ТЕМА 3.3. УПРАВЛЕНИЕ ВВОДОМ-ВЫВОДОМ И ФАЙЛАМИ

В данной теме рассматриваются следующие вопросы:

- Типы устройств ввода/вывода.
- Обработка аппаратных прерываний.
- Синхронный и асинхронный ввод/вывод.
- Классификация функций ввода-вывода.
- Буферизация операций ввода-вывода.
- Дисковое планирование.
- Файловые системы.
- Файлы и директории.
- Управление внешней памятью.
- Кеширование в операционной системе.

Лекции – 2 часа, лабораторные занятия – 2 часа, самостоятельная работа – 2 часа.

Экзаменационные вопросы по теме:

- Типы устройств ввода/вывода. Обработка внешних прерываний. Синхронный и асинхронный ввод/вывод.
- Файловые системы. Файлы и директории. Управление внешней памятью.

3.3.1. Типы устройств ввода/вывода

Операционная система управляет всеми устройствами ввода-вывода, подключенными к компьютеру: выдает команды устройствам, перехватывает прерывания и обрабатывает ошибки. Также она предоставляет простой и легкий в использовании интерфейс между устройствами и всей остальной системой [1].

Устройства ввода-вывода можно условно разделить на две категории: блочные устройства и символьные устройства. К блочным относятся такие устройства, которые хранят информацию в блоках фиксированной длины, у каждого из которых есть собственный адрес. Обычно размеры блоков варьируются от 512 до 65 536 байт. Вся передача данных ведется пакетами из одного или нескольких целых (последовательных) блоков. Важным свойством блочного устройства является то, что оно способно читать или записывать каждый блок независимо от всех других блоков. Среди наиболее распространенных блочных устройств жесткие диски, приводы оптических дисков и флеш-накопители USB.

Другой тип устройств ввода-вывода — символьные устройства. Они выдают или воспринимают поток символов, не относящийся ни к какой блочной структуре. Они не являются адресуемыми и не имеют никакой операции позиционирования. В качестве символьных устройств могут рассматриваться терминалы, принтеры, сетевые интерфейсы, мыши (в качестве устройства-указателя) и множество других устройств, не похожих на дисковые устройства.

Некоторые устройства не подпадают под эту классификацию, например, часы и сенсорные экраны. Тем не менее модель блочных и символьных устройств является достаточно общей для того, чтобы использовать ее в качестве основы для придания части программного обеспечения операционной системы независимости от устройства ввода-вывода.

Устройства ввода-вывода зачастую состоят из механической и электронной составляющих. Зачастую эти две составляющие удается разделить, чтобы получить модульную конструкцию и придать устройству более общий вид. Электронный компонент называется контроллером устройства, или адаптером. На персональных компьютерах он часто присутствует в виде микросхемы на системной плате или печатной платы, вставляемой в слот расширения (PCIe). Механический компонент представлен самим устройством.

На плате контроллера обычно имеется разъем, к которому может быть подключен кабель, ведущий непосредственно к самому устройству. Многие контроллеры способны управлять двумя, четырьмя или даже восемью одинаковыми устройствами. Если интерфейс между контроллером и устройством подпадает под какой-нибудь стандарт, будь то один из официальных стандартов ANSI, IEEE или ISO или же один из ставших де-факто стандартов, то компании могут производить контроллеры или устройства, соответствующие этому интерфейсу. К примеру, многие компании производят дисковые приводы, соответствующие интерфейсу SATA, SCSI, USB, Thunderbolt или FireWire (IEEE 1394).

Задача контроллера состоит в преобразовании последовательного потока битов в блок байтов и коррекции ошибок в случае необходимости. Блок байтов обычно проходит первоначальную побитовую сборку в буфере, входящем в состав контроллера. После проверки контрольной суммы блока и объявления его не содержащим ошибок он может быть скопирован в оперативную память.

У каждого контроллера для связи с центральным процессором имеется несколько регистров. Путем записи в эти регистры операционная система может давать устройству команды на предоставление данных, принятие данных, включение, выключение или выполнение каких-нибудь других действий. Считывая данные из этих регистров,

операционная система может узнать о текущем состоянии устройства, о том, готово ли оно принять новую команду, и т. д.

В дополнение к регистрам управления у многих устройств имеется буфер данных, из которого операционная система может считывать данные и в который она может их записывать. Например, наиболее распространенный способ отображения компьютерами пикселов на экране предусматривает наличие видеопамяти, которая по сути является буфером данных, куда программы или операционная система могут вести запись.

Для доступа к этим регистрам используются два подхода.

1. Каждый регистр получает уникальный номер порта в пространстве ввода-вывода.
2. Регистры отображаются на пространство памяти.

Возможен гибридный вариант.

Эти две схемы обращения к контроллерам имеют свои достоинства и недостатки.

Преимущества отображения на пространство памяти.

- Для обращения к портам ввода-вывода нужно использовать ассемблерные команды, в то время как при втором подходе можно использовать обычные переменные в языках высокого уровня.
- Легко обеспечить защиту от осуществления ввода-вывода со стороны пользовательских процессов.
- Любая команда, которая может обращаться к памяти, может также обращаться и к регистрам управления (иначе регистр управления сначала должен быть считан в центральный процессор, а затем проверен, для чего потребуются две команды вместо одной).

Недостатки отображения на пространство памяти.

Кэширование регистров управления устройством может привести к пагубным последствиям. Аппаратура должна иметь возможность выборочного отключения кэширования, например, на постраничной основе. Это свойство приводит к дополнительному усложнению как аппаратного обеспечения, так и операционной системы, которым приходится управлять избирательным кэшированием.

Если используется только одно адресное пространство, то все модули памяти и все устройства ввода-вывода должны проверять все обращения к памяти, чтобы понять, кому из них следует отвечать. При использовании выделенной высокоскоростной шины памяти устройства ввода-вывода не могут увидеть адреса памяти, выставляемые процессором на эту шину, следовательно, они не могут реагировать на эти адреса (рис. 3.3.1). Поэтому, чтобы заставить отображаемый на память ввод-вывод работать на системе с несколькими шинами, нужно предпринять какие-то специальные меры.

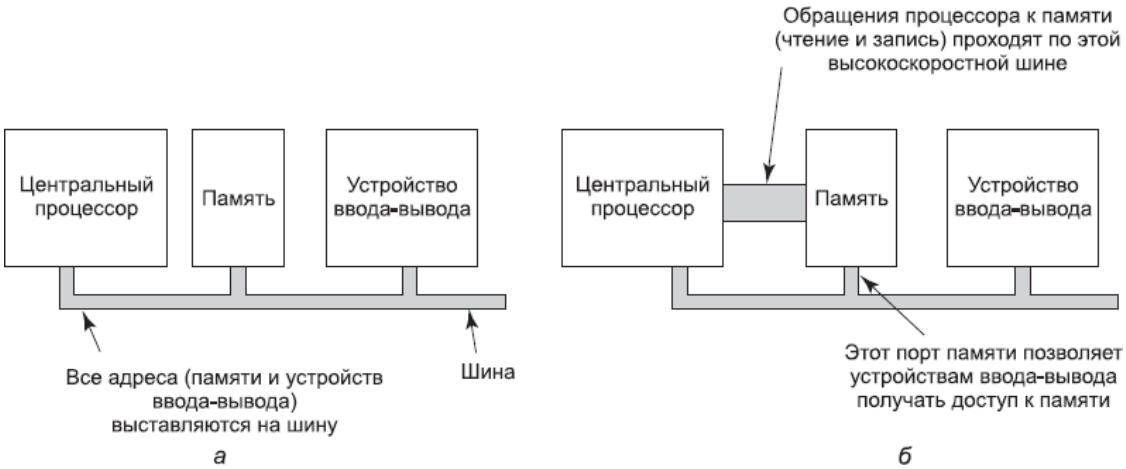

Рис. 3.3.1. Архитектура: а — использующая одну шину; б — с двойной шиной памяти

Независимо от наличия или отсутствия у центрального процессора ввода-вывода, отображаемого на пространство памяти, ему необходимо обращаться к контроллерам устройств, чтобы осуществлять с ними обмен данными. Центральный процессор может запрашивать данные у контроллера ввода-вывода побайтно, но при этом будет нерационально расходоваться его рабочее время, поэтому чаще всего используется другая схема, которая называется прямым доступом к памяти (Direct Memory Access (DMA)).

Режимы работы DMA:

- **Пакетный режим**. Весь блок данных передается в одной непрерывной последовательности. ЦП может быть неактивным на относительно длительные периоды времени.
- **Режим похищения цикла**. Транзакции DMA и ЦП чередуются.
- **Прозрачный режим**. Контроллер DMA передает данные только тогда, когда ЦП выполняет операции, не использующие системные шины.

Удалённый прямой доступ к памяти (remote direct memory access, RDMA) — аппаратное решение для обеспечения прямого доступа к оперативной памяти другого компьютера при помощи высокоскоростной сети. Такой доступ позволяет получить доступ к данным, хранящимся в удалённой системе без привлечения средств операционных систем обоих компьютеров. Является методом пересылки данных с высокой пропускной способностью и низкой задержкой сигнала, и особенно полезен в больших параллельных вычислительных системах — кластерах.

RDMA реализован в различных протоколах, например, в Virtual Interface Architecture, InfiniBand, iWARP, RoCE, Intel Omni-Path.

Аппаратная реализация RDMA позволяет реализовать метод zero-copy для сетей. При передаче данных с помощью RDMA исключаются лишние копирования между приложением и буферами операционной системы; соответственно, снижается объём работы центрального процессора, нагрузка на кэш-память, уменьшается количество переключений контекста, а сами передачи могут производиться одновременно с другой полезной работой. Когда приложение исполняет запрос на чтение или запись в удалённую оперативную память, данные могут доставляться напрямую в сетевой адаптер, уменьшая задержки при передаче данных.

Ввод-вывод в системе Linux

Система ввода-вывода в Linux довольно проста и не отличается от присущих другим UNIX-системам. Как правило, все устройства ввода-вывода выглядят как файлы и доступ к ним осуществляется с помощью тех же системных вызовов **`read`** и **`write`**, которые используются для доступа ко всем обычным файлам. В некоторых случаях должны быть заданы параметры устройства — это делается при помощи специального системного вызова.

В операционной системе Linux все устройства интегрируются в файловую систему в виде так называемых специальных файлов (special files). Каждому устройству ввода-вывода назначается путь (обычно в каталоге `/dev`). Например, диск может иметь путь `/dev/hd1`, у принтера может быть путь `/dev/lp`, а у сети — `/dev/net`.

Доступ к этим специальным файлам осуществляется так же, как и к любым другим файлам. Для этого не требуется никаких специальных команд или системных вызовов. Вполне подойдут обычные системные вызовы **`read`** и **`write`**. Например, команда

```
cp file /dev/lp
```

скопирует файл `file` на принтер, в результате чего этот файл будет распечатан (при условии, что у пользователя есть разрешение на доступ к `/dev/lp`).

Специальные файлы подразделяются на две категории: блочные и символьные. Блочный специальный файл (block special file) состоит из последовательности пронумерованных блоков. Основное свойство блочного специального файла заключается в том, что к каждому его блоку можно адресоваться и получить доступ отдельно. Блочные специальные файлы обычно используются для дисков.

Символьные специальные файлы (character special files) обычно используются для устройств ввода или вывода символьного потока. Символьные специальные файлы используются такими устройствами, как клавиатуры, принтеры, сети, мыши, плоттеры и т. д.

С каждым специальным файлом связан драйвер устройства, осуществляющий работу с соответствующим устройством. Каждый драйвер разделен на две части, причем обе они являются частью ядра Linux и работают в режиме ядра. Верхняя часть драйвера работает в контексте вызывающей стороны и служит интерфейсом к остальной системе Linux. Нижняя часть работает в контексте ядра и взаимодействует с устройством. Драйверам разрешается делать вызовы процедур ядра для выделения памяти, управления таймером, управления DMA и т. д. Набор функций ядра, которые они могут вызывать, определен в документе под названием «Интерфейс драйвер — ядро» (Driver-Kernel Interface).

Хотя большую часть операций ввода-вывода можно выполнить при помощи соответствующего файла, иногда возникает необходимость в обращении к неким специфическим устройствам. До принятия стандарта POSIX в большинстве версий систем UNIX был системный вызов ioctl, выполнявший со специальными файлами большое количество операций, специфических для различных устройств. С течением времени все это привело к путанице. В стандарте POSIX здесь был наведен порядок, для чего функции системного вызова ioctl были разбиты на отдельные функциональные вызовы, главным образом для управления терминалом. В системе Linux и современных UNIX-системах только от конкретной реализации зависит, является ли каждый из них отдельным системным вызовом или они все вместе используют один системный вызов.

Чтобы минимизировать латентность перемещений дисковых головок, Linux использует планировщик ввода-вывода (I/O scheduler). Цель планировщика — переупорядочить или собрать в пакеты запросы ввода-вывода к блочным устройствам.

В дополнение к обычным дисковым файлам существуют также блочные специальные файлы (называемые также сырыми блочными файлами — raw block files). Эти файлы позволяют программам обращаться к диску при помощи абсолютных номеров блоков (помимо файловой системы). Чаще всего они используются для таких вещей, как подкачка и обслуживание системы.

Ввод-вывод в Windows

Операционная система узнает информацию о подключённых устройствах с помощью диспетчера Plug-and-Play. Диспетчер Plug-and-Play отправляет запрос на каждый разъем шины и просить установленное в нем устройство идентифицироваться. Обнаружив, что именно там установлено, диспетчер Plug-and-Play назначает аппаратные ресурсы (такие, как уровни прерываний), находит соответствующие драйверы и загружает их в память.

В операционной системе Windows все файловые системы, антивирусные фильтры и даже службы ядра (которым не соответствуют никакие аппаратные средства) реализованы при помощи драйверов ввода-вывода. В системной конфигурации должно быть указание на необходимость загрузки хотя бы некоторых из этих драйверов, поскольку не существует устройства-счетчика на шине. Другие (подобно файловым системам) загружаются специальным кодом, который распознает их необходимость, — например, распознаватель файловых систем обращается к тому и определяет, файловую систему какого типа он содержит.

Еще один аспект Windows — это ее поддержка асинхронного ввода-вывода. Поток может начать операцию ввода-вывода, а затем продолжить выполнение параллельно с вводом-выводом. Эта функциональная возможность особенно важна для серверов. Есть несколько способов, при помощи которых поток может определить завершение ввода-вывода. Один из них — указать объект события в момент выполнения вызова и затем дожидаться его. Другой — указать очередь, в которую система поместит событие о завершении ввода-вывода. Третий — предоставить процедуру обратного вызова, которую система вызовет после завершения ввода-вывода. Четвертый — опросить адрес памяти, который диспетчер ввода-вывода обновляет после завершения ввода-вывода.

Интерфейсы системных вызовов, предоставляемые диспетчером ввода-вывода, не очень отличаются от предлагаемых большинством других операционных систем. Основные операции: **`open`**, **`read`**, **`write`**, **`ioctl`** и **`close`**, но есть и другие операции: Plug-and-Play, управления энергопотреблением, установки параметров, сброса системных буферов и т. д. На уровне Win32 эти API заключаются в оболочку интерфейсов, которые предоставляют операции более высокого уровня (специфичные для конкретных устройств).

На нижнем уровне эти оболочки открывают устройства и выполняют основные операции. Даже некоторые операции метаданных (такие, как переименование файла) реализованы без специфичных системных вызовов. Они просто используют специальную версию операции **`ioctl`**.

Система ввода-вывода в Windows состоит из служб Plug-and-Play, диспетчера электропитания, менеджера ввода-вывода, а также модели драйвера устройств. Plug-and-Play обнаруживает изменения в конфигурации аппаратного обеспечения, создает (или уничтожает) стеки устройств (для каждого устройства), а также загружает и выгружает драйверы устройств. Диспетчер электропитания настраивает состояние электропитания устройств ввода-вывода, чтобы уменьшить потребление энергии системой, когда устройства не используются. Диспетчер ввода-вывода предоставляет поддержку манипулирования объектами ядра для ввода-вывода, а также операций типа **`IoCallDrivers`** и **`IoCompleteRequest`**. Однако большая часть работы по поддержке ввода-вывода в Windows реализована в самих драйверах устройств.

Чтобы гарантировать, что драйверы устройств хорошо работают с Windows, компания Microsoft описала модель **WDM** (Windows Driver Model), которой должны соответствовать драйверы устройств. Существует набор разработчика (Windows Driver Kit), который содержит документацию и примеры, помогающие создавать драйверы, соответствующие WDM. Большинство драйверов Windows начинается с копирования подходящего образцового драйвера из WDK и его модификации создателем нового драйвера.

Устройства в Windows представлены объектами устройств. Объекты устройств используются для представления аппаратных средств (таких, как шины), а также как программные абстракции (наподобие файловых систем, сетевых протоколов и расширений ядра вроде антивирусных драйверов-фильтров). Все они формируют то, что Windows называет стеком устройств (рис. 3.3.2).

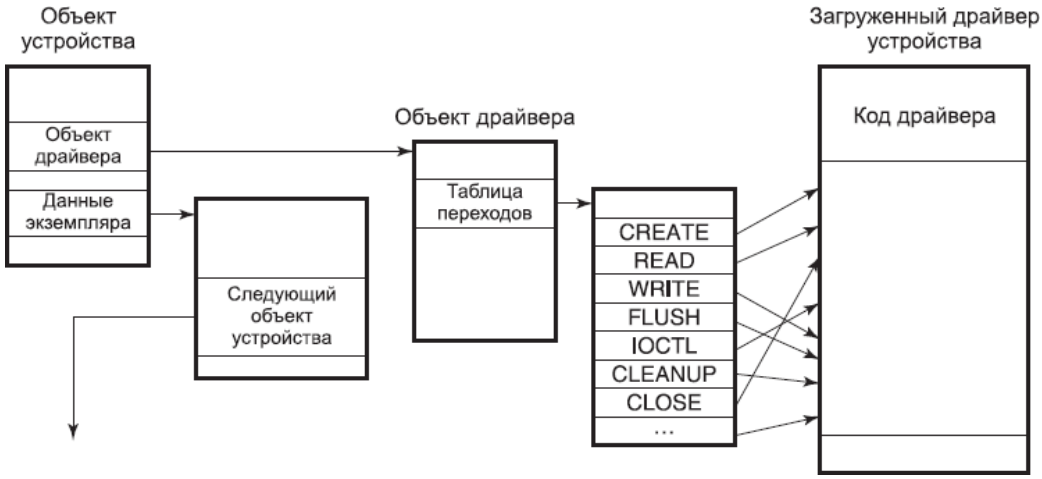

Рис. 3.3.2. Один уровень в стеке устройств

Для каждой операции драйвер должен указать точку входа. **`IoCallDriver`** берет тип операции из IRP, использует объект устройства на текущем уровне стека устройств (для поиска объекта драйвера) и ищет (по типу операции) в таблице переходов для драйверов соответствующую точку входа в драйвер. Затем драйвер вызывается, и ему передается объект устройства и IRP.

После того как драйвер завершил обработку представленного пакетом IRP запроса, у него есть три варианта. Он может еще раз вызвать **`IoCallDriver`**, передав ему IRP и следующий объект устройства в стеке устройств. Он может объявить запрос ввода-вывода завершенным и вернуться к его вызывающей стороне. Либо он может поставить IRP во внутреннюю очередь и вернуться к его вызывающей стороне (объявив, что запрос ввода-вывода еще не завершен). В последнем случае получается операция асинхронного ввода-вывода — по крайней мере, если все драйверы выше по стеку с этим соглашаются и также возвращаются к своим вызывающим сторонам.

На рис. 3.3.3 показаны основные поля IRP. По существу, IRP — это массив (с динамически изменяемым размером), содержащий поля, которые могут быть использованы любым драйвером (для обрабатывающего запрос стека устройств). Эти стековые поля позволяют драйверу указать процедуру, которую нужно вызвать при завершении запроса ввода-вывода. При завершении все уровни стека устройств проходятся в обратном порядке, при этом по очереди вызываются все процедуры завершения (для всех драйверов). На каждом уровне драйвер может завершить запрос или принять решение, что еще есть некая работа (которую нужно сделать), и оставить запрос незавершенным (отложив на данный момент завершение ввода-вывода).

Рис. 3.3.3. Основные поля пакета IRP

IRP содержит флаги, код операции для поиска по таблице переходов, указатели для пользовательского буфера и буфера ядра, а также список MDL (Memory Descriptor Lists), которые используются для описания представленных буферами физических страниц (то есть для операций DMA). Имеются также поля для операций отмены и завершения.

3.3.2. Обработка аппаратных прерываний

На аппаратном уровне прерывания работают следующим образом [2]. Когда устройство ввода-вывода заканчивает свою работу, оно инициирует прерывание (при условии, что прерывания разрешены операционной системой). Для этого устройство выставляет сигнал на выделенную устройству специальную линию шины. Этот сигнал распознается микросхемой контроллера прерываний, расположенной на материнской плате. Контроллер прерываний принимает решение о дальнейших действиях.

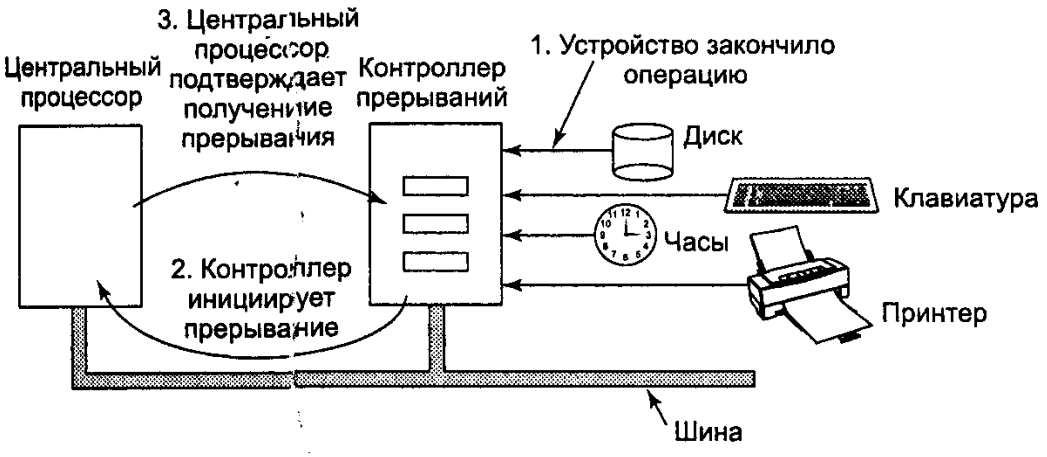

Рис. 3.3.4. Общая схема обработки аппаратного прерывания

При отсутствии других необработанных запросов прерывания контроллер прерываний обрабатывает прерывание немедленно. Если прерывание уже обрабатывается, и в это время приходит запрос от другого устройства по линии с более низким приоритетом, то новый запрос просто игнорируется. В этом случае устройство продолжает удерживать сигнал прерывания на шине до тех пор, пока оно не будет обслужено центральным процессором.

Для обработки прерывания контроллер выставляет на адресную шину номер устройства, требующего к себе внимания, и устанавливает сигнал прерывания на соответствующий контакт процессора.

Этот сигнал заставляет процессор приостановить текущую работу и начать выполнять обработку прерывания. Номер, выставленный на адресную шину, используется в качестве индекса в таблице, называемой вектором прерываний, из которой извлекается новое значение счетчика команд. Новый счетчик команд указывает на начало соответствующей процедуры обработки прерывания. Обыч-но с этого места аппаратные и эмулированные прерывания используют один и тот же механизм и часто пользуются одним и тем же вектором. Расположение вектора может быть либо жестко прошито на аппаратном уровне, либо, наоборот, располагаться в произвольном месте памяти, на которое указывает специальный регистр процессора, загружаемый операционной системой.

Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, записывая определенное значение в порт контроллера прерываний. Это подтверждение разрешает контроллеру издавать новые прерывания. Благодаря тому, что центральный процессор откладывает выдачу подтверждения до

момента, когда он уже готов к обработке нового прерывания, удается избежать ситуации состязаний при появлении почти одновременных прерываний от нескольких устройств.

Аппаратура всегда, прежде чем начать процедуру обработки прерывания, со-храняет определенную информацию. Сохраняемая информация и место ее хра-нения широко варьируются в зависимости от центрального процессора. Как минимум сохраняется счетчик команд, что позволяет продолжить выполнение прерванного процесса. Другая крайность представляет собой сохранение всех программно доступных регистров и большого количества внутренних регистров центрального процессора.

Место сохранения этой информации также оказывается проблемой. Один из вариантов состоит в том, чтобы сохранять эти данные в неких внутренних регист-рах, доступных операционной системе. Недостаток так ого подхода — до тех пор, пока вся сохраненная информация не будет считана обработчиком прерываний, новые прерывания будет нельзя разрешать. В противном случае любое новое прерывание просто стерло бы всю сохраненную таким образом информацию, записав поверх ее новые данные. В результате прерывания оказываются запрещенными в течение довольно длительных интервалов времени, что приводит к возможному игнорированию некоторых сигналов прерывания от устройств и, соответственно, к возможной потере данных.

Обслуживание прерывания в Windows

Обслуживание прерывания состоит из двух, а иногда и трех этапов [3]:

1. Быстрое сохранение непостоянной информации (например, регистр содержимого) в процедуре прерывания, которая выполняется в IRQL = DIRQL.
2. Обработка сохраненных переменных данных в отложенном вызове процедуры (DPC), который выполняется в IRQL = DISPATCH_LEVEL.
3. Выполнение дополнительных работ в IRQL = PASSIVE_LEVEL, если это необходимо.

Когда устройство создает аппаратное прерывание, платформа вызывает подпрограмму обслуживания прерываний драйвера (ISR), которую драйверы на основе платформы реализуют как функцию обратного вызова **`EvtInterruptIsr`**.

Функция обратного вызова **`EvtInterruptIsr`**, которая выполняется в DIRQL устройства, должна быстро сохранять сведения об прерывании, такие как содержимое регистра, которые будут потеряны в случае другого прерывания.

Как правило, функция обратного вызова **`EvtInterruptIsr`** планирует отложенный вызов процедуры (DPC) для обработки сохраненных сведений позже в более низком IRQL (DISPATCH_LEVEL). Драйверы на основе платформы реализуют подпрограммы DPC как функции обратного вызова **`EvtInterruptDpc`** или **`EvtDpcFunc`**.

Большинство драйверов используют одну функцию обратного вызова **`EvtInterruptDpc`** для каждого типа прерывания. Чтобы запланировать выполнение функции обратного вызова **`EvtInterruptDpc`**, драйвер должен вызвать **`WdfInterruptQueueDpcForIsr`** из функции обратного вызова **`EvtInterruptIsr`**.

Если драйвер создает несколько объектов очередей платформы для каждого устройства, можно использовать отдельный объект DPC и функцию обратного вызова **EvtDpcFunc** для каждой очереди. Чтобы запланировать выполнение функции обратного вызова **`EvtDpcFunc`**, драйвер должен сначала создать один или несколько объектов DPC, вызвав **`WdfDpcCreate`**, обычно в функции обратного вызова **`EvtDriverDeviceAdd`** драйвера. Затем функция обратного вызова **`EvtInterruptIsr`** драйвера может вызвать **`WdfDpcEnqueue`**.

Драйверы обычно выполняют запросы ввода-вывода в функциях обратного вызова **`EvtInterruptDpc`** или **`EvtDpcFunc`**.

3.3.3. Синхронный и асинхронный ввод/вывод

Асинхронный ввод-вывод используется там, где можно оптимизировать производительность приложения. При асинхронном вводе-выводе приложение инициирует операцию ввода-вывода, а затем может продолжить свою работу (во время выполнения этого запроса). При синхронном вводе-выводе приложение блокируется до завершения выполнения операции ввода-вывода.

С точки зрения вызывающего потока асинхронный ввод-вывод более эффективен, поскольку позволяет продолжать выполнение, в то время как операция ввода-вывода ставится диспетчером ввода-вывода в очередь и впоследствии выполняется. Однако приложение, использующее асинхронный ввод-вывод, требует механизма определения завершенности этой операции.

По умолчанию все файловые дескрипторы в **Unix-системах** создаются в «блокирующем» режиме. Это означает, что системные вызовы для ввода-вывода, такие как **`read`**, **`write`** или **`connect`**, могут заблокировать выполнение программы вплоть до готовности результата операции [4].

Легче всего понять это на примере чтения данных из потока **`stdin`** в консольной программе. Как только вы вызываете **`read`** для **`stdin`**, выполнение программы блокируется, пока данные не будут введены пользователем с клавиатуры и затем прочитаны системой.

То же самое происходит при вызове функций стандартной библиотеки, таких как **`fread`**, **`getchar`**, **`std::getline`**, поскольку все они в конечном счёте используют системный вызов **`read`**. Если говорить конкретнее, ядро погружает процесс в спящее состояние, пока данные не станут доступны в псевдо-файле **`stdin`**.

То же самое происходит и для любых других файловых дескрипторов. Например, если вы пытаетесь читать из TCP-сокета, вызов **`read`** заблокирует выполнение, пока другая сторона TCP-соединения не пришлёт ответные данные.

Блокировки - это проблема для всех программ, требующих конкурентного выполнения, поскольку заблокированные потоки процесса засыпают и не получают процессорное время. Существует два различных, но взаимодополняющих способа устранить блокировки:

- неблокирующий режим ввода-вывода
- мультиплексирование с помощью системного API, такого как **`select`** либо **`epoll`**

Эти решения часто применяются совместно, но предоставляют разные стратегии решения проблемы.

Файловый дескриптор помещают в "неблокирующий" режим, добавляя флаг O_NONBLOCK к существующему набору флагов дескриптора с помощью fcntl:

```
/* Добавляем флаг O_NONBLOCK к дескриптору fd */
const int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

С момента установки флага дескриптор становится неблокирующим. Любые системные вызовы для ввода-вывода, такие как **`read`** и **`write`**, в случае неготовности данных в момент вызова ранее могли бы вызвать блокировку, а теперь будут возвращать -1 и глобальную переменную errno устанавливать в EWOULDBLOCK. Это интересное изменение поведения, но само по себе мало полезное: оно лишь является базовым примитивом для построения эффективной системы ввода-вывода для множества файловых дескрипторов.

Такой подход работает, но имеет свои минусы:

- Если данные приходят очень медленно, программа будет постоянно просыпаться и тратить впустую процессорное время

- Когда данные приходят, программа, возможно, не прочитает их сразу, т.к. выполнение приостановлено из-за **nanosleep**
- Увеличение интервала сна уменьшит бесполезные траты процессорного времени, но увеличит задержку обработки данных
- Увеличение числа файловых дескрипторов с таким же подходом к их обработке увеличит долю расходов на проверки наличия данных

Для решения этих проблем операционная система предоставляет мультиплексирование ввода-вывода.

Существует несколько мультиплексирующих системных вызовов:

- Вызов **select** существует во всех POSIX-совместимых системах, включая Linux и MacOSX
- Группа вызовов **epoll_*** существует только на Linux
- Группа вызовов kqueue существует на FreeBSD и других *BSD

Все три варианта реализуют единый принцип: делегировать ядру задачу по отслеживанию прихода данных для операций чтения/записи над множеством файловых дескрипторов. Все варианты могут заблокировать выполнение, пока не произойдёт какого-либо события с одним из дескрипторов из указанного множества. Например, вы можете сообщить ядру ОС, что вас интересуют только события чтения для файлового дескриптора X, события чтения-записи для дескриптора Y, и только события записи - для Z.

Все мультиплексирующие системные вызовы, как правило, работают независимо от режима файлового дескриптора (блокирующего или неблокирующего). Программист может даже все файловые дескрипторы оставить блокирующими, и после **select** либо **epoll** возвращённые ими дескрипторы не будут блокировать выполнение при вызове **read** или **write**, потому что данные в них уже готовы.

Группа вызовов **epoll** является наиболее развитым мультиплексером в ядре Linux и способна работать в двух режимах:

- level-triggered - похожий на **select** упрощённый режим, в котором файловый дескриптор возвращается, если остались непрочитанные данные
    - если приложение прочитало только часть доступных данных, вызов **epoll** вернёт ему недопрочитанные дескрипторы
- edge-triggered - файловый дескриптор с событием возвращается только если с момента последнего возврата **epoll** произошли новые события (например, пришли новые данные)
    - если приложение прочитало только часть доступных данных, в данном режиме оно всё равно будет заблокировано до прихода каких-либо новых данных

Win32 API предоставляет четыре различных способа извещения о завершении ввода-вывода.

Сигнал объекту устройства ядра. При завершении операции ввода-вывода устанавливается индикатор, связанный с объектом устройства. Поток, вызвавший операцию ввода-вывода, может продолжить свое выполнение до тех пор, пока не достигнет точки, в которой он должен дождаться завершения выполнения операции ввода-вывода. В этой точке поток может находиться в состоянии ожидания до завершения операции ввода-вывода, после чего продолжить свою работу. Эта технология проста и легка в использовании, но не подходит для обработки множественных запросов ввода-вывода. Например, если потоку необходимо выполнить множество одновременных операций над одним файлом (таких, как чтение одной и за-пись другой части в файл), то при описанной методике поток будет не в состоянии

отличить завершение операции чтения от завершения операции записи. Он будет просто знать о том, что завершена некоторая операция ввода-вывода для этого файла.

Сигнал объекту события ядра. Эта методика допускает одновременные запросы ввода-вывода к единственному устройству или файлу. Для каждого запроса поток создает событие; позже поток может ожидать завершения одного из этих запросов (или завершения серии запросов).

Оповещение о вводе-выводе. При этом методе используется очередь, связанная с потоком и известная как очередь вызовов асинхронных процедур (asynchronous procedure call — APC). Поток создает запросы ввода-вывода, а диспетчер ввода-вывода размещает результаты этих запросов в очереди APC вызывающего потока.

Порты завершения ввода-вывода. Эта технология используется для оптимизации использования потоков. По сути, для использования доступен пул потоков, так что нет необходимости в создании нового потока для обработки нового запроса.

3.3.4. Классификация функций ввода-вывода

Таблица 3.3.1. Некоторые из файловых операций, поддерживаемых для типичных символьных устройств

Устройство	**Open**	**Close**	**Read**	**Write**	**Ioctl**	**Other**
Память	null	null	mem_read	mem_write	null	
Клавиатура	k_open	k_close	k_read	error	k_ioctl	
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	
Принтер	lp_open	lp_close	error	lp_write	lp_ioctl	

Для создания нового файла можно использовать системный вызов **creat**. В качестве параметров этому системному вызову следует задать имя файла и режим защиты. Так, команда

```
fd = creat("abc", mode);
```

создает файл abc с режимом защиты, указанным в `mode`. Биты `mode` определяют круг пользователей, которые могут получить доступ к файлу, а также уровень предоставляемого им доступа [1].

Системный вызов **creat** не только создает новый файл, но и открывает его для записи. Чтобы последующие системные вызовы могли получить доступ к файлу, успешный системный вызов **creat** возвращает небольшое неотрицательное целое число, называемое дескриптором файла (file descriptor) (fd в приведенном ранее примере). Если системный вызов выполняется с уже существующим файлом, то длина этого файла уменьшается до 0, а все его содержимое теряется. Файлы можно создавать также при помощи вызова **open** с соответствующими аргументами.

Файл может быть закрыт при помощи вызова **close**, после чего дескриптор файла можно использовать повторно (для последующего **creat** или **open**). Системные вызовы **creat** и **open** всегда возвращают наименьший неиспользуемый в данный момент дескриптор файла.

Таблица 3.3.2. Некоторые системные вызовы для работы с файлами. В случае ошибки возвращаемое значение s равно –1, fd — дескриптор файла, position — смещение в файле.

Системный вызов	Описание
fd=creat(name, mode)	Один из способов создания нового файла
fd=open(file, how)	Открыть файл для чтения, записи либо и того и другого одновременно

s=close(fd)	Закрыть открытый файл
n=read(fd, buffer, nbytes)	Прочитать данные из файла в буфер
n=write(fd, buffer, nbytes)	Записать данные из буфера в файл
position=lseek(fd, offset, whence)	Переместить указатель в файле
s=stat(name, &buf)	Получить информацию о состоянии файла
s=fstat(fd, &buf)	Получить информацию о состоянии файла
s=pipe(&fd[0])	Создать канал
s=fcntl(fd, cmd, ...)	Блокировка файла и другие операции

Когда программа начинает выполнение стандартным образом, файловые дескрипторы 0, 1 и 2 уже открыты для стандартного ввода, стандартного вывода и стандартного потока сообщений об ошибках соответственно. Таким образом, фильтр (например, программа sort) может просто читать свои входные данные из файла с дескриптором 0, а писать выходные данные в файл с дескриптором 1, не заботясь о том, что это за файлы. Работа этого механизма обеспечивается оболочкой, которая проверяет, чтобы эти дескрипторы соответствовали нужным файлам (прежде чем программа начнет свою работу).

Чаще всего программы используют системные вызовы **read** и **write**. У обоих вызовов по три параметра: дескриптор файла (указывающий, с каким из открытых файлов будет производиться операция чтения или записи), адрес буфера (сообщающий, куда положить данные или откуда их взять), а также счетчик (указывающий, сколько байтов следует передать). Пример типичного вызова:

```
n = read(fd, buffer, nbytes)
```

Основные функции ввода-вывода в Windows

Функция **CreateFileW** (fileapi.h) создает или открывает файл или устройство ввода-вывода [5]. Наиболее часто используемые устройства ввода-вывода: файловый поток, каталог, физический диск, том, буфер консоли, ленточный накопитель, ресурс связи, mailslot и канал. Функция возвращает дескриптор, который можно использовать для доступа к файлу или устройству для различных типов операций ввода-вывода в зависимости от файла или устройства, а также указанных флагов и атрибутов.

Синтаксис

```
HANDLE CreateFileW(
  [in]           LPCWSTR               lpFileName,
  [in]           DWORD                 dwDesiredAccess,
  [in]           DWORD                 dwShareMode,
  [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  [in]           DWORD                 dwCreationDisposition,
  [in]           DWORD                 dwFlagsAndAttributes,
  [in, optional] HANDLE                hTemplateFile );
```

Параметры:

`[in]` **lpFileName**

Имя создаваемого или открываемого файла или устройства. В этом имени можно использовать косую черту (/) или обратную косую черту (\).

`[in]` **dwDesiredAccess**

Запрошенный доступ к файлу или устройству, который можно обобщить как чтение, запись, оба или ни один из них). Чаще всего используются значения **GENERIC_READ**, **GENERIC_WRITE** или и то, и другое (**GENERIC_READ | GENERIC_WRITE**).

`[in]` **dwShareMode**

Запрошенный режим общего доступа для файла или устройства, который можно читать, записывать, удалять, все или нет (см. следующую таблицу). Этот флаг не влияет на запросы доступа к атрибутам или расширенным атрибутам.

0	FILE_SHARE_READ	FILE_SHARE_WRITE	FILE_SHARE_DELETE
0x00000000	0x00000001	0x00000002	0x00000004

[in, optional] **lpSecurityAttributes**

Указатель на структуру SECURITY_ATTRIBUTES, содержащую два отдельных, но связанных элемента данных: необязательный дескриптор безопасности и логическое значение, определяющее, может ли возвращаемый дескриптор наследоваться дочерними процессами. Этот параметр может принимать значение NULL.

[in] **dwCreationDisposition**

Действие, выполняемое с файлом или устройством, которые существуют или не существуют.

CREATE_NEW	CREATE_ALWAYS	OPEN_EXISTING	OPEN_ALWAYS	TRUNCATE_EXISTING
1	2	3	4	5

[in] **dwFlagsAndAttributes**

Атрибуты и флаги файла или устройства, **FILE_ATTRIBUTE_NORMAL** являются наиболее распространенным значением по умолчанию для файлов.

[in, optional] **hTemplateFile**

Допустимый дескриптор файла шаблона с правом доступа **GENERIC_READ** . Файл шаблона предоставляет атрибуты файла и расширенные атрибуты для создаваемого файла.

Этот параметр может принимать значение NULL.

При открытии существующего файла **CreateFile** игнорирует этот параметр.

При открытии нового зашифрованного файла файл наследует список управления доступом на уровне пользователей из родительского каталога.

Функция **ReadFile** (fileapi.h) считывает данные из указанного файла или устройства ввода-вывода [6]. Операции чтения выполняются в позиции, указанной указателем файла, если устройство поддерживает.

Эта функция предназначена как для синхронных, так и для асинхронных операций. Аналогичная функция, предназначенная исключительно для асинхронных операций, — **ReadFileEx.**

Синтаксис

```
BOOL ReadFile(
  [in]                HANDLE       hFile,
  [out]               LPVOID       lpBuffer,
  [in]                DWORD        nNumberOfBytesToRead,
  [out, optional]     LPDWORD      lpNumberOfBytesRead,
  [in, out, optional] LPOVERLAPPED lpOverlapped );
```

[in] **hFile**

Дескриптор для устройства (например, файл, файловый поток, физический диск, том, буфер консоли, ленточный накопитель, сокет, ресурс связи, почтовый слот или канал). Параметр hFile должен быть создан с доступом на чтение.

Пример открытия файла и записи в файл

```
HANDLE hFile;

char filename[] = "C:\\folder\\file.txt";
char DataBuffer[] = "This is some test data to write to the file.";
DWORD dwBytesToWrite = (DWORD)strlen(DataBuffer);
```

```
DWORD dwBytesWritten = 0;
BOOL bErrorFlag = FALSE;
hFile = CreateFile(filename,              // name of the write
                   GENERIC_WRITE,         // open for writing
                   0,                     // do not share
                   NULL,                  // default security
                   CREATE_NEW,            // create new file only
                   FILE_ATTRIBUTE_NORMAL, // normal file
                   NULL);                 // no attr. template
if (hFile == INVALID_HANDLE_VALUE) { }    // обработка ошибки
bErrorFlag = WriteFile(hFile,           // open file handle
                       DataBuffer,      // start of data to write
                       dwBytesToWrite,  // number of bytes to write
                       &dwBytesWritten, // number of bytes that were written
                       NULL);           // no overlapped structure
if (FALSE == bErrorFlag)
{}                                       // обработка ошибки
else
{
    if (dwBytesWritten != dwBytesToWrite)
    { }                                  // обработка ошибки
    else
    { }                                  // это успех

}
CloseHandle(hFile);
```

3.3.5. Буферизация операций ввода-вывода

Буферизация — метод организации обмена, в частности, ввода и вывода данных в компьютерах и других вычислительных устройствах, который подразумевает использование буфера для временного хранения данных. При вводе данных одни устройства или процессы производят запись данных в буфер, а другие — чтение из него, при выводе — наоборот. Процесс, выполнивший запись в буфер, может немедленно продолжать работу, не ожидая, пока данные будут обработаны другим процессом, которому они предназначены. В свою очередь, процесс, обработавший некоторую порцию данных, может немедленно прочитать из буфера следующую порцию. Таким образом, буферизация позволяет процессам, производящим ввод, вывод и обработку данных, выполняться параллельно, не ожидая, пока другой процесс выполнит свою часть работы. Поэтому буферизация данных широко применяется в многозадачных ОС.

Необходимость буферизации диктуется следующими проблемами:

1. При страничной организации данных, страница памяти, в которую считываются данные с внешнего устройства (или наоборот) должна на все время операции ввода-вывода оставаться в ОП. Однако если в это время нужна выгрузка процесса из ОП, то страница должна блокироваться и не выгружаться. Это усложняет работу с виртуальной памятью.

2. При выполнении операции ввода-вывода процесс приостанавливает работу и выгружается из ОП в виртуальную, где ожидает завершения операции ввода-вывода. Однако, т.к. вместе с процессом из ОП выгружена и сама страница, то некуда выводить данные и ввод-вывод блокируется, образуется взаимная блокировка: процесс ждет завершения операции ввода-вывода, а операция ввода-вывода ждет появления страницы в памяти. Поэтому, процесс должен быть заблокирован и оставаться в ОП, хотя операция ввода-вывода может быть выполнена не сразу, а поставлена в очередь.

Для эффективности работы можно выполнить чтение блока данных в специальное место памяти – буфер, а затем удалить страницу из ОП. При этом устройство будет считывать и сохранять данные не в память процесса, а в буфер. Такая схема называется буферизацией операций ввода-вывода.

1. **Одинарная буферизация**. Часть ОП в системной области отводится для процесса как буфер. Данные сначала отправляются в буфер, а затем в устройство (или наоборот, из устройства в буфер, а затем в ОП). После считывания 1 блока из буфера в ОП процесс начинает его обработку, а в это время параллельно идет считывание в буфер следующего блока из внешнего устройства (в предположении, что именно он и потребуется т.к. считывание данных с устройства чаще всего выполняется последовательного числа блоков). За счет этого и происходит ускорение ввода-вывода.

2. **Двойная буферизация** (сменный буфер). Если использовать не 1, а 2 буфера, то операции ввода-вывода можно еще ускорить. Во время считывания данных из первого буфера в ОП процесса можно одновременно выполнять считывание данных с устройства во 2 буфер

3. **Циклическая буферизация**. При двойной буферизации скорости считывании из буфера в ОП и из устройства в буфер отличаются, что приводит к торможению работы процесса. Поэтому используется еще система циклических буферов, когда буферов не 2, а несколько, и считывание из них в ОП выполняется последовательно по кругу. Такая буферизация сглаживает разницу в скорости считывания из ОП и из устройства и ускоряет работу процессов

3.3.6. Дисковое планирование

При работе диска его скорость вращения постоянна. Для того чтобы выполнить чтение или запись, головка должна находиться над искомой дорожкой, а кроме того — над началом искомого сектора на этой дорожке. Процедура выбора дорожки включает в себя перемещение головки (в системе с подвижными головками) или электронный выбор нужной головки (в системе с неподвижными головками). В системе с подвижными головками на позиционирование головки над дорожкой затрачивается время, известное как время поиска. В любом случае после выбора дорожки контроллер диска ожидает момент, когда начало искомого сектора достигнет головки. Время, необходимое для достижения головки началом сектора, известно как время задержки из-за вращения, или время ожидания вращения. Сумма времен поиска (если таковой выполняется) и времени за-держки из-за вращения составляет время доступа — время, которое требуется для позиционирования для чтения или записи. Как только головка попадает в искомую позицию, выполняется операция чтения или записи, осуществляемая во время движения сектора под головкой, — это и есть непосредственная передача данных при выполнении операции ввода-вывода [7].

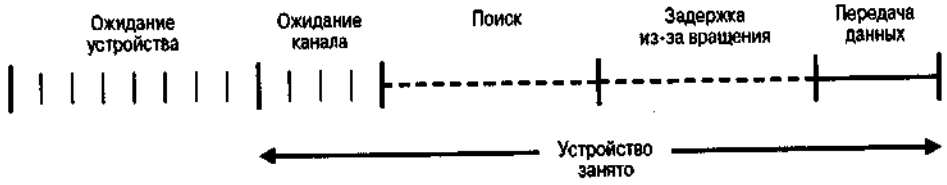

Рис. 3.3.5. Временная диаграмма работы диска

Рассмотрим типичную ситуацию в многозадачной среде, когда операционная система поддерживает очередь запросов для каждого устройства ввода-вывода. Соответственно, в очереди одного диска будет находиться некоторое количество запросов на ввод-вывод (чтение или запись) от различных процессов. Если выбирать запросы из очереди случайным образом, то следует ожидать, что искомые дорожки будут располагаться в произвольном порядке, это приведет к очень низкой производительности. Такое случайное распределение (**случайное планирование**) может служить точкой отсчета для оценки других методик.

Простейшей формой планирования является планирование «**первым вошел — первым вышел» (FIFO)**, что просто означает обработку запросов из очереди в порядке их

поступления. Преимущество такой стратегии — в ее беспристрастности. При использовании стратегии FIFO надеяться на высокую производительность можно только при небольшом количестве процессов и запросах в основном к близким группам секторов. Однако при работе большого количества процессов произво-дительность будет почти такой же, как и при случайном планировании.

В системе **с использованием приоритета (PRI)** управление планированием явля-ется внешним по отношению к программному обеспечению управления диском. Та-кой подход не имеет отношения к оптимизации использования диска, но зато удов-летворяет некоторым другим целям операционной системы. Коротким пакетным за-даниям, а также интерактивным заданиям часто присваивается более высокий приоритет, чем длинным заданиям, требующим более длительных вычислений. Эта схема позволяет быстро завершить большое количество коротких заданий в системе и обеспечивает малое время отклика. Однако при использовании этого метода у больших заданий оказывается слишком длительным ожидание выполнения диско-вых операций. Кроме того, такая стратегия может привести к противодействию со стороны пользователей, которые будут разделять свои задания на малые подзадания. Не подходит эта стратегия и для работы с базами данных.

LIFO «Последним вошел — первым вышел». Как это ни удивительно, но стратегия выполнения первым самого последне-го запроса имеет свои преимущества. В системах обработки транзакций при пре-доставлении устройства для последнего пользователя должно выполняться лишь небольшое перемещение указателя последовательного файла. Использование преимуществ локализации позволяет повысить пропускную способность и уменьшить длину очереди. К сожалению, если нагрузка на диск велика, сущест-вует очевидная возможность голодания процесса.

Стратегия выбора наименьшего времени обслуживания (Shortest Service Time First — **SSTF**) заключается в выборе того дискового запроса на ввод-вывод, который требует наименьшего перемещения головок из текущей позиции. Следовательно, мы минимизируем время поиска. Естественно, постоянный выбор минимального времени поиска не дает гарантии, что среднее время поиска при всех перемещениях будет минимальным, но тем не менее эта стратегия обеспечивает лучшую по сравнению с FIFO производительность дисковой системы. Поскольку головки могут перемещаться в двух направлениях, то при равных расстояниях для принятия решения может быть использован случайный выбор направления.

При использовании этого алгоритма **SCAN** перемещение головки происходит только в одном направлении, удовлетворяя те запросы, которые соответствуют выбранному направлению. После достижения последней дорожки в выбранном направлении (или когда исчерпаются возможные запросы), направление изме-нится на противоположное.

Стратегия **C-SCAN** (циклическое сканирование) ограничивает сканирование только одним направлением. Когда обнаруживается последняя дорожка в задан-ном направлении, головка возвращается в противоположный конец диска, и сканирование начинается снова. Это уменьшает максимальную задержку, вы-званную новыми запросами. Если при использовании стратегии SCAN ожидаемое время сканирования от внутренней к внешней дорожке равно t, то ожидае-мый интервал обслуживания секторов, находящихся на периферии, будет равен 2t. При использовании стратегии C-SCAN этот интервал будет порядка t+smax, где smax — максимальное время поиска.

Стратегия **N-step-SCAN** позволяет произвести сегментацию очереди дисковых запросов на подочереди длиной *N.* Каждая подочередь обрабатывается за один прием с использованием стратегии SCAN. В ходе обработки очереди к некоторой другой очереди могут добавляться новые запросы. Если в конце текущего сканирования доступными оказываются менее *N* запросов, то все они обрабатываются в следующем цикле

сканирования. При больших значениях *N* выполнение алгоритма N-step-SCAN похоже на выполнение SCAN; предельный случай *N=1* соответствует стратегии FIFO.

FSCAN — стратегия, использующая две подочереди. С началом сканирования все запросы находятся в одной из очередей; другая при этом остается пустой. Во время сканирования первой очереди все новые запросы попадают во вторую очередь. Таким образом, обслуживание новых очередей откладывается, пока не будут обработаны все старые запросы.

3.3.8. Файловые системы

В любой операционной системе (включая Linux) пользователь прежде всего видит файловую систему. При минимальном алгоритме и сильно ограниченном количестве системных вызовов операционная система Linux предоставляет мощную и элегантную файловую систему [1].

Файловая система (file system) — порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах, а также в другом электронном оборудовании: цифровых фотоаппаратах, мобильных телефонах и т. п. Файловая система определяет формат содержимого и способ физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имен файлов (и каталогов), максимальный возможный размер файла и раздела, набор атрибутов файла. Некоторые файловые системы предоставляют сервисные возможности, например, разграничение доступа или шифрование файлов.

Файловая система ext позволяла использовать имена файлов длиной 255 символов и размер файлов 2 Гбайт, но она была медленной. В итоге была изобретена файловая система ext2 (с длинными именами файлов, большими файлами и более высокой производительностью). Однако Linux поддерживает несколько десятков файловых систем при помощи уровня виртуальной файловой системы Virtual File System (VFS). Наиболее широко используемыми файловыми системами являются Ext4 и XFS. Последняя является файловой системой по умолчанию в дистрибутивах на основе RHEL. А Ext4 является стандартной файловой системой в дистрибутивах Debian и Ubuntu. При выборе файловой системы необходимо учитывать такие факторы, как масштабируемость, стабильность и целостность данных [8].

VFS определяет набор основных абстракций файловой системы и разрешенные с этими абстракциями операции. Описанные в предыдущем разделе системные вызовы обращаются к структурам данных VFS, определяют тип файловой системы (к которой принадлежит нужный файл) и при помощи хранящихся в структурах данных VFS указателей на функции запускают соответствующую операцию в указанной файловой системе.

В табл. 3.3.3 даны четыре основные структуры файловой системы, поддерживаемые VFS.

Таблица 3.3.3. Поддерживаемые в VFS абстракции файловой системы

Объект	Описание	Операция
Суперблок	Конкретная файловая система	read_inode, sync_fs
Элемент каталога (dentry)	Элемент каталога, компонент пути	create, link
i-узел	Конкретный файл	d_compare, d_delete
Файл (file)	Открыть связанный с процессом файл	read, write

Суперблок содержит критичную информацию о компоновке файловой системы. Разрушение суперблока делает файловую систему нечитаемой. i-узел или inode

(сокращение от «индекс-узлы», никто их так не называет) описывает один файл. Обратите внимание на то, что в Linux каталоги и устройства также представлены файлами, так что они тоже имеют соответствующие i-узлы. И суперблок, и i-узлы имеют соответствующие структуры на том физическом диске, где находится файловая система.

Суперблок хранится в первом блоке каждой группы блоков (за исключением группы 1, в которой в первом блоке расположена загрузочная запись). Суперблок является начальной точкой файловой системы. Он имеет размер 1024 байта и всегда располагается по смещению 1024 байта от начала файловой системы. Наличие нескольких копий суперблока объясняется чрезвычайной важностью этого элемента файловой системы. Дубликаты суперблока используются при восстановлении файловой системы после сбоев [9].

Информация, хранимая в суперблоке, используется для организации доступа к остальным данным на диске. В суперблоке определяется размер файловой системы, максимальное число файлов в разделе, объем свободного пространства и содержится информация о том, где искать незанятые участки. При запуске ОС суперблок считывается в память и все изменения файловой системы вначале находят отображение в копии суперблока, находящейся в ОП, и записываются на диск только периодически. Это позволяет повысить производительность системы, так как многие пользователи и процессы постоянно обновляют файлы. С другой стороны, при выключении системы суперблок обязательно должен быть записан на диск, что не позволяет выключать компьютер простым выключением питания. В противном случае, при следующей загрузке информация, записанная в суперблоке, окажется не соответствующей реальному состоянию файловой системы.

Каждому файлу на диске соответствует один и только один **индексный дескриптор файла**, который идентифицируется своим порядковым номером - индексом файла (inode). Это означает, что число файлов, которые могут быть созданы в файловой системе, ограничено числом индексных дескрипторов, которое либо явно задается при создании файловой системы, либо вычисляется исходя из физического объема дискового раздела. В частности, он содержит следующую информацию, тип файла и права доступа, идентификатор владельца размер в байтах, отметки времени о доступе, счетчик числа связей.

Чтобы улучшить некоторые операции с каталогами и перемещение по путям (таким, как `/usr/ast/bin`), VFS поддерживает структуру данных dentry, которая представляет элемент каталога (directory entry). Эта структура данных создается файловой системой на ходу. Элементы каталога кэшируются в так называемом dentry_cache. Например, dentry_cache будет содержать элементы для `/`, `/usr`, `/usr/ast` и т. д. Если несколько процессов обращаются к одному и тому же файлу при помощи одной и той же жесткой ссылки (то есть одного и того же пути), то их объект файла будет указывать на один и тот же элемент в этом кэше. В отличие от предыдущих двух объектов, объект dentry не соответствует какой бы то ни было структуре данных на жестком диске. Подсистема VFS создает эти объекты на лету на основании строкового представления имени пути. Поскольку объекты элементов каталога не хранятся физически на дисках, то в структуре struct dentry нет никаких флагов, которые указывают на то, изменен ли объект (т.е. должен ли он быть записан назад на диск).

И наконец, структура данных **`file`** является представлением открытого файла в памяти, она создается в ответ на системный вызов **`open`**. Она поддерживает такие операции, как **`read`**, **`write`**, **`sendfile`**, **`lock`** (и прочие описанные в предыдущем разделе системные вызовы).

Реализованные под уровнем VFS реальные файловые системы не обязаны использовать внутри себя точно такие же абстракции и операции. Однако они должны реализовать

семантически эквивалентные операции файловой системы (такие же, как указанные для объектов VFS). Элементы структур данных operations для каждого из четырех объектов VFS — это указатели на функции в нижележащей файловой системе.

Файловая система EXT2

Это вторая расширенная файловая система, созданная для преодоления ограничений файловой системы EXT.

Представлена в 1993 году компанией Remy Card. Это была первая файловая система коммерческого уровня для Linux.

Не поддерживает ведение журнала

Подходит для SD-карт и USB-накопителей, поскольку имеет высокую производительность и меньшее количество операций записи (поскольку ведение журнала недоступно). Хранилища USB и SD имеют ограниченное количество циклов записи, поэтому EXT2 лучший выбор для них.

Ограничения: размер отдельного файла от 16 ГБ до 2 ТБ. Размер файловой системы от 2 ТБ до 32 ТБ.

Лимиты рассчитываются на основе размера используемого блока. Размер блока варьируется от 1 КБ до 8 КБ. Например, если используется размер блока 1 КБ, максимальный размер файла может достигать 16 ГБ, а для 8 КБ — 2 ТБ. Размеры среднего диапазона составляют 2 КБ и 4 КБ, а ограничения на размер файла составляют 256 ГБ и 2 ТБ (не упомянуты в приведенных выше ограничениях) соответственно. То же самое относится и к ограничениям размера файловой системы, определенным выше.

Файловая система EXT3

Третья расширенная файловая система была создана для преодоления ограничений файловой системы EXT2.

Представлена в 2001 году Стивеном Твиди. Это была самая распространенная файловая система в любом дистрибутиве Linux.

Поддерживает ведение журнала

Ведение журнала отслеживает изменения файлов, что помогает быстрому восстановлению и снижает вероятность потери данных в случае сбоя системы.

Ограничения: размер отдельного файла от 16 ГБ до 2 ТБ. Размер файловой системы от 4 ТБ до 32 ТБ.

Обновление ФС с ext2 до ext3 — это онлайн-процесс без простоев.

Файловая система EXT4

Это четвертая расширенная файловая система, созданная для преодоления ограничений файловой системы EXT3.

Представлена в 2008 году командой разработчиков. Это самая последняя файловая система в семействе ext.

Поддерживает ведение журнала

Введено много новых функций. Экстенты, обратная совместимость, постоянное предварительное выделение, отложенное выделение, неограниченное количество подкаталогов, контрольная сумма журнала, более быстрая проверка FS, прозрачное шифрование.

Ограничения: максимальный размер отдельного файла от 16 ГБ до 16 ТБ. Размер файловой системы до 1EB.

Каталог может содержать максимум 64 000 подкаталогов (в отличие от 32 000 в ext3).

Улучшенные временные метки — в Ext4 реализованы временные метки, измеряемые в наносекундах, что является улучшением по сравнению с детализацией временных меток на основе секунд. Так как метки в секундах считаются недостаточными. Кроме того, к отметке времени было добавлено еще 408 лет, чтобы преодолеть установленный лимит 2038 года.

Обновление ФС не требуется. Благодаря обратной совместимости ext2, ext3 можно монтировать напрямую как ext4.

Достоинства Ext4:

- очень стабильная, проверенная временем файловая система;
- поддерживается во всех дистрибутивах по умолчанию;
- продолжает развиваться и улучшаться;
- поддерживает прозрачное шифрование;

Недостатки Ext4:

- не поддерживает менеджер томов;
- реальный максимальный объем раздела намного меньше одного экзабайта;
- не поддерживает новые модные технологии вроде шифрования и сжатия на лету, copy-on-write, дедупликацию, снапшоты и многое другое.

Таблица 3.3.4. Сравнение файловых систем семейства EXT

Параметр	EXT2	EXT3	EXT4
Год представления	1993 год	2001 г.	2008 год
Разработан	Реми Кард	Стивен Твиди	Команда разработчиков
Ведение журнала	Нет	Доступный	Доступный
Индивидуальный размер файла	от 16 ГБ до 2 ТБ	от 16 ГБ до 2 ТБ	от 16 ГБ до 16 ТБ
Размер файловой системы	от 2 ТБ до 32 ТБ	от 4 ТБ до 32 ТБ	до 1EB
Обновление	Можно сделать онлайн на EXT3. Может быть смонтирована как EXT4. Обновление не требуется	Может быть смонтирована как EXT4. Обновление не требуется	NA

Файловая система XFS

- **Максимальный размер раздела:** 8 экзабайт.
- **Максимальный размер файла:** 8 экзабайт.
- **Максимальное количество файлов:** 2 в 64 степени.
- **Максимальная длина имени:** 256 символов.
- **Количество вложенных каталогов:** не ограничено.

XFS считается расширенной файловой системой. Это высоко производительная 64-битная, журналируемая файловая система. Поддержка XFS была добавлена в ядро в 2002 году. А в 2009 она впервые была использована в Red Hat Enterprise Linux 5.4. Файловая система рассчитана на очень высокую производительность для больших файловых систем, а также может обеспечивать хорошую производительность для большого количества параллельных операций [10].

Тем не менее эта файловая система не завоевала большой популярности. Она очень стабильная и производительная, поэтому используется по умолчанию в RHEL 7 и уже в 8, однако в ней нет ничего такого, чего бы не было в Ext4. XFS явно не относится к файловым системам следующего поколения, как Btrfs. К тому же она имеет ограничения, которые могут мешать обычным пользователям.

Достоинства XFS:

- Очень стабильная и производительная.

Недостатки XFS:

- Не поддерживает возможности файловых систем следующего поколения, такие как управление томами, сжатие, дедупликацию и другие возможности.
- Разделы с XFS можно только расширять, уменьшать нельзя.

Файловая система Btrfs

- **Максимальный размер раздела:** 16 экзабайт.
- **Максимальный размер файла:** 16 экзабайт.
- **Максимальное количество файлов:** 2^{64} степени.
- **Максимальная длина имени:** 256 символов.
- **Количество вложенных каталогов:** не ограничено.

Btrfs - это новая файловая система, разработанная с нуля. Расшифровывается как B-Tree Filesystem. Она была анонсирована Крисом Масоном в 2006 году во время его работы в Oracle. Btrfs поддерживает управление несколькими томами на одном разделе, контрольные суммы для блоков, асинхронную репликацию, прозрачное сжатие, а также многие другие возможности современных файловых систем.

В наши дни её уже можно считать стабильной и использовать в качестве основой файловой системы для Linux. Её можно использовать в качестве альтернативы Ext4 как файловую систему для одного раздела, но такие дополнительные функции как менеджер томов, топография из нескольких дисков и управление снапшотами, лучше не использовать. Они могут вызывать проблемы с производительностью или даже потерю данных.

Btrfs уже давно полноценно добавлена в ядро и некоторые дистрибутивы поддерживают её выбор на этапе установки. Например, SUSE Linux использует её в качестве файловой системы по умолчанию с 2015 года. Зато Red Hat больше поддерживает Btrfs с 2017. Возможно для серьезных Enterprice систем эта файловая система ещё не подходит, зато для домашнего использования она вполне готова.

Преимущества Btrfs:

- Большие лимиты и хорошая масштабируемость по сравнению с Ext4.
- Поддержка большинства возможностей современных файловых систем, таких как менеджер томов, сжатие на лету, дедупликация, copy-on-write, снапшоты и многое другое.
- Поддержка проверки контрольных сумм, что позволяет точно обнаружить повреждение данных из-за аппаратных проблем.

Недостатки Btrfs:

- Файловая система относительно новая и совсем недавно она начала считаться стабильной. Использование новых возможностей может привести к повреждению данных.
- Отсутствие шифрования на лету.

Файловые системы в Windows

Типы файловых систем Windows включают:

Таблица размещения файлов (**FAT**), **FAT32** и расширенная таблица размещения файлов (**exFAT**).

Файловая система NT (**NTFS**).

Устойчивая файловая система (**ReFS**).

Файловая система:

- Предоставляет ряд функций, реализующих хранение и извлечение файлов на устройствах хранения.
- Позволяет организовывать файлы в иерархическую структуру и контролировать их формат и соглашение об именах.
- Поддерживает широкий спектр устройств хранения данных.

Все файловые системы, доступные в операционной системе Windows, состоят из следующих компонентов хранения:

- Файлы
- Каталоги
- Объемы

Файловые системы семейства FAT

Файловая система FAT — это самая простая файловая система, поддерживаемая операционными системами Windows. Она отслеживает объекты файловой системы, используя таблицу уровней томов[11]. Пространство тома делится на кластеры одинакового размера. В каталоге для каждого файла указывается только номер первого кластера с данными и размер файла. Остальные данные находятся по цепочке ссылок, которые хранятся в специальных таблицах. Конец цепочки обозначается специальным значением. FAT поддерживает две копии таблицы для обеспечения устойчивости. Обе таблицы и корневой каталог должны находиться в фиксированном месте на отформатированном диске.

Из-за ограничения размера таблицы размещения файлов вы не можете использовать FAT для создания томов размером более 4 гигабайт (ГБ). Для размещения дисков большего размера Microsoft разработала файловую систему FAT32, которая поддерживает разделы размером до 64 ГБ.

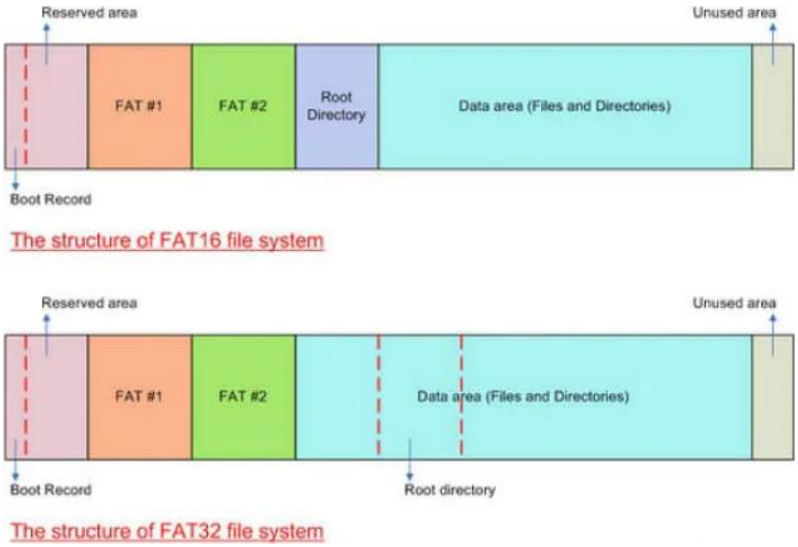

Рис. 3.3.6 Изменения в структуре FAT32 в сравнении с FAT16

Файловая система FAT32 создает для каждого файла запись, состоящую из 32 бит вместо прежних 16. Дополнительная информация позволяет создавать большие разделы диска

при сохранении высокой скорости чтения/записи. Это также ценно для отслеживания дополнительной информации о файлах, такой как размер значков и цветовые схемы, которые используются в графическом интерфейсе. Он был настолько эффективен, что его до сих пор можно использовать для современных компьютеров [12].

FAT32 по своей конструкции аналогична своим предшественникам FAT12 и FAT16. Она хранит файлы в фиксированных кластерах вдоль диска и создает архив, содержащий необходимую информацию, такую как адрес файла, конечный указатель и тип распределения секторов. Доступ к архиву осуществляется через загрузочный сектор ОС, что позволяет пользователю получить доступ к файлу при условии, что он имеет некоторое представление о его местонахождении. Кроме того, данные не обязательно должны быть полными, поскольку вы можете получить доступ к определенному файлу с частичной информацией, лишь бы она оставалась достоверной.

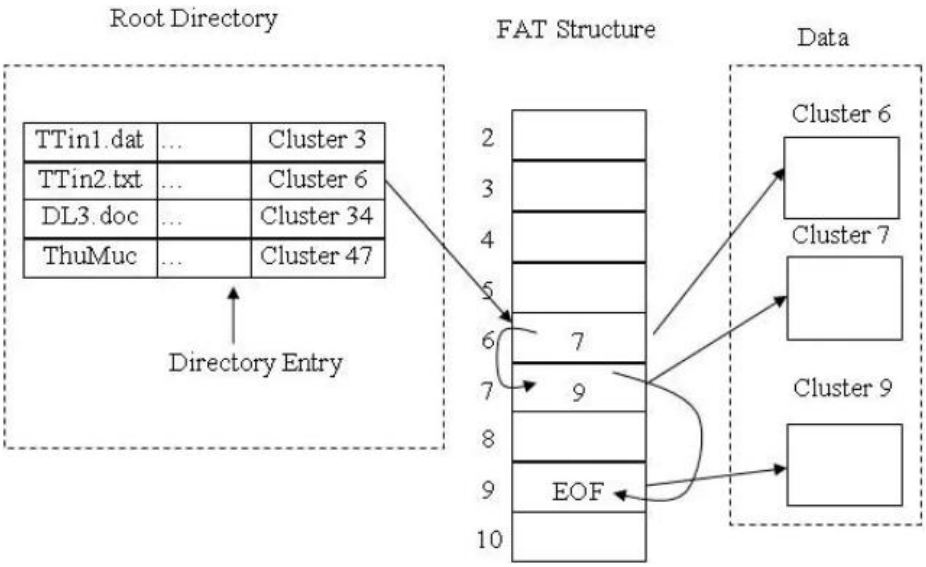

Рис. 3.3.7. Цепочка кластеров в FAT

exFAT — это файловая система, разработанная для флэш-накопителей, с поддержкой томов большего размера, чем доступные в FAT32. Он работает с мультимедийными устройствами, такими как современные телевизоры с плоским экраном, медиацентры и портативные медиаплееры.

Основными преимуществами exFAT перед предыдущими версиями FAT являются:

- уменьшение количества перезаписей одного и того же сектора, что важно для флеш-накопителей, у которых ячейки памяти необратимо изнашиваются после определённого количества операций записи (это сильно смягчается выравниванием износа (wear leveling), встроенным в современные USB-накопители и SD-карты);
- теоретический лимит на размер файла 2^{64} байт (16 эксабайт);
- максимальный размер кластера увеличен до 2^{25} байт (32 мегабайта);
- улучшение распределения свободного места за счёт введения бит-карты свободного места, что может уменьшать фрагментацию диска;
- поддержка транзакций (опциональная возможность, должна поддерживаться устройством).

Ни FAT, ни FAT32 не обеспечивают безопасность на уровне файловой системы. Не следует создавать тома FAT или FAT32 на дисках, подключенных к серверам под управлением любой из операционных систем Windows Server. Однако вы можете рассмотреть возможность использования FAT, FAT32 или exFAT для форматирования внешних носителей, таких как USB-накопители.

NTFS

Традиционно NTFS была наиболее распространенным выбором файловой системы для операционных систем Windows Server. NTFS предлагает многочисленные улучшения по сравнению с FAT, которые используют расширенные структуры данных для повышения производительности, надежности и использования дискового пространства. NTFS также обеспечивает встроенную безопасность с такими возможностями контроля доступа, как списки управления доступом (ACL), аудит, ведение журнала файловой системы и шифрование. NTFS также поддерживает сжатие и шифрование файловой системы, хотя они являются взаимоисключающими, поэтому их нельзя применять к одному и тому же файлу или папке.

ReFS

Microsoft представила ReFS в Windows Server 2012 для расширения возможностей NTFS. Одной из основных сильных сторон ReFS, как следует из ее названия, является ее повышенная устойчивость к повреждению данных за счет более точного механизма обнаружения и способности устранять проблемы целостности в Интернете. ReFS также предлагает поддержку больших размеров отдельных файлов и томов, включая их дедупликацию.

В большинстве случаев ReFS является оптимальным выбором файловой системы для томов данных в Windows Server 2022. Однако следует иметь в виду, что ReFS не обеспечивает полную функциональность с NTFS. Например, ReFS не поддерживает сжатие и шифрование на уровне файлов. Он также не подходит для загрузочных томов и съемных носителей.

ReFS имеет множество преимуществ перед NTFS:

- целостность метаданных с помощью контрольных сумм;
- потоки целостности с целостностью пользовательских данных;
- Распределение по транзакционной модели записи;
- большие размеры томов, файлов и каталогов (2^{78} байт с размером кластера 16 КБ);
- пулы хранения и виртуализация;
- распределение данных для повышения производительности и избыточности;
- очистка диска для защиты от скрытых ошибок диска;
- устойчивость к коррупции с восстановлением;
- общие пулы хранения данных на машинах.

Таблица. 3.3.5. Сравнение файловых систем Windows Server

	FAT	NTFS	REFS
Размер тома	**FAT = <=4GB** **FAT32 = <=64GB** **exFAT = >=64GB**	**До 8 ПБ**	**До 35 ПБ**
		Сжатие файлов	**Нет сжатия файлов**
Безопас-ность	**Не поддерживается**	**Разрешения на файлы/папки с помощью ACLs**	**Разрешения на файлы/папки с помощью ACLs**
		Шифрование	**Нет шифрования**

Сценарии использо-вания	Внешние носители (USB, Flash Drive)	Загрузочные тома и тома данных	Только тома данных, загрузочные тома не поддерживаются

3.3.8. Файлы и директории

Файл в системе Linux — это последовательность байтов произвольной длины (от 0 до некоторого максимума), содержащая произвольную информацию. Не делается различия между текстовыми (ASCII) файлами, двоичными файлами и любыми другими типами файлов. Значение битов в файле целиком определяется владельцем файла. Системе это безразлично. Имена файлов ограничены 255 символами. В именах файлов разрешается использовать все ASCII-символы, кроме символа NUL [1].

По соглашению многие программы ожидают, что имена файлов будут состоять из основного имени и расширения, разделенных точкой (которая также считается символом). Так, prog.c — это обычно программа на языке C, prog.py — это обычно программа на языке Python, а prog.o — чаще всего объектный файл (выходные данные компилятора). Эти соглашения никак не регламентируются операционной системой, но некоторые компиляторы и другие программы ожидают файлов именно с такими расширениями. Расширения имеют произвольную длину, причем файлы могут иметь по несколько расширений, например, prog.java.Z, что, скорее всего, представляет собой сжатую программу на языке Java.

Для удобства файлы могут группироваться в каталоги. Каталоги хранятся на диске в виде файлов, и с ними можно работать практически так же, как с файлами. Каталоги могут содержать подкаталоги, что приводит к иерархической файловой системе.

Корневой каталог называется **/** и всегда содержит несколько подкаталогов. Символ **/** используется также для разделения имен каталогов, поэтому имя **/usr/ast/x** обозначает файл **x**, расположенный в каталоге **ast**, который в свою очередь находится в каталоге **usr**.

Существует два способа задания имени файла в системе Linux (как в оболочке, так и при открытии файла из программы). Первый способ заключается в использовании абсолютного пути (absolute path), указывающего, как найти файл от корневого каталога. Пример абсолютного пути: **/usr/ast/books/mos4/chap-10**. Он сообщает системе, что в корневом каталоге следует найти каталог **usr**, затем в нем найти каталог **ast**, который содержит каталог **books**, в котором содержится каталог **mos4**, а в нем расположен файл **chap-10**.

Абсолютные имена путей часто бывают длинными и неудобными. По этой причине операционная система Linux позволяет пользователям и процессам обозначить каталог, в котором они работают в данный момент, как рабочий каталог (working directory). Имена путей могут указываться относительно рабочего каталога. Путь, заданный относительно рабочего каталога, называется относительным путем (relative path). Например, если каталог **/usr/ast/books/mos4** является рабочим каталогом, тогда команда оболочки

```
cp chap-10 backup-10
```

имеет тот же самый эффект, что и более длинная команда

```
cp /usr/ast/books/mos4/chap-10 /usr/ast/books/mos4/backup-10
```

Ссылки (link) представляют собой записи каталога, указывающие на существующие файлы. Бывают жесткие (hard) и символьные.

При создании каталога в нем автоматически создаются две записи, «.» и «..». Первая запись обозначает сам каталог. Вторая является ссылкой на родительский каталог, то есть каталог, в котором данный каталог числится как запись.

Точка . в начале имени файла делает его скрытым, то есть, он не показывается в выводе команды **ls**.

По умолчанию домашний каталог пользователя будет содержать много скрытых файлов. Они часто используются для установки пользовательских настроек конфигурации и должны быть изменены только опытным пользователем

Для отображения скрытых файлов в команде ls предназначена опция **-a** или **--all**.

Домашний каталог (домашняя папка, домашняя директория) – предназначен для хранения собственных данных пользователя Linux и личных настроек для программ. Как правило, становится текущим непосредственно после регистрации пользователя в системе.

Полный путь к домашнему каталогу хранится в переменной окружения `HOME`, в полном пути к файлу можно заменять на знак тильды ~, то есть, `/home/user1/file1`, `~/file1` и `$HOME/file1` эквивалентны

Для обычных пользователей домашний каталог находится в директории `/home`.

3.3.9. Управление внешней памятью

В основе файловой системы NFS лежит представление о том, что пользоваться общей файловой системой может произвольный набор клиентов и серверов. Во многих случаях все клиенты и серверы располагаются в одной и той же локальной сети, но это не требуется. Файловая система NFS может также работать через Глобальную сеть (если сервер находится далеко от клиента). Для простоты мы будем говорить о клиентах и серверах так, как если бы они работали на различных компьютерах, хотя файловая система NFS позволяет каждой машине одновременно быть и клиентом, и сервером [1].

Каждый сервер NFS экспортирует один или несколько своих каталогов, предоставляя доступ к ним удаленным клиентам. Как правило, доступ к каталогу предоставляется вместе со всеми его подкаталогами, так что, фактически все дерево каталогов экспортируется как единое целое. Список экспортируемых сервером каталогов хранится в файле (обычно это файл `/etc/exports`), чтобы эти каталоги экспортировались автоматически при загрузке сервера. Клиенты получают доступ к экспортируемым каталогам, монтируя эти каталоги. Если клиент монтирует (удаленный) каталог, то этот каталог становится частью иерархии каталогов клиента.

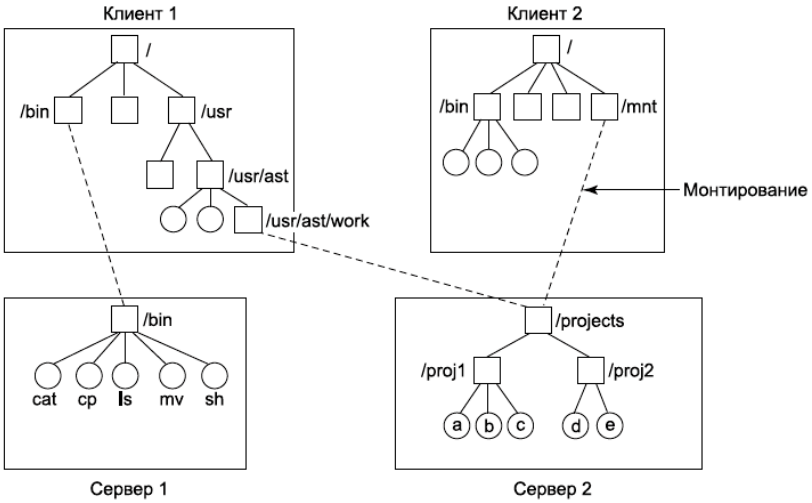

Рис. 3.3.8. Примеры монтирования удаленных файловых систем. Каталоги показаны на рисунке в виде квадратов, а файлы — в виде кружков

В этом примере клиент 1 смонтировал каталог bin сервера 1 в собственном каталоге bin, поэтому он теперь может сослаться на оболочку как на bin/sh и получить оболочку

сервера 1. У бездисковых рабочих станций часто есть только скелет файловой системы (в ОЗУ), и все свои файлы они получают с удаленных серверов (как в данном примере). Аналогично, клиент 1 смонтировал каталог `/projects` сервера 2 в своем каталоге `/usr/ast/work`, поэтому он теперь может получать доступ к файлу a как к `/usr/ast/work/proj1/a`. И наконец, клиент 2 также смонтировал каталог projects и тоже может обращаться к файлу `a`, но уже так: `/mnt/proj1/a`. Как видно из этого примера, у одного и того же файла могут быть различные имена на различных клиентах, так как он может монтироваться в различных местах деревьев каталогов. Точка монтирования является локальной для клиента — сервер не знает, где клиент монтирует его каталог.

Файловой системой NFS поддерживается большинство системных вызовов операционной системы Linux, за исключением (как ни странно) системных вызовов **`open`** и **`close`**.

Исключение системных вызовов **`open`** и **`close`** не случайно. Это сделано преднамеренно. Нет необходимости открывать файл, прежде чем прочитать его. Также не нужно закрывать файл после того, как данные из него прочитаны. Вместо этого, чтобы прочитать файл, клиент посылает на сервер сообщение **`lookup`** (содержащее имя файла) с запросом найти этот файл и вернуть описатель файла, представляющий собой структуру, идентифицирующую файл (то есть содержащую идентификатор файловой системы и номер i-узла вместе с прочей информацией). В отличие от системного вызова **`open`**, операция **`lookup`** не копирует никакой информации во внутренние системные таблицы.

Системный вызов **`read`** содержит описатель файла (который предстоит прочитать), смещение в файле (с которого надо начинать чтение), а также требуемое количество байтов. Таким образом, каждое сообщение является самодостаточным.

Преимущество этой схемы заключается в том, что серверу не нужно помнить (между обращениями к нему) что-либо об открытых соединениях. Поэтому если на сервере произойдет сбой с последующей перезагрузкой, то не будет потеряно никакой информации об открытых файлах, так как ее просто нет. Серверы, которые не поддерживают информации состояния открытых файлов, называются серверами без состояния (stateless).

3.3.10. Кеширование в операционной системе

Цель той части системы, которая занимается операциями ввода-вывода с блочными специальными файлами (например, дисками), заключается в минимизации количества операций передачи данных. Для достижения данной цели в Linux-системах между дисковыми драйверами и файловой системой имеется кэш (рис. 3.2.9). До версии ядра 2.2 система Linux поддерживала отдельные кэш страниц и буферный кэш, так что находящийся в дисковом блоке файл мог кэшироваться в обоих кэшах. Более новые версии Linux имеют единый кэш. Обобщенный уровень блоков связывает эти компоненты вместе и выполняет необходимые преобразования между дисковыми секторами, блоками, буферами и страницами данных.

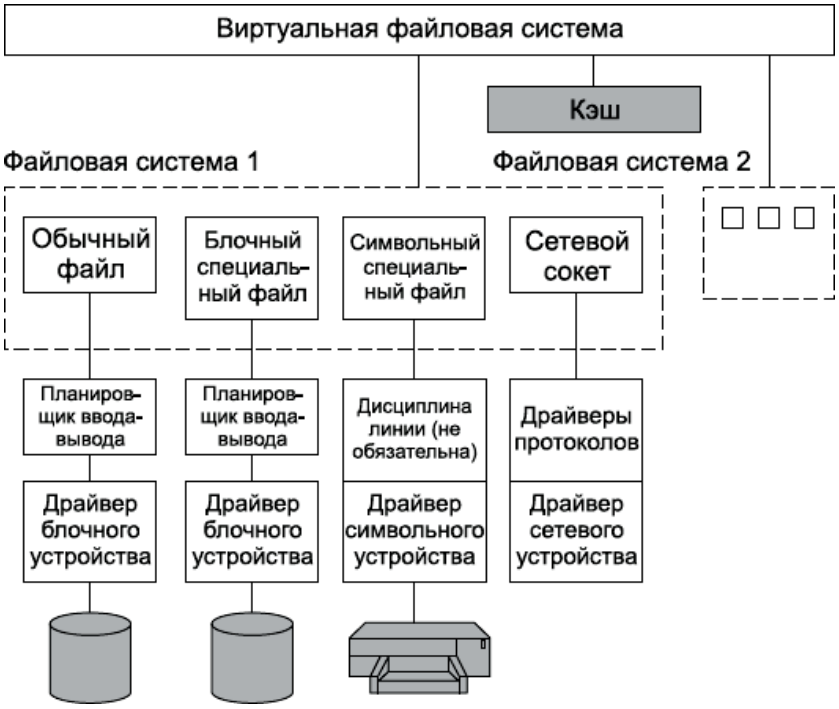

Рис. 3.2.9. Система ввода-вывода в Linux (подробно показана одна файловая система)

Кэш представляет собой таблицу в ядре, в которой хранятся тысячи недавно использованных блоков. Когда файловой системе требуется блок диска (например, блок i-узла, каталога или данных), то сначала проверяется кэш. Если нужный блок есть в кэше, он берется оттуда, при этом обращения к диску удается избежать (что значительно улучшает производительность системы).

Если же блока в кэше страниц нет, то он считывается с диска в кэш, а оттуда копируется туда, куда нужно. Поскольку в кэше страниц есть место только для фиксированного количества блоков, то используется описанный в предыдущем разделе алгоритм замещения страниц.

Кэш страниц работает не только при чтении, но и при записи. Когда программа пишет блок, то этот блок не попадает напрямую на диск, а отправляется в кэш. Демон pdflush сбросит блок на диск тогда, когда кэш вырастет свыше установленного значения. Чтобы блоки не хранились в кэше слишком долго, принудительный сброс на диск «грязных» блоков производится каждые 30 с.

Список использованных источников

1. Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.

2. Алгоритм обработки прерываний по вводу-выводу
https://studfile.net/preview/9460535/page:6/

3. Обслуживание прерывания
https://learn.microsoft.com/ru-ru/windows-hardware/drivers/wdf/servicing-an-interrupt

4. Асинхронный ввод-вывод средствами POSIX
https://ps-group.github.io/os/nonblocking_io_posix

5. Функция CreateFileW (fileapi.h)
https://learn.microsoft.com/ru-ru/windows/win32/api/fileapi/nf-fileapi-createfilew

6. Функция ReadFile (fileapi.h)
https://learn.microsoft.com/ru-ru/windows/win32/api/fileapi/nf-fileapi-readfile

7. Дисковое планирование
https://studfile.net/preview/3718945/page:5/

8. Какую файловую систему следует использовать Ext4 или XFS
https://setiwik.ru/kakuyu-faylovuyu-sistemu-sleduet-ispolzovat-ext4-ili-xfs/

9. Суперблок
https://www.opennet.ru/docs/RUS/linux_base/node67.html

10. Обзор Ext4 vs Btrfs vs XFS
https://losst.pro/obzor-ext4-vs-btrfs-vs-xfs

11. Define the Windows Server file system
https://docs.microsoft.com/en-us/learn/modules/manage-windows-server-file-servers/2-define-windows-server-file-system

12. Что такое файловая система FAT32 - подробное руководство
https://recoverit.wondershare.com.ru/file-system/fat32-file-system.html