Новые возможности операционных систем

Содержание

Эффективное использование легковесных процессов в симметричных мультипроцессорах

Контекст процесса

Ядерные нити

Пользовательские легковесные процессы

Пользовательские нити

Методология применения легковесных процессов

Современные файловые системы

Ограничения традиционных файловых систем

Распространенные файловые системы

Файловые системы с журнализацией

### Эффективное использование легковесных процессов в симметричных мультипроцессорах

Поддерживаемые в современных операционных системах (в частности, в ОС UNIX) понятия нити (thread), потока управления, или легковесного процесса на самом деле появились и получили реализацию около 30 лет тому назад. Наиболее известной операционной системой, ориентированной на поддержку множественных процессов, которые работают в общем адресном пространстве и с общими прочими ресурсами, была легендарная ОС Multics. Эта операционная система заслуживает длительного отдельного обсуждения, но, естественно не в данном курсе. Мы рассмотрим (в общих чертах) особенности легковесных процессов в современных вариантах операционной системы UNIX. По всей видимости, все или почти все содержимое этого раздела можно легко отнести к любой операционной системе, поддерживающей легковесные процессы. Несмотря на различия в терминологии, в различных реализациях легковесных процессов выделяются три класса. Но прежде, чем перейти к рассмотрению этих классов, обсудим общую природу процесса в ОС UNIX.

#### Контекст процесса

Каждому процессу соответствует контекст, в котором он выполняется. Этот контекст включает содержимое пользовательского адресного пространства - пользовательский контекст (т.е. содержимое сегментов программного кода, данных, стека, разделяемых сегментов и сегментов файлов, отображаемых в виртуальную память), содержимое аппаратных регистров - регистровый контекст (таких, как регистр счетчика команд, регистр состояния процессора, регистр указателя стека и регистров общего назначения), а также структуры данных ядра (контекст системного уровня), связанные с этим процессом. Контекст процесса системного уровня в ОС UNIX состоит из "статической" и "динамических" частей. У каждого процесса имеется одна статическая часть контекста системного уровня и переменное число динамических частей.

Статическая часть контекста процесса системного уровня включает следующее:

A.Описатель процесса, т.е. элемент таблицы описателей существующих в системе процессов. Описатель процесса включает, в частности, следующую информацию:

* состояние процесса;
* физический адрес в основной или внешней памяти u-области процесса;
* идентификаторы пользователя, от имени которого запущен процесс;
* идентификатор процесса;
* прочую информацию, связанную с управлением процессом.

B. U-область (u-area), индивидуальная для каждого процесса область пространства ядра, обладающая тем свойством, что хотя u-область каждого процесса располагается в отдельном месте физической памяти, u-области всех процессов имеют один и тот же виртуальный адрес в адресном пространстве ядра. Именно это означает, что какая бы программа ядра не выполнялась, она всегда выполняется как ядерная часть некоторого пользовательского процесса, и именно того процесса, u-область которого является "видимой" для ядра в данный момент времени. U-область процесса содержит:

* указатель на описатель процесса;
* идентификаторы пользователя;
* счетчик времени, которое процесс реально выполнялся (т.е. занимал процессор) в режиме пользователя и режиме ядра;
* параметры системного вызова;
* результаты системного вызова;
* таблица дескрипторов открытых файлов;
* предельные размеры адресного пространства процесса;
* предельные размеры файла, в который процесс может писать;
* и т.д.

Динамическая часть контекста процесса - это один или несколько стеков, которые используются процессом при его выполнении в режиме ядра. Число ядерных стеков процесса соответствует числу уровней прерывания, поддерживаемых конкретной аппаратурой.

#### Ядерные нити

Базовым классом являются ядерные нити. В мире UNIX это не новость. Когда в пользовательском процессе происходит системный вызов или прерывание, выполняется ядерная составляющая пользовательского процесса в своем собственном контексте, включающем набор ядерных стеков и регистровое окружение. Естественно, все ядерные составляющие пользовательских процессов работают в общем адресном пространстве с общим набором ресурсов ядра. Поэтому их вполне можно назвать ядерными легковесными процессами. Наличие ядерных нитей, в частности, облегчает обработку прерываний в режиме ядра. Как и в случае прерывания обычного пользовательского процесса, обработка прерывания ядерной нити производится в ее контексте, и после возврата из прерывания продолжается выполнение прерванной ядерной нити. Кроме того, каждая ядерная нить, вообще говоря, обладает собственным приоритетом по отношению к праву выполняться на процессоре (конечно, этот приоритет связан с приоритетом соответствующего пользовательского процесса). Это позволяет использовать гибкую политику планирования процессорных ресурсов для ядерных составляющих. Итак, ядерные нити должны существовать независимо от того, поддерживаются ли легковесные процессы в режиме пользователя. Наверное, трудно найти сегодня какую-либо многопользовательскую операционную систему, в ядре которой в каком-то виде не поддерживались бы нити.

#### Пользовательские легковесные процессы

Видимо, следующим по важности классом легковесных процессов являются пользовательские LWP (LightWeight Processes). Механизмы этого рода позволяют пользователям организовать несколько потоков управления в одном адресном пространстве. Все LWP одного пользовательского процесса совместно используют все ресурсы процесса. При поступлении процессу сигнала на этот сигнал реагируют все LWP в соответствии со своими собственными установками. С другой стороны, каждый LWP обладает своим собственным контекстом, включающим, как и в случае ядерных нитей, стек и регистровое окружение (в частности, содержимое индивидуального счетчика команд). Любому LWP пользовательского процесса соответствует отдельная ядерная нить. Это означает, что каждый LWP может отдельно планироваться (и поэтому LWP одного пользовательского процесса могут параллельно выполняться на разных процессорах симметричного мультипроцессорного компьютера), и системные вызовы и прерывания LWP могут обрабатываться независимо. Основным преимуществом использования LWP является возможность достижения реального распараллеливания программы при ее выполнении на симметричном мультипроцессоре (на недостатках мы остановимся ниже).

#### Пользовательские нити

Наконец, к третьему классу легковесных процессов относятся пользовательские нити. Они называются пользовательскими, поскольку реализуются не ядром ОС, а с помощью специальной библиотеки функций (поэтому, например, в ОС Mach их называют C-Threads). Это тоже очень старая идея, к использованию которой неоднократно прибегали все опытные программисты (здесь уже даже не важно, в среде какой операционной системы выполняется программа). Суть идеи состоит в том, что вся программа пользователя строится в виде набора сопрограмм (coroutine), которые выполняются под управлением общего монитора. Естественно, что в мониторе поддерживаются контексты всех сопрограмм, но и монитор, и сопрограммы являются составляющими одного пользовательского процесса, которому соответствует одна ядерная нить. Конечно, с использованием пользовательских нитей невозможно достичь реального распараллеливания программы. Единственный реальный эффект, которого можно добиться, состоит в возможности распараллеливания обменов при использовании асинхронного режима системных вызовов. Как считают некоторые специалисты (к числу которых не относится автор этой части курса), основное достоинство использования пользовательских нитей состоит в лучшей структуризации программы.

#### Методология применения легковесных процессов

Мы проследили достоинства разных видов легковесных процессов и можем перейти к краткому анализу их недостатков. Многие программисты (включая автора статьи) испытали эти недостатки на себе. При программировании с явным использованием техники легковесных процессов возникает потребность в явной синхронизации по отношению к общим ресурсам. В современных вариантах ОС UNIX имеется несколько разновидностей средств синхронизации: блокировки, семафоры, условные переменные. Но в любом случае, механизм синхронизации является явным, оторванным от ресурса, для доступа к которому производится синхронизация. Если у легковесных процессов одного пользовательского процесса общих ресурсов немного, то такую программу написать и отладить сравнительно несложно. Но при наличии большого количества общих ресурсов отладка программы становится очень сложным делом (даже при использовании только пользовательских нитей).

Основной проблемой является недетерминированность поведения программы, поскольку время выполнения каждого легковесного процесса, вообще говоря, различно при каждом запуске программы. При использовании явных примитивов для синхронизации набора легковесных процессов наиболее распространенной ошибочной ситуацией является возникновение синхронизационных тупиков. Если при отладке параллельной программы возник тупик, нужно исследовать ситуацию, установить причину ее возникновения (как правило, эта причина состоит в несогласованном выполнении синхронизационных примитивов в разных легковесных процессах) и устранить причину тупика. Но по причине недетерминированности поведения программы тупики могут возникать при одном из ста запуске программы, и никогда нельзя быть полностью уверенным, что при некотором сочетании временных характеристик тупик все-таки не проявится. Заметим, что это относится и к LWP, и к пользовательским нитям.

Трудно также согласиться с тем, что использование явных средств распараллеливания улучшает структурность программы. Человеку свойственно последовательное мышление. Для программиста наиболее естественна модель фон Неймана. С другой стороны, было бы странно не использовать возможности мультипроцессоров для повышения скорости выполнения программ. В настоящее время явное использование пользовательских LWP является единственным доступным решением.

Конечно, появление в современных операционных системах механизма процессов, работающих на общей памяти, не является слишком прогрессивным явлением. Традиционный UNIX стимулировал простое и понятное программирование. Современный UNIX провоцирует сложное, запутанное и опасное программирование.

С другой стороны, системные программисты с успехом используют новые средства. При наличии большого желания, времени и денег параллельную программу можно надежно отладить, что демонстрирует большинство компаний, разрабатывающих программное обеспечение баз данных. При этом используются и LWP, поддерживаемые операционной системой, и пользовательские нити. Как кажется (это мнение не только автора), использование LWP безусловно оправдано, если сервер БД конфигурируется в расчете на работу на симметричном мультипроцессоре. На сегодняшний день это единственная возможность реально распараллелить работу сервера. Но совершенно неочевидно, что применение пользовательских нитей при разработке сервера в целях его структуризации (а это делается во многих серверах), является лучшим решением. Известны другие методы структуризации, которые, по меньшей мере, не менее удобны.

### Современные файловые системы

#### Ограничения традиционных файловых систем

При работе с внешней памятью мы по-прежнему в основном имеем дело с магнитными дисками с подвижными головками. Если несколько лет тому назад казалось, что это временное явление, и что наступит такое время, когда электро-механические задержки доступа к внешней памяти скоро исчезнут, то теперь уже понятна устойчивость таких устройств, сопровождаемая постоянным снижением их стоимости. В чем состоит проблема?

На практике наиболее распространена последовательная работа с файлами. Если требуется произвести последовательный просмотр файла, хранящегося на магнитном диске с подвижными головками, то основные задержки создают именно электро-механическом действия, связанные с перемещением магнитных головок. Известно, что в среднем время движения головок на два порядка превышает время двух последующих актов - ожидания подкрутки диска до нужного блока (тоже долго, но не так) и собственно выполнения обмена (это как раз достаточно быстро, и чем дальше, тем быстрее по мере развития технологии магнитных носителей; к сожалению заставить быстро двигаться пакеты магнитных головок не так легко).

Основной неприятностью традиционных файловых систем являлось хаотическое распределение внешней памяти. Один блок внешней памяти файла мог отстоять на произвольное количество цилиндров, т.е. элементов передвижения головок. Коренной перелом произошел в так называемой "быстрой файловой системе", разработанной Кирком МакКусиком в рамках проекта BSD 4.3. МакКусик решил, что лучше головкам магнитного диска двигаться не слишком сильно. Поэтому было введено понятие группы магнитных цилиндров, в пределах которых должен располагаться файл. Вместо произвольного передвижения магнитных головок в масштабе всей поверхности диска для последовательного чтения файла требуется ограниченное смещение головок в выделенной группе цилиндров. Но это не решает всех проблем.

Появились устройства с магнитными дисками, предъявляющие внешнему миру интерфейс с 24 магнитными головками в то время, когда на самом деле (физически) их было 15. И что же теперь оптимизируется? Аппаратура и встроенное программное обеспечение контроллеров магнитных дисков сами производят собственную оптимизацию, а файловая система считает, что все уже оптимизировано. Еще хуже дела стали с появлением дисковых массивов, особенно начиная с пятого уровня организации. RAID обеспечивает надежное хранение данных с 90-процентной гарантией и разделенное хранение блока данных на всех дисках, входящих в массив. Что же теперь оптимизируют операционная система по отношению к доступу к файлам? Это одна из основных проблем современных файловых систем; она не решена, и не ясно, будет ли решена в ближайшее время. Тем не менее, файловые системы развиваются, и мы далее приведем обзор наиболее интересных существующих файловых систем (из мира UNIX) и некоторые примеры перспективных файловых систем.

#### Распространенные файловые системы

Понятие файла является одним из наиболее важных для ОС UNIX. Все файлы, с которыми могут манипулировать пользователи, располагаются в файловой системе, представляющей собой дерево, промежуточные вершины которого соответствуют каталогам, и листья - файлам и пустым каталогам. Реально на каждом логическом диске (разделе физического дискового пакета) располагается отдельная иерархия каталогов и файлов. Для получения общего дерева в динамике используется "монтирование" отдельных иерархий к фиксированной корневой файловой системе.

Замечание: в мире ОС UNIX по историческим причинам термин "файловая система" является перегруженным, обозначая одновременно иерархию каталогов и файлов и часть ядра, которая управляет каталогами и файлами. Видимо, было бы правильнее называть иерархию каталогов и файлов архивом файлов, а термин "файловая система" использовать только во втором смысле. Однако, следуя традиции ОС UNIX, мы будем использовать этот термин в двух смыслах, различая значения по контексту.

Каждый каталог и файл файловой системы имеет уникальное полное имя (в ОС UNIX это имя принято называть full pathname - имя, задающее полный путь, посколько оно действительно задает полный путь от корня файловой системы через цепочку каталогов к соответствующему каталогу или файлу; мы будем использовать термин "полное имя", поскольку для pathname отсутствует благозвучный русский аналог). Каталог, являющийся корнем файловой системы (корневой каталог) в любой файловой системе имеет предопределенное имя "/" (слэш). Полное имя файла, например, /bin/sh означает, что в корневом каталоге должно содержаться имя каталога bin, а в каталоге bin должно содержаться имя файла sh. Коротким или относительным именем файла (relative pathname) называется имя (возможно, составное), задающее путь к файлу начиная от текущего рабочего каталога (существует команда и соответствующий системный вызов, позволяющие установить текущий рабочий каталог.

В каждом каталоге содержатся два специальных имени, имя ".", именующее сам этот каталог и имя "..", именующее "родительский" каталог данного каталога, т.е. каталог, непосредственно предшествующий данному в иерархии каталогов.

UNIX поддерживает многочисленные утилиты, позволяющие работать с файловой системой и доступные как команды командного интерпретатора. Вот некоторые из них (наиболее употребительные):

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

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

В мире UNIX существует несколько разных видов файловых систем со своей структурой внешней памяти. Наиболее известны традиционная файловая система UNIX System V (s5) и файловая система семейства UNIX BSD (ufs). Файловая система s5 состоит из четырех секций (рисунок 6.1(a)). В файловой системе ufs на логическом диске (разделе реального диска) находится последовательность секций файловой системы.

Кратко опишем суть и назначение каждой области диска:

* Boot-блок содержит программу раскрутки, которая служит для первоначального запуска ОС UNIX. В файловых системах s5 реально используется boot-блок только корневой файловой системы. В дополнительных файловых системах эта область присутствует, но не используется.

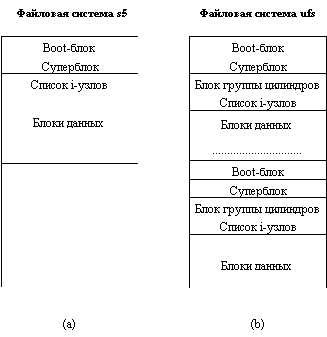


Рис. 6.1.

* Суперблок - это наиболее ответственная область файловой системы, содержащая информацию, которая необходима для работы с файловой системой в целом. Суперблок содержит список свободных блоков и свободные i-узлы (information nodes - информационные узлы). В файловых системах usf для повышения устойчивости поддерживается несколько копий суперблока (как видно из рисунка 6.1(b), по одной копии на группу цилиндров). Каждая копия суперблока имеет размер 8196 байт, и только одна копия суперблока используется при монтировании файловой системы (см. ниже). Однако, если при монтировании устанавливается, что первичная копия суперблока повреждена или не удовлетворяет критериям целостности информации, используется резервная копия.
* Блок группы цилиндров содержит число i-узлов, специфицированных в списке i-узлов для данной группы цилиндров, и число блоков данных, которые связаны с этими i-узлами. Размер блока группы цилиндров зависит от размера файловой системы. Для повышения эффективности файловая система ufs старается выделять i-узлы и блоки данных в одной и той же группе цилиндров.
* Список i-узлов (ilist) содержит список i-узлов, соответствующих файлам данной файловой системы. Максимальное число файлов, которые могут быть созданы в файловой систем, определяется числом доступных i-узлов. В i-узле хранится информация, описывающая файл: режимы доступа к файлу, время создания и последней модификации, идентификатор пользователя и идентификатор группы создателя файла, описание блочной структуры файла и т.д.
* Блоки данных - в этой части файловой системы хранятся реальные данные файлов. В случае файловой системы ufs все блоки данных одного файла пытаются разместить в одной группе цилиндров. Размер блока данных определяется при форматировании файловой системы командой mkfs и может быть установлен в 512, 1024, 2048, 4096 или 8192 байт.

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

Для монтирования файловой системы используется системный вызов mount. Монтирование файловой системы означает следующее. В имеющейся к моменту монтирования дереве каталогов и файлов должен иметься листовой узел - пустой каталог (в терминологии UNIX такой каталог, используемый для монтирования файловой системы, называется directory mount point - точка монтирования). В любой файловой системе имеется корневой каталог. Во время выполнения системного вызова mount корневой каталог монтируемой файловой системы совмещается с каталогом - точкой монтирования, в результате чего образуется новая иерархия с полными именами каталогов и файлов.

Смонтированная файловая система впоследствии может быть отсоединена от общей иерархии с использованием системного вызова umount. Для успешного выполнения этого системного вызова требуется, чтобы отсоединяемая файловая системы к этому моменту не находилась в использовании (т.е. ни один файл из этой файловой системы не был открыт). Корневая файловая система всегда является смонтированной, и к ней не применим системный вызов umount.

Как мы отмечали выше, отдельная файловая система обычно располагается на логическом диске, т.е. на разделе физического диска. Для инициализации файловой системы не поддерживаются какие-либо специальные системные вызовы. Новая файловая система образуется на отформатированном диске с использованием утилиты (команды) mkfs. Вновь созданная файловая система инициализируется в состояние, соответствующее наличию всего лишь одного пустого корневого каталога. Команда mkfs выполняет инициализацию путем прямой записи соответствующих данных на диск.

Ядро ОС UNIX поддерживает для работы с файлами несколько системных вызовов. Среди них наиболее важными являются open, creat, read, write, lseek и close.

Важно отметить, что хотя внутри подсистемы управления файлами обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла как линейной последовательности байтов. Такое представление позволяет использовать абстракцию файла при работе в внешними устройствами, при организации межпроцессных взаимодействий и т.д.

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

open(pathname, oflag [,mode])

Одним из признаков, могущих участвовать в параметре oflag, является признак O\_CREAT, наличие которого указывает на необходимость создания файла, если при выполнении системного вызова open файл с указанным именем не существует (параметр mode имеет смысл только при наличии этого признака). Тем не менее по историческим причинам и для обеспечения совместимости с предыдущими версиями ОС UNIX отдельно поддерживается системный вызов creat, выполняющий практически те же функции.

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

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

Здесь fd - дескриптор файла (полученный при ранее выполненном системном вызове open или creat), buffer - указатель символьного массива и count - число байтов, которые должны быть прочитаны из файла или в него записаны. Значение функции read или write - целое число, которое совпадает со значением count, если операция заканчивается успешно, равно нулю при достижении конца файла и отрицательно при возникновении ошибок.

В каждом открытом файле существует текущая позиция. Сразу после открытия файл позиционируется на первый байт. Другими словами, если сразу после открытия файла выполняется системный вызов read (или write), то будут прочитаны (или записаны) первые count байт содержимого файла (конечно, они будут успешно прочитаны только в том случае, если файл реально содержит по крайней мере count байт). После выполнения системного вызова read (или write) указатель чтения/записи файла будет установлен в позицию count+1 и т.д.

Такой чисто последовательный стиль работы оказывается во многих случаях достаточным, но часто бывает необходимо читать или изменять файл с произвольной позиции (например, как без такой возможности хранить в файле прямо индексируемые массивы данных?). Для явного позиционирования файла служит системный вызов

lseek(fd, offset, origin)

Как и раньше, здесь fd - дескриптор ранее открытого файла. Параметр offset задает значение относительного смещения указателя чтения/записи, а параметр origin указывает, относительно какой позиции должно применяться смещение. Возможны три значения параметра origin. Значение 0 указывает, что значение offset должно рассматриваться как смещение относительно начала файла. Значение 1 означает, что значение offset является смещением относительно текущей позиции файла. Наконец, значение 2 говорит о том, что задается смещение относительно конца файла. Заметим, что типом данных параметра offset является long int. Это значит, что, во-первых, могут задаваться достаточно длинные смещения и, во-вторых, смещения могут быть положительными и отрицательными.

Например, после выполнения системного вызова

lseek(fd, 0, 0)

указатель чтения/записи соответствующего файла будет установлен на начало (на первый байт) файла. Системный вызов

lseek(fd, 0, 2)

установит указатель на конец файла. Наконец, выполнение системного вызова

lseek(fd, 10, 1)

приведет к увеличению текущего значения указателя на 10.

Естественно, системный вызов успешно завершается только в том случае, когда заново сформированное значение указателя не выходит за пределы существующих размеров файла.

#### Файловые системы с журнализацией

Одним из наиболее существенных недостатков традиционных файловых систем является трудность восстановления согласованного состояния файлов после сбоев, в результате которых теряется содержимое основной памяти. Это связано с тем, что для повышения эффективности работа с внешней памятью производится через буфера основной памяти, в которых в момент сбоя могут находиться как данные (содержимое блоков файла), так и метаданные (например, содержимое i-узлов). Обычным приемом восстановления файловой системы после сбоя является применение утилиты fsck, которая, работая на уровне логического диска, обходит всю файловую систему и, по мере возможности, находит и исправляет ошибочные ситуации. Естественно, если используются большие диски, то эта работа занимает много времени, приводя к серьезным задержкам в использовании компьютера. Для устранения этого недостатка все чаще применяются файловые системы с журнализацией.

Как и в случае журнализации изменений в системах управления базами данных, основным принципом журнализующих файловых систем является поддержание специального файла-журнала, в который в последовательном и только последовательном режиме записывается информация обо всех изменениях файловой системы. Запись, как правило, производится порциями большого объема, что обеспечивает высокий уровень полезного использования дисковой памяти и высокую эффективность. При восстановлении после сбоя требуется использовать только "хвост" журнала, что позволяет производить восстановление быстро и надежно.

Файловые системы с журнализацией разделяются на две категории: системы, производящие журнализацию всех изменений, и системы, журнализующие только изменения метаданных. Среди систем второй категории выделяются те, которые журнализуют только некоторую выделенную информацию (например, не помещают в журнал информацию о смене владельца файла).

Различаются подходы с журнализацией операций и журнализацией результатов операций. Если, например, журнализуется операция изменения таблицы распределения памяти на диске, то выгоднее поместить в журнал информацию о самой операции (поскольку она изменяет только несколько бит информации на диске). В случае же журнализации операции записи блока данных выгоднее занести в журнал все содержимое блока до его изменения.

Среди файловых систем с журнализацией выделяются такие, в которых журнал используется как вспомогательное средство, а структура самой файловой системы не меняется (в частности, как и в традиционных файловых системах, поддерживаются структуры i-узлов и суперблоков). Другой класс журнализующих файловых систем составляют те, в которых журнал является единственным средством представления файлов на магнитном диске.

Имеется два типа журналов: журнал, ориентированный только на повторное выполнение операций (redo-only), и журнал, способный поддерживать как повторное выполнение операций, так и их обратное выполнение (undo-redo). В журнале "undo-redo" сохраняются как новые, так и старые значения данных. При использовании журнала типа "redo-only" операции восстановления упрощаются, но требуется ограничивать порядок записи метаданных в журнал и на место их постоянного хранения. Журнал "undo-redo" больше по объему и требует применения более сложного механизма журнализации, но использование этого типа журнализации допускает более высокий уровень параллельности.

Хотя имеются файловые системы, производящие архивизацию наиболее старых порций журнала, наиболее распространенным подходом является использование циклического файла-журнала конечного размера. Для поддержки такого журнала применятся специализированная процедура "сборки мусора", выявляющая "устарелые" порции журнальной информации. В некоторых системах эта процедура запускается на фоне работающей системы, в других - только при проведении профилактических работ.

Как упоминалось выше, для лучшего использования дисковой памяти и увеличения эффективности записи в журнал производятся порциями большого объема. В результате часто в одну физическую запись пакуются несколько логических записей об изменении файловой системы. Естественно, это снижает надежность файловой системы, поскольку в случае сбоя последний буфер с журнальной информацией будет утрачен. На практике приходится делать выбор между эффективностью и надежностью.

Наиболее известной файловой системой, основанной исключительно на журнализации и журнализующей все изменения, является BSD-LFS (UNIX BSD 4.4 Log-Structured File System). Среди файловых систем, поддерживающих журнализацию только метаданных, можно выделить Cedar, Calaveras и Veritas.