СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящее учебное пособие строится на материале, который читается студентам специальности 230101 в первой части дисциплины «Системное программное обеспечение» - операционная система UNIX. В ней рассматриваются основные понятия ОС UNIX, принципы построения и функционирования различных подсистем.

Первая работа, в которой упоминалось о UNIX, была представлена Томпсоном и Ритчи на симпозиуме по операционным системам в октябре 1973 и опубликована в июле 1974 г. Система UNIX начала свое развитие с небольшого набора программ и за эти годы переросла в гибкую ОС, использующуюся для работы огромного количества программных сред и приложений. Операционная система UNIX – это среда выполнения и системные службы, под управлением которых функционируют входящие в набор ОС пользовательские программы, утилиты и библиотеки.

В учебном пособии рассматриваются основные понятия и определения, ставятся основные задачи системного программирования, рассматриваются ресурсы компьютера и операционные системы как средство распределения и управления ресурсами на примере операционной системы UNIX.

Изложены основы организации UNIX, программный интерфейс и пользовательская среда UNIX. Представлена организация системы UNIX в целом: ядро системы, файловая подсистема, подсистема управления процессами, подсистема ввода-вывода.

Пособие рассчитано на студентов, обучающихся по направлению 230100 (бакалавриат) и слушателей высших учебных заведений, обучающихся по техническим дисциплинам. Может быть использовано как при курсовом проектировании, так и при самостоятельном решении задач разработки современного системного программного обеспечения и построения автоматизированных систем.

Для более детального изучения операционной системы UNIX можно рекомендовать следующую литературу:

1. Робачевский А.М. Операционная система UNIX. - СПб.: BHV-Санкт-Петербург, 2002. – 528 с.

2. Вахалия Ю. UNIX изнутри. - СПб.: Питер,2003. –844 с.

3. Митчел М., Джеффри Д., Самьюэл А. Программирование для Linux. Профессиональный подход/ пер. с англ. - М.: Изательский дом Вильямс, 2003. – 288 с.

РАЗДЕЛ 1. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

Программное обеспечение (Software) - совокупность программ и правил, позволяющая использовать ЭВМ для решения различных задач.

По своему назначению программы, выполняемые компьютером, можно разделить на три основные группы:

- системные и сервисные программы;
- языки программирования;
- прикладные программы.

Системное программное обеспечение (СПО, System Software) - совокупность программ и языковых средств, предназначенных для поддержания функционирования ЭВМ и наиболее эффективного выполнения его целевого назначения. По функциональному назначению в СПО можно выделить две системы:

- операционную систему;
- систему программирования.

Системные программы можно разделить на две группы:

а) программы, записанные в постоянную память компьютера и составляющие базовую систему ввода/вывода (BIOS);

б) программы, записанные во внешней памяти, основную часть которых составляет операционная система (ОС).

Системные программы в BIOS являются промежуточным звеном между программным обеспечением компьютера и его электронными компонентами. Эти программы обеспечивают выполнение всех операций ввода/ вывода, соответствующих специфическим особенностям работы каждого из периферийных устройств данного компьютера.

В зарубежной литературе термин «System Software» означает программы и комплексы программ, являющиеся общими для всех, кто совместно использует технические средства компьютера, и применяемые как для автоматизации разработки (создания) новых программ, так и для организации выполнения программ существующих.

С этих позиций системное программное обеспечение может быть разделено на пять групп:

1) операционные системы (ОС);
2) системы управления файлами;
3) интерфейсные оболочки для взаимодействия пользователя с ОС и программные среды;
4) системы программирования;
5) утилиты.

Операционные системы – комплекс управляющих и обрабатывающих программ, который выступает как интерфейс между аппаратурой компьютера (с одной стороны) и пользователем с его задачами, а с другой стороны –

предназначен для наиболее эффективного использования ресурсов вычислительной системы и организации надежных вычислений.

Назначение *систем управления файлами* – организация более удобного доступа к данным, организованным как файлы. Именно благодаря системе управления файлами вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной записи используется логический доступ с указанием имени файла. Следует понимать, что система управления файлами не существует сама по себе – она разработана для конкретной ОС, с конкретной файловой системой. Можно сказать, что всем известная файловая система FAT имеет множество реализаций как система управления файлами, например, FAT16 для MS DOS, superFAT – для OS/2, FAT32 для WinNT и т.д.

Для удобства взаимодействия с ОС могут использоваться дополнительные *интерфейсные оболочки*. Их основное назначение – либо расширить возможности по управлению ОС, либо изменить встроенные в систему возможности. Например, варианты графического интерфейса X Windows (K Desktop Environment в Linux), PM Shell в OS/2 и т.д.

Операционная среда – интерфейс, необходимый программам для обращения к ОС с целью получить определенный сервис – выполнить операцию ввода-вывода, получить или освободить участок памяти.

Системы программирования - совокупность языка программирования и соответствующего ему языкового процессора, обеспечивающие автоматизацию отработки и отладки программ. Программные компоненты системы программирования работают под управлением операционной системы наравне с прикладными программами пользователя: отладчики, компоновщики, редакторы, трансляторы, библиотеки подпрограмм.

Утилиты – специальные системные программы, с помощью которых можно обслуживать как саму ОС, так и подготавливать для работы носители данных, осуществлять оптимизацию размещения данных на носителе и т.д.

РАЗДЕЛ 2. АРХИТЕКТУРА UNIX

2.1. История развития ОС UNIX

Этимология слова UNIX, как рассказывает в своей книге Питер Салюс, обязана своим происхождением шуткам коллег-хакеров. Multics (проект был закрыт в марте 1969 г.) была многопользовательской системой, а первая UNIX работала всего с двумя пользователями. Латинский корень «много» заменили на «один» («единственный»). Получилось - UNICS (Uniplexed Information and Computing Service). Позже название было изменено на UNIX.

Несмотря на достаточно скромный «выход в свет» UNIX завоевала популярность - во второй половине 80-х годов XX века ее выбрали многие коммерческие организации, учебные заведения и научно-исследовательские лаборатории. Чаще всего результатом приобретения того или иного варианта системы был осознанный выбор. Позже позиции UNIX были потеснены корпорацией Microsoft, предлагающей операционные системы семейства Windows. Постепенно ОС UNIX стала проигрывать в битве за рынок настольных систем.

Почему же система UNIX остается такой же мощной и популярной, как и 40 лет назад? Причин много. Это и переносимость, и конструктивные особенности этой операционной системы, а главное философия, заложенная в ее фундаменте. Многие видят в UNIX запутанную, сложную и непонятную систему. На самом деле – это яркий пример разработки системы по принципу KISS (Keep It Simple, Stupid). Конструктивно UNIX базируется на множестве небольших программ, каждая из которых отлично решает совершенно определенную задачу. Кроме того, разработчики заложили замечательную идею: при необходимости несколько маленьких, не связанных между собой программ можно объединить в единое целое, решая тем самым задачи любой сложности. Исходная философия по-прежнему живет в основных командах, доступных на всех системах UNIX. Ее ключевые элементы:

- простые команды;
- команды, соединенные каналами (pipe);
- преимущественно общий стиль интерфейса;
- отсутствие типов файлов;
- возможность работы в сети и высокая устойчивость системы (в конце 70-х годов Министерство обороны США принимает решение - использовать в ARPANET в качестве операционной системы UNIX).

С момента создания в 1969 г. система была перенесена на множество различных аппаратных платформ, появилось большое количество ее реализаций, созданных коммерческими компаниями, учебными заведениями и научно–исследовательскими организациями. На сегодняшний день существуют варианты UNIX для самых различных аппаратных платформ, начиная с небольших встроенных процессоров, рабочих станций и настольных систем, и

заканчивая высокопроизводительными многопроцессорными системами, объединяющими большое количество пользователей.

Существует несколько вариантов системы UNIX. Часть из них – основополагающих (базовых):

- это различные реализации системы SYSTEM V компании AT&T (на сегодняшний день последняя версия System V под названием SVR4 (System V Release 4) является собственностью корпорации Novell);
- реализации системы BSD (Berkley Software Distribution) Калифорнийского университета в Беркли, (NETBSD – FreeBSD – самая известная из семейства BSD);
- Mach университета Карнеги-Меллона;
- XENIX - совместное производство корпораций MICROSOFT и SCO (одна из первых коммерческих);
- OSF/1 организации Open Software Foundation;
- также SunOS и Solaris, поставляемые компанией Sun Microsystems;
- Digital UNIX компании Digital Equipment Corporation;
- HP-UX корпорации Hewlett-Packard Corporation.

2.2. Стандарты

Чем больше появлялось версий UNIX, тем очевиднее становилась необходимость стандартизации системы. В результате возникло несколько организаций, связанных со стандартизацией, и был разработан ряд стандартов, оказывающих влияние на развитие UNIX.

В 1980 году была создана инициативная группа под названием /usr/group с целью стандартизации программного интерфейса UNIX, т.е. формального определения услуг, предоставляемых операционной системой приложениям. Решение этой задачи упростило бы переносимость приложений между различными версиями UNIX. Такой стандарт был создан в 1984 году и использовался комитетом ANSI, отвечающим за стандартизацию языка C, при описании библиотек. Однако с ростом числа версий операционной системы эффективность стандарта уменьшилась, и через год, в 1985 году, был создан Portable Operating System Interface for Computing Environment, сокращенно POSIX (переносимый интерфейс операционной системы для вычислительной среды).

В 1988 году группой был разработан стандарт POSIX 1003.1-1988, который определил программный интерфейс приложений (Application Programming Interface, API). Этот стандарт нашел широкое применение во многих операционных системах, в том числе и с архитектурой, отличной от UNIX. Спустя два года стандарт был принят как стандарт IEEE 1003.1-1990. Заметим, что поскольку этот стандарт определяет интерфейс, а не конкретную реализацию, он не делает различия между системными вызовами и библиотечными

функциями, называя все элементы программного интерфейса просто функциями.

Другими наиболее значительными стандартами POSIX, относящимися к UNIX, являются:

- POSIX 1003.2-1992 - включает определение командного интерпретатора UNIX и набора утилит;
- POSIX 1003.1 b-1993 - содержит дополнения, относящиеся к поддержке приложений реального времени;
- POSIX 1003.1c-1995 - включает определения "нитей" (threads) POSIX, известных также как pthreads.

Кроме перечисленных стандартов, следует отметить документ XPG3, который появился в 1992 году. Он относится к стандарту на графическую систему X Windows. В 1996 году организация OSF и X/OPEN объединились для того, чтобы разработать пользовательский интерфейс и его сопряжение со специальной графической оболочкой CDE – Common Desktop Environment и DCE – Distributed computing – для распределенных вычислительных сред.

2.3. Общие черты UNIX

К общим чертам операционных систем семейства UNIX можно отнести следующее:

- *Мультипрограммная обработка* в режиме разделения времени, основанная на вытесняющей многозадачности.
- *Поддержка многопользовательского режима,* наличие средств защиты данных от несанкционированного доступа.
- *Использование механизмов виртуальной памяти и свопинга.*
- *Иерархическая файловая система,* образующая единое дерево каталогов независимо от количества физических устройств, используемых для размещения файлов.
- *Унификация операций ввода/вывода* на основе расширенного использования понятия «файл».
- *Переносимость системы.* Большая часть ОС UNIX написана на машинно-независимом языке С. Эта часть является переносимой. Та часть ядра, которая не может быть машинно-независимой, имеет небольшой размер, что облегчает модификацию этой части ОС при переходе на новую платформу
- *Кэширование диска* для уменьшения среднего времени доступа к файлам.
- *Наличие разнообразных средств взаимодействия процессов, в том числе и через сеть.*

Преимущества UNIX

1) Способ распространения.

UNIX завоевала высокую степень популярности, на которую, возможно, даже не рассчитывали ее разработчики. Одной из причин успеха являлся способ распространения системы. Корпорация AT&T, ограниченная законами в своих возможностях, продавала лицензии и исходные коды системы по достаточно

низкой цене, поэтому UNIX стала популярной среди многих пользователей по всему миру. Так как в комплект поставки входили и исходные коды, пользователи имели возможность экспериментировать с ними, улучшать их, а также обмениваться друг с другом созданными изменениями. Многие из новшеств корпорация встраивала в следующие версии системы.

2) «Красота в краткости».

Оригинальная версия UNIX имела хороший дизайн, который являлся базисом последующего успеха более поздних реализаций и вариантов системы. Одну из сильнейших сторон системы можно охарактеризовать выражением «красота в краткости». Ядро небольшого размера имело минимальный набор основных служб. Механизм конвейера совместно с программируемой оболочкой позволял комбинировать эти утилиты различными способами, создавая мощные, производительные инструменты.

3) Файловая система.

В отличие от других операционных систем, обладающих сложными методами доступа к файлам, такими как ISAM (Indexed Sequential Access Method) - индексированный последовательный доступ к файлам или ISAP (Hierarchical Sequential Access Method) - иерархический последовательный доступ к файлам, система UNIX интерпретирует файлы как простую последовательный байтов. Приложения могут записывать в содержимое файлов любую структуру и использовать различные методы доступа, не ставя об этом в известность операционную систему.

4) Представление данных.

Многие системные приложения используют для представления своих данных символьные строки. К примеру, важнейшие базы данных, такие как /etc/passwd, /etc/fstab и /etc/ttys являются обычными текстовыми файлами. Представление в виде текста позволило пользователям легко просматривать и производить манипуляции с этими файлами без применения специальных инструментов. Текст является общей, универсальной и обладающей высокой степенью переносимости формой данных, которые можно легко обрабатывать множеством различных утилит.

5)Унифицированный интерфейс с устройствами ввода-вывода.

Еще одной особенностью системы UNIX стал простой унифицированный интерфейс с устройствами ввода-вывода. Представляя все устройства как файлы, система позволяет пользователям применять один и тот же набор команд и системных вызовов для доступа и работы с различными устройствами, равно как и для работы с файлами. Разработчики могут создавать программы, производящие ввод-вывод, не заботясь, с чем именно производится обмен- с файлом, терминалом пользователя, принтером или другим устройством. Таким образом, используемый совместно с перенаправлением данных интерфейс ввода-вывода системы UNIX – простой и одновременно мощный.

6) Высокая степень переносимости.

Причиной успеха и распространения UNIX стала ее высокая степень переносимости. Большая часть ядра системы написана на языке С, что позволило относительно легко адаптировать UNIX к новым аппаратным платформам. Многие производители «железа» после создания новых компьютеров имеют возможность просто перенести на них уже имеющуюся систему UNIX, вместо того, чтобы создать для своих разработок операционную систему заново.

7) Управление с удаленного терминала без использования специального программного обеспечения.

8) Большое количество программного обеспечения, в том числе и бесплатного.

Недостатки Unix

На конференции USENIX в 1987 г. Денис Ритчи сделал доклад, в котором провел наиболее объективный анализ недостатков системы. Как сказал Ритчи: «UNIX является простой и понятной системой, но чтобы ее понять и принять ее простоту, требуется гений (или, как минимум программист)».

- Большинству пользователей нужна не сама ОС, а в первую очередь возможность выполнять определенные приложения. Пользователей не интересовала элегантность структуры файловой системы или модели вычислений. Они хотели работать с определенными программами (например, текстовыми редакторами), потратив на это минимум расходов и усилий. Недостатки простого унифицированного (обычно графического) пользовательского интерфейса в первых системах UNIX были основной причиной его неприятия массами.

- Получилось так, что элегантность и эстетичность, свойственная UNIX, требует от пользователей, желающих эффективно работать в системе, творческого мышления и определенной изобретательности. Однако большинство пользователей предпочитают простые в изучении интегрированные многофункциональные программы.

- Хотя UNIX изначально была весьма простой системой, это вскоре закончилось. Например, была добавлена буферизация данных в стандартную библиотеку ввода-вывода. Это повысило ее эффективность и сделало программы переносимыми на системы, отличные от UNIX. Но со временем библиотека разрасталась и стала более сложной, чем системный вызов, лежащий в ее основе. Системные вызовы read и write были неделимыми операциями над файлами, в то время как буферизация библиотеки ввода-вывода уничтожила эту целостность.

Кроме того, простота лицензионных условий и переноса на различные аппаратные платформы стали причиной неконтролируемого роста и лавинообразного распространения различных реализаций ОС. Пользователи имели право вносить свои изменения в систему. И как следствие – несовместимые между собой варианты ОС UNIX.

2.4. Модель системы UNIX

Основной функцией операционной системы является представление среды, под управлением которой могут выполняться пользовательские программы (приложения). Система определяет базовую структуру для выполнения программ, а также предлагает набор различных служб, например, такие операции работы с файлами или ввод-вывод, и предоставляет интерфейс взаимодействия с ними.

Самый общий взгляд позволяет увидеть двухуровневую модель системы так, как она представлена на рис. 2.1.

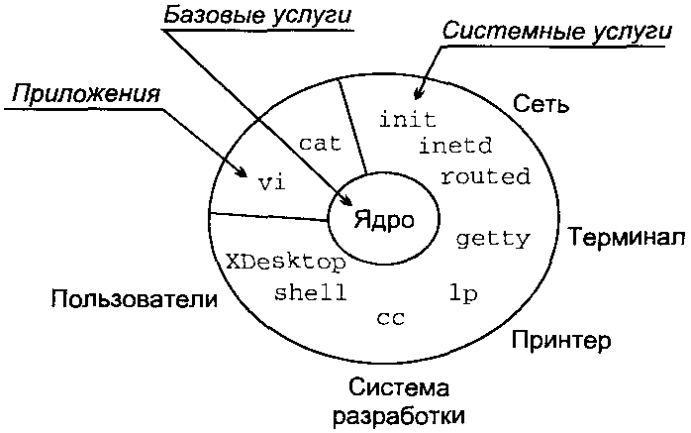


Рис. 2.1. Модель системы UNIX

В центре находится ядро системы (kernel). Ядро непосредственно взаимодействует с аппаратной частью компьютера, изолируя прикладные программы от особенностей ее архитектуры. Ядро имеет набор услуг, предоставляемых прикладным программам. К услугам ядра относятся операции ввода/вывода (открытия, чтения, записи и управления файлами), создания и управления процессами, их синхронизации и межпроцессного взаимодействия. Все приложения запрашивают услуги ядра посредством системных вызовов.

Второй уровень составляют приложения или задачи, как системные, определяющие функциональность системы, так и прикладные, обеспечивающие пользовательский интерфейс UNIX. Однако, несмотря на внешнюю разнородность приложений, схемы их взаимодействия с ядром одинаковы.

2.4.1. Ядро системы

Ядро обеспечивает базовую функциональность операционной системы: создает процессы и управляет ими, распределяет память и обеспечивает доступ к файлам и периферийным устройствам.

Взаимодействие прикладных задач с ядром происходит посредством стандартного интерфейса системных вызовов. *Интерфейс системных вызовов* представляет собой набор услуг ядра и определяет формат запросов на услуги. Процесс запрашивает услугу посредством системного вызова определенной процедуры ядра, внешне похожего на обычный вызов библиотечной функции.

Ядро от имени процесса выполняет запрос и возвращает процессу необходимые данные. Например, программа открывает файл, считывает из него данные и закрывает этот файл. При этом операции открытия (open), чтения (read) и закрытия (close) файла выполняются ядром по запросу задачи, а функции *open(), read()* и *close()* являются системными вызовами.

Структура ядра представлена на рис. 2.2.

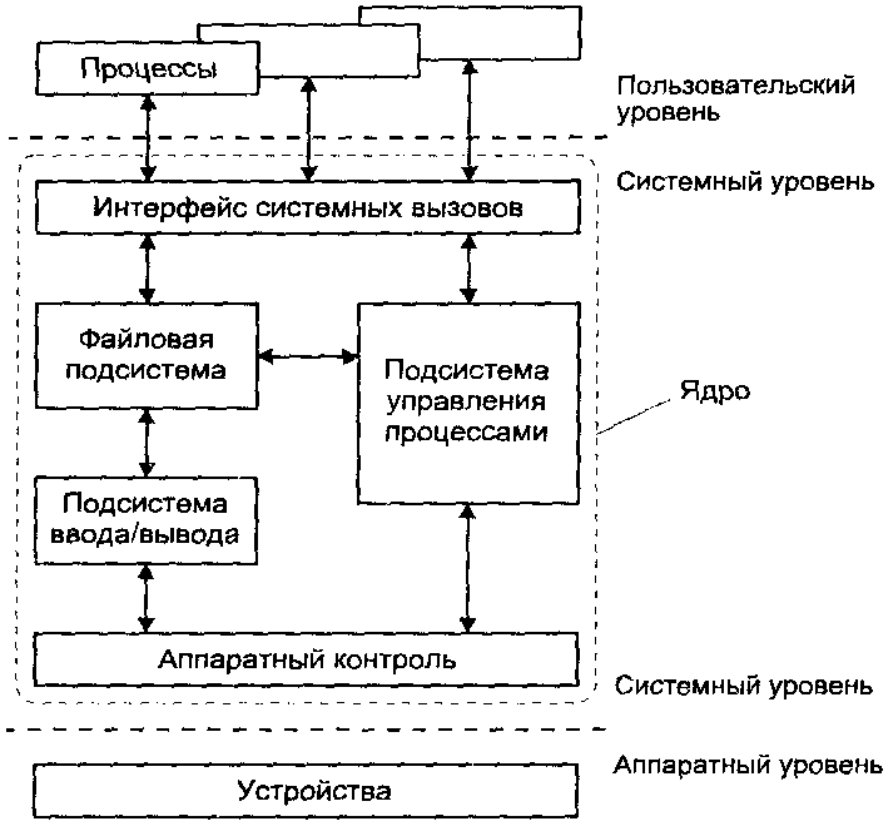


Рис. 2.2. Внутренняя структура ядра UNIX

Ядро состоит из трех основных подсистем:
- файловая подсистема;
- подсистема управления процессами и памятью;
- подсистема ввода/вывода.

2.4.2. Файловая подсистема

Файловая подсистема обеспечивает унифицированный интерфейс доступа к данным, расположенным на дисковых накопителях, и к периферийным устройствам. Одни и те же функции *open(), read(), write()* могут использоваться как при чтении или записи данных на диск, так и при выводе текста на принтер или терминал.

Файловая подсистема контролирует права доступа к файлу, выполняет операции размещения и удаления файла, а также выполняет запись/чтение данных файла. Поскольку большинство прикладных функций выполняется

через интерфейс файловой системы (в том числе и доступ к периферийным устройствам), права доступа к файлам определяют привилегии пользователя в системе.

Файловая подсистема обеспечивает перенаправление запросов, адресованных периферийным устройствам, соответствующим модулям подсистемы ввода/вывода.

2.4.3. Подсистема управления процессами

Запущенная на выполнение программа порождает в системе один или более *процессов* (или *задач).* Подсистема управления процессами контролирует:

- создание и удаление процессов;
- распределение системных ресурсов (памяти, вычислительных ресурсов) между процессами;
- синхронизацию процессов;
- межпроцессное взаимодействие.

Очевидно, что в общем случае число активных процессов превышает число процессоров компьютера, но в каждый конкретный момент времени на каждом процессоре может выполняться только один процесс. Операционная система управляет доступом процессов к вычислительным ресурсам, создавая ощущение одновременного выполнения нескольких задач.

Специальная задача ядра, называемая *распорядителем* или *планировщиком* процессов (scheduler), разрешает конфликты между процессами в конкуренции за системные ресурсы (процессор, память, устройства ввода/вывода). Планировщик запускает процесс на выполнение, следя за тем, чтобы процесс монопольно не захватил разделяемые системные ресурсы. Процесс освобождает процессор, ожидая длительной операции ввода/вывода, или по прошествии кванта времени. В этом случае планировщик выбирает следующий процесс с наивысшим приоритетом и запускает его на выполнение.

Модуль управления памятью обеспечивает размещение оперативной памяти для прикладных задач. Оперативная память является дорогостоящим ресурсом и, как правило, ее редко бывает "слишком много". В случае если для всех процессов недостаточно памяти, ядро перемещает части процесса или нескольких процессов во вторичную память (как правило, в специальную область жесткого диска), освобождая ресурсы для выполняющегося процесса. Все современные системы реализуют так называемую *виртуальную память,* процесс выполняется в собственном логическом адресном пространстве, которое может значительно превышать доступную физическую память. Управление виртуальной памятью процесса также входит в задачи модуля управления памятью.

Модуль межпроцессного взаимодействия отвечает за уведомление процессов о событиях с помощью сигналов и обеспечивает возможность передачи данных между различными процессами.

2.4.4. Подсистема ввода/вывода

Подсистема ввода/вывода выполняет запросы файловой подсистемы и подсистемы управления процессами для доступа к периферийным устройствам (дискам, магнитным лентам, терминалам и т. д.). Она обеспечивает необходимую буферизацию данных и взаимодействует с *драйверами устройств* - специальными модулями ядра, непосредственно обслуживающими внешние устройства.

Контрольные вопросы

1. Что понимается под операционной системой UNIX?
2. Перечислите основные достоинства и недостатки ОС UNIX.
3. Перечислите ключевые элементы ОС UNIX.
4. Перечислите основные компоненты архитектуры UNIX.
4. Перечислите основные документы, регламентирующие переносимость приложений между различными версиями UNIX.
6. Что включает в себя внутренняя структура ядра?
7. Назовите основные функции файловой подсистемы.
8. Назначение подсистемы управления вводом-выводом.
9. Назовите функции контроля подсистемы управления процессами и памятью.
10. Перечислите основные функции модуля управления памятью.

РАЗДЕЛ 3. ФАЙЛЫ И ФАЙЛОВАЯ СИСТЕМА ОС UNIX

3.1. Типы файлов

Файлы в UNIX играют ключевую роль, что не всегда справедливо для других операционных систем. Каждый файл имеет имя, определяющее его расположение в дереве файловой системы. Корнем этого дерева является корневой каталог (root directory), имеющий имя «/». Имена всех остальных файлов содержат путь – список каталогов (ветвей), которые необходимо пройти, чтобы достичь файла. В UNIX все доступное пользователям файловое пространство объединено в единое дерево каталогов, корнем которого является каталог «/». Таким образом, полное имя любого файла начинается с «/» и не содержит идентификатора устройства (дискового накопителя, CD-ROM или удаленного компьютера в сети), на котором он фактически хранится.

Имена файлов могут иметь произвольную длину. Буквы в нижнем и верхнем регистрах различаются операционной системой. В некоторых интерпретаторах (bash и tcsh) имеется механизм, облегчающий набор длинных имен файла. Достаточно набрать первые символы имени файла, а затем нажать клавишу табуляции Tab и будет осуществлено дополнение или выдан список файлов, содержащих данные символы.

В UNIX существует 6 типов файлов, различающихся по функциональному назначению и действиям операционной системы при выполнении тех или иных операций над файлами:

- обычный файл (regular file);
- каталог (directory);
- специальный файл устройства (special device file);
- FIFO или именованный канал (named pipe);
- связь (link);
- сокет.

Обычный файл - представляет собой наиболее общий тип файлов, содержащий данные в некотором формате. Для ОС это просто последовательность байтов. Вся интерпретация содержимого файла производится прикладной программой, обрабатывающей файл. К этим файлам относятся текстовые, бинарные данные, исполняемые программы и так далее.

Каталог (directory) - это файл, содержащий имена находящихся в нем файлов, а также указатели на дополнительную информацию – метаданные, позволяющие операционной системе производить операции над этими файлами. С помощью каталогов формируется логическое дерево файловой системы. Каталоги определяют положение файла в дереве файловой системы, поскольку сам файл не содержит информации о своем местонахождении. Любая задача, имеющая право на чтение каталога, может прочесть его содержимое, но только ядро имеет право на запись в каталог. По существу каталог представляет собой таблицу, каждая запись которой соответствует

некоторому файлу. Первое поле каждой записи содержит указатель на методанные (номер inode), а второе определяет имя файла.

Специальный файл устройства (special device file) обеспечивает доступ к физическому устройству. В UNIX различают *символьные* (character) и *блочные* (block) файлы устройств. Доступ к устройствам осуществляется путем открытия, чтения и записи в специальный файл устройства. Символьные файлы устройств используются для небуферизированного обмена данными с устройством, в противоположность этому блочные файлы позволяют производить обмен данными в виде пакетов фиксированной длины - *блоков.* Доступ к некоторым устройствам может осуществляться как через символьные, так и через блочные специальные файлы.

FIFO или именованный канал (named pipe) - это файл, используемый для связи между процессами. FIFO впервые появились в System V UNIX, но большинство современных систем поддерживают этот механизм.

Связь (link) - как уже говорилось, каталог содержит имена файлов и указатели на их метаданные. В то же время сами метаданные не содержат ни имени файла, ни указателя на это имя. Такая архитектура позволяет одному файлу иметь несколько имен в файловой системе. Имена жестко связаны с метаданными и соответственно с данными файла, в то время как сам файл существует независимо от того, как его называют в файловой системе. Такая связь имени файла с его данными называется *жесткой связью* (hard link). Например, с помощью команды *1n* мы можем создать еще одно имя **(file2)** файла, на который указывает имя **file1 :**

$ pwd

/home/student

$ ln file1 /home/student/file2

Жесткие связи абсолютно равноправны. В списках файлов каталогов, которые можно получить с помощью команды *ls,* файлы **file1** и **file2** будут отличаться только именем. Все остальные атрибуты файла будут абсолютно одинаковыми. С точки зрения пользователя - это два разных файла. Изменения, внесенные в любой из этих файлов, затронут и другой, поскольку оба они ссылаются на одни и те же данные файла. Вы можете переместить один из файлов в другой каталог — все равно эти имена будут связаны жесткой связью с данными файла. Легко проверить, что удаление одного из файлов **(file1** или **file2)** не приведет к удалению самого файла, т. е. его метаданных и данных (если это не специальный файл устройства).

Следует иметь в виду, что жесткая связь является естественной формой связи имени файла с его метаданными и не принадлежит к особому типу файла. Особым типом файла является *символическая связь,* позволяющая косвенно адресовать файл. В отличие от жесткой связи, символическая связь

адресует файл, который, в свою очередь, ссылается на другой файл. В результате, последний файл адресуется символической связью косвенно. Данные файла, являющегося символической связью, содержат только имя целевого файла.

Символическая связь является особым типом файла (об этом свидетельствует символ *l* в первой позиции вывода *ls),* и операционная система работает с таким файлом не так, как с обычным.

Сокеты - предназначены для взаимодействия между процессами. Интерфейс сокетов часто используются для доступа к сети TCP/IP. В системах ветви BSD UNIX на базе сокетов реализована система межпроцессного взаимодействия, с помощью которой работают многие системные сервисы, например, система печати.

3.2. Структура файловой системы UNIX

Использование общепринятых имен основных файлов и структуры каталогов существенно облегчает работу в операционной системе, ее администрирование и переносимость. Эта структура используется в работе системы, например, при ее инициализации и конфигурировании, при работе почтовой системы и системы печати. Нарушение этой структуры может привести к неработоспособности системы или отдельных ее компонентов. Файловая система основана на модели иерархического дерева каталогов и с этой точки зрения аналогична файловой системе Windows и MS-DOS. Однако в отличие от названных систем в UNIX отсутствует понятие логического устройства (диска), все каталоги являются подкаталогами единого дерева и начинаются с так называемого корневого каталога. Корневой каталог системы обозначается символом /, подкаталог корневого каталога с именем КАТАЛОГ1: обозначается /КАТАЛОГ1, подкаталог этого каталога - /КАТАЛОГ1/КАТАЛОГ2, а файл, находящийся в каталоге /КАТАЛОГ1 обозначается /КАТАЛОГ1/ФАЙЛ1 (то есть с точки зрения обозначения, никакой разницы между файлами и каталогами нет). Структура файловой системы UNIX приведена на рис. 3.1:

/bin

В каталоге /bin находятся наиболее часто употребляемые команды и утилиты системы, как правило, общего пользования.

/dev

Каталог /dev содержит специальные файлы устройств, являющиеся интерфейсом доступа к периферийным устройствам. Каталог /dev может содержать несколько подкаталогов, группирующих специальные файлы устройств одного типа. Например, каталог /dev/dsk содержит специальные файлы устройств для доступа к гибким и жестким дискам системы.

/etc

В этом каталоге находятся системные конфигурационные файлы и многие утилиты администрирования. Среди наиболее важных файлов — скрипты инициализации системы. Эти скрипты хранятся в каталогах /etc/rcO.d, /etc/rcl.d,

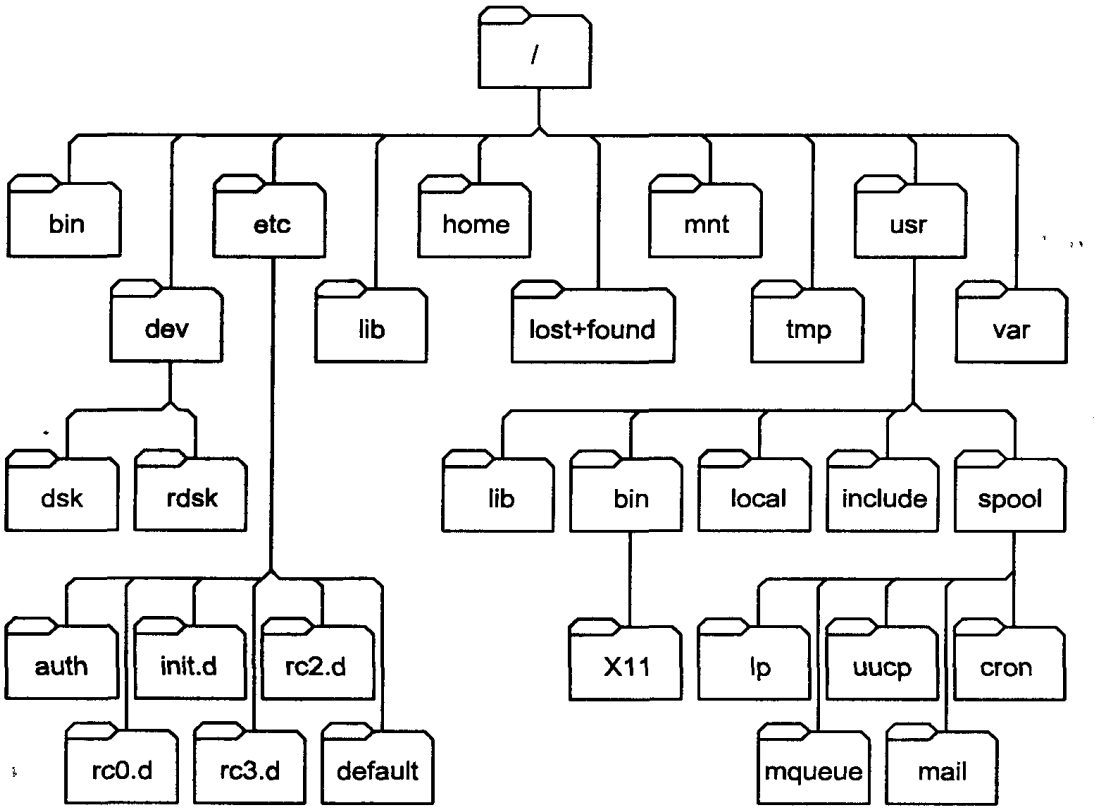


Рис. 3.1 Структура файловой системы UNIX

/etc/rc2.d и т. д., соответствующих уровням выполнения системы (run level), и управляются скриптами /etc/rcO, /etc/rcl, /etc/rc2 и т. д. Во многих версиях BSD

UNIX указанные каталоги отсутствуют, и загрузка системы управляется скриптами /etc/rc.boot, /etc/rc и /etc/rc.Iocal. В UNIX ветви System V здесь находится подкаталог default, где хранятся параметры по умолчанию многих команд (например, /etc/default/su содержит параметры для команды *su* (смена пользователя). В UNIX System V большинство исполняемых файлов перемещены в каталог /sbin или /usr/sbin.

/lib

В каталоге /lib находятся библиотечные файлы языка С и других языков программирования. Стандартные названия библиотечных файлов имеют вид libx.a (или libx.so), где x — это один или более символов, определяющих содержимое библиотеки. Например, стандартная библиотека С называется libc.a, библиотека системы X Window System имеет имя libXll.a. Часть библиотечных файлов также находится в каталоге /usr/lib.

/lost+found

Каталог "потерянных" файлов. Ошибки целостности файловой системы, возникающие при неправильном останове UNIX или аппаратных сбоях, могут привести к появлению т. н. "безымянных" файлов — структура и содержимое файла являются правильными, однако для него отсутствует имя в каком-либо из каталогов. Программы проверки и восстановления файловой системы помещают такие файлы в каталог /lost+found под системными числовыми именами.

/mnt

Стандартный каталог для временного связывания (монтирования) физических файловых систем к корневой для получения единого дерева логической файловой системы. Обычно содержимое каталога **/mnt** пусто, поскольку при монтировании он перекрывается связанной файловой системой.

/u или ***/home***

Общеупотребительный каталог для размещения домашних каталогов пользователей. Например, имя домашнего каталога пользователя student будет, скорее всего, называться **/home/student** или **/u/student.** В более ранних версиях UNIX домашние каталоги пользователей размещались в каталоге /usr.

/usr

В этом каталоге находятся подкаталоги различных сервисных подсистем — системы печати, электронной почты и т. д. **(/usr/spool),** исполняемые файлы утилит UNIX (/usr/bin), дополнительные программы, используемые на данном компьютере (/usr/local), файлы заголовков **(/usr/include),** электронные справочники (/usr/man)n т. д.

/var

В UNIX System V этот каталог является заменителем каталога **/usr/spool,** используемого для хранения временных файлов различных сервисных подсистем — системы печати, электронной почты и т. д.

/tmp

Каталог хранения временных файлов, необходимых для работы различных подсистем UNIX. Обычно этот каталог открыт на запись для всех пользователей системы.

В системе FreeBSD существует еще ряд ключевых элементов файловой системы:

/boot

Каталог содержит конфигурационные и исполняемые файлы, необходимые для загрузки системы, кроме того, начиная с версии 5.0, в этом каталоге находится ядро.

/modules

В этом каталоге находятся загружаемые модули ядра.

/sys@

Ссылка на файлы с исходным кодом ядра, если они были установлены.

В именах файлов и каталогов могут встречаться практически любые символы (причем прописные и строчные буквы различаются), однако далеко не

все программы могут работать с именами, в которых используется, например, символ звездочка (*). Необходимо проявлять особую осторожность и при работе с файлами, содержащими в именах символы с кодами, превышающими 127 (например, буквы национальных алфавитов).

3.3. Владельцы файлов

Файлы в UNIX имеют двух владельцев: пользователя (user owner) и группу (group owner). *Группой* называется определенный список пользователей системы. Пользователь системы может быть членом нескольких групп, одна из которых является *первичной* (primary), остальные — *дополнительными* (supplementary).

Важной особенностью является то, что владелец-пользователь может не являться членом группы, владеющей файлом.

Прежде чем вы сможете начать работу в UNIX, вы должны стать *пользователем системы,* т. е. получить имя, пароль и ряд других атрибутов.

С точки зрения системы, пользователь - не обязательно человек. Пользователь является объектом, который обладает определенными правами, может запускать на выполнение программы и владеть файлами. В качестве пользователей могут, например, выступать удаленные компьютеры или группы пользователей с одинаковыми правами и функциями. Такие пользователи называются *псевдопользователями.* Они обладают правами на определенные файлы системы и от их имени запускаются задачи, обеспечивающие ту или иную функциональность UNIX.

Как правило, большинство пользователей являются реальными людьми, которые регистрируются в системе, запускают те или иные программы, короче говоря, используют UNIX в своей работе.

В системе существует один пользователь, обладающий неограниченными правами. Это *суперпользователь* или *администратор системы.*

Каждый пользователь системы имеет уникальное *имя* (или *регистрационное имя* — login name). Однако система различает пользователей по ассоциированному с именем *идентификатору пользователя* или UID (User Identifier). Идентификаторы пользователя также должны быть уникальными. Пользователь является членом одной или нескольких *групп* — списков пользователей, имеющих сходные задачи (например, пользователей, работающих над одним проектом). Принадлежность к группе определяет дополнительные права, которыми обладают все пользователи группы. Каждая группа имеет уникальное имя (уникальное среди имен групп, имя группы и пользователя могут совпадать), но, как и для пользователя, внутренним представлением группы является ее идентификатор GID (Group Identifier). В конечном счете, UID и GID определяют, какими правами обладает пользователь в системе.

Вся информация о пользователях хранится в файле **/etc/passwd.** Это обычный текстовый файл, право на чтение которого имеют все пользователи

системы, а право на запись имеет только администратор (суперпользователь). В этом файле хранятся пароли пользователей (в зашифрованном виде). Подобная открытость — недостаток с точки зрения безопасности, поэтому во многих системах зашифрованные пароли хранятся в отдельном закрытом для чтения и записи файле **/etc/shadow.**

Аналогично, информация о группах хранится в файле **/etc/group** и содержит списки пользователей, принадлежащих той или иной группе.

Какую же информацию хранит система о пользователе? Для этого рассмотрим фрагмент файла **/etc/passwd:**

root:x:0:l:0000-Admin(0000):/:/bin/sh
daemon:x:1:1:0000-Admin(0000):/:
bin:x:2:2:0000-Admin(0000):/usr/bin:
sys:x:3:3:0000-Admin(0000):/:
adm:x:4:4:0000-Admin(0000):/var/adm:
lp:x:71:8:0000-lp(0000):/usr/spool/lp:
uucp:x:5:5:0000-uucp(0000):/usr/lib/uucp:
nobody:x:60001:60001:uid no body:/:
user:x:500:500:USER EVM:/home/user:/bin/sh

Каждая строка файла является записью конкретного пользователя и имеет следующий формат:

name:passwd-encod: UID:GID: comments:home-dir:shell

- т.е. семь полей (атрибутов), разделенных двоеточиями, где:

name - регистрационное имя пользователя. Это имя пользователь вводит в ответ на приглашение системы login. Для небольших систем имя пользователя назначается произвольно. В больших системах, в которых зарегистрированы сотни пользователей, требования уникальности заставляют применять определенные правила выбора имен;

passwd-encod - пароль пользователя в закодированном виде. Алгоритмы кодирования известны, но они не позволяют декодировать пароль. При входе в систему пароль, который вы набираете, кодируется, и результат сравнивается с полем *passwd-encod.* В случае совпадения пользователю разрешается войти в систему. Даже в закодированном виде доступность пароля представляет некоторую угрозу для безопасности системы. Поэтому часто пароль хранят в отдельном файле, а в поле passwd-encod ставится символ X (в некоторых системах !). Пользователь, в данном поле которого стоит символ '*', никогда не сможет попасть в систему. Дело в том, что алгоритм кодирования не позволяет символу '*' появиться в закодированной строке. Таким образом, совпадение введенного и затем закодированного пароля и '*' ;

UID - идентификатор пользователя, является внутренним представлением пользователя в системе. Этот идентификатор наследуется

задачами, которые запускает пользователь, и файлами, которые он создает. По этому идентификатору система проверяет пользовательские права (например, при запуске программы или чтении файла). Суперпользователь имеет UID = 0, что дает ему неограниченные права в системе.

GID - определяет *идентификатор первичной группы пользователя.* Этот идентификатор соответствует идентификатору в файле **/etc/group,** который содержит имя группы и полный список пользователей, являющихся ее членами. Принадлежность пользователя к группе определяет дополнительные права в системе. Группа определяет общие для всех членов права доступа и тем самым обеспечивает возможность совместной работы (например, совместного использования файлов);

comments - обычно, это полное "реальное" имя пользователя. Это поле может содержать дополнительную информацию, например, телефон или адрес электронной почты. Некоторые программы (например, *finger()* и почтовые системы) используют это поле;

home-dir - домашний каталог пользователя. При входе в систему пользователь оказывается в этом каталоге. Как правило, пользователь имеет ограниченные права в других частях файловой системы, но домашний каталог и его подкаталоги определяют область файловой системы, где он является полноправным хозяином;

shell - имя программы, которую UNIX использует в качестве командного интерпретатора. При входе пользователя в систему UNIX автоматически запустит указанную программу. Обычно это один из стандартных командных интерпретаторов **/bin/sh** (Bourne shell), **/bin/csh** (C shell) или **/bin/ksh** (Korn shell), позволяющих пользователю вводить команды и запускать задачи. В принципе, в этом поле может быть указана любая программа, например, командный интерпретатор с ограниченными функциями (restricted shell), клиент системы управления базой данных или даже редактор. Важно то, что, завершив выполнение этой задачи, пользователь автоматически выйдет из системы. Некоторые системы имеют файл **/etc/shells,** содержащий список программ, которые могут быть использованы в качестве командного интерпретатора.

3.4. Стандартные пользователи и группы

После установки UNIX обычно уже содержит несколько зарегистрированных пользователей. Перечислим основные из них (в разных версиях системы UID этих пользователей могут незначительно отличаться):

Имя	**Пользователь**
root	Суперпользователь, администратор системы, UID=0. Пользователь с этим именем имеет неограниченные полномочия в системе. Для него не проверяются права доступа, и таким образом он имеет все "рычаги" для управления системой. Для выполнения большинства функций

	администрирования требуется вход именно с этим именем. Следует отметить, что root - это только имя. На самом деле значение имеет UID. Любой пользователь с UID=0 имеет полномочия суперпользователя.
adm	Псевдопользователь, владеющий файлами системы ведения журналов
bin	Обычно это владелец всех исполняемых файлов, являющихся командами Unix
cron	Псевдопользователь, владеющий соответствующими файлами, от имени которого выполняются процессы подсистемы запуска программ по расписанию
1p или lpd	Псевдопользователь, от имени которого выполняются процессы системы печати, владеющий соответствующими файлами
news	Псевдопользователь, от имени которого выполняются процессы системы телеконференций
nobody	Псевдопользователь, используемый в работе NFS
uucp	Псевдопользователь подсистемы UNIX - to-UNIX copy , позволяющей передавать почтовые сообщения и файлы между UNIX-хостами.

Новая система также содержит ряд предустановленных групп. Поскольку группы, как правило, менее значимы, приведем лишь две категории:

root или wheel	Административная группа, GID=0
user или users или staff	Группа, в которую по умолчанию включаются все обычные пользователи

3.5. Права доступа к файлу

В ОС UNIX существуют три базовых класса доступа к файлу, в каждом из которых установлены соответствующие права доступа:

User access (u)	Для владельца-пользователя файла;
Group access (g)	Для членов группы, являющейся владельцем файла;
Other access (o)	Для остальных пользователей (кроме суперпользователя).

UNIX поддерживает три типа прав доступа для каждого класса: на чтение (read, обозначается символом г), на запись (write, обозначается символом w) и на выполнение (execute, обозначается символом x).

С помощью команды *Is -l* можно получить список прав доступа к файлу:

-rw-r—г—	*1*	*andy*	*group*	*36482*	*Dec*	*22*	*19:13*	*report.txt.l*
drwxr-xr—	*2*	*andy*	*group*	*64*	*Aug*	*15*	*11:03*	*temp*
-rwxr-xr—	*1*	*andy*	*group*	*4889*	*Dec*	*22*	*15:13*	*a.out*

-rw-r—r— 1 andy group 7622 Feb 11 09:13 cont.c

Права доступа листинга отображаются в первой колонке (за исключением первого символа, обозначающего тип файла). Наличие права доступа обозначается соответствующим символом, а отсутствие — символом '-'. Рассмотрим, например, права доступа к файлу **a.out:**

Тип файла	Права владельца-пользователя	Права владельца-группы	Права остальных пользователей
-	r w x	r - x	r - -
обычный файл	Чтение, запись, выполнение	Чтение и выполнение	Только чтение

Права доступа могут быть изменены только владельцем файла или суперпользователем (superuser) — администратором системы. Для этого используется команда *chmod().* Ниже приведен общий формат этой команды.

chmod [u g o a] [+ − =] [r w x] *file*1 *file*2...

В качестве аргументов команда принимает указание классов доступа ('u' — владелец-пользователь, 'g' — владелец-группа, 'o[1] — остальные пользователи, 'a — все классы пользователей), права доступа ('г — чтение, 'w' — запись и ‘x’ — выполнение) и операцию, которую необходимо произвести ('+' — добавить, '-' — удалить и '=' — присвоить) для списка файлов *fuel, file2* и т.д.

Например, команда $ *chmod g-wx ownflie* лишит членов группы-владельца файла *ownfile* права на запись и выполнение этого файла.

В одной команде можно задавать различные права для нескольких классов доступа, разделив их запятыми. Приведем еще несколько примеров:

$ chmod a+w text — Предоставить право на запись для всех пользователей;

$ chmod go=r text — Установить право на чтение для всех пользователей, за исключением владельца;

$ chmod g+x-w runme — Добавить для группы право на выполнение файла runme и снять право на запись;

$ chmod u+w,og+r-w textl text2 — Добавить право записи для владельца, право на чтение для группы и остальных пользователей, отключить право на запись для всех пользователей, исключая владельца

3.6. Мнемоника названий специальных файлов устройств в файловой системе UNIX

Названия специальных файлов устройств в большой степени зависят от конкретной версии UNIX. Тем не менее, в этих названиях присутствует

общая логика, позволяющая даже в незнакомой системе определить, какие файлы отвечают за конкретные устройства. Например, имена файлов доступа к дисковым устройствам обычно содержат указание на тип диска, номер контроллера, логический номер устройства, раздел диска и т. д. По названию также легко определить, какой вид доступа предоставляет данный интерфейс (блочный или символьный).

В качестве примера рассмотрим специальный файл устройства для доступа к разделу диска в операционной системе Solaris:

/dev/dsk/cOt4dOs2.

Данный файл предоставляет блочный интерфейс, а соответствующий ему символьный (или необработанный) файл имеет имя:

/dev/rdsk/cOt4dOs2.

Файлы доступа к дисковым устройствам располагаются в специальных подкаталогах — **/dev/dsk** (для блочных устройств) и **/dev/rdsk** (для символьных устройств). Такая структура хранения специальных файлов характерна для систем UNIX версии System V.

Имя файла, характерное для систем версии SVR4, можно представить в общем виде:

cktldmsn,

где *k* — номер контроллера, *l* — номер устройства (для устройств SCSI это идентификатор устройства ID); *m* — номер раздела, n — логический номер устройства (LUN) SCSI.

Таким образом, файл устройства **/dev/rdsk/cOt4dOs2** обеспечивает доступ к первому разделу (нумерация разделов начинается с 0) диска с ID=4, LUN=2 первого контроллера.

Такой формат имен файлов в версии SVR4 применяется для всех дисковых устройств и накопителей на магнитной ленте. Иногда для этих стандартных имен в файловой системе имеются символические связи с более простыми названиями. Например, в Solaris имя **/dev/sdOa** может использоваться вместо **/dev/dsk/cOt3dOs,** также обеспечивая доступ к устройству:

lrwxrwxrwx 1 root root 12 Jan 27 11:48 /dev/sdOa ->dsk/cOt3dOs

В SCO UNIX имеются специальные файлы с более простыми именами **/dev/root, /dev/usr** и т.п., которые предоставляют доступ к разделам диска с такими же именами (root, usr).

Более простая мнемоника обнаруживается в именах специальных файлов других устройств. Так, например, параллельный порт в большинстве систем имеет имя /dev/lpn, где n — номер порта (О, 1 и т. д.). Терминальные линии, подключенные к последовательным портам компьютера, обозначаются как /dev/ttynn, где nn является идентификатором линии. В табл.1 приведены примеры имен других специальных файлов устройств.

Таблица 1

Имена некоторых специальных файлов устройств

Общий вид имени	Описание устройства, доступ к которому обеспечивается через файл
/dev/rmtn	Накопитель на магнитной ленте
/dev/nrmtn /dev/nrmtO	Накопитель на магнитной ленте в режиме без перемотки назад по окончании работы
/dev/rstn	SCSI-накопитель на магнитной ленте
/dev/cdn	CD-ROM
/dev/cdrom	
/dev/ttypn	Псевдотерминал (подчиненный)
/dev/ptypn	Псевдотерминал (мастер)
/dev/console	Системная консоль
/dev/tty	Синоним терминальной линии управляющего терминала для данного процесса
/dev/mem	Физическая оперативная память
/dev/kmem	Виртуальная память ядра
/dev/null	Нулевое устройство — весь вывод на него уничтожается, а при попытке ввода с этого устройства возвращается 0 байтов
/dev/zero	Нулевое устройство — весь вывод на него уничтожается, а ввод приводит к получению последовательности нулей

Имена физических устройств компьютера для ОС ASPLinux выглядят как имена файлов в подкаталоге первого уровня /dev (и действительно являются файлами особого вида). Разделы жесткого диска с интерфейсом IDE (EIDE) имеют имена вида **/dev/hdXY**, где X — это одна из букв a, b, c, d, обозначающие соответственно с 1 по 4 физический диск (от Primary Master до Secondary Slave), а Y — число, обозначающее номер раздела на диске (разделы нумеруются в том порядке, в котором они перечислены в таблице разделов диска). Например, единственный раздел второго (Slave) диска, присоединенного к первичному (Primary) контроллеру, обозначается /dev/hdb1.

Контрольные вопросы

1. Перечислите типы файлов в ОС UNIX.
2. Перечислите уровни безопасности по отношению к ресурсам ОС UNIX.
3. Что является основой любой файловой системы UNIX.
4. Что содержит каталог /bin, /dev, /home, /etc, /u, /lost+found, /usr/spool, /usr/local, /usr/bin, /usr/insclude?

5. Перечислите основных пользователей системы.
6. Какую информацию содержится запись файла /etc/passwd ?
7. Где и в каком поле записи указывается командный интерпретатор пользователя?
8. Перечислите файлы, относящиеся к служебным учетным записям UNIX.
9. Расшифруйте поля файла /etc/group: user::10: dave, lory, james .
10. Назначение файла /etc/shell.
11. Какие существуют соглашения для имен специальных файлов устройств?

РАЗДЕЛ 4. ФАЙЛОВАЯ ПОДСИСТЕМА

Файловая система в UNIX может располагаться на жестком диске, CD-ROM, дискетах, на удаленном компьютере и других носителях.

Традиционно файловая система UNIX расположена на жестком диске компьютера, хотя существует специальная файловая система NFS (Network File System), обеспечивающая хранение файлов на удаленном компьютере.

Исконной файловой системой UNIX System V считается s5fs. Появившаяся позже FFS (начиная с BSD 4.2), обладает лучшей производительностью, функциональностью, надежностью.

Современные файловые системы UNIX имеют довольно сложную архитектуру для различных версий, но все поддерживают две базовых системы: s5fs (System V) и FFS (BSD).

Для поддержки нескольких типов файловых систем введена независимая или *виртуальная файловая система,* т.е. архитектура, позволяющая обеспечить работу с несколькими «физическими» файловыми системами различных типов.

4.1. Базовая файловая система System V

В UNIX в качестве независимых устройств, доступ к которым осуществляется как к различным носителям данных, выступают разделы (partions) – логические части дела.

В каждом разделе может располагаться только одна файловая система.

Файловая система s5fs занимает раздел диска и состоит из трех основных компонент, представленных на рис. 4.1:

- суперблок (superblock);
- массив индексных дескрипторов (ilist);
- блоки хранения данных.

Рис. 4.1. Состав s5fs

4.1.1. Суперблок

В каждой файловой системе существует суперблок. Он считывается в память при монтировании файловой системы и находится там до размонтирования. Суперблок содержит общую информацию о файловой системе, необходимую для монтирования и управления работой файловой системы:

- тип файловой системы (s_type);
- размер файловой системы в логических блоках, включая суперблок ilist и блоки хранения данных (s_fsize);
- размер массива индексных дескрипторов (s_isize);

число свободных блоков, доступных для размещения (s_tfree);

- число свободных inode, доступных для размещения (s_tinode);
- флаги:
- модификации s_fmod;
- режима монтирования (s_fronly);
- размер логического блока (512, 1024, 2048);
- список номеров свободных inode;
- список адресов свободных блоков.

Так как число свободных inode и блоков хранения данных может быть значительным, хранить последние полностью в суперблоке нецелесообразно.

Следует иметь в виду, что когда число свободных индексных дескрипторов в частичном списке, хранящемся в суперблоке стремится к нулю, ядро просматривает массив индексных дескрипторов ilist и вновь формирует список свободных inode. Для этого ядро анализирует поле di_mode (тип файла, дополнительные атрибуты выполнения и права доступа) индексного дескриптора, которое равно нулю у свободных inode.

Однако такой подход не применим в отношении свободных блоков данных, так как нельзя определить по содержимому блока – свободен он или нет. Поэтому необходимо хранить список адресов свободных блоков целиком. Суперблок содержит только один блок из списка адресов свободных блоков. Первый элемент этого блока указывает на блок, хранящий продолжение списка и т.д.

Выделение свободных блоков для размещения файла производится в конце списка суперблока. Когда остается в списке единственный элемент, ядро интерпретирует его как указатель на блок, содержащий продолжение списка. В этом случае содержимое этого блока считывается в суперблок и блок становится свободным.

Таким образом, дисковое пространство под списки используется пропорционально свободному месту в файловой системе.

4.1.2. Массив индексных дескрипторов

Индексный дескриптор (inode) содержит статусную информацию о файле, необходимую для обработки данных, и указывает на расположение

данных этого файла. Ядро обращается к inode по индексу в массиве ilist. Один inode является корневым (root) inode файловой системы, через него осуществляется доступ к структуре каталогов и файлов после монтирования файловой системы.

Размер массива ilist является фиксированным и задается при создании файловой системы, т.е. s5fs имеет ограничения по числу файлов, которые могут в ней храниться, независимо от размера этих файлов.

Структура дискового inode представлена на рис. 4.2.

Поле di_mode хранит несколько атрибутов файла: тип файла, дополнительные атрибуты выполнения и права доступа: IFREG – для обычных файлов; IFDIR – для каталогов; IFBLK /IFCHR – для специальных файлов блочных и символьных устройств соответственно.

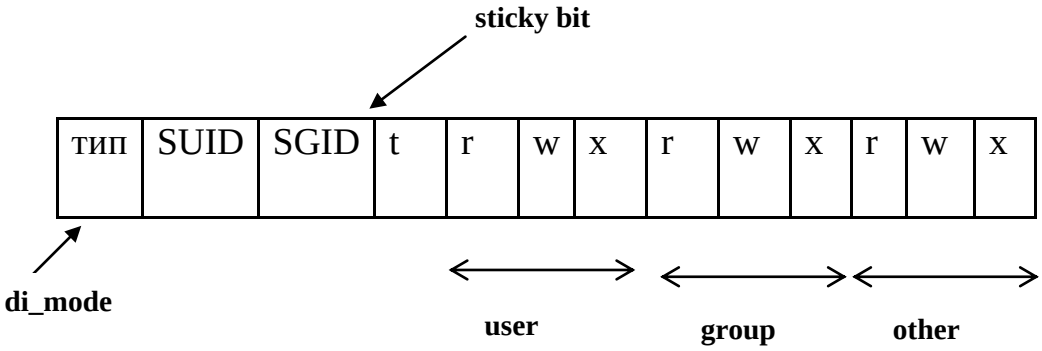


Рис. 4.2. Структура дискового inode

Дополнительные атрибуты выполнения:

- sticky bity – позволяет пользователю удалять только файлы, которыми он владеет или имеет право на запись, т.е. дополнительная защита файлов, находящихся в данном каталоге.

- SUID (Set UID) - позволяет изменить правило установки владельца–создаваемых файлов.

- SGID (Set GID) – позволяет изменить правило установки владельца–группы создаваемых файлов. При установке этого атрибута для каталога вновь созданные файлы этого каталога будут наследовать владельца-группу по владельцу-группе каталога, т.е. для System V удается имитировать поведение систем BSD, для которых такое правило наследования действует по умолчанию.

В индексном дескрипторе отсутствует информация о времени создания файла. Так же inode хранит три значения времени:

- время последнего доступа (di_atime);
- время последней модификации (di_mtime);
- время последней модификации метаданных (di_ctime), не учитываются изменения полей 1,2. Значение di_itime изменяется при изменении: размера файла, владельца, группы, числа связей.

Индексный дескриптор хранит физические адреса всех блоков, принадлежащих данному файлу. Эта информация хранится в виде массива, каждый элемент которого содержит физический адрес дискового блока, а индексом массива является номер логического блока файла.

Массив имеет фиксированный размер и состоит из 13 элементов: первые десять – адресуют непосредственно блоки хранения данных файла; 11-й элемент адресует блок, в свою очередь содержащий адреса блоков хранения данных файла; 12-й элемент указывает на дисковый блок, хранящий адреса блоков, каждый их которых адресует блок хранения данных файла; 13-й элемент используется для тройной косвенной адресации, когда для нахождения адреса блока хранения данных файла используется три дополнительных блока.

Такой подход позволяет при относительно небольшом фиксированном размере индексного дескриптора поддерживать работу с файлами, размер которых может варьироваться. При размере блока -1024 байта для файлов до 10 Кбайт используется прямая индексация, обеспечивающая высокую производительность. Для файлов не более 266 Кбайт (10 Кбайт + 256х1024) – простая косвенная адресация. При 3-ой косвенной адресации можно обеспечить доступ к 16777216 (256х256х256) блокам.

Файлы в UNIX могут содержать «дыры». Например, процесс создает пустой файл, помощью системного вызова *lseek* смещает файловый указатель относительно начала файла и записывает данные. При этом между началом файла и началом записанных данных образуется дыра – незаполненная область. При чтении процесс получит обнуленные байты, так как логические блоки, соответствующие дыре, не содержат данные, для них не размещают дисковые блоки (нет смысла).

В этом случае соответствующие элементы массива inode содержат нулевой указатель.

Процесс производит чтение такого блока, и как следствие, ядро возвращает последовательность нулей.

Имена файлов хранятся в файлах специального типа – каталогах. Каталог файловой системы s5fs представляет собой таблицу, каждый элемент которой имеет фиксированный размер в 16 байтов:

- 2 байта хранят номер индексного дескриптора файла;
- 14 байтов – его имя (существует ограничение – не более 14 символов).

Все это накладывает ограничения на число inode – не более 65535.

Первые два элемента каталога адресуют сам каталог, под именем «**.**» и родительский каталог под именем «**..**».

При удалении имени файла (например, rm …) номер inode устанавливается равным нулю. Но ядро не удаляет такие свободные элементы, поэтому размер каталога не уменьшается, и как следствие – проблема – объем памяти.

Например, команда, обеспечивающая вывод не интерпретированного содержимого файла (16-тиричный дамп):

```
$ hd .
:
0000fc        0a        …..   dead.letta
:
0060da        91        …..    @bin
:        029000        00        …..        . jpg
02a000        00        …..
```

Можно заметить, что имен файлов, расположенных во 2-й части вывода команды *hd* на самом деле не существует, так как inode – нулевые. Тоже самое подтвердит команда

$ ls –a.

Подведем итоги о файловой системе s5fs:

Достоинства	*Недостатки*
простота	- низкая надежность и производительность (слабое звено наличие суперблока в одном экземпляре, метаданные хранятся в начале файловой системы, далее блоки хранения данных и как следствие, постоянное перемещение головки (аналогия- фрагментация в DOS)); - не оптимально используется дисковое пространство (желательно использовать блоки больших размеров и как следствие, считывание большего количества данных за одну операцию ввода-вывода); - массив inode имеет фиксированный размер, задаваемый при создании файловой системы. Это накладывает ограничения на максимальное число файлов, которые могут существовать и, как следствие, нехватка inode или дисковых блоков для хранения файлов большего размера; -ограничения на длину имени и максимальное количество inode является слишком жестким.

Устранение вышеперечисленных недостатков привело к разработке новой архитектуры файловой системы – FFS (Berkley Fast File System), впервые появившейся в версии 4.2 BSD UNIX.

4.2. Файловая система FFS

Данная файловая система использует те же структуры данных ядра. Основные изменения:

- расположение файловой системы на диске;
- дисковые структуры данных;
- алгоритмы размещения свободных блоков.

Суперблок FFS в отличие от суперблока s5fs не хранит данные о свободном пространстве файловой системы, такие как массив свободных блоков и inode. Поэтому данные суперблока не меняются на протяжении всего времени существования файловой системы. Данные суперблока для повышения надежности дублируются (рис. 4.3).

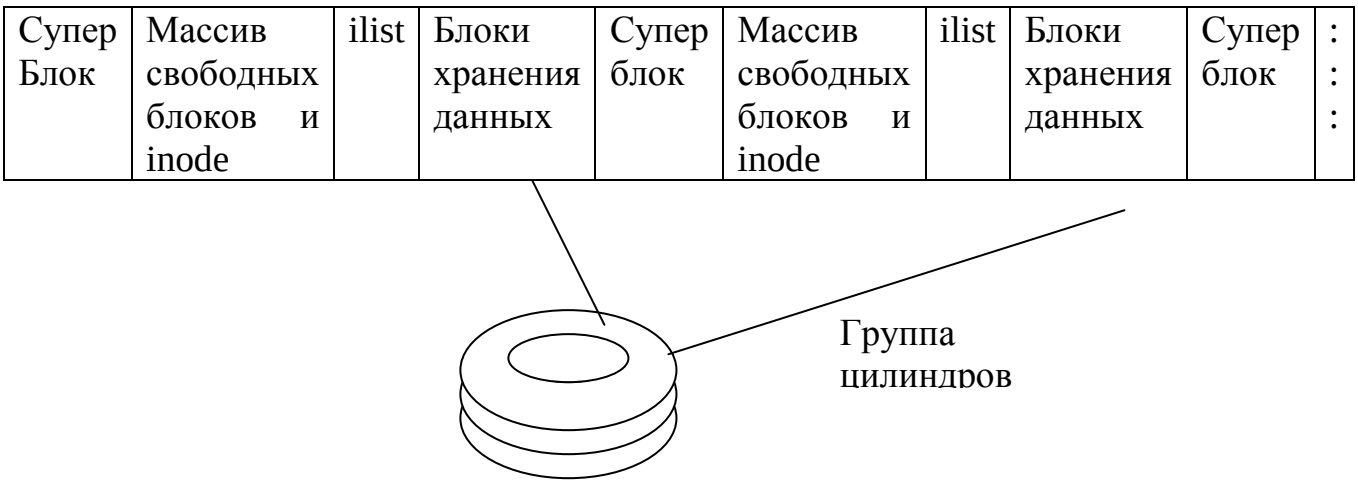


Рис. 4.3. Структура данных ядра

Организация файловой системы предусматривает логическое деление дискового раздела на одну или несколько групп цилиндров (cylinder group). Группа цилиндров представляет собой несколько последовательных дисковых цилиндров. Каждая группа цилиндров содержит управляющую информацию, включающую:

- резервную копию суперблока;
- массив inode;
- данные о свободных блоках;
- итоговую информацию об использовании дисковых блоков в группе.

Для каждой группы цилиндров при создании файловой системы выделяется место под определенное количество inode, на каждые 2 Кбайт блоков хранения данных создается один inode. Так как размеры группы

цилиндров и массива inode фиксированы, в BSD UNIX присутствуют ограничения, аналогичные s5fs.

Для уменьшения вероятности потери всех индексных дескрипторов в результате сбоя, уменьшения времени доступа к данным конкретного файла предложено:

- создание кластеров inode, распределенных по всему разделу, а не сгруппированных inode в начале;

- управляющая информация располагается с различным смещением от начала группы цилиндров. Смещение равно одному сектору относительно предыдущей группы, т.е. управляющая информация для соседних групп начинается на различных пластинах дисков, (это позволяет при выходе из строя одной пластины не потерять информацию, что было бы, если управляющая информация находилась бы на одной пластине).

Т.е. потеря одного сектора, цилиндра или пластины не приведет к потере всех копий суперблоков.

Производительность файловой системы существенно зависит от размера блока хранения данных. Чем больше блок, тем больше количество данных будет прочитано без перемещения дисковой головки. Файловая система FFS поддерживает размер блока до 64 Кбайт. Так как файловая система UNIX состоит из файлов небольшого размера, то имеет место частичная занятость блока и как следствие, потеря до 60 процентов полезной емкости диска. Этот недостаток преодолен следующим образом: возможность фрагментации блока (может быть разделен на 2, 4 или 8 фрагментов). Если блок является единицей передачи данных в операциях ввода/вывода, то *фрагмент* определяет адресуемую единицу хранения данных на диске. Так был найден компромисс между производительностью ввода/вывода и эффективностью хранения данных.

Размер фрагмента задается при создании файловой системы (максимальное значение – 0,5 размера блока, минимальное – физическим ограничением дискового устройства, а именно, размером *сектора* – минимальной единицей адресации диска).

Информация о свободном пространстве в группе хранится в виде битовой карты блоков, а не в виде списка свободных блоков, как в s5fs. Карта блоков, связанная с определенной группой цилиндров, описывает свободное пространство во фрагментах. Ядро анализирует биты фрагментов, составляющих блок.

Изменения в алгоритмах размещения свободных блоков

Если в s5fs свободные блоки и inode выбираются из конца соответствующего списка, и как следствие, значительный разброс данных файла по разделу диска, то в FFS используются следующие принципы, направленные на повышение производительности:

- файл размещается (по возможности) в блоке хранения данных, принадлежащий одной группе цилиндров, где расположены его метаданные (уменьшается время доступа);

- все файлы каталога по возможности размещаются в одной группе цилиндров (увеличивает скорость последовательного доступа к этим файлам);

- каждый новый каталог по возможности помещается в группу цилиндров, отличную от группы родительского каталога (таким образом, достигается равномерное распределение данных по диску);

- между последовательными блоками должен быть по возможности пропуск из нескольких секторов, так как между временем завершения чтения одного и второго блоков есть промежуток времени, что исключает «холостые» обороты диска.

Каталоги

Структура каталога файловой системы FFS была изменена для поддержки длинных имен (до 255 символов). Запись каталога представлена структурой, имеющей следующие поля (рис. 4.4):

№ inode (индекс в массиве ilist)	длина записи	длина имени файла	имя файла

d_ino d_reclen d_namlen d_name []

Рис. 4.4. Структура записи каталога

Запись не фиксированной длины, а структура. Имя файла имеет переменную длину, максимальная длина – 4 байта, дополняется нулями.

Если имя файла удаляется, то запись, принадлежащая ему, присоединяется к предыдущей, и значение поля d_reclen увеличивается на соответствующую длину.

Если удаляется первая запись, то полю d_ino присваивается нулевое значение.

4.3. Архитектура виртуальной файловой системы

Любую файловую систему разделяют на зависимый и независимый (общий) от реализации уровни.

Общий уровень представляет для остальных подсистем ядра некоторую абстрактную файловую систему. Иногда этот уровень называется *виртуальной файловой системой (vfs)*. При этом дополнительные файловые системы могут быть встроены в ядро UNIX, подобно драйверам устройств (рис. 4.5).

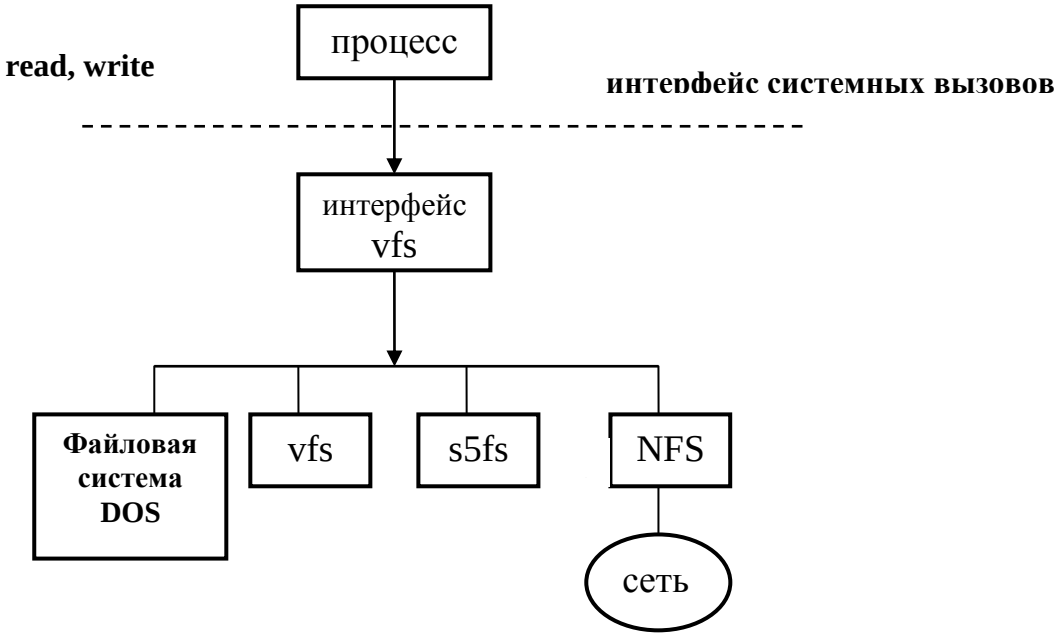


Рис. 4.5. Архитектура vfs

Виртуальная файловая система основана на представлении метаданных файла в виде сходных с традиционной семантикой UNIX.

Интерфейсом работы с файлами является *vnode* (virtual inode) – *виртуальный индексный дескриптор.* Метаданные всех активных файлов (файлов, на которые ссылается один или более процессов) представлены в памяти в виде in-core inode, в качестве которых в виртуальной файловой системе выступают inode. Структура данных vnode одинакова для всех файлов, независимо от типа реальной файловой системы. Данные содержат:

- информацию, необходимую для работы виртуальной файловой системы;
- неизмененные характеристики файла (например, тип файла).

Каждый vnode содержит число ссылок v_count, который увеличивается при открытии процессом файла и уменьшении его при закрытии. Когда число ссылок равно нулю, вызывается операция vn_inactive, которая сообщает реальной файловой системе, что никто больше не ссылается. После этого файловая система может освободить vnode, поместить его в кэш для дальнейшего использования. Поле v_vfsp указывает на файловую систему, в которой расположен файл, адрес vnode.

4.4. Монтирование файловой системы

Для того чтобы сделать доступными файловые системы, размещенные на физических устройствах, требуется совместить корневые каталоги этих устройств некоторыми подкаталогами файловой системы. Только после этого ядро сможет выполнять файловые операции такие как, создание, открытие, чтение/запись в файл.

Операция встраивания получила название *подключение* или *монтирование файловой системы.*

Каждая подключенная файловая система представлена на независимом уровне в виде структуры vfs, аналоге записи таблицы монтирования файловой системы.

Структуры всех подключенных файловых систем организованы в виде односвязного списка. Этот список называется *таблицей монтирования.* Первой записью всегда является корневая файловая система. На рис. 4.6 представлены поля структуры vfs.

Следующая файловая система в списке монтирования (*vfs_next)	Операции файловой системы (*vfs_op)	vnode, перекрываемый файловой системой (*vfs_vnodecovered)	Флаги: только для чтения, запрещен бит SUID и т.д. (vfs_flag)	Размер блока файловой системы vfs_bsize	Указатель на специфические данные, относящиеся к реальной файловой системе

Рис. 4.6. Поля структуры vfs

Операции файловой системы называют *виртуальными методами объекта vfs*:

- подключение файловой системы (т.е. размещение суперблока в память и инициализация записи в таблице монтирования vfs_mount);
- отключение файловой системы (vfs_unmount);
- возвращение корневого виртуального индексного дескриптора vnode файловой системы;
- актуализирует все кэш-данные файловой системы и т.д.

Для инициализации и монтирования реальной файловой системы UNIX хранит коммутатор файловых систем, адресующий процедурный интерфейс для каждого типа файловой системы, поддерживаемой ядром.

В UNIX SV используется глобальная таблица, каждый элемент которой соответствует определенному типу реальной файловой системы, например s5fs, ufs, nfs. Элемент этой таблицы vfssw имеет поля:

имя типа файловой системы	адрес процедуры инициализации	указатель на вектор операций файловой системы	флаги

Жесткие диски обычно монтируются при запуске системы (например, основной раздел, куда вы установили ASPLinux, монтируется корневой каталог /), однако их можно монтировать и отдельно.

Другие устройства, например, дисководы для гибких дисков A: (устройство /dev/fdO), B: (/dev/fdl), дисковод CD-ROM (/dev/cdrom), флэш-память (например, /dev/sde1 или /dev/sda1) монтируются по мере необходимости.

По окончании работы с устройствами они размонтируются, и размещенные них файлы становятся недоступными. Для жестких дисков это действие обычно выполняется автоматически при отключении системы. Иначе обстоит дело с CD-ROM, соответствующие устройства необходимо размонтировать перед тем, как удалять из компьютера носители. Нельзя удалить компакт-диск из неразмонтированного привода (устройство не реагирует на кнопку извлечения); вынуть сменный носитель (например, флеш-память) из неразмонтированного дисковода физически возможно, однако это приводит к полному уничтожению информации на ней.

Монтирование некоторых устройств можно осуществлять средствами GNOME, KDE и т.п. (но более гибкое решение предоставляет текстовая консоль.) Монтирование файловой системы осуществляется системным вызовом *mount*:

mount тип монтируемой системы точка монтирования флаги и дополнительные данные

где *точка монтирования* – это имя каталога, к которому подключается файловая система;

тип_монтируемой файловой_системы — ключевое слово, обозначающее стандарт, в котором записываются на устройстве файлы каталоги. Чаще всего могут встретиться следующие типы файловых систем:

ext2 — файловая система, используемая в разделах жесткого диска Linux;

iso9660 — файловая система, используемая на большинстве CD-ROM;

vfat — файловая система, используемая (с вариациями) в MS-DOS и Window 9x для жестких дисков и дискет (включает в себя системы, в терминологии Windows называющиеся FAT и FAT32).

Таким образом, чтобы смонтировать привод CD-ROM в каталог /MyCD, необходимо ввести команду

mount -t iso9660 /dev/cdrom /MyCD.

Для того чтобы смонтировать дисковод B: для чтения и записи дискет в формате MS-DOS на каталог /diskB, введите команду -

mount -t vfat /dev/fdl /diskB.

При этом происходит:

- поиск vnode, соответствующего файлу (точке монтирования) – операция lookup;
- проверка:
- является ли файл каталогом;
- не используется ли он в настоящее время для монтирования других файловых систем;
- поиск элемента коммутатора файловых систем, соответствующего типу монтируемой файловой системы. Если элемент найден, то выполняется следующая последовательность действий:
- вызов операции инициализации (при этом выполняется размещение специфических для данного типа файловой системы данных);
- ядро размещает структуру vfs;
- помещает ее в связанный список подключенных файловых систем, причем поле vnode, перекрываемое файловой системой указывает на vnode точки монтирования. Это поле устанавливается равным нулю для корневого каталога (root), элемент vfs которого всегда расположен первым в списке подключаемых файловых систем;
- вызывается операция подключения файловой системы vfs_mount (размещение суперблока в память и инициализация). При этом действия могут различаться и зависят от типа файловой системы, однако есть ряд общих операций:
- проверка соответствующих прав на выполнение монтирования;
- размещение и инициализация специфических для файловой системы данного типа данных;
- сохранение адреса этих данных в поле vfs_data элемента vfs;
- размещение vnode для корневого каталога подключаемой файловой системы, доступ к которому осуществляется с помощью операции определения корневого vnode - vfs_root.

После подключения файловая система может быть адресована по имени точки монтирования. На рис. 4.7 показан пример монтирования системы **A** к каталогу **/usr/local** системы **B**. Исследовать описанные структуры можно с помощью утилиты *crash*, а также команд *vfs* и *vnode*.

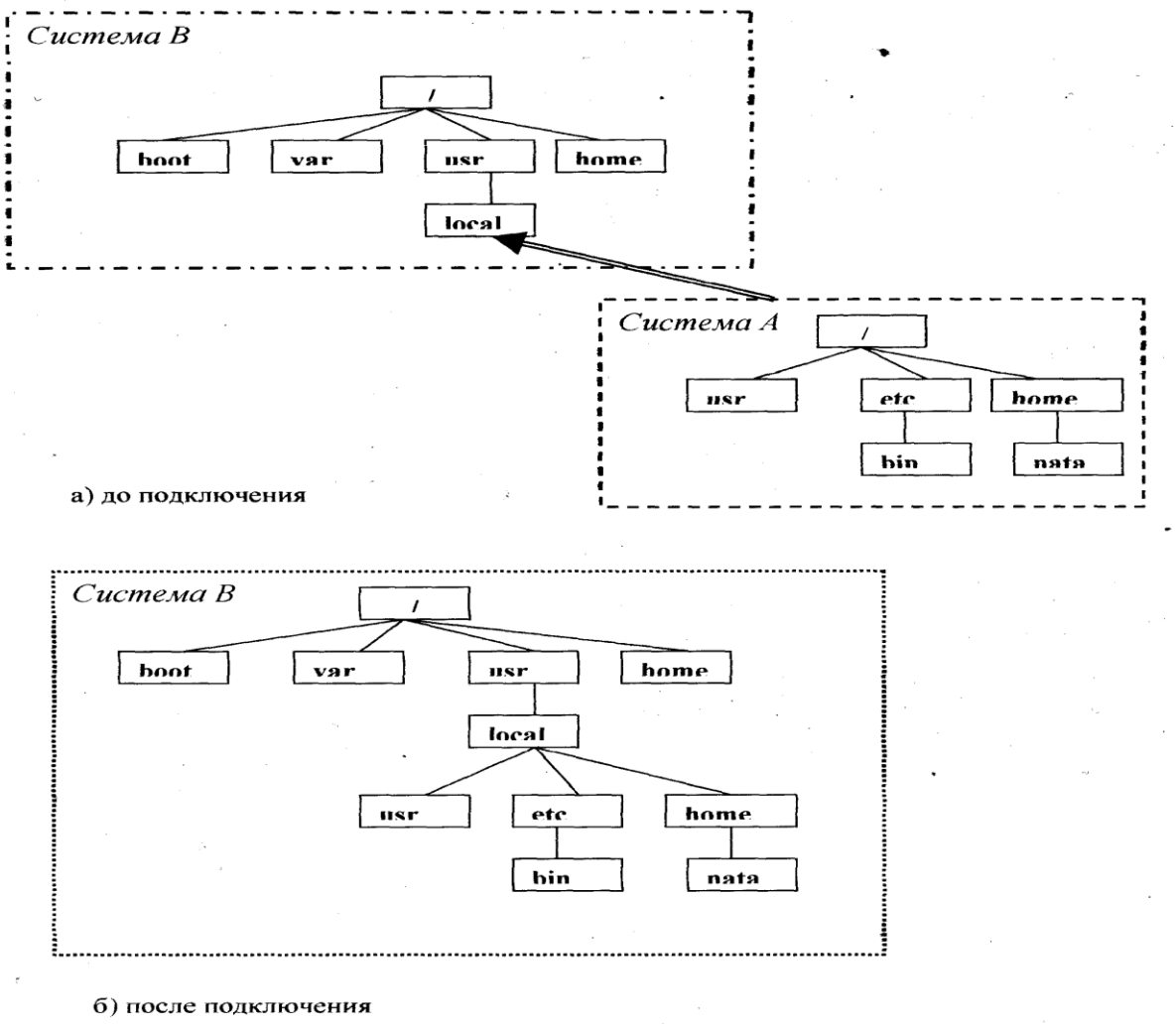


Рис. 4.7. Пример монтирования системы **A** к каталогу /usr/local

4.5. Трансляция имен

Ядро системы для обеспечения работы с файлами использует не имена файлов, а индексные дескрипторы. Таким образом, необходима трансляция имени файлов, передаваемого, например, в качестве аргумента системному вызову *open (),* в номер соответствующего vnode. Системные вызовы, для выполнения которых требуется трансляция имени файла:

- *exec ()* – запустить программу на выполнение;
- *chown () /chgrp ()* – изменить владельца-пользователя /владельца-группу соответственно;
- *chmod ()* – изменить права доступа;
- *statfs ()* – получить метаданные файла;
- *mkdir ()/ rkdir ()* – создать/удалить каталог;
- *mknod ()* – создать специальный файл устройства;
- *open ()* – открыть файл;
- *link ()* – создать жесткую связь.

Два каталога играют ключевую роль при трансляции имени: корневой каталог и текущий каталог. Каждый процесс адресует эти каталоги двумя

полями структуры u_area: *u_cdir (указатель на vnode текущего каталога), *u_rdir (указатель на vnode корневого каталога). В зависимости от имени файла трансляция начинается с vnode, адресованного полем u_cdir, либо u_rdir. Трансляция имени осуществляется покомпонентно, при этом vnode текущего каталога вызывается соответствующая ему операция vn_lookup(), в качестве аргумента которой передается имя следующего компонента. В результате операции возвращается vnode, соответствующий искомому компоненту.

Если для vnode каталога установлен указатель vn_vfsmountedhere, то данный каталог является точкой монтирования. Если имя файла требует дальнейшего спуска по дереву файловой системы (т.е. пересечение точки монтирования), то операция vn_lookup() следует указателю vn_vfsmountedhere для перехода в подключенную файловую систему и вызывает для нее операцию vfs_root() для получения корневого vnode. Трансляция имени затем продолжается с этого места.

Если искомый файл начинается с “/”, т.е. является абсолютным, то трансляция начинается с vnode корневого каталога, адресованного полем u_rdir области u_area.

Процесс трансляции имени продолжается пока не просмотрены все компоненты имени или не обнаружена ошибка (например, отсутствие прав доступа).

4.6. Блокировка доступа к файлу

Традиционно архитектура файловой подсистемы UNIX разрешает нескольким процессам одновременный доступ к файлу для чтения или записи.

В UNIX по умолчанию отсутствует синхронизация между отдельными вызовами, т.е. между двумя последовательными вызовами read() одного процесса другой процесс может модифицировать данные файла, что приведет к нарушению целостности данных.

UNIX позволяет обеспечить блокирование заданного диапазона байтов файла или записи файла. Для этого служат базовый системный вызов управления файлом fcntl() и библиотечная функция lockf(). При этом перед фактической файловой операцией (чтение или запись) процесс устанавливает блокировку соответствующего типа. Если блокировка завершилось успешно, то требуемая файловая операция не создает конфликта или нарушения целостности данных.

По умолчанию блокировка является рекомендательной, т.е. кооперативно работающие процессы могут руководствоваться собственными блокировками, однако ядро не запрещает чтение или запись в заблокированный участок файла. Правила блокировки следующие:

1) может быть установлено несколько блокировок для чтения на конкретный байт файла, при этом в установке блокировки для записи на этот байт будет отказано;

2) при блокировке для записи на конкретный байт должна быть единственной, при этом в установке блокировки для чтения будет отказано.

В отличие от рекомендательного в UNIX существует обязательное блокирование, при котором ограничение на доступ к записям файла накладывается самим ядром. Реализация обязательных блокировок может быть различной. Следует отметить, что использование обязательных блокировок может привести к аварийному останову операционной системы (например, процесс блокирует доступ к важному системному файлу и по каким-либо причинам теряет контроль).

4.7 Целостность файловой системы

Значительная часть файловой системы находится в оперативной памяти, а именно, суперблок примонтированной системы, метаданные активных файлов и отдельные блоки хранения данных файлов, временно находящихся в буферном кэше.

Для операционной системы рассогласование между буферным кэшем и блоками хранения данных отдельных файлов не приведет к катастрофическим последствиям. В то же время операции с метаданными могут оказать влияние на целостность файловой системы.

Ядро выбирает порядок совершения операций таким образом, чтобы вред от них был минимальным. Единственной возможностью сохранить выбранный порядок является синхронизация операций со стороны файловой подсистемы. Отсутствие синхронизации между образом файловой системы в памяти и ее данными на диске в случае аварийного останова может привести к появлению следующих ошибок, которые схематично показаны на рис. 4.8:

1) один блок адресуется несколькими inode (принадлежит нескольким файлам);

2) блок помечен как свободный, но в тоже время занят (на него ссылается inode);

3) блок помечен как занятый, но в тоже время свободен (ни один inode на него не ссылается);

4) неправильное число ссылок в inode (недостаток или избыток ссылающихся записей в каталогах);

5) несовпадение между размером файла и суммарным размером адресуемых inode блоков;

6) недопустимые адресуемые блоки (например, расположенные за пределами файловой системы);

7) «потерянные» файлы (правильные inode, на которые не ссылаются записи каталогов);

8) недопустимые или неразмещенные номера inode в записях каталогов.

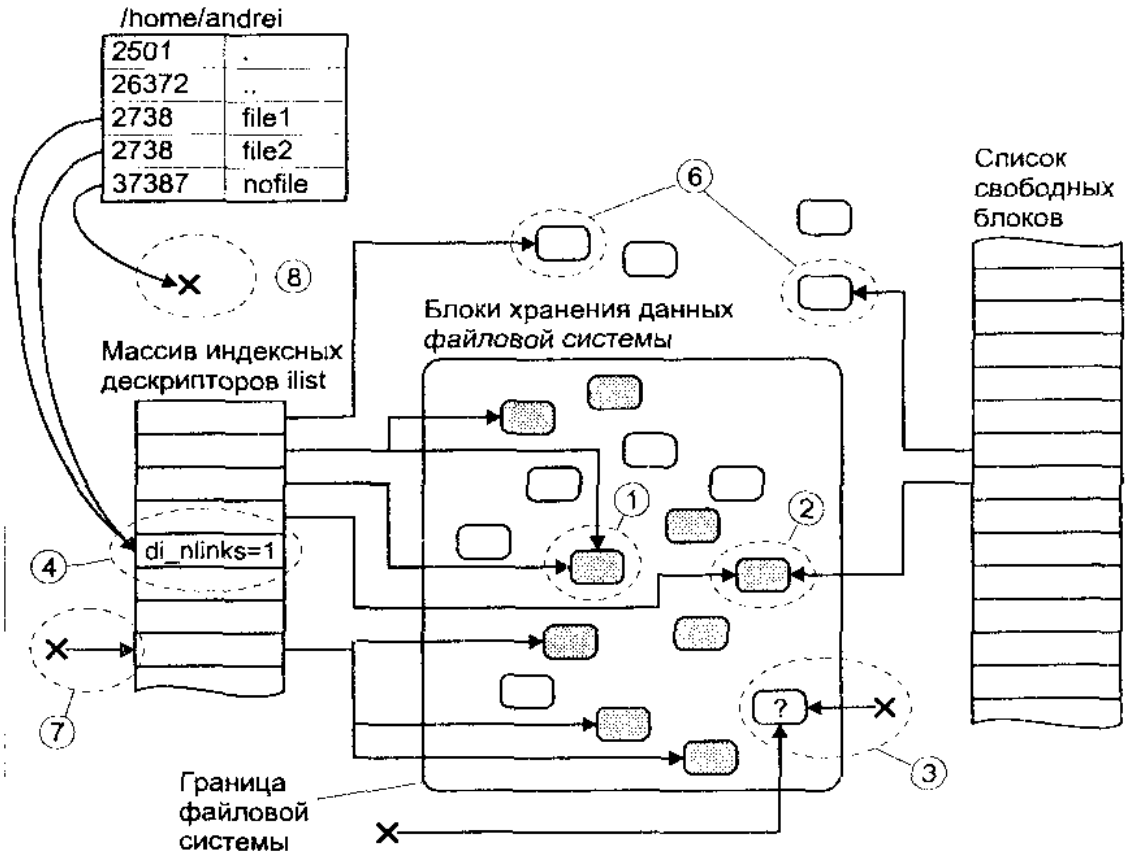


Рис. 4.8. Возможные ошибки файловой системы

Если нарушение все же произошло, необходимо воспользоваться утилитой *fsdck()*. Запуск утилиты производится автоматически каждый раз при запуске системы, или администратором с помощью команды:

fsdck [options] filesystem ,

где filesystem – специальный файл устройства, на котором находится файловая система.

Проверка и исправление должно проводится только на размонтированной системе. Исключение составляет корневая файловая система, тогда следует использовать опцию *b*, обеспечивающую немедленный перезапуск системы после проверки.

Контрольные вопросы

1. Назовите основные отличия s5fs и FFS.
2. С помощью чего осуществляется доступ к структуре каталогов и файлов после монтирования файловой системы?
3. Как поддерживается работа с файлами, размер которых варьируется?
4. Какие ограничения накладываются в s5fs?
5. Перечислите способы адресации блока устройства, содержащего файловую систему.
6. Приведите структуру записи каталога файловой системы FFS.
7. Что такое виртуальная файловая система?

8.Зависит ли структура vnode от типа реальной файловой системы?
9.Как организована структура подключенных файловых систем?
10.Что происходит при монтировании файловой системы?
11.Для чего используется трансляция имен?
12.Перечислите ошибки файловой системы.

РАЗДЕЛ 5. ПОДСИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССОМ

5.1. Концепция процессов в ОС UNIX

Концепция процессов – базовая в ОС UNIX. Операционная система управляет ансамблем процессов, приостанавливая выполнение одних, активизируя другие. Процесс в UNIX понимается в классическом смысле этого термина – выполняющийся экземпляр программы; совокупность данных ядра системы, необходимых для описания образа программы в памяти и управления ее выполнением. Например, на экране отображается два терминальных окна, т.е. одна и та же терминальная программа запущена дважды – ей соответствует два процесса. В каждом окне, очевидно, работает интерпретатор команд – это еще один процесс. Когда пользователь вводит команду в интерпретаторе, соответствующая программа запускается в виде процесса. По завершению работы программы управление вновь передается процессу интерпретатора.

Операционная система создает процесс, когда пользователь запускает программу на исполнение. Однако нельзя отождествлять процесс и программу, которая исполняется в данном процессе. Программой является машинный код и начальные данные, содержащиеся в исполняемом файле на диске или скопированные ОС для исполнения в оперативную память. Для каждого процесса ОС создает дополнительный набор данных, который называется *средой выполнения процесса*. Одна часть данных из среды процесса доступна только операционной системе, другая часть – исполняемому процессу, а третья часть – другим процессам. Данные из этого набора называются *атрибутами процесса.* Среди них: переменные окружения, текущий каталог, стандартные файлы ввода, вывода, ошибок и другие. Таким образом, **процесс в ОС UNIX** – это *совокупность программы и связанных с ней системных данных, среды процесса.*

Процесс состоит из инструкций, выполняемых процессором, данных и информации о выполняемой задаче, такой как размещенная память, открытые файлы и статус процесса. Не следует отождествлять процесс и программу, так как программа может породить несколько процессов. ОС UNIX является многозадачной. Это значит, что одновременно могут выполняться несколько процессов, причем часть процессов могут являться образцами одной программы.

Выполнение процесса заключается в точном следовании набору инструкций, который никогда не передает управление набору инструкций

другого процесса. Процесс считывает и записывает информацию в раздел данных и стек, но ему не доступны стек и данные другого процесса.

В то же время в UNIX существуют средства взаимодействия между процессами:

- сигналы (signals);
- каналы (pipes);
- разделяемая память (shared memory);
- семафоры (semaphores);
- сообщения (messages);
- файлы (files).
-

5.2. Типы процессов

Существует три типа процессов: системные процессы; демоны; прикладные процессы.

1) Системные процессы

Системные процессы являются частью ядра и всегда расположены в оперативной памяти. Системные процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы. Таким образом они могут вызывать функции и обращаться к данным, недоступным для остальных процессов. Системными процессами являются: диспетчер свопинга (shed); диспетчер страничного замещения (vhand); диспетчер буферного кэша (bdfflush); диспетчер памяти ядра (kmadaemon). К системным процессам следует отнести init, являющийся прародителем других процессов в UNIX. Хотя init не является частью ядра, и его запуск происходит из исполняемого файла (/etc/init), его работа жизненно важна для работы других процессов.

2) Демоны

Демоны – это неинтерактивные процессы, которые запускаются обычным образом – путем загрузки в память соответствующих им программ (исполняемых файлов), и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы (но после инициализации ядра) и обеспечивают работу различных подсистем UNIX: системы терминального доступа, системы печати, системы сетевого доступа и сетевых услуг и т.п. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть времени демоны ожидают пока тот или иной процесс запросит определенную услугу, например, доступ к файловому архиву или печать документа.

3) Прикладные процессы

К прикладным процессам относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. Важнейшим пользовательским процессом является основной командный интерпретатор login shell, который обеспечивает

работу пользователя в UNIX. Он запускается сразу после регистрации пользователя в системе, а завершение работы login shell приводит к отключению от системы.

Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но в любом случае время их жизни (и выполнения) ограничено сеансом работы пользователя. При выходе из системы все пользовательские процессы будут уничтожены.

Интерактивные процессы монопольно владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не сможет работать с другими приложениями.

5.3. Атрибуты процесса

Процесс в UNIX имеет несколько атрибутов, позволяющих ОС эффективно управлять его работой, важнейшие из которых приведены ниже.

Идентификатор процесса - PID

Каждый процесс имеет свой уникальный идентификатор (Process ID), позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему следующий свободный идентификатор. Присвоение идентификатора происходит по возрастающий, то есть идентификатор нового процесса больше, чем идентификатор процесса, созданного перед ним. Если идентификатор достиг максимального значения, следующий процесс получит минимальный свободный PID из списка освобожденных идентификаторов и цикл повторяется. Когда процесс завершает свою работу, ядро освобождает занятый им идентификатор.

Идентификатор родительского процесса - PPID

Идентификатор родительского процесса (Parent Process ID) - это идентификатор процесса, породившего данный процесс. Родительский процесс может закончиться раньше дочернего. Для того, чтобы получить информацию об идентификаторе процесса необходимо ввести команду: **$***ps -f* , для вывода родительского идентификатора - *ps -o.*

Приоритет процесса - NN

Nice Number - относительный приоритет процесса, учитываемый планировщиком при определении очередности запуска. Фактическое же распределение процессорных ресурсов определяется *приоритетом выполнения,* зависящим от нескольких факторов, в частности от заданного относительного приоритета. Относительный приоритет не изменяется системой на всем протяжении жизни процесса (хотя может быть изменен пользователем или администратором) в отличие от приоритета выполнения, динамически обновляемого ядром.

Терминальная линия - TTY

TTY - терминал или псевдотерминал, ассоциированный с процессом, если такой существует. Процессы-демоны не имеют ассоциированного терминала.

Реальный идентификатор пользователя - RID

Real ID – реальный идентификатор пользователя. Реальным идентификатором пользователя данного процесса является идентификатор пользователя, запустившего данный процесс.

Эффективный идентификатор пользователя - EUID

EUID – эффективный идентификатор пользователя. Эффективный идентификатор служит для определения прав доступа процесса к системным ресурсам.

Реальный идентификатор группы - RGID

RGID – реальный идентификатор группы. Реальный идентификатор группы равен идентификатору первичной или текущей группы пользователя, запустившего процесс.

Эффективный идентификатор группы - EGID

EGID – эффективный идентификатор группы. Эффективный идентификатор служит для определения прав доступа к системным ресурсам по классу доступа группы.

Команда *ps (process status)* позволяет вывести список процессов, выполняющихся в системе, и их атрибуты, например:

```
$    ps –ef | head  -20
UID    PID    PPID    C    STIME    TTY    TIME    CMD
root    1       0       0    Nov 23    ?        0:01      /etc/init
nata    345    304     0     Nov 23    pts/3    0:01      /usr/bin/bash
```

5.4. Создание и выполнение процесса

Существует два способа создания процессов:

- с помощью функции *system ();*
- с помощью использования двух функций *fork(), exec().*

Функция *system ()* определена в стандартной библиотеке С, позволяет вызывать из программы системную команду (т.е. по сути запускается стандартный интерпретатор Bourne shell (/bin/sh)). Функция возвращает код завершения указанной команды. Если интерпретатор не может быть запущен, возвращается код 127, в случае возникновения других ошибок возвращается -1. Однако создание процесса с использованием функции *system ()* не эффективный способ, так как связан со значительным риском для безопасности системы (неодинаковые последствия в разных системах).

В ОС Linux нет функции типа spawn (), которая бы за один проход создавала процесса и запускала его. Вместо этого функция *fork()* создает дочерний процесс, который является точной копией родительского процесса, и семейство функций *exec()*, заставляющих требуемый процесс перестать быть экземпляром одной программы и превратиться в экземпляр другой программы. Таким образом, сначала с помощью функции *fork()* создается копия текущего процесса, а затем с помощью функции *exec()* преобразовывается одна из копий процесса в экземпляр запускаемой программы.

После вызова функции *fork()* программа создает дубликат, называемый дочерним процессом. Родительский процесс продолжает выполнять программу с той точки, где была вызвана функция *fork().* То же самое делает и дочерний процесс. Как различать оба процесса между собой? Во-первых, идентификатор дочернего процесса отличается от родительского. Программа может вызвать функцию *getpid()* и узнать, где именно она находится. Во-вторых, сама функция *fork()* возвращает разные значения в родительский и дочерний процессы. Родительский процесс получает идентификатор своего потомка, а дочернему возвращается 0. В системе нет процессов с нулевым идентификатором, так что программа легко разбирается в ситуации.

На рис. 5.1 представлена схема создания процесса и запуска программы.

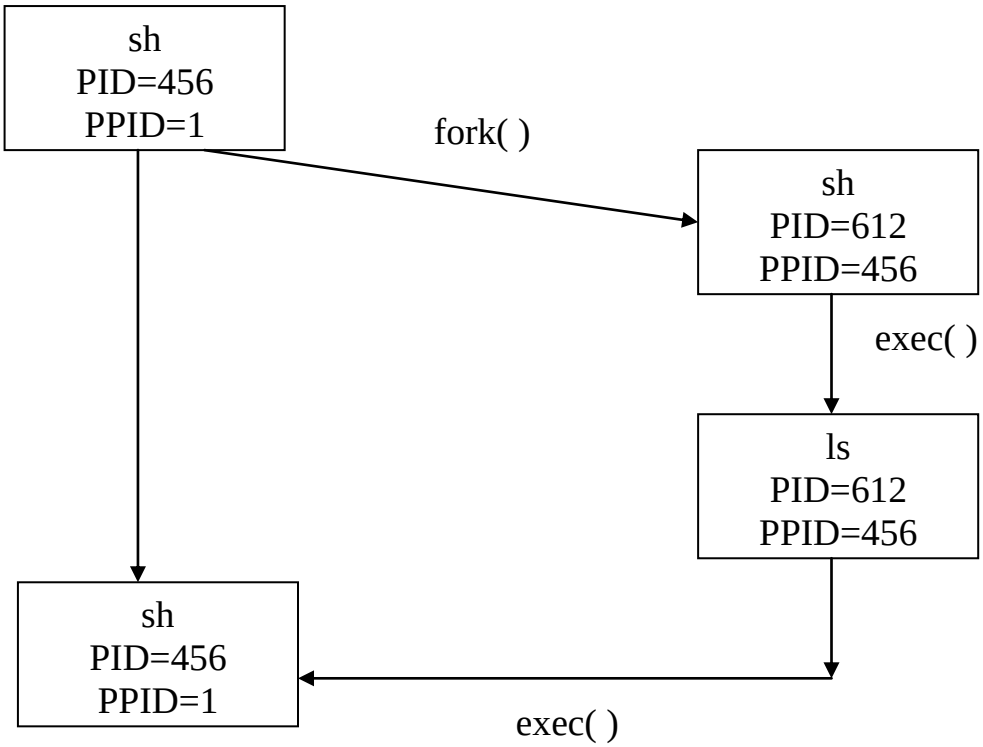


Рис. 5.1 Схема создания процесса и запуска программы

5.5. Основы управления процессом

Как говорилось ранее, процесс UNIX представляет собой исполняемый образ программы, включающий отображение в памяти исполняемого файла, полученного в результате компиляции, стек, код и данные библиотек, а также ряд структур данных ядра, необходимых для управления процессом. На рис. 5.2 схематично представлены компоненты, необходимые для создания и выполнения процесса.

Процесс во время выполнения использует различные системные ресурсы: память, процессор, услуги файловой подсистемы и подсистемы ввода/вывода. Операционная система UNIX обеспечивает иллюзию одновременного выполнения нескольких процессов, эффективно распределяя системные ресурсы

между активными процессами и не позволяя в то же время ни одному из них монополизировать использование этих ресурсов.

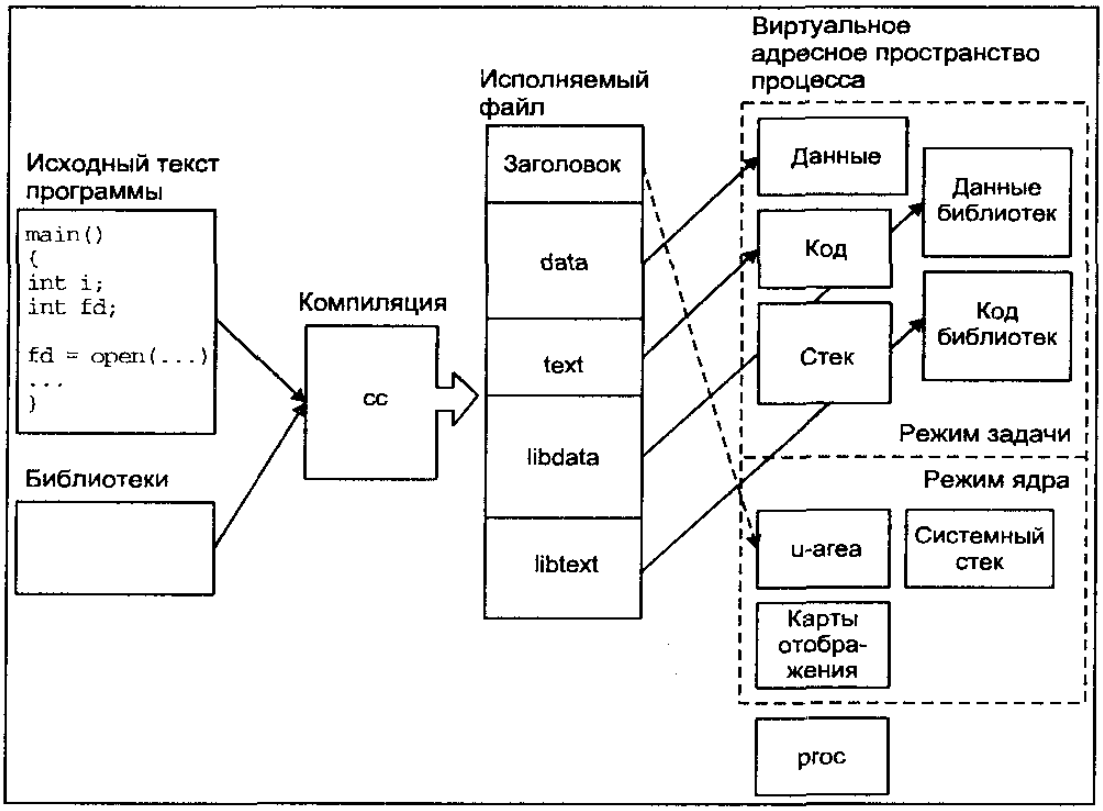


Рис. 5.2. Компоненты процесса

В начале создания операционная система UNIX обеспечивала выполнение всего двух процессов, по одному на каждый подключенный к PDP-7 терминал. Спустя год, на той же PDP-7 число процессов заметно увеличилось, появился системный вызов *fork().* В Первой редакции UNIX появился вызов *exec(),* но операционная система по-прежнему позволяла размещать в памяти только один процесс в каждый момент времени. После реализации аппаратной подсистемы управления памятью на PDP-11 операционная система была модифицирована, что позволило загружать в память сразу несколько процессов, уменьшая тем самым время на сохранение образа процесса во вторичной памяти (на диске) и считывание его, когда процесс продолжал выполнение.

Однако до 1972 года UNIX нельзя было назвать действительно многозадачной системой, т.к. операции ввода/вывода оставались синхронными, и другие процессы не могли выполняться, пока один из процессов не завершал операцию ввода/вывода (обычно достаточно продолжительную). Истинная многозадачность появилась только после того, как код UNIX был переписан на языке С в 1973 году. С тех пор основы управления процессами практически не изменились.

Выполнение процесса может происходить в двух режимах — в *режиме ядра* (kernel mode) или в *режиме задачи* (user mode). В режиме задачи процесс выполняет инструкции прикладной программы, допустимые на непривиле-

легированном уровне защиты процессора. При этом процессу недоступны системные структуры данных. Когда процессу требуется получение каких-либо услуг ядра, он делает системный вызов, который выполняет инструкции ядра, находящиеся на привилегированном уровне. Несмотря на то, что выполняются инструкции ядра, это происходит от имени процесса, сделавшего системный вызов. Выполнение процесса при этом переходит в режим ядра. Таким образом, ядро системы защищает собственное адресное пространство от доступа прикладного процесса, который может нарушить целостность структур данных ядра и привести к разрушению операционной системы. Более того, часть процессорных инструкций, например, изменение регистров, связанных с управлением памятью, могут быть выполнены только в режиме ядра.

Соответственно и образ процесса состоит из двух частей: данных режима ядра и режима задачи. Образ процесса в режиме задачи состоит из сегмента кода, данных, стека, библиотек и других структур данных, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в режиме задачи, которые используются ядром для управления процессом. Сюда относятся данные, диктуемые аппаратным уровнем, например состояния регистров, таблицы для отображения памяти и т. д., а также структуры данных, необходимые ядру для обслуживания процесса. Вообще говоря, в режиме ядра процесс имеет доступ к любой области памяти.

5.6. Структура данных процесса

Каждый процесс представлен в системе двумя основными структурами данных — *proc* и *user*, описанными соответственно в файлах <sys/proc.h> и <sys/user.h>. Содержимое и формат этих структур различны для разных версий UNIX.

В любой момент времени данные структур *proc* для всех процессов должны присутствовать в памяти, хотя остальные структуры данных, включая образ процесса, могут быть перемещены во вторичную память - область свопинга. Это позволяет ядру иметь под рукой минимальную информацию, необходимую для определения местонахождения остальных данных, относящихся к процессу, даже если они отсутствуют в памяти. Структура *proc* является записью системной таблицы процессов, которая всегда находится в оперативной памяти. Запись этой таблицы для выполняющегося в настоящий момент времени процесса адресуется системной переменной *curproc*. Каждый раз при переключении контекста, когда ресурсы процессора передаются другому процессу соответственно изменяется значение переменной *curproc*, которая теперь указывает на структуру *proc* активного процесса. Вторая упомянутая структура - *user*, также называемая *u-area* или и *block*, содержит дополнительные данные о процессе, которые требуются ядру только во время выполнения процесса (т. е., когда процессор выполняет инструкции процесса в режиме ядра или задачи). В отличие от структуры *proc*, адресованной

указателем *curproc*, данные *user* отображаются в определенном месте виртуальной памяти ядра и адресуются переменной *u*. На рис. 5.3 показаны две основные структуры данных процесса и способы их адресации ядром UNIX.

В *u-area* хранятся данные, которые используются многими подсистемами ядра и не только для управления процессом. В частности, там содержится информация об открытых файловых дескрипторах, диспозиция сигналов, статистика выполнения процесса, а также сохраненные значения регистров, когда выполнение процесса приостановлено. Очевидно, что процесс не должен иметь возможности модифицировать эти данные произвольным образом, поэтому *u-area* защищена от доступа в режиме задачи.

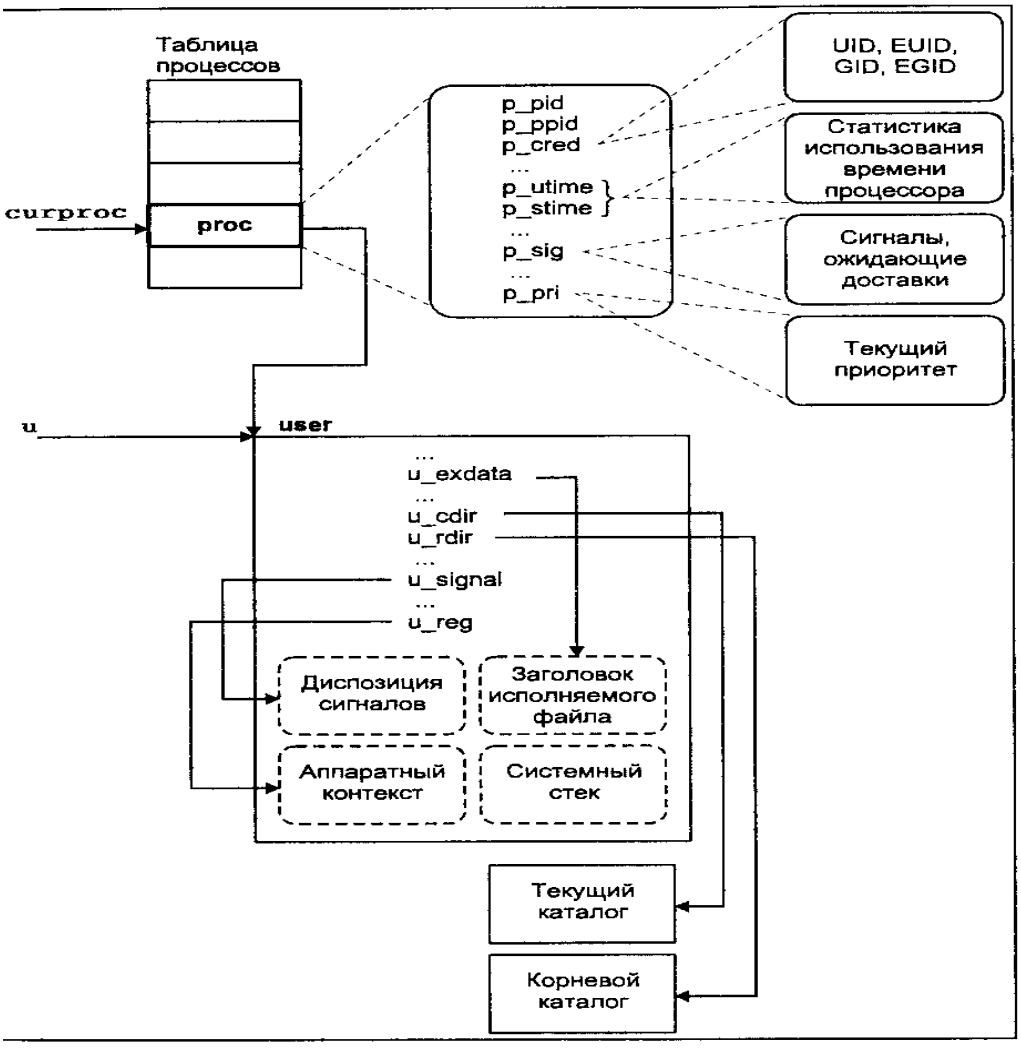


Рис. 5.3. Основные структуры данных процесса

Как видно из рис.5.3, u-area также содержит стек фиксированного размера, — *системный стек* или *стек ядра* (kernel stack). При выполнении процесса в режиме ядра операционная система использует этот стек, а не обычный стек процесса.

5.7. Алгоритмы планирования процессов

Алгоритмы планирования процессов в UNIX обеспечивают возможность одновременного выполнения интерактивных и фоновых приложений, и обеспечивают малое время реакции для интерактивных приложений, и предоставляют ресурсы громоздким фоновым приложениям.

Планирование процессов в UNIX основано на приоритете процессов. Планировщик всегда выбирает процесс с наивысшим приоритетом. Приоритет процесса не является фиксированным и динамически изменяется системой в зависимости от использования вычислительных ресурсов, времени ожидания запуска и текущего состояния процесса.

Традиционно ядро UNIX является «непрерываемым». Это означает, что процесс, находящийся в режиме ядра и выполняющий системные инструкции, не может быть прерван системой, а вычислительные ресурсы переданы другому, более высокоприоритетному процессу.

Каждый процесс имеет два атрибута приоритета:

- *текущий приоритет,* на основании которого происходит планирование;
- *относительный приоритет,* который задается при порождении процесса и влияет на текущий приоритет.

Текущий приоритет варьируется в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет). Процессы, выполняющиеся в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи приоритет меняется в диапазоне 0–65, для режима ядра – 66-95 (системный диапазон).

Процессы, приоритеты которых лежат в диапазоне 96–127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой, и предназначены для поддержки приложений, работающих в реальном масштабе времени.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение приоритета сна. Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна, так как приоритет сна имеет более высокий приоритет, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет быстро завершить системный вызов, выполнение которого может блокировать некоторые системные ресурсы.

После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова.

Текущий приоритет процесса в режиме задачи зависит от:

- значения относительного приоритета;
- степени использования вычислительных ресурсов.

Большему значению соответствует больший приоритет. Более высокий относительный приоритет соответствует более низкому. Пока процесс выполняется в режиме задачи его текущий приоритет линейно уменьшается.

В каждую секунду ядро пересчитывает текущие приоритеты процессов, готовых к запуску, последовательно увеличивая их. Это перемещает процессы в более приоритетные очереди и повышает вероятность их последующего запуска. Очередь на выполнение процессов, как правило, не одна.

5.8. Взаимодействие между процессами

Концепция UNIX основана на взаимодействии между отдельными процессами. Для реализации взаимодействия требуется:

- обеспечить средства взаимодействия между процессами;
- исключить нежелательное влияние одного процесса на другой.

Взаимодействия между процессами необходимо для решения следующих задач:

- *Передача данных.* Один процесс передает данные другому процессу, при этом их объем может варьироваться от десятков байтов до нескольких мегабайтов.
- *Совместное использование данных.* Вместо копирования информации от одного процесса к другому, процессы могут совместно использовать одну копию данных, причем изменения, сделанные одним процессом, будут сразу же заметны для другого.
- *Извещения о наступлении некоторого события.* Процесс может известить другой процесс или группу процессов о наступлении некоторого события.

Решить данную задачу можно средствами самих процессов, но это будет неэффективно. Поэтому операционная система должна обеспечить механизмы межпроцессного взаимодействия (Inter Process Communication, IPC).

К средствам межпроцессного взаимодействия можно отнести: сигналы, каналы, FIFO (именованные каналы), сообщения (очереди сообщений), семафоры, разделяемую память. Сигналы не могут использоваться для синхронизации процессов.

5.9. Сигналы

Сигнал – это простейшая форма процессорного взаимодействия. Сигналы позволяют уведомлять процесс или группу процессов о наступлении некоторого события.

Группа процессов. Каждый процесс принадлежит определенной группе процессов. Каждая группа имеет уникальный идентификатор. Группа может иметь в своем составе *лидера группы* – процесс, чей идентификатор PID равен идентификатору группы.

Управляющий терминал. Процесс может быть связан с терминалом, который называется управляющим. Все процессы группы имеют один и тот же управляющий терминал.

Специальный файл устройства /dev/tty. Этот файл связан с управляющим терминалом процесса. Драйвер для этого псевдоустройства по существу

перенаправляет запросы на фактический терминальный драйвер, который может быть различным для различных процессов.

Сигналы обеспечивают механизм вызова определенной процедуры при наступлении некоторого события. Каждое событие имеет свой идентификатор и символьную константу.

Необходимо различать две фазы сигналов:

- генерация или отправление сигнала;
- доставка и обработка сигнала.

Сигнал отправляется, когда происходит определенное событие, о наступлении которого должен быть уведомлен процесс. Сигнал считается доставленным, когда процесс, которому был отправлен сигнал, получает его и выполняет его обработку. В промежутке между этими двумя моментами сигнал ожидает доставки.

Ядро генерирует и отправляет процессу сигнал в ответ на ряд событий, которые могут быть вызваны:

1) самим процессом;
2) другим процессом;
3) прерыванием или какими - либо внешними событиями.

К основным причинам отправки сигнала можно отнести:

- *терминальные прерывания* - нажатие некоторых клавиш терминала вызывает отправление сигнала текущему процессу, связанному с терминалом;

- *особые ситуации* - когда выполнение процесса вызывает особую ситуацию, например, деление на ноль, процесс получает соответствующий сигнал;

- *другие процессы* - процесс может отправить сигнал другому процессу или группе процессов с помощью системного вызова *kill().* В этом случае сигналы являются элементарной формой межпроцессного взаимодействия;

-*управление заданиями* - командные интерпретаторы, поддерживающие систему управления заданиями, используют сигналы для манипулирования фоновыми и текущими задачами. Например, когда процесс, выполняющийся в фоновом режиме, делает попытку чтения или записи на терминал, ему отправляется сигнал останова. Или, когда дочерний процесс завершает свою работу, родитель уведомляется об этом также с помощью сигнала;

- *квоты* - когда процесс превышает выделенную ему квоту вычислительных ресурсов или ресурсов файловой системы, ему отправляется соответствующий сигнал;

- *уведомления* - процесс может запросить уведомление о наступлении тех или иных событий, например, готовности устройства и т.д. Такое уведомление отправляется процессу в виде сигнала.

- *алармы* - если процесс установил таймер, ему будет отправлен сигнал, когда значение таймера станет равным нулю.

5.10. Доставка и обработка сигнала

Для каждого сигнала в системе определена обработка по умолчанию, которую выполняет ядро, если процесс не указал другого действия. В общем случае существует четыре возможных действия:

- завершить выполнение процесса;
- игнорировать сигнал;
- остановить процесс;
- продолжить процесс.

Процесс может заблокировать сигнал, отложив на некоторое время его обработку. Это возможно не для всех сигналов. Например, для сигналов SIGKILL и SIGSTOP единственным действием является действие по умолчанию, эти сигналы нельзя ни перехватить, ни заблокировать, ни игнорировать. Для ряда сигналов, преимущественно связанных с аппаратными ошибками и особыми ситуациями, обработку выполнять надо по умолчанию.

Доставка сигнала происходит после того, как ядро от имени процесса вызывает системную процедуру *issig().* Эта процедура проверяет, существуют ли ожидающие доставки сигналы, адресованные данному процессу. Функция *issig ()* вызывается ядром в трех случаях:

1) Непосредственно перед возвращением из режима ядра в режим задачи после обработки системного вызова или прерывания.
2) Непосредственно перед переходом процесса в состояние сна с приоритетом, допускающим прерывание сигналом.
3) Сразу же после пробуждения после сна с приоритетом, допускающим прерывание сигналом.

Если процедура *issig ()* обнаруживает ожидающие доставки сигналы, ядро вызывает функцию доставки сигнала, которая выполняет действия по умолчанию или вызывает специальную функцию *sendsig (),* запускающую обработчик сигнала, зарегистрированный процессом. Функция *sendsig ()* возвращает процесс в режим задачи, передает управление обработчику сигнала, а затем восстанавливает контекст процесса, для продолжения прерванного сигналом выполнения.

Рассмотрим типичные ситуации, связанные с отправлением и доставкой сигналов. Допустим, пользователь, работая за терминалом, нажимает клавишу прерывания <Del> или <Ctrl>+<C>. На рис. 5.4 представлена схема отправления и доставки сигнала.

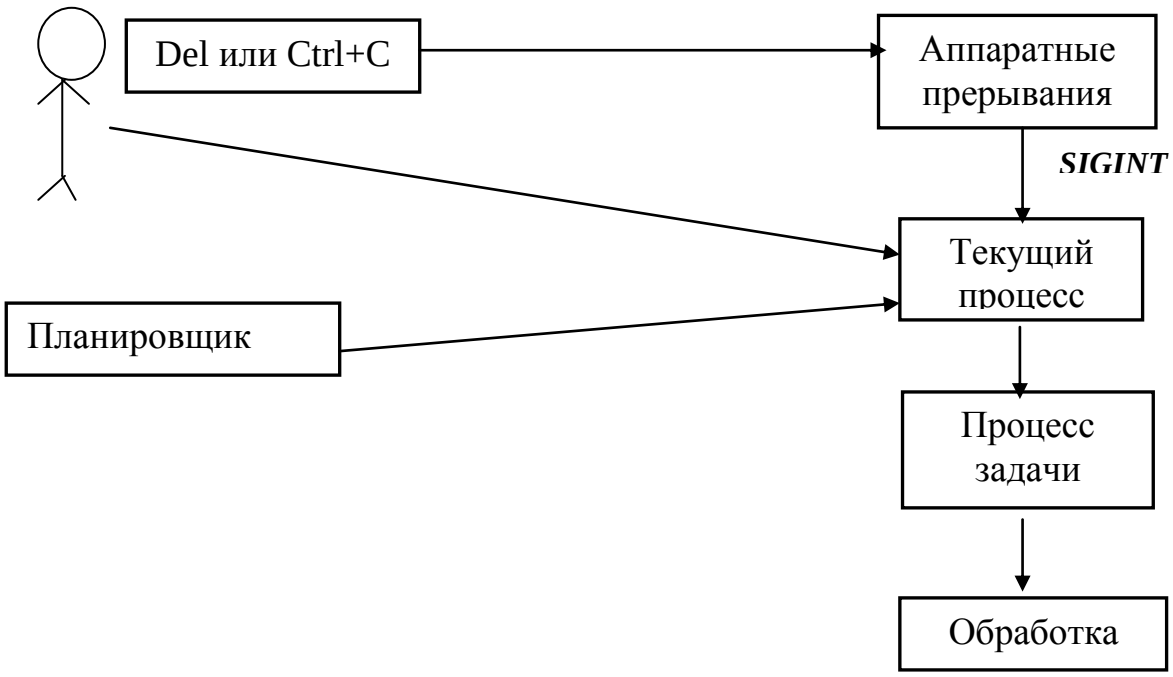


Рис. 5.4. Схема отправления и доставки сигнала

5.11. Обработка прерываний таймера

Обработка прерываний таймера подразделяется на отложенные вызовы (callout) и алармы (alarm) .

Следует вспомнить, что каждый компьютер имеет аппаратный таймер или системные часы, которые генерируют аппаратные прерывания через фиксированные интервалы времени, называемые *тиком* процессора (CPU tick, clock tick - временной интервал между соседними прерываниями или максимальное временное разрешение, которое может обеспечить система).

Для UNIX CPU tick обычно составляет 10 миллисекунд, значение хранится в константе HZ, которая определена в файле заголовков <param.h>.

Обработка прерываний таймера зависит от конкретной аппаратной архитектуры и версии ОС. В общем случае обработчик решает следующие задачи:

- обновление статистики использования процессора для текущего процесса;
- выполнение ряда функций, связанных с планированием процессов, например, пересчет приоритетов, проверка истечения временного кванта для процесса;
- проверка превышения процессорной квоты для данного процесса и отправка этому процессу сигнала SIGXCPU в случае превышения;
- обновление системного времени и других, связанных с ним таймеров;
- обработка отложенных вызовов (callout);
- обработка алармов (alarm);
- пробуждение в случае необходимости системных процессов (например, диспетчера страниц).

Часть перечисленных задач не требует выполнения на каждом тике, поэтому системы вводят нотацию *главного тика (major tick)*, который происходит каждые n тиков (в зависимости от конкретной версии). Определенный набор функций выполняется только на главных тиках.

5.11.1. Отложенные вызовы

Отложенный вызов определяет функцию, вызов которой производится ядром системы через некоторое время. Например, регистрация отложенного вызова подсистемой ядра:

*int co_ID=timeout (void (*fn) (), caddr_t arg, long delta),*

где fn – адрес функции, которую нужно вызвать, при этом ей будет передан аргумент arg, а сам вызов будет произведен через delta тиков.

Функции отложенных вызовов выполняются в системном контексте, а не в контексте прерываний, т.е. вызов этих функций выполняется отдельным обработчиком отложенных вызовов (а не обработчиком прерываний таймера), который допускается после завершения обработки прерываний таймера.

Отложенные вызовы применяются для выполнения таких функций, как:

- опрос устройств, не поддерживающих прерывания;
- выполнение некоторых функций планировщика и подсистемы управления памятью;
- выполнение функций драйверов устройств для событий, вероятность наступления которых относительно мала (например, модуль протокола TCP, реализующий передачу сетевых пакетов по тайм-ауту).

При обработке прерываний таймера система:

- проверяет необходимость запуска тех или иных функций отложенного вызова;
- устанавливает соответствующий флаг.

Далее управление передается обработчику отложенных вызовов, который:

- проверяет флаги;
- запускает необходимые функции в системном контексте.

Функции хранятся в системной таблице отложенных вызовов. Таблица может быть организована в виде списка, отсортированного по времени запуска. Каждый элемент хранит разницу между временем вызова функции и временем вызова предыдущего элемента таблицы $\Delta = t_{ввф} - t_{ввф} - 1$. На каждом тике величина Δ уменьшается на единицу для первого элемента таблицы. Когда Δ=0, производится вызов соответствующей функции и запись удаляется.

5.11.2. Алармы

Процесс может запросить ядро отправить сигнал по прошествии определенного интервала времени. Существует три типа алармов:

- реального времени (real time);
- профилирования (profiling);

- виртуального времени (virtual time).

С каждым из этих типов связан *таймер интервала (interval timer* или *itimer).* Значение *itimer* уменьшается на единицу при каждом тике, как только значение достигнет нуля, процессу отправляется соответствующий сигнал ITIMER_REAL, ITIMER_PROF, ITIMER_VIRT.

Рассмотренные таймеры обладают следующими характеристиками:

- ITIMER_REAL используется для отсчета реального времени, как только itimer_real=0 процессу отправляется сигнал SIGALRM;

- ITIMER_PROF уменьшается, когда процесс выполняется в режиме ядра или задачи, как только itimer_prof=0 процессу отправляется сигнал SIGPROF;

- ITIMER_VIRT уменьшается, когда процесс выполняется в режиме задачи, как только itimer_virt=0 процессу отправляется сигнал SIGVTALRM.

Для установки таймеров всех типов используется системный вызов:

- settimer (BSD UNIX), измеряется в микросекундах;

- alarm (System V) - в секундах;

- hrtsys (UNIX SVR4) - в микросекундах.

С помощью вызова hrtsys совместимость достигается с BSD. Таймеры реального времени не обладают высокой точностью. Высокой точностью они обладают лишь для больших интервалов времени или для высокоприоритетных процессов.

Таймеры профилирования и виртуального времени обладают высокой точностью, так как не имеют отношение к реальному течению времени (процессу засчитывается тик целиком, даже если процесс выполняется лишь часть тика).

5.12. Контекст процесса

Каждый процесс UNIX имеет *контекст* – вся информация, требуемая для описания процесса.

Если выполнение процесса приостановлено, то информация сохраняется. Когда планировщик предоставляет процессу вычислительные ресурсы, информация восстанавливается.

Контекст состоит из:

1) адресного пространства процесса в режиме задачи (код, данные, стек процесса), других областей (разделяемая память, код, данные динамической библиотеки);

2) управляющей информации (proc, user, данные для отображения виртуального адресного пространства процесса в физическое);

3) окружения процесса (строки вида *переменная=значение*), наследуется дочерним процессом от родительского, хранятся в нижней части стека;

4) аппаратного контекста, включающего значения общих и ряда системных регистров:

- указатель инструкций – содержит адрес следующей структуры, которую нужно выполнить;

- указатель стека – содержит адрес последнего элемента стека;
- регистры плавающей точки;
- регистры управления памятью – отвечают за трансляцию виртуального адреса процесса в физический (рис. 5.5)

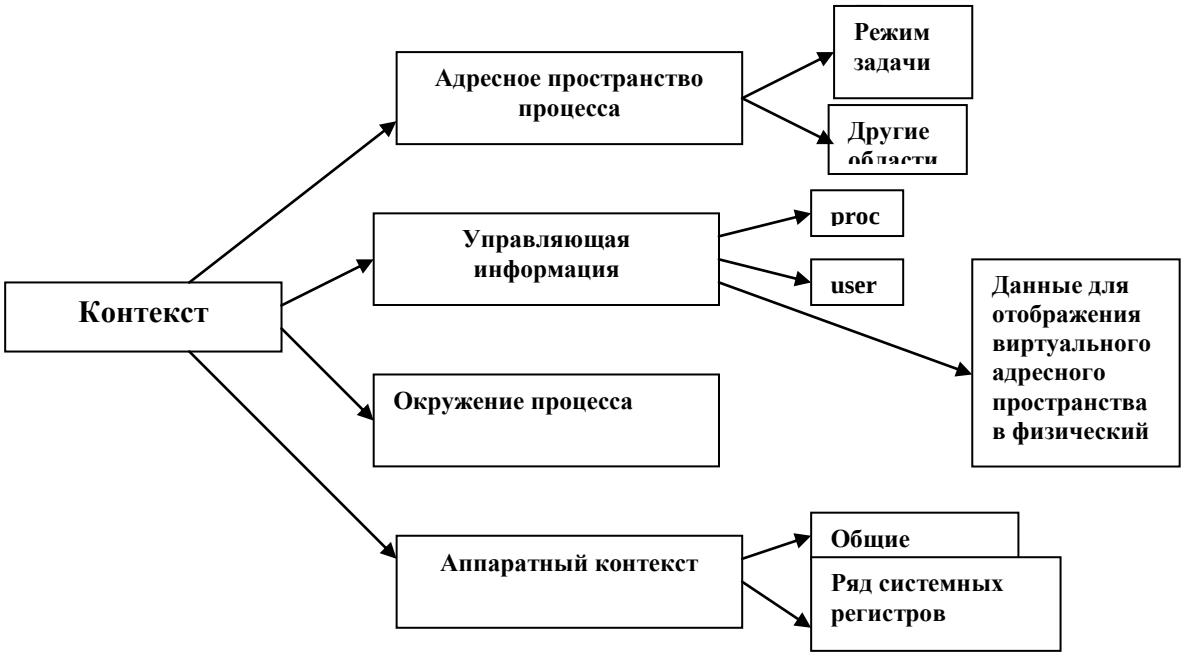


Рис.5.5. Состав контекста

Переключение между процессами выражается в переключении контекста и является достаточно трудоемкой задачей. Переключение контекста производится при 4-х ситуациях:

1) текущий процесс переходит в сон;
2) текущий процесс завершает выполнение;
3) пробуждение процесса с более высоким приоритетом;
4) пересчет приоритетов в очереди (в очереди находится процесс с более высоким приоритетом).

В-первых двух случаях ядро вызывает процедуры переключения из функций *sleep/exit.* В других случаях ядро устанавливает специальный флаг *runrun.* При переходе процесса из режима ядра в режим задачи ядро проверяет наличие этого флага и при его наличии вызывает функцию переключения контекста.

5.13. Жизненный цикл процесса

Жизненный цикл процесса может быть разбит на несколько состояний. Переход процесса из одного состояния в другое происходит в зависимости от наступления тех или иных событий в системе. На рис. 5.6 показаны состояния, в которых процесс может находиться с момента создания до завершения выполнения.

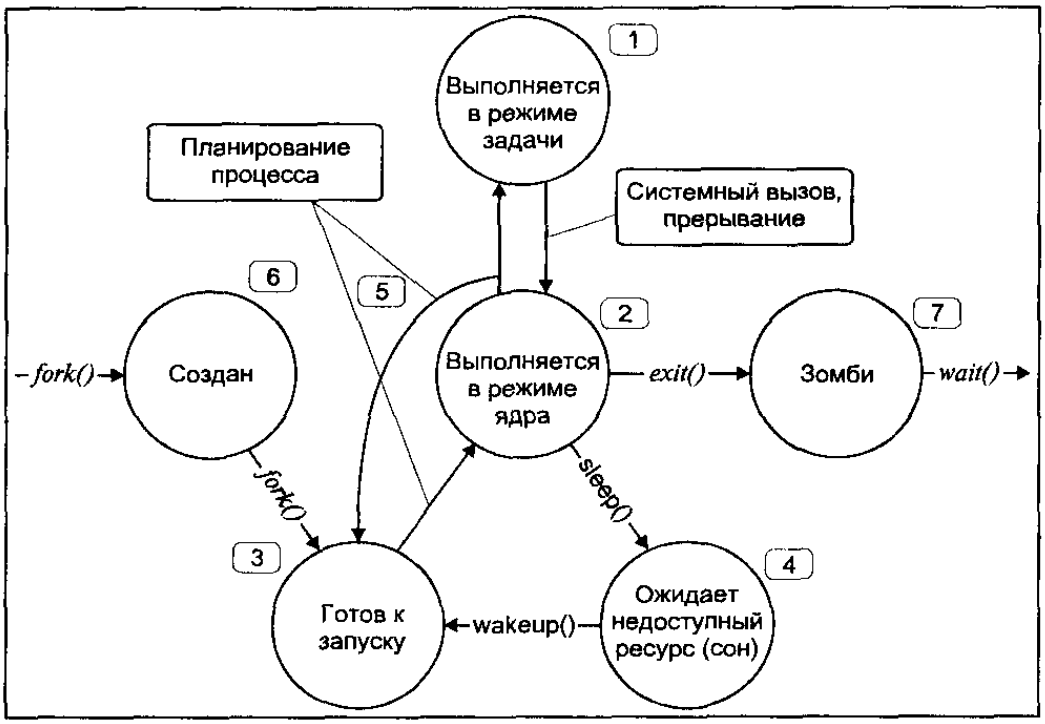


Рис. 5.6. Состояние процесса

1. Процесс выполняется в режиме задачи. При этом процессором выполняются прикладные инструкции данного процесса.
2. Процесс выполняется в режиме ядра. При этом процессором выполняются системные инструкции ядра операционной системы от имени процесса.
3. Процесс не выполняется, но готов к запуску, как только планировщик выберет его (состояние runnable). Процесс находится в очереди на выполнение и обладает всеми необходимыми ему ресурсами, кроме вычислительных.
4. Процесс находится в состоянии сна (asleep), ожидая недоступного в данный момент ресурса, например, завершения операции ввода/вывода.
5. Процесс возвращается из режима ядра в режим задачи, но ядро прерывает его и производит переключение контекста для запуска более высокоприоритетного процесса.
6. Процесс только что создан вызовом *fork(2)* и находится в переходном состоянии: он существует, но не готов к запуску и не находится в состоянии сна.
7. Процесс выполнил системный вызов *exit(2)* и перешел в состояние зомби (zombie, defunct). Как такового процесса не существует, но остаются записи, содержащие код возврата и временную статистику его выполнения, доступную для родительского процесса. Это состояние является конечным в жизненном цикле процесса.

Необходимо отметить, что не все процессы проходят через все множество состояний, приведенных выше. Процесс начинает свой жизненный путь с состояния 6, когда родительский процесс выполняет системный вызов *fork().* После того как создание процесса полностью завершено, процесс завершает "дочернюю часть" вызова *fork()* и переходит в состояние 3 готовности к запуску, ожидая своей очереди на выполнение. Когда планировщик выбирает процесс для выполнения, он переходит в состояние 1 и выполняется в режиме задачи.

Выполнение в режиме задачи завершается в результате системного вызова или прерывания, и процесс переходит в режим ядра, в котором выполняется код системного вызова или прерывания. После этого процесс опять может вернуться в режим задачи. Однако во время выполнения системного вызова в режиме ядра процессу может понадобиться недоступный в данный момент ресурс. Для ожидания доступа к такому ресурсу, процесс вызывает функцию ядра sleep () и переходит в состояние сна (4). При этом процесс добровольно освобождает вычислительные ресурсы, которые предоставляются следующему наиболее приоритетному процессу. Когда ресурс становится доступным, ядро "пробуждает процесс", используя функцию wakeup (), помещает его в очередь на выполнение, и процесс переходит в состояние "готов к запуску" (3).

При предоставлении процессу вычислительных ресурсов происходит *переключение контекста* (context switch), в результате которого сохраняется образ, или контекст текущего процесса, и управление передается новому. Переключение контекста может произойти, например, если процесс перешел в состояние сна, или если в состоянии готовности к запуску находится процесс с более высоким приоритетом, чем текущий. В последнем случае ядро не может немедленно прервать текущий процесс и произвести переключение контекста. Дело в том, что переключению контекста при выполнении в режиме ядра может привести к нарушению целостности самой системы. Поэтому переключение контекста откладывается до момента перехода процесса из режима ядра в режим задачи, когда все системные операции завершены, и структуры данных ядра находятся в нормальном состоянии.

Таким образом, после того как планировщик выбрал процесс на запуск, последний начинает свое выполнение в режиме ядра, где завершает переключение контекста. Дальнейшее состояние процесса зависит от его предыстории: если процесс был только что создан или был прерван, возвращаясь в режим задачи, он немедленно переходит в этот режим. Если процесс начинает выполнение после состояния сна, он продолжает выполняться в режиме ядра, завершая системный вызов. Заметим, что такой процесс может быть прерван после завершения системного вызова в момент перехода из режима ядра в режим задачи, если в очереди существует более высокоприоритетный процесс.

В UNIX 4.xBSD определены дополнительные состояния процесса, в первую очередь связанные с системой управления заданиями и взаимодействием

процесса с терминалом. Процесс может быть переведен в состояние "остановлен" с помощью сигналов останова SIGSTOP, SIGTTIN или SIGTTOU. В отличие от других сигналов, которые обрабатываются только для выполняющегося процесса, отправление этих сигналов приводит к немедленному изменению состояния процесса. Существует исключение из этого правила, касающееся процессов, находящихся в состоянии сна для низкоприоритетного события, т. е. события, вероятность наступления которого относительно мала, (например, ввода с клавиатуры, который может и не наступить). В этом случае отправление процессу сигнала приведет к его пробуждению. В этом случае, если процесс выполняется или находится в очереди на запуск, его состояние изменяется на "остановлен". Если же процесс находился в состоянии сна, его состояние изменится на "остановлен в состоянии сна". Выход из этих состояний осуществляется сигналом продолжения SIGCONT, при этом из состояния "остановлен" процесс переходит в состояние "готов к запуску", а для процесса, остановленного в состоянии сна, следующим пунктом назначения является продолжение "сна". Описанные возможности полностью реализованы и в SVR4.

Наконец, процесс выполняет системный вызов *exit()* и заканчивает свое выполнение. Процесс может быть также завершен вследствие получения сигнала. В обоих случаях ядро освобождает ресурсы, принадлежавшие процессу, за исключением кода возврата и статистики его выполнения, и переводит процесс в состояние "зомби". В этом состоянии процесс находится до тех пор, пока родительский процесс не выполнит один из системных вызовов *wait(),* после чего вся информация о процессе будет уничтожена, а родитель получит код возврата завершившегося процесса.

5.14. Принципы управления памятью

Одной из основных функций операционной системы является эффективное управление памятью. Оперативная память, или основная память, или память с произвольным доступом (Random Access Memory, RAM) является достаточно дорогостоящим ресурсом. Время доступа к оперативной памяти составляет всего несколько циклов процессора, поэтому работа с данными, находящимися в памяти, обеспечивает максимальную производительность. К сожалению, данный ресурс, как правило, ограничен. В большей степени это справедливо для многозадачной операционной системы общего назначения, какой и является UNIX. Поэтому данные, которые не могут быть размещены в оперативной памяти, располагаются на вторичных устройствах хранения, или во вторичной памяти, роль которой обычно выполняют дисковые накопители. Время доступа к вторичной памяти на несколько порядков превышает время доступа к оперативной памяти и требует активного содействия операционной системы. Подсистема управления памятью UNIX отвечает за справедливое и эффективное распределение разделяемого ресурса оперативной памяти между процессами и за обмен данными между оперативной и вторичной памятью. Часть операций производится аппаратно устройством управления памятью

(Memory Management Unit, MMU) процессора под управлением операционной системы, чем достигается требуемое быстродействие.

Примитивное управление памятью значительно уменьшает функциональность операционной системы. Такие системы, как правило, позволяют загрузить в заранее определенное место в оперативной памяти единственную задачу и передать ей управление. При этом задача получает в свое распоряжение все ресурсы компьютера (разделяя их, разумеется, с операционной системой), а адреса, используемые задачей, являются физическими адресами оперативной памяти. Такой способ запуска и выполнения одной программы, безусловно, является наиболее быстрым и включает минимальные накладные расходы.

Этот подход часто используется в специализированных микропроцессорных системах, однако практически неприменим в операционных системах общего назначения, какой является UNIX. Можно сформулировать ряд возможностей, которые должна обеспечивать подсистема управления памятью современной многозадачной операционной системы:

- Выполнение задач, размер которых превышает размер оперативной памяти.
- Выполнение частично загруженных в память задач для минимизации времени их запуска.
- Размещение нескольких задач в памяти одновременно для повышения эффективности использования процессора.
- Размещение задачи в произвольном месте оперативной памяти.
- Размещение задачи в нескольких различных частях оперативной памяти.
- Совместное использование несколькими задачами одних и тех же областей памяти. Например, несколько процессов, выполняющих одну и ту же программу, могут совместно использовать сегмент кода.

Все эти возможности реализованы в современных версиях UNIX с помощью *виртуальной памяти.* Виртуальная память не является "бесплатным приложением", повышая накладные расходы операционной системы: структуры данных управления памятью размещаются в оперативной памяти, уменьшая ее размер; управление виртуальной памятью процесса может требовать ресурсоемких операций ввода/вывода; для системы со средней загрузкой около 1% процессорного времени приходится на подсистему управления памятью. Поэтому от эффективности реализации и работы этой подсистемы во многом зависит производительность операционной системы в целом.

5.15. Адресное пространство процесса в режимах ядра и задачи

Адресное пространство ядра обычно совпадает с адресным пространством выполняющегося в данный момент процесса. В этом случае говорят, что ядро расположено в том же *контексте,* что и процесс. Каждый раз, когда процессу передаются вычислительные ресурсы, система восстанавливает контекст задачи этого процесса, включающий значения регистров общего назначения, сегментных

регистров, а также указатели на таблицы страниц, отображающие виртуальную память процесса в режиме задачи. При этом системный контекст остается неизменным для всех процессов.

Специальный регистр R указывает на расположение каталога таблиц страниц в памяти. В SCO UNIX используется только один каталог, независимо от выполняющегося процесса, таким образом, значение регистра R не меняется на протяжении жизни системы. Поскольку ядро (код и данные) является частью выполняющегося процесса, таблицы страниц, отображающие старший 1 Гбайт виртуальной памяти, принадлежащей ядру системы, не изменяются при переключении между процессами. Для отображения ядра используются старшие 256 элементов каталога.

При переключении между процессами, однако, изменяется адресное пространство режима задачи, что вызывает необходимость изменения оставшихся 768 элементов каталога. В совокупности они отображают 3 Гбайт виртуального адресного пространства процесса в режиме задачи. Таким образом, при смене процесса адресное пространство нового процесса становится видимым (отображаемым), в то время как адресное пространство предыдущего процесса является недоступным. При этом физические страницы, принадлежащие предыдущему процессу, могут по-прежнему оставаться в памяти, однако доступ к ним невозможен ввиду отсутствия установленного отображения. Любой допустимый виртуальный адрес будет отображаться либо в странице ядра, либо в страницы нового процесса.

Формат виртуальной памяти процесса в режиме задачи зависит в первую очередь от типа исполняемого файла, образом которого является процесс. На рис. 5.7 изображено расположение различных сегментов процесса в виртуальной памяти для форматов исполняемых файлов - COFF и ELF.

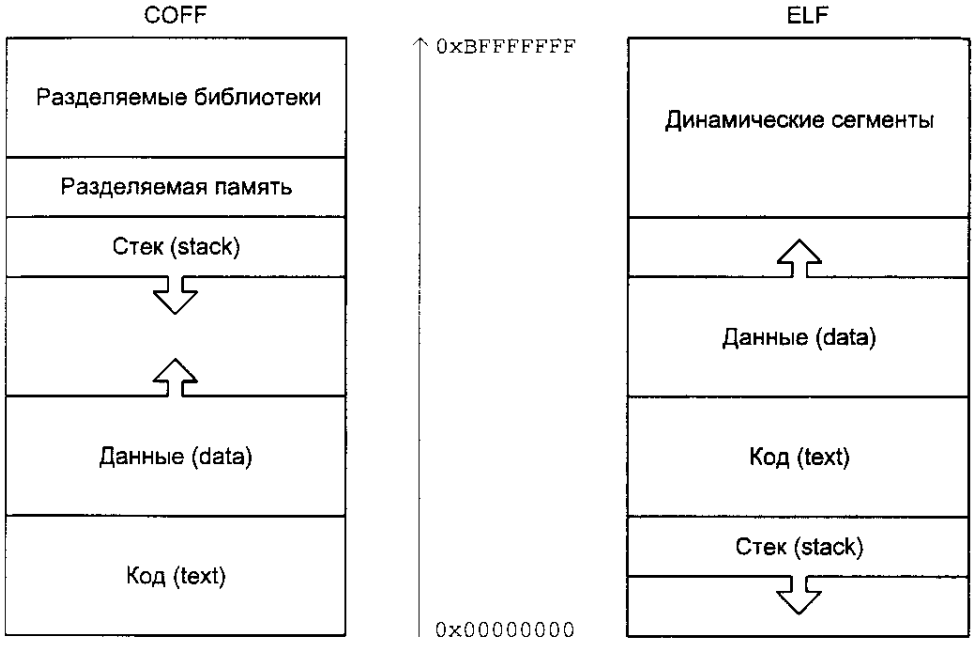


Рис. 5.7. Виртуальная память процесса в режиме задачи

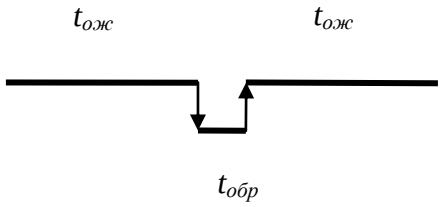
Заметим, что независимо от формата исполняемого файла виртуальные адреса процесса не могут выходить за пределы 3 Гбайт. Для защиты виртуальной памяти процесса от модификации другими процессами прикладные задачи не могут менять заданное отображение. Поскольку ядро системы выполняется на привилегированном уровне, оно может управлять отображением, как собственного адресного пространства, так и адресного пространства процесса.

5.16. Планирование выполнения процессов

В UNIX выполняются несколько приложений, так это многозадачная система. И, как следствие, приложения предъявляют различные требования к системе с точки зрения их планирования и общей производительности. Выделяют три класса приложений:

1) *Интерактивные.* Большую часть времени данные приложения проводят в ожидании пользовательского ввода, и должны быстро обрабатывать такие действия, допустимая задержка – от 100 до 200 миллисекунд. К ним относятся командные интерпретаторы, текстовые редакторы и другие программы, непосредственно взаимодействующие с пользователем. В качестве критерия выступает минимальное время отклика:

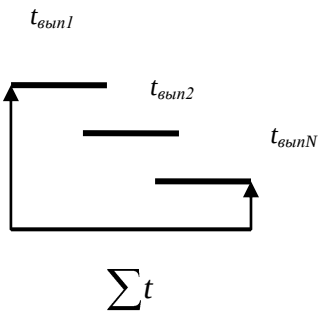
$$И^{П} = \begin{cases} ТР \\ КИ \\ идр. \end{cases} \Rightarrow$$



$$t_{отклика} \to \min$$

2) *Фоновые.* Данный класс приложений не требует вмешательства пользователя. К ним относятся сложные вычислительные программы, компиляция программного обеспечения. Для этого класса приложений необходимо обеспечить высокую производительность системы, т.е. минимизировать суммарное время выполнения задания в системе, загруженной другими процессами:

$$ФП = \begin{cases} КПО \\ СлВП \end{cases} \Rightarrow$$



$$\sum t \to \min$$

3) *Приложения реального времени.* К этому классу относятся видеоприложения, системы управления, измерительные комплексы. Система должна обеспечивать гарантированное время обработки кадра (для

видеоприложения), время совершения операции (для измерительного комплекса), время отклика (для систем управления)

$$ПРВ = \begin{cases} ВПр & t_{обрк} \\ ИзК & \Rightarrow гарантия\, t_{совоп} \\ СУ & t_{от} \end{cases}$$

Планировщик выбирает, какому из процессов предоставить вычислительные ресурсы системы. Его задача – найти золотую середину, т.е. обеспечить максимальную эффективность и производительность системы в целом. В традиционных UNIX системах существуют различные механизмы планирования процессов.

Контрольные вопросы

1. Перечислите пользовательские идентификаторы процесса.
2. Какие идентификаторы процесса определяют права доступа процесса к файлам в процессе выполнения?
3. Какие идентификаторы процесса определяют реального владельца процесса?
4. Какой процесс порождается с использованием системного вызова fork?
5. Как процесс может получить дополнительные привилегии?
6. Перечислите команды, позволяющие вывести список выполняемых в системе процессов и их атрибутов.
7. Где хранится информация об открытых файловых дескрипторах, о диспозиции сигналов, о статистике выполняемых процессов?
8. Какой стек используется процессом при выполнении в режиме ядра?
9. Перечислите режимы выполнения процесса?
10. Что включает образ процесса в режиме задачи?

РАЗДЕЛ 6. ПОДСИСТЕМА ВВОДА /ВЫВОДА

Уведомление обменом данных между памятью и периферийными устройствами осуществляется при помощи подсистемы ввода/вывода. Подсистема ввода/вывода ядра обеспечивает работу интерфейсов как высокого уровня (файловая система), так и низкого (взаимодействие с физическим устройством).

Основными компонентами подсистемы ввода/вывода являются *драйверы* – модули ядра, обеспечивающие непосредственную работу с периферийными устройствами. Драйвер управляет одним или несколькими устройствами и представляет собой интерфейс между устройством и относительной частью ядра (рис. 6.1).

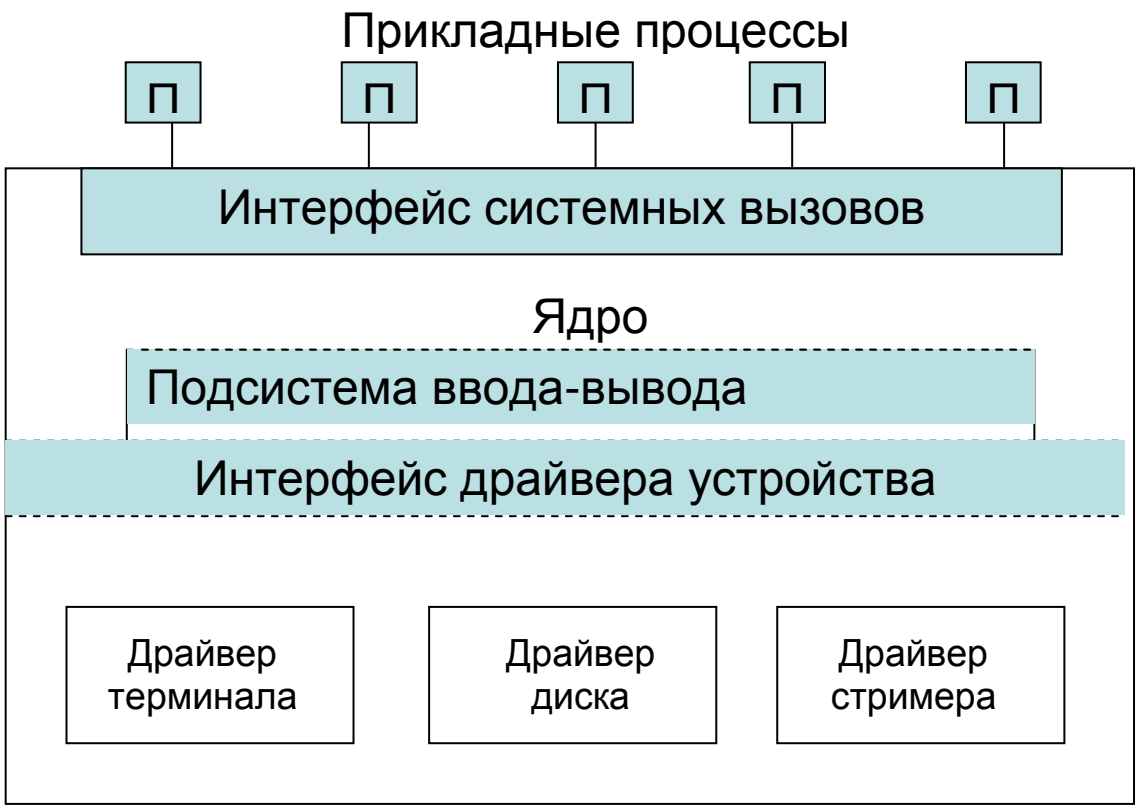


Рис. 6.1. Схема взаимодействие подсистемы ввода/вывода с ядром

Драйвер - единственный модуль ядра, который может взаимодействовать с устройством, является обособленным. Существует большое количество драйверов. Они различаются:

- по возможностям, которые предоставляют;
- по тому, каким образом обеспечивается к ним доступ и управление.

Можно выделить три основных типа драйверов:

Символьные

- обеспечивают работу с устройствами с побайтовым доступом и обменом данными (модемы, терминалы, принтеры, манипуляторы мышь и т.д.);
- не используется буферный кэш;
- при необходимости буферизации используется подход, основанный на структурах данных, называемых clist.

Блочные

- обмен данными с устройством осуществляется фиксированными порциями (блоками);
- используется буферный кэш, который является интерфейсом между файловой системой и устройством;
- допускается передача и обмен данными, размер которых меньше размера блока.

Драйверы низкого уровня

- позволяют производить обмен данными с блочными устройствами, минуя кэш;

- ядро производит передачу данных непосредственно между пользовательским процессом и устройством, без дополнительного копирования.

Часть драйверов не имеют непосредственного отношения к аппаратной части компьютера и служат для предоставления различных услуг ядра прикладным процессам. Такие драйверы называются *программными* или драйверами *псевдоустройств.*

Рассмотрим несколько примеров псевдоустройств и соответствующих им программных драйверов:

- **/dev/mem** – обеспечивает доступ к физической памяти компьютера;

- **/dev/kmem** – обеспечивает доступ к виртуальной памяти ядра;

- **/dev/ksyms** – обеспечивает доступ к разделу исполняемого файла ядра, содержащего таблицу символов, совместно с драйвером /dev/kmem обеспечивает удобный интерфейс для анализа внутренних структур ядра;

- **/dev/null** – «нулевое устройство», служит двум целям:

1) удаляет любые данные, направленные в это устройство. В тех случаях, когда выводные данные программы не нужны, в качестве выходного файла назначают устройство /dev/null, например:

$ verbose _command > /dev/null

2) при чтении из устройства /dev/null всегда возвращается признак конца строки. Если открыть файл /dev/null с помощью функции open() и попытаться прочесть данные из него с помощью функции read(), функция вернет 0 байтов. При копировании файла /dev/null в другое место будет создан пустой файл нулевой длины.

$ cp /dev/null empty – file

$ ls - l empty – file

-rw-rw---- 1 nata nata 0 Nov 12 00:14 empty – file

- **/dev/zero** – обеспечивает заполнение указанного буфера нулями. Этот драйвер часто используется для инициализации области памяти и ведет себя так, как файл бесконечной длины. Файл используется в функциях выделения памяти, которые отображают этот файл в памяти, чтобы инициализировать выделяемые сегменты нулями.

- **/dev/full** – устройство ведет себя так, как если бы оно было файлом в файловой системе, в которой нет свободного места. Операция записи в этот файл будет завершаться ошибкой.

$ cp /etc/fstab /dev/full

cp: /dev/full: No space left on device.

Этот файл удобен для проверки того, как будет вести себя программа в том случае, что при записи ее в файл возникнет нехватка места.

6.1. Базовая архитектура драйверов

Доступ к драйверу осуществляется ядром через специальную структуру данных, которая называется *коммутатор устройств*, каждый элемент которой содержит указатели на соответствующие функции драйвера – *точки входа*. Коммутатор устройств определяет абстрактный интерфейс драйвера устройства. Драйвер устройства адресуется *старшим номером* устройства *(major number)* и *младшим номером* устройства *(minor numbe)*.

Младший номер (minor number) интерпретируется самим драйвером, например, к какому разделу устройства требуется обеспечить доступ.

Используя различные младшие номера:

- процесс может получить доступ к различным разделам жесткого диска, который обслуживается одним драйвером;

- несколько процессов могут осуществлять одновременную, независимую работу с устройством или псевдоустройством. В качестве примера можно привести сетевые протоколы, реализованные в виде драйверов.

Старший номер – это указатель на элемент коммутатора устройств, то есть обеспечивает ядру возможность вызова необходимой функции указанного драйвера.

Ядро содержит коммутаторы устройств двух типов: для блочных (bdevsw) и для символьных (cdevsw) устройств. Для каждого типа коммутатора ядро размещает соответствующий массив. Если драйвер обеспечивает как блочный, так и символьный интерфейсы, его точки входа будут представлены в обоих массивах.

Ядро вызывает функцию открыть *open()* требуемого драйвера следующим образом:

*(*bdevsw [getmajor (dev)] .d_open) (dev, ...);*

передавая ей в качестве одного из параметров переменную dev, содержащую старший и младший номер.

Макрос getmajor() служит для извлечения старшего номера из переменной dev. Благодаря этому драйвер имеет возможность определить, с каким младшим номером была вызвана функция open(), и выполнить соответствующие действия.

В ядре системы одновременно присутствует большое количество различных драйверов, поэтому каждый из них должен иметь уникальное имя во избежание проблем при редактировании связей. В названиях точек входа драйвера используются определенные соглашения -уникальное двухсимвольное обозначение, используемое в качестве префикса названий функций. Например: драйвер /dev/kmem имеет префикс mm, таким образом, функции этого драйвера будут иметь название *mmwrite(), mmread()* и т.д.

Стандартные точки входа драйвера отличаются от разных версий UNIX. К типичным точкам входа относятся:

- xxopen() – обеспечивает необходимую реинициализацию физического устройства и внутренних данных драйвера;

- xxread() - производит чтение данных из устройства;
- xxwrite() - производит запись данных в устройство;
- xxpol() – производит опрос устройства. Обычно используется для устройств, не поддерживающих прерывания, например, для определения поступления данных для чтения;
- xxstrategy() – общая точка входа для операций блочного ввода/вывода;
- xxintr() - вызывается при поступлении прерывания, связанного с данным устройством, может выполнить копирование данных от устройств в промежуточные буферы, которые затем считываются функцией xxread() по запросу прикладного процесса;
- xxhalt() – вызывается для останова драйвера при останове системы или при выгрузке драйвера.
- xxioctl – является общим интерфейсом управления устройством.

6.2. Вызов кодов драйвера

Ядро обращается к функциям драйвера в зависимости от запроса. Можно выделить основные случаи, в которых ядро обращается к функциям драйвера:

- настройка – ядро обращается к драйверу во время загрузки системы с целью проверки и инициализации устройства;
- ввод/вывод – подсистема ввода-вывода вызывает драйвер для чтения или записи данных;
- обработка прерываний – по завершении операции или при изменении состояния устройства генерируется прерывание;
- управление – пользователь может создать управляющий запрос к устройству, например, открытие или закрытие устройства;
- реинициализация или останов.

Функции настройки устройства обрабатываются только один раз при загрузке системы. Операции ввода-вывода и управления являются синхронными. Они запускаются в ответ на определенные запросы процесса и выполняются в его контексте. Процедура strategy блочного драйвера являются исключением из правила. Прерывание – это асинхронные события в системе, так как ядро не может заранее знать о времени их возникновения. Их обработка не производится в контексте какого-либо определенного процесса.

Схема обработки запроса ядром UNIX различна для символьных (рис. 6.2) и блочных устройств (рис. 6.3).

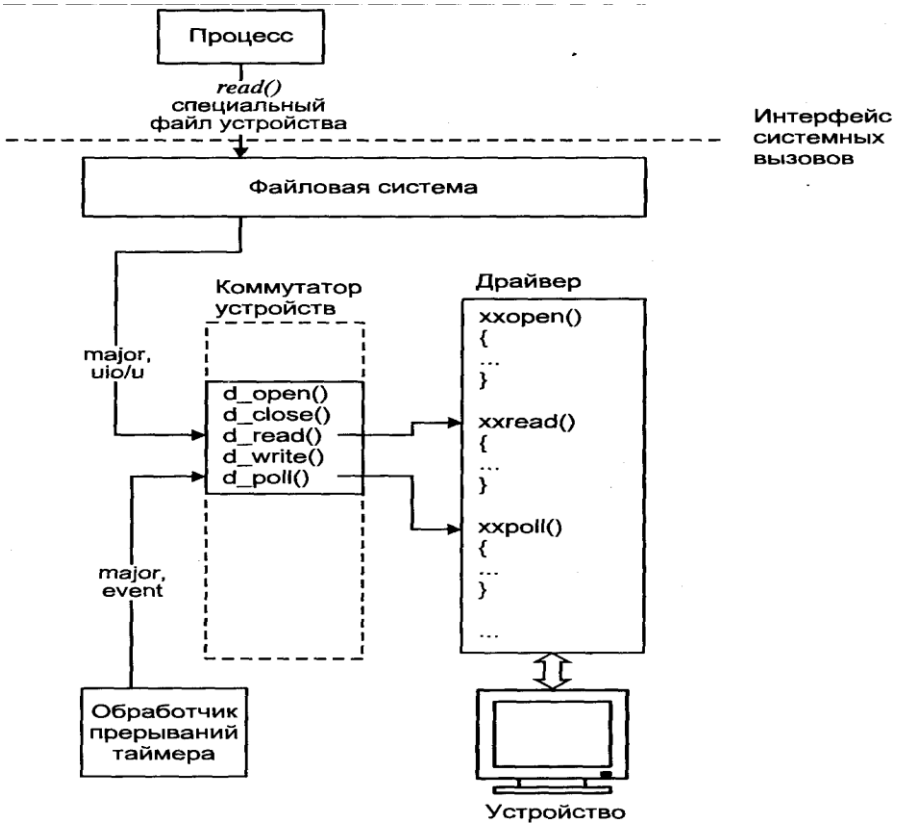


Рис. 6.2. Доступ к драйверу символьного устройства

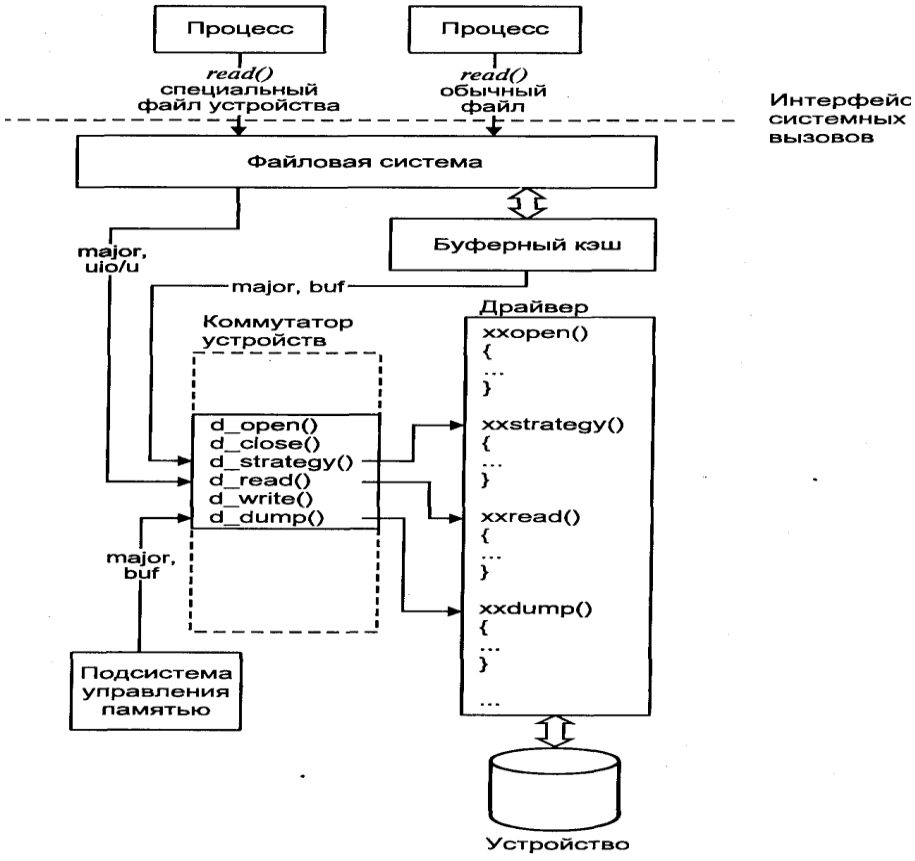
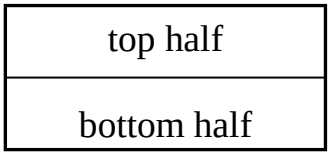


Рис. 6.3. Доступ к драйверу блочного устройства

Большинство функций драйвера отвечают за передачу данных и осуществляют копирование информации из адресного пространства ядра, в котором находится сам драйвер, в адресное пространство задачи. Когда ядро вызывает функцию драйвера, то все действия выполняются в системном контексте процесса. Схема вызова функций может быть различной:

- если функция вызвана по запросу процесса, то она имеет *контекст задачи*;
- если функция вызвана по запросу другой подсистемы ядра операционной системы, то речь идет о *системном контексте*;
- если функция вызвана в процессе обработки прерывания, то она имеет *контекст прерывания* – специальный вид системного контекста.

Различия в контексте и причинах вызова тех или иных функций драйвера позволяют представить драйвер устройства, состоящим из двух частей:



где top half – верхняя часть, a bottom half – нижняя часть.

Верхняя часть:

- содержит синхронные процедуры, то есть функции вызываются по определенным запросам прикладного процесса и выполняются в его контексте;
- для функций доступно адресное пространство u-area процесса;
- функции могут осуществлять перевод процесса в состояние сна.

Таким образом, функции ввода/вывода и управления принадлежат верхней части драйвера.

Нижняя часть:

- вызов функций носит асинхронный характер. Момент вызова функции обработки нельзя предугадать;
- ядро не может контролировать время вызова функции;
- выполнение функций происходит в контексте ядра и не имеет никакого отношения к контексту текущего процесса;
- функциям не доступно адресное пространство u-area процесса;
- функции не могут осуществлять перевод процесса в состояние сна, так как это может привести к блокировке другого процесса.

Обе составляющие драйвера должны обладать средствами синхронизации своих действий друг с другом. Если объект доступен обеим частям, то процедуры верхнего уровня должны производить блокировку прерываний (повысить уровень ipl) при манипуляции таким объектом. В ином случае прерывание может поступить в тот момент, когда объект находится во внутренне противоречивом состоянии, что может привести к нарушению целостности данных и системы.

6.3. Файловый интерфейс

Доступ к периферии осуществляется с помощью специальных файловых устройств, расположенных в корневой файловой системе некоторого типа, например ufs. В соответствии с архитектурой виртуальной файловой системы все операции с этими файлами будут обслуживаться соответствующими функциями реальной файловой системы, в данном случае ufs.

Однако специальный файл устройства не является обычным файлом системы ufs. Фактически все операции со специальным файлом устройства выполняются драйвером и не зависят от типа файловой системы.

Драйвер конкретного устройства может адресоваться несколькими специальными файлами устройств, в том числе расположенными в различных файловых системах. Следовательно, ядро не в состоянии определить фактическое число связей прикладных процессов с данным устройством. Решением проблемы является наличие специального snode, который позволяет контролировать доступ к конкретному устройству.

Этот объект получил название *common snode* – общий snode, и является единственным интерфейсом доступа к драйверу устройства.

Для каждого устройства существует единственный common snode, который создается при первом доступе к устройству. Каждый специальный файл устройства имеет собственный snode в файловой системе specfs и соответствующей ему vnode, а также inode физической файловой системы и соответствующей ему vnode.

6.4. Встраивание драйверов в ядро

Существует два основных метода встраивания кода и данных драйвера в ядро операционной системы:

- *перекомпиляция ядра:*
  - статическое размещение драйвера;
  - встраивается независимо от фактического наличия устройства;
  - код и данные драйвера будут присутствовать в ядре до следующей перекомпиляции;
  - традиционный способ в UNIX.
- *динамическая загрузка драйвера*:
  - в процессе загрузки системы;
  - динамические расширения функциональности ядра;
  - ядро содержит набор функций, позволяющих загрузить необходимые драйверы и выгрузить их, когда уже завершена работа;
  - тенденция в современных UNIX системах.

Динамическая установка драйвера в ядро операционной системы требует выполнения следующих операций:

- размещение и динамическое связывание символов драйвера. Эта операция аналогична загрузке динамических библиотек, и выполняется специальным загрузчиком;

- инициализация драйвера и устройства;
- добавление точек входа драйвера в соответствующий коммутатор устройств;
- установка обработчика драйвера.

Код динамически загружаемых драйверов сложнее, и содержит стандартные точки входа, ряд функций, отвечающих за загрузку и выгрузку драйвера, а также ряд дополнительных функций и структур. Например: функция инициализации и установки, вызываемая при загрузке драйвера; структура, экспортируемая ядру при загрузке драйвера, в частности, содержит адреса точек входа в драйвер.

6.5. Блочные устройства

У блочных устройств обмен происходит фрагментами фиксированной длины (блоками). Эти драйверы используются:

- файловой подсистемой;
- подсистемой управления памятью.

Типичными представителями блочных устройств являются жесткие и гибкие диски.

Блочные устройства можно разделить на два типа, в зависимости от того, используются ли они для хранения файловой системы или являются «неформатированными» устройствами. Следовательно, различается схема доступа к этим устройствам.

Если блочное устройство не используется для хранения файловой системы, то доступ к устройству осуществляется только через специальный файл устройства, предоставляющий интерфейс низкого уровня. Доступ к таким устройствам, как правило, осуществляется процессом косвенно (через запросы к файловой системе).

Операции ввода/вывода для блочного устройства могут быть вызваны рядом событий:

- чтением или записью в обычный файл;
- чтением или записью в специальный файл устройства;
- операциями подсистемы управления памятью;
- страничным замещением или свопингом.

Доступ к блочным устройствам осуществляется с помощью трех основных точек входа: xxopen(), xxclose(), xxstrategy().

Аргументом, передаваемым функцией xxstrategy(), является указатель на структуру *buf* (заголовок буфера обмена). Структура buf используется как интерфейс взаимодействия между ядром и драйвером блочного устройства и содержит всю информацию для операций ввода/вывода информации, в том числе: старший и младший номер устройства; номер начального блока данных в устройстве; число передаваемых байтов; месторасположение данных в памяти; флаги, указывающие на тип операции, а также на ее синхронность; адрес процедуры завершения, вызываемой обработчиком прерываний.

Ядро адресует дисковый блок, указывая vnode и смещение. Если доступ осуществляется к специальному файлу устройства, то смещение является физическим, отсчитываемым начала устройства. Если vnode представляет обычный файл, то смещение является логическим, отсчитываемым от начала файла.

Таким образом, блок устройства, содержащего файловую систему, может быть адресован двумя способами:

- через обычный файл и логическое смещение;
- через специальный файл устройства и физическое смещение на этом устройстве.

Это может привести к различной идентификации одного и того же блока, и как следствие, к двум различным копиям блока в памяти. Данная ситуация может привести к потере или нарушению целостности данных. Поэтому непосредственный доступ к специальному файлу такого устройства возможен только при размонтированной файловой системе.

6.6. Символьные устройства

Подсистема ввода/вывода играет малую роль в организации ввода/вывода символьных устройств. Большую часть операций выполняет драйвер устройства. Основное отличие символьных устройств от блочных заключается в том, что они, как правило, передают небольшие объемы данных (побайтно). Обмен данными с символьными устройствами происходит непосредственно через драйвер, минуя буферный кэш. При этом данные, как правило, копируются в драйвер из адресного пространства процесса, запросившего операцию ввода / вывода.

Если процесс сделал системный вызов ввода/вывода, то специальным файлом символьного устройства запрос направляется в файловую подсистему.

Ядро выполняет вызов функции *spec_read()* или *spec_write().* Далее проверяется тип объекта vnode и определяется, что устройство является символьным. С помощью коммутатора выбирают точки входа драйвера, используя старший номер. Затем вызывается сама функция (xxread или xxwrite) и передает в качестве параметров:

- старший и младший номера;
- ряд дополнительных параметров;
- явно и неявно адресует область копирования данных в адресном пространстве процесса.

После завершения передачи данных ядро возвращает количество считанных байтов.

6.7. Интерфейс доступа низкого уровня

Символьные драйверы могут обеспечивать доступ к блочным устройствам, таким как диски или накопители на магнитной ленте, то есть

служат в качестве интерфейса доступа низкого уровня. Такие драйверы отличаются характером выполнения операций ввода вывода.

Если блочные устройства осуществляют обмен данными через буферный кэш, то драйверы низкого уровня непосредственно осуществляют связь с адресным пространством процесса. Данный интерфейс используется системными утилитами (fsck), приложениями, работающими с накопителями на магнитной ленте (tar, cpio), СУБД, обеспечивающими самостоятельно оптимизированные механизмы кэширования данных на уровне задачи.

На рис. 6.4 показаны различные типы доступа к блочным устройствам: при участии буферного кэша (драйверы блочных устройств) и манипуляция буфером производится драйвером самостоятельно (драйверы низкого уровня).

Очевидно, что побайтовая передача между драйвером символьного устройства и прикладным процессом весьма неэффективна (так как байт сначала копируется в адресное пространство драйвера, далее некоторое время должно пройти, и только после этого символ передается физическому устройству). Если устройство занято, процесс ожидает завершения операции и, как правило, процесс переходит в состояние сна или переключению контекста.

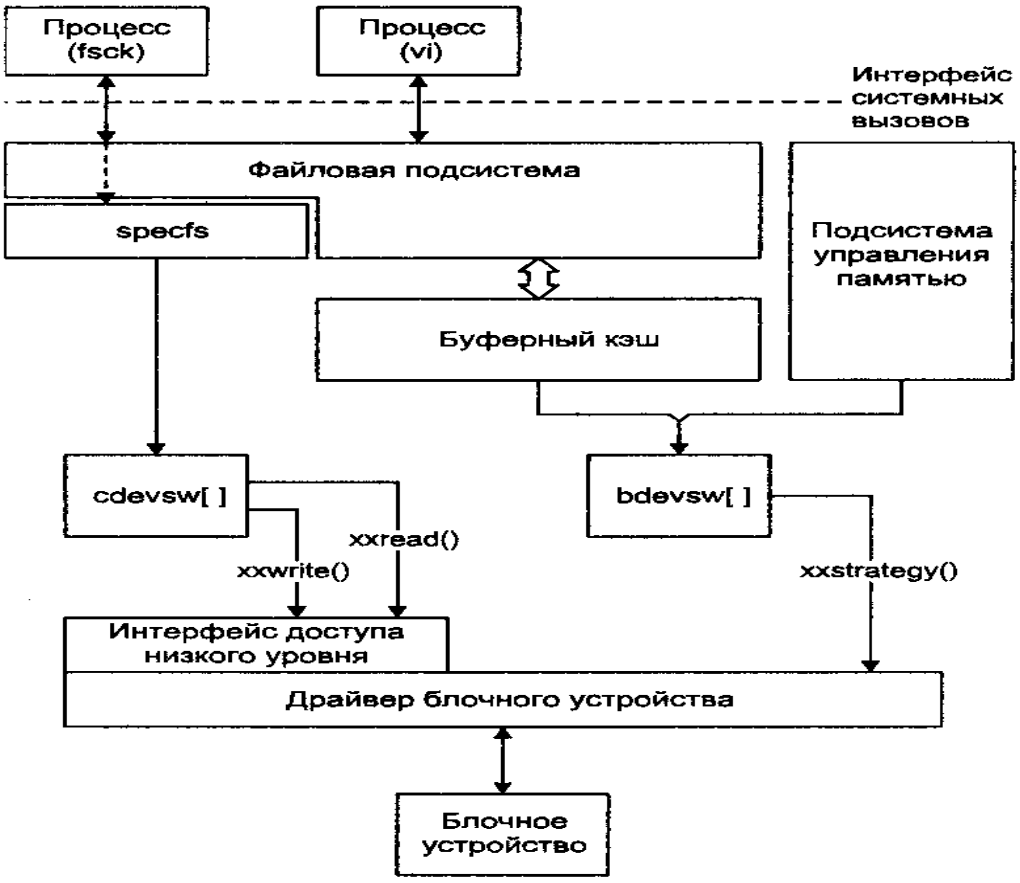


Рис. 6.4. Различные типы доступа к блочным устройствам

Есть два решения преодоления данной ситуации:

- использование прерываний - функция обработки записывает данные в буфер, и затем считываются функцией xxwrite ().

- эмуляция прерываний - вызов функции ядром через определенные промежутки времени, что обеспечивает независимое считывание и буферизацию данных.

Буферизация для символьных устройств осуществляется с помощью специальных структур данных – *clist*. Каждая структура *clist* имеет следующие поля:

- int c_cc - поле содержит число символов в буфере cblock;
- struct cblock *c_cf - первый элемент cblock;
- struct cblock *c_cl - последний элемент cblock.

Первый и последние элементы организованы в виде списка и фактически обеспечивают буферы хранения данных. Каждая структура cblock может хранить несколько символов. Когда буфер хранения заполняется, ядро автоматически выделяет новую структуру и помещает ее в связанный список.

Контрольные вопросы

1. Перечислите основные классификационные признаки драйверов в UNIX .
2. Как ядром осуществляется доступ к драйверу?
3. Перечислите основные компоненты подсистемы ввода/вывода ОС UNIX.
4. Как называется программный драйвер, подавляющий вывод сообщений об ошибках?
5. Как называется программный драйвер, обеспечивающий доступ к физической памяти компьютера?
6. Как называется программный драйвер, обеспечивающий доступ к виртуальной памяти компьютера?
7. Посредством чего ядро предоставляет различные услуги прикладным процессам?
8. Как называется программный драйвер, подавляющий вывод сообщений об ошибках?
9. Как ядро определяет фактическое число связей прикладных процессов с данным устройством?
10. Перечислите отличия блочных и символьных драйверов.
11. Перечислите отличия символьных и драйверов низкого уровня

РАЗДЕЛ 7. ПОЛЬЗОВАТЕЛЬСКАЯ СРЕДА UNIX

Базовой пользовательской средой UNIX является окно алфавитно-цифрового терминала. Однако сегодня, в условиях конкуренции на рынке ОС для ПК, характер работы в UNIX существенно отличается от того, каким он был, скажем, двадцать лет назад. Графический многооконный интерфейс, системы меню, техника drag-and-drop - все это, казалось бы, стирает различия в работе с ОС UNIX и, например, с ОС Windows.

Несмотря на это, интерфейс командной строки - это самый непосредственный способ выполнения множества небольших задач администрирования в среде UNIX. Программа, с которой работает пользователь - командный интерпретатор shell. Поэтому рассмотрим базовый пример работы в UNIX - использование командной строки интерпретатора shell.

7.1. Командный интерпретатор Shell

Все современные системы UNIX поставляются, по крайней мере, с тремя командными интерпретаторами: Bourne shell **(/bin/sh),** C shell **(/bin/csh)** и Korn shell **(/bin/ksh).** Существует еще несколько интерпретаторов, например Bourne-Again shell (bash), со сходными функциями.

Командный интерпретатор занимает важное место в операционной системе UNIX, прежде всего, благодаря следующим обстоятельствам:

1) первая программа, с которой по существу начинается работа пользователя, - shell;

2) командный интерпретатор является удобным средством программирования. Синтаксис языка различных командных интерпретаторов несколько отличается. С помощью shell можно создавать сложные программы, конструируя их из существующих утилит UNIX. Программы на языке shell часто называют *скриптами* или *сценариями* (script). Интерпретатор считывает строки из файла-скрипта и выполняет их, как если бы они были введены пользователем в командной строке;

3) при входе пользователя в систему запускается его инициализационный скрипт, выполняющий несколько функций: установку пути поиска программ, инициализацию терминала, определение расположения почтового ящика. Помимо этого может быть выполнен целый ряд полезных действий, например, установка приглашения. В этом файле можно добавить необходимые пути поиска. Инициализационный скрипт находится в домашнем каталоге пользователя. Для разных командных интерпретаторов используются различные скрипты инициализации:

Командный интерпретатор	Скрипт инициализации
Bourne shell (sh)	.profile
C shell (csh)	.login и .cshrc
Korn shell (ksh)	.profile и .kshrc
Bourne-Again shell (bash)	.profile и .bashrc

Скрипты .profile и .login выполняются при первом входе в систему. Скрипты .cshrc, .kshrc и .bashrc выполняются при каждом запуске интерпретатора;

4) основная инициализация операционной системы происходит в результате выполнения скриптов shell. Если понадобиться модифицировать процесс инициализации (например, добавить новый системный сервис), то необходимо обратиться к текстам этих скриптов.

7.2. Сценарий работы

В UNIX реализуется следующий сценарий работы в системе:

- При включении терминала активизируется процесс *getty(),* который является сервером терминального доступа и запускает программу *login(),* которая, в свою очередь, запрашивает у пользователя имя и пароль.

Процесс *getty()* устанавливает параметры, относящиеся к типу терминала и безопасности системы;

- Если пользователь зарегистрирован в системе и ввел правильный пароль, *login()* запускает программу, указанную в последнем поле записи пользователя в файле **/etc/passwd.** В принципе это может быть любая программа, как правило, это командный интерпретатор shell.

- Shell выполняет соответствующий командный файл инициализации, и выдает на терминал пользователя приглашение. С этого момента пользователь может вводить команды.

- Shell считывает ввод пользователя, производит синтаксический анализ введенной строки, подстановку шаблонов и выполняет действие, предписанное пользователем (это может быть запуск программы, выполнение внутренней функции интерпретатора) или сообщает об ошибке, если программа или функция не найдены.

- По окончании работы пользователь завершает работу с интерпретатором, вводя команду *exit,* и выходит из системы.

7.3. Общий синтаксис скрипта

Любой из стандартных командных интерпретаторов имеет развитый язык программирования, позволяющий создавать командные файлы, или скрипты, для выполнения достаточно сложных задач. Следует, однако, иметь в виду, что shell является интерпретатором, он последовательно считывает команды из скрипта и выполняет их, как если бы они последовательно вводились пользователем с терминала. Эффективность скриптов определяется простотой и наглядностью. Если же производительность программы играет главную роль, то самым эффективным средством по-прежнему остается язык программирования C.

Скрипт представляет собой обычный текстовый файл, в котором записаны инструкции, понятные командному интерпретатору. Это могут быть команды, выражения shell или функции. Командный интерпретатор считывает эти инструкции из файла и последовательно выполняет их. Безусловно, как и в случае любого другого языка программирования, применение комментариев существенно облегчает последующее использование и модификацию написанной программы. В Bourne shell комментарии начинаются с символа '#':

 # Этот скрипт выполняет поиск "мусора" (забытых временных
 # файлов, файлов core и т . п .) в каталогах пользователей

Комментарии могут занимать не всю строку, а следовать после команды:

find /home -name core -print *# Выполним поиск файлов core*

Поскольку в системе могут существовать скрипты для различных интерпретаторов, имя интерпретирующей команды обычно помещается в первой строке следующим образом:

#!/bin/sh

В данном случае последующий текст скрипта будет интерпретироваться Bourne shell. Заметим, что при запуске скрипта из командной строки (для этого он должен обладать правом на выполнение — x), будет запущен новый командный интерпретатор, ввод команд для которого будет выполняться из файла скрипта.

В интерпретатор встроен мощный язык программирования, используемый для самых разных задач: от автоматизации повторяющихся команд до написания сложных интерактивных программ обработки данных. По соглашению в именах локальных переменных используются символы нижнего регистра, а в именах переменных среды – верхнего.

7.4. Форматы исполняемых файлов

В большинстве современных операционных систем UNIX используются два стандартных формата исполняемых файлов – COFF (Common Object File Format) и ELF (Executable and Linking Format).

Описание форматов исполняемых файлов необходимо для описания базовой функциональности ядра операционной системы. Информация, хранящаяся в исполняемых файлах форматах COFF и ELF, позволяет получить информацию для работы приложения и системы в целом: какие части программы необходимо загрузить в память; как создается область неинициализированных данных; где в памяти располагаются инструкции и данные программы; какие библиотеки необходимы для выполнения программы; как связан исполняемый файл на диске; образ программы в памяти и дисковая область свопинга.

Базовая структура памяти для процессов, загруженных из исполняемых файлов форматов COFF и ELF, содержит одни и те же основные компоненты (сегменты кода, данных, стека), хотя расположение сегментов различно. Независимо от формата исполняемого файла виртуальные адреса процесса не могут выходить за пределы 3 Гбайт.

Формат ELF имеет файлы нескольких типов. Стандарт ELF различает следующие типы:

- *Перемещаемый файл* (relocatable file), хранящий инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такого связывания может быть исполняемый файл или разделяемый объектный файл.

- *Разделяемый объектный файл* (shared object file) также содержит инструкции и данные, но может быть использован двумя способами. В первом случае он может быть связан с другими перемещаемыми файлами и разделяемыми объектными файлами, в результате чего будет создан

новый объектный файл. Во втором случае при запуске программы на выполнение операционная система может динамически связать его с исполняемым файлом программы, в результате чего будет создан исполняемый образ программы. В последнем случае речь идет о разделяемых библиотеках.

- *Исполняемый файл* хранит полное описание, позволяющее системе создать образ процесса. Он содержит инструкции, данные, описание необходимых разделяемых объектных файлов, а также необходимую символьную и отладочную информацию.

Заголовок ELF-файла имеет фиксированное расположение. Остальные компоненты размещаются в соответствии с информацией, хранящейся в заголовке. Таким образом, заголовок содержит общее описание структуры файла, расположение отдельных компонентов и их размеры. Поскольку заголовок ELF-файла определяет его структуру, в табл. 2 приведены более подробно поля заголовка.

Таблица 2

Поля заголовка ELF-файла

Поле	Описание
e_ident [ ]	Массив байт, каждый из которых определяет некоторую общую характеристику файла: формат файла (ELF), номер версии, архитектуру „ системы (32-разрядная или 64-разрядная) и т. д.
e_type	Тип файла
e_machine	Архитектура аппаратной платформы, для которой создан данный файл
e_version	Номер версии ELF-формата. Обычно определяется как EV_CURRENC (текущая), что означает последнюю версию
e_entry	Виртуальный адрес, по которому системой будет передано управление после загрузки программы (точка входа)
e_phoff	Расположение (смещение от начала файла) таблицы заголовков программы
e_shof f	Расположение таблицы заголовков секций
e_ehsize	Размер заголовка
e_phentsize	Размер каждого заголовка программы
e_phnum	Число заголовков программы
e_shentsize	Размер каждого заголовка сегмента (секции)
e_shnum	Число заголовков сегментов (секций)
e_shstrndx	Расположение сегмента, содержащего таблицу строк

Информация, содержащаяся в таблице заголовков программы, указывает ядру, как создать образ процесса из сегментов. Большинство сегментов копируются (отображаются) в память и представляют собой соответствующие сегменты процесса при его выполнении, например, сегменты кода или данных. Каждый заголовок сегмента программы описывает один сегмент и содержит следующую информацию: тип сегмента и действия операционной системы с данным сегментом; расположение сегмента в файле; стартовый адрес сегмента в виртуальной памяти процесса; размер сегмента в файле; размер сегмента в памяти; флаги доступа к сегменту (запись, чтение, выполнение).

Часть сегментов имеет тип LOAD, предписывающий ядру при запуске программы на выполнение создать соответствующие этим сегментам структуры данных, называемые *областями,* определяющие непрерывные участки виртуальной памяти процесса и связанные с ними атрибуты. Сегмент, расположение которого в ELF-файле указано в соответствующем заголовке программы, будет отображен в созданную область, виртуальный адрес начала которой также указан в заголовке программы. К сегментам такого типа относятся, например, сегменты, содержащие инструкции программы (код) и ее данные. Если размер сегмента меньше размера области, неиспользованное пространство может быть заполнено нулями. Такой механизм, в частности, используется при создании неинициализированных данных процесса (BSS).

В сегменте типа INTERP хранится программный интерпретатор. Данный тип сегмента используется для программ, которым необходимо динамическое связывание. Суть динамического связывания заключается в том, что отдельные компоненты исполняемого файла (разделяемые объектные файлы) подключаются не на этапе компиляции, а на этапе запуска программы на выполнение. Имя файла, являющегося *динамическим редактором связей,* хранится в данном сегменте. В процессе запуска программы на выполнение ядро создает образ процесса, используя указанный редактор связей. Таким образом, первоначально в память загружается не исходная программа, а динамический редактор связей. На следующем этапе динамический редактор связей совместно с ядром UNIX создают полный образ исполняемого файла. Динамический редактор загружает необходимые разделяемые объектные файлы, имена которых хранятся в отдельных сегментах исходного исполняемого файла, и производит требуемое размещение и связывание. В заключение управление передается исходной программе.

Завершает файл таблица заголовков *разделов* или *секций* (section). Разделы (секций) определяют разделы файла, используемые для связывания с другими модулями в процессе компиляции или при динамическом связывании. Соответственно, заголовки содержат всю необходимую информацию для описания этих разделов. Как правило, разделы содержат более детальную информацию о сегментах. Так, например, сегмент кода может состоять из нескольких разделов, таких как хэш-таблица для хранения индексов используемых в программе символов, раздел инициализационного кода программы,

таблица связывания, используемая динамическим редактором, а также раздел, содержащий собственно инструкции программы.

Следует обратить внимание на то, что в начале ELF-файла стоит сигнатура 'Ox7fELF' , по которой можно определить формат файла (массив байт, каждый из которых определяет некоторую общую характеристику файла: формат файла (ELF), номер версии, архитектуру системы (32-разрядная или 64-разрядная) и т. д.).

7.5. Система X Windows System (X11)

Система X Windows System – стандарт мобильной сетевой графической оконной пользовательской системы. X11 (или просто X) представляет собой клиент-серверную систему. Взаимодействие X-клиента и X-сервера основано на событиях. В каждый момент времени X-сервера посылает информацию о событиях какому-либо одному клиенту. Говорят, что этот клиент имеет *фокус ввода*.

X11 поддерживает такие возможности, как перекрывающиеся иерархические окна, текстовые и графические окна, текстовые и графические операции. Для запуска X11 необходимо ввести команду *startx*.

ПРИЛОЖЕНИЕ
Итоговый тест

1) Язык, на котором написана большая часть кода Linux/UNIX:

а) C б) Pascal
в) Ada г) Basic

2) Типы файлов в UNIX:

а) обычные файла, жесткие ссылки,LILO, каталоги
б) жесткие ссылки, файлы, семафоры, каталоги, сокеты
в) файлы, каталоги, FIFO, сокеты, ссылки, специальные файлы устройств
г) FIFO, сокеты, ссылки, специальные файлы устройств, сигналы, каталоги

3) Какой из дистрибутивов Linux производится в России?

а) Debian б) ASPLinux
в) Red Hat г) Slackware

4) Какая из этих операционных систем не является POSIX-совместимой?

а) IRIX б) BeOS
в) MacOS X г) OS/2

5) Какие права установлены для группы?

-rwxr---w- 1 root root 93 1 янв 2003 file.txt

а) Чтение и запись б) Чтение
в) Запись г) Чтение, запись, исполнение

6) Файловая система *Ext3* в отличие от файловой системы *Ext2*:

а) Журналируемая б) Виртуальная
в) Псевдо-файловая г) Сетевая

7) На разделы какой файловой системы Linux может записывать файлы?

а) ISO9660 б) NTFS
в) HPFS г) ReiserFS

8) В какие разделы файловой системы Linux не может записывать файлы?

а) Ext2 б) NTFS
в) msdos г) ReiserFS

9) Что можно сказать о файлах, метаданные которых приведены ниже:

/home/sergey	/home/ivan
12567 file1	12567 file2

а) символическая связь б) сокеты
в) жесткая связь г) FIFO

10) Сколько корневых каталогов может быть в UNIX?

а) 1 б) 2
в) 26 г) любое количество

11) Что содержит каталог */dev* в UNIX?

а) файлы пользователей б) файлы, необходимые для загрузки
в) файлы устройств г) конфигурационные файлы

12) В каком файле содержатся данные о монтируемых устройствах и точках монтирования?

а) /etc/fstab б) /etc/inittab
в) /etc/crontab г) /etc/motd

13) Атрибуты “темного каталога”:

а) rwx------ б) r--r--r--
в) --x--x--x г) rw-rw-rw-

14) Какой формат исполняемых файлов используется в UNIX?

а) PE б) ELF
в) MZ г) LE

15) Для кого установлено право на запись в файл?

-rwxr-x-r-- 1 root root 93 1 янв 2003 file.txt

а) для группы б) для владельца
в) для остальных г) для всех

16) команда *chmod u=4 file* установит атрибуты файла file:

а) Чтение для владельца б) Запись для владельца
в) Исполнение для группы г) Чтение и запись для владельца

17) Команда *chmod o-2 file* снимет атрибуты файла file:

а) Чтение для остальных б) Запись для владельца
в) Запись для остальных г) Исполнение для группы

18) Команда *ls* предназначена для вывода информации:

а) о текущем пользователе б) о файлах
в) о дисках г) о процессах

19) Опция команды *ls* для отображения файлов в длинном формате:

а) –a б) –l
в) –d г) –f

20) Команда *pwd* выводит на экран:

а) пароль текущего пользователя б) имя текущего каталога
в) имя текущего пользователя г) список подкаталогов текущего каталога

21) Какая команда является внутренней в интерпретаторе?

а) pwd б) cd
в) ls г) df

22) Команда для просмотра информации о смонтированных устройствах:

а) ps б) df
в) wc г) pwd

23) Какой процесс запрашивает имя пользователя при загрузке?

а) login б) getty
в) lpd г) cron

24) Для запуска процесса в фоновом режиме в конце командной строки необходимо поставить:

а) # б) &
в) $ г) ~

25) Создание неименованного канала между процессами происходит при помощи символа:

а) & б) >
в) | г) ~

26) Выполнение нижеприведенной команды позволяет:
$ cat file1 | wc

а) создать неименованный канал
б) перенаправить ввод
в) выполнить последовательный запуск программ
г) запустить задание в фоновом режиме

27) Для создания учетной записи пользователя user с неявным указанием пароля необходимо ввести команду:

а) adduser -d /home/user -p1q2w3e user
б) useradd -d /home/user -p1q2w3e user
в) adduser /user user
г) adduser -d /home/user user

28) К стандартным пользователям UNIX относятся:

а) root, adm, bin, user б) root, adm, bin, 1p, lpd , news, nobody, uucp
в) 1p, lpd , news, nobody, cron г) root, adm, uucp, staff , bin, cron

29) Файлы системы печати электронной почты находятся в подкаталогах:

а) /usr/spool б) /user/bin
в) /boot г) /var

30) Системные вызовы, для выполнения которых требуется трансляция имени файла:

а) exec,chown, chgrp, chmod, statfs, mkdir б) chmod, statfs, root, open, lp
в) pipe, exec,chown, chgrp, fcntl г) read, write, open, exec, readv

31) Атрибутами процесса являются:

а) PID, UID, GID, PPID б) PID, PPID, NN, TTY, EUID, EGID
в) UID, GID, EUID, EGID г) PID, UID, EUID

32) К форматам исполняемых файлов UNIX относятся:

а) coff, elf б) out, exe
в) exe, elf , com г) все файлы каталога /bin

33) К компонентам FFS относятся:

а) superblock, ilist, блоки хранения данных, массив свободных блоков и inode
б) группа цилиндров: superblock, ilist, блоки хранения данных, массив свободных блоков и inode
в) группа цилиндров: superblock, ilist, массив свободных блоков и inode
г) карта свободных блоков, inode, группа цилиндров

34) К компонентам s5fs относятся:

а) superblock, ilist, блоки хранения данных б) superblock, ilist, vnode
в) superblock, ilist, inode г) vnode, блоки хранения данных

35) Событие, которое не может вызвать операцию ввода/вывода для блочного устройства:
а) чтение или запись в обычный файл;
б) страничное замещение или свопинг;
в) чтение или запись в специальный файл устройства;
г) запуск процедуры spec_read() или d_read();
д) операции подсистемы управления памятью.

36) Для подавления вывода ошибок необходимо ввести команду:

а) run 2>/dev/null б) run 2>/dev/null 2>&1
в) run 2>/dev/null 3>&2 г) run 3>/dev/null