Архитектура и реализация операционной системы 4.4BSD

Содержание

Chapter 2.1. Обзор архитектуры 4.4BSD

2.1.1. Подсистемы и ядро 4.4BSD

Ядро 4.4BSD предоставляет четыре базовых подсистемы: процессы, файловую систему, коммуникации и запуск системы. Этот раздел перечисляет, в каком месте этой книги описана каждая из этих подсистем.

1. Процессы образуют поток управления в адресном пространстве. Механизмы создания, завершения и другие управляющие процессы описаны в Главе 4. Для каждого процесса система мультиплексирует отдельное виртуальное адресное пространство; такое управление памятью обсуждается в Главе 5.
2. Механизм доступа пользователя к файловой системе и устройствам один и тот же; общие аспекты обсуждаются в Главе 6. Файловая система является набором именованных файлов, организованных в древовидную иерархию каталогов, а операции по управлению ими представлены в Главе 7. Файлы располагаются на таких физических носителях, как диски. 4.4BSD поддерживает несколько типов организации данных на диске, как описано далее в Главе 8. Доступ к файлам на удаленных машинах является предметом обсуждения в Главе 9. Для доступа к системе Терминалы используются терминалы; их функционированию посвящена глава 10.
3. Механизмы коммуникаций, предоставляемые традиционными UNIX-системами, включают однонаправленные потоки байтов между связанными процессами (смотрите материал о конвейерах в Разделе 11.1) и извещение об исключительных событиях (смотрите материал о сигналах в Разделе 4.7). В 4.4BSD имеется также механизм межпроцессного взаимодействия между процессами. Этот механизм, описываемый в Главе 11, использует способы доступа, отличающиеся от тех, что используются в файловой системе, но, как только соединение установлено, процесс может работать с ним, как будто это конвейер. Имеется и механизм работы с сетью, описываемый в Главе 12, который обычно используется как слой ниже механизма IPC. В Главе 13 даётся детальное описание конкретной реализации механизма работы с сетью.
4. В любой операционной системе присутствуют вопросы управления, такие, как её запуск. Запуск и вопросы управления обсуждаются в Главе 14.

Разделы с 2.3 по 2.14 представляют собой вводный материал, относящийся к главам с 3 по 14. Мы определим понятия, коснемся основных системных вызовов и рассмотрим исторические разработки. Наконец, мы расскажем о причинах многих ключевых архитектурных решений.

2.1.1.1. Ядро

Ядро является частью системы, которая работает в защищенном режиме и управляет доступом всех пользовательских программ к низкоуровнему аппаратному обеспечению (к примеру, ЦПУ, дискам, терминалам, сетевым связям) и программным компонентам (к примеру, файловой системе, сетевым протоколам). Ядро предоставляет основные подсистемы; оно создает процессы и управляет ими, предоставляет функции для доступа к файловой системе и службам связи. Такие функции, называемые *системными вызовами*,

доступны процессам пользователей в виде библиотечных подпрограмм. Эти системные вызовы являются единственным способом доступа к этим подсистемам. Подробно механизм работы системных вызовов даётся в Главе 3, вместе с описанием некоторых механизмов ядра, работа которых не является прямым результатом процесса, выполняющего системный вызов.

Ядро, по традиционной терминологии операционных систем, является маленьким куском программного обеспечения, которое предоставляет только минимальный набор подсистем, необходимый для реализации дополнительных служб операционной системы. В современных исследовательских операционных системах — таких как Chorus [Rozier et al], Mach [Accetta et al], Tunis [Ewens et al], и V Kernel [Cheriton] - такое разделение функциональности выполнено не только логически. Такие службы, как файловые системы и сетевые протоколы, выполнены в виде прикладных процессов клиентов ядра или микроядра.

Ядро 4.4BSD не разбивается на несколько процессов. Это основополагающее архитектурное решение было сделано в самых ранних версиях UNIX. В первых двух реализациях Кена Томпсона (Ken Thompson) не было отображаемой памяти, и поэтому не было аппаратного различия между адресным пространством пользователя и ядра [Ritchie]. Могла бы быть придумана система обмена сообщениями как реально реализуемая модель процессов ядра и пользователя. Для простоты и увеличения производительности было выбрано монолитное ядро. К тому же ранние ядра были маленькими; включение таких подсистем, как сетевые коммуникации, в ядро увеличило его размер. Современные тенденции в области операционных систем сводятся к уменьшению размера ядра за счет перевода таких служб в пользовательское адресное пространство.

Пользователи обычно общаются с системой через интерпретатор языка команд, называемый оболочкой (*shell*), и, может быть, через дополнительные прикладные пользовательские программы. Такие программы и оболочка реализованы в виде процессов. Подробное описание таких программ выходит за рамки этой книги, которая практически полностью посвящена работе ядра.

В разделах 2.3 и 2.4 описываются сервисы, предоставляемые ядром 4.4BSD, и даётся обзор их архитектуры. Последующие главы описывают подробности архитектуры и реализации этих сервисов в 4.4BSD.

2.1.2. Организация ядра

В этом разделе мы рассматриваем организацию ядра 4.4BSD с двух точек зрения:

1. Как статический блок программного обеспечения, категоризуемый по функциональности модулей, составляющих ядро
2. В его динамике, категоризуемой по услугам, предоставляемым пользователям

Самая большая часть ядра реализует системные услуги, к которым приложения обращаются через системные вызовы. В 4.4BSD это программное обеспечение организуется по следующим принципам:

- Базовые подсистемы ядра: обработка таймеров и системного таймера, управление

дескрипторами и процессами

- Поддержка управления памятью: подкачка и выгрузка
- Общесистемные интерфейсы: ввод/вывод, управление и мультиплексирование операций, выполняемых над дескрипторами
- Файловая система: файлы, каталоги, преобразование маршрутов, блокировка файлов и управление буфером ввода/вывода
- Поддержка работы с терминалами: драйвер терминального интерфейса и режимы работы терминального канала
- Подсистемы межпроцессного взаимодействия: сокеты
- Поддержка сетевых коммуникаций: коммуникационные протоколы и общесетевые подсистемы, такие, как маршрутизация

Таблица 1. Машинонезависимое программное обеспечение в ядре 4.4BSD

Категория	**Строк кода**	**Процент от строк ядра**
заголовки	9,393	4.6
инициализация	1,107	0.6
подсистемы ядра	8,793	4.4
универсальные интерфейсы	4,782	2.4
межпроцессное взаимодействие	4,540	2.2
работа с терминалом	3,911	1.9
виртуальная память	11,813	5.8
управление vnode	7,954	3.9
имена в файловой системы	6,550	3.2
быстрое хранилище файлов	4,365	2.2
логическая структура файлового хранилища	4,337	2.1
хранилище файлов в памяти	645	0.3
cd9660 файловая система	4,177	2.1
различные файловые системы (10)	12,695	6.3
сетевая файловая система	17,199	8.5
сетевое взаимодействие	8,630	4.3
интернет-протоколы	11,984	5.9
протоколы ISO	23,924	11.8
протоколы X.25	10,626	5.3
протоколы XNS	5,192	2.6

Большая часть программного обеспечения в этих категориях является машинонезависимой и переносима между различными аппаратными архитектурами.

Машинозависимые аспекты ядра отделены от основного кода. В частности, ни в одной части машинонезависимого кода не содержится кода, зависимого от конкретной архитектуры. Когда требуется произвести действия, зависимые от архитектуры, машинонезависимый код вызывает функцию, зависимую от архитектуры машины, которая находится в машинозависимой части кода. Машинозависимое программное обеспечение включает в себя

- Низкоуровневые действия по запуску системы
- Обработка исключительных ситуаций и прерываний
- Низкоуровневые манипуляции процессом во время работы
- Конфигурация и инициализация аппаратных устройств
- Поддержка устройств ввода/вывода во время работы

Таблица 2. Машинозависимое программное обеспечение для HP300 в ядре 4.4BSD

Категория	**Строк кода**	**Процент от строк ядра**
заголовочные файлы, зависимые от машины	1,562	0.8
заголовочные файлы драйверов устройств	3,495	1.7
исходный код драйверов устройств	17,506	8.7
виртуальная память	3,087	1.5
код, зависящий от других машин	6,287	3.1
подпрограммы на языке ассемблера	3,014	1.5
совместимость с HP/UX	4,683	2.3

Машинонезависимое программное обеспечение в ядре 4.4BSD показывает статистику машинонезависимого кода, который составляет ядро 4.4BSD для HP300. Числа во второй колонке обозначают количество строк исходного кода на языке C, заголовочных файлов и ассемблерного кода. Практически весь код ядра написан на языке программирования C; менее двух процентов написано на языке ассемблера. Как показывает статистика в Машинозависимое программное обеспечение для HP300 в ядре 4.4BSD, машинозависимый код, не включающий поддержку HP/UX и устройств, составляет менее 6.9 процента ядра.

Лишь малая часть ядра отвечает за инициализацию системы. Этот код используется при *начальной загрузке* системы для перехода в рабочий режим и отвечает за настройку аппаратного и программного окружения ядра (обратитесь к Главе 14). Некоторые операционные системы (особенно те, что ограничены объёмом физической памяти) выполняют действия по выгрузке или *перекрытию* программного кода, выполняющего эти

функции, после окончания его работы. Ядро 4.4BSD не работает повторно с памятью, использованной начальным кодом, потому что этот объём памяти составляет менее 0.5 процентов ресурсов ядра, используемых на типичной машине. Также начальный код не находится только в одном месте ядра - он рассредоточен везде, и обычно появляется там, где логически связан с объектом инициализации.

2.1.3. Службы ядра

Разграничение между кодом уровней ядра и пользователя обеспечивается аппаратными методами, предоставляемыми оборудованием. Ядро работает в отдельном адресном пространстве, которое недоступно процессам пользователя. Привилегированные операции - такие, как осуществление ввода/вывода и остановка модуля центрального процессора (CPU) - доступны только ядру. Приложения делают запросы ядру на доступ к его сервисам при помощи *системных вызовов*. Системные вызовы используются для указания ядру на выполнение как сложных операций, таких как запись данных во вторичный носитель, так и простых, таких как получение текущего времени. Все системные вызовы выполняются *синхронно* с приложением: Приложение не будет продолжать работу, пока ядро не выполнит действия, соответствующие системному вызову. Ядро может завершить некоторые операции, связанные с системным вызовом, после его окончания. Например, системный вызов *write* будет копировать записываемые данные от пользовательского процесса в буфер ядра, пока процесс находится в ожидании, но, как правило, будет немедленно завершаться до того, как буфер ядра реально будет записан на диск.

Системный вызов обычно реализуется как аппаратное прерывание, которое изменяет режим работы CPU и текущее отображение адресного пространства. Параметры, передаваемые пользователями системным вызовам, перед использованием проверяются ядром. Такая проверка обеспечивает целостность системы. Все параметры, передаваемые в ядро, копируются в адресное пространство ядра, для того, чтобы проверенные параметры не могли быть изменены в результате побочного действия системного вызова. Результаты выполнения системного вызова возвращаются ядром либо в аппаратных регистрах, либо копированием их значений в области памяти, указанные пользователем. Как и параметры, переданные в ядро, адреса, используемые для возвращения результатов, должны быть проверены на то, что они являются частью адресного пространства приложения. Если при обработке системного вызова ядром возникает ошибка, код ошибки возвращается пользователю. В случае языка программирования C код этой ошибки сохраняется в глобальной переменной *errno*, а функция, соответствующая системному вызову, возвращает в качестве результата значение -1.

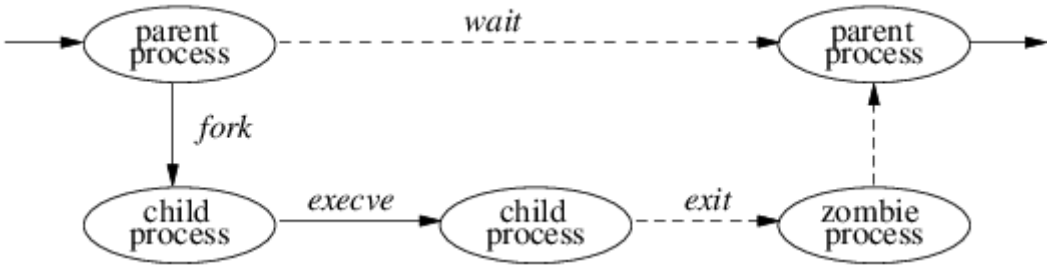
Пользовательские приложения и ядро работают независимо друг от друга. 4.4BSD не хранит управляющие блоки ввода/вывода и другие связанные с операционной системой структуры данных в адресном пространстве приложения. Каждому пользовательскому приложению предоставляется независимое адресное пространство, в котором оно и выполняется. Ядро выполняет большинство управляющих действий, таких как приостановка процесса на время выполнения другого, незаметно для участвующих процессов.

2.1.4. Управление процессами

4.4BSD поддерживает многозадачность. Каждая задача или выполняющийся поток называется *процессом*. *Контекст* процесса 4.4BSD состоит из состояния пользовательского уровня, включая содержимое его адресного пространства и окружения времени выполнения, и состояния уровня ядра, в который включаются параметры планировщика задач, управляющие ресурсы и идентифицирующая информация. В контекст включается все, что используется ядром при предоставлении своих сервисов процессу. Пользователи могут создавать процессы, управлять их выполнением и получать уведомления при изменении состояния выполнения процессов. Каждому процессу назначается уникальное число, называемое *идентификатором процесса* (PID). Это число используется ядром для идентификации процесса при сообщении пользователю об изменении его состояния, и пользователем для указания процесса в системном вызове.

Ядро создает процесс, дублируя контекст другого процесса. Новый процесс считается *порожденным процессом* исходного *родительского процесса*. Контекст, копируемый в ходе создания процесса, включает как состояние выполнения процесса уровня пользователя, так и системное состояние процесса, управляемое ядром. Важные компоненты состояния ядра описаны в Главе 4.

Жизненный цикл процесса



Жизненный цикл процесса изображен на рисунке Жизненный цикл процесса. Процесс может создать новый процесс, который является копией исходного процесса с помощью системного вызова *fork*. Возврат из вызова *fork* происходит два раза: один раз в родительском процессе, в котором возвращаемое значение является идентификатором порожденного процесса, и второй раз в порожденном процессе, в котором возвращаемое значение равно 0. Связь родитель-потомок порождает иерархическую структуру процессов в системе. Новый процесс имеет доступ ко всем ресурсам его родителя, таким как файловые дескрипторы, состояние обработки сигналов и распределение памяти.

Хотя есть ситуации, когда процесс должен быть копией своего родителя, наиболее типичным и полезным действием является загрузка и выполнение другой программы. Процесс может заместить себя образом памяти другой программы, передавая вновь созданному образу набор параметров, при помощи системного вызова *execve*. Одним из параметров является имя файла, содержимое которого имеет формате, распознаваемый системой - это либо двоичный выполняемый файл, либо файл, который приводит к запуску указанной программы интерпретации для обработки его содержимого.

Процесс может завершить работу, выполнив системный вызов *exit*, посылающий 8-битовое значение состояния завершения своему родителю. Если процесс хочет передать родительскому процессу информацию, превышающую один байт, он должен либо создать

канал межпроцессных коммуникаций при помощи конвейеров или сокетов, или при помощи промежуточного файла. Коммуникации между процессами подробно обсуждаются в Главе 11.

Процесс может приостановить выполнение до тех пор, пока не завершит работу любой из порожденных им процессов, при помощи системного вызова *wait*, который возвращает PID и статус завершения выполненного дочернего процесса. Родительский процесс может быть настроен на получение сигнала в случае, когда порожденный процесс завершает работу или аварийно прекращает выполнение. При помощи системного вызова *wait4* родитель может получить информацию о событии, приведшем к завершению порожденного процесса и о ресурсах, использованных процессом за время его работы. Если процесс становится сиротой из-за того, что процесс, его породивший, завершил работу до окончания работы потомка, то ядро перенаправляет состояние завершения порожденного процесса особому системному процессу *init*: обратитесь к разделам 3.1 и 14.6).

Подробное описание того, как ядро создает и уничтожает процессы, даётся в Главе 5.

Планирование выполнения процессов осуществляется согласно параметру *приоритетности процесса*. Этот приоритет управляется алгоритмом планирования задач в ядре. Пользователи могут влиять на выполнение процесса, задавая этот параметр (*nice*), который влияет на суммарный приоритет, но ограничен использованием ресурсов CPU согласно алгоритму планировщика задач ядра.

2.1.4.1. Сигналы

В системе определен набор *сигналов*, которые могут быть отправлены процессу. Сигналы в 4.4BSD сделаны по образу аппаратных прерываний. Процесс может определить пользовательскую подпрограмму, которая будет являться *обработчиком*, и которой должен будет перенаправляться сигнал. Когда сигнал генерируется, он блокируется от повторного появления до тех пор, пока не будет *перехвачен* обработчиком. Перехват сигнала включает в себя сохранение контекста текущего процесса и построение нового, в котором запускается обработчик. Затем сигнал направляется обработчику, который может либо прервать процесс, либо передать управление обратно выполняемому процессу (может быть, после установки значения глобальной переменной). Если обработчик возвратил управление, сигнал разблокировывается и может быть сгенерирован (и получен) снова.

Либо процесс может определить, что сигнал будет *игнорироваться* или будет выполняться действие по умолчанию, определяемое ядром. Действием по умолчанию для некоторых сигналов является прекращение процесса. Это завершение работы может сопровождаться созданием *файла дампа*, содержащего текущий образ памяти процесса для использования в последующей отладке.

Некоторые сигналы не могут быть перехвачены или проигнорированы. К таким сигналам относятся *SIGKILL*, прерывающий неуправляемый процесс, и сигнал управления заданиями *SIGSTOP*.

Процесс может выбрать получение сигналов в специальный стек для выполнения хитроумных программных манипуляций стеком. Например, подпрограммам поддержки языка нужно иметь стек для каждой подпрограммы. Система времени выполнения языка

может выделять эти стеки, разделяя единственный стек, предоставляемый в 4.4BSD. Если ядро не поддерживает отдельный стек сигналов, то пространство, выделяемое каждой подпрограмме, должно быть расширено на объём, требуемый для перехвата сигнала.

Все сигналы имеют один и тот же *приоритет*. Если обработки ожидают несколько сигналов, то порядок их направления процессу зависит от реализации. Обработчики сигналов, выполняемые по сигналу, который их вызвал, блокируются, но при этом могут быть сгенерированы дополнительные сигналы. Имеется механизм, позволяющий защитить критический участок кода от появления заданных сигналов.

Подробное описание архитектуры и реализации механизма сигналов даётся в Разделе 4.7.

2.1.4.2. Группы управления и сеансы

Процессы организованы в *группы управления*. Группы управления используются для управления доступом к терминалам и для обеспечения передачи сигналов наборам связанных процессов. Процесс наследует группу управления от своего родительского процесса. Ядром обеспечиваются механизмы, позволяющие процессу изменять свою группу управления или группу управления своих наследников. Создание новой группы управления просто; значение, соответствующее новой группе управления, обычно является идентификатором создающего её процесса.

Группу процессов в группе управления иногда называют *заданием* и оно управляется высокоуровневым системным программным обеспечением, таким как командный процессор. Типичным примером задания, созданного командным процессором, является *конвейер* из нескольких связанных процессов, так что выходной поток первого процесса является входным потоком для второго, выходной поток второго процесса является входным потоком для третьего, и так далее. Командный процессор создает такое задание, порождая процесс для каждого участка конвейера, а затем помещая все эти процессы в отдельную группу обработки.

Пользовательский процесс может послать сигнал как всем процессам в группе управления, так и конкретному процессу. Процесс в заданной группе управления может получать программные прерывания, отражающиеся на группе, приводящие к приостановке или продолжению выполнения, или к прерыванию или завершению работы.

Терминалу ставится в соответствие идентификатор группы управления. Этот идентификатор обычно равен идентификатору группы управления, соответствующей терминалу. Управляющий заданиями командный процессор может создать несколько групп управления, связанных с одним и тем же терминалом; терминал является *управляющим терминалом* для каждого процесса в этих группах. Процесс может выполнять чтение из дескриптора своего управляющего терминала, если только идентификатор группы управления соответствует идентификатору группы этого процесса. Если идентификаторы не совпадают, процесс будет блокирован при попытке чтения с терминала. Изменяя идентификатор группы управления терминала, командный процессор может распределять терминал между несколькими различными заданиями. Такое распределение называется *управлением заданиями* и описывается вместе с группами управления в Разделе 4.8.

Так же, как и наборы связанных процессов могут объединяться в группы управления, набор групп управления может быть объединен в *сеанс*. Основное назначение сеансов заключается создании изолированного окружения для процесса-даемона и порожденных им процессов, а также для объединения начального командного процессора пользователя и заданий, которые он порождает.

2.1.5. Управление памятью

Каждый процесс имеет собственное адресное пространство. Адресное пространство изначально разделяется на три логических сегмента: *код*, *данные* и *стек*. Сегмент кода доступен только для чтения и содержит машинные коды программы. Сегменты данных и стека оба доступны как для чтения, так и для записи. Сегмент данных содержит как инициализированные, так и неинициализированные области данных программы, когда как стековый сегмент представляет собой стек программы на этапе выполнения. На большинстве машин сегмент стека автоматически расширяется ядром в процессе работы программы. Процесс может расширять или уменьшать свой сегмент данных, выполняя системный вызов, когда как размер сегмента кода процесс может изменить только когда содержимое сегмента перекрывается данными файловой системы или в процессе отладки. Начальное содержимое сегментов порожденного процесса копируется из сегментов родительского процесса.

Для выполнения процесса вовсе не обязательно постоянно хранить в памяти полное содержимое его адресного пространства. Если процесс обращается к области адресного пространства, которая не присутствует в оперативной памяти, то система *подгружает страницу* с необходимой информацией в память. Когда возникает нехватка системных ресурсов, то система использует двухуровневый подход к управлению имеющимися ресурсами. Если не хватает памяти, то система будет забирать ресурсы памяти от процессов, если они давно не использовались. Если ресурсов не хватает очень сильно, то система будет прибегать к *выгрузке* всего контекста процесса во вторичную подсистему хранения данных. *Постраничная подгрузка по требованию* и *выгрузка* выполняются системой абсолютно незаметно для процессов. Процесс может, однако, указать системе объём памяти, который будет использоваться, в качестве помощи.

2.1.5.1. Решения BSD по архитектуре управления памятью

В 4.2BSD требовалось реализовать поддержку больших несвязанных адресных пространств, отображаемых в память файлов и совместно используемой памяти. Был спроектирован интерфейс, который назвали *mmap*, позволяющий несвязанным процессам запрашивать отображение в их адресное пространство файла в режиме совместного использования. Если несколько процессов отображают в свое адресное пространство один и тот же файл, то изменение адресного пространства процесса, соответствующего файлу, в одном процессе, будет отображено в области отображения этого файла в другом процессе, а также и в самом файле. Однако в конце концов 4.2BSD была выпущена без интерфейса *mmap* из-за необходимости сделать в первую очередь другие возможности, такие, как работа с сетью.

Затем разработка интерфейса *mmap* продолжалась во время работы над 4.3BSD. Более 40 компаний и исследовательских групп принимали участие в обсуждениях, которые привели к появлению обновленной концепции, описанной в Berkeley Software Architecture Manual

[McKusick et al]. Несколько компаний реализовали этот обновленный интерфейс [Gingell et al].

И снова сроки разработки не позволили включить в 4.3BSD реализацию этого интерфейса. Хотя позже она могла быть встроена в имеющуюся подсистему виртуальной памяти 4.3BSD, разработчики решили не включать её сюда. потому что этой реализации было уже более 10 лет. Более того, оригинальная архитектура виртуальной памяти была основана на предположении, что компьютерная память мала и дорога, а диски подключены непосредственно к компьютеру, быстры и дешевы. Поэтому подсистема виртуальной памяти была разработана с упором на бережное использование памяти ценой более частых обращений к диску. Вдобавок реализация в 4.3BSD была пронизана зависимостями от аппаратной системы управления памятью машин VAX, что препятствовало её переносу на другие аппаратные платформы. И наконец, подсистема виртуальной памяти не была предназначена для поддержки связных многопроцессорных систем, которые сейчас становятся все более распространёнными и необходимыми.

Попытки постепенно усовершенствовать старую реализацию заведомо были обречены на неудачу. Полностью новая архитектура, с другой стороны, могла бы использовать большие объёмы памяти, уменьшить дисковые операции и обеспечивать работу с несколькими процессорами. Наконец, система виртуальной памяти в 4.4BSD была полностью изменена. Система виртуальной памяти 4.4BSD основана на системе виртуальной памяти (VM) [Tevanian] с заимствованиями из Mach 2.5 и Mach 3.0. В ней была эффективная поддержка совместного использования, полное разделение машинозависимой и машинонезависимой частей, а также (сейчас не используемая) поддержка работы с несколькими процессорами. Процессы могут отображать файлы в любую область своего адресного пространства. Они могут совместно использовать части своих адресных пространств посредством отображения в память одного и того же файла. Изменения, сделанные одним процессом, видны в адресном пространстве другого процесса, а также записываются и в сам файл. Процессы могут также запрашивать эксклюзивное отображение файла в память, при котором любые изменения, сделанные процессом, не видны другим процессам, которые отображают файл в память и не записываются обратно в файл.

Еще одной проблемой с системой виртуальной памяти является способ, которым информация передаётся ядру при выполнении системного вызова. 4.4BSD всегда копирует данные из адресного пространства процесса в буфер ядра. Для операций чтения и записи, при которых передаются большие объёмы данных, выполнение копирования может оказаться занимающим время процессом. Альтернативным способом является манипуляции с адресным пространством процесса в ядре. Ядро 4.4BSD всегда копирует данные о нескольким причинам:

- Зачастую пользовательские данные не выравнены по границе страницы памяти и их объём не кратен размеру аппаратной страницы памяти.
- Если страница памяти забирается от процесса, он не может больше ссылаться на эту страницу. Некоторые программы зависят от данных, остающихся в буфере, даже после записи этих данных.
- Если процесс позволяет хранить копию страницы памяти (как это делается в существующей 4.4BSD), то страница должна иметь атрибут *копирования-при-записи*. Такая страница является одной из таковых, что защищается от записи при помощи

атрибута только-для-чтения. Если процесс пытается модифицировать страницу памяти, в ядре возникает ситуация ошибки записи. После этого ядро делает копию страницы, которую процесс может изменять. К несчастью, большинство процессов будет немедленно пытаться записать новые данные в свой буфер вывода, что приводит в любом случае к копированию данных.

- Когда страницы переносятся в новые адреса виртуальной памяти, большинство аппаратных менеджеров памяти требуют, чтобы кэш аппаратного переназначения адресов был выборочно очищен. Очистка кэша зачастую выполняется медленно. В итоге получается, что переназначение адресов оказывается медленнее, чем копирование блоков данных, не превышающих 4 или 8 килобайт.

Больше всего отображение памяти нужно для работы к большими файлами и передачи больших объёмов данных между процессами. Интерфейс *mmap* даёт методы для выполнения обеих этих операций без копирования.

2.1.5.2. Управление памятью внутри ядра

Ядро часто выполняет выделение памяти, которое нужно только для выполнения единственного системного вызова. В пользовательском процессе такая кратковременно используемая память будет выделяться в стеке во время выполнения. Так как ядро имеет ограниченный объём стека времени выполнения, то неэффективно выделять в нём даже блоки памяти среднего размера. Таким образом, такая память должна выделяться посредством более гибкого механизма. Например, когда системный вызов должен преобразовать имя каталога, он должен выделить буфер размером 1 Кбайт для хранения имени. Другие блоки памяти должны выделяться на более продолжительный срок, чем один системный вызов, и поэтому не могут выделяться в стеке, даже если там есть место. В качестве примера можно взять блоки управления протоколами, которые существуют на всем протяжении сетевого соединения.

Необходимость в динамическом выделении памяти в ядре становилась все более острой вместе с добавлением количества сервисов. Общий механизм выделения памяти уменьшает сложность написания кода в ядре. Поэтому в 4.4BSD ядро имеет единый механизм выделения памяти, который может использоваться в любой части системы. У него есть интерфейс, похожий на функции библиотеки языка C *malloc* и *free*, которые обеспечивают выделение памяти в прикладных программах [McKusick & Karels]. Как интерфейс библиотеки языка C, функция выделения памяти получает параметр, указывающий на размер памяти, который необходим. Диапазон запрашиваемых объёмов выделяемой памяти не ограничен; однако выделяемая физическая память не подвергается постраничной подгрузке. Функции освобождения памяти передаётся указатель на освобождаемый участок памяти, но указывать размер освобождаемого участка памяти не нужно.

2.1.6. Система ввода/вывода

Базовой моделью системы ввода/вывода UNIX является последовательность байт, доступ к которым может осуществляться как последовательно, так и в в произвольном порядке. В типичном пользовательском процессе UNIX нет таких понятий, как *методы доступа* или

управляющие блоки.

Различные программы используют разнообразные структуры данных, но ядро не связывает ввод/вывод с используемыми структурами. Например, текстовым файлом считается файл из строк символов набора ASCII, которые разделены одним символом новой строки (символ ASCII перевода строки), но ядро не знает ничего об этом соглашении. Для удовлетворения потребностей большинства программ модель ещё более упрощена и сводится к потоку байт данных, или *потоку ввода/вывода*. Такое единое представление данных позволяет работать характерному для UNIX подходу на основе инструментов [Kernighan & Pike]. Поток ввода/вывода одной программы может быть подан в качестве входной информации практически любой другой программе. (Этот тип традиционных для UNIX потоков ввода/выводы не нужно путать с потоковой системой ввода/вывода из Eighth Edition или с потоками из System V, Release 3 (STREAMS), оба из которых доступны как обычные потоки ввода/вывода.)

2.1.6.1. Дескрипторы и ввод/вывод

Процессы UNIX для работы с потоками ввода/вывода используют *дескрипторы*. Дескрипторы представляют собой беззнаковые целые числа, получаемые после выполнения системных вызовов *open* и *socket*. Системный вызов *open* получает в качестве аргументов имя файла и режим доступа, который определяет, должен ли файл открываться для чтения, для записи или для обеих операций. Этот системный вызов может также использоваться для создания нового пустого файла. Системные вызовы *read* и *write* могут применяться к дескриптору для переноса данных. Системный вызов *close* может использоваться для уничтожения любого дескриптора.

Дескрипторы представляют низкоуровневые объекты, поддерживаемые ядром, и создаваемые системными вызовами, специфичными для каждого типа объектов. В 4.4BSD дескрипторы могут представлять три типа таких объектов: файлы, каналы и сокеты.

- *Файл* представляет собой линейную последовательность байт, имеющую по крайней мере одно имя. Файл существует, пока все его имена не удалены и ни один из процессов не хранит его дескриптор. Процесс получает дескриптор файла, открывая имя файла посредством системного вызова *open*. Работа с устройствами ввода/вывода осуществляется как с файлами.
- *Каналом* является линейная последовательность байт, такая же, как файл, но используемая исключительно как поток ввода/вывода, причем однонаправленный. У канала нет имени, и поэтому он не может быть открыт при помощи *open*. Вместо этого он создается посредством системного вызова *pipe*, который возвращает два дескриптора, один из которых принимает входные данные, без искажений, без повторений и в той же самой последовательности посылаемый на другой дескриптор. Система также поддерживает именованный канал, или FIFO. FIFO имеет те же самые свойства, что и канал, за исключением того, что он располагается в файловой системе; поэтому он может быть открыт системным вызовом *open*. Процессы, которые хотят обмениваться данными, открывают FIFO: Один процесс открывает его для чтения, а другой для записи.
- *Сокет* является промежуточным объектом, который используется для межпроцессных коммуникаций; он существует, пока какой-либо процесс хранит дескриптор, ссылающийся на него. Сокет создается системным вызовом *socket*, который возвращает

его дескриптор. Имеется несколько типов сокетов, которые поддерживают различные коммуникационные возможности, такие, как надёжную доставку данных, сохранение последовательности передаваемых сообщений, и сохранение границ сообщений.

В системах, предшествующих 4.2BSD, каналы были реализованы в файловой системе, когда в 4.2BSD появились сокеты, то каналы были повторно реализованы как сокеты.

Для каждого процесса ядро хранит *таблицу дескрипторов*, которая является таблицей, используемой ядром для преобразования внешнего представления дескриптора в его внутреннее представление. (Дескриптор является просто индексом в этой таблице.) Таблица дескрипторов процесса наследуется от родительского процесса, и вместе с ней наследуется и доступ к объектам, на которые ссылаются дескрипторы. Основными способами, при помощи которых процесс может получить дескриптор, является открытие или создание объекта, а также наследование от родительского процесса. Кроме того, межпроцессные коммуникации при помощи сокетов позволяют передавать дескрипторы в сообщениях между несвязанными процессами на одной и той же машине.

Любой рабочий дескриптор имеет связанное с ним *смещение в файле* в байтах от начала объекта. Операции чтения и записи начинаются от этого смещения, который обновляется после каждой передачи данных. Для объектов, к которым разрешен произвольный доступ, смещение в файле может быть установлено посредством системного вызова *lseek*. Обычные файлы, а также некоторые устройства, разрешают произвольный доступ к ним. Каналы и сокеты этого делать не позволяют.

Когда процесс завершается, ядро освобождает все дескрипторы, которые использовались этим процессом. Если процесс хранил последнюю ссылку на объект, то менеджер объектов уведомляется для выполнения всех необходимых действий, таких как окончательное удаление файла или уничтожение сокета.

2.1.6.2. Управление дескрипторами

Большинство процессов ожидают, что перед началом их работы уже будут открыты три дескриптора. Это дескрипторы 0, 1 и 2, больше известные как *стандартный ввод*, *стандартный вывод* и *стандартный поток диагностических сообщений*, соответственно. Как правило, все они связываются с пользовательским терминалом по время входа в систему (смотри Раздел 14.6) и наследуются через вызовы *fork* и *exec* процессами, запускаемыми пользователем. Таким образом, программа может считывать то, что набирает пользователь, из стандартного ввода, и программа может выдавать результат на экран пользователя, осуществляя запись в стандартный вывод. Дескриптор потока диагностических сообщений также открыт для записи и используется для вывода ошибок, когда как стандартный вывод используется для обычного вывода.

Эти (и другие) дескрипторы могут отображаться на объекты, отличающиеся от терминала; такое отображение называется *перенаправлением ввода/вывода*, и все стандартные командные процессоры позволяют пользователю это делать. Оболочка может направить вывод программы в файл, закрывая дескриптор 1 (стандартный вывод) и открывая выбранный выходной файл для создания нового дескриптора 1. Подобным же образом стандартный ввод может браться из файла, при этом закрывается дескриптор 0 и открывается файл.

Каналы позволяют выводу одной программы становиться вводом другой программы без переписывания и даже перекомпоновки программ. Вместо того, чтобы дескриптор 1 (стандартный вывод) исходной программы был настроен на запись на терминал, он настраивается на входной дескриптор канала. Аналогично дескриптор 0 (стандартный ввод) принимающей программы настраивается на обращение к выводу канала, а не к клавиатуре терминала. Результирующий набор двух процессов и соединяющий канал называется *конвейером*. Конвейеры могут быть весьма большими последовательностями процессов, соединенных каналами.

Системные вызовы *open*, *pipe* и *socket* порождают новые дескрипторы с наименьшим неиспользуемым номером, подходящим для дескриптора. Для того, чтобы конвейеры могли работать, должен существовать механизм для отображения таких дескрипторов в 0 и 1. Системный вызов *dup* создает копию дескриптора, которая указывает на ту же самую запись в таблице файлов. Новый дескриптор также является наименьшим неиспользуемым, но если нужный дескриптор сначала закрыть, то *dup* можно использовать для выполнения нужного отображения. Однако здесь требуется некоторая осторожность: если нужен дескриптор 1, а дескриптор 0 уже закрыт, то в результате получится дескриптор 0. Во избежание этой проблемы в системе имеется системный вызов *dup2*; он похож на *dup*, но воспринимает дополнительный аргумент, указывающий номер нужного дескриптора (если нужный дескриптор уже открыт, то *dup2* его закроет перед повторным использованием).

2.1.6.3. Устройства

Аппаратные устройства имеют связанные с ними имена файлов, и к ним может обращаться пользователь при помощи тех же самых системных вызовов, что используются для обычных файлов. Ядро может различать *специальный файл устройства* или просто *специальный файл*, и может определять, к какому устройству он относится, но большинство процессов не выполняют такого распознавания. Терминалы, принтеры и стримеры все доступны как последовательности байт, как дисковые файлы 4.4BSD. Таким образом, особенности работы устройств максимально скрываются ядром, и даже в ядре большинство из них отличаются в драйверах.

Аппаратные устройства могут быть разделены на *структурированные* или *неструктурированные*; они известны под названиями *блочные* и *посимвольные*, соответственно. Как правило, процессы обращаются к устройствам посредством *специальных файлов* в файловой системе. Операции ввода/вывода, выполняемые с такими файлами, обрабатываются постоянно находящимися в ядре программными модулями, называемыми *драйверами устройств*. Большинство аппаратных устройств для сетевых коммуникаций доступны только при помощи подсистемы межпроцессного взаимодействия, и не имеют специальных устройств в пространстве имён файловой системы, так как интерфейс *низкоуровневых сокетов* даёт более естественный интерфейс, чем специальный файл.

Структурированные или блочные устройства разделяются на диски и магнитные ленты и включают в себя большинство устройств с произвольным доступом. Ядро поддерживает операции буферизации типа чтение-изменение-запись с блочными структурированными устройствами для того, чтобы разрешить последним осуществлять чтение и запись

полностью произвольным образом, как с обычными файлами. Файловые системы создаются на блочных устройствах.

Неструктурированными устройствами являются те, что не поддерживают блочную структуру. Типичными неструктурированными устройствами являются линии связи, растровые графопостроители и небуферизируемые магнитные ленты и диски. Неструктурированные устройства, как правило, поддерживают перенос больших объёмов данных.

Неструктурированные файлы называют *символьными устройствами*, потому что первые из них являлись драйверами терминальных устройств. Интерфейс ядра к драйверу для этих устройств доказал удобство его использования для других неструктурированных устройств.

Специальные файлы устройств создаются системным вызовом *mknod*. Имеется дополнительный системный вызов, *ioctl*, для управления низкоуровневыми параметрами специальных файлов. Выполняемые операции для каждого устройства различны. Этот системный вызов позволяет осуществлять доступ к специальным характеристикам устройств, не перегружая смысл других системных вызовов. Например, для стримера существует *ioctl* для записи метки конца ленты, но нет особой или измененной версии функции *write*.

2.1.6.4. Механизм межпроцессных коммуникаций посредством сокетов

В ядре 4.2BSD появился механизм межпроцессного взаимодействия, более гибкий, чем каналы, основанный на *сокетах*. Сокет является конечной точкой коммуникаций, доступный через дескриптор, как файл или канал. Каждый из двух процессов может создать сокет, а затем соединить эти конечные точки для получения надёжного канала передачи потока байт. После соединения процесс может выполнять с дескрипторами операции чтения и записи, как это делалось с каналами. Прозрачность сокетов позволяет ядру перенаправить вывод одного процесса на вход другого, работающего на другой машине. Большим различием между каналами и сокетами является то, что каналы требуют наличия общего родительского процесса для установки коммуникации. Соединение между сокетами может быть установлено двумя несвязанными процессами, возможно, работающими на разных машинах.

System V предоставляет механизм локального межпроцессного взаимодействия через FIFO (также называемые *именованными каналами*). FIFO отображаются как объекты файловой системы, которые могут быть открыты несвязанными процессами, и в которые можно открывать и посылать данные так же, как в случае каналов. Таким образом, FIFO не требуют общего родительского процесса для установки соединения; они могут быть соединены после того, как будут запущены два процесса. В отличие от сокетов, FIFO могут быть использованы только на локальной машине; они не могут быть использованы для связи между процессами, работающими на разных машинах. FIFO реализованы в 4.4BSD, потому что это требует стандарт POSIX.1. Их функциональность является подмножеством функций интерфейса сокетов.

Механизм сокетов требует расширения традиционных для UNIX системных вызовов ввода/вывода для обеспечения соответствующих имён и смыслов соединениям. Вместо того, чтобы перегружать существующий интерфейс, разработчики использовали существующие

интерфейсы, расширив их так, что они продолжили работать без изменений, и разработали новые интерфейсы для работы с новыми возможностями. Системные вызовы *read* и *write* использовались для соединений типа потока байт, и было добавлено шесть новых системных вызовов, что позволило посылать и принимать адресованные сообщения, такие, как сетевые датаграммы. Системные вызовы для записи сообщений включают в себя *send*, *sendto* и *sendmsg*. Системные вызовы для чтения сообщений включают *recv*, *recvfrom* и *recvmsg*. В ретроспективе, первые два в каждом классе являются особыми случаями других; *recvfrom* и *sendto*, наверное, должны были быть добавлены как библиотечные интерфейсы к *recvmsg* и *sendmsg*, соответственно.

2.1.6.5. Множественный ввод/вывод

Кроме традиционных системных вызовов *read* и *write*, в 4.2BSD появилась возможность выполнять множественный ввод/вывод. Множественный ввод использует системный вызов *readv* для размещения результата единственной операции чтения в нескольких различных буферах. Обратно, системный вызов *writev* позволяет осуществлять запись нескольких различных буферов за одну атомарную операцию записи. Вместо передачи одного буфера и его длины в качестве параметров, как это делается при использовании системных вызовов *read* и *write*, процесс передаёт указатель на массив буферов и их длин, а также счетчик, определяющий размер массива.

Такой механизм позволяет буферам в различных областях адресного пространства процесса записываться атомарно, без необходимости копировать их в один буфер. Атомарные операции записи необходимы в случае, когда низкоуровневые абстракции основаны на записях, например, стримеры, которые выводят блок ленты при каждом запросе на запись. Также полезна возможность помещать результат одного запроса на чтение в нескольких различных буферах (например, заголовок записи в одно место, а данные в другое). Хотя приложение может симулировать возможность выполнять множественные операции посредством чтения данных в большой буфер с последующим копированием их частей в нужные области, и накладные расходы на копирование в памяти в таких случаях часто увеличивает время выполнения приложения чуть ли не вдвое.

Так же, как *send* и *recv* могут быть реализованы в виде библиотечных интерфейсов к *sendto* и *recvfrom*, возможно симулирование *read* через *readv* и *write* через *writev*. Однако *read* и *write* используются столь часто, что накладные расходы на такую симуляцию не стоят того.

2.1.6.6. Поддержка нескольких файловых систем

Вместе с распространением сетевых вычислений возникла потребность в поддержке как локальных, так и удаленных файловых систем. Для облегчения поддержки нескольких файловых систем разработчики добавили в ядро интерфейс виртуальных узлов файловой системы, или интерфейс *vnode*. Набор операций, экспортируемых через интерфейс vnode, похож на операции файловой системы, ранее поддерживаемые локальной файловой системой. Однако они могут поддерживаться широким спектром типов файловых систем:

- Локальные файловые системы, использующие диск
- Файлы, импортируемые при помощи разнообразных протоколов удаленных файловых систем

- Файловые системы CD-ROM, доступные только для чтения
- Файловые системы, предоставляющие специализированные услуги - к примеру, файловая система /proc

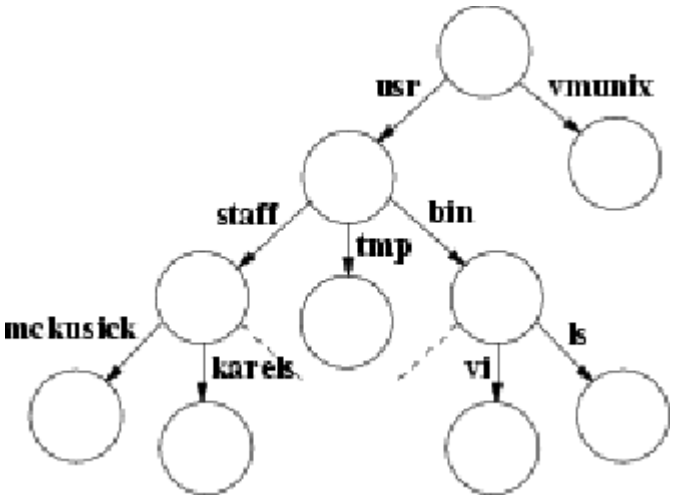
Некоторые варианты 4.4BSD, такие, как FreeBSD, позволяют выполнять динамическую загрузку файловых систем при первом обращении к ним при помощи системного вызова *mount*. Интерфейс vnode описан в Разделе 6.5; вдобавок он поддерживает функции, описанные в Разделе 6.6; некоторые из файловых систем специального назначения описаны в Разделе 6.7.

2.1.7. Файловые системы

Обычный файл представляет собой массив байтов, и может читаться и записываться, начиная с произвольного байта файла. Ядро не различает в обычных файлах границ записей, хотя многие программы воспринимают символы перевода строки в качестве признаков конца строк, но другие программы могут предполагать наличие других структур. В самом файле не хранится никакой системной информации о файле, но в файловой системе размещается некоторая информация о владельце, правах доступа и об использовании каждого файла.

Компонент под названием *имя файла* является строкой длиной до 255 символов. Эти имена хранятся в файле особого типа, который называется *каталогом*. Информация о файле в каталоге называется *записью каталога* и включает, кроме имени файла, указатель на сам файл. Записи каталога могут ссылаться как на другие каталоги, так и на обычные файлы. Таким образом формируется иерархия каталогов и файлов, которая и называется файловой системой *filesystem*;

Небольшая файловая система



Одна небольшая файловая система показана на A small filesystem. Каталоги могут содержать подкаталоги, и нет ограничений вложенности одного каталога в другой по глубине. Для соблюдения целостности файловой системы, ядро не позволяет процессу производить запись непосредственно в каталоги. Файловая система может хранить не только обычные файлы и каталоги, но также ссылки на другие объекты, такие, как устройства и сокеты.

Файловая система образует дерево, начало которого находится в *корневом каталоге*, иногда

называемому по имени *слэш*, которое соответствует символу одинарной наклонной черты (/). Корневой каталог содержит файлы; в нашем примере на Рисунке 2.2, он содержит vmunix, копию выполнимого объектного файла ядра. В нём также расположены каталоги; в этом примере он содержит каталог usr. Внутри каталога usr располагается каталог bin, который в основном содержит выполнимый объектный код программ, таких как ls и vi.

Процесс обращается к файлу, указывая *путь* до него, который является строкой, состоящей из нескольких или ни одного имён файлов, разделенных символами слэша (/). С каждым процессом ядро связывает два каталога, при помощи которых можно интерпретировать маршруты до файлов. *Корневой каталог* процесса является самой верхней точкой файловой системы, которую может достичь процесс; обычно он соответствует корневому каталогу всей файловой системы. Маршрут, начинающийся с символа слэша, называется *абсолютным маршрутом*, и интерпретируется ядром, начиная с корневого каталога процесса.

Имя пути, которое не начинается со слэша, называется *относительным маршрутом*, и интерпретируется относительно *текущего рабочего каталога* процесса. (Этот каталог кратко также называют *текущим каталогом* или *рабочим каталогом*.) Текущий каталог сам по себе можно обозначить непосредственно по имени *dot*, что соответствует одной точке (.). Имя файла *dot-dot* (..) обозначает родительский каталог текущего каталога. Корневой каталог является предком самому себе.

Процесс может задать собственный корневой каталог при помощи системного вызова *chroot*, и установить текущий каталог системным вызовом *chdir*. Каждый процесс может в любой момент выполнить вызов *chdir*, но *chroot* позволено выполнять только процессу с административными привилегиями. *Chroot* обычно используется для ограничения доступа к системе.

Взяв файловую систему, изображенную на Рисунке 2.2, и полагая, что процесс имеет в качестве корневого каталога корневой каталог файловой системы, и в качестве текущего каталога /usr, он может обратиться к файлу vi либо от корня по абсолютному имени /usr/bin/vi, либо из текущего каталога с относительным именем bin/vi.

Системные утилиты и базы данных располагаются в нескольких всем известных каталогах. Частью предопределённой иерархии является каталог, содержащий *домашний каталог* для каждого пользователя - например, /usr/staff/mckusick и /usr/staff/karels на Рисунке 2.2. Когда пользователи регистрируются в системе, то рабочий каталог их командного процессора устанавливается в домашний каталог. В своих домашних каталогах пользователи могут создавать каталоги так же легко, как и обычные файлы. Таким образом, пользователь может строить иерархии каталогов произвольной сложности.

Пользователь обычно знает только об одной файловой системе, но система может знать, что одна виртуальная файловая система на самом деле состоит из нескольких физических файловых систем, каждая из которых расположена на отдельном устройстве. Физическая файловая система не может располагаться на нескольких физических устройствах. Так как большинство физических дисковых устройств разбиваются на несколько логических устройств, то на одном физическом устройстве может располагаться более одной файловой системы, но не более одной для каждого логического устройства. Одна из файловых систем - та, с которой начинаются все абсолютные имена - называется *корневой файловой системой*,

и она всегда доступна. Другие файловые системы могут монтироваться; это значит, что они могут интегрироваться в иерархию каталогов корневой файловой системы. Ссылки на каталог, в котором находится смонтированная в него файловая системе, прозрачно преобразуются ядром в ссылки на корневой каталог смонтированной файловой системы.

Системный вызов *link* в качестве параметров принимает имя существующего файла и новое имя, которое будет присвоено файлу. После успешного выполнения вызова *link*, файл может быть доступен по любому из имен. Имя файла может быть удалено при помощи системного вызова *unlink*. Когда удаляется последнее имя для файла (и последний процесс, который держал файл открытым, закрыл его), удаляется и сам файл.

Файлы организованы иерархически в *каталоги*. Каталог является типом файла, но, в отличие от обычных файлов, каталог имеет структуру, определяемую системой. Процесс может читать каталог, как будто это обычный файл, но только ядру разрешено изменять каталог. Каталоги создаются системным вызовом *mkdir* и удаляются системным вызовом *rmdir*. До 4.2BSD системные вызовы *mkdir* и *rmdir* были реализованы как последовательность системных вызовов *link* и *unlink*. Имелось три причины для добавления системных вызовов специально для создания и удаления каталогов:

1. Операция может быть сделана атомарной. Если система завершила работу аварийно, то каталог не может оставаться в промежуточном состоянии, что может случиться при последовательном вызове серии операций.
2. При работе сетевой файловой системы создание и удаление файлов и каталогов должны выполняться атомарно, чтобы могли выполняться последовательно.
3. При реализации поддержки не-UNIX файловых систем, таких как файловая система MS-DOS, на другом разделе диска, может оказаться, что эта файловая система не поддерживает ссылочных операций. Хотя другие файловые системы могут поддерживать концепцию каталогов, скорее всего, они не будут создавать и удалять каталоги со ссылками, как это делается в файловой системе UNIX. Соответственно они могут создавать и и удалять каталоги только при наличии явных запросов на удаление или создание каталогов.

Системный вызов *chown* устанавливает владельца и группу файла, а *chmod* изменяет атрибуты защиты. Вызов *stat*, примененный к имени файла, может использоваться для чтения этих свойств файла. Системные вызовы *fchown*, *fchmod* и *fstat* применяются с дескрипторами, а не с именами файлов, для выполнения того же самого набора операций. Системный вызов *rename* может использоваться для присвоения файлу нового имени в файловой системе с заменой старого имени файла. Как и операции по созданию и удалению каталогов, системный вызов *rename* был добавлен в 4.2BSD для придания атомарности изменению имён в локальной файловой системе. Позже он оправдал свою исключительную полезность для экспортирования операций по переименованию в сторонних файловых системах и по сети.

Системный вызов *truncate* был добавлен в 4.2BSD для того, чтобы файлы могли обрезаться по указанному смещению. Вызов был добавлен первоначально для поддержки библиотеки времени выполнения языка Fortran, в котором применялось понятие конца файла с произвольным доступом, который мог устанавливаться в любую позицию, в которой был последний раз доступ к файлу. Без системного вызова *truncate* единственным способом

обрезать файл было копирование нужной части в новый файл, удаление старого и переименование копии в первоначальное имя. Библиотека могла теоретически отказываться работать на заполненной файловой системе, к тому же такой алгоритм оказывался медленным.

После того, как файловая система получила возможность обрезать файлы, ядро применяло эту возможность для уменьшения больших пустых каталогов. Преимущество в уменьшении пустых каталогов заключается в сокращении времени ядра на поиск в них при создании или удалении имен.

Вновь создаваемым файлам присваивается идентификатор пользователя процесса, который их создал, и идентификатор группы каталога, в котором они были созданы. Для защиты файлов применяется трёхуровневый механизм управления доступом. Эти три уровня определяют доступность файла для

1. Пользователя, который является владельцем файла
2. Группы, которая приписана файлу
3. Всех остальных

Каждый уровень доступа имеет отдельные индикаторы прав для чтения, записи и выполнения.

Файлы создаются с нулевым размером, который может увеличиться при выполнении операций записи. Пока файл открыт, система отслеживает указатель на файл, соответствующий текущему положению в файле, связанном с дескриптором. Этот указатель может перемешаться по файлу в произвольном порядке. Процессы, использующие один и тот же дескриптор файла посредством системных вызовов *fork* или *dup*, используют одновременно один и тот же указатель текущей позиции. Дескрипторы, созданные различными системными вызовами *open*, имеют различные указатели текущей позиции. В файлах могут присутствовать *дыры*. Дыры представляют собой пустые пространства в теле файла, в которые никаких данных не записывалось. Процесс может создать такие дыры, перемещая указатель за текущий конец файла и производя запись. При чтении дыры интерпретируются системой как заполненные нулевыми байтами.

Ранние версии UNIX имели ограничение в 14 символов на имя файла. Это ограничение зачастую вызывало проблемы. Например, кроме естественного желания пользователей давать файлам длинные описательные имена, распространённым способом формировать имена файлов является использование формата basename.extension, где расширение (указывающее на тип файла, скажем, `.c` для исходного года на языке C или `.o` для промежуточного двоичного объекта) имеет длину от одного до трёх символов, оставляя от 10 до 12 символов на имя файла. Системы управления исходным кодом и редакторы обычно используют дополнительно два символа для своих целей, для префикса или суффикса имени файла, при этом остаётся от восьми до 10 символов. В качестве имени файла легко использовать от 10 до 12 символов одного английского слова (например, `multiplexer`).

Можно смириться с этими ограничениями, но это непоследовательно и даже опасно, потому что другие системы UNIX могут работать со строками, превышающими этот лимит, при создании файлов, но затем имя будет *обрезано*. Исходный файл с именем multiplexer.c,

содержащий исходный код на языке C, (уже 13 символов) может иметь соответствующий файл из системы управления исходным кодом с префиксом `s.`, при этом получается имя файла s.multiplexer, которое не будет отличаться от файла системы управления исходным кодом для файла multiplexer.ms, содержащего исходный код `troff` для документации программы на языке C. Содержимое двух оригинальных файлов может оказаться перепутанным без каких-либо предупреждений от системы управления исходным кодом. При тщательном кодировании эту проблему можно обнаружить, но поддержка длинных имён файлов, впервые появившаяся в 4.2BSD, практически полностью ликвидировала эту проблему.

2.1.8. Размещение файлов

Операции, определённые для локальных файловых систем, делятся на две категории. Общими для всех локальных систем являются иерархический принцип именования, блокировка, квоты, управление атрибутами и защита. Эти механизмы не зависят от того, как хранятся данные. В 4.4BSD имеется единая реализация для предоставления этих сервисов.

Другой частью локальной файловой системы является организация и управление данными на носителях информации. Размещение содержимого файлов на носителях является вопросом хранилища файлов. В 4.4BSD поддерживает три различных типа хранилищ файлов:

- Традиционная файловая система Berkeley Fast Filesystem
- Журналируемая файловая система, основанная на архитектуре операционной системы Sprite [Rosenblum & Ousterhout]
- Файловая система в памяти

Хотя организация этих хранилищ совершенно различна, эти различия скрыты от процессов, использующих файловые системы.

В файловой системе Fast Filesystem организует данные в группы дорожек. Файлы, к которым, скорее всего, будет осуществляться доступ одновременно (на основе их расположения в иерархии файловой системы), хранятся на одной и той же группе дорожек. Файлы, к которым не предполагается одновременный доступ, перемещаются на разные группы дорожек. Таким образом, файлы, записываемые в одно и то же время, могут располагаться в абсолютно разных областях диска.

Файловая система с журнальной организацией организует данные в виде журнала. Все данные, записываемые в некоторый момент времени, собираются вместе и записываются в одно и то же место диска. Данные никогда не перезаписываются; вместо этого записывается новая копия файла, которая заменяет старую. Старые файлы уничтожаются процессом-сборщиком мусора, который запускается, когда файловая система переполняется и появляется необходимость в свободном пространстве.

Файловая система в памяти предназначена для хранения данных в виртуальной памяти. Она используется для файловых систем, в которых должны храниться временные данные с обеспечением быстрого доступа к ним, к примеру, /tmp. При организации файловой

системы в памяти преследуется цель организовать максимально компактное хранение данных для минимизации использования ресурсов виртуальной памяти.

2.1.9. Сетевая файловая система

Изначально сетевые возможности использовались для передачи данных от одной машины к другой. Позже это получило свое развитие в обеспечении подключения пользователей удаленно к другим машинам. Следующим логическим шагом было предоставление данных пользователю, а не приближение пользователя к данным - так родились сетевые файловые системы. Пользователи, работающие локально, не ощущают сетевых задержек при каждом нажатии клавиши, так что они получают более удобное рабочее окружение.

Подключение файловой системы к локальной машине было одним из первых основных клиент-серверных приложений. *Сервер* является удаленной машиной, которая экспортирует одну или более своих файловых систем. *Клиентом* является локальная машина, которая импортирует эти файловые системы. С точки зрения локального клиента, смонтированные удаленные файловые системы появляются в пространстве имён дерева файлов, как любая другая локально смонтированная файловая система. Локальные клиенты могут перемещаться в каталоги на удаленной файловой системе, и могут осуществлять чтение, запись и выполнение двоичных файлов на удаленной файловой системе точно так же, как они выполняют эти операции на локальной файловой системе.

Когда локальный клиент выполняет операцию на удаленной файловой системе, оформляется и посылается запрос к серверу. Сервер выполняет запрошенную операцию и возвращает либо запрошенную информацию, либо ошибку, почему запрос был отклонен. Для получения удовлетворительной производительности, клиент должен кэшировать данные, к которым доступ осуществляется часто. Сложность удаленных файловых систем отражается на поддержке соответствия между сервером и множеством его клиентов.

Хотя за эти годы было разработано множество протоколов работы с удаленными файловыми системами, самой распространённой на системах UNIX является сетевая файловая система Network Filesystem (NFS), которая была спроектирована и реализована в Sun Microsystems. Ядро 4.4BSD поддерживает протокол NFS, хотя его реализация была выполнена независимо от спецификаций протокола [Macklem]. Протокол NFS описан в Главе 9.

2.1.10. Терминалы

Терминалы поддерживают стандартные системные операции ввода/вывода, а также набор операций, специфичных для терминалов, для управления редактированием входных символов и задержек вывода. На самом нижнем уровне находятся драйверы терминальных устройств, которые управляют портами аппаратных терминалов. Терминальный ввод обрабатывается согласно низлежащим характеристикам связи, таким как скорость передачи, и согласно набору программно контролируемых параметров, таких как контроль чётности.

Выше уровня драйверов терминальных устройств находятся режимы каналов, которые обеспечивают различные уровни обработки символов. По умолчанию режим работы

канала выбирается, когда порт используется для интерактивного входа в систему. Режим работы канала устанавливается в *канонический*; входной поток обрабатывается так, что обеспечиваются стандартные функции, ориентированные на редактирование строк, и он представляется процессу в виде целых строк.

Экранные редакторы и программы, которые взаимодействуют с другими машинами, обычно работают в *неканоническом режиме* (часто называемом *raw-режимом* или *посимвольным режимом*). В этом режиме входной поток передаётся в читающий процесс сразу же и без всякой обработки. Выключается вся обработка специальных символов, не выполняется удаление символов и другое редактирование строк, все символы передаются программе, которая выполняет чтение с терминала.

Терминал может быть настроен тысячами различных способов, промежуточных между этими двумя. Например, экранный редактор, которому необходимо получать прерывания от пользователя асинхронно, может разрешить использование специальных символов, которые генерируют сигналы и разрешить управление выходным потоком, в противном случае работать в неканоническом режиме; все остальные символы будут передаваться в процесс необработанными.

Что касается выходного потока, то терминальный обработчик предоставляет простые службы по его форматированию, включая

- Преобразование символа перевода строки на двухсимвольную последовательность из символов возврата каретки и перевода строки
- Выдерживание пауз после некоторых стандартных управляющих символов
- Замещение символов табуляции
- Вывод неграфических символов ASCII в виде двухсимвольных последовательностей вида `^C` (другими словами, вывод знака вставки, за которым следует символ, который находится по смещению от символа `@`, соответствующему значению этого символа).

Каждый из этих сервисов преобразования может быть независимо выключен процессом при помощи управляющих запросов.

2.1.11. Межпроцессное взаимодействие

Межпроцессные коммуникации в 4.4BSD организованы в *коммуникационные домены*. К поддерживаемым на данный момент доменам относятся *локальный домен* для взаимодействия между процессами, выполняющимися на одной и той же машине; *межсетевой домен* для связи между процессами посредством набора протоколов TCP/IP (возможно, в сети Интернет); семейство протоколов ISO/OSI для взаимодействия между сайтами, которым нужна именно такая связь, и *домен XNS* для коммуникаций между процессами при помощи протоколов XEROX Network Systems (XNS).

В пределах домена соединения имеют место между конечными точками связи, также называемыми *сокетами*. Как отмечено в Разделе 2.6, системный вызов *socket* создает сокет и возвращает дескриптор; другие системные вызовы IPC описаны в Главе 11. Каждый сокет имеет тип, определяющий его коммуникационные свойства; к ним относятся такие характеристики, как надёжность, сохранение последовательности передаваемой

информации и предупреждение дублирования сообщений.

с каждым сокетом связан некоторый *коммуникационный протокол*. Этот протокол обеспечивает выполнение операций, требуемых сокету, согласно его типу. Приложения могут задавать нужный протокол при создании сокета или могут разрешить системе выбрать протокол, который соответствует типу создаваемого сокета.

Сокеты могут иметь адреса, связанные с ними. Формат и смысл адресов сокетов зависят от коммуникационного домена, в котором был создан сокет. Привязка имени к сокету в локальном домене приводит к созданию файла в файловой системе.

Обычные данные, передаваемые и получаемые при помощи сокетов, не имеют типа. Вопросы представления данных зависят от библиотек, которые находятся на верху коммуникационной подсистемы. Вдобавок к передаче обычных данных, коммуникационные домены могут поддерживать передачу и прием специальных типов данных, которые называются *правами доступа*. Например, локальный домен использует эту возможность для передачи дескрипторов между процессами.

До 4.2BSD сетевые реализации в UNIX обычно работали через интерфейсы символьных устройств. Одной из целей создания интерфейса сокетов было обеспечение работы простеньким программам без изменения на потоковых соединениях. Такие программы могут работать, если только не меняются системные вызовы *read* и *write*. Соответственно, оригинальные интерфейсы не трогались, но были исправлены для работы с потоковыми сокетами. Для более сложных сокетов, таких как те, что используются для посылки датаграмм и в которых при каждом вызове *send* должен указываться адрес назначения, был добавлен новый интерфейс.

Другим достоинством является то, что новый интерфейс легко переносим. Вскоре после тестового релиза, полученного из Беркли, интерфейс сокетов был перенесён в System III поставщиком UNIX (хотя AT&T не поддерживала интерфейс сокетов до выхода System V Release 4, решив использовать вместо него механизм потоков из Eighth Edition). Интерфейс сокетов был также перенесён для работы на многих адаптерах Ethernet поставщиками, такими, как Excelan и Interlan, который продавался на рынке PC, где компьютеры были слишком слабыми, чтобы обрабатывать сетевой код на основном процессоре. Сравнительно недавно интерфейс сокетов был использован в качестве основы для сетевого интерфейса Winsock от Microsoft для Windows.

2.1.12. Сетевые коммуникации

Некоторые из коммуникационных доменов, поддерживаемых IPC-механизмом *сокетов* дают доступ к сетевым протоколам. Эти протоколы реализованы как отдельный программный слой, логически находящийся ниже программного обеспечения сокетов в ядре. Ядро предоставляет много вспомогательных сервисов, таких как управление буферами, маршрутизация сообщений, стандартные интерфейсы к протоколам и интерфейсы к драйверам сетевых интерфейсов для использования в различных сетевых протоколах.

В те времена, когда разрабатывалась 4.2BSD, использовалось или разрабатывалось много сетевых протоколов, каждый со своими сильными и слабыми сторонами. Не существует

единственного подходящего на все случаи жизни протокола или набора протоколов. Поддерживая много протоколов, 4.2BSD может обеспечить взаимодействие и обмен ресурсами между различными машинами, которые были доступны в Беркли. Поддержка многих протоколов необходим также для изменений в будущем. Современные протоколы, разработанные для Ethernet со скоростями работы 10 и 100 Mbit в секунду, вряд ли будут соответствовать для завтрашних оптических сетей пропускной способностью 1 и 10 Gbit в секунду. Поэтому уровень сетевых коммуникаций разработан с учётом поддержки многих протоколов. Новые протоколы добавляются к ядру, не затрагивая поддержку старых протоколов. Старые приложения могут продолжать работать с использованием старых протоколов в той же самой физической сети, что использовалась для новых приложений, работающих с новым сетевым протоколом.

2.1.13. Сетевая реализация

Первым набором протоколов, реализованным в 4.2BSD, был Transmission Control Protocol/Internet Protocol (TCP/IP) от DARPA. CSRG выбрала TCP/IP в качестве первого для включения в набор протоколов IPC, потому что реализация на основе 4.1 была всем доступна из проекта, спонсируемого DARPA, в Bolt, Beranek и Newman (BBN). Это был выбор, повлиявший на многое: Реализация в 4.2BSD стала основной причиной очень широкой распространённости и использования этого набора протоколов. Более поздние усовершенствования производительности и возможностей TCP/IP были также широко приняты. Реализация TCP/IP подробно описана в Главе 13.

В релизе 4.3BSD появился набор протоколов Xerox Network Systems (XNS), частично основанный на работе, выполненной в Университете Мэрилэнда и Университете Корнелла. Этот набор был нужен для объединения отдельных машин, которые не могли работать с протоколом TCP/IP.

В релиз 4.4BSD был добавлен набор протоколов ISO из-за его все большей распространённости как внутри, так и во вне США. По причине использования в протоколах ISO несколько другого подхода к сети, в интерфейсе сокетов потребовалось сделать некоторые небольшие изменения для реализации этого подхода. Изменения были сделаны так, что они были незаметны для клиентов других существующих протоколов. Протоколы ISO требуют также большой работы с двухуровневыми таблицами маршрутизации, имеющимися в 4.3BSD. К значительно расширенным возможностям по маршрутизации в 4.4BSD относятся раздельные уровни маршрутизации с адресами переменной длины и сетевыми масками.

2.1.14. Работа системы

Механизмы начальной загрузки используются для запуска системы. Сначала ядро 4.4BSD должно быть загружено в основную память процессора. После загрузки оно должно пройти через фазу инициализации для установки аппаратуры в известное состояние. Затем ядро должно выполнить автоконфигурацию, в процессе которой распознаются и настраиваются периферийные устройства, подключенные к процессору. Система начинает работу в однопользовательском режиме, пока начальный скрипт выполняет проверку дисков и включает подсчет статистики и использования квот. Наконец, начальный скрипт запускает

общесистемные службы и переводит систему в полностью многопользовательский режим.

При работе в многопользовательском режиме процессы ждут запросов на вход в систему с терминальных линий и сетевых портов, которые были настроены на вход пользователей. После обнаружения запроса на вход, вызывается процесс входа в систему и выполняется аутентификация пользователя. Если она прошла успешно, запускается начальная оболочка, из которой пользователь может запускать дополнительные процессы.

Список литературы

Accetta et al, 1986 Mach: A New Kernel Foundation for UNIX Development" M.Accetta R.Baron W.Bolosky D.Golub R.Rashid A.Tevanian M.Young 93-113 USENIX Association Conference Proceedings USENIX Association June 1986

Cheriton, 1988 The V Distributed System D. R.Cheriton 314-333 Comm ACM, 31, 3 March 1988

Ewens et al, 1985 Tunis: A Distributed Multiprocessor Operating System P.Ewens D. R.Blythe M.Funkenhauser R. C.Holt 247-254 USENIX Assocation Conference Proceedings USENIX Association June 1985

Gingell et al, 1987 Virtual Memory Architecture in SunOS R.Gingell J.Moran W.Shannon 81-94 USENIX Association Conference Proceedings USENIX Association June 1987

Kernighan & Pike, 1984 The UNIX Programming Environment B. W.Kernighan R.Pike Prentice-Hall Englewood Cliffs NJ 1984

Macklem, 1994 The 4.4BSD NFS Implementation R.Macklem 6:1-14 4.4BSD System Manager’s Manual O’Reilly & Associates, Inc. Sebastopol CA 1994

McKusick & Karels, 1988 Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel M. K.McKusick M. J.Karels 295-304 USENIX Assocation Conference Proceedings USENIX Assocation June 1998

McKusick et al, 1994 Berkeley Software Architecture Manual, 4.4BSD Edition M. K.McKusick M. J.Karels S. J.Leffler W. N.Joy R. S.Faber 5:1-42 4.4BSD Programmer’s Supplementary Documents O’Reilly & Associates, Inc. Sebastopol CA 1994

Ritchie, 1988 Early Kernel Design private communication D. M.Ritchie March 1988

Rosenblum & Ousterhout, 1992 The Design and Implementation of a Log-Structured File System M.Rosenblum K.Ousterhout 26-52 ACM Transactions on Computer Systems, 10, 1 Association for Computing Machinery February 1992

Rozier et al, 1988 Chorus Distributed Operating Systems M.Rozier V.Abrossimov F.Armand I.Boule M.Gien M.Guillemont F.Herrmann C.Kaiser S.Langlois P.Leonard W.Neuhauser 305-370 USENIX Computing Systems, 1, 4 Fall 1988

Tevanian, 1987 Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach Technical Report CMU-CS-88-106, A.Tevanian Department of Computer Science, Carnegie-Mellon University Pittsburgh PA December 1987