Основы ОС UNIX. Учебный курс.

Введение

Этот краткий (предположительно, 16 часов, из которых 6 - практические занятия) вводный курс предназначен для ознакомления с архитектурой, особенностями и основными средствами ОС UNIX. При успешном освоении, курс позволит свободно и продуктивно работать в ОС UNIX в качестве пользователя и продолжить изучение администрирования или программирования этой операционной системы. Изложение ведется, в основном, без привязки к особенностям какой-либо версии UNIX, но при необходимости конкретизации, она делается для систем SVR4, в частности, ОС Solaris 8.

История, версии и основные характеристики ОС UNIX

История ОС UNIX началась в 1969 году в одном из подразделений **AT&T Bell Laboratories**, когда на "малоиспользуемой" машине DEC PDP-7 **Кен Томпсон** (Ken Thompson), **Деннис Ричи** (Dennis Ritchie) и другие (прежде занимавшиеся созданием ОС Multics) начали работу над операционной системой, названной ими первоначально Unics (*UNiplexed Information and Computing System*). В течение первых 10 лет развитие UNIX происходило, в основном, в Bell Labs. Соответствующие начальные версии назывались "Version n" (Vn) и предназначались для ЭВМ DEC PDP-11 (16-битовая) и VAX (32-битовая).
Версии Vn разрабатывались группой Computer Research Group (CRG) в Bell Labs. Поддержкой занималась другая группа, Unix System Group (USG). Разработкой также занималась группа Programmer's WorkBench (PWB), привнесшая систему управления исходным кодом **sccs**, именованные каналы и ряд других идей. В 1983 году эти группы были объединены в одну, **Unix System Development Lab**.

Хронология основных событий в истории ОС UNIX

Ниже в хронологическом порядке представлены наиболее существенные версии и события в истории UNIX вплоть до 2002 года, а также информация о взаимосвязях между ними:

1971

V1. Первая версия UNIX Time-Sharing System на ассемблере для PDP-11/20. Включала файловую систему, системный вызов **fork()** для порождения процессов, утилиты типа **cat**, **ed**, **roff**. Использовалась для обработки текстов при подготовке патентов. Системный вызов **pipe()** и поддержка программных каналов появилась в **V2**.

1973

V4. Версия, переписанная на языке C, что сделало UNIX легко переносимой на другие платформы. Язык C создавался для разработки ОС UNIX.

1974

V5. Появление первых версий в Bell Labs (PWB/UNIX, MERT).

1975

V6. Первая версия UNIX, широко распространенная за пределами Bell Labs, в частности, в университетах. С этого времени начинается появление множества других версий и UNIX становится популярной ОС. На базе этой версии в Калифорнийском университете в Беркли (UCB) создавалась **1.xBSD** (для PDP-11).

1978

Версия **2.xBSD** (Berkeley Systems Development) для PDP-11, созданная группой **Computer Systems Research Group** (CSRG) в Беркли. Поддержка сети DARPA, первая реализация стека протоколов TCP/IP. Командный интерпретатор **csh**. В дальнейших версиях (до 1980): поддержка виртуальной памяти, **termcap**, **curses**, редактор **vi**.

1979

V7. "*Последняя настоящая UNIX*", включала компилятор языка C, командный интерпретатор **sh**, систему **uucp**, была перенесена на 32-разрядный VAX. При этом размер ядра составлял около **40 Кбайт**!

1981

4.1BSD: управление заданиями, автоматическое конфигурирование ядра. **System III** - первый коммерческий UNIX от AT&T, реализация *именованных каналов* (FIFO).

1982

UNIX начинают использовать создатели рабочих станций: **SunOS** 1.0 (на базе 4.1BSD) от Sun Microsystems и **HP-UX** (на базе System III) от Hewlett-Packard.

1983

4.2BSD: полная поддержка TCP/IP, сокетов, Ethernet. Файловая система UFS с поддержкой длинных имен файлов и символьных связей. AT&T **System V**: поддержка основных утилит и средств BSD, добавлен пакет *средств межпроцессного взаимодействия* (IPC).

1984

SVR2: функции в командном интерпретаторе **sh**, первые попытки стандартизации. **SCO** XENIX - первый коммерческий UNIX на Intel-архитектуре. Создание **Free Software Foundation** (FSF) и начало проекта GNU - создание свободно распространяемой UNIX-подобной ОС и соответствующих утилит.

1985

V8 (модули STREAMS). Появление архитектуры микроядра **Mach**. Реализации стандарта SVR2: SCO XENIX SystemV/286, Interactive 386/ix. Появление ОС Minix.

1986

4.3BSD для VAX. **SVR3**: модули STREAMS из V8, TLI, поддержка динамически загружаемых библиотек. **V9** (дополнения из 4.3BSD). Появление операционных систем **AIX** (IBM) и **A/UX** (Apple).

1987

SVR3.2: SCO XENIX SV/386. Появление ОС **IRIX** (SVR3.0).

1988

4.3BSD Tahoe - 4.3BSD с исходниками. Создание **SVR4** на базе System V, BSD и SunOS (X11, NFS, система виртуальной памяти, динамически подключаемые библиотеки). Добавлены: командный интерпретатор **ksh**, ANSI C, возможности поддержки национальных языков, соответствие стандартам POSIX, X/Open. Появление компьютера NeXT с ОС **NeXTSTEP** (4.3BSD + Mach 2.0).

1990

4.3BSD Reno: поддержка различных платформ, NFS, SLIP, Kerberos. SUN **Solaris 1** (SunOS 4.1.4). Появление **OSF/1** от Open Software Foundation: микроядро Mach 2.6 + SVR4, SMP, нити, Motif GUI.

1991

BSD Net2 (**4.3BSD Lite**) - не содержит спорного кода AT&T. Появление ОС GNU HURD. Появление ОС **Linux** (на базе Minix). Выделение из AT&T отдельного подразделения **USL** (Unix System Laboratories), владеющего кодом AT&T UNIX и System V.

1992

4.4BSD: виртуальная память как в Mach 2.5, журнализируемая файловая система UFS. Закрытие CSRG в Беркли. **SVR4.2**: журнализируемая файловая система Veritas FS, списки контроля доступа ACL, динамически загружаемые модули ядра. USL **UnixWare 1** - реализация SVR4.2. **SunOS 5** = **Solaris 2** (SunOS 4 + SVR4).

1993

Появление ОС **FreeBSD**. Solaris 2.2. NeXTSTEP 3.2. IRIX 5.3, HP-UX 9.04, AIX 4.0, Linux 0.99, UnixWare 1.1.

1994

OSF 1.3: микроядро Mach 3, **поддержка 64-битовых платформ**. FreeBSD 2.0, SCO OpenDesktop 3.2.4. **UnixWare 1.1.2**. Linux 1.0.9. USL куплена компанией Novell.

1995

Появление **OpenBSD** и **NetBSD**. Solaris 2.5. Появление **Digital UNIX** (DEC OSF/1). Появление SCO OpenServer 5.0. **UnixWare 2.0**: SVR4.2 MP от Novell. Novell продает UnixWare и весь исходный код

AT&T компании **SCO**. Выход HP-UX 10 (с добавлениями из UnixWare). Завершение работ над A/UX.

1996

FreeBSD 2.1.6. OpenBSD 2.0. IRIX 6.3. Linux 2.0.21. OpenSTEP 4 - завершение проекта NeXTSTEP. SCO UnixWare 2.1. Микроядро Mach 4.

1997

FreeBSD 2.2.5, OpenBSD 2.2, NetBSD 1.3, Solaris 2.6 (под SPARC и Intel), SCO OpenServer 5.0.4. IRIX 6.4. GNU Hurd 0.2 (+ Mach 4). Linux 2.0.28.

1998

FreeBSD 3.0 (+4.4BSD), Solaris 7, DigitalUNIX 4, SCO: OpenServer 5.0.5, **UnixWare 7** (SVR5). HP-UX 11.0. **Linux 2.0.36**. IBM: проект Monterey (AIX 4.3 + SVR5).

1999

FreeBSD 3.4. OpenBSD 2.6, NetBSD 1.4. Появление **Mac OS X** и проекта **Darwin** (Mach 4 + FreeBSD 3.1). Solaris 8 beta. Компанию DEC купил Compaq: **Tru64 Unix V.5.0.** (DigitalUNIX). IRIX 6.5.6. SCO: OpenServer 5.0.5a, UnixWare 7.1.1. AIX 4.3.3. Linux 2.2.13.

2000

FreeBSD 4.0-4.2. OpenBSD 2.8. NetBSD 1.5. **Solaris 8**. Apple: Mac OS X Server, Darwin 1.2.1. Tru64 Unix V.5.1. IRIX 6.5.10. SCO: OpenServer 5.0.6. Компания SCO продала все свои ОС компании **Caldera** (Caldera OpenLinux). Hurd A1, **Linux 2.4.0**, 2.2.18. HP-UX 11i. AIX 5L alpha (проект Monterey).

2001

FreeBSD 4.4. **OpenBSD 3.0**. NetBSD 1.5.2. Mac OS X 10.1.2. SUN: Solaris 8 10/01, Solaris 9 alpha (не для Intel-архитектуры). Tru64 Unix V.5.1A, IRIX 6.5.13. SCO OpenServer 5.0.6a. Hurd H2. Caldera: **OpenUNIX 8**: UnixWare 7.1.1 + LKP=Linux 2.4.0 - прозрачная поддержка Linux-приложений. Linux **2.4.17**, 2.0.39, 2.2.20. AIX 5L v.5.1.

2002 (начало года)
Linux 2.5.2 - экспериментальное ядро.
Ожидается:

- Linux 2.6 (*это ядро вроде пока не вышло*) и обновления по старым веткам ядер (2.0, 2.2);
- Solaris 9 (*вышла*);
- FreeBSD 5 (*вышла*); последовательное развитие OpenBSD, NetBSD.
- дальнейшее развитие OpenUNIX (*теперь это снова **SCO UnixWare 7.1.3***);
- дальнейшее развитие Mac OS X;
- дальнейшее развитие существующих коммерческих версий UNIX на не-Intel архитектурах.

Современные версии ОС UNIX

Итак, в настоящее время (начало 2002 года - *В.К.*) мы имеем на платформе Intel следующие основные версии UNIX:

- FreeBSD 4.4;
- OpenBSD 3.0;
- NetBSD 1.5.2;
- Linux 2.0, 2.2, 2.4 в виде множества различных дистрибутивов;
- Solaris 8;
- SCO OpenServer 5.0.6 от Caldera;
- Caldera OpenUNIX 8.

На других платформах (основные версии):

- Linux 2.4.x (практически все платформы);
- NetBSD 1.5.x (практически все платформы);
- Mac OS X 10.x (PowerPC);
- AIX 5L (PowerPC);
- Solaris 8, 9 (SPARC);
- HP-UX 11i (PA-RISC);
- Tru64 Unix V.5.1A (Alpha);
- IRIX 6.5.13 (MIPS)

Основные характеристики

ОС UNIX имеет следующие основные характеристики:

- переносимость;

- *вытесняющая многозадачность* на основе процессов, работающих в изолированных адресных пространствах в виртуальной памяти;

- поддержка одновременной работы многих пользователей;

- поддержка асинхронных процессов;

- иерархическая файловая система;

- поддержка независимых от устройств операций ввода-вывода (через специальные файлы устройств);

- стандартный интерфейс для программ (программные каналы, IPC) и пользователей (командный интерпретатор, не входящий в ядро ОС);

- встроенные средства учета использования системы.

Архитектура ОС UNIX

Архитектура ОС UNIX - многоуровневая. На нижнем уровне, непосредственно над оборудованием, работает *ядро* операционной системы. Функции ядра доступны через *интерфейс системных вызовов*, образующих второй уровень. На следующем уровне работают *командные интерпретаторы*, команды и утилиты системного администрирования, коммуникационные *драйверы* и *протоколы*, - все то, что обычно относят к *системному программному обеспечению*. Наконец, внешний уровень образуют *прикладные программы* пользователя, сетевые и другие коммуникационные службы, СУБД и утилиты.

Основные функции ядра

Основные функции ядра UNIX (которое может быть *монолитным* или *модульным*) включают:

- планирование и переключение процессов;

- управление памятью;

- обработку прерываний;

- низкоуровневую поддержку устройств (через драйверы);

- управление дисками и буферизация данных;

- синхронизацию процессов и обеспечение средств межпроцессного взаимодействия (IPC).

Системные вызовы

Системные вызовы обеспечивают:

- сопоставление действий пользователя с запросами драйверов устройств;
- создание и прекращение процессов;
- реализацию операций ввода-вывода;
- доступ к файлам и дискам;
- поддержку функций терминала.

Системные вызовы преобразуют процесс, работающий в режиме пользователя, в защищенный процесс, работающий *в режиме ядра*. Это позволяет процессу вызывать защищенные процедуры ядра для выполнения системных функций.
Системные вызовы обеспечивают программный интерфейс для доступа к процедурам ядра. Они обеспечивают управление системными ресурсами, такими как память, пространство на дисках и периферийные устройства. Системные вызовы оформлены в виде библиотеки времени выполнения. Многие системные вызовы доступны через командный интерпретатор.

Пользовательские процессы и процессы ядра

Пользовательские процессы образуют следующие два уровня и:

- защищены от других пользовательских процессов;
- не имеют доступа к процедурам ядра, кроме как через системные вызовы;
- не могут непосредственно обращаться к пространству памяти ядра.

Пространство (памяти) ядра - это область памяти, в которой процессы ядра (процессы, работающие в контексте ядра) реализуют службы ядра. Любой процесс, выполняющийся в пространстве ядра, считается работающим в режиме ядра. Пространство ядра - привилегированная область; пользователь получает к ней доступ только через интерфейс системных вызовов. Пользовательский процесс не имеет прямого доступа ко всем инструкциям и физическим устройствам, - их имеет процесс ядра. Процесс ядра также может менять карту памяти, что необходимо для *переключения процессов* (смены контекста).
Пользовательский процесс работает в режиме ядра, когда начинает выполнять код ядра через системный вызов.

Обмен данными между пространством ядра и пользовательским пространством

Поскольку пользовательские процессы и ядро не имеют общего адресного пространства памяти, необходим механизм передачи данных между ними. При выполнении системного вызова, аргументы вызова и соответствующий идентификатор процедуры ядра передаются из пользовательского пространства в пространство ядра. Идентификатор процедуры ядра передается либо через аппаратный регистр процессора, либо через стек. Аргументы системного вызова передаются через пользовательскую область вызывающего процесса.
Пользовательская область процесса содержит информацию о процессе, необходимую ядру:

- корневой и текущий каталоги, аргументы текущего системного вызова, размеры сегмента текста, данных и стека для процесса;

- указатель на запись в таблице процессов, содержащую информацию для планировщика, например, приоритет;

- таблицу дескрипторов файлов пользовательского процесса с информацией об открытых файлах;

- стек ядра для процесса (пустой, если процесс работает в режиме пользователя).

Пользовательский процесс не может обращаться к пространству ядра, но ядро может обращаться к пространству процесса.

Системное программное обеспечение

ОС UNIX обеспечивает ряд стандартных системных программ для решения задач администрирования, переконфигурирования и поддержки файловой системы, в частности:

- для настройки параметров конфигурации системы;

- для перекомпоновки ядра (если она необходима) и добавления новых драйверов устройств;

- для создания и удаления учетных записей пользователей;

- создания и подключения физических файловых систем;

- установки параметров контроля доступа к файлам.

Для решения этих задач системное ПО (работающее в пользовательском режиме) часто использует *системные вызовы*.

Пользователи и группы

UNIX - *многопользовательская* операционная система. *Пользователи*, занимающиеся общими задачами, могут объединяться в *группы*. Каждый пользователь обязательно принадлежит к одной или нескольким группам. Все команды выполняются от имени определенного пользователя, принадлежащего в момент выполнения к определенной группе.
В многопользовательских системах необходимо обеспечивать защиту объектов (файлов, процессов), принадлежащих одному пользователю, от всех остальных. ОС UNIX предлагает базовые средства защиты и совместного использования файлов на основе отслеживания пользователя и группы, владеющих файлом, трех *уровней доступа* (для пользователя-владельца, для пользователей группы-владельца, и для всех остальных пользователей) и трех базовых *прав доступа* к файлам (на *чтение*, на *запись* и на *выполнение*).
Базовые средства защиты процессов основаны на отслеживании принадлежности процессов пользователям. Для отслеживания владельцев процессов и файлов используются числовые идентификаторы.
Идентификатор пользователя и *группы* - целое число (обычно) в диапазоне от 0 до 65535. Присвоение уникального идентификатора пользователя выполняется при заведении системным администратором нового регистрационного имени. Значения идентификатора пользователя и группы - не просто числа, которые идентифицируют пользователя, - они определяют владельцев файлов и процессов. Среди пользователей системы выделяется один пользователь - *системный администратор* или *суперпользователь*, обладающий всей полнотой прав на использование и конфигурирование системы. Это пользователь с идентификатором 0 и регистрационным именем **root**.
При представлении информации человеку удобнее использовать вместо соответствующих идентификаторов символьные имена - регистрационное имя пользователя и имя группы. Соответствие идентификаторов и символьных имен, а также другая информация о пользователях и группах в системе (*учетные записи*), как и большинство другой информации о конфигурации системы UNIX, по традиции, представлена в виде

текстовых файлов. Эти файлы - **/etc/passwd**, **/etc/group** и **/etc/shadow** (в системах с теневым хранением паролей) - детально описаны ниже.

Файл /etc/passwd

Каждая строка (учетная запись) в файле **/etc/passwd** описывает одного известного системе пользователя и имеет семь разделенных двоеточиями полей. Пример записи:

```
user_01:x:169:10:Student:/home/user_01:/bin/sh
```

Назначение полей этой записи представлено в следующей таблице.

Таблица 1. Поля файла /etc/passwd и их назначение

Поле	Назначение
Имя пользователя (регистрационное имя)	Содержит символьное имя пользователя, используемое при регистрации в системе. В пределах одной машины должно быть уникальным. Регистрационное имя должно состоять из алфавитно-цифровых символов (нижнего регистра), без пробелов, с максимальной длиной, определяемой конкретной ОС. Наиболее часто используется максимальная длина - восемь символов. Дублирование имен пользователей приводит к определенным осложнениям. Например, дубликаты появляются тогда, когда администратор использует в имени более 8 символов. Тогда для системы **jarmstrong** то же, что **jarmstroff**. Когда имя так продублировано, система использует первую найденную для него запись в файле **/etc/passwd** и игнорирует последующие.
Пароль	Поле хранит зашифрованный пароль. Допускается пустое поле. При использовании системы теневого хранения паролей, в этом поле находится только метка пароля (**x**), а зашифрованный пароль хранится в другом месте. Правила задания пароля обычно находятся в файле **/etc/default/passwd**, (например, директива **PASSLENGTH=число** в этом файле задает минимальное количество символов в пароле). Некоторые системы также учитывают регистр, а в некоторых предусматривается использование как минимум одного не алфавитно-цифрового символа.
Идентификатор пользователя	Поле хранит числовой идентификатор пользователя, который связан с его регистрационным именем. Любой созданный пользователем файл или запущенный процесс ассоциируется с его числовым идентификатором.
Идентификатор группы	Содержит числовой идентификатор группы. Любой созданный пользователем файл ассоциируется с его идентификатором группы. Указанная здесь группа является основной (первичной) для данного пользователя.
Комментарий	Содержит комментарий - любую алфавитно-цифровую строку. Предположительно это поле содержит информацию о реальном владельце регистрационного имени. ОС UNIX не задает его формат, так что подойдет любой. Некоторые программы печати и электронной почты используют это поле для вывода настоящего имени пользователя.
Начальный каталог	Определяет начальный каталог пользователя. Когда пользователь начинает сеанс работы, система помещает его в данный каталог. Пользователь должен иметь соответствующие права доступа к нему.
Начальная команда	Определяет командную среду пользователя (обычно запускается один из *командных интерпретаторов* UNIX, но, теоретически, можно указать любую команду). Это поле можно изменять.

Файл /etc/group

Этот файл соотносит числовые идентификаторы групп с символьными именами. Каждая строка файла **/etc/group** содержит четыре поля. Поля разделяются двоеточиями. Назначение полей этой записи представлено в табл. 2.

Таблица 2. Поля файла /etc/group и их назначение

Поле	Назначение
Имя группы	Содержит (уникальное) символьное имя группы.
Пароль группы	Группы могут иметь пароли, хотя использование паролей групп - явление редкое. В примере данное поле пустое - это значит, что пароль отсутствует.
Идентификатор группы	Содержит числовой идентификатор группы.

Список пользователей	Содержит список регистрационных имен пользователей данной группы. Имена в этом списке разделяются запятыми. Пользователи могут принадлежать к нескольким группам и, при необходимости, переключаться между ними с помощью команды **newgrp**.

Пример записи из файла /etc/group:

```
bin::2:root,bin,daemon
```

Файл /etc/shadow

Этот файл используется в системах с *теневым хранением паролей*, где они вынесены из доступного всем пользователям на чтение файла **/etc/passwd** для повышения безопасности системы. Здесь (помимо собственно зашифрованных паролей) хранятся дополнительные ограничения, связанные с регистрационным именем и паролем пользователя. Доступ к этому файлу на чтение имеет только пользователь **root**, а работают с ним команды **passwd** и **login**.
Файл содержит по одной записи из восьми полей, разделенных двоеточиями, для каждой учетной записи в системе. Назначение полей этой записи представлено в табл. 3.

Таблица 3. Поля файла /etc/shadow и их назначение

Номер поля	Назначение
1	Имя пользователя.
2	Зашифрованный по особому алгоритму (обычно, **DES** или **MD5**) пароль.
3	Количество дней между 01.01.1970 (началом *эры UNIX*) и днем последнего изменения пароля.
4	Минимальное количество дней между изменениями пароля.
5	Срок действия пароля пользователя.
6	За сколько дней система будет начинать предупреждать пользователя о необходимости изменения пароля.
7	Сколько дней пользователь может не работать в системе, прежде чем его регистрационное имя будет заблокировано.
8	Дата, после которой имя пользователя нельзя будет использовать в системе.

Системные регистрационные имена

Каждая версия ОС UNIX резервирует несколько специальных регистрационных имен для предопределенных системных целей. Так, в UNIX SVR4 системными считаются регистрационные имена, соответствующие идентификаторам от 0 до 100. Наиболее часто резервируются регистрационные имена, представленные в табл. 4.

Таблица 4. Системные регистрационные имена в ОС UNIX SVR4

Регистрационное имя	Назначение
root	Регистрационное имя суперпользователя, администратора системы, соответствующее идентификатору 0. Единственное имя, **обязательно** имеющееся в любой UNIX-системе. Пользователь **root** не связан никакими ограничениями по доступу. Для выполнения большинства программ администрирования используется регистрационное имя **root**, обеспечивающее гарантированный доступ к необходимым ресурсам.
daemon	Владелец процессов, реализующих пользовательские службы.
sys	Владелец выполняемых пользовательских системных команд UNIX (часто соответствует идентификатору 0).
bin	Владелец стандартных пользовательских утилит UNIX (часто соответствует идентификатору 0).
adm	Псевдопользователь, владеющий файлами системы журнализации.
cron	Псевдопользователь, владеющий соответствующими файлами, от имени которого выполняются процессы подсистемы запуска программ по расписанию.
news	Псевдопользователь, от имени которого выполняются процессы системы телеконференций (*дискуссионных групп* или *групп новостей*).
nobody	Псевдопользователь, используемый при работе сетевой файловой системы NFS.

uucp	Псевдопользователь подсистемы UUCP, позволяющий передавать почтовые сообщения и файлы между UNIX-хостами.
lp, **lpd**	Псевдопользователь, от имени которого выполняются процессы системы печати, владеющий соответствующими файлами.

Точно так же задаются и системные группы в файле **/etc/group**. В SVR4 зарезервированными считаются имена групп с идентификаторами от 0 до 100.

Изменение действующего идентификатора пользователя

Команда **su** предназначена для временного изменения действующего (эффективного) идентификатора пользователя и сеанса пользователя. Она имеет следующий синтаксис:

su [-] [**регистрационное_имя** [**аргументы** ...]

Команда **su** запрашивает пароль (у всех пользователей, кроме **root**, и если пароль существует). В случае соответствия пароля создается новый сеанс от имени нового пользователя. В следующем примере сохраняется среда пользователя с именем **user01**, включая текущий рабочий каталог и переменные среды:

```
$ logname
user01
$ su informix
Password:
$ logname
user01
$ echo $LOGNAME
user01
$ set
HOME=/home/user01
LOGNAME=user01
MAIL=/var/mail/user01
PWD=/home/user01
...
$ exit
$
```

Если введена команда **su - регистрационное_имя**, то система предоставляет пользователю командный интерпретатор и среду в соответствии с указанным регистрационным именем:

```
$ logname
user01
$ su - informix
Password:
$ logname
user01
$ echo $LOGNAME
informix
$ set
HOME=/home3/informix
LOGNAME=informix
MAILPATH=/usr/mail/informix
PWD=/home3/informix
...
$ exit
$
```

Команда в формате **su регистрационное_имя -c аргументы** воспринимает **аргумент** как команду, которую необходимо выполнить с регистрационным именем нового пользователя. Для выполнения команды запрашивается пароль нового пользователя и используются его права доступа. После завершения выполнения происходит возврат в среду пользователя, вызвавшего команду **su**. Таким образом, если пользователю, например, надо удалить файл пользователя с регистрационным именем **new_user**, необходимо выполнить команду:

```
$ su new_user -c "rm file"
Password:
$
```

Команда **su** без указания регистрационного имени позволяет получить права пользователя **root**. При этом необходимо знать и правильно ввести пароль пользователя **root**. Если пользователь работает под регистрационным именем **root**, вводить пароль при изменении действующего идентификатора не нужно.

Изменение действующего идентификатора группы

Сразу после регистрации пользователь работает от имени основной группы (задана в файле **/etc/passwd**). Кроме основной, пользователь может принадлежать к любому количеству *дополнительных групп*. Эти группы задаются путем указания регистрационного имени в четвертом поле строки в файле **/etc/group**, описывающей дополнительную группу. Членство в дополнительных группах либо учитывается при определении прав доступа автоматически (**BSD-системы**), либо для перехода в дополнительную группу и

изменения тем самым действующего идентификатора группы используется команда **newgrp** (SVR4) со следующим синтаксисом:

/usr/bin/newgrp [**группа**]

Команда **newgrp** встроена в некоторые командные интерпретаторы (**sh**, **ksh**).

Команда **newgrp** переводит пользователя в новую группу путем запуска нового командного интерпретатора с реальным и эффективным идентификатором (GID) новой группы. При этом новый командный интерпретатор запускается даже если переход в группу завершился ошибкой (например, указана несуществующая группа). Естественно, в новом командном интерпретаторе будут иметь нестандартные и непустые значения только переменные, экспортированные в среду.

При вызове без операнда, команда **newgrp** переводит пользователя в его основную группу, отменяя тем самым действие предыдущих команд **newgrp**.

Если во втором поле записи соответствующей группы в файле **/etc/group** указан пароль (т.е. если это поле не пустое) и пользователь не указан в четвертом поле как член группы, при переходе в группу у пользователя **запрашивается пароль**. Единственный способ создать пароль группы - воспользоваться командой **passwd** для задания пароля одной из учетных записей пользователей, а затем скопировать зашифрованный пароль из файла **/etc/shadow** в файл **/etc/group**. Пароли групп сейчас используют редко.

Изменение пароля и характеристик учетной записи, связанных с регистрацией

Команда **passwd** позволяет любому пользователю изменить пароль или получить список атрибутов текущего пароля для своего **регистрационного_имени**. Привилегированные пользователи могут запускать **passwd** для выполнения этих функций для любого пользователя, а также для установки атрибутов пароля для любого пользователя.

Пароль обычно задается администратором при создании учетной записи пользователя для владельца **регистрационного_имени**. В дальнейшем пользователь может изменить пароль с помощью команды **passwd**.

Команда **passwd** имеет следующий синтаксис:

passwd [**регистрационное_имя**]
passwd [-l|-d][-f][-x **max**][-n **min**][-w **warn**] **регистрационное_имя**
passwd -s [-a]
passwd -s [**регистрационное_имя**]

Опции команды представлены в табл. 5. Обычные пользователи могут использовать только опцию **-s**.

Таблица 5. Опции команды passwd

Опция	**Назначение**
-s	Показывает атрибуты пароля для **регистрационного_имени** пользователя. Любой пользователь может задавать данную опцию.
-l	Блокирует запись пароля для **регистрационного_имени**.
-d	Удаляет пароль для **регистрационного_имени**, так что у пользователя с этим регистрационным_именем пароль не запрашивается.
-f	Заставляет пользователя изменить пароль при следующей регистрации в системе, делая пароль для **регистрационного_имени** устаревшим.
-x max	Задает для пользователя с указанным **регистрационным_именем** количество дней, в течение которых пароль будет действителен.
-n min	Задает минимальное количество дней между изменениями пароля для пользователя с указанным **регистрационным_именем**. Всегда используйте эту опцию с опцией **-x**, если только **max** не установлен в **-1** (устаревание отключено). В этом случае, **min** устанавливать не нужно.
-w warn	Задает, за сколько дней (относительно **max**) пользователя с данным **регистрационным_именем** будут предупреждать о предстоящем устаревании пароля.
-s -a	Показывает атрибуты паролей для всех пользователей.

Правила построения паролей

При создании паролей обычно необходимо выполнять следующие требования:

- Пароль должен содержать не менее **PASSLENGTH** символов, как определено в файле **/etc/default/passwd**. Значение **PASSLENGTH** должно быть не менее 3. Учитываются только первые восемь символов пароля.

- Пароль должен содержать не менее двух буквенных символов и одной цифры или специального символа. (В данном случае к буквенным символам относятся все прописные и строчные буквы.)

- Пароль должен отличаться от регистрационного имени пользователя и от любого слова, получаемого *циклическим* (circular shift) или *обратным* (reverse shift) сдвигом этого регистрационного имени. (Соответствующие прописные и строчные буквы считаются совпадающими.)

Эти требования не распространяются на пользователя **root**.

Действие команды passwd

При использовании для изменения пароля команда **passwd** запрашивает у обычных пользователей их старый пароль, если он задан. Если с момента задания старого пароля прошло достаточно много времени, **passwd** затем предлагает пользователю дважды ввести новый пароль; в противном случае программа прекращает работу. Затем **passwd** проверяет, удовлетворяет ли новый пароль описанным выше правилам построения. При вводе нового пароля второй раз, две копии нового пароля сравниваются. Если они не совпадают, цикл запроса нового пароля повторяется, но не более двух раз.
Пользователь **root** может изменять любой пароль; команда **passwd** не запрашивает у него старый пароль.

Устаревание паролей

Пароли действительны в течение ограниченных периодов времени (определяемых системным администратором), после чего их необходимо изменить. Поэтому необходимо хранить информацию о периоде активности для каждого пароля. Когда приближается дата истечения срока действия пароля, его владельцу предлагается выбрать новый пароль в течение определенного количества ближайших дней. Процесс отслеживания сроков действия паролей и уведомления пользователей о необходимости сменить пароль называется *устареванием паролей* (password aging).
Информация о паролях всех пользователей системы хранится в файле **/etc/shadow**, который могут читать только привилегированные пользователи. Каждая строка пользователя в файле **/etc/shadow** содержит четыре параметра, определяющих устаревание пароля (поля 3-6, см. табл. 3). Последние три из этих параметров можно установить опциями командной строки **-n**, **-x** и **-w**, соответственно. При отсутствии опций, их значения берутся из файла **/etc/default/passwd**.

Показ атрибутов пароля

Когда команда **passwd** используется для показа атрибутов пароля, результаты выдаются в следующем формате:

```
login_name status lastchanged minimum maximum warn
```

или, если отсутствует информация, связанная с устареванием пароля,

```
login_name status
```

Поля определены следующим образом:

login_name
Регистрационное имя пользователя.

status
Статус пароля для **регистрационного_имени**: **PS** означает наличие пароля, **LK** означает, что регистрация заблокирована, а **NP** означает отсутствие пароля.

Стандартные значения атрибутов

Присваивая значения набору параметров в файле **/etc/default/passwd**, администратор может управлять устареванием и длиной паролей. Можно задать следующие параметры:

MINWEEKS
Минимальное количество недель перед тем, как пароль можно будет изменить. Сразу после установки системы этот параметр имеет значение 0.

MAXWEEKS
Максимальное количество недель, в течение которых пароль можно не изменять. Сразу после установки системы этот параметр имеет значение 24.

WARNWEEKS

Количество недель перед устареванием пароля, когда необходимо предупреждать пользователя. Сразу после установки системы этот параметр имеет значение 1.

PASSLENGTH

Минимальное количество символов в пароле. Сразу после установки системы этот параметр имеет значение 6.

Обратите внимание, что аргументы опций команды **passwd** (**min**, **max** и **warn**), а также соответствующие поля файла **/etc/shadow** задают параметры устаревания в днях; тогда как соответствующие поля файла **/etc/default/passwd** (**MINWEEKS**, **MAXWEEKS** и **WARNWEEKS**) - в неделях.

Просмотр базы данных учетных записей

Для просмотра базы данных учетных записей системы предназначена команда **logins**. Команда **logins** выдает информацию о пользовательских и системных регистрационных именах. Содержание выдаваемой информации управляется опциями команды и может включать: регистрационное имя, идентификатор пользователя, описание учетной записи в файле **/etc/passwd** (реальное имя пользователя или другая информация), имя основной группы, идентификатор основной группы, имена групп, идентификаторы групп, начальный каталог, начальный командный интерпретатор и четыре параметра устаревания пароля.
По умолчанию выдается следующая информация: регистрационное имя, идентификатор пользователя, имя основной группы, идентификатор основной группы и поле описания учетной записи в файле **/etc/passwd**. Результат сортируется по идентификатору пользователя, в результате чего сначала идут системные регистрационные имена, а затем - пользовательские.
Команда logins имеет следующий синтаксис:

logins [-dmopstuxa] [-g **группы**] [-l **рег_имена**]

Действие опций команды logins представлено в табл. 6.

Таблица 6. Опции команды logins

Опция	Назначение
-d	Выбирает регистрационные имена с дублирующимися идентификаторами пользователя.
-m	Показывает все группы, к которым принадлежит пользователь.
-o	Форматирует вывод в виде одной строки полей, разделенных двоеточиями.
-p	Выбирает регистрационные имена без паролей.
-s	Выбирает все системные регистрационные имена.
-t	Сортирует результат по регистрационному имени, а не по идентификатору пользователя.
-u	Выбирает все пользовательские регистрационные имена.
-x	Выдает расширенную информацию о каждом выбранном пользователе. Эта расширенная информация включает начальный каталог, начальный командный интерпретатор и информацию об устаревании паролей, причем каждый элемент выдается в отдельной строке. Информация о пароле содержит статус пароля (**PS** при наличии пароля, **NP** при отсутствии пароля или **LK** для заблокированного регистрационного имени), дату последнего изменения пароля, количество дней, через которое потребуется изменить пароль, минимальное количество дней между изменениями и за сколько дней пользователь начнет получать (при регистрации) предупреждающее сообщение об устаревании пароля.
-a	Добавляет к результату два поля, связанных с устареванием пароля. Они показывают, сколько дней пароль можно не использовать, перед тем как он автоматически деактивируется, и дату устаревания пароля.
-g	Выбирает всех пользователей, принадлежащих указанной группе, сортируя список по идентификатору пользователя. Можно указывать несколько групп в виде списка через запятую.
-l	Выбирает указанное регистрационное имя. Можно указывать несколько регистрационных имен в виде списка через запятую.

При совместном использовании нескольких опций будут показаны учетные записи, удовлетворяющие любому из критериев. При совместном использовании опций **-l** и **-g** информация о пользователе будет выдаваться один раз, даже если он принадлежит к нескольким указанным группам.

Получение списка зарегистрировавшихся пользователей

Для получения списка пользователей, работающих сейчас в системе, используется команда **who** со следующим синтаксисом:

```
/usr/bin/who [ -abdHlmpqrstTu ] [ файл ]
/usr/bin/who -q [ -n x ] [ файл ]
/usr/bin/who am i
```

Последний вариант выдает строку, соответствующую запрашивающему сеансу, и может использоваться для самоидентификации.

Утилита **who** выдает имя пользователя, терминал, время регистрации, время, прошедшее после последней выполненной команды, а также идентификатор процесса командного интерпретатора. Для получения этой информации она просматривает файл **/var/adm/utmp**. Если указан **файл** (который должен иметь формат **utmp(4)**), информация берется из него.

В общем случае, результат имеет следующий вид:

имя [**состояние**] **терминал время** [**ожидание**] [**pid**] [**комментарий**] [**статус выхода**]

где:

имя
: регистрационное имя пользователя

состояние
: возможность записи на терминал

терминал
: имя терминала из каталога **/dev**

время
: время регистрации пользователя

ожидание
: время, прошедшее после последнего действия пользователя

pid
: идентификатор процесса командного интерпретатора

комментарий
: строка комментария из файла **/etc/inittab** (SVR4)

статус выхода
: статус возврата для "мертвых" процессов

Опции команды **who** представлены в табл. 7.

Таблица 7. Опции команды who

Опция	Назначение
-a	Обрабатывает **/var/adm/utmp** или указанный **файл** с опциями **-b**, **-d**, **-l**, **-p**, **-r**, **-t**, **-T** и **-u**.
-b	Выдает дату и время последней перезагрузки.
-d	Выдает все процессы, прекращенные и не перезапущенные процессом **init**. Для "мертвых" процессов будет выдано поле статуса выхода. Это может пригодиться для выяснения причины прекращения процесса. Только для SVR4.
-H	Выдает заголовки столбцов.
-l	Выдает только терминалы, на которых система ожидает регистрации пользователей. В качестве имени для них выдается **LOGIN**. Остальные поля - такие же, как и для пользователей, но поле состояния не выводится.
-m	Выдает информацию только о текущем терминале.
-n x	Выдает по **x** пользователей в строке. Значение **x** должно быть не менее 1. Опция **-n** может использоваться только с опцией **-q**.
-p	Выдает информацию об активных процессах, запущенных ранее процессом **init**. В поле имени выдается имя программы, запущенной процессом **init** в соответствии с файлом **/sbin/inittab**. Поля состояния, терминала и ожидания в этом случае не имеют смысла. Поле комментария показывает идентфикатор строки из файла **/sbin/inittab**, запустившей этот процесс. Только для SVR4.
-q	(quick who) Выдает только имена и количество зарегистрированных пользователей. Если задана эта опция, другие опции игнорируются.
-r	Показывает текущий уровень выполнения процесса **init**. Только для SVR4.

-s	Выдает только поля имени, терминала и времени регистрации. Используется по умолчанию.
-T	То же, что и опция **-s**, но также выдаются поля состояния, времени ожидания, pid и комментарий. В поле состояния выдается один из следующих символов: **+** терминал разрешает запись другим пользователям; **-** терминал запрещает запись другим пользователям; **?** возможность записи на терминал не определена.

Рассмотрим примеры выполнения команды who в ОС Solaris 8:

```
[kravchuk@arturo 09:40:03 /]$ who -a | more
   .       system boot  Фев 23 15:39
   .       run-level 3  Фев 23 15:39     3      0  S
rc2            .       Фев 23 15:41  old      84  id=  s2 term=0   exit=0
root     + console      Фев 27 21:34  0:28   4612       (:0)
rc3            .       Фев 23 15:41  old     359  id=  s3 term=0   exit=0
sac            .       Фев 23 15:41  old     411  id=  sc
LOGIN      console      Фев 23 15:41  0:28    428
panaslog       .       Фев 23 15:41  old     413  id=  e1
netwatch       .       Фев 25 12:02  old     415  id=  up term=15  exit=0
zsmon          .       Фев 23 15:41  old     423
informix + pts/1        Map 25 10:13 15:21   1796       (khomjak.profix.com)
eugene   + pts/3        Map 22 18:23 15:24  23392       (khomjak.profix.com)
serj     + pts/4        Map 18 10:41  old   13278       (sysadm.profix.com)
serj     + pts/15       Map 25 11:32 14:51   3004       (sysadm.profix.com)
kravchuk + pts/14       Map 26 09:39   .    11615       (creator.profix.com)
slavik   + pts/2        Map 21 14:18 16:13  14526       (slavik.profix.com)
informix + pts/17       Map 21 13:19 17:50  14012       (bachin.profix.com)
informix   pts/6        Map 25 18:34 15:05   3572  id=t800 term=0   exit=0
(lyapota.profix.com)
lyapota    pts/7        Map 25 18:34 17:58   3577  id=t900 term=0   exit=0
(lyapota.profix.com)
informix + pts/5        Map  5 14:48 15:33  27664       (slavik.profix.com)
kravchuk   pts/8        Map 25 18:24 15:15   8916  id=tB00 term=0   exit=0
--More--
```

В простейшем случае программа **who** вызывается без параметров:

```
[kravchuk@arturo 09:45:35 /]$ who
root      console      Фев 27 21:34    (:0)
informix  pts/1        Map 25 10:13    (khomjak.profix.com)
eugene    pts/3        Map 22 18:23    (khomjak.profix.com)
serj      pts/4        Map 18 10:41    (sysadm.profix.com)
serj      pts/15       Map 25 11:32    (sysadm.profix.com)
kravchuk  pts/14       Map 26 09:39    (creator.profix.com)
slavik    pts/2        Map 21 14:18    (slavik.profix.com)
informix  pts/17       Map 21 13:19    (bachin.profix.com)
informix  pts/5        Map  5 14:48    (slavik.profix.com)
root      pts/13       Фев 27 21:35    (:0.0)
root      pts/16       Map 25 17:24    (:0.0)
```

Наконец, вот как используется команда **who** для самоидентификации:

```
[kravchuk@arturo 09:45:38 /]$ who am i
kravchuk   pts/14       Map 26 09:39    (creator.profix.com)
```

Средства создания, изменения и удаления учетных записей пользователей

Поскольку база данных учетных записей организована в виде обычных текстовых файлов, основные задачи управления учетными записями могут решаться с помощью обычного текстового редактора, например, **vi**. Однако поскольку при этом требуется согласованное изменение нескольких файлов, в системе для управления учетными записями предлагается ряд утилит командной строки, средства на основе меню или на основе графического пользовательского интерфейса.

Для создания, изменения и удаления учетных записей все версии ОС UNIX предлагают три команды, **useradd**, **usermod** и **userdel**, соответственно. Они в большинстве систем имеют следующий синтаксис:

useradd [-u **идентификатор** [-o] [-i]] [-g **группа**]
[-G **группа**[[,**группа**]...]] [-d **каталог**] [-s **shell**]
[-c **комментарий**] [-m [-k **skel_dir**]] [-f **inactive**]
[-e **expire**] **рег_имя**

usermod [-u **идентификатор** [-o]] [-g **группа**]
[-G **группа**[[,**группа**]...]] [-d **каталог** [-m]]
[-s **shell**] [-c **комментарий**] [-l **новое_рег_имя**]
[-f **inactive**] [-e **expire**] **рег_имя**

userdel [-r] **рег_имя**

Эти команды позволяют выполнить только согласованные и допустимые изменения в файлах **/etc/passwd**, **/etc/shadow** и **/etc/group**. Команды управления учетными записями, в общем случае, может выполнять только пользователь **root**. Основные опции команд управления учетными записями представлены в табл. 8.

Таблица 8. Основные опции команд управления учетными записями

Опция	Назначение
-u идентификатор	*Идентификатор пользователя* (UID). Должен быть неотрицательным целым числом, не превосходящим **MAXUID**, определенный в **sys/param.h**. По умолчанию используется следующий доступный (уникальный) не устаревший UID в диапазоне пользовательских идентификаторов.
-o	Эта опция позволяет продублировать UID (сделать его не уникальным). Поскольку защита системы в целом, а также целостность *контрольного журнала* (audit trail) и *регистрационной информации* (accounting information) в частности, зависит от однозначного соответствия каждого UID определенному физическому лицу, использовать эту опцию не рекомендуется.
-i	Позволяет использовать устаревший идентификатор UID.
-g группа	Целочисленный идентификатор или символьное имя существующей группы. Эта опция задает *основную группу* (primary group) для нового пользователя. По умолчанию в SVR4 используется стандартная группа, указанная в файле **/etc/default/useradd**. В ОС FreeBSD и Linux обычно принято по умолчанию создавать для каждого пользователя отдельную приватную основную группу, имя которой совпадает с именем пользователя.
-G группа [[,группа] ...]	Один или несколько элементов в списке через запятую, каждый из которых представляет собой целочисленный идентификатор или символьное имя существующей группы. Этот список определяет принадлежность к *дополнительным группам* (supplementary group membership) для пользователя. Повторения игнорируются. Количество элементов в списке не должно превосходить **NGROUPS_MAX-1**, поскольку общее количество дополнительных групп для пользователя плюс основная группа не должно превосходить **NGROUPS_MAX**.
-d каталог	*Начальный каталог* (home directory) нового пользователя. Длина этого поля не должна превосходить определенного предела (обычно - от 256 до 1024 символов). По умолчанию используется **HOMEDIR/рег_имя**, где **HOMEDIR** - базовый каталог для начальных каталогов новых пользователей, а **рег_имя** - регистрационное имя нового пользователя.
-s shell	Полный путь к программе, используемой в качестве начального командного интерпретатора для пользователя сразу после регистрации. Длина этого поля не должна превосходить определенного предела (обычно - от 256 до 1024 символов). По умолчанию в этом поле используется стандартный командный интерпретатор **/bin/sh**. В качестве значения **shell** должен быть указан существующий выполняемый файл. В противном случае, пользователь не сможет зарегистрироваться в системе.
-c комментарий	Любая текстовая строка. Обычно, это краткое описание регистрационного имени, например, фамилия и имя реального пользователя. Эта информация хранится в записи пользователя в файле **/etc/passwd**. Длина этого поля не должна превосходить 128 символов.
-m	Создает начальный каталог нового пользователя, если он еще не существует. Если каталог уже существует, добавляемый пользователь должен иметь права на доступ к указанному каталогу.
-k skel_dir	Копирует содержимое *скелетного каталога* **skel_dir** в начальный каталог нового пользователя, вместо содержимого стандартного скелетного каталога, **/etc/skel**. Каталог **skel_dir** должен существовать. Стандартный скелетный каталог содержит стандартные файлы, определяющие среду работы пользователя. Заданный администратором каталог **skel_dir** может содержать аналогичные файлы и каталоги, созданные для определенной цели.
-f inactive	Максимально допустимое количество дней между регистрациями, когда это имя еще не объявляется недействительным. Обычно в качестве значений используются положительные целые числа.
-e expire	Дата, начиная с которой регистрационное имя больше нельзя будет использовать; после этой даты никакой пользователь не сможет получить доступ под этим регистрационным именем. (Эта опция удобна при создании временных регистрационных имен.) Вводить значение аргумента **expire** (представляющего собой дату) можно в любом **поддерживаемом локалью** формате (кроме Julian date). Например, можно ввести **10/6/99** или **October 6, 1999**.
-l новое_рег_имя	Строка печатных символов, задающая новое регистрационное имя для пользователя. Она не должна содержать двоеточий (**:**) и переводов строк (**\n**). Кроме того, она не должна начинаться с прописной буквы.

-r	При удалении учетной записи удалить начальный каталог пользователя из системы. Этот каталог должен существовать. После успешного выполнения команды файлы и подкаталоги в начальном каталоге будут недоступны.
рег_имя	Строка печатных символов, задающая регистрационное имя для нового пользователя. В ней не должно быть двоеточий (**:**) и символов перевода строки (**\n**). Она также не должна начинаться с прописной буквы.

Учтите, что вновь созданная учетная запись блокируется до тех пор, пока не будет выполнена команда **passwd**, задающая пароль новому пользователю.
Рассмотрим ряд простых примеров управления учетными записями:

```
# useradd -c "Student 1" -d /home/user01 -g ixusers -m -s /bin/bash user01
# usermod -c "Student 1 of UNIX Course" -G others -s /bin/ksh user01
# userdel -r user01
```

Средства создания, изменения и удаления групп

Для создания, изменения и удаления групп все версии ОС UNIX предлагают три команды, **groupadd**, **groupmod** и **groupdel**, соответственно. Они имеют следующий синтаксис:

groupadd [-g **идентификатор** [-o]] **группа**
groupmod [-g **идентификатор** [-o]] [-n **имя**] **группа**
groupdel **группа**

Эти команды позволяют выполнить только согласованные и допустимые изменения в файле **/etc/group**. Команды управления группами, в общем случае, может выполнять только пользователь **root**. Опции и операнды команд управления группами представлены в табл. 9.

Таблица 9. Опции команд управления группами

Опция	Назначение
-g идентификатор	Идентификатор новой группы (GID). Этот идентификатор группы должен быть неотрицательным десятеричным целым числом, не превышающим значения **MAXUID**, определенного в заголовочном файле <param.h>. По умолчанию выделяется уникальный идентификатор группы, не относящийся к зарезервированным. В UNIX SVR4 идентификаторы групп в диапазоне 0-100 зарезервированы.
-o	Эта опция позволяет задавать дублирующийся (не уникальный) идентификатор группы.
-n имя	Строка печатных символов, задающая новое имя для группы при изменении. Строка не должна содержать двоеточия (**:**) или переводы строк (**\n**).
группа	Имя создаваемой, изменяемой или удаляемой группы. Имя группы не должно содержать символы двоеточия (**:**) или перевода строки (**\n**).

Учтите, что при удалении группы просто удаляется строка из файла **/etc/group**. Никакие изменения в файловой системе и в учетных записях пользователей команды **groupmod** и **groupdel** не производят. Соответствующие действия по согласованию, при необходимости, должен выполнять системный администратор - пользователь root.
Рассмотрим ряд простых примеров управления группами:

```
# groupadd -g 101 informix
# groupmod -g 102 -o -n ixusers informix
# groupdel ixusers
```

Файлы и каталоги

Операционная система выполняет две основные задачи: манипулирование данными и их хранение. Большинство программ в основном манипулирует данными, но, в конечном счете, они где-нибудь хранятся. В системе UNIX таким местом хранения является *файловая система*. Более того, в UNIX **все устройства**, с которыми работает операционная система, также представлены в виде специальных файлов в файловой системе.

Понятие логической файловой системы

Логическая файловая система в ОС UNIX (или просто *файловая система*) - это иерархически организованная структура всех каталогов и файлов в системе, начинающаяся с *корневого* каталога. Файловая система UNIX обеспечивает унифицированный интерфейс доступа к данным, расположенным на различных носителях, и к периферийным устройствам. Логическая файловая система может состоять из одной или нескольких *физических файловых (под)систем*, являющихся разделами физических носителей (дисков, CD-ROM или дискет).

Файловая система контролирует права доступа к файлам, выполняет операции создания и удаления файлов, а также выполняет запись/чтение данных файла. Поскольку большинство прикладных функций выполняется через интерфейс файловой системы, следовательно, права доступа к файлам определяют привилегии пользователя в системе.
Файловая система обеспечивает перенаправление запросов, адресованных периферийным устройствам, соответствующим модулям подсистемы ввода-вывода.

Ориентация и навигация в файловой системе

Иерархическая структура файловой системы UNIX упрощает ориентацию в ней. Каждый каталог, начиная с корневого (/), в свою очередь, содержит файлы и другие каталоги (*подкаталоги*). Каждый каталог содержит также ссылку на родительский каталог (для корневого каталога родительским является он сам), представленную каталогом с именем две точки (**..**) и ссылку на самого себя, представленную каталогом с именем точка (**.**).
Каждый процесс имеет *текущий каталог*. Сразу после регистрации текущим каталогом пользователя (на самом деле, процесса - начальной программы, обычно, командного интерпретатора) становится *начальный каталог*, указанный в файле **/etc/passwd**.
Каждый процесс может сослаться (назвать) на любой файл или каталог в файловой системе по имени. Способам задания имен файлов посвящен следующий подраздел.

Имена файлов в ОС UNIX

В ОС UNIX поддерживается три способа указания имен файлов:

- *Краткое имя*. Имя, не содержащее специальных метасимволов косая черта (/), является кратким именем файла. По краткому имени можно сослаться на файлы текущего каталога. Например, команда **ls -l .profile** требует получить полную информацию о файле **.profile** в текущем каталоге.

- *Относительное имя*. Имя, не начинающееся с символа косой черты (/), но включающее такие символы. Оно ссылается на файл относительно текущего каталога. При этом для ссылки на файл или каталог в каком-то другом каталоге используется метасимвол косой черты (/). Например, команда **ls -l ../.profile** требует получить полную информацию о файле **.profile** в родительском каталоге текущего каталога, а команда **vi doc/text.txt** требует открыть в редакторе **vi** файл **text.txt** в подкаталоге **doc** текущего каталога.

- *Полное имя*. Имя, начинающееся с символа косой черты (/). Оно ссылается на файл относительно корневого каталога. Это имя еще называют *абсолютным*, так как оно, в отличие от предыдущих способов задания имени, ссылается на один и тот же файл независимо от текущего каталога. Например, команда **ls -l /home/user01/.profile** требует получить полную информацию о файле **. profile** в каталоге **/home/user01** независимо от того, в каком каталоге выполняется.

Другие символы, кроме косой черты, не имеют в именах файлов UNIX особого значения (это не метасимволы). В частности, нет системного понятия *расширения* файла.
В ОС UNIX нет теоретических ограничений на количество вложенных каталогов. Тем не менее, в каждой реализации имеются практические ограничения на максимальную длину имени файла, которое указывается в командах (как и на длину командной строки в целом). Оно задается константой **PATH_MAX** в заголовочном файле **/usr/include/limits.h**. Так, в ОС Solaris 8 имя файла не может быть длиннее 1024 символов.

Получение информации о текущем каталоге

Команда **pwd** выдает полное имя *текущего* (*рабочего*) каталога. Команда **pwd** не имеет параметров. Вот пример ее использования:

```
$ pwd
/home/user01
$
```

Изменение текущего каталога

Для изменения текущего каталога используется команда **cd**:

cd [**каталог**]

Если **каталог** не указан, используется значение переменной среды **$HOME** (обычно это *начальный каталог* пользователя). Чтобы сделать новый каталог текущим (войти в каталог), нужно иметь для него **право на выполнение**. Команда **cd** является встроенной командой интерпретатора и использует для изменения текущего каталога соответствующий системный вызов.

Рассмотрим пример совместного использования команд **cd** и **pwd** для переходов по каталогам файловой системы:

```
$ pwd
/home/user01
$ cd ..
$ pwd
/home
$ cd user01/tmp
$ pwd
/home/user/tmp
$ cd
$ pwd
/home/user01
```

Получение информации о файлах

Для просмотра информации о типах (и других атрибутах) файлов в ОС UNIX используется команда **ls** со следующим синтаксисом:

ls [-abCcdeFfgiLlmnopqRrstux1] [**файл** ...]

Команда **ls** выдает информацию об указанных файлах или о файлах и каталогах в текущем каталоге (если **файл** не задан). Формат и подробность выдаваемой информации зависит от опций. Основные опции команды **ls** представлены в табл. 10:

Таблица 10. Основные опции команды ls

Опция	Назначение
-a	Выдает все файлы и подкаталоги, включая те, имена которых начинаются с точки (**.**). По умолчанию такие файлы не выдаются (они считаются *скрытыми*).
-F	Добавляет к имени файла суффикс, показывающий его *тип* (см. следующий раздел). Помечает каталоги косой чертой (/), выполняемые файлы - звездочкой (*****), именованные каналы (FIFO) - вертикальной чертой (|), символические связи - "собакой" (**@**), а сокеты - знаком равенства (=).
-i	Для каждого файла выдает в первом столбце листинга номер *индексного дескриптора* (i-node). Об индексных дескрипторах см. в разделе, посвященном физическим файловым системам UNIX.
-l	Выдает длинный листинг, включающий права доступа, количество связей, владельца, группу, размер в байтах, время последнего изменения каждого файла и, естественно, имя файла. Если файл является специальным файлом устройства, вместо размера выдаются главный и второстепенный номера устройства. Если с момента последнего изменения прошло более 6 месяцев, оно обычно выдается в формате '**месяц день год**'. Для файлов, измененных позднее, чем 6 месяцев назад, время выдается в формате '**месяц день время**'. Если файл является символической связью, в длинном листинге после имени файла указывается стрелочка (**->**) и имя файла, на который связь ссылается.
-r	Изменяет порядок сортировки на обратный стандартному (обратный лексикографический или сначала самые старые файлы, в зависимости от других опций).
-R	Рекурсивно выдает содержимое подкаталогов.
-t	Сортирует листинг по временно*й* отметке (сначала - самые новые), а не по имени файла. По умолчанию используется время последнего изменения. (Опции **-u** и **-c** позволяют сортировать по времени последнего обращения и времени создания, соответственно.)

Как видно из синтаксиса, можно задавать одновременно несколько опций. Вот как можно посмотреть подробную информацию о файлах в каталоге **/tmp**, начиная с самых давно изменявшихся:

```
[kravchuk@arturo 14:28:07 /tmp]$ cd /tmp
[kravchuk@arturo 14:28:07 /tmp]$ ls -lrt
-rw-rw-r--   1 root     sys         8296 Фев 23 15:39 ps_data
drwx------   2 root     root         110 Фев 23 15:41 smc898
-rw-rw----   1 root     informix       0 Фев 23 18:53 bar_dbug.log
drwxr-xr-x   2 informix informix     115 Фев 24 13:05 txt
-rw-------   1 root     other          0 Фев 25 13:50 mpQ1aGEp
-rw-------   1 remisov  software       0 Фев 25 15:03 mpwsaWPr
-rw-------   1 remisov  software       0 Фев 25 15:37 mpOFaqZs
-rw-------   1 remisov  software       0 Фев 25 16:25 mppfaa.t
```

```
-rw-------   1 remisov  software      0 Фев 26 13:30 mpUCaGDG
-rw-------   1 remisov  software      0 Фев 26 14:25 mpRfaaSI
-rw-------   1 remisov  software      0 Фев 26 16:41 mpCJaqUL
-rw-------   1 remisov  software      0 Фев 26 16:56 mpUgaatM
-rw-------   1 remisov  software      0 Фев 26 17:01 mpYua4AM
-rw-r--r--   1 informix informix    565 Фев 27 17:27 mkdb1994.err
-rw-r--r--   1 informix informix   2062 Фев 27 17:29 mkdb2003.err
-rw-------   1 root     other     314872 Фев 27 21:35 dtdbcache_:0
-rw-r--r--   1 root     other          0 Фев 27 21:35 sdtvolcheck4684
...
```

Основным форматом результатов **ls** является так называемый *длинный листинг* (задаваемый опцией **-l**). По умолчанию выдаются только имена файлов в несколько столбцов:

```
[kravchuk@arturo 14:31:40 /tmp]$ cd /
[kravchuk@arturo 14:31:47 /]$ ls
INFORMIXTMP   dev          home         opt          sbin
INFORMIXTMP9  devices      kernel       opt.orig     tmp
TT_DB         dt           lib          platform     usr
bin           etc          lost+found   prj          var
boot          export       mnt          proc         vol
cdrom         fs           net          root         xfn
[kravchuk@arturo 14:31:48 /]$
```

Часто также используется опция **-F**:

```
[kravchuk@arturo 14:31:48 /]$ ls -F
INFORMIXTMP/   dev/          home/        opt@         sbin/
INFORMIXTMP9/  devices/      kernel/      opt.orig/    tmp/
TT_DB/         dt/           lib@         platform/    usr/
bin@           etc/          lost+found/  prj@         var/
boot/          export/       mnt/         proc/        vol/
cdrom/         fs/           net/         root/        xfn/
```

Многочисленные примеры использования и результатов выполнения команды **ls** представлены в следующих разделах.

Типы файлов

В UNIX существует несколько типов файлов, различающихся по функциональному назначению и действиям операционной системы при выполнении тех или иных операций над ними. В следующих подразделах кратко представлены основные типы файлов, их признаки в длинных листингах, а также способы их создания.

Обычный файл

Представляет собой наиболее общий тип файлов, содержащий данные в некотором формате. Для операционной системы такие файлы представляют собой просто последовательность байтов. К этим файлам относятся текстовые файлы, двоичные данные и выполняемые программы.

В длинном листинге признаком обычного файла является дефис (**-**) в первой позиции первого столбца:

```
-rw-rw-r--   1 root     sys        8296 Фев 23 15:39 ps_data
```

Обычные файлы создаются текстовыми редакторами (текстовые), компиляторами (двоичные), прикладными программами с помощью соответствующего системного вызова или путем перенаправления вывода:

```
[kravchuk@arturo 14:40:01 /]$ cd /tmp
[kravchuk@arturo 14:40:04 /tmp]$ >f1.txt
[kravchuk@arturo 14:40:09 /tmp]$ ls -l f1.txt
-rw-r--r--   1 kravchuk 50            0 Map 26 14:40 f1.txt
```

Каталог

С помощью каталогов формируется логическое дерево файловой системы. *Каталог* - это файл, содержащий имена находящихся в нем файлов, а также указатели на дополнительную информацию - метаданные, позволяющие операционной системе производить действия с этими файлами. Каталоги определяют положение файла в дереве файловой системы. Любой процесс, имеющий право на чтение каталога, может прочесть его содержимое, но только ядро имеет право на запись данных каталога.

В длинном листинге признаком каталога является символ **d** в первой позиции первого столбца:

```
drwxr-xr-x   2 informix informix     115 Фев 24 13:05 txt
```

Каталоги создаются командой **mkdir**:

```
mkdir каталог ...
```

Специальный файл устройства

Обеспечивает доступ к физическим устройствам. В UNIX различают *символьные* (character special device) и *блочные* (block special device) файлы устройств. Доступ к устройствам осуществляется путем открытия, чтения и записи в специальный файл устройства.
Символьные файлы устройств используются для небуферизованного обмена данными с устройством.
Блочные файлы устройств позволяют производить обмен данными в виде пакетов фиксированной длины - блоков.
В длинном листинге признаком специального символьного и блочного устройств являются символы **c** и **b** в первой позиции первого столбца, соответственно:

```
$ cd /devices/pci\@0,0/pci-ide\@7,1/ide\@0
$ ls -l | more
total 0
crw-------   1 root     sys       77,  0 Фев 14 14:03 nv@0,0:0
brw-r-----   1 root     sys       29,  0 Апр 20  2001 sd@0,0:a
crw-r-----   1 root     sys       29,  0 Апр 20  2001 sd@0,0:a,raw
brw-r-----   1 root     sys       29,  1 Апр 20  2001 sd@0,0:b
crw-r-----   1 root     sys       29,  1 Апр 20  2001 sd@0,0:b,raw
brw-r-----   1 root     sys       29,  2 Апр 20  2001 sd@0,0:c
crw-r-----   1 root     sys       29,  2 Апр 20  2001 sd@0,0:c,raw
...
```

Специальные файлы устройство создаются командой **mknod**:

mknod **имя** b **главный_номер второстепенный_номер**
mknod **имя** c **главный_номер второстепенный_номер**

Главный номер устройства задает *драйвер* (индекс в таблице драйверов системы), или *тип* устройства, а *второстепенный - экземпляр* устройства.
Создавать специальные файлы устройств обычно может только пользователь **root**. Вот как можно создать новый специальный файл устройства для одного из представленных в листинге выше устройств:

```
# mknod slice1 b 29 1
# ls -l slice1
brw-r-----   1 root     sys       29,  1 Map 25  2001 slice1
```

FIFO - именованный канал

Этот файл используется для связи между процессами по принципу очереди. *Именованные каналы* впервые появились в UNIX System V, но большинство современных систем поддерживают этот механизм.
В длинном листинге признаком именованного канала является символ **p** в первой позиции первого столбца:

```
[kravchuk@arturo 15:20:46 /tmp]$ find / -type p -print 2>/dev/null
/var/spool/lp/fifos/FIFO
/etc/cron.d/FIFO
/etc/saf/zsmon/_pmpipe
/etc/saf/_sacpipe
/etc/saf/_cmdpipe
/etc/initpipe
/etc/utmppipe
^C
[kravchuk@arturo 15:22:29 /tmp]$ ls -l /etc/cron.d/FIFO
prw-------   1 root     root           0 Фев 23 15:41 /etc/cron.d/FIFO
```

Именованные каналы создаются командой **mknod**:

mknod **имя** p

Например:

```
[kravchuk@arturo 15:27:17 /tmp]$ mknod p1 p
[kravchuk@arturo 15:27:18 /tmp]$ ls -l p* >p1 & cat p1
[2] 22380
prw-r--r--   1 kravchuk 50            0 Map 26 15:17 p1
-rw-rw-r--   1 root     sys        8296 Фев 23 15:39 ps_data
[2]-  Done                    ls -l p* >p1
[kravchuk@arturo 15:27:22 /tmp]$
```

Связь

Каталог содержит имена файлов и указатели на их метаданные. 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.
Стандарт POSIX (**Portable Operating System Interface**) требует реализовать поддержку двух типов связей - жестких и символических. *Жесткой связью* (hard link) считается элемент каталога, указывающий непосредственно на некоторый *индексный дескриптор*. Жесткие связи очень эффективны, но у них существуют определенные ограничения, так как они могут создаваться только в пределах одной физической файловой системы. Когда создается такая связь, связываемый файл должен уже существовать. Кроме того, каталоги не могут связываться жесткой связью.

Символическая связь (symbolic link) - это специальный файл, который содержит путь к другому файлу. Указание на то, что данный элемент каталога является символической связью, находится в индексном дескрипторе. Поэтому обычные команды доступа к файлу вместо получения данных из физического файла, берут их из файла, имя которого приведено в связи. Этот путь может указывать на что угодно: это может быть каталог, он может даже находиться в другой физической файловой системе, более того, указанного файла может и вовсе не быть.
Некоторые системы накладывают ограничение на количество символических связей в пути. POSIX требует, чтобы их поддерживалось не менее 20, но действительное значение зависит от конкретной реализации. Конечно, в описании пути можно использовать сочетание символических и жестких связей.
Количество жестких связей файла (а также количество файлов в каталоге, если файл является каталогом) отображается во втором поле длинного листинга:

```
[kravchuk@arturo 15:27:22 /tmp]$ ls >f2.txt
[kravchuk@arturo 15:37:22 /tmp]$ ln f3.txt f2.txt
ln: cannot access f3.txt
[kravchuk@arturo 15:37:31 /tmp]$ ln f2.txt f3.txt
[kravchuk@arturo 15:37:59 /tmp]$ ls -l f?.txt
-rw-r--r--   1 kravchuk 50            0 Map 26 14:40 f1.txt
-rw-r--r--   2 kravchuk 50          643 Map 26 15:37 f2.txt
-rw-r--r--   2 kravchuk 50          643 Map 26 15:37 f3.txt
[kravchuk@arturo 15:38:05 /tmp]$
```

В этом примере мы создали текстовый файл с листингом текущего каталога, а затем создали на него жесткую связь. Для этого используется команда **ln** со следующим синтаксисом:

ln [-fns] **исходный_файл** [**цель**]
ln [-fns] **исходный_файл** ... **цель**

Если в качестве цели указан несуществующий файл, или файл, не являющийся каталогом, используется первая форма. При этом количество операндов должно быть не более двух. В результате выполнения создается жесткая (по умолчанию) или символическая (если указана опция **-s**) связь с заданным именем **цель**. Если файл с таким именем уже существует, он перезаписывается. При вызове с одним аргументом создается связь на указанный **исходный_файл** с таким же именем в текущем каталоге.
Если **цель** задает существующий каталог, создается связь с таким же именем в этом каталоге. При наличии более двух аргументов используется вторая форма команды, причем **цель** должна ссылаться на существующий каталог.
Опции **-f** и **-n** требуют, соответственно, принудительно создать связь или не создавать ее, если **цель** задает существующий файл.
Обратите внимание, что первый аргумент команды **ln** должен указывать существующий файл или каталог.
В длинном листинге признаком символической связи является символ **l** в первой позиции первого столбца. Рассмотрим простой пример создания символической связи:

```
[kravchuk@arturo 15:57:41 /tmp]$ ln -s f2 f4
[kravchuk@arturo 15:57:50 /tmp]$ ls -l f*
-rw-r--r--   1 kravchuk 50            0 Map 26 14:40 f1.txt
-rw-r--r--   2 kravchuk 50          643 Map 26 15:37 f2.txt
-rw-r--r--   2 kravchuk 50          643 Map 26 15:37 f3.txt
lrwxrwxrwx   1 kravchuk 50            2 Map 26 15:57 f4 -> f2
```

Сокет

Сокеты позволяют представить в виде файла в логической файловой системе сетевое соединение. Создание сокетов выходит за пределы данного курса, хотя понятно, что для этого ядро предлагает соответствующий системный вызов.
В длинном листинге признаком сокета является символ **s** в первой позиции первого столбца. Вот какие сокеты можно найти в Solaris 8:

```
[kravchuk@arturo 15:38:05 /tmp]$ find / -type s -print 2>/dev/null
/var/spool/prngd/pool
/tmp/.X11-unix/X0
^C
[kravchuk@arturo 15:41:54 /tmp]$ ls -l /var/spool/prngd/pool
srwxrwxrwx   1 root     other          0 Map 14 11:25 /var/spool/prngd/pool
```

Определение типа файла

Для более точного определения типа файла (например, если файл двоичный, какой программой он мог быть создан) используется команда **file** со следующим синтаксисом:

file [-h] [-m **файл_сигнатур**] [-f **файл_списка**] **файл**...
file [-h] [-m **файл_сигнатур**] -f **файл_списка**
file -c [-m **файл_сигнатур**]

Утилита **file** выполняет ряд проверок каждого из указанных файлов и всех файлов, указанных в **файле_списка**, если он задан, пытаясь проклассифицировать файлы. Если файл не является обычным, выдается его тип. Если же обычный файл имеет нулевую длину, он классифицируется как пустой (**empty**).

Если файл является текстовым, команда **file** проверяет первых 512 байтов и пытается определить, на каком языке программирования написан файл. Если файл является символической связью, происходит проверка и классификация файла, на который связь указывает.
При определении типа файла используется файл сигнатур. Стандартный файл сигнатур - **/etc/magic**. В нем хранятся числа или строки, показывающие тип файла:

```
0      string      PK\003\004      ZIP archive
0      string      MZ              DOS executable (EXE)
0      string      LZ              DOS built-in
0      byte        0xe9            DOS executable (COM)
0      byte        0xeb            DOS executable (COM)
24     long        60012           ufsdump archive file
0      string      TZif            zoneinfo timezone data file
```

Формат файла сигнатур детально описан на странице справочного руководства **magic(4)**.
Если проверяемый файл не существует, не может быть прочитан или его тип не удается определить, это не считается ошибкой. Результат тестирования командой **file** не гарантирует 100% корректности. Не полагайтесь на него с полной уверенностью.
Поддерживаются следующие опции:

-c

Проверяет формат файла сигнатур.

-h

Не следует по символическим связям.

-f

Задает файл, содержащий список файлов для классификации.

-m

Задает альтернативный файл сигнатур, вместо **/etc/magic**.

Рассмотрим простой пример:

```
[kravchuk@arturo 16:05:40 /tmp]$ file -f f*
f2.txt:         ascii text
f3.txt:         ascii text
f4:             symbolic link to f2
```

Основные команды для работы с файлами

К основным операциям для работы с файлами, помимо создания и просмотра характеристик, можно отнести копирование, удаление, перемещение и переименование, а также просмотр содержимого. Команды для выполнения этих действий представлены в следующих подразделах.

Копирование файлов

Команда **cp** копирует исходный файл в целевой файл или каталог. Она имеет следующий синтаксис:

cp [-p] **исходный целевой**
cp [-r] [-p] **исходный**... **каталог**

Исходный файл не должен совпадать с целевым. Если целевой файл является каталогом, то исходные файлы копируются в него под теми же именами. Только в этом случае можно указывать несколько исходных файлов. Если целевой файл существует и не является каталогом, его старое содержимое теряется. Права доступа, владелец и группа целевого файла при этом не меняются.
Если целевой файл не существует или является каталогом, новые файлы создаются с теми же правами доступа, что и исходные. Время последнего изменения целевого файла (последнего доступа, если он не существовал), а также время последнего доступа к исходным файлам устанавливается равным времени копирования. Если целевой файл был связью на другой файл, все связи сохраняются, а содержимое файла изменяется.
Команда **cp** поддерживает следующие основные опции:

-p

сохраняет информацию о владельце, по возможности - права доступа и времена доступа для нового файла;

-r

копирует рекурсивно, включая подкаталоги.

Два дефиса (**--**) позволяют явно указать конец опций командной строки, что дает возможность команде **cp** работать с именами файлов, начинающимися с дефиса (**-**). Если в одной командной строке указаны **--** и **-**, второй дефис будет интерпретироваться как имя файла.

Рассмотрим ряд простых примеров копирования. Вот как, независимо от типа, копируется в каталог несколько файлов:

```
[kravchuk@arturo 16:53:35 /tmp]$ mkdir d1
[kravchuk@arturo 16:53:37 /tmp]$ rm f*
[kravchuk@arturo 16:54:08 /tmp]$ ls >f1.txt
[kravchuk@arturo 16:54:24 /tmp]$ ln f1.txt f2.txt
[kravchuk@arturo 16:54:39 /tmp]$ ln -s f1.txt f3.txt
[kravchuk@arturo 16:54:45 /tmp]$ cp f1.txt f2.txt f3.txt d1
[kravchuk@arturo 16:54:58 /tmp]$ ls -l d1
total 24
-rw-r--r--   1 kravchuk 50          639 Map 26 16:54 f1.txt
-rw-r--r--   1 kravchuk 50          639 Map 26 16:54 f2.txt
-rw-r--r--   1 kravchuk 50          639 Map 26 16:54 f3.txt
```

Вот пример обычного копирования файлов "один в один":

```
[kravchuk@arturo 16:55:22 /tmp]$ cp f1.txt f5.txt
[kravchuk@arturo 16:55:29 /tmp]$ ls -l f*.txt
-rw-r--r--   2 kravchuk 50          639 Map 26 16:54 f1.txt
-rw-r--r--   2 kravchuk 50          639 Map 26 16:54 f2.txt
lrwxrwxrwx   1 kravchuk 50            6 Map 26 16:54 f3.txt -> f1.txt
-rw-r--r--   1 kravchuk 50          639 Map 26 16:55 f5.txt
```

Рекурсивное копирование:

```
[kravchuk@arturo 16:55:34 /tmp]$ cp -r d1 d2
[kravchuk@arturo 16:56:47 /tmp]$ ls -l d2
total 24
-rw-r--r--   1 kravchuk 50          639 Map 26 16:56 f1.txt
-rw-r--r--   1 kravchuk 50          639 Map 26 16:56 f2.txt
-rw-r--r--   1 kravchuk 50          639 Map 26 16:56 f3.txt
```

Копирование файлов, имена которых начинаются с дефиса:

```
[kravchuk@arturo 16:56:50 /tmp]$ ls > -1
[kravchuk@arturo 16:58:44 /tmp]$ cp -1 1.txt
cp: illegal option -- 1
cp: Insufficient arguments (1)
Usage: cp [-f] [-i] [-p] f1 f2
       cp [-f] [-i] [-p] f1 ... fn d1
       cp -r|R [-f] [-i] [-p] d1 ... dn-1 dn
[kravchuk@arturo 16:58:48 /tmp]$ cp -- -1 1.txt
[kravchuk@arturo 16:58:53 /tmp]$ ls -l ?.txt
-rw-r--r--   1 kravchuk 50          666 Map 26 16:58 1.txt
```

Удаление файлов

Для удаления файлов используется команда **rm** со следующим синтаксисом:

rm [-firR] **файл**...

При этом происходит удаление записи файла из соответствующего каталога и уменьшение на 1 количества связей в индексном дескрипторе. Если количество связей в результате становится равным 0, файл уничтожается (после его закрытия всеми открывшими процессами) - соответствующий индексный дескриптор становится свободным, и блоки данных файла также освобождаются.

Для удаления файла пользователь должен обладать правом записи в соответствующий каталог. Если нет права на запись в файл и входной поток связан с терминалом, на терминал выдаются (в восьмеричном виде) права доступа к файлу и запрашивается подтверждение; если введен ответ **y** - файл удаляется, иначе - нет.

Команда **rm** воспринимает следующие основные опции:

-f

Удаляет без запросов подтверждения все файлы, независимо от прав доступа к ним, если имеется право записи для каталога.

-i

Запрашивает подтверждения, прежде чем удалить файл. Опция **-i** отменяет действие опции **-f**; она действует даже тогда, когда стандартный входной поток не связан с терминалом.

-r

Рекурсивное удаление, с подкаталогами, в том числе, не пустыми.

-R

То же, что и опция **-r**.

Команда **rm** без опций рекурсивного удаления не удаляет каталоги. Для удаления **пустых** каталогов предназначена команда **rmdir**. Если в каталоге есть другие файлы, кроме ссылок на текущий и родительский каталог, команда **rmdir** его не удаляет. Эта команда имеет следующий синтаксис:

rmdir [-p][-s] **каталог**...

Команда **rmdir** воспринимает следующие опции:

-p

Позволяет удалить каталог и его родительские каталоги, если они - пустые. В стандартный выходной поток выдается сообщение об удалении всех указанных каталогов или о сохранении части из них по каким-либо причинам.

-s

Подавляет выдачу сообщений при использовании опции **-p**.

Рассмотрим ряд примеров удаления файлов и каталогов (продолжая предыдущие примеры):

```
[kravchuk@arturo 17:23:09 /tmp]$ ls f* d*
dogovor_trg.sql  f1.txt          f3.txt
dtdbcache_:0     f2.txt          f5.txt

d1:
f1.txt  f2.txt  f3.txt

d2:
f1.txt  f2.txt  f3.txt
[kravchuk@arturo 17:23:17 /tmp]$ rm -r d1
[kravchuk@arturo 17:23:28 /tmp]$ rm f1.txt f2.txt
[kravchuk@arturo 17:23:47 /tmp]$ ls -l f*
lrwxrwxrwx   1 kravchuk 50           6 Map 26 16:54 f3.txt -> f1.txt
-rw-r--r--   1 kravchuk 50         639 Map 26 16:55 f5.txt
[kravchuk@arturo 17:23:51 /tmp]$ mkdir d2/d3
[kravchuk@arturo 17:24:12 /tmp]$ rm d2/*
rm: d2/d3 is a directory
[kravchuk@arturo 17:24:19 /tmp]$ ls -l d2
total 8
drwxr-xr-x   2 kravchuk 50          69 Map 26 17:24 d3
[kravchuk@arturo 17:24:26 /tmp]$ rmdir -p d2/d3
[kravchuk@arturo 17:25:24 /tmp]$ ls -l d2
d2: No such file or directory
```

Перемещение и переименование файлов

Команда **mv** перемещает (переименовывает) исходный файл (или файлы) в целевой файл (или каталог). Она имеет следующий синтаксис:

mv [-f][-i] **исходный_файл целевой_файл**
mv [-f][-i] **исходный_файл** ... **каталог**

Имя исходного файла не должно совпадать с именем целевого файла. Если целевой файл является каталогом, то исходные файлы перемещаются в него под теми же именами. Только в этом случае можно указывать несколько исходных файлов. Если целевой файл существует и не является каталогом, его старое содержимое теряется. Если при этом обнаруживается, что в целевой файл не разрешена запись, то выводится информация о правах доступа к этому файлу и с терминала запрашивается подтверждение его перезаписи. Для перемещения файла необходимо иметь права записи в исходном и целевом каталоге.

Команда **mv** поддерживает следующие опции:

-f

Принудительное перемещение - если целевой файл уже существует, то он удаляется.

-i

Запрашивает подтверждение удаления существующего файла.

Рассмотрим примеры:

```
[kravchuk@arturo 17:37:52 /tmp]$ ls -l f*
lrwxrwxrwx   1 kravchuk 50           6 Map 26 16:54 f3.txt -> f1.txt
-rw-r--r--   1 kravchuk 50         639 Map 26 16:55 f5.txt
[kravchuk@arturo 17:37:56 /tmp]$ mv f5.txt f4.txt
[kravchuk@arturo 17:38:09 /tmp]$ mv f4.txt f4.txt
mv: f4.txt and f4.txt are identical
[kravchuk@arturo 17:38:14 /tmp]$ mv f4.txt f3.txt
[kravchuk@arturo 17:38:20 /tmp]$ ls -l f*
-rw-r--r--   1 kravchuk 50         639 Map 26 16:55 f3.txt
[kravchuk@arturo 17:38:24 /tmp]$ mkdir d1
[kravchuk@arturo 17:38:54 /tmp]$ mv f3.txt d1
[kravchuk@arturo 17:39:00 /tmp]$ ls -l d1
total 8
-rw-r--r--   1 kravchuk 50         639 Map 26 16:55 f3.txt
```

Просмотр содержимого файлов

Стандартным средством просмотра содержимого файлов (помимо редакторов или команд типа **od**), является команда **cat**. Она читает файлы из командной строки в заданной последовательности и помещает их содержимое в стандартный выходной поток. Команда **cat** имеет следующий синтаксис:

cat [-u][-s][-v][-t][-e] [**файл** ...]

Если ни один файл не указан или указан символ дефиса (**-**), то команда читает стандартный входной поток.
Команда **cat** - полезный инструмент для *конкатенации* нескольких файлов.
Команда **cat** воспринимает следующие основные опции:

-u

Вывод не буферизуется (по умолчанию - буферизуется).

-s

Не сообщается о несуществующих файлах.

-v

Визуализация непечатных символов (кроме табуляций, переводов строк и переходов к новой странице). Управляющие символы изображаются в виде **^X** (CTRL+X); символ DEL (восьмеричное 0177) - в виде **^?**. Символы, не входящие в набор ASCII (то есть с ненулевым восьмым битом) выдаются в виде **M-x**, где **x** - определяемый младшими семью битами символ.

Рассмотрим несколько примеров использования команды **cat**:

```
[kravchuk@arturo 17:55:26 /tmp]$ ls *.txt > 1.txt
[kravchuk@arturo 17:55:36 /tmp]$ cat 1.txt
1.txt
[kravchuk@arturo 17:55:39 /tmp]$ cp 1.txt 2.txt
[kravchuk@arturo 17:55:48 /tmp]$ cat 1.txt 2.txt > 3.txt
[kravchuk@arturo 17:56:00 /tmp]$ ls -l *.txt
-rw-r--r--   1 kravchuk 50            6 Map 26 17:55 1.txt
-rw-r--r--   1 kravchuk 50            6 Map 26 17:55 2.txt
-rw-r--r--   1 kravchuk 50           12 Map 26 17:56 3.txt
[kravchuk@arturo 17:56:05 /tmp]$ cat 3.txt
1.txt
1.txt
[kravchuk@arturo 17:56:10 /tmp]$ cat >4.txt
Hello!
^D
[kravchuk@arturo 17:56:29 /tmp]$ cat 4.txt
Hello!
```

Права доступа к файлам

Каждый пользователь UNIX (не говоря уже о системном администраторе) должен управлять дисковым пространством. Пользователь несет ответственность за содержимое своего начального каталога и обеспечение целостности любых имеющихся у него данных. Целостность данных обеспечивается проверкой и изменением *прав доступа*. Защищая файлы и каталоги, пользователь предотвращает неавторизированный доступ.
Каждый файл в ОС UNIX содержит набор прав доступа, по которому определяется, как пользователь взаимодействует с данным файлом. Этот набор хранится в индексном дескрипторе данного файла в виде целого значения, из которого обычно используется 12 битов. Причем каждый бит используется как переключатель, разрешая (значение 1) или запрещая (значение 0) тот или иной доступ.
Три первых бита устанавливают различные виды поведения при выполнении. Оставшиеся девять делятся на три группы по три, определяя права доступа для владельца, группы и остальных пользователей. Каждая группа задает права на чтение, запись и выполнение.
Базовые биты прав доступа представлены в табл. 11. Там дано восьмеричное значение, задающее соответствующий бит, вид этого бита в первом столбце длинного листинга и право, задаваемое этим битом.

Таблица 11. Права доступа к файлам в ОС UNIX

Восьмеричное значение	**Вид в столбце прав доступа**	**Право или назначение бита**
4000	**---s------**	Установленный эффективный идентификатор владельца (бит SUID)
2000	**------s---**	Установленный эффективный идентификатор группы (бит SIID)
1000	**---------t** **---------T**	*Клейкий* (sticky) бит. Вид для каталогов и выполняемых файлов, соответственно.
0400	**-r--------**	Право владельца на чтение
0200	**--w-------**	Право владельца на запись
0100	**---x------**	Право владельца на выполнение
0040	**----r-----**	Право группы на чтение
0020	**-----w----**	Право группы на запись

0010	**------x---**	Право группы на выполнение
0004	**-------r--**	Право всех прочих на чтение
0002	**--------w-**	Право всех прочих на запись
0001	**---------x**	Право всех прочих на выполнение

Бит чтения для всех типов файлов имеет одно и то же значение: он позволяет читать содержимое файла (получать листинг каталога командой **ls**).
Бит записи также имеет одно и то же значение: он позволяет писать в этот файл, включая и перезапись содержимого. Если у пользователя отсутствует право доступа на запись в каталоге, где находится данный файл, то пользователь не сможет его удалить. Аналогично, без этого же права пользователь не создаст новый файл в каталоге, хотя может сократить длину доступного на запись файла до нуля.
Если для некоторого файла установлен бит выполнения, то файл может выполняться как команда. В случае установки этого бита для каталога, этот каталог можно сделать текущим (перейти в него командой **cd**).
Установленный бит SUID означает, что доступный пользователю на выполнение файл будет выполняться с правами (с *эффективным идентификатором*) владельца, а не пользователя, вызвавшего файл (как это обычно происходит).
Установленный бит SGID означает, что доступный пользователю на выполнение файл будет выполняться с правами (с *эффективным идентификатором*) группы-владельца, а не пользователя, вызвавшего файл (как это обычно происходит).
Если бит SGID установлен для файла, не доступного для выполнения, он означает обязательное *блокирование*, т.е. неизменность прав доступа на чтение и запись пока файл открыт определенной программой.
Установленный клейкий бит для обычных файлов ранее (**во времена PDP-11**) означал необходимость сохранить образ программы в памяти после выполнения (для ускорения повторной загрузки). Сейчас при установке обычным пользователем он сбрасывается. Значение этого бита при установке пользователем **root** зависит от версии ОС и иногда необходимо. Так, в ОС Solaris необходимо устанавливать клейкий бит для обычных файлов, используемых в качестве *области подкачки*.
Установка клейкого бита для каталога означает, что файл в этом каталоге может быть удален или переименован только в следующих случаях:

- пользователем-владельцем файла;
- пользователем-владельцем каталога;
- если файл доступен пользователю на запись;
- пользователем **root**.

Для расчета прав доступа необходимо сложить восьмеричные значения всех необходимых установленных битов. В результате получится четырехзначное восьмеричное число. Если старший разряд имеет значение 0, его можно не указывать.
Например, если необходимо задать права доступа на чтение, запись и выполнение для владельца, на чтение и выполнение для группы и на выполнение для всех остальных пользователей, получаем следующее восьмеричное значение:

Чтение для владельца:	**0400**
Запись для владельца:	**0200**
Выполнение для владельца:	**0100**
Чтение для группы:	**0040**
Выполнение для группы:	**0010**
Выполнение для прочих:	**0001**
Сумма:	**0751**

Итак, соответствующие права доступа - **751**. В длинном листинге эти права будут представлены в виде "**-rwxr-x--x**" (при "сложении" буквы с дефисом в символьном представлении остается буква).

Изменение прав доступа к файлу

Для установки (изменения) прав доступа к файлу используется команда **chmod**. Она имеет следующий синтаксис:

chmod [-fR] **абсолютные_права файл** ...
chmod [-fR] **символьное_изменение_прав файл** ...

Команда **chmod** устанавливает права доступа к указанным файлам. Права доступа к файлу может изменять или устанавливать только его владелец или пользователь **root**. Опция **-f** означает, что команда не будет сообщать о невозможности установки прав доступа. Опция **-R** означает, что заданное изменение прав доступа будет применяться рекурсивно для всех подкаталогов, указанных в списке файлов.
Абсолютные права доступа задаются восьмеричным числом, расчет которого (в соответствии с табл. 11) описан в предыдущем разделе. Описанию синтаксиса, используемого для задания символьного изменения прав доступа, посвящен следующий подраздел.

Символьное представление изменения прав доступа

Символьное изменение прав доступа задается в виде списка, через запятую, выражений следующего вида:

[**пользователи**] **оператор** [**права**]

Компонент пользователи определяет, для кого задаются или изменяются права. Он может иметь значения **u**, **g**, **o** и **a**, задающие изменения прав для владельца, группы, прочих пользователей и всех категорий пользователей. Если пользователи не указаны, права изменяются для всех категорий пользователей. Однако при этом не переопределяются установки, задаваемые маской создания файлов (**umask**).
Компонент **оператор** может иметь значения **+**, **-** или **=**, означающие добавление, отмену права доступа и установку в точности указанных прав, соответственно. Если после оператора = права не указаны, все права доступа для соответствующих категорий пользователей отменяются.
Компонент **права** задается в виде любой совместимой комбинации следующих символов:

r
: право на чтение

w
: право на запись

x
: право на выполнение

l
: блокирование изменения прав доступа

s
: выполнение с эффективным идентификатором владельца или группы-владельца

t
: клейкий бит

Не все сочетания символов для компонента пользователи и компонента права допустимы. Так, **s** можно задавать только для **u** или **g**, а **t** - только для **u**. Права **x** и **s** не совместимы с **l** и т.д.
Изменения прав доступа в списке выполняются последовательно, в порядке их перечисления.
Рассмотрим пример изменения прав доступа:

```
[kravchuk@arturo 10:51:43 /]$ cd /tmp
[kravchuk@arturo 10:51:46 /tmp]$ >f1.txt
[kravchuk@arturo 10:52:01 /tmp]$ chmod +w f1.txt
[kravchuk@arturo 10:52:13 /tmp]$ ls -l *.txt
-rw-r--r--   1 kravchuk 50            0 Map 27 10:52 f1.txt
[kravchuk@arturo 10:52:17 /tmp]$ chmod a+w f1.txt
[kravchuk@arturo 10:52:32 /tmp]$ ls -l *.txt
-rw-rw-rw-   1 kravchuk 50            0 Map 27 10:52 f1.txt
[kravchuk@arturo 10:52:33 /tmp]$ chmod u+x,g=x,o= f1.txt
[kravchuk@arturo 10:53:18 /tmp]$ ls -l *.txt
-rwx--x---   1 kravchuk 50            0 Map 27 10:52 f1.txt
[kravchuk@arturo 10:53:20 /tmp]$ chmod ug-x,og+r,u=rwx f1.txt
[kravchuk@arturo 10:54:46 /tmp]$ ls -l *.txt
-rwxr--r--   1 kravchuk 50            0 Map 27 10:52 f1.txt
[kravchuk@arturo 10:55:15 /tmp]$ chmod 644 f1.txt
[kravchuk@arturo 10:55:23 /tmp]$ ls -l *.txt
-rw-r--r--   1 kravchuk 50            0 Map 27 10:52 f1.txt
```

Рассмотрим еще один пример, показывающий значение и изменение прав доступа к каталогу:

```
[kravchuk@arturo 11:05:38 /tmp]$ ls -l | grep d1
drw-r--r--   2 kravchuk 50          108 Map 26 17:39 d1
[kravchuk@arturo 11:05:47 /tmp]$ cd d1
bash: cd: d1: Permission denied
[kravchuk@arturo 11:05:57 /tmp]$ chmod 744 d1
```

```
[kravchuk@arturo 11:06:26 /tmp]$ cd d1
[kravchuk@arturo 11:06:27 /tmp/d1]$ cd ..
[kravchuk@arturo 11:06:39 /tmp]$ chmod -w d1
[kravchuk@arturo 11:06:51 /tmp]$ cd d1
[kravchuk@arturo 11:06:58 /tmp/d1]$ ls
f3.txt
[kravchuk@arturo 11:06:59 /tmp/d1]$ rm f3.txt
rm: f3.txt not removed: Permission denied
```

Установка режима создания файла

Новый файл создается с правами доступа, определяемыми пользовательской маской режима создания файлов. Команда **umask** (встроенная команда интерпретатора) присваивает пользовательской маске режима создания файлов указанное восьмеричное значение. Три восьмеричные цифры соответствуют правам на чтение/запись/выполнение для владельца, членов группы и прочих пользователей, соответственно.
Команда **umask** имеет следующий синтаксис:

umask [-S] [**маска**]

Если параметры не указаны, команда **umask** выдает текущее значение маски. По умолчанию, значение выдается и задается в восьмеричном виде как число, которое необходимо "вычесть" из максимальных прав доступа (777 для выполняемых файлов, которые создаются компиляторами, и 666 для обычных файлов):

```
[kravchuk@arturo 11:22:55 /tmp/d1]$ umask
022
```

При такой маске обычные текстовые файлы будут создаваться с правами **666 - 022 = 644**:

```
[kravchuk@arturo 11:33:43 /tmp]$ >f5.txt
[kravchuk@arturo 11:33:48 /tmp]$ ls -l f5*
-rw-r--r--   1 kravchuk 50           0 Map 27 11:33 f5.txt
```

Операция "вычитания" для значения маски формально выполняется как побитовое логическое **И** дополнения маски и максимальных прав доступа. Рассмотрим пример расчета:

Двоичное значение маски:	**000010010**
Дополнение маски:	**111101101**
Максимальное значение прав:	**110110110**
Логическое И предыдущих двух строк:	**110100100**
Результирующие биты прав:	**110100100 (644)**

Для выполняемых файлов, создаваемых, например, компилятором языка C:

Двоичное значение маски:	**000010010**
Дополнение маски:	**111101101**
Максимальное значение прав:	**111111111**
Логическое И предыдущих двух строк:	**111101101**
Результирующие биты прав:	**111101101 (755)**

Опция **-S** требует выдавать маску в символьном виде, показывая, какие биты прав доступа будут установлены у создаваемого файла (также с учетом того, создается ли выполняемый или обычный текстовый файл):

```
[kravchuk@arturo 11:23:00 /tmp/d1]$ umask -S
u=rwx,g=rx,o=rx
```

Команду **umask** целесообразно включить в файлы начального запуска, задающие среду для начального командного интерпретатора.
Рассмотрим еще один пример создания файла при другом значении маски:

```
[kravchuk@arturo 11:33:52 /tmp]$ umask 257
[kravchuk@arturo 11:41:39 /tmp]$ umask -S
u=rx,g=w,o=
[kravchuk@arturo 11:41:43 /tmp]$ >f6.txt
[kravchuk@arturo 11:41:55 /tmp]$ ls -l f6*
-r---w----   1 kravchuk 50           0 Map 27 11:41 f6.txt
```

Изменение принадлежности файла

Владелец файла, а также пользователь **root** может изменять владельца и группу-владельца файла. Для изменения владельца (и группы-владельца) файла используется команда **chown** со следующим синтаксисом:

chown [-h][-R] **владелец**[:**группа**] **файл** ...

Опция **-h** требует изменять владельца файла, на который указывает символическая связь, а не самой связи, как происходит по умолчанию. Опция **-R** требует рекурсивно изменить владельца во всех подкаталогах.
Для изменения только группы, владеющей файлом, используется команда **chgrp**:

chgrp [-h][-R] **группа файл** ...

Ее опции аналогичны команде **chown**.
Учтите, что после передачи файла другому владельцу, первоначальный владелец перестает им обладать, и будет иметь права доступа, установленные новым владельцем.

Рассмотрим простой пример:

```
$ ls -l
total 2
-rw-r--r--    1 user01   others          6 Dec 10 16:19 testfile
$ chown informix testfile
$ ls -l
total 2
-rw-r--r--    1 informix others          6 Dec 10 16:19 testfile
$ logname
user01
$ chown user01 testfile
UX:chown: ERROR: testfile: Not privileged
```

Поиск файлов

В логической файловой системе ОС UNIX - тысячи файлов, поэтому необходимы средства поиска файлов по различным критериям. Для поиска файлов предназначена команда **find** со следующим синтаксисом:

find **каталог** ... **выражение**

Утилита **find** просматривает иерархии каталогов в поисках файлов, удовлетворяющих критерию, задаваемому выражением. Выражения строятся из элементов с использованием следующих конструкций:

(элемент)

Истинно, если истинен элемент в скобках (поскольку скобки - метасимвол командного интерпретатора, их надо экранировать). Скобки используются для группировки элементов.

! элемент

Истинно, если элемент не истинен.

элемент [-a] элемент

Истинно, если истинны оба элемента. Если элементы просто перечислены подряд, предполагается эта же логическая операция И.

элемент -o элемент

Истинно, если истинен хотя бы один элемент.

Имена найденных (удовлетворяющих критерию, задаваемому выражением) файлов по умолчанию выдаются в стандартный выходной поток.

В качестве элементов выражения используются основные конструкции, представленные в табл. 12.

Выражение проверяется **слева направо**, с учетом скобок.

Таблица 12. Основные элементы выражения в команде find

Элемент	Назначение или критерий истинности
-name шаблон	Истинен, если имя файла соответствует шаблону. При использовании метасимволов необходимо маскировать шаблоны от командного интерпретатора.
-type тип	Истинен, если файл - указанного типа. Типы файлов задаются символами **b**, **c**, **d**, **f**, **l**, **p** и **s**, обозначающими, соответственно, специальное блочное устройство, специальное символьное устройство, каталог, обычный файл, символьную связь, именованный канал и сокет.
-user пользователь	Истинен, если файл принадлежит **пользователю**, указанному по идентификатору или регистрационному имени.
-group группа	Истинен, если файл принадлежит **группе**, указанной по идентификатору или имени.
-perm [-]права	Если дефис не задан, то истинен только если права доступа в точности соответствуют указанным (как в команде **chmod**, проще - абсолютные). Если задан дефис, истинен, если в правах доступа файла, как минимум, установлены те же биты, что и в указанных правах.
-size n[c]	Истинен, если файл имеет длину **n** блоков (блок - 512 байтов) или символов (если указан суффикс **c**). Перед размером можно указывать префикс + (не меньше), - (не больше) или = (в точности равен).
-atime n	Истинен, если к файл последний раз обращались **n** дней назад. Перед **n** в элементах **-atime**, **-ctime** и **-mtime** можно указывать префикс + (не позже), - (не ранее) или = (ровно).
-ctime n	Истинен, если файл создан **n** дней назад.
-mtime n	Истинен, если файл был изменен **n** дней назад.

-newer файл	Истинен, если файл - более новый, чем указанный.
-ls	Всегда истинен. Выдает информацию о файле, аналогичную длинному листингу.
-print	Истинен всегда. Выдает полное имя файла в стандартный выходной поток.
-exec команда {} \;	Истинен, если выполненная **команда** возвращает код возврата 0. Команда заканчивается замаскированной точкой с запятой. В команде можно использовать конструкцию **{}**, заменяемую полным именем рассматриваемого файла.
-ok команда {} \;	Аналогичен **exec**, но полученная после подстановки имени файла вместо **{}** команда выдается с вопросительным знаком и выполняется только если пользователь ввел символ **y**.
-depth	Истинен всегда. Требует так обходить иерархию каталогов, чтобы файлы любого каталога всегда обрабатывались раньше, чем сам каталог (обход "в глубину").
-prune	Истинен всегда. Требует не проверять файлы в каталоге, сопоставившемся с предыдущим элементом выражения. Не действует, если ранее указан элемент **-depth**.

В различных версиях ОС UNIX могут поддерживаться и другие компоненты выражений в команде **find**.
Если командная строка сформирована неправильно, команда немедленно завершает работу.
Рассмотрим несколько примеров использования команды **find**:

```
[kravchuk@arturo 15:05:25 /tmp]$ find . -user kravchuk -size +0c -ls
find: cannot read dir ./smc898: Permission denied
475898122    4 -rw-r--r--  1 kravchuk 50          666 Mar 26 16:58 ./-1
473866040    4 -rw-r--r--  1 kravchuk 50            6 Mar 26 17:55 ./1.txt
475472259    4 dr-xr--r--  2 kravchuk 50          108 Mar 26 17:39 ./d1
474199552    4 -rw-r--r--  1 kravchuk 50          639 Mar 26 16:55
./d1/f3.txt
476732956    4 -rw-r--r--  1 kravchuk 50            6 Mar 26 17:55 ./2.txt
476732980    4 -rw-r--r--  1 kravchuk 50           12 Mar 26 17:56 ./3.txt
476142563    4 -rw-r--r--  1 kravchuk 50            7 Mar 26 17:56 ./4.txt
[kravchuk@arturo 15:26:41 /tmp]$ find . -name "??.txt" -print
find: cannot read dir ./smc898: Permission denied
./d1/f3.txt
./f1.txt
[kravchuk@arturo 15:26:58 /tmp]$ find . -name d1 -prune -name "??.txt" -print
find: cannot read dir ./smc898: Permission denied
[kravchuk@arturo 15:27:09 /tmp]$ find . -name d1 -prune -o -name "??.txt" -print
find: cannot read dir ./smc898: Permission denied
./f1.txt
[kravchuk@arturo 15:27:13 /tmp]$ find . -user kravchuk -ok rm {} \;
find: cannot read dir ./smc898: Permission denied
< rm ... ./p1 >?   y
< rm ... ./-1 >?   y
< rm ... ./1.txt >?   y
< rm ... ./mpDfa4ZT >?   y
< rm ... ./d1 >?   y
rm: Unable to remove directory ./d1: File exists
< rm ... ./d1/f3.txt >?   y
< rm ... ./2.txt >?   y
< rm ... ./3.txt >?   y
< rm ... ./4.txt >?   y
< rm ... ./f1.txt >?   y
[kravchuk@arturo 15:28:35 /tmp]$ find . -user kravchuk -print
find: cannot read dir ./smc898: Permission denied
./d1
./d1/f3.txt
```

Структура и свойства файловых систем

Логическая файловая система - основные каталоги и их назначение

Использование общепринятых имен основных файлов и структуры каталогов существенно облегчает работу в операционной системе, ее администрирование и повышает переносимость. Типичная структура и назначение каталогов файловой системы UNIX представлена в табл. 13.

Таблица 13. Основные каталоги логической файловой системы UNIX

Каталог	Назначение и содержание
/	Корневой каталог. Является основой любой файловой системы UNIX. Все остальные каталоги и файлы располагаются в рамках структуры, порожденной корневым каталогом (в нем и в его подкаталогах), независимо от их физического местонахождения. Для корневого каталога обязательно должна создаваться **отдельная** физическая файловая система, а сам он является точкой ее *монтирования*, о чем свидетельствует наличие подкаталога **lost+found**.
/bin	Пользовательские выполняемые программы. Сейчас обычно является символической связью, указывающей на **/usr/bin**.
/dev	Каталог для специальных файлов устройств. Может иметь подкаталоги для различных классов и типов устройств, например, **dsk**, **rdsk**, **rmt**, **inet** (в SVR4).
/etc	Каталог для конфигурационных файлов. Может иметь подкаталоги для различных компонентов и служб. Конфигурационные файлы в UNIX - обычные текстовые.
/home	Каталог для размещения начальных каталогов пользователей. Часто является точкой монтирования отдельной физической файловой системы.
/lib	Каталог для библиотек. Сейчас обычно является символической связью, указывающей на **/usr/lib**.
/lost+found	Подкаталог, имеющийся в каждом каталоге, являющемся точкой монтирования физической файловой системы на диске. Корневой каталог всегда представлен отдельной физической файловой системой, должен быть всегда доступен, и монтируется автоматически при запуске системы. Все остальные физические файловые системы формально не нужны для функционирования ОС UNIX.
/mnt	Точка монтирования для файловых систем на съемных носителях или дополнительных дисках. Может содержать подкаталоги для отдельных типов носителей, например, **cdrom** или **floppy**. Может юыть пустым.
/opt	Каталог для дополнительного коммерческого программного обеспечения. Может быть пустым или отсутствовать (в BSD-системах).
/proc	Каталог псевдо-файловой системы, представляющей в виде каталогов и файлов информацию о ядре, памяти и процессах, работающих в системе.
/sbin	Каталог для системных выполняемых программ, необходимых для решения задач системного администрирования.
/tmp	Каталог для временных файлов. Имеет установленный *клейкий бит* и доступен для записи и чтения всем пользователям. Обычно создается в виде отдельной физической файловой системы, в том числе, в виртуальной памяти.
/usr	В этом каталоге находятся выполняемые программы, библиотеки, заголовочные файлы, справочные руководства (**/usr/share/man**), исходные тексты ядра и утилит системы (Linux), растущие файлы и очереди печати (**/usr/spool** в BSD-системах) и т.д. Часто каталог является точкой монтирования отдельной физической файловой системы. Ниже представлены основные его подкаталоги.
/usr/bin	Основные выполняемые программы и утилиты.
/usr/include	Заголовочные файлы библиотек. Может содержать подкаталоги.
/usr/lib	Статически и динамически компонуемые библиотеки. Может содержать подкаталоги.
/usr/local	Каталог для дополнительного *свободно распространяемого программного обеспечения* (GNU). Содержит структуру подкаталогов, аналогичную корневому каталогу (**bin**, **etc**, **include**, **lib** и т.д.).
/var	В UNIX System V и Linux этот каталог является заменителем каталога (**/usr/spool**), используемого для хранения растущих файлов различных сервисных подсистем, например, файлов журналов системы. Так, основной журнал системы, ведущийся демоном **syslogd**, размещается в виде нескольких файлов в подкаталоге **/var/adm**. Там же, в файле **/var/adm/messages**, сохраняются сообщения времени загрузки. Имеет смысл создавать отдельную физическую файловую систему для размещения этого каталога (и, возможно, его подкаталога **/var/run**).

Наличие, назначение и использование других каталогов верхнего уровня и подкаталогов зависит от версии ОС UNIX, установленного системного и прикладного программного обеспечения и конфигурации системы, созданной администратором.

Физические файловые системы UNIX - основные компоненты

Каждый жесткий диск состоит из одной или нескольких логических частей (групп цилиндров), называемых *разделами* (partitions). Расположение и размер раздела определяется при форматировании диска. В ОС UNIX разделы выступают в качестве независимых устройств, доступ к которым осуществляется как к различным носителям данных. Обычно в разделе может располагаться только одна *физическая файловая система*. Имеется много типов физических файловых систем, например FAT16 и NTFS, с разной структурой. Более того, имеется множество типов физических файловых систем UNIX (**ufs**, **s5fs**, **ext2**, **vxfs**, **jfs**, **ffs** и т.д.). Ниже мы рассмотрим основные их общие особенности.
Физическая файловая система UNIX занимает раздел диска и состоит из таких основных компонентов:

- *Суперблок* (superblock). Содержит общую информацию о файловой системе.

- Массив *индексных дескрипторов* (ilist). Содержит метаданные всех файлов файловой системы. *Индексный дескриптор* (inode) содержит информацию о статусе файла и указывает на расположение данных этого файла. Ядро обращается к индексному дескриптору по индексу в массиве. Один дескриптор является корневым для физической файловой системы, через него обеспечивается доступ к структуре каталогов и файлов после монтирования файловой системы. Размер массива индексных дескрипторов является фиксированным и задается при создании физической файловой системы.

- Блоки хранения данных. Данные обычных файлов и каталогов хранятся в блоках. Обработка файла осуществляется через индексный дескриптор, содержащий ссылки на блоки данных.

Суперблок

Суперблок содержит информацию, необходимую для *монтирования* и управления файловой системой в целом. В каждой файловой системе существует только один суперблок, который располагается в начале раздела. Суперблок считывается в память ядра при монтировании файловой системы и находиться там до ее отключения - *демонтирования*.
Суперблок содержит:

- тип файловой системы;

- размер файловой системы в логических блоках, включая сам суперблок, массив индексных дескрипторов и блоки хранения данных;

- размер массива индексных дескрипторов;

- количество свободных блоков;

- количество свободных индексных дескрипторов;

- флаги;

- размер логического блока файловой системы (512, 1024, 2048, 4096, 8192).

- список номеров свободных индексных дескрипторов;

- список адресов свободных блоков.

Поскольку количество свободных индексных дескрипторов и блоков хранения данных может быть значительным, хранение двух последних списков целиком в суперблоке непрактично. Для индексных дескрипторов храниться только часть списка. Когда число свободных дескрипторов приближается к 0, ядро просматривает список и вновь формирует список свободных дескрипторов.
Такой подход неприемлем в отношении свободных блоков хранения данных, поскольку по содержимому блока нельзя определить, свободен он или нет. Поэтому необходимо хранить список адресов свободных блоков целиком. Список адресов свободных блоков может занимать несколько блоков хранения данных, но суперблок содержит только один блок этого списка. Первый элемент этого блока указывает на блок, хранящий продолжение списка.
Выделение свободных блоков для размещения файла производиться с конца списка суперблока. Когда в списке остается 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. Такой подход позволяет использовать дисковое пространство под списки, пропорциональное свободному месту в файловой системе. Когда свободного места практически не остается, список адресов свободных блоков целиком помещается в суперблоке.

Индексные дескрипторы

Индексный дескриптор, или **inode**, содержит информацию о файле, необходимую для обработки данных, т.е. метаданные файла. Каждый файл ассоциирован с одним индексным дескриптором, хотя может иметь несколько имен (*жестких связей*) в файловой системе, каждое из которых указывает на один и тот же индексный дескриптор.
Индексный дескриптор не содержит:

- имени файла, которое содержится в блоках хранения данных каталога;
- содержимого файла, которое размещено в блоках хранения данных.

Индексный дескриптор содержит:

- номер;
- тип файла;
- права доступа к файлу;
- количество связей (ссылок на файл в каталогах) файла;
- идентификатор пользователя и группы-владельца;
- размер файла в байтах;
- время последнего доступа к файлу;
- время последнего изменения файла;

- время последнего изменения индексного дескриптора файла;
- указатели на блоки данных файла (обычно 10);
- указатели на косвенные блоки (обычно 3).

Размер индексного дескриптора обычно составляет 128 байтов.
Индексный дескриптор содержит информацию о расположении данных файла. Поскольку дисковые блоки хранения данных, в общем случае, располагаются не последовательно, индексный дескриптор должен хранить физические адреса всех блоков, принадлежащих данному файлу.
Каждый дескриптор содержит 13 указателей. Первые 10 указателей непосредственно ссылаются на блоки данных файла. Если файл большего размера - 11-ый указатель ссылается на первый *косвенный блок* (indirection block) из 128 (256) ссылок на блоки данных. Если и этого недостаточно, 12-ый указатель ссылается на *дважды косвенный блок*, содержащий 128 (256) ссылок на косвенные блоки. Наконец последний, 13-ый указатель ссылается на *трижды косвенный блок* из 128 (256) ссылок на дважды косвенные блоки. Количество элементов в косвенном блоке зависит от его размера.
Поддерживая множественные уровни косвенности, индексные дескрипторы позволяют отслеживать огромные файлы, не растрачивая дисковое пространство для небольших файлов.

Синхронизация структуры файловой системы

При открытии файла ядро помещает копию дискового индексного дескриптора в соответствующую таблицу в памяти, которая содержит дополнительные атрибуты. В дальнейшем изменение индексного дескриптора происходит в памяти, и измененная структура файловой системы сбрасывается на диск только при выполнении специальной команды, **sync**. Эта команда выполняется при штатной остановке системы или явно администратором.
Если произошло нештатное прекращение работы системы, структура суперблока и массива индексных дескрипторов на диске не соответствует структуре блоков данных и может быть несогласованной.
Отсутствие синхронизации между образом файловой системы в памяти и ее данными на диске (в случае аварийной остановки системы) может привести к появлению следующих ошибок в файловой системе:

1. Один блок адресуется несколькими дескрипторами (принадлежит нескольким файлам).
2. Блок помечен как свободный, но в тоже время занят (на него ссылается дескриптор).
3. Блок помечен как занятый, но в то же время свободен (ни один дескриптор на него не ссылается).
4. Неправильное количество ссылок в дескрипторе.
5. Несовпадение между размером файла и суммарным размером адресуемых дескриптором блоков.
6. Недопустимые адресуемые блоки (например, расположенные за пределами файловой системы).
7. "Потерянные" файлы (правильные дескрипторы, на которые не ссылаются записи каталогов).
8. Недопустимые номера дескрипторов в записях каталогов.

Часть этих проблем может быть устранена специальной утилитой, **fsck** (см. далее в разделе, посвященном управлению файловой системой). Но принципиальное решение проблемы согласованности и целостности

данных в файловых системах UNIX возможно только при использовании *журнализации* - предварительной записи всех изменений дисковой структуры в отдельную область на диске.

Журнализируемые файловые системы

В журнализируемой файловой системе после того, как *транзакция* (изменение) записана, она может быть выполнена повторно, что предотвращает возникновение ошибок и несогласованностей в файловой системе и необходимость запуска программы **fsck**. Тем самым, уменьшается время перезагрузки в случае сбоя или некорректной остановки системы.
Журнал выделяется из свободных блоков файловой системы и, обычно, имеет размер порядка 1 Мбайта на каждый 1 Гбайт файловой системы. Журнал сбрасывается по мере заполнения, после синхронизации структуры файловой системы с диском.
Различные версии ОС UNIX поддерживают разные реализации журнализируемых файловых систем. Это, например, файловая система **ufs** (Solaris), **vxfs** (Solaris, UnixWare), **RaisorFS** и **ext3** (Linux), **jfs** (AIX и Linux) и другие. Некоторые файловые системы позволяют включать и отключать журнализацию (**ufs**, **ext2/ext3**). Естественно, журнализация несколько замедляет работу файловой системы, но, в большинстве случаев, гарантирует целостность данных.

Управление файловой системой

Основными задачами администрирования файловых систем являются создание, монтирование и демонтирование физических файловых систем, а также проверка их целостности. В следующих подразделах мы рассмотрим соответствующие команды и обобщенно опишем выполняемые ими действия.

Создание физической файловой системы

Команда **mkfs** создает файловую систему путем записи на указанное устройство (необходимо указать специальное символьное устройство). Файловая система создается на основе указанных в командной строке типа файловой системы (**ТипФС**), **специфических_опций** и **операндов**. Команда имеет следующий синтаксис:

mkfs [-F **ТипФС**][-V][-m] [-o **специфические_опции**]
устройство размер [**операнды**]

Специфические опции и операнды зависят от конкретного типа создаваемой файловой системы. Их можно посмотреть на соответствующей странице справочного руководства (например, **man mkfs_ufs** для файловой системы **ufs**).
Основные опции и параметры команды **mkfs** представлены в табл. 14.

Таблица 14. Основные опции и параметры команды mkfs

Опция	Назначение
-F	Указывает тип файловой системы, которую необходимо создать. Тип файловой системы должен быть либо указан здесь, либо находится в файле таблицы стандартных файловых систем (**/etc/vfstab** в SVR4, **/etc/fstab** в других версиях UNIX) путем сопоставления устройства с записью в таблице.
-V	Выдает результирующую командную строку, но не выполняет команду. Командная строка генерируется с использованием опций и аргументов, указанных пользователем, путем добавления к ним информации, взятой из таблицы стандартных файловых систем. Эта опция используется для проверки правильности командной строки.
-m	Возвращает командную строку, использованную для создания файловой системы. Файловая система должна уже существовать. Эта опция обеспечивает средства получения команды, использованной при создании файловой системы. Для нее не применимы **специфические_опции**, **размер** и **операнды**.
-o	Задает опции, специфические для указанного типа физической файловой системы.
устройство	Задает специальное символьное устройство, на котором будет создана файловая система.
размер	Задает количество 512-байтовых блоков в файловой системе. Максимальный размер многих физических файловых систем в UNIX - 4194304 блока размером 512 байт (или 2 Гбайта).

Проверка и восстановление целостности файловых систем

Программа **fsck** ищет и, автоматически или в интерактивном режиме, исправляет противоречия в файловых системах. Если файловая система находится в несогласованном состоянии, которое нельзя однозначно

исправить, у пользователя спрашивают подтверждения перед попыткой выполнить каждое исправление. Следует иметь в виду, что некоторые исправления приводят к определенным потерям данных. Объем и серьезность потери данных можно определить по диагностическому сообщению. Стандартным действием при каждом исправлении является ожидание от пользователя утвердительного (**yes**) или отрицательного (**no**) ответа.

При использовании **fsck** файловая система должна быть неактивной (размонтирована или смонтирована только для чтения). Если это невозможно, необходимо обеспечить, чтобы машина находилась в состоянии покоя (без работающих пользователей) и чтобы сразу после завершения команды она была перезагружена, если исправляется критическая файловая система, например, корневая.

Команда fsck имеет следующий синтаксис:

fsck [-F **ТипФС**] [-V] [-m] [**устройство** ...]
fsck [-F **ТипФС**] [-V] [-o **специфические_опции**] [**устройство** ...]

Основные опции и параметры команды **fsck** представлены в табл. 15.

Таблица 15. Основные опции команды fsck

Опция	Назначение
-F	Задает тип проверяемой файловой системы. Если тип не указан, команда обращается к таблице стандартных файловых систем.
-V	Выдает результирующую командную строку, но не выполняет команду. Командная строка генерируется с использованием опций и аргументов, указанных пользователем, путем добавления к ним информации, взятой из таблицы стандартных файловых систем.
-m	Проверять, но не восстанавливать. Эта опция позволяет проверить, может ли файловая система быть смонтирована.
-o	Позволяет задать опции, специфические для типа файловой системы.

Для работы команде **fsck** необходимо указывать специальное символьное устройство.

Корневая файловая система обычно проверяется при запуске автоматически. Система при запуске может автоматически проверять и другие физические файловые системы, для которых в таблице стандартных файловых систем указана необходимость такой проверки. Эта проверка может вестись параллельно, путем запуска отдельного процесса **fsck** для каждой проверяемой файловой системы с одним и тем же порядковым номером проверки. Параллельно имеет смысл проверять файловые системы, расположенные на разных физических дисках.

Монтирование и демонтирование физических файловых систем

Физические файловые системы, кроме корневой (/), считаются *съемными* (removable) в том смысле, что они могут быть как доступны для пользователей, так и не доступны. Команда **mount** уведомляет систему, что блочное устройство или удаленный ресурс доступны для пользователей в **точке_монтирования**, которая уже должна существовать; точка монтирования становится именем корня вновь смонтированного устройства или ресурса. Говорят, что эта команда *монтирует* или подключает физическую файловую систему или ресурс к общей логической файловой системе.

Команда **mount** имеет следующий синтаксис:

mount [-v | -p]
mount [-F **ТипФС**] [-V] [-o **специфические_опции**]
{**устройство**|**точка_монтирования**}
mount [-F **ТипФС**] [-V] [-o **специфические_опции**]
устройство точка_монтирования}

Команда **mount**, при вызове с аргументами, проверяет все аргументы, за исключением устройства, и вызывает специфический модуль монтирования для указанного типа файловой системы. При вызове без аргументов **mount** выдает список всех смонтированных файловых систем из соответствующей таблицы. При вызове с неполным списком аргументов (например, только с указанием **устройства** или **точки_монтирования**, или когда указаны оба эти аргумента, но не задан тип файловой системы), **mount** будет просматривать таблицу стандартных файловых систем в поисках недостающих аргументов. Затем она вызывает специфический модуль монтирования для соответствующего типа файловой системы.

Специфические опции монтирования зависят от типа физической файловой системы. Все физические файловые системы можно монтировать только для чтения (**-o ro**).

Обратная процедура по отношению к монтированию называется *демонтированием* и выполняется командой **umount** со следующим синтаксисом:

umount [-V] [-o **специфические_опции**]
{**устройство**|**точка_монтирования**}

Для большинства типов файловых систем нет специфического модуля демонтирования. Если такой модуль существует, он выполняется; иначе файловая система демонтируется стандартным модулем.

Команды **mount** и **umount** воспринимают следующие основные опции:

-v

Выдает результаты в "новом" стиле. При этом дополнительно отображается тип файловой системы и флаги. Поля **точка_монтирования** и **устройство** переставлены.

-p

Выдает список смонтированных файловых систем в формате таблицы смонтированных файловых систем.

-F

Задает тип файловой системы для монтирования. Тип файловой системы должен быть либо задан, либо определяется по таблице стандартных файловых систем в ходе монтирования.

-V

Выдает результирующую командную строку, но не выполняет команду. Командная строка генерируется с использованием опций и аргументов, указанных пользователем, путем добавления к ним, при необходимости, информации, взятой из таблицы стандартных файловых систем.

-o

Задает специфические опции для указанного типа физической файловой системы.

Любой пользователь может вызывать команду **mount** для получения списка смонтированных файловых систем и ресурсов. Например:

```
[kravchuk@arturo 13:05:48 /]$ mount -p
/dev/dsk/c1t0d0s0 - / ufs - no
rw,intr,largefiles,logging,onerror=panic,suid,dev=740040
/dev/dsk/c1t0d0s3 - /usr ufs - no
rw,intr,largefiles,logging,onerror=panic,suid,dev=740043
/dev/dsk/c1t0d0p0:boot - /boot pcfs - no rw,nohidden,nofoldcase,dev=763050
/proc - /proc proc - no dev=2c00000
fd - /dev/fd fd - no rw,suid,dev=2cc0000
mnttab - /etc/mnttab mntfs - no dev=2dc0000
/dev/dsk/c1t0d0s1 - /var ufs - no
rw,intr,largefiles,logging,onerror=panic,suid,dev=740041
swap - /var/run tmpfs - no dev=1
swap - /tmp tmpfs - no dev=2
/dev/dsk/c1t0d0s4 - /home ufs - no
rw,intr,largefiles,logging,onerror=panic,suid,dev=740044
/dev/dsk/c2t0d0s1 - /fs ufs - no
rw,intr,largefiles,logging,onerror=panic,suid,dev=740401
```

Только пользователь **root** может монтировать или демонтировать файловые системы.

Таблица смонтированных файловых систем

Команда **mount** по умолчанию добавляет запись в *таблицу смонтированных файловых систем* (файл **/etc/mnttab** в SVR4); **umount** удаляет запись из этой таблицы. Поля в таблице смонтированных устройств разделены пробелами и представляют блочное специальное устройство, точку монтирования, тип смонтированной файловой системы, опции монтирования и время, когда файловая система была смонтирована.

Таблица стандартных файловых систем

Таблица стандартных файловых систем (в файле **/etc/vfstab** или **/etc/fstab**, в зависимости от разновидности UNIX) описывает стандартные параметры для физических файловых систем. Поля в таблице (их 7) разделены пробелами и символами табуляции, и представляют, соответственно:

- специальное блочное устройство или имя монтируемого ресурса;
- неформатированное (специальное символьное) устройство для проверки утилитой **fsck**;
- стандартный каталог монтирования;
- тип файловой системы;

- число, используемое **fsck** для принятия решения об автоматической проверке файловой системы и о порядке этой проверки по отношению к другим файловым системам;

- признак автоматического монтирования файловой системы;

- опции монтирования.

Если в поле нет значения, используется дефис (**-**). Рассмотрим пример записей из таблицы стандартных файловых систем из ОС Solaris 8:

```
#device         device               mount   FS      fsck  mount   mount
#to mount       to fsck              point   type    pass  at boot options
/dev/dsk/c1t0d0s0 /dev/rdsk/c1t0d0s0     /       ufs     1     no      logging
/dev/dsk/c1t0d0s3 /dev/rdsk/c1t0d0s3     /usr    ufs     1     no      logging
/dev/dsk/c1t0d0s1 /dev/rdsk/c1t0d0s1     /var    ufs     1     no      -
/dev/dsk/c1t0d0s4 /dev/rdsk/c1t0d0s4     /home   ufs     2     yes     logging
...
```

Получение информации о файловых системах

Для получения информации о смонтированных физических файловых системах используется команда **df** со следующим синтаксисом:

```
df [ -F ТипФС ] [ -abegklntV ] [ -o специфические_опции ]
  [ устройство | каталог | файл | ресурс ... ]
```

Опции и параметры определяют формат выдаваемой информации и файловые системы, о которых информирует команда. Чаще всего, команда **df** вызывается без опций или с опцией **-k**. Опция **-k** выдает информацию об объемах в килобайтах. Для каждой физической файловой системы выдается отдельная строка, включающая (при использовании опции **-k**) специальный файл или имя смонтированного ресурса, общий объем, использованный объем, доступный объем для использования обычными пользователями, процент свободного места в файловой системе и точку монтирования.
Рассмотрим примеры выполнения команды **df** в ОС Solaris:

```
[kravchuk@arturo 12:11:00 /]$ df -k
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c1t0d0s0     245983   20713  200672    10%    /
/dev/dsk/c1t0d0s3    3096090 1782106 1252063    59%    /usr
/dev/dsk/c1t0d0p0:boot
                       10797    1622    9175    16%    /boot
/proc                      0       0       0     0%    /proc
fd                         0       0       0     0%    /dev/fd
mnttab                     0       0       0     0%    /etc/mnttab
/dev/dsk/c1t0d0s1     491983  204863  237922    47%    /var
swap                  324832      16  324816     1%    /var/run
swap                  337828   13012  324816     4%    /tmp
/dev/dsk/c1t0d0s4    2305873 1021225 1238531    46%    /home
/dev/dsk/c2t0d0s1    6192197 5633827  496449    92%    /fs
[kravchuk@arturo 12:45:58 /]$ df
/                 (/dev/dsk/c1t0d0s0 ):  450540 blocks   120616 files
/usr              (/dev/dsk/c1t0d0s3 ): 2627968 blocks   338652 files
/boot             (/dev/dsk/c1t0d0p0:boot):   18350 blocks       -1 files
/proc             (/proc             ):       0 blocks     3615 files
/dev/fd           (fd                ):       0 blocks        0 files
/etc/mnttab       (mnttab            ):       0 blocks        0 files
/var              (/dev/dsk/c1t0d0s1 ):  574236 blocks   240784 files
/var/run          (swap              ):  647568 blocks    43108 files
/tmp              (swap              ):  647568 blocks    43108 files
/home             (/dev/dsk/c1t0d0s4 ): 2569298 blocks   379999 files
/fs               (/dev/dsk/c2t0d0s1 ): 1116738 blocks   688872 files
```

Управление процессами

Основным ресурсом компьютера является его процессор (или процессоры). В каждый момент времени один процессор может выполнять только один процесс. Организация планирования процессов так, чтобы за счет их переключения создавалась **иллюзия** одновременной работы нескольких процессов - одна из основных задач любой многопользовательской и многозадачной операционной системы.
В ОС UNIX основным средством организации и единицей многозадачности является *процесс* - уникальным образом идентифицируемая программа, которая нуждается в получении доступа к ресурсам компьютера. Операционная система манипулирует образом процесса, который представляет собой программный код, а

также разделами данных процесса, определяющими среду выполнения. Сегмент кода содержит реальные инструкции процессора, включающие как строки, скомпилированные и написанные пользователем, так и стандартный код, сгенерированный компилятором для системы. Этот системный код обеспечивает взаимодействие между программой и операционной системой.
Основой операционной системы UNIX является ядро. *Ядро* представляет собой специальную программу (или несколько программных модулей, в случае *модульного ядра*), которая постоянно находится в оперативной памяти и работает, пока работает операционная система. Ядро управляет всеми таблицами, используемыми для отслеживания процессов и других ресурсов. Ядро загружается в память во время начальной загрузки и немедленно запускает необходимые процессы, в частности процесс инициализации операционной системы - **init**.
Данные, связанные с процессом, также являются частью образа процесса. Некоторые из них хранятся в регистрах, обычно представленных регистрами процессора. Кроме того, существуют динамические области хранения данных (*куча*), выделяемые процессом по ходу работы при необходимости.
Еще у процесса есть *стек*, содержащийся в памяти и используемый для хранения локальных переменных программы и передачи параметров. Когда процесс выполняет обращение к функции или подпрограмме, в стек отправляется новый *фрейм*. Одной из частей каждого фрейма является указатель на базу предыдущего фрейма, который позволяет легко вернуться из вызова функции. При этом важно знать местоположение текущего фрейма и вершину стека.
Регистры играют важную роль в работе процессов. Обычно выделяется четыре регистра, имеющих специальное значение:

Регистр	**Назначение**
PC	Программный счетчик - указывает на текущую строку кода.
PS	Указывает состояние процессора.
SP	Указывает на вершину стека.
FP	Указывает на текущий фрейм стека.

Во время исполнения или в ожидании "своего часа" процессы содержатся в виртуальной памяти со страничной организацией. Часть этой виртуальной памяти сопоставляется с физической. Часть физической памяти резервируется для ядра операционной системы. Пользователи могут получить доступ только к оставшейся для процессов памяти. При необходимости, страницы памяти процессов откачиваются из физической памяти на диск, в *область подкачки*. При обращении к странице в виртуальной памяти, если она не находится в физической памяти, происходит ее подкачка с диска.
Виртуальная память реализуется и автоматически поддерживается ядром ОС UNIX.

Типы процессов

В ОС UNIX выделяется три типа процессов: *системные*, *процессы-демоны* и *прикладные 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.
К системным процессам можно отнести и процесс начальной инициализации, **init**, являющийся прародителем всех остальных процессов. Хотя **init** не является частью ядра, и его запуск происходит из выполняемого файла, его работа жизненно важна для функционирования всей системы в целом.
Демоны - это не интерактивные процессы, которые запускаются обычным образом - путем загрузки в память соответствующих им программ, 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, сетевых служб и т.д. Демоны не связаны ни с одним пользователем. Большую часть времени демоны ожидают, пока тот или иной процесс запросит определенную услугу.
К прикладным процессам относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. Важнейшим пользовательским процессом является *начальный командный интерпретатор*, который обеспечивает выполнение команд пользователя в системе UNIX.
Пользовательские процессы могут выполняться как в интерактивном (приоритетном), так и в фоновом режимах. Интерактивные процессы монопольно владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не имеет доступа к командной строке.
В следующем примере, показывающем часть списка процессов в ОС Solaris 8, **полужирным** выделены системные процессы. В этой ОС системными являются процесс- планировщик (**sched**), процесс откачки страниц виртуальной памяти (**pageout**) и процесс, синхронизирующий файловые системы (**fsflush**). Пользовательские процессы представлены на более темном фоне. Все остальные процессы - это демоны, реализующие те или иные службы. Имена команд, начинающиеся с дефиса, представляют начальные командные интерпретаторы пользователей.

```
[kravchuk@arturo 13:48:53 /]$ ps -ecf | more
    UID   PID  PPID  CLS PRI    STIME TTY      TIME CMD
   root     0     0  SYS  96   Фев 23 ?       0:19 sched
   root     1     0   TS  58   Фев 23 ?       0:04 /etc/init -
   root     2     0  SYS  98   Фев 23 ?       0:08 pageout
   root     3     0  SYS  60   Фев 23 ?      87:49 fsflush
   root   411     1   TS  58   Фев 23 ?       0:00 /usr/lib/saf/sac -t 300
   root   259     1   TS  50   Фев 23 ?       2:20 /usr/sbin/nscd
   root   184     1   TS  46   Фев 23 ?       0:00 /usr/lib/netsvc/yp/ypxfrd
   root    68     1   TS  58   Фев 23 ?       0:02
/usr/lib/sysevent/syseventd
   root   144     1   TS  59   Фев 23 ?       0:00 /usr/sbin/in.rdisc -s
   root   161     1   TS  58   Фев 23 ?       0:41 /usr/sbin/rpcbind
...
  markov  5724  5723   TS  48 12:07:16 pts/1    0:00 -bash
   root  3705   215   TS  54 09:46:57 ?        0:00 in.telnetd
   root  6804  6803   IA  48   Map 25 ??       0:00 /usr/dt/bin/dtterm
   root    87   310   TS  59   Map 19 ?        0:02 /usr/local/samba/bin/smbd
-D -s/usr/local/samba/lib/smb.conf
   root 27210   215   TS  54   Map 27 ?        0:00 in.telnetd
   root  3918   215   TS  54 10:11:00 ?        0:00 in.telnetd
kravchuk  3697  3679   TS  38 09:46:39 pts/14   0:00 -bash
...
```

Атрибуты процесса

Процесс в UNIX имеет ряд атрибутов, позволяющих операционной системе управлять его работой. Основные атрибуты представлены в следующих подразделах.

Идентификатор процесса (PID)

Каждый процесс имеет уникальный идентификатор PID, позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему следующий свободный (т.е. не ассоциированный ни с каким процессом) идентификатор. Присвоение идентификатора обычно происходит по возрастающий, т.е. идентификатор нового процесса больше, чем идентификатор процесса, созданного перед ним. Если идентификатор достигает максимального значения (обычно - 65737), следующий процесс получит минимальный свободный PID и цикл повторяется. Когда процесс завершает работу, ядро освобождает использовавшийся им идентификатор.

Идентификатор родительского процесса (PPID)

Идентификатор процесса, породившего данный процесс. Все процессы в системе, кроме системных процессов и процесса **init**, являющегося прародителем остальных процессов, порождены одним из существующих или существовавших ранее процессов.

Поправка приоритета (NI)

Относительный приоритет процесса, учитываемый планировщиком при определении очередности запуска. Фактическое же распределение процессорных ресурсов определяется приоритетом выполнения (атрибут **PRI**), 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 изменяемого планировщиком.

Терминальная линия (TTY)

Терминал или псевдотерминал, связанный с процессом. С этим терминалом по умолчанию связаны стандартные *потоки*: *входной*, *выходной* и *поток сообщений* об ошибках. Потоки (*программные каналы*) являются стандартным средством межпроцессного взаимодействия в ОС UNIX.
Процессы-демоны не связаны с терминалом.

Реальный (UID) и эффективный (EUID) идентификаторы пользователя

Реальным идентификатором пользователя данного процесса является идентификатор пользователя, запустившего процесс. Эффективный идентификатор служит для определения прав доступа процесса к

системным ресурсам (в первую очередь к ресурсам файловой системы). Обычно реальный и эффективный идентификаторы совпадают, т.е. процесс имеет в системе те же права, что и пользователь, запустивший его. Однако существует возможность задать процессу более широкие права, чем права пользователя, путем установки *бита SUID*, когда эффективному идентификатору присваивается значение идентификатора владельца выполняемого файла (например, пользователя **root**).

Реальный (GID) и эффективный (EGID) идентификаторы группы

Реальный идентификатор группы равен идентификатору основной или текущей группы пользователя, запустившего процесс. Эффективный идентификатор служит для определения прав доступа к системным ресурсам от имени группы. Обычно эффективный идентификатор группы совпадает с реальным. Но если для выполняемого файла установлен *бит SGID*, такой файл выполняется с эффективным идентификатором группы-владельца.

Жизненный цикл процесса в UNIX и основные системные вызовы

Жизненный цикл процесса в ОС UNIX может быть разбит на несколько состояний. Переход из одного состояния в другое происходит в зависимости от наступления определенных событий в системе. Возможны следующие состояния процесса:

1. Процесс выполняется в пользовательском режиме. При этом процессором выполняются прикладные инструкции данного процесса.

2. Процесс выполняется в режиме ядра. При этом процессом выполняются системные инструкции ядра от имени процесса.

3. Процесс не выполняется, но готов к запуску, как только планировщик выберет его (состояние **runnable**). Процесс находиться в очереди на выполнение и обладает всеми необходимыми ему ресурсами, кроме процессора.

4. Процесс находиться 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 приоритетного процесса.

6. Процесс только что создан системным вызовом **fork** и находится в переходном состоянии: он существует, но не готов к запуску и не находиться в состоянии сна.

7. Процесс выполнил системный вызов **exit** 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.

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 пользовательском режиме.
Выполнение в пользовательском режиме 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 пользовательский режим. Однако во время выполнения системного вызова процесса в режиме ядра процессу может понадобиться недоступный в данный момент ресурс. Для

ожидания доступа к такому ресурсу, процесс делает системный вызов **sleep** и переходит в состояние 4 - сна. При этом процесс добровольно освобождает вычислительные ресурсы, которые предоставляются следующему наиболее приоритетному процессу. Когда ресурс становиться доступным, ядро "пробуждает процесс", используя вызов **wakeup**, помещает его в очередь на выполнение, и процесс переходит в состояние 3 готовности к запуску.
При предоставлении процессу вычислительных ресурсов происходит переключение контекста, 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 переключение контекста при выполнении в режиме ядра может произвести к нарушению целостности самой системы. Поэтому переключение контекста откладывается до момента перехода процесса из режима ядра в пользовательский режим, 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. Далее состояние процесса зависит от предыстории.
Наконец, процесс выполняет системный вызов **exit** и заканчивает свое выполнение. Процесс может быть также завершен вследствие получения сигнала. В обоих случаях ядро освобождает ресурсы, принадлежащие 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 системный вызов **wait**, после чего вся информация о процессе будет уничтожена, а родитель получит код возврата завершившегося процесса.

Контекст процесса

Каждый процесс UNIX имеет *контекст*, под которым понимается вся информация, требуемая для описания процесса. Эта информация сохраняется, когда выполнение процесса приостанавливается, и восстанавливается, когда планировщик предоставляет процессу вычислительные ресурсы.
Контекст процесса в ОС UNIX состоит из нескольких частей:

- Адресное пространство процесса в пользовательском режиме
  Сюда входят код, данные и стек процесса, а также другие области, например, разделяемая память или код и данные динамических библиотек.

- Управляющая информация
  Ядро использует две основные структуры для управления процессом - **proc** и **user**. Сюда же входят данные, необходимые для отображения виртуального адресного пространства процесса в физическую память.

- Среда процесса
  Переменные среды процесса, значения которых задаются в командном интерпретаторе или в самом процессе с помощью системных вызовов, а также наследуются порожденным процессом от родительского и обычно хранятся в нижней части стека. Среду процесса можно получать или изменять с помощью функций.

- Аппаратный контекст
  Сюда входят значения общих и ряда системных регистров процессора, в частности, указатель текущей инструкции и указатель стека (см. в начале раздела).

Переключение между процессами, необходимое для распределения вычислительного ресурса, по существу, выражается в переключении контекста, когда контекст выполнявшегося процесса запоминается, а восстанавливается контекст процесса, выбранного планировщиком. Переключение процесса является достаточно ресурсоемкой операцией. Помимо сохранения состояния регистров процесса, ядро вынуждено выполнить множество других действий.
Контекст переключается в четырех случаях:

1. Текущий процесс переходит в состояние сна, ожидая недоступного ресурса.

2. Текущий процесс завершает свое выполнение.

3. Если после пересчета приоритетов в очереди на выполнение есть более высокоприоритетный процесс.

4. Происходит пробуждение более высокоприоритетного процесса.

Первые два случая соответствуют добровольному переключению контекста и действия ядра при этом достаточно просты. Ядро вызывает процедуру переключения контекста из функций **sleep** или **exit**.
Третий и четвертый случаи переключения контекста происходят не по воле процесса, который в это время выполняется в режиме ядра и поэтому не может быть немедленно приостановлен. В этой ситуации ядро устанавливает специальный флаг **runrun**, который указывает, что в очереди находится более высокоприоритетный процесс, требующий предоставления вычислительных ресурсов. Перед переходом процесса из режима ядра в режим задачи ядро проверяет этот флаг и, если он установлен, вызывает функцию переключения контекста.

Приоритеты процессов

Планирование процессов и UNIX основано на *приоритете* процесса. 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. Если процесс готов к запуску и имеет наивысший приоритет, планировщик приостановит выполнение текущего процесса (с более низким приоритетом), даже если последний не "выработал" свой временной квант.
Ядро UNIX является *непрерываемым* (nonpreemptive). Это означает, что процесс, находящийся в режиме ядра (в результате системного вызова или прерывания) и выполняющий системные инструкции, не может быть прерван системой, а вычислительные ресурсы переданы другому высокоприоритетному процессу. В этом состоянии выполняющийся процесс не может освободить процессор "по собственному желанию", в результате недоступности какого-либо ресурса перейдя в состояние сна. В противном случае система может прервать выполнение процесса только при переходе из режима ядра в пользовательский режим. Такой подход значительно упрощает решение задач синхронизации и поддержки целостности структур данных ядра.
Каждый процесс имеет два атрибута приоритета: текущий приоритет, на основании которого происходит планирование, и относительный приоритет, называемый также поправкой приоритета - **nice number**, который задается при порождении процесса и влияет на текущий приоритет.
Диапазон значений текущего приоритета различен, в зависимости от версии ОС UNIX и используемого планировщика. В любом случае, процессы, выполняющиеся в пользовательском режиме, имеют более низкий приоритет, чем работающие в режиме ядра.

Создание процесса

Новый процесс создается в UNIX только путем системного вызова **fork**. Процесс, сделавший вызов **fork**, называется *родительским*, а вновь созданный процесс - *порожденным*. Новый процесс является точной копией родительского. При порождении (разветвлении) процесса проверяется, достаточно ли памяти и места в таблице процессов для данного процесса. Если да, то образ текущего процесса копируется в новый образ процесса, и в таблице процессов возникает новый элемент. Новому процессу присваивается новый уникальный идентификатор (**PID**). Когда изменение таблицы процессов ядра завершается, процесс добавляется к списку процессов, доступных для выполнения и ожидающих в очереди планировщика подобно другим процессам.
Порожденный процесс **наследует** от родительского процесса следующие основные характеристики:

1. Способы обработки сигналов (адреса функций обработки сигналов).

2. Реальные и эффективные идентификаторы пользователя и группы.

3. Значение поправки приоритета.

4. Все присоединенные разделяемые сегменты памяти.

5. Идентификатор группы процессов.

6. Терминальную линию.

7. Текущий каталог.

8. Корневой каталог.

9. Маска создания файлов (**umask**).

10. Ограничения ресурсов (**ulimit**).

Порожденный процесс **отличается** от родительского процесса следующими основными характеристиками:

1. Порожденный процесс имеет свой уникальный идентификатор.

2. Порожденный процесс имеет другой идентификатор родительского процесса, равный идентификатору породившего процесса.

3. Порожденный процесс имеет свои собственные копии дескрипторов файлов (в частности, стандартных потоков), открытых родительским процессом. Каждый дескриптор файла порожденного процесса имеет первоначально такое же значение текущей позиции в файле, что и соответствующий родительский.

4. У порожденного процесса обнуляются счетчики времени, потраченного системой для его обслуживания.

Системный вызов **fork** завершается неудачей и новый процесс не порождается, если:

- Создать процесс запрещает системное ограничение на общее количество процессов.

- Создать процесс запрещает системное ограничение на количество процессов у одного пользователя.

- Общее количество системной памяти, предоставленной для физического ввода-вывода, временно оказалось недостаточным.

При успешном завершении порожденному процессу возвращается значение **0**, а родительскому процессу возвращается идентификатор порожденного процесса. В случае ошибки родительскому процессу возвращается **-1**, новый процесс не создается и переменной **errno** присваивается код ошибки.
Обычно после порождения порожденный процесс выполняет системный вызов **exec**, перекрывающий сегменты текста и данных процесса новыми сегментами текста и данных, взятыми из указанного выполняемого файла. При этом аппаратный контекст процесса инициализируется заново.
Выполняемый файл состоит из заголовка, сегмента команд и сегмента данных. Данные (глобальные переменные) состоят из инициализированной и неинициализированной частей.

Если системный вызов **exec** закончился успешно, то он не может вернуть управление, так как вызвавший процесс уже заменен новым процессом. Возврат из системного вызова **exec** свидетельствует об ошибке. В таком случае результат равен **-1**, а переменной **errno** присваивается код ошибки.
Новый процесс наследует у процесса, вызвавшего **exec**, следующие основные характеристики:

1. Значение поправки приоритета.
2. Идентификатор процесса.
3. Идентификатор родительского процесса.
4. Идентификатор группы процессов.
5. Терминальную линию.
6. Текущий каталог.
7. Корневой каталог.
8. Маску создания файлов.
9. Ограничения ресурсов.
10. Счетчики времени, потраченного системой на обслуживание этого процесса.
11. Блокировки доступа к сегментам файлов.

Сон и пробуждение

Процесс обычно переводится в состояние сна при обработке системной функции. Если для завершения обработки запроса требуется недоступный ресурс, процесс снимается с процессора и переводится в состояние сна.
Состояние сна - это логическое состояние процесса, при этом он не перемещается физически в памяти. Переход в состояние сна, в первую очередь, определяется занесением в системную таблицу процессов соответствующего флага состояния и события, пробуждающего процесс.
События информируют о доступности того или иного ресурса. Как правило, события связаны с работой периферийных устройств. Наступление одного и того же события может ожидать несколько процессов. Поскольку переход из состояния в состояние акт скорее логический, то и пробуждаются все процессы ожидающие данное событие. Это приводит к смене состояния со "сна" на "готов к выполнению", и соответствующие процессы помещаются в очередь на запуск. Задачу выбора процесса для запуска решает планировщик.
Поскольку планировщик принимает решение о запуске процесса, основываясь на приоритетах, единственным способом установить "справедливый" порядок запуска процессов является присвоение определенного приоритета каждому событию.

Завершение выполнения процесса

Процесс завершает работу при выполнении системного вызова **exit**. Процесс может сам завершить свою работу, в соответствии с алгоритмом, либо может быть прекращен ядром.
При завершении процесса последовательно выполняются следующие действия:

1. Отключаются все сигналы.

2. В вызвавшем процессе закрываются все дескрипторы открытых файлов.

3. Если родительский процесс находится в состоянии вызова **wait**, то системный вызов **wait** завершается, выдавая родительскому процессу в качестве результата идентификатор завершившегося процесса, и младшие 8 бит его кода завершения.

4. Если родительский процесс не находится в состоянии вызова **wait**, то завершающийся процесс переходит в состояние зомби.

У всех существующих потомков завершенных процессов, а также у зомби-процессов идентификатор родительского процесса устанавливается равным 1. Таким образом, они становятся потомками процесса инициализации (**init**).
Если идентификатор процесса, терминальная линия и идентификатор группы процессов у завершающегося процесса совпадают, то всем процессам с тем же идентификатором группы процессов посылается сигнал **SIGHUP**. Тем самым, завершаются и все порожденные в приоритетном режиме процессы.
Родительскому процессу посылается сигнал **SIGCHLD** (завершение порожденного процесса). Этот сигнал пробуждает родительский процесс, если тот ожидает завершения порожденных процессов.

Получение информации о процессах

Для получения информации о состоянии процессов используется команда **ps**. Она имеет следующий синтаксис:

ps [-acdelfjLP]
[-t **список_терминалов**]
[-p **список_идентификаторов_процессов**]
[-u|U **список_идентификаторов_пользователей**]
[-g **список_идентификаторов_лидеров_групп**]
[-G **список_числовых_идентификаторов_групп**]

Основные опции команды ps в системах SVR4 и BSD описаны в табл. 16.

Таблица 16. Опции команды ps

Опция	**Назначение**
-a	Предоставляет информацию обо всех процессах, кроме групповых, и не связанных с терминалом.
-d	Предоставляет информацию обо всех процессах, исключая лидеров сеанса.
-e	Предоставляет информацию обо всех процессах.
-l	Генерирует длинный листинг.
-f	Генерирует полный листинг.
-g список	Выводит информацию только о процессах, для которых указаны идентификаторы лидеров групп. Лидер группы - это процесс, номер которого идентичен его идентификатору группы. Командный интерпретатор, запускаемый при входе в систему, является стандартным примером лидера группы.
-G список	Предоставляет информацию обо всех процессах, имеющих отношение к указанным номерам групп.
-p список	Предоставляет информацию по процессам с указанными идентификаторами.
-t список	Предоставляет информацию по процессам, имеющим отношение к указанным терминалам.
-U список	Предоставляет информацию обо всех процессах, имеющих отношение к указанным идентификаторам пользователей.
-u список	Предоставляет информацию обо всех процессах, имеющих отношение к указанным именам пользователей.

Основные поля в результатах выполнения команды **ps** представлены в табл. 17.

Таблица 17. Основные характеристики процессов, предоставляемые командой ps

Заголовок	**Значение**
ADDR	Адрес процесса в памяти.
C	Доля выделенного планировщиком времени ЦП.
COMD	Имя команды и аргументы (для опции **-f**).
F	Флаги (шестнадцатеричные), логическая сумма которых дает следующие сведения о процессе: **00** - процесс терминирован; элемент таблицы процессов свободен; **01** - системный процесс: всегда в основной памяти; **02** - процесс трассируется родительским процессом; **04** - родительский трассировочный сигнал остановил процесс; родительский процесс ждет [см. **ptrace(2)**]; **08** - процесс не может быть разбужен сигналом; **10** - процесс в основной памяти; **20** - процесс в основной памяти; блокирован до завершения события; **40** - идет сигнал к удаленной системе; **80** - процесс в очереди на ввод-вывод.
NI	Поправка приоритета.
PID	Идентификатор процесса.
PPID	Идентификатор родительского процесса.
PRI	Текущий приоритет процесса.
S	Состояние процесса: **B**,**W** - процесс находиться в состоянии ожидания; **I** - создание процесса; **O** - процесс выполняется; **R** - находиться в очереди готовых к выполнению процессов; **S** - процесс не активен; **T** - процесс трассируется; **X** - ожидает дополнительной оперативной памяти; **Z** - процесс "зомби".
STIME	Время запуска процесса.
SZ	Размер (в блоках по 512 байт) образа процесса в памяти.
TIME	Общее время выполнения для процесса
TTY	Терминальная линия процесса
UID	Идентификатор пользователя владельца процесса
WCHAN	Адрес события, которого ожидает процесс. У активного процесса этот столбец - пустой.

В зависимости от переданных опций и реализации, команда **ps** может выдавать и другие атрибуты. Команду **ps** может выполнять любой пользователь. Рассмотрим простой пример:

```
[kravchuk@arturo 15:59:30 /]$ ps
   PID TTY      TIME CMD
  3697 pts/14   0:00 bash
[kravchuk@arturo 15:59:33 /]$ ps -l
 F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME CMD
 8 S 31061  3697  3679  0  51 20 e3110048    499 e31100b4 pts/14   0:00 bash
[kravchuk@arturo 15:59:38 /]$ ps -p 5726
   PID TTY      TIME CMD
  5726 pts/1    0:00 mc
```

Управление приоритетом процессов

Во всех UNIX-системах пользователи могут при запуске процесса задавать значение поправки приоритета с помощью команды **nice**. (Говорят, что команда запускается "из- под" **nice**.) Эта команда имеет следующий синтаксис:

nice [-**инкремент** | -n [-|+]**инкремент**] **команда** [**аргумент**...]

Диапазон значений **инкремента** в большинстве систем - от -20 до 20. Если **инкремент** не задан, используется стандартное значение 10. Положительный инкремент означает снижение текущего приоритета. Обычные пользователи могут задавать только положительный инкремент и, тем самым, только снижать приоритет.

Пользователь **root** может задать отрицательный инкремент, который повышает приоритета процесса и, тем самым, способствует его более быстрой работе:
```
# nice -n -10 make
```

Сигналы: посылка и обработка

Сигналы обеспечивают механизм вызова определенной процедуры при наступлении некоторого события (аналогично прерываниям). Каждое событие имеет свой числовой идентификатор (обычно в диапазоне от 1 до 36) и соответствующую символьную константу - имя. При работе с сигналами необходимо различать две фазы:

1. Генерация или посылка сигнала.

2. 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 событиями сигнал ожидает доставки.
Сигнал может посылаться одним процессом другому (с помощью соответствующего системного вызова) и будет доставлен, если оба процесса - одного пользователя или сигнал послан от имени пользователя **root**.
Сигналы посылаются также ядром.
Ядро генерирует и посылает процессу сигнал в ответ на ряд событий, которые могут быть вызваны самим процессом, другим процессом, прерыванием или каким либо внешним событием. Основные причины отправки сигнала:

Исключительные ситуации
: Выполнение процесса вызывает исключительну ситуацию, например, деление на 0.

Терминальные прерывания
: Нажатие клавиш терминала, например, <Del>, <Ctrl+C>, <Ctrl+\>, вызывает посылку сигнала текущему процессу, связанному с терминалом.

Другие процессы
: Процесс может посылать 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 средства управления заданиями, используют сигналы для манипулирования фоновыми и текущими процессами. Когда процесс, выполняющийся в фоновом режиме, делает попытку чтения или записи на терминал, ему посылается сигнал останова. Когда порожденный процесс завершает свою работу, родительский процесс уведомляется об этом также с помощью сигнала.

Квоты
: Когда процесс превышает выделенную ему квоту вычислительных ресурсов или ресурсов файловой системы, ему посылается соответствующий сигнал.

Уведомления
: Процесс может запросить уведомление о наступлении тех или иных событий, например, готовности устройства и т.д. Такое уведомление посылается процессу в виде сигнала.

Будильники
: Если процесс установил таймер, ему будет послан сигнал, когда значение таймера станет равным 0.

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 возможны действия: завершить выполнение процесса (с созданием образа памяти **core** и без), игнорировать сигнал, остановить процесс и продолжить процесс.

Следует заметить, что любая обработка сигнала, в том числе и обработка по умолчанию, подразумевает, что процесс выполняется. На системах с высокой загрузкой это может привести к задержкам между отправлением и доставкой сигнала, т.к. процесс не может получить сигнал, пока не будет выбран планировщиком, и ему не будут предоставлены вычислительные ресурсы.
Доставка сигнала происходит после того, как ядро от имени процесса вызывает системную процедуру **issig()**, которая проверяет, существуют ли ожидающие доставки сигналы, адресованные данному процессу.
Процедура **issig()** вызывается ядром в трех случаях:

1. Непосредственно перед возвращением из режима ядра в пользовательский режим 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 ожидание доставки сигнала, ядро вызывает функцию доставки сигнала, которое выполняет действие по умолчанию или вызывает специальную функцию **sendsig()**, запускающую обработчик сигнала, зарегистрированный процессом. Функция **sendsig()** возвращает процесс в пользовательский режим, передает управление обработчику сигнала, а затем восстанавливает контекст процесса для продолжения прерванного сигналом выполнения.
Работа с сигналами, связанными с исключительными ситуациями, незначительно отличается от описанной выше. Исключительная ситуация возникает при выполнении процессом определенной инструкции, вызывающей в системе ошибку. Если такое происходит, вызывается системный обработчик исключительной ситуации, и процесс переходит в режим ядра, почти так же, как и при обработке любого другого прерывания. Обработчик отправляет процессу соответствующий сигнал, который доставляется, когда процесс возвращается в пользовательский режим.
В состоянии сна существуют две категории событий, вызвавших состояние сна процесса: допускающие прерывание сигналом и не допускающие такого прерывания. В последнем случае сигнал будет терпеливо ожидать нормального пробуждения процесса.

Основные сигналы

Информация об основных сигналах представлена в табл. 18.
Таблица 18. Основные сигналы

Сигнал	Стандартная обработка	Значение
SIGTERM 15	Завершение процесса	Стандартный сигнал, посылаемый для остановки процесса.
SIGHUP 1	Завершение процесса	Отключился терминал (или закрыто терминальное окно). Сигнал посылается всем не фоновым процессам, связанным с соответствующей терминальной линией.
SIGKILL 9	Завершение процесса	Не перехватываемый сигнал, позволяющий завершить любой процесс.
SIGILL 4	Завершение процесса и сброс образа памяти	На центральный процессор была послана запрещенная инструкция. Это могло быть следствием недопустимого перехода в машинном коде программы, например, попытки выполнить строку данных.
SIGTRAP 5	Завершение процесса и сброс образа памяти	Была установлена ловушка точки прерывания процесса. Этим управляет системный вызов **ptrace**, который полезен для отладки.
SIGFPE 8	Завершение процесса и сброс образа памяти	Была попытка выполнить запрещенную арифметическую операцию, например, взятие логарифма отрицательного числа или деление на 0.
SIGBUS 10	Завершение процесса и сброс образа памяти	Ошибка на шине ввода-вывода. Обычно это является результатом попытки выполнить чтение или запись вне границ памяти программы.

SIGSEGV 11	Завершение процесса и сброс образа памяти	Это нарушение сегментации - проклятие разработчиков программ! Оно означает, что вы попытались получить доступ к сегменту памяти запрещенным образом. Может быть, это было присваивание значения части сегмента кода или чтение из нулевого адреса.
SIGPIPE 13	Завершение процесса	Программа попыталась выполнить чтение или запись в программный канал, другой конец которого уже завершил работу. Этот сигнал помогает завершить работу конвейера, когда одна из его команд дала сбой.
SIGALRM 14	Завершение процесса	Программист может установить будильник, чтобы позволить вам в определенный момент времени выполнить какое-нибудь действие.
SIGCHLD 18	Игнорируется	Сначала это был сигнал завершения работы дочернего процесса, но сейчас он означает изменение состояния дочернего процесса.
SIGTSTP 24	Остановка процесса	Это запрос от терминала на остановку процесса. Посылка этого сигнала процессу происходит при нажатии комбинации клавиш **Ctrl-Z**.
SIGCONT 25	Игнорируется	Этот сигнал указывает процессу на возобновление его работы. Процессу посылается либо команда **fg**, либо **bg**, а командный интерпретатор выполняет внутренний системный вызов **wait** для привилегированного процесса, либо не выполняет его для фонового процесса.

Детальная информация о сигналах представлена на страницах справочного руководства **signal**.
Процесс с помощью системного вызова **signal()** может задать нестандартный обработчик любого сигнала, кроме **SIGKILL** (**9**).

Посылка сигналов

Для посылки сигналов из командного интерпретатора используется команда **kill**. Она имеет следующий синтаксис:

kill [**-сигнал**] **pid** ...

Эта команда посылает указанный сигнал (по умолчанию - **SIGTERM**) всем процессам с указанными идентификаторами. Посылать сигнал можно и не существующему процессу - выдается предупреждение, но другим процессам сигнал посылается. Посылаемый сигнал задается по имени без префикса **SIG** или по номеру, например:

```
[kravchuk@arturo 16:56:55 /]$ echo $$
3697
[kravchuk@arturo 16:56:58 /]$ kill -STOP 3697
```

В результате текущий сеанс зависает.

Средства обработки текста

Редактирование и вообще обработка текстов - одна из основных операций, выполняемых на компьютерах. Стандартные средства редактирования в ОС UNIX появились давно и ориентированы на простейшие текстовые терминалы. К таким средствам можно отнести строковый редактор **ed** и экранный редактор **vi**. При первом знакомстве они могут показаться сложными и явно устаревшими с точки зрения "дружественности", однако в мире UNIX хорошо не то, что является самым новым и "красивым", а, скорее, то, что используется давно, многими людьми и есть в любой версии. Это в полной мере относится к стандартным средствам обработки текстов.

Регулярные выражения и сопоставление с образцом

Эффективность обработки текста определяется эффективностью поиска необходимых фрагментов. Для задания образцов поиска в ОС UNIX используется ряд *метасимволов регулярных выражений*, впервые появившихся в редакторе **ed** и представленных в табл. 19.

Таблица 19. Метасимволы регулярных выражений

Метасимвол	Описание
c	Любой конкретный символ задает совпадение с таким же символом
\c	Отменяет специальный смысл символа **c**

^	Соответствует началу строки, когда ^ начинает образец
$	Соответствует концу строки, когда **$** заканчивает образец
.	Совпадает с любым одиночным символом
[...]	Соответствует одному любому символу в **...**; допустимы диапазоны типа **a-z**
[^...]	Соответствует любому одиночному символу, не входящему в **...**; допустимы диапазоны
r*	Соответствует нулевому или более числу вхождений **r**, где **r** - символ или **[...]**
&	Используется только в правой части команд замены (**s**); вставляет фрагмент, совпавший с образцом
\(...\)	Помечает регулярное выражение; найденные строки доступны как **\1**, **\2** и т.д. до **\9** в левой и правой частях соответствующей команды замены **s**, а также в шаблонах поиска сразу после закрытия соответствующей круглой скобки.

Примеры регулярных выражений

Простые примеры регулярных выражений и задаваемых ими шаблонов поиска представлены в табл. 20.

Таблица 20. Примеры использования регулярных выражений

Образец	**Соответствие**
/^$/	пустая строка, т.е. только конец строки
/./	непустая строка, по крайней мере один символ
/^/	все строки
/thing/	**thing** где-либо в строке
/^thing/	**thing** в начале строки
/thing$/	**thing** в конце строки
/^thing$/	строка, состоящая лишь из **thing**
/thing.$/	**thing** плюс любой символ в конце строки
/\/thing\//	**/thing/** где-либо в строке
/[tT]hing/	**thing** или **Thing** где-либо в строке
/thing[0-9]/	**thing**, за которой идет одна цифра
/thing[^0-9]/	**thing**, за которой идет не цифра
/thing1.*thing2/	**thing1**, затем любая строка, затем **thing2**
/^thing1.*thing2$/	**thing1** в начале и **thing2** в конце

Помеченные регулярные выражения

Чтобы манипулировать не только целыми фрагментами, выбираемыми регулярными выражениями, но и их частями, используются *помеченные регулярные выражения*: если конструкция **\(...\)** появляется в регулярном выражении, то часть соответствующего ей фрагмента доступна как **\1**. Допускается использование до девяти помеченных выражений, на которые ссылаются **\1**, **\2** и т.д.

Вот ряд примеров использования помеченных регулярных выражений:

s/\(...\)\(.*\)/\2\1/	Поместить 3 первых символа в конец строки
/\(..*\)\1/	Найти строки, содержащие повторяющиеся смежные цепочки символов
s/^\(..*\)\.\(..*\)/\1.\ \2/	Перенести остаток строки после первой точки на следующую строку

Поиск в тексте по образцу - утилита grep

Программы семейства **grep** осуществляют поиск шаблона (задаваемого на языке регулярных выражений) в указанных файлах или во входном потоке и (обычно) выдают соответствующие шаблону строки в выходной поток. В это семейство входят три программы, **grep**, **egrep** и **fgrep**, отличающиеся алгоритмами (а, значит, скоростью работы и используемыми ресурсами системы) и, частично, возможностями при задании шаблонов.

Вызов программы grep

Вызов команды **grep** имеет следующий синтаксис:

grep [**опции**] [**регулярное_выражение**] [**файл** ...]

Команда ищет строки, задаваемые шаблоном в виде *ограниченного регулярного выражения* (используют подмножество допустимых алфавитно-цифровых и специальных символов), аналогичного используемым в **ed**, в указанных файлах или во входном потоке. Возможные опции приведены в табл. 21.

Таблица 21. Опции командной строки grep

Опция	Назначение
-b	Перед каждой строкой выдавать номер блока, в котором она найдена. Это может пригодиться при определении номера блока по контексту
-c	Выдавать только количество строк, соответствующих шаблону
-i	Игнорировать разницу между прописными и строчными буквами
-h	Предотвращает выдачу имени файла перед совпавшей строкой. Используется при многофайловом поиске.
-l	Выдавать имена файлов, содержащих совпавшие строки, один раз, разделяя их переводом строки. Не повторяет имена файлов, если шаблон найден более одного раза.
-n	Предваряет каждую строку ее порядковым номером (первая строка имеет номер 1)
-s	Подавляет выдачу сообщений об ошибках, связанных с не существованием файлов или недоступностью для чтения
-v	Выдает все строки, кроме тех, что содержат шаблон
-e se	Ищет специальное выражение **se** (полное регулярное выражение, начинающееся с **-**)
-f файл	Берет список полных регулярных выражений из файла

Будьте внимательны при использовании символов **$**, *****, **[**, ^, |, **(**, **)** и \ в шаблоне, так как они также имеют значение для командного интерпретатора. Лучше заключать искомый шаблон в апострофы: **'...'**.

Статус выхода равен 0, если найдены совпадающие строки, 1 - если строки не найдены и 2 если имеется синтаксическая ошибка или недоступные файлы (даже если совпадения найдены).

Рассмотрим простые примеры:

```
[kravchuk@arturo 17:30:29 /]$ echo abc abc | grep '\([abc][abc]*\) \1'
abc abc
[kravchuk@arturo 17:31:13 /]$ echo abc abc | grep 'c a'
abc abc
[kravchuk@arturo 17:31:22 /]$ echo abc abc | grep '^c a'
[kravchuk@arturo 17:31:26 /]$ cd $INFORMIXDIR/etc
[kravchuk@arturo 17:31:45 /usr/inf.731/etc]$ grep -n $INFORMIXDIR
^C
[kravchuk@arturo 17:32:03 /usr/inf.731/etc]$ grep -n tmp *.sh
beta_evidence.sh:306:    DUMPDIR=/tmp
bldutil.sh:40:# remove tmp salvage_file
bldutil.sh:55:    RESFILE=/tmp/bldutil.$$
evidence.sh:302:    DUMPDIR=/tmp
logevent.sh:46:TMPFILE=${TMPDIR:-/tmp}/$PROG.`date +%y-%m-%d-%H%M-%S`
```

Редактор vi

vi (vedit) - экранно-ориентированный текстовый редактор. Он позволяет видеть одновременно целую страницу текста, перемещаться по нему курсором и непосредственно видеть вносимые изменения.

Редактор **vi** является наследником строкового редактора **ex**, который, в свою очередь, является расширением базового текстового редактора **ed**. Тем самым, обеспечивается преемственность средств редактирования и использование эффективного механизма поиска и замены на базе регулярных выражений.

Вызов

Для вызова редактора **vi** в простейшем случае используется следующий синтаксис:

vi [+**строка**] [**файл**]

В результате, указанный файл открывается в окне редактора. Если файл не указан, редактируется первоначально пустой буфер (т.е. новый файл, имя которого первоначально не задано).

Строку, на которой открывается файл, можно задавать следующим образом:

+номер	Строка с указанным **номером**
+	Последняя строка файла
+/re	Строка, соответствующая указанному регулярному выражению **re**

Указанная строка будет находиться в центре экрана (если только файл не меньше, чем размер экрана) и в ее начале будет установлен курсор.

Режимы работы

Редактор **vi** поддерживает несколько режимов работы:

Командный режим	Нормальный и начальный режим. По завершении других режимов происходит возврат в командный режим. Для форсированного перехода в этот режим используется клавиша **Esc**
Режим ввода	В режим ввода входят при задании одной из следующих команд: **a A i I o O c C s S R**. При этом может набираться произвольный текст. Из этого режима выходят либо по **Esc**, либо он автоматически прерывается редактором. При этом обычно подается звуковой сигнал.
Режим последней строки	Чтение ввода для команды **: / ?** или **!**; прекращается нажатием клавиши **Enter**.

Основные команды

Основные команды редактора **vi** представлены в табл. 22.

Таблица 22. Сводка основных команд редактора vi

Перемещение курсора	
H (Ctrl-h)	курсор влево
J (Ctrl-n)	курсор вниз
K (Ctrl-p)	курсор вверх
L (Space)	курсор право
Ctrl-u	Переход вверх на половину экрана
Ctrl-d	Переход вниз на половину экрана
Ctrl-f	На страницу вперед (**PageDn**)
Ctrl-b	На страницу назад (**PageUp**)
0	Переход в начало текущей строки
$	Переход в конец текущей строки
nG	Переход на строку с номером n
Добавление текста	
a	Добавить текст после курсора
A	Добавить текст в конце текущей строки
i	Вставить текст перед курсором
I	Вставить текст в начале текущей строки
o	Образовать новую строку под текущей
O	Образовать новую строку над текущей
Изменение текста	
~	Изменить регистр символа над курсором
r	Замена одного символа
R	Замена символов
Удаление текста	
x	Удаление символа
dd	Удаление строки
Ndd	Удаление N строк
Поиск и замена	
/str	Поиск строки str вперед. str может быть регулярным выражением
?/str	Поиск строки str назад
n	Повторить поиск в том же направлении
N	Повторить поиск в обратном направлении

:[range]s/old/new/[g]	Заменить **old** на **new** в указанном диапазоне строк **range**. **new** и **old** могут быть регулярными выражениями, а **range** задается аналогично диапазону строк в редакторе **ed**. Суффикс **g** означает заменить во всем файле.
Копирование текста	
yy	Копирование строки в целом
Nyy	Копирование N строк
p	Вставить из буфера после (курсора, текущей строки)
P	Вставить из буфера перед (курсором, текущей строкой)
Выход из редактора	
:wq ENTER	Запись и выход. Записать текст из буфера в файл и выйти из редактора.
:x ENTER	Условная запись и выход. Записать текст из буфера только при наличии изменений и выйти из редактора.
:q! ENTER	Закончить редактирование без записи изменений.
Другие команды	
!	Выполнить одну команду интерпретатора
.	Повторить последнюю команду
u	Отменить действие последней команды
J	Соединить строки
Ctrl-G	Показать номер текущей строки

Курсор можно перемещать и клавишами перемещения курсора или клавишами **PageUp**, **PageDn**, но эти возможности, в отличие от описанных в таблице, поддерживаются не на всех терминалах.

Командный интерпретатор

Пользователь ОС UNIX общается с системой через *командный интерпретатор* (shell). Через него происходит доступ к командам, файловой системе и другим функциям ядра UNIX. Это обычная программа (т.е. не входит в ядро операционной системы UNIX). Ее можно заменить другой или использовать несколько разных версий. Наиболее известны следующие версии:

sh
: Классический интерпретатор версии UNIX V7, иначе называемый по фамилии автора ***Bourne shell***.

ksh
: Интерпретатор ***Korn shell***, дополняющий классический shell возможностями работы с заданиями пользователя, историей работы и позволяющий редактировать командную строку при помощи команд, аналогичных **vi**. Является фактически стандартом для POSIX-совместимых систем, в частности, UNIX System V.

csh
: Стандартный интерпретатор BSD UNIX и производных от него систем. Отличается улучшенными диалоговыми возможностями, способом присваивания и экспортирования переменных в среду, управляющими конструкциями и рядом других моментов; тоже поддерживает историю и редактирование командной строки. Главное и, по моему мнению, отрицательное его отличие от других интерпретаторов, - это свои управляющие конструкции, не совпадающие с **sh**.

bash
: Свободно распространяемый в виде исходных текстов интерпретатор, называемый "***Bourne another shell***", объединяющий все лучшее из остальных интерпретаторов с удобными возможностями редактирования командной строки и работы с историей команд. В настоящее время - фактический стандарт.

В рамках этого курса мы будем рассматривать, в основном, средства, не выходящие за пределы возможностей командных интерпретаторов **sh** и **ksh** (в классической версии 1988 года).

Структура командной строки

Командные строки рассматриваются по одной и имеют определенную структуру. Чтобы понять ее, рассмотрим ряд синтаксических определений:

<пробел> ::=
<символ пробела> | <символ табуляции>
<имя> ::=
<буква или подчеркивание> {<допустимый символ имени>}
<буква или подчеркивание> ::=
<буква> | _
<допустимый символ имени> ::=
<буква> | <цифра> | _
<параметр> ::=
<имя> | <цифра> | * | @ | # | ? | - | $ | !
<слово> ::=
<не пробел> {<не пробел>}
<простая команда> ::=
<слово> {<пробел> <слово>}

Итак, *простая команда* - это последовательность слов через пробел. Нажатие клавиши **Enter** при вводе команды или перевод строки при обработке сценария являются для командного интерпретатора признаком завершения команды. Она обрабатывается и выполняется.

Значением простой команды является ее *статус выхода* (см. далее) в случае нормального завершения или (восьмеричное) 200+статус при ненормальном завершении.

Пример простой команды:

```
$ who
oracle    pts000      Aug 20 10:08
root      console     Aug 20 09:03
intdbi    pts004      Aug 20 12:45
$
```

Из простых команд строятся более сложные конструкции: *конвейеры* и *списки*.

<конвейер> ::=
<команда> {| <команда>}
<список> ::=
<конвейер> {<разделитель> <конвейер>} [<терминатор команды>]
<разделитель> ::=
&& | || | <терминатор команды>
<терминатор команды> ::=
; | **&**

Конвейер - это последовательность одной или более команд, разделенных |. Стандартный выходной поток каждой команды, кроме последней, соединяется при помощи *программного канала* со стандартным входным потоком следующей команды. Каждая команда выполняется как отдельный процесс; интерпретатор ожидает окончания последней команды. Статусом выхода конвейера является статус выхода его последней команды.

Вот пример простого конвейера:

```
$ ls | tee save | wc
    15      15     100
$
```

Список - это последовательность одного или более конвейеров, разделенных **;**, **&**, **&&** или || и, возможно, заканчивающаяся **;** или **&**. Из этих четырех символов, **;** и **&** имеют равный приоритет, который ниже, чем у **&&** и || (эти символы тоже имеют равный приоритет). Точка с запятой (**;**) вызывает последовательное выполнение предшествующего конвейера (т.е. командный интерпретатор ожидает окончания конвейера перед выполнением любых команд, следующих за точкой с запятой). Амперсанд (**&**) вызывает асинхронное выполнение предшествующего конвейера (т.е. командный интерпретатор не ожидает окончания работы конвейера). Символ **&&** (||) ведет к тому, что следующий за ним список выполняется только в том случае, когда предыдущий конвейер вернул нулевой (ненулевой) статус выхода. В список может входить произвольное количество переводов строк и точек с запятой, разделяющих команды.

Теперь можно дать общее определение команды:

<команда> ::=
<простая команда> |
<оператор управления> |
<определение функции> |
<список> | **(**<список>**)** | **{** <список>**; }**

Список в круглых скобках выполняется в порожденном командном интерпретаторе. Круглые скобки обычно используют для группировки команд.

Список в фигурных скобках выполняется в текущем командном интерпретаторе, без порождения дополнительного процесса, и замещает образ командного интерпретатора (это аналог системного вызова **exec**).

Операторы управления и синтаксис определения функций рассматривается далее.

Рассмотрим пример сложной команды:

```
bash$ (sleep 5; date) & date
[1] 1148
```

```
Wed Aug 20 15:00:11 ???? 1997
bash$ Wed Aug 20 15:00:16 ???? 1997
```
Фоновый процесс начинается, но сразу "засыпает"; тем временем вторая команда **date** выдает текущее время, а интерпретатор - приглашение для ввода новой команды. Через пять (примерно, зависит от загрузки системы и т.п.) секунд прекращается выполнение команды **sleep** и первая команда **date** выдает новое время.

Метасимволы командного интерпретатора

Ряд символов, как было описано выше, имеют для командного интерпретатора специальное значение - это *метасимволы*. Они описаны в табл. 23.
Метасимволы не входят в команды и обрабатываются в несколько проходов **до** начала выполнения реальных программ.

Таблица 23. Метасимволы командного интерпретатора

Метасимвол	Интерпретация
>	**prog>file** - переключить стандартный выходной поток в файл
>>	**prog>>file** - добавить стандартный выходной поток к файлу
<	**prog<file** - извлечь стандартный входной поток из файла
|	**p1 | p2** - передать стандартный выходной поток **p1** как стандартный входной поток **p2**
<<str	"*Документ здесь*": стандартный входной поток задается в последующих строках до строки, состоящей только из символов **str**. По умолчанию в данных интерпретируются метасимволы \, **$** и ``. Если необходимо предотвратить в данных интерпретацию всех метасимволов, необходимо экранировать строку **str**, предварив обратной косой или взяв в двойные или одиночные кавычки.
*****	Задает в имени файла любую строку из нуля или более символов
?	Задает любой символ в имени файла
[abc]	Задает любой символ из **[abc]** в имени файла, при этом допускаются диапазоны, задаваемые при помощи дефиса **-**. Если первым символом после **[** является **!**, с этой конструкцией сопоставляется любой символ, не входящий в квадратные скобки.
;	Разделитель команд: **p1; p2** - выполнить **p1**, затем **p2**.
&	Выполняет предшествующую команду в фоновом режиме
`...`	Инициирует выполнение команд(ы) в **...**; **`...`** заменяется на полученный в результате выполнения стандартный выходной поток
(...)	Инициирует выполнение команд(ы) **...** в порожденном командном интерпретаторе
$1,$2,...$9	Заменяются аргументами командного файла
$var	Значение *переменной* (*ключевого параметра*) **var** в сеансе
${var}	Значение **var**: исключает коллизии в случае конкатенации переменной с последующим текстом
****	**\c** - использовать непосредственно символ **c**, \перевод строки - отбрасывается
'...'	Непосредственное использование того, что в кавычках
"..."	Непосредственное использование, но после того, как будут интерпретированы метасимволы **$**, **`...`** и \
#	Начало комментария
var=value	Присваивает значение **value** переменной **var**
p1&&p2	Выполнить **p1**; в случае успеха выполнить **p2**
p1||p2	Выполнить **p1**; в случае неудачи выполнить **p2**

Примечание

Большинство метасимволов будет рассматривается по ходу изложения. Здесь мы остановимся на тех из них, которые используются для генерации имен файлов и экранирования.

Перед выполнением команды каждое слово-аргумент команды просматривается в поисках метасимволов *****, **?** и **[**. Если в слове есть один из этих символов, слово рассматривается как *шаблон* (обратите внимание на синтаксические отличия от шаблонов **ed**). Такое слово заменяется отсортированными в алфавитном порядке именами файлов, соответствующих шаблону. Если ни одно их имен файлов не соответствует шаблону, слово оставляется без изменений. Символ **.** в начале имени файла или сразу после /, а также сам символ /, должны сопоставляться **явно**.

При таком количестве метасимволов интерпретатора необходимо иметь возможность экранировать специальный символ от интерпретации. Для этого можно использовать апострофы, кавычки или обратную косую. При этом кавычки одного вида могут экранировать кавычки другого вида. Обратите внимание, что кавычки **""**, в отличие от апострофов, не экранируют строку полностью - интерпретатор заглядывает внутрь кавычек в поисках **$**, **`...`** и \.
В кавычках могут содержаться переводы строк, пробелы, табуляции, символы **;**, **&**, **(**, **)**, |, ^, **<** и **>**. Задавая имя файла в кавычках, можно создать файлы с такими нетривиальными символами в именах. Впрочем, делать это не рекомендуется, так как работать с ними будет явно неудобно.

Создание сценариев

Имея последовательность команд, которую придется многократно использовать, преобразуем ее для удобства в "новую" команду со своим именем и будем использовать ее как обычную команду. Предположим, что вам как администратору предстоит часто подсчитывать количество пользователей, работающих в настоящий момент в системе, при помощи простого конвейера

```
$ who | wc -l
```

и для этой цели нужна новая программа **nu**.
Первым шагом будет создание обычного текстового файла, содержащего текст конвейера. Можно воспользоваться вашим любимым текстовым редактором, а можно проявить изобретательность:

```
$ echo 'who | wc -l' >nu
```

Интерпретатор является такой же программой, как редактор, **who** или **wc**. Раз это программа, ее можно вызвать и переключить ее входной поток. Итак, запускаем интерпретатор с входным потоком, поступающим из файла **nu**, а не с терминала:

```
$ who
oracle    pts000      Aug 20 10:08
root      console     Aug 20 09:03
intdbi    pts004      Aug 20 12:45
$ cat nu
who | wc -l
$ sh < nu
     3
$
```

Результат получился таким же, как и при вводе команды с терминала. Как и многие другие программы, интерпретатор обрабатывает файл, если он указан в качестве аргумента; вы с тем же успехом могли бы задать:

```
$ sh nu
```

На самом деле, этот вызов отличается, так как входной поток **sh** остается связанным с терминалом.
Не хотелось бы вводить **sh** каждый раз, кроме того, это создает различие между командами, написанными на языке shell, и другими выполняемыми файлами. Поэтому, если текстовый файл предназначен для выполнения, то интерпретатор считает, что он состоит из команд (интерпретатор **csh** требует, чтобы файл начинался с #).

Примечание

Если в первой строке выполняемого текстового файла указано:

```
#!/полный_путь_к_программе опции_программы
```

то данный текстовый файл будет интерпретироваться указанной программой, при вызове которой будут установлены заданные опции. Так можно выполнять, например, программы командного интерпретатора **csh**, не выходя из **sh**. Точно так же можно автоматически вызвать и интерпретаторы других языков сценариев, например, Perl.

Все, что вам нужно сделать, это объявить файл nu выполняемым, задав:

```
$ chmod +x nu
```

а затем вы можете вызывать его посредством

```
$ nu
```

На самом деле, при выполнении команды **nu** создается новый процесс интерпретатора (порожденный интерпретатор), который и выполняет команды, содержащиеся в **nu**. Чтобы команда выполнялась в том же интерпретаторе, необходимо поставить перед вызовом команды точку (**.**). Заметьте, что

```
$ . nu
```

выполняется быстрее, чем простой вызов **nu**.
В большинстве программ надо использовать аргументы - файлы, флаги и т.д. В соответствии с синтаксисом командной строки, это можно сделать, передавая их после имени команды через пробелы.
Предположим, необходимо программу cx для установки права доступа к файлу на выполнение, так что

```
$ cx nu
```

есть сокращенная запись для

```
$ chmod +x nu
```

Создать такой сценарий не сложно. Остается только один вопрос - как получить в программе доступ к имени файла-аргумента. Для этого в командном интерпретаторе используются *позиционные параметры*.
При выполнении командного файла, каждое вхождение **$1** заменяется первым аргументом, **$2** - вторым и так далее до **$9**. **$0** заменяется именем выполняемой команды. Поэтому, если файл **cx** содержит строку

```
chmod +x $1
```
то при выполнении команды
```
$ cx nu
```
порожденный интерпретатор заменит **$1** первым аргументом команды **nu**.
Значения позиционным параметрам присваиваются при вызове сценария, при вызове функции в сценарии или явно, с помощью команды **set**.
Как быть, если нужно работать с несколькими аргументами, например, заставить программу **cx** делать выполняемыми несколько файлов? Можно включить девять аргументов в командный файл (явно можно задавать только девять аргументов, так как конструкция **$10** распознается как "первый аргумент, за которым следует 0"):
```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```
Если пользователь такого командного файла задаст менее девяти аргументов, то оставшиеся окажутся пустыми строками и только настоящие аргументы будут переданы **chmod** порожденным интерпретатором.
Но такое решение не подходит, если количество аргументов превышает девять.
Интерпретатор предоставляет специальный параметр **$***, который заменяется всеми аргументами команды, независимо от их количества. С учетом этого, правильное определение **cx** будет таким:
```
chmod +x $*
```
Все позиционные и специальные параметры, поддерживаемые командным интерпретатором, представлены в табл. 24.

Таблица 24. Позиционные и специальные параметры командного интерпретатора

Параметр	Назначение
$0	Имя выполняемой команды
$1,$2,...$9	Заменяются аргументами командного файла
$#	Количество аргументов
$*	Все аргументы, передаваемые интерпретатору. **"$*"** является единым словом, образованным из всех аргументов, объединенных вместе с пробелами.
$@	Аналогично **$***. **"$@"** идентично аргументам: пробелы в аргументах игнорируются, и получается список слов, идентичных исходным аргументам.
$-	Флаги, установленные в интерпретаторе.
$?	Значение, возвращенное последней выполненной командой (*статус выхода*).
$$	Номер процесса интерпретатора.
$!	Номер процесса последней команды, запущенной асинхронно с помощью **&**.

Переменные и присваивание

Подобно большинству языков программирования, командный интерпретатор имеет *переменные*, которые называют еще *ключевыми параметрами* (поскольку они предаются *по имени* - ключу).
Переменные можно создавать, изменять и выбирать их значения. Для присваивания значения переменной применяется конструкция:
```
переменная=значение
```
Обратите внимание на отсутствие пробелов до и после знака присваивания. Вспомните, что командный интерпретатор считает пробелы разделителями слов. Если поставить пробел после знака присваивания, то интерпретатор не изменит значения переменной, но будет искать команду с именем **значение**.
Присваиваемое значение должно выражаться одним словом, и его следует взять в кавычки, если оно содержит метасимволы, которые не нужно обрабатывать.
Создание (определение) переменной происходит при первом присваивании ей значения. Переменные не нужно явно объявлять до момента их использования. Если переменная не объявлена (не получила значения), при обращении к ней будет получена пустая строка. Все переменные в командном интерпретаторе - строковые, поэтому тип переменной задавать не надо. Некоторые команды интерпретируют строки как числа.
Многие переменные, как, например, **PATH**, имеют специальное значение для интерпретатора. По традиции, такие переменные называют *встроенными* и обозначают прописными буквами, а обычные переменные - строчными. Основные встроенные переменные представлены в табл. 25.

Таблица 25. Встроенные переменные командного интерпретатора

Переменная	Значение
$HOME	Начальный каталог пользователя.
$PATH	Путь для поиска выполняемых команд.
$CDPATH	Путь поиска для команды **cd**.

$IFS	Список символов, разделяющих слова в аргументах
$MAIL	Файл почтового ящика. Командный интерпретатор информирует пользователя о получении почты в указанный файл.
$MAILCHECK	Эта переменная определяет, как часто (в секундах) интерпретатор будет проверять поступление почты в файл, определяемый переменной **MAIL**. По умолчанию принято значение 600 секунд. При установке в 0, интерпретатор будет проверять почту перед каждой выдачей строки-приглашения.
$PS1	Строка-приглашение, по умолчанию принята **'$ '**
$PS2	Строка-приглашение при продолжении командной строки, по умолчанию принята **'> '**

Типичным примером использования переменных является хранение в них длинных строк, таких как имена файлов:

```
$ pwd
/home/intdbi/dosapps/doc/book/unix/shell
$ dir=`pwd`
$ cd
$ ln $dir/cx .
...
$ cd $dir
$ pwd
/home/intdbi/dosapps/doc/book/unix/shell
```

Встроенная команда интерпретатора **set**, при вызове без параметров показывает значение всех определенных переменных.

Присваивание значения переменной при вызове

Переменные называют *ключевыми параметрами*, поскольку им можно передавать значение по имени при вызове. Рассмотрим пример:

```
$ echo 'echo $x' >echox
$ cx echox
$ echo $x
Hello
$ echox
$ x=Hi echox
Hi
```

По умолчанию ключевые параметры можно передавать только до имени команды. Если установить флаг интерпретатора **-k** (**set -k**), то можно будет передавать значения переменным и после имени команды.

Экспортирование переменных в среду

Каждый экземпляр командного интерпретатора имеет свой набор переменных, размещаемых в отдельной области памяти. Если необходимо, чтобы определенная переменная в порожденных процессах имела конкретное значение, необходимо *экспортировать* ее в *среду*. Такая переменная называется переменной среды.

Для всех экспортированных переменных при запуске порожденного процесса создаются их локальные копии с теми же значениями. Рассмотрим пример:

```
$ x=Hello
$ export x
$ PS1='new$ ' sh
new$ echo $x
Hello
new$ x='Good Bye'
new$ echo $x
Good Bye
new$ exit
$
$ echo $x
Hello$
```

Изменение значение переменной в порожденном интерпретаторе не влияет на ее значение в родительском интерпретаторе.

Для просмотра значений всех переменных среды предназначена команда **env**.

Циклы в командном интерпретаторе

Командный интерпретатор поддерживает циклическую обработку. Чаще всего на практике используется цикл **for** - цикл *по списку слов*. Он описан в следующем подразделе.

Обратите внимание, что выделенные **полужирным** ключевые слова должны быть первым словом команды, т.е. первым словом в строке или идти сразу после точки с запятой.

Цикл for

Цикл **for** имеет следующий синтаксис:
<цикл for> ::=
for <имя переменной> [in <список слов>] **do** <команды> **done**
<список слов> ::=
<слово>{<пробел> <слово>}
<команды> ::=
<команда> {<; или перевод строки> <команда>}
Переменная последовательно получает значение очередного слова из списка, и для этого значения выполняются команды в теле цикла. Цикл завершается, когда пройден весь список слов. По умолчанию в качестве списка слов используются аргументы командной строки.
Рассмотрим пару примеров таких циклов:

```
$ for i in 1 2 3 4 5
> do
>               echo $i
> done
```

Обратите внимание, что командный интерпретатор распознает цикл, выдает вторичное приглашение, и выполняет цикл только после его завершения ключевым словом **done**.
Список слов для цикла обычно порождается динамически. Например, путем раскрытия шаблонов имен файлов:

```
$ for i in *.c *.h
> do
>   echo $i
>   diff -b old/$i $i
>   echo
> done | pr -h "diff `pwd`/old `pwd`" | lp &
[4] 1430
```

Можно также порождать его командой, подставляя ее результаты:

```
$ for i in `pick *.c *.h`
> do
>   echo $i:
>   diff -b old/$i $i
> done | pr | lp
```

Операторы цикла while и until

Командный интерпретатор поддерживает также традиционные циклы по условию со следующим синтаксисом:
<оператор while> ::=
while <команды> **do** <команды> **done**
<оператор until> ::=
until <команды> **do** <команды> **done**
Выполняются **команды**, задающие условие, и проверяется код возврата последней из них. Если это ноль (**истина**), выполняются команды в теле цикла **while** или завершается выполнение цикла **until**. Если это не ноль (**ложь**), завершается работа цикла **while** или выполняется очередная итерация цикла **until**.
На основе этих циклов часто создаются программы-"*следилки*", работающие бесконечно:

```
$ cat watchfor
# watchfor: watching for log ins and log outs...
PATH=/usr/bin
new=/tmp/wfor1.$$
old=/tmp/wfor2.$$
>$old                                           # создает пустой файл

while :                                         # бесконечный цикл
do
        who >$new
        diff $old $new
        mv $new $old
        sleep 60
done | awk ' />/ { $1 = "in:   "; print }
             /</ { $1 = "out:          "; print }'
$
```

Оператор выбора

Командный интерпретатор поддерживает выполнение того или иного блока команд в зависимости от значения некоторого слова. Для этого предлагается оператор **case** со следующим синтаксисом:
<оператор выбора> ::=
case <слово> in
<описание варианта>) <команды> ;;
{<описание варианта>) <команды> ;; }
esac
<описание варианта> ::=
<шаблон> { | <шаблон>}
<команды> ::=
<команда> {<разделитель> <команда>}
<разделитель> ::=
<перевод строки> | ;
Слово (обычно - значение переменной) сравнивается последовательно с шаблонами. Если произошло сопоставление (**по правилам сопоставления шаблонов имен файлов**) выполняются команды, соответствующие данному варианту и оператор завершается. Учтите, что шаблон ***)** сопоставляется с любым словом, и, тем самым, задает вариант по умолчанию.
В шаблонах оператора **case** символы **.** и /, в отличие от шаблонов имен файлов, не обязательно задавать явно.

Условный оператор

Командный интерпретатор поддерживает условный оператор следующего общего вида:
<условный оператор> ::=
if <команды> **then** <команды>
{**elif** <команды> **then** <команды>}
[**else** <команды>]
fi
Выполняются команды после **if** и проверяется код возврата последней из них. Если это 0 (**истина**) выполняются соответствующие команды после **then** и выполнение оператора завершается. Если же это не 0 (**ложь**), то при наличии конструкций **elif** выполняются последовательно соответствующие команды-условия и, если они возвращают код 0, команды после **then**, а затем оператор завершается. Если ни одно из условий не было истинным, выполняются команды в части **else** и оператор завершается.
В качестве условия в условном операторе может использоваться любая команда. Однако, имеется стандартная команда для проверки условий в традиционном понимании. Это команда **test**, представленная в следующем разделе.

Проверка условий в командном интерпретаторе

Команда **test** имеет следующий синтаксис:
<команда test> ::=
test <выражение> | [<выражение>]
Выражение строится из примитивов, представленных в табл. 26, при необходимости, с помощью следующих операторов:

!	Унарный оператор отрицания.
-a	Бинарный оператор "и".
-o	Бинарный оператор "или".
(<выражение>)	Скобки для группировки. Учтите, что скобки распознаются командным интерпретатором, поэтому их надо брать в кавычки.

Таблица 26. Основные примитивы команды test

Примитив	Условие
-r файл	**файл** существует и доступен для чтения
-w файл	**файл** существует и доступен для записи
-x файл	**файл** существует и является выполняемым
-f файл	истина, если **файл** существует и является обычным файлом (не каталогом)
-d файл	**файл** существует и является каталогом
-h файл	**файл** существует и является символьной связью
-s файл	**файл** существует и не пуст
-t [дескриптор]	истина, если открытый файл с указанным **дескриптором** (по умолчанию, 1) ассоциирован с терминалом

-z s1	истина, если строка **s1** имеет нулевую длину
-n s1	истина, если строка **s1** имеет ненулевую длину
s1 = s2	истина, если строки **s1** и **s2** идентичны
s1 != s2	истина, если строки **s1** и **s2** не совпадают
s1	истина, если строка **s1** непустая
n1 -eq n2	сравнение целых чисел на равенство (=). Можно использовать также и другие сравнения: **-ne** (**!=**), **-gt** (**>**), **-ge** (**>=**), **-lt** (**<**) и **-le** (**<=**).

Рассмотрим пример использования условного оператора и команды **test**:

```
$ cat which
# which cmd: Безопасная версия сценария для выдачи каталога,
# из которого будет вызываться выполняемая программа

opath=$PATH
PATH=/usr/bin

# Это гарантирует использование настоящих команд
# echo, sed и test в любом случае!

case $# in
0)      echo 'Usage: which command' 1>&2; exit 2
esac

for i in `echo $opath | sed 's/^:/.:/
        s/::/:.:/g
        s/:$/:./
        s/:/ /g'`
do
        if test -x $i/$1
        then
                echo $i/$1
                exit 0                  # команда найдена
        fi
done
exit 1                                  # не найдена
$ which sed
./sed
$ which which
./which
```

Перехват и обработка сигналов

В программах командного интерпретатора можно перехватывать и обрабатывать сигналы. Для этого используется команда **trap**, устанавливающая с момента выполнения обработчик в виде последовательности команд (**одним словом**) для всех перечисленных сигналов. Эта команда имеет следующий синтаксис:

<оператор trap> ::=
    trap <последовательность команд> <список сигналов>
<список сигналов> ::=
    <сигнал> {<пробелы> <сигнал>}

Рассмотрим пример реализации команды **nohup**, позволяющей запустить программу так, чтобы она продолжала работать при выключении терминала:

```
$ cat nohup
# nohup: no kill and hangup
trap "" 1 15
if test -t 2>&1
then
        echo "Redirect stdout to 'nohup.out'"
        exec nice -5 $* >>nohup.out 2>&1
else
        exec nice -5 $* 2>&1
fi
$
```

Запрос информации у пользователя

Командный интерпретатор позволяет, при необходимости, запрашивать у пользователя информацию, которая помещается в указанную переменную. Для этого используется команда **read**:

```
$ read greeting
Hello, world!
$ echo $greeting
```

```
Hello, world!
$
```
На практике имеет смысл перед запросом выдать приглашение с помощью команды **echo**. Например, вот так:
```
$ cat pick
# pick: select arguments

PATH=/bin:/usr/bin

for i                                                   # for each argument, try $*, "$*" and "$@"
do
        echo -n "$i? " > /dev/tty
        read responce
        case $responce in
        y*)     echo $i;;
        q*)     break
        esac
done </dev/tty
$
```
Представленная выше программа **pick** выдает каждое указанное в качестве аргумента слово в отельной строке со знаком вопроса и требует от пользователя подтвердить необходимость его выдачи в стандартный выходной поток. Поскольку эта программа может использоваться в других сценариях, входной и выходной потоки которых перенаправлены, она взаимодействует непосредственно с текущим терминалом (через устройство **/dev/tty**).

Вычисления в командном интерпретаторе

Вычисления можно выполнять с помощью любой программы, воспринимающей свои параметры как выражение, значение которого необходимо вычислить, и выдающей результат вычисления в стандартный выходной поток. Одна из таких программ, **expr**, рассмотрена далее. Но современные командные интерпретаторы включают встроенную команду для выполнения простейших арифметических действий. Это команда **let**:

<команда let> ::=
        let <аргумент> {<аргумент>}

Вот как ее можно использовать:
```
$ let a=5
$ echo $a
5
$ let a=a*a+34/2
$ echo $a
42
$ let "a = 7"
$ echo $a
7
```
Обратите внимание, что если вокруг знака равенства идут пробелы, необходимо брать выражение в кавычки. Команда **let** требует, чтобы выражение было одним словом. Кроме того, для обращения к значению переменной в этой команде **не нужно** использовать метасимвол **$**.

Команда expr

Одной из стандартных программ-калькуляторов является программа **expr**. Ее основные операторы представлены в табл. 27.

Таблица 27. Основные операторы, распознаваемые командой expr

Оператор	**Результат**
выр1 | выр2	Возвращает значение первого выражения, если оно не пустое и не равно 0, иначе, возвращает значение второго выражения.
выр1 \& выр2	Возвращает значение первого выражения, если оба выражения - не пустые и не равны 0, иначе, возвращает 0.
выр1 { +, - } выр2	Складывает или вычитает целочисленные аргументы.
выр1 { *, /, % } выр2	Умножает, делит или возвращает остаток от деления для целочисленных аргументов.
length строка	Возвращает длину строки.

Рассмотрим простой пример вычисления с помощью **expr**:
```
$ a=5
$ echo $a
5
```

```
$ a=`expr $a \* $a + 34 / 2`
$ echo $a
42
```
Обратите внимание, что между элементами выражения надо указывать пробелы.

Функции в командном интерпретаторе

Стандартным способом разбиения программ на модули в командном интерпретаторе является оформление необходимых действий в виде отдельного выполняемого файла с программой командного интерпретатора - создание новой команды. Тем не менее, для некоторых модулей такой подход может оказаться неэффективным и избыточным, так как модули могут не представлять самостоятельного значения вне программы, в которой они используются. Поэтому в современных версиях командных интерпретаторов предлагается возможность создавать и вызывать *функции*.

Синтаксис определения функции

Для определения функций используется ключевое слово **function**. Функции читаются и хранятся внутренне командным интерпретатором. Функции выполняются как команды, причем аргументы передаются **как позиционные параметры**. Синтаксис определения функции следующий:
<определение функции> ::=
function <идентификатор> { <список команд> } |
<идентификатор> () { <список команд> }
где **список команд** задает команды, выполняемые в качестве тела функции. Команды обычно разделяются точкой с запятой или переводами строк.

Выполнение и использование функций

Функции выполняются вызвавшим их процессом и используют все его файлы и текущий рабочий каталог. Сигналы, перехватываемые вызывающим процессом, внутри фунции обрабатываются стандартным образом. Сигналы, не перехватываемые или игнорируемые функцией, прекращают ее выполнение и передаются вызвавшей команде.
Обычно переменные совместно используются вызывающей программой и функцией. Однако, специальная команда **typeset**, используемая внутри функции, позволяет определять локальные переменные, область действия которых - текущая функция и все вызываемые ею функции.
Для выхода из функции используется специальная команда **return**. В случае ошибки в функции, управление передается вызывающей команде.
Идентификаторы определенных функций можно получить с помощью опций **-f** или **+f** специальной команды **typeset**. Текст функций показывается при использовании опции **-f**. Определение функции можно отменить с помощью опции **-f** специальной команды **unset**.
Обычно при выполнении сценария командным интерпретатором никакие функции не заданы. Опция **-xf** команды **typeset** позволяет экспортировать функцию для использования сценариями, выполняемыми без отдельного вызова интерпретатора. Функции, которые должны быть определены для всех вызовов интерпретатора, необходимо задавать в файле начального запуска с помощью опций **-xf** команды **typeset**.
Рассмотрим классический пример итеративной реализации функции вычисления факториала:

```
# test.sh - test shell functions

factorial () {
  typeset i
  typeset n

  i=1; n=1
  while [ $i -le $1 ]
  do
    let n=n*i
    let i=i+1
  done
  echo $n
  return
}

a=`factorial $11`
echo $a
```

При вызове эта программа, как и ожидалось, вычислит факториал своего первого параметра:

```
bash$ test.sh 5
120
```

Часто в виде функций оформляется выдача сообщений о параметрах вызова программы. В любом случае, если задача может быть разбита на подзадачи, решение этих подзадач имеет смысл оформлять в виде

отдельной команды, если они полезны не только в контексте решаемой задачи, или в виде функции в противном случае.

Файлы начального запуска командного интерпретатора

Стандартная среда для работы командных интерпретаторов задается в *файлах начального запуска*, которые автоматически выполняются в начальном интерпретаторе с помощью команды точка (**.**), т.е. без порождения. Файлы начального запуска размещаются в начальном каталоге пользователя и называются **.profile** (**sh**, **ksh**) или **.bash_profile** (**bash**). Переменные, составляющие среду, в файле начального запуска надо экспортировать.
Рассмотрим пример содержимого файла начального запуска:

```
INFORMIXDIR=/usr/inf.731
INFORMIXSERVER=onarturo7
ONCONFIG=onconfig
SQLHOSTS=sqlhosts
PATH=$PATH:$INFORMIXDIR/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$INFORMIXDIR/lib/esql:$INFORMIXDIR/lib
DB_LOCALE=ru_ru.8859-5
CLIENT_LOCALE=ru_ru.8859-5
KAIOON=1
NODEFDAC=YES
DBDATE=DMY4/
DBTIME='%H:%M:%S %d/%m/%Y'
DBMONEY=.4
export KAIOON
export INFORMIXDIR INFORMIXSERVER LANG PATH ONCONFIG SQLHOSTS
export DBDATE DBTIME DBMONEY NODEFDAC
export DB_LOCALE CLIENT_LOCALE
```

Управление заданиями

Управление заданиями - это механизм для отслеживания процессов, которые порождаются из текущего сеанса работы с терминалом. Можно запустить любое число заданий. Они могут работать, завершиться или находиться в других состояниях. Для управления заданиями можно использовать несколько команд, чтобы проследить результаты их работы или потребовать от системы уведомления об окончании задания.

Запуск задания в фоновом режиме

Фоновый режим позволяет продолжить использование сеанса работы с терминалом, пока выполняется команда. Для запуска команды в фоновом режиме, достаточно к команде добавить символ амперсанд (**&**).
Командный интерпретатор вернет номер задания и идентификатор процесса:

```
$ make &
[2] 254
$
```

Если задание не требует от пользователя ввода, оно продолжает свою работу до полного завершения. Если команде нужен ввод, она переходит в состояние ожидания, на экран выводится соответствующее уведомление, которое выглядит примерно так:

```
[1] + Suspended (tty input) programm 0
```

В данном случае в ожидании ввода приостановилось выполнение программы **programm**. Пользователю необходимо перевести из фонового режима в привилегированный и выполнить ввод.

Просмотр состояния заданий

С помощью команды **jobs** пользователь имеет возможность просмотреть состояние своих заданий и получит список всех заданий запущенных в сеансе работы с терминалом.

```
$ jobs
[1]   Stopped (user)        du
[2]-  Stopped (user)        du -a /home/intdbi
[3]+  Stopped (user)        du -r /home/intdbi
$
```

Команда **jobs** принимает два флага. Флаг **-l** включает идентификатор процесса с номером задания.

```
$ jobs -l
[1]   1351 Stopped (user)        du
[2]-  1381 Stopped (user)        du -a /home/intdbi
[3]+  1383 Stopped (user)        du -r /home/intdbi
$
```

Флаг **-p** заменяет номер задания на идентификатор процесса.

```
$ jobs -p
```

```
1351
1381
1383
$
```
Идентификатор процесса может использоваться при обращении к команде **ps**.

Номера заданий

Номер задания позволяет командному интерпретатору наблюдать за процессами. Его можно рассматривать как головной элемент группы процессов, поскольку пользовательское задание порождает любые команды либо в конвейере, либо, как подзадания.

Перевод задания в привилегированный режим

Команда **fg** переводит задания в привилегированный режим. При наличии приостановленного задания, его можно сделать привилегированным (перевести на передний план) с помощью команды **fg #номер_задания** (или **fg номер_задания** в **bash**). После этого задание либо выведет на экран сообщение о том, что ему нужно от терминала, либо будет принимать ожидаемый ввод. Переведя задание в привилегированный режим, можно приостановить его выполнение, нажав комбинацию калвиш **Ctrl-Z**, и заняться им позже.
Любое задание из списка, предоставленного командой **jobs**, доступно, если пользователь захочет сделать его привилегированным, даже в том случае, когда оно уже работает в фоновом режиме. Если в этом списке приведено только одно задание, то при использовании команды **fg** пользователю не нужно задавать его номер. Если номер задания не задан, предполагается текущее задание.

Перевод задания в фоновый режим

С помощью команды **bg** можно возобновить в фоновом режиме работу приостановленного или остановленного задания. Для этого нужно указать соответствующий номер задания, после чего оно перейдет в фоновый режим, и будет работать до своего завершения, или пока ему снова не потребуется ввод с терминала.

Команда ожидания завершения процесса

Это последняя существенная команда управления заданиями. При вводе **wait** приостанавливается работа командного интерпретатора до тех пор, пока не будут завершены все фоновые задания. Сюда входят и любые остановленные задания, поэтому при вводе **wait** стоит убедиться, что все фоновые задания работают. Команда **wait** может также принимать в качестве параметра номер задания. В этом случае командный интерпретатор приостанавливается до тех пор, пока не завершится выполнение указанного задания.

Основные утилиты

Помимо рассмотренных выше команд, ОС UNIX предлагает десятки полезных утилит и средств. Основные утилиты для обработки текстов представлены в табл. 27, а клиенты основных сетевых служб - в табл. 28. Полезные команды для создания резервных копий представлены в табл. 29.

Обработка текстов

Таблица 27. Основные утилиты обработки текстов

Утилита	Назначение
awk	Язык обработки шаблонов. Позволяет выполнять произвольную программу при выявлении в тексте определенной строки, соответствующей шаблону. По синтаксису аналогичен C. Содержит множество встроенных функций. Используется для обработки и преобразования текстовых данных, состоящих из столбцов и строк, а также для построения отчетов и анализа журналов.
diff	Команда, сравнивающая два файла и выдающая найденные различия в разных форматах.
ed	Стандартный строчный текстовый редактор. Воспринимает команды из стандартного входного потока, изменяет файлы и часто используется в сценариях.
ex	Расширенная версия редактора **ed**. Поддерживает множество установок, которые можно запоминать для каждого пользователя в файле **.exrc** в его начальном каталоге.

head	Выдает указанное количество начальных строк из файла.
more	Позволяет просматривать файл постранично в обоих направлениях, искать в нем по шаблону.
pr	Форматирует файл или входной поток для печати, разбивая его на страницы и снабжая, при необходимости, заголовками.
sed	Потоковая версия редактора **ed**. Позволяет эффективно выполнять поиск и замену в стандартном входном потоке или указанных файлах.
tail	Выдает указанное количество конечных строк из файла.
tr	Преобразовывает символы во входном потоке, заменяя одни цепочки на другие. Поддерживает весь набор символов.

Работа в сети

Таблица 28. Основные сетевые команды

Утилита	Назначение
ftp	Клиент для обмена файлами с удаленной машиной по сети. Позволяет просматривать каталоги, создавать каталоги на удаленной машине, загружать и выгружать файлы с помощью стандартного набора команд. Допускает автоматизацию операций по обмену файлами.
netstat	Выдает разнообразную статистику о работе сети, содержимое таблиц маршрутизации и т.д.
ping	Посылает специальный пакет **ICMP**, требующий ответа от удаленного сервера. Позволяет проверить доступность удаленного хоста и скорость работы сети. Обычно доступна только пользователю **root**.
rlogin	Программа удаленной регистрации. Позволяет работать с удаленной машиной так же, как с локальной. Поддерживает *доверительные отношения*.
rsh	Удаленный командный интерпретатор. Позволяет выполнить любую команду интерпретатора на удаленной машине и получить ее выходной поток на локальной. Поддерживает доверительные отношения.
ssh	Защищенный командный интерпретатор, функционально аналогичный программам **telnet**, **rsh** и **rlogin**, но передающий пароли и данные **в зашифрованном виде**.
telnet	Программа удаленного подключения к указанной сетевой службе. Обычно используется для удаленной регистрации. Не поддерживает доверительные отношения.
traceroute	Программа трассировки пакетов. Показывает маршрут, по которому будут направляться пакеты на указанный удаленных хост и скорость передачи на каждом из переходов. Обычно доступна только пользователю **root**.

Резервное копирование и восстановление

Таблица 29. Основные средства резервного копирования и восстановления

Утилита	Назначение
compress	Сжимает (упаковывает) указанный файл, обычно удаляя исходный вариант. Сжатые файлы обычно имеют суффикс **.Z** и разжимаются командой **uncompress**.
cpio	Команда создания архивов. Помещает все указанные файлы, включая содержимое подкаталогов, в архив, выдаваемый в стандартный выходной поток. Получает архив из входного потока и раскрывает в текущем каталоге. Поддерживает различные платформы и форматы, в том числе, формат архивов **tar**.
dd	Команда копирования блоков данных с одного файла или устройства в другой. Позволяет выполнять преобразования при копировании.
gzip	Утилита GNU для сжатия указанного файла. Упаковывает файлы лучше, чем **compress**. Работает на всех платформах. Сжатые файлы обычно имеют суффикс **.gz** и разжимаются командой **ungzip** (**gzip -d**).
tar	Утилита архивирования. Помещает все указанные файлы, включая содержимое подкаталогов, в архив, записываемый в указанный файл. Создает необходимые каталоги и файлы при разархивировании. Первоначально предназначалась для архивирования на ленту.

Программа практических занятий

Занятие 1. 2 часа.

1. Регистрация в системе. Изменение пароля (**passwd**). Просмотр информации о работающих пользователях (**who**).

2. Работа со справочным руководством (**man**).

3. Просмотр содержимого файлов (**cat**, **more**).

4. Создание файлов различных типов (**cat**, **mkdir**, **mknod**, **ln**). Копирование, перемещение и удаление файлов и каталогов (**cp**, **mv**, **rm**, **rmdir**).

5. Просмотр информации о файлах, изменение прав доступа и владельца файлов (**ls**, **chmod**, **chown**, **chgrp**, **umask**).

6. Просмотр информации о процессах и установка поправки приоритета (**ps**, **nice**).

7. Посылка сигналов процессам (**kill**).

Занятие 2. 2 часа.

1. Поиск файлов по различным критериям (**find**).

2. Использование утилиты grep для поиска в текстах по образцу

3. Практическая работа с редактором **vi**.

4. Управление заданиями (**jobs**, **bg**, **fg**).

Занятие 3. 2 часа.

1. Создание простых конвейеров. Запуск процесса в приоритетном и фоновом режиме. Динамическое формирование командной строки.

2. Использование цикла **for** в командном интерпретаторе.

3. Проверка условий в командном интерпретаторе.

4. Создание цикла по счетчику в командном интерпретаторе.

5. Создание и использование функций в командном интерпретаторе.

6. Создание простого сценария командного интерпретатора.

7. Файлы начального запуска командного интерпретатора. Настройка среды.

8. Использование утилит **tar**, **dd**, **gzip** для резервного копирования.

9. Практическое использование утилит **telnet** и **ftp**.

Литература

1. Керниган Б.В., Пайк Р. "UNIX - универсальная среда программирования" -М.: Финансы и статистика, 1992. - 304 с.

2. Дегтярев Е.К. "Введение в UNIX" - М.; МП "Память", 1991. - 96 с.

3. Топхем Д., Чыонг Х.В. "Юникс и Ксеникс" -М.: Мир, 1988. - 392 с.

4. Кэвин Р., Фостер-Джонсон Э. "UNIX: справочник" -СПб: Питер Ком, 1999. - 384 с.

5. Немет Э., Снайдер Г., Сибиасс С., Хейн Т.Р. "UNIX: руководство системного администратора" -К.: BHV, 1997 - 832 с.

6. Робачевский А.М. "Операционная система UNIX" -СПб.: БХВ - Санкт-Петербург, 1999. - 528 с., ил.

7. Дайсон П. "Операционная система UNIX. Настольный справочник" -М.: ЛОРИ, 1997. - 400 с.

8. Беляков М.И., Рабовер Ю.И., Фридман А.Л. "Мобильная операционная система: Справочник" - М.: Радио и связь, 1991. - 208 с.

9. Анонимный автор. "Максимальная безопасность в Linux" -К: Издательство "ДиаСофт", 2000, - 400 с.

10. Шенк Т. "Red Hat Linux для системных администраторов. Энциклопедия пользователя" -К: Издательство "ДиаСофт", 2001, - 672 с.

11. Эбен М., Таймэн Б. "FreeBSD. Энциклопедия пользователя" -К: ООО "ТИД "ДС", 2002, - 736 с.

Примечание

К переводу последних трех книг на русский я имел непосредственное отношение как переводчик и научный редактор.

Источники информации в Internet

История UNIX

http://perso.wanadoo.fr/levenez/unix/
Генеалогическое дерево версий UNIX вплоть до настоящего времени.
http://www.faqs.org/faqs/unix-faq/faq/part6/

http://www.crackmonkey.org/unix.html

http://catb.org/~esr
Персональная страница Эрика Реймонда. Обратите внимание на его новую книгу "The Art of UNIX Programming"!

Справочные руководства по командам:

http://opennet.ru
Ну, кто же их не знает... Основной источник информации о UNIX на русском языке.
http://citforum.ru
Центр информационных технологий моего родного МГУ. Источник огромного объема информации по ИТ на русском. Есть и переводы страниц справочного руководства BSD.
http://ln.com.ua/~openxs/projects/man
Проект "Страницы справочного руководства ОС UNIX на русском". Мой.
http://www.gnu.org/manual/manual.html
Справочные руководства по утилитам проекта GNU

Другие источники информации

http://catb.org/~esr
Персональная страница Эрика Реймонда. Это один из идеологов разработки свободно распространяемого программного обеспечения. Его работы надо читать как Евангелие...
http://linuxnews.ru
Новости Linux на русском. Полезная документация. Форум
http://www.freebsd.org
Основной сайт ОС FreeBSD
http://docs.sun.com
Документация по ОС Solaris
http://www.gnu.org
Сайт проекта GNU