость достигается за счёт использования примитивов RPC, построенных на основе протокола внешнего представления данных (XDR), используемого между двумя независимыми от реализации интерфейсами. Таким образом, если разнородные машины и файловые системы системы правильно подключены к NFS, файловые системы различных типов могут быть смонтированы как локально, так и удаленно.

Спецификация NFS различает сервисы, предоставляемые механизмом монтирования, и собственно сервисы удалённого доступа к файлам. Соответственно, для этих сервисов определены два отдельных протокола: протокол монтирования и протокол удалённого доступа к файлам — протокол NFS . Протоколы представлены в виде наборов RPC. Эти RPC являются строительными блоками, используемыми для реализации прозрачного удалённого доступа к файлам.

15.8.2 Протокол монтирования

Протокол монтирования устанавливает начальное логическое соединение между сервером и клиентом. В Solaris на каждой машине существует серверный процесс вне ядра, выполняющий функции протокола.

Операция монтирования включает в себя имя удалённого каталога, который необходимо монтировать, и имя сервера, на котором он хранится. Запрос на монтирование сопоставляется с соответствующим RPC-вызовом и пересылается серверу монтирования, работающему на данном сервере. Сервер ведёт список экспорта , в котором указаны локальные файловые системы, которые он экспортирует для монтирования, а также имена машин, которым разрешено их монтировать. (В Solaris этот список — /etc/dfs/dfstab, который может редактировать только суперпользователь.) Этот список также может включать права доступа, например, «только чтение». Для упрощения обслуживания списков экспорта и таблиц монтирования можно использовать распределённую схему именования для хранения этой информации и предоставления её соответствующим клиентам.

Напомним, что любой каталог в экспортированной файловой системе может быть удалённо смонтирован аккредитованным компьютером. Таким каталогом является компонентный модуль. Когда сервер получает запрос на монтирование, соответствующий его экспортному списку, он возвращает клиенту дескриптор файла, который служит ключом для дальнейшего доступа к файлам в смонтированной файловой системе. Дескриптор файла содержит всю информацию, необходимую серверу для различения хранящегося на нём файла. В терминах UNIX дескриптор файла состоит из идентификатора файловой системы и номера инода, позволяющего точно определить смонтированный каталог в экспортированной файловой системе.

Сервер также ведёт список клиентских машин и соответствующих им каталогов, смонтированных в данный момент. Этот список используется в основном в административных целях, например, для уведомления всех клиентов о выходе сервера из строя.
Только путем добавления и удаления записей в этом списке протокол монтирования может повлиять на состояние сервера.

Обычно система имеет статическую предварительную конфигурацию монтирования, которая устанавливается во время загрузки (/etc/vfstab в Solaris); однако эту конфигурацию можно изменить. Помимо самой процедуры монтирования, протокол монтирования включает в себя несколько других процедур, таких как размонтирование и возврат списка экспорта.

15.8.3 Протокол NFS

Протокол NFS предоставляет набор RPC-вызовов для удалённых файловых операций. Процедуры поддерживают следующие операции:

• Поиск файла в каталоге • Чтение набора записей каталога • Управление ссылками и каталогами • Доступ к атрибутам файла • Чтение и запись файлов

Эти процедуры могут быть вызваны только после создания дескриптора файла для удаленно смонтированного каталога.

Пропуск операций открытия и закрытия является намеренным. Характерной особенностью NFS- серверов является то, что они не сохраняют состояние. Серверы не хранят информацию о своих клиентах от одного доступа к другому. На стороне сервера не существует аналогов таблице открытых файлов или файловым структурам UNIX . Следовательно, каждый запрос должен предоставлять полный набор аргументов, включая уникальный идентификатор файла и абсолютное смещение внутри файла для соответствующих операций. Результирующая конструкция является надежной; не требуется никаких специальных мер для восстановления сервера после сбоя. Для этой цели файловые операции должны быть идемпотентными, то есть одна и та же операция, выполненная несколько раз, должна иметь тот же эффект, как если бы она была выполнена только один раз. Для достижения идемпотентности каждый запрос NFS имеет порядковый номер, позволяющий серверу определить, был ли запрос дублирован или какие-либо из них отсутствуют.

Ведение списка клиентов, о котором мы упомянули, по-видимому, нарушает принцип отсутствия состояния сервера. Однако этот список не является необходимым для корректной работы клиента или сервера, и, следовательно, его не нужно восстанавливать после сбоя сервера. Следовательно, он может содержать несогласованные данные и рассматривается лишь как подсказка.

Ещё одним следствием философии сервера без сохранения состояния и результатом синхронности RPC является то, что изменённые данные (включая блоки косвенной адресации и состояния) должны быть зафиксированы на диске сервера до того, как результаты будут возвращены клиенту. То есть клиент может кэшировать блоки записи, но при их сбросе на сервер предполагается, что они достигли дисков сервера. Сервер должен записывать все данные NFS синхронно. Таким образом, сбой сервера и его восстановление будут незаметны для клиента; все блоки, которыми сервер управляет для клиента, останутся нетронутыми. Результирующее падение производительности может быть значительным, поскольку теряются преимущества кэширования. Производительность можно повысить, используя хранилище с собственным энергонезависимым кэшем (обычно это память с резервным питанием от батареи). Контроллер диска подтверждает запись на диск, когда запись сохраняется в энергон По сути, хост видит очень быструю синхронную запись. Эти блоки остаются нетронутыми даже после сбоя системы и периодически записываются из этого стабильного хранилища на диск.

Один вызов процедуры записи NFS гарантированно является атомарным и не смешивается с другими вызовами записи в тот же файл. Однако протокол NFS не предоставляет механизмов управления параллельными операциями. Системный вызов awrite() может быть разбит на несколько RPC- записей, поскольку каждый вызов записи или чтения NFS может содержать до 8 КБ данных, а UDP- пакеты ограничены 1500 байтами. В результате данные двух пользователей, записывающих данные в один и тот же удалённый файл, могут быть перемешаны. Утверждается, что, поскольку управление блокировками по своей сути является состоянием, служба вне NFS должна обеспечивать блокировку (и Solaris это делает). Пользователям рекомендуется координировать доступ к общим файлам, используя механизмы, не входящие в NFS.

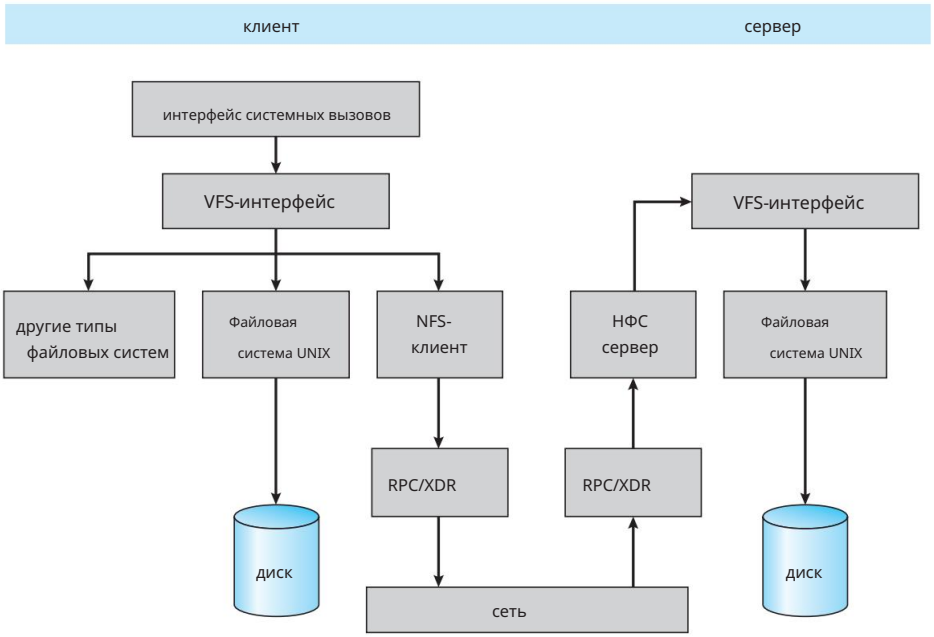


Рисунок 15.8 Схематическое изображение архитектуры NFS.

NFS интегрирована в операционную систему через VFS. В качестве иллюстрации архитектуры давайте проследим, как обрабатывается операция над уже открытым удалённым файлом (см. пример на рисунке 15.8). Клиент инициирует операцию обычным системным вызовом. Уровень операционной системы сопоставляет этот вызов с операцией VFS на соответствующем vnode. Уровень VFS идентифицирует файл как удалённый и вызывает соответствующую процедуру NFS . Выполняется вызов RPC к сервисному уровню NFS на удалённом сервере. Этот вызов повторно инжектируется на уровень VFS на удалённой системе, который обнаруживает его локальность и вызывает соответствующую операцию файловой системы. Этот путь прослеживается для возврата результата. Преимущество этой архитектуры заключается в том, что клиент и сервер идентичны; таким образом, машина может быть клиентом, сервером или обоими. Фактическое обслуживание на каждом сервере выполняется потоками ядра.

15.8.4 Перевод имени пути

Перевод имени пути в NFS включает в себя разбор имени пути, такого как /usr/local/dir1/file.txt, на отдельные записи каталога или компоненты: (1) usr, (2) local и (3) dir1. Перевод имени пути выполняется путем разбиения пути на имена компонентов и выполнения отдельного вызова поиска NFS для каждой пары имени компонента и vnode каталога. После пересечения точки монтирования каждый поиск компонента вызывает отдельный RPC к серверу. Эта дорогостоящая схема обхода имени пути необходима, поскольку структура логического пространства имен каждого клиента уникальна и определяется выполненными клиентом монтированиями. Было бы гораздо эффективнее передать серверу имя пути и получить целевой vnode при обнаружении точки монтирования. Однако в любой момент для конкретного клиента может существовать другая точка монтирования, о которой сервер без сохранения состояния не знает.

Для ускорения поиска кэш поиска по имени каталога на стороне клиента хранит виртуальные узлы (vnode) для имён удалённых каталогов. Этот кэш ускоряет обращение к файлам с тем же начальным путём. Кэш каталога сбрасывается, если атрибуты, возвращаемые сервером, не соответствуют атрибутам кэшированного vnode.

Напомним, что некоторые реализации NFS позволяют монтировать удалённую файловую систему поверх другой уже смонтированной удалённой файловой системы (каскадное монтирование). При каскадном монтировании у клиента в обходе пути может участвовать более одного сервера. Однако, когда клиент выполняет поиск в каталоге, в котором сервер смонтировал файловую систему, он видит базовый каталог вместо смонтированного.

15.8.5 Удалённые операции

За исключением открытия и закрытия файлов, существует практически однозначное соответствие между обычными системными вызовами UNIX для файловых операций и RPC-вызовами протокола NFS . Таким образом, удалённая файловая операция может быть напрямую транслирована в соответствующий RPC. Концептуально NFS придерживается парадигмы удалённого обслуживания; но на практике для повышения производительности используются методы буферизации и кэширования. Прямого соответствия между удалённой операцией и RPC не существует. Вместо этого блоки файлов и атрибуты файлов извлекаются RPC- вызовами и кэшируются локально. Последующие удалённые операции используют кэшированные данные с учётом ограничений согласованности.

Существует два кэша: кэш атрибутов файлов (информации об инодах) и кэш блоков файлов. При открытии файла ядро обращается к удалённому серверу, чтобы определить, следует ли извлекать или повторно проверять кэшированные атрибуты. Кэшированные блоки файлов используются только в том случае, если соответствующие кэшированные атрибуты актуальны. Кэш атрибутов обновляется при поступлении новых атрибутов с сервера. Кэшированные атрибуты по умолчанию сбрасываются через 60 секунд. Между сервером и клиентом используются как методы упреждающего чтения, так и методы отложенной записи. Клиенты не освобождают блоки отложенной записи, пока сервер не подтвердит, что данные были записаны на диск. Отложенная запись сохраняется даже при одновременном открытии файла в конфликтующих режимах. Следовательно, семантика UNIX (раздел 15.7.1) не сохраняется.

Настройка системы для повышения производительности затрудняет определение семантики согласованности NFS. Новые файлы, созданные на компьютере, могут быть не видны на других компьютерах в течение 30 секунд. Более того, записи в файл на одном сайте могут быть видны или не видны на других сайтах, где этот файл открыт для чтения. При новом открытии файла учитываются только изменения, уже сброшенные на сервер. Таким образом, NFS не обеспечивает ни строгой эмуляции семантики UNIX , ни семантики сеансов Andrew (раздел 15.7.2). Несмотря на эти недостатки, практичность и высокая производительность механизма делают его наиболее широко используемой распределенной системой, работающей с оборудованием разных производителей.

15.9 Резюме

- Операционные системы общего назначения предоставляют множество типов файловых систем: от специализированных до общих.

- Тома, содержащие файловые системы, могут быть смонтированы в файловую систему компьютера.
  системное пространство.
- В зависимости от операционной системы, файловое пространство является бесшовным
  (смонтированные файловые системы, интегрированные в структуру каталогов) или отдельные
  (каждая смонтированная файловая система имеет свое обозначение).
- Для запуска системы должна быть хотя бы одна файловая система загрузочной.
  — то есть, он должен содержать операционную систему. Сначала запускается загрузчик;
  это простая программа, которая может найти ядро в файловой системе, загрузить
  и запустить его выполнение. Система может содержать несколько загрузочных разделов,
  позволяя администратору выбирать, какой из них запустить во время загрузки.
- Большинство систем являются многопользовательскими и поэтому должны предоставлять средства
  для совместного доступа к файлам и их защиты. Файлы и каталоги часто содержат метаданные,
  такие как разрешения на доступ владельца, пользователя и группы.
- Разделы массового хранения данных используются либо для блочного ввода-вывода , либо для файловых систем.
  Каждая файловая система находится в томе, который может состоять из одного
  раздел или несколько разделов, работающих вместе через менеджер томов.
- Для упрощения реализации нескольких файловых систем операционная система
  можно использовать многоуровневый подход с виртуальным интерфейсом файловой системы
  бесперебойный доступ к потенциально разнородным файловым системам.
- Удаленные файловые системы могут быть реализованы просто с помощью такой программы
  как FTP или веб-серверы и клиенты во Всемирной паутине, или с более
  Функциональность по модели клиент-сервер. Запросы на монтирование и идентификаторы пользователей должны быть
  быть аутентифицирован для предотвращения несанкционированного доступа.
- Клиент-серверные возможности изначально не обмениваются информацией, а распределены
  Информационная система, такая как DNS, может использоваться для обеспечения такого обмена,
  обеспечивая единое пространство имен пользователей, управление паролями и систему
  Идентификация. Например, Microsoft CIFS использует Active Directory, которая
  использует версию протокола сетевой аутентификации Kerberos для предоставления полного
  набора услуг именования и аутентификации между компьютерами
  в сети.
- Как только станет возможным совместное использование файлов, необходимо выбрать и реализовать модель
  семантики согласованности, чтобы контролировать множественный одновременный доступ к одному и тому же файлу.
  Файл. Семантические модели включают UNIX, сеансовые и неизменяемые общие файлы.
  семантика.
- NFS — это пример удалённой файловой системы, предоставляющей клиентам беспрепятственный
  доступ к каталогам, файлам и даже целым файловым системам. Полнофункциональная
  удаленная файловая система включает в себя протокол связи с удаленными операциями и
  преобразованием имен путей.

Практические упражнения

15.1 Объясните, как уровень VFS позволяет операционной системе легко поддерживать несколько
типов файловых систем.

15.2 Зачем в одной системе может быть более одного типа файловой системы?