Оглавление

1 Типы устройств ввода-вывода

Традиционно в ОС UNIX выделяются два типа организации устройств ввода-вывода.

1.1 Символьный ввод/вывод

Символьный ввод/вывод служит для прямого (без буферизации) выполнения обменов между адресным пространством пользователя и соответствующим устройством. Ядро обеспечивает функцию пересылки данных между пользовательскими и ядерным адресными пространствами.

1.2 Блочный ввод/вывод

Блочный ввод/вывод главным образом предназначен для работы с каталогами и обычными файлами файловой системы, которые на базовом уровне имеют блочную структуру.

На пользовательском уровне можно работать с файлами, прямо отображая их в сегменты виртуальной памяти. Эта возможность рассматривается как верхний уровень блочного ввода/вывода.

На нижнем уровне блочный ввод/вывод поддерживается блочными

драйверами. Блочный ввод/вывод, кроме того, поддерживается системной буферизацией.

1.3 Потоковый ввод/вывод

Похож на символьный ввод/вывод, но имеет возможность включения в поток ввода/вывода промежуточных обрабатывающих модулей и обладает существенно большей гибкостью.

2 Драйверы устройств

Поскольку драйверы главным образом предназначены для управления внешними устройствами, программный код драйвера должен содержать соответствующие средства для обработки прерываний от устройства. Вызов индивидуальной программы обработки прерываний в драйвере происходит из ядра операционной системы.

Подобным же образом в драйвере может быть объявлен вход "timeout", к которому обращается ядро при истечении ранее заказанного драйвером времени (такой временной контроль является необходимым при управлении не слишком интеллектуальными устройствами).

С точки зрения интерфейсов и общесистемного управления различаются два вида драйверов - **символьные и блочные**.

С точки зрения внутренней организации выделяется еще один вид драйверов - **потоковые** (stream) драйверы. Однако, по своему внешнему интерфейсу потоковые драйверы не отличаются от символьных.

2.1 Символьные драйверы

Символьные драйверы главным образом предназначены для обслуживания устройств, обмены с которыми выполняются посимвольно, либо строками символов переменного размера. Типичным примером символьного устройства является простой принтер, принимающий один символ за один обмен.

Символьные драйверы не используют системную буферизацию. Они напрямую копируют данные из памяти пользовательского процесса при выполнении операций записи или в память пользовательского процесса при

выполнении операций чтения, используя собственные буфера.

Следует отметить, что имеется возможность обеспечить символьный интерфейс для блочного устройства.

В этом случае блочный драйвер использует дополнительные возможности процедуры **strategy**, позволяющие выполнять обмен без применения системной буферизации.

Для драйвера, обладающего одновременно блочным и символьным интерфейсами, в файловой системе заводится два специальных файла, блочный и символьный. При каждом обращении драйвер получает информацию о том, в каком режиме он используется.

2.2 Блочные драйверы

Блочные драйверы предназначаются для обслуживания внешних устройств с блочной структурой (магнитных дисков, лент и т.д.) и отличаются от прочих тем, что они разрабатываются и выполняются с использованием системной буферизации. Другими словами, такие драйверы всегда работают через системный буферный пул.

Любое обращение к блочному драйверу для чтения или записи всегда проходит через предварительную обработку, которая заключается в попытке найти копию нужного блока в буферном пуле.

В случае, если копия требуемого блока не находится в буферном пуле или если по какой-либо причине требуется заменить содержимое некоторого обновленного буфера, ядро ОС UNIX обращается к процедуре **strategy** соответствующего блочного драйвера.

Strategy обеспечивает стандартный интерфейс между ядром и драйвером. С использованием библиотечных подпрограмм, предназначенных для написания драйверов, процедура strategy может организовывать очереди обменов с устройством, например, с целью оптимизации движения магнитных головок на диске. Все обмены, выполняемые блочным драйвером, выполняются с буферной памятью. Перепись нужной информации в память соответствующего пользовательского процесса производится программами ядра, заведующими

управлением буферами.

2.3 Драйверы низкого уровня

Обмен данными происходит независимо от файловой подсистемы и буферного кэша, что позволяет ядру производить передачу непосредственно между пользовательским процессом и устройством, без дополнительного копирования в буфер.

2.4 Потоковые драйверы

Основным назначением механизма потоков (streams) является повышение уровня модульности и гибкости драйверов со сложной внутренней логикой (более всего это относится к драйверам, реализующим развитые сетевые протоколы). Спецификой таких драйверов является то, что большая часть программного кода не зависит от особенностей аппаратного устройства. Более того, часто оказывается выгодно по-разному комбинировать части программного кода.

Все это привело к появлению потоковой архитектуры драйверов, которые представляют собой двунаправленный конвейер обрабатывающих модулей. В начале конвейера (ближе всего к пользовательскому процессу) находится заголовок потока, к которому прежде всего поступают обращения по инициативе пользователя. В конце конвейера (ближе всего к устройству) находится обычный драйвер устройства. В промежутке может располагаться произвольное число обрабатывающих модулей, каждый из которых оформляется в соответствии с обязательным потоковым интерфейсом.

2.5 Программные драйверы

3 Технология ввода вывода

3.1 *Системная буферизация ввода/вывода*

Традиционным способом снижения накладных расходов при выполнении обменов с устройствами внешней памяти, имеющими блочную структуру, является буферизация блочного ввода/вывода.

Это означает, что любой блок устройства внешней памяти считывается прежде всего в некоторый буфер области основной памяти, называемой в ОС UNIX системным кэшем, и уже оттуда полностью или частично (в зависимости от вида обмена) копируется в соответствующее пользовательское пространство.

Принципами организации традиционного механизма буферизации является

- во-первых, то, что копия содержимого блока удерживается в системном буфере до тех пор, пока не возникнет необходимость ее замещения по причине нехватки буферов (для организации политики замещения используется разновидность алгоритма LRU).
- во-вторых, при выполнении записи любого блока устройства внешней памяти реально выполняется лишь обновление (или образование и наполнение) буфера кэша.

Действительный обмен с устройством выполняется либо при выталкивании буфера вследствие замещения его содержимого, либо при выполнении специального системного вызова sync (или fsync), поддерживаемого специально для насильственного выталкивания во внешнюю память обновленных буферов кэша.

3.2 *Системные вызовы для управления вводом/выводом*

Для доступа (т.е. для получения возможности последующего выполнения операций ввода/вывода) к файлу любого вида (включая специальные файлы) пользовательский процесс должен выполнить предварительное подключение к файлу с помощью одного из системных вызовов **open, creat**, **dup** или **pipe**.

Последовательность действий системного вызова **open (pathname, mode)**

- анализируется непротиворечивость входных параметров (главным образом, относящихся к флагам режима доступа к файлу);
- выделяется или находится пространство для описателя файла в системной области данных процесса (U-области);

- в общесистемной области выделяется или находится существующее пространство для размещения системного описателя файла (структуры file);
- производится поиск в архиве файловой системы объекта с именем "pathname" и образуется или обнаруживается описатель файла уровня файловой системы(vnode в терминах UNIX V System 4);
- выполняется связывание vnode с ранее образованной структурой file.

Системные вызовы **open** и **creat** (почти) функционально эквивалентны.

Любой существующий файл можно открыть с помощью системного вызова **crea**t, и любой новый файл можно создать с помощью системного вызова **open**.

Однако, применительно к системному вызову **creat** надо подчеркнуть, что в случае своего естественного применения (для создания файла) этот системный вызов создает новый элемент соответствующего каталога (в соответствии с заданным значением pathname), а также создает и соответствующим образом инициализирует новый i-узел.

Наконец, системный вызов **dup** (**duplicate** - скопировать) приводит к образованию нового дескриптора уже открытого файла. Этот специфический для ОС UNIX системный вызов служит исключительно для целей перенаправления ввода/вывода. Его выполнение состоит в том, что в u-области системного пространства пользовательского процесса образуется новый описатель открытого файла, содержащий вновь образованный дескриптор файла (целое число), но ссылающийся на уже существующую общесистемную структуру file и содержащий те же самые признаки и флаги, которые соответствуют открытому файлу-образцу.

Другими важными системными вызовами являются системные вызовы read и write.

Системный вызов **read** выполняется следующим образом:

- в общесистемной таблице файлов находится дескриптор указанного файла, и определяется, законно ли обращение от данного процесса к данному файлу в указанном режиме;

- на некоторое (короткое) время устанавливается синхронизационная блокировка на vnode данного файла (содержимое описателя не должно изменяться в критические моменты операции чтения);
- выполняется собственно чтение с использованием старого или нового механизма буферизации, после чего данные копируются, чтобы стать доступными в пользовательском адресном пространстве.

Операция записи выполняется аналогичным образом, но меняет содержимое буфера буферного пула.

Системный вызов **close** приводит к тому, что драйвер обрывает связь с соответствующим пользовательским процессом и (в случае последнего по времени закрытия устройства) устанавливает общесистемный флаг "драйвер свободен".

Наконец, для специальных файлов поддерживается еще один "специальный" системный вызов **ioctl**.

Это единственный системный вызов, который обеспечивается для специальных файлов и не обеспечивается для остальных разновидностей файлов. Фактически, системный вызов **ioctl** позволяет произвольным образом расширить интерфейс любого драйвера.

Параметры **ioctl** включают код операции и указатель на некоторую область памяти пользовательского процесса. Всю интерпретацию кода операции и соответствующих специфических параметров проводит драйвер.

3.3 Подсистема STREAMS

3.3.1 Архитектура STREAMS

Подсистема STREAMS предоставляет интерфейс обмена данными, основанный на сообщениях, и обеспечивает стандартные механизмы буферизации, управления потоком данных и различную приоритетность обработки.

В STREAMS дублирование кода сводится к минимуму, поскольку однотипные функции обработки реализованы в независимых модулях, которые могут быть использованы различными драйверами.

Подсистема STREAMS поддерживается большинством производителей операционных систем UNIX и является основным способом реализации сетевых драйверов и модулей протоколов. Использование STREAMS охватывает и другие устройства, например, терминальные драйверы.

Подсистема STREAMS обеспечивает создание **потоков - полнодуплексных** коммуникационных каналов между прикладным процессом и драйвером устройства.

Сам поток полностью располагается в пространстве ядра, соответственно и все функции обработки данных выполняются в системном контексте.

Типичный поток состоит из **головного модуля, драйвера и, возможно, одного или более модулей.**

Головной модуль взаимодействует с прикладными процессами через интерфейс системных вызовов.

Драйвер, замыкающий поток, взаимодействует непосредственно с физическим устройством или псевдоустройством, в качестве которого может выступать другой поток. Модули выполняют промежуточную обработку данных.

Процесс взаимодействует с потоком, используя стандартные системные вызовы **open(2), close(2), read(2), write(2) и ioctl(2)**.

Передача данных по потоку осуществляется в виде ***сообщений***, содержащих данные, тип сообщения и управляющую информацию.

Для передачи данных каждый модуль, включая головной модуль и сам драйвер, имеет две очереди — ***очередь чтения*** **(read queue)** и ***очередь записи*** **(write queue)**.

Каждый модуль обеспечивает необходимую обработку данных и передает их в очередь следующего модуля. При этом передача в очередь записи осуществляется вниз по потоку (downstream), а в очередь чтения — вверх по потоку (upstream).

Базовая архитектура потока

Например, из очереди записи модуля 2 сообщение может быть передано в

очередь записи модуля 1, но не наоборот. В свою очередь сообщение из очереди чтения модуля 2 передается в очередь чтения головного модуля, который далее передает данные процессу в ответ на системный вызов *read(2).* Когда процесс выполняет системный вызов *write(2),* данные передаются головному модулю и далее вниз по потоку.

3.3.2 Мультиплексирование

Подсистема STREAMS обеспечивает возможность мультиплексирования потоков с помощью мультиплексора, который может быть реализован только драйвером STREAMS.

Различают три типа мультиплексоров - **верхний, нижний и гибридный.**

Верхний мультиплексор, называемый также мультиплексором **N: 1**, обеспечивает подключение нескольких каналов вверх по потоку к одному каналу вниз по потоку.

Нижний мультиплексор, называемый также мультиплексором **1:M**, обеспечивает подключение нескольких каналов вниз по потоку к одному каналу вверх по потоку.

Гибридный мультиплексор позволяет мультиплексировать несколько каналов вверх по потоку с несколькими каналами вниз по потоку.

Подсистема STREAMS обеспечивает возможность мультиплексирования, но за идентификацию различных каналов и маршрутизацию данных между ними отвечает сам мультиплексор.

Подсистема ввода/вывода, основанная на архитектуре STREAMS, позволяет в полной мере отразить уровневую структуру коммуникационных протоколов, когда каждый уровень имеет стандартные интерфейсы взаимодействия с другими (верхним и нижним) уровнями, и может работать независимо от конкретной реализации протоколов на соседних уровнях.

Архитектура STREAMS полностью соответствует этой модели, позволяя создавать драйверы, которые являются объединениями независимых модулей.

Обмен данными между модулями STREAMS также соответствует характеру взаимодействия отдельных протоколов:

данные передаются в виде сообщений, а каждый модуль выполняет требуемую их обработку.

Реализации протоколов TCP/IP в UNIX System V.

Модуль IP является **гибридным мультиплексором**, позволяя обслуживать несколько потоков, приходящих от драйверов сетевых адаптеров (в данном случае Ethernet и FDDI), и несколько потоков к модулям транспортных протоколов (TCP и UDP)

Модули TCP и UDP являются верхними **мультиплексорами**, обслуживающими прикладные программы, такие как

сервер маршрутизации routed(lM),

сервер удаленного терминального доступа telnetd(W),

сервер FTP ftpd(W),

программы-клиенты пользователей (например talk(l)).

Анализ программного обеспечения сетевой поддержки показывает, что как правило сетевые и транспортные протоколы, составляющие базовый стек TCP/IP, поставляются одним производителем, в то время как поддержка уровней сетевого интерфейса и приложений может осуществляться продуктами различных разработчиков.

Соответственно, можно выделить два основных интерфейса взаимодействия, стандартизация которых позволяет обеспечить совместную работу различных компонентов программного обеспечения.

Первый интерфейс определяет взаимодействие транспортного уровня и уровня приложений и называется **интерфейсом поставщика транспортных услуг** (Transport Provider Interface, TPI).

Второй интерфейс устанавливает правила и формат сообщений, передаваемых между сетевым уровнем и уровнем сетевого интерфейса, и называется **интерфейсом поставщика услуг канала данных** (Data Link Provider Interface, DLPI).

Сетевая архитектура, основанная на архитектуре STREAMS, позволяет обеспечить поддержку любого стека протоколов, соответствующего модели OSI.