

Основы UNIX и Linux

Р.С.Клочков, Н.А.Коршенин

Last modified: Oct 30 2004

Основы интерфейса командной строки

- У меня UNIX не грузится, ошибку выводит
- Какую?
- login:

После загрузки Linux возможны три варианта. Либо сразу загрузится графическая среда, либо будет запрошено имя пользователя в графическом режиме, либо будет запрошено имя пользователя в текстовом режиме (та самая строка login: из эпиграфа).

Если Вы оказались в графическом режиме, то запустите терминал (В меню выберите "Системные" -> "Терминал").

Теперь Вы находитесь в текстовом режиме управления операционной системой. Этот режим еще называется shell (в переводе с английского "оболочка") и существует во всех UNIX системах. Ещё его называют режимом командной строки (command line interface). Все, что можно здесь делать, это вводить команды и читать результат их выполнения. Например, если хотите узнать текущие дату и время, то введите команду

```
$ date
```

```
Mon Oct 4 19:47:03 MSD 2004
```

Если же, к примеру, Вы хотите создать в текущем каталоге файл foo.txt, введите команду

```
$ touch foo.txt
```

Заметьте, что эта команда ничего не выводит на экран при успешном выполнении. Это является общим принципом почти всех команд UNIX: молча делать своё дело и ругаться лишь на ошибки. Если же надо что-нибудь вывести на экран при успешном выполнении, то для этого существует команда echo. Она просто выводит на экран все свои параметры. Например,

```
$ echo Hello world
```

```
Hello world
```

Кстати, обратите внимание на знак приглашения. Если это \$ как в примере, то Вы работаете от имени простого пользователя, а если знак приглашения #, то Вы вошли в систему под именем root и имеете все права (например всё уничтожить). Запомните: никогда не делайте под root'ом то, что можете сделать под другим пользователем. То есть нельзя под root'ом заходить в графический режим, получать почту, разрабатывать программы или ходить по интернету. Дело в том, что любая программа запущенная под root'ом при некорректном поведении (например, из-за ошибки в программе) нанесет гораздо больший ущерб, чем под любым другим пользователем.

Идеальным вариантом является ситуация, когда для каждой функции есть отдельный пользователь: один пользователь для почты, другой для написания программ, третий для хождения по интернету и т.д. В этом случае неправильно написанная программа не уничтожит Вашу почту, а вирус из интернета не сможет испортить или удалить Ваши данные.

В UNIX у команд есть возможность влиять друг на друга. Самый простейший способ такого влияния: передача кода возврата. Например, "создать файл и в случае удачи вывести OK" в UNIX выглядит как

```
$ touch foo.txt && echo OK
```

```
OK
```

Символы && означают, что вторую команду надо выполнять только в случае успешного завершения первой. Аналогично символы || выполняют вторую команду в случае неуспешного завершения первой:

```
$ touch foo.txt || echo Что-то не так
```

Недостатками кодов возврата являются ожидание окончания первой команды и то, что код возврата лишь одно число, в то время как иногда желательно передать несколько строк и не дожидаясь окончания первой программы. Для этих целей существуют потоки ввода-вывода и возможность их перенаправления. Уже упомянутая команда echo пишет в стандартный поток вывода (stdout). Можно записать что-нибудь в файл:

```
echo Это будет в файле > foo.txt
```

При этом исходное содержимое файла сотрется. Если надо дописать в конец файла можно воспользоваться такой командой

```
$ echo Это допишется в конец >> foo.txt
```

Чтобы убедиться, что команда выполнена правильно (все нормально, значит ничего не выводится на экран), можете воспользоваться командой `cat` (вывод файла). Например,

```
$ cat foo.txt
```

Это будет в файле

Это допишется в конец

Можно записать сразу всё содержимое файла командой `cat`

```
$ cat > foo.txt
```

Первая строка

Вторая строка

Ctrl+D

Здесь строки после команды вводятся с клавиатуры. Символ Ctrl+D находящийся отдельно на строке является символом конца файла и используется для окончания ввода.

Все сообщения об ошибках выводятся в стандартный поток ошибок (его называют `stderr` или поток 2). В обычной ситуации и то и другое - просто вывод на экран. Предположим, надо скомпилировать программу. Компилятор по ходу работы выводит сообщения обо всех ошибках в программе и они могут занять несколько экранов. Чтобы их можно было посмотреть можно выполнить компиляцию следующим образом

```
$ cc test.c 2> log.err
```

В этом случае будет перенаправлен поток ошибок. Затем их можно будет прочитать в файле `log.err`.

Если надо перенаправить не только ошибки но и стандартный вывод, то команда будет выглядеть как

```
$ cc test.c > log.err 2>&1
```

Последний кусок команды (`2>&1`) можно читать как "перенаправить поток ошибок в стандартный поток вывода".

Для демонстрации использования перенаправления стандартного ввода посмотрим, что делает команда `cat` без параметров. В этом случае она выводит всё, что ей передается на стандартный ввод. Если же выполнить команду

```
$ cat < foo.txt
```

Это будет в файле

Это допишется в конец

получаем также содержимое файла `foo.txt`.

Если какая-либо команда в процессе выполнения выводит нежелательные сообщения (например, нам нужен только статус возврата, а не сообщение об ошибке), то можно перенаправить ее вывод в специальный файл `/dev/null`. В этот файл все могут писать, но результат нигде не храниться и прочитать из него ничего нельзя. Можете считать его аналогом мусорной корзины или черной дыры.

```
$ touch /chk 2> /dev/null || echo Не получилось
```

Не получилось

При выполнении данной команды стандартного сообщения об ошибке Вы не увидите.

Теперь рассмотрим ещё одну удобную возможность: перенаправления вывода одной команды на ввод другой. Например, команда `ls` позволяет посмотреть список файлов в текущем каталоге. Но, предположим, что файлы копировались с windows-системы и их имена в неправильной кодировке. Есть программа, позволяющая перекодировать текст. Она называется `iconv`. И поставленную задачу можно решить двумя командами

```
$ ls > tmpfile
```

```
$ iconv -f cp1251 -t koi8-r < tmpfile
```

foo.txt

При этом команды выполняются последовательно.

Можно также заставить эти команды выполняться одновременно и без промежуточного файла

```
$ ls | iconv -f cp1251 -t koi8-r
```

foo.txt

Этот метод выполнения называется конвейер (в англоязычных текстах `pipe`). Так же как и на конвейере все процессы обрабатывают данные одновременно и данные поступают с выхода одного процесса на вход другого.

Если надо вывод команды направить не в стандартный ввод, а в параметры, используется обратный апостроф ` (слева от 1 на клавиатуре). Например, нужно вывести на экран описание (ls -l) несколько файлов, а список файлов лежит в другом файле. Тогда команда будет выглядеть как

```
$ ls -l `cat list`
```

где list - имя файла, где лежит список

В UNIX существует встроенная система документации. Например, если Вы хотите узнать, что делает команда ls, выполните команду

```
$ man ls
```

Выйти из документации можно кнопкой "q". По некоторым частям системы доступна более подробная документация, получить которую можно командой info. Например по ней самой получить документацию можно командой

```
$ info info
```

При работе с командной строкой необходимо знать, что некоторые символы являются специальными: \$, !, #, ?, *, а также упоминавшиеся выше <, >, |, &. Символ \$ используется для подстановки значения переменных shell'a. Например свое имя пользователя можно получить командой

```
$ echo $USER
```

```
monk
```

Заметьте, что большие и маленькие буквы в именах переменных различаются. "!" используется для обращения к истории команд, "#" позволяет использовать комментарии в строке:

```
$ echo это есть # этого нет
```

```
это есть
```

Символы "*" и "?" используются для подстановки в командную строку имен файлов. Строку с ними еще называют \em маской. "?" - значит один произвольный символ в имени файла, "*" - любое число символов. Например,

```
$ echo *
```

foo.txt выводит все файлы в текущем каталоге, кроме тех, у которых первый символ ".". Дело в том, что такие файлы считаются "невидимыми" и используются для хранения различных настроек. Однако, их всё равно можно увидеть. Для этого нужно использовать шаблон ".*".

```
$ echo .*
```

```
...
```

Команда echo ?o*.txt выведет все файлы, у которых вторая буква в имени "o" и заканчивающиеся на ".txt". Если файлов удовлетворяющих маске нет, то в командной строке остаётся сама маска. Например:

```
$ echo *d
```

```
*d
```

так как в текущем каталоге нет файлов, заканчивающихся на d.

Если необходимо использовать спецсимволы в их неспециальном значении, следует перед ними вставлять символ "\". Например:

```
$ echo \# \$ \%
```

```
# $ %
```

Еще один символ, который формально не является специальным, но который нежелательно ставить первым в имени файла - это "-". Дело в том, что имя этого файла всеми программами будет рассматриваться не как имя, а как ключ (параметр выполнения). Например, если попробовать создать файл "-" командой touch

```
$ touch -c
```

```
touch: file arguments missing
```

```
Try 'touch -help' for  
more information.
```

Тем не менее к таким файлам можно получить доступ при помощи пути вида "./-с" либо любого другого полного пути. Также в такие файлы можно писать из программ или команд вида

```
$ echo test > -c
```

При использовании подстановок типа * нужно всегда помнить, что в UNIX в именах файлов могут быть почти любые символы (все кроме "/" и нулевого символа). Например могут быть пробелы, символы конца строки, дефис, управляющие символы и прочее. Хотя использование в именах файлов не-латинских символов не рекомендуется, эту возможность всегда следует учитывать.

Работа с файлами и директориями

Введение в файловые системы

Данные на жестком диске в UNIX организованы в именованные блоки переменного размера, называемые файлами. Фактически, файл - это последовательность байт снабженная именем. Кроме имени у файла есть еще так называемые метаданные: права доступа, время создания, чтения, модификации. Представление о метаданных можно получить посмотрев на результат команды

```
$ stat foo.txt
```

```
File: `foo.txt`
Size: 40 Blocks: 8 IO Block: 4096 regular file
Device: 303h/771d Inode: 1739262 Links: 1
Access: (0644/-rw-r--r--) Uid: (1000/monk) Gid: (100/users)
Access: 2004-10-06 22:42:10.000000000 +0400
Modify: 2004-10-06 22:50:19.000000000 +0400
Change: 2004-10-06 22:50:19.000000000 +0400
```

Размер файлов измеряется не только в байтах, но и блоках. Дело в том, что для жесткого диска или дискеты практически нет разницы, будет ли прочитан один байт или 512. Большую часть времени занимает позиционирование головки. Поэтому данные с таких устройств читают блоками. Для простоты устройства файловой системы каждый блок на диске принадлежит только одному файлу. Причем в современных файловых системах блок файловой системы содержит несколько блоков жесткого диска. Так что, даже если, как в данном случае, размер файла всего 40 байт, то на диске он занимает 4096 байт (параметр IO Block как раз сообщает размер блока файловой системы). В параметре Blocks размер блока равен 512 байт, так как это минимальный размер блока на жестком диске и мы видим, что файл занимает 8 дисковых блоков = $8 \times 512 = 4096$ байт.

Еще одно незнакомое понятие: Inode. Дело в том, что в UNIX у одного файла может быть несколько имен. При этом данные должны быть общими. Так вот Inode и есть уникальный идентификатор данных файла, а параметр Links сообщает сколько имен (их еще называют ссылками) у данного файла. Device - номер устройства, где находится файл. Ниже можно увидеть права доступа и времена доступа или изменения файла. Создать дополнительную ссылку на файл можно командой ln. Например

```
$ ln foo.txt foo2.txt
```

Теперь, если что-нибудь записать в один из них, то эти данные окажутся и в другом. Эти имена являются полностью равнозначными. Если одну из них удалить, то реального удаления данных не произойдет, а произойдет оно только в случае удаления всех имен данного файла. Кроме этих ссылок (называемых ещё жесткими) существуют символические ссылки. Создаются они той же командой, но с ключом -s:

```
$ ln -s foo.txt sym.txt
```

Символическая ссылка хранит в себе имя файла-оригинала. То есть пока мы просто пишем/читаем/редактируем файл всё нормально, но если мы удалим файл foo.txt, то получить доступ к данным через sym.txt уже будет нельзя. Кроме того, если переместить sym.txt в другой каталог, то он будет ссылаться на файл foo.txt в том же каталоге. Если это нежелательно, то при создании ссылки необходимо использовать полное имя файла. Например:

```
$ ln -s /home/monk/foo.txt sym.txt
```

Кроме понятия "файл" было введено понятие "каталог". Так же как файл используется для доступа к группам данных, каталог объединяет группы файлов. Во время создания UNIX также была еще одна причина создания каталогов: при большом количестве файлов очень долго проводился поиск по имени файла при открытии файла. С точки зрения файловой системы каталог - это такой файл, в котором вместо данных хранятся записи вида "имя файла, номер Inode". Файлы и каталоги у которых нет вышестоящего каталога находятся в специальном каталоге без имени. Его называют "корневой каталог" (root directory). При указании \em полного имени имена вышестоящих каталогов разделяются символом "/". Например полное имя /boot/grub/grub.conf значит, что в корневом каталоге есть каталог boot, в котором есть каталог grub, в котором находится искомый файл grub.conf. В каждом каталоге кроме обычных файлов и подкаталогов всегда есть два специальных подкаталога "." и "..". "." - каталог указывающий на Inode текущего каталога, а ".." указывает на Inode родительского каталога. Например рассмотренный выше путь к файлу можно было записать как /boot/grub/./grub.conf или /boot/./boot/grub/grub.conf или еще тысячами разных способов. Из-за этих двух специальных каталогов количество ссылок у каталогов всегда не меньше двух. Одна - сам каталог, вторая - каталог "." в нем. И каждый подкаталог добавляет по дополнительной ссылке от своего каталога "..". При обращении к каталогу можно заканчивать полное имя на знак "/". Поэтому, если хотите получить какую-либо информацию по корневому каталогу необходимо использовать имя "/". Например

```
$ stat /
```

```
File: '/'
Size: 4096 Blocks: 8 IO Block: 4096 directory
Device: 303h/771d Inode: 2 Links: 20
Access: (0755/drwxr-xr-x) Uid: (0/root) Gid: (0/root)
Access: 2004-10-02 17:06:15.000000000 +0400
Modify: 2004-10-05 13:26:57.000000000 +0400
Change: 2004-10-05 13:26:57.000000000 +0400
```

Кроме файлов и каталогов есть еще несколько специальных типов файлов: блочные и символьные устройства, сокет и файлы FIFO. Файлы устройств в UNIX, как правило, обеспечивают доступ к различному оборудованию компьютера, в том числе виртуальному. Например, программа разбивки жесткого диска на разделы работает не с самим диском и не с системными вызовами напрямую, а со специальным файлом /dev/hda. Аналогично файл /dev/null является виртуальным устройством поглощающим любые данные, из файла /dev/random можно прочитать псевдослучайную последовательность произвольной длины, а все что будет записано в файл /dev/dsp будет звучать через звуковую карту компьютера.

Сокеты позволяют имитировать сетевое соединение через файл (чтобы не разделять код программы на сетевой и локальный), а файлы FIFO являются теми же конвейерами, но именованными. Практического применения для непрограммиста они не имеют.

Теперь обратимся к структуре файловой иерархии UNIX. То есть как должны называться каталоги и какие файлы в них должны находиться. Полную документацию по этому поводу можно узнать командой

```
$ man hier
```

Файлы по иерархии UNIX распределены согласно их назначению. Например, все системные исполняемые файлы, необходимые для загрузки должны быть в /sbin, а аналогичные пользовательские программ в /bin. В /lib находятся библиотеки, необходимые для вышеупомянутых программ, а в /etc - все конфигурационные файлы. Все изменяемые данные программы должны хранить либо в каталоге /var (если они нужны) или в /tmp (если это временные данные, которые в случае перезагрузки компьютера можно безболезненно потерять). Все, что некритично для загрузки компьютера должно находиться в каталоге /usr. В нем находится такая же структура каталогов, как и в корне (только var и tmp используются общие и etc обычно не используется). Все неизменяемые данные программ хранятся в /usr/share. Например, документацию можно найти в /usr/share/doc. Для программ, которые не относятся к базовой системе (например, компилируемых вручную) выделен каталог /usr/local с той же структурой, что и /usr. Настройки, зависящие от пользователя, хранятся в его домашнем каталоге. Также, как правило, пользователь имеет право писать только в свой домашний каталог и /tmp. Домашние каталоги пользователей традиционно создаются в каталоге /home и имеют вид /home/имя_пользователя. Эта структура позволяет легко и удобно прописывать пути запуска файлов, пути, по которым происходит поиск динамически подгружаемых библиотек и т.д. Данный подход, разумеется, имеет и недостатки: у каждой программы должно быть уникальное имя, неудобно использовать несколько версий одной и той же программы, нельзя простыми методами определить, какая программа какие файлы использует. Большинство этих проблем успешно решаются в Linux при помощи программ управления программными пакетами.

ls: посмотреть список файлов

Теперь более подробно рассмотрим команды управления файлами. Во-первых, команда ls: она позволяет посмотреть список файлов в любом каталоге либо их списке, а также увидеть различную информацию об этих файлах. Самая распространенная опция "-l" (от слова long) позволяет посмотреть дополнительную информацию о файлах:

```
$ ls -l
```

```
total 4
```

```
-rw-r--r-- 1 monk users 40 Oct 8 15:29 foo2.txt
```

total 4 значит, что перечисленные файлы занимают 4Кб. В строках по каждому файлу выдается информация: права доступа (-rw-r--r--), количество ссылок, владелец, группа-владелец, длина файла, время модификации и имя файла.

Создание файлов

Новые файлы создаются при записи в них какой-либо информации (команда touch именно это и делает: открывает файл для записи, затем сразу закрывает и у файла меняется время модификации но не меняется содержимое). Можно создавать файлы сразу командами вида

```
$ echo test > newfile.txt
$ echo > newfile.txt
$ > newfile.txt
$ echo -n > newfile.txt
```

Последние две команды эквивалентны: они либо создают пустой файл илбо обнуляют существующий. Ключ -n у команды echo запрещает ей добавлять в конец строки символ конца строки. Кстати, следует знать, что символы конца строки в windows, UNIX и MacOS различны. В UNIX это один байт, который обозначается \backslashn. В windows два символа с кодами \backslashr\backslashn. В MacOS те же два символа, но в обратном порядке. Всё это приводит к тому, что текстовые документы необходимо перекодировать, чтобы их можно было использовать в другой системе, даже если используются только коды ASCII. Перекодировку текста выполняет уже упоминавшаяся ранее команда iconv.

rm: удаление файлов

Теперь обсудим самую коварную команду UNIX: rm. Она удаляет файлы, имена которых переданы ей в качестве параметров. У неё есть несколько полезных ключей: "-r" позволяет удалять каталоги и их содержимое. "-i" запрашивает перед удалением \bf каждого файла подтверждение. "-f", наоборот, удаляет все без подтверждения.

Её коварство заключается в том, что при её использовании следует очень опасаться опечаток. Например, желая удалить все файлы в текущем каталоге, Вы набираете команду

```
$ rm -rf /*
но случайно опечатайтесь и введете
```

```
$ rm -rf /*
или
```

```
$ rm -rf . /*
```

По окончании выполнения данная команда удалит \bf все файлы в системе, права на удаление которых Вы имеете. Ключ -i помогает в случае, когда необходимо удалить немного файлов, но в остальных случаях неудобен. Поэтому общая рекомендация: тщательно проверять все параметры команды rm и, если используется неочевидная подстановка, сначала ввести эту команду заменив rm на echo, чтобы посмотреть весь список подставленных параметров.

Каталоги: создание, использование и удаление

Для создания каталога используется команда mkdir. Её параметрами являются имена создаваемых каталогов. В случае, если каталог уже существует, mkdir возвращает сообщение об ошибке. Также возвращается ошибка при попытке создать подкаталог при отсутствии родительского каталога. Например

```
$ mkdir dir1/dir2
```

```
mkdir: cannot create directory `dir1/dir2': No such file or directory
```

Если это поведение нежелательно, можно использовать ключ -p. Тогда mkdir автоматически создает все необходимые родительские каталоги и считает успехом существование каталога по окончании работы (то есть, если он уже есть, то это тоже успех).

Удалить пустой каталог можно командой rmdir.

Чтобы перейти в другой каталог, используется команда cd. Она получает один параметр: каталог, куда переходить. При запуске без параметров происходит переход в домашний каталог (он же каталог \$HOME, он же каталог ~). Можно так же перейти в предыдущий каталог командой "cd -" (она также выводит полное имя этого каталога). Команда pwd выводит полное имя текущего каталога. Теперь при помощи cd, pwd и ls Вы можете обозревать структуру файловой системы.

cp: скопировать файл

Если необходимо скопировать файл, то для этого существует команда cp. Её ключи такие же, как и у rm (-i, -f, -r) и ещё несколько. Дело в том, что копировать файлы можно по разному. По умолчанию

```
$ cp file1 file2
делает то же самое, что и
```

```
$ cat < file1 > file2
```

Такое поведение не всегда желательно: например, при копировании каталога /dev желательно, чтобы создавались такие же файлы устройств, а не читалось содержимое этих файлов. Для этого существует опция -R. Она делает то же, что и -r, но кроме того правильно копирует специальные файлы. Если необходимо сохранить права доступа используется ключ -p. Ещё одно тонкое место: как копировать символические ссылки. Можно либо копировать ссылку как ссылку, либо копировать содержимое файла доступного по ссылке. По умолчанию копируется содержимое. Если нужно скопировать саму ссылку, используйте ключ -P.

Также следует знать, что в UNIX команда `cp` копирует жесткие ссылки в отдельные файлы. Для сохранения жестких ссылок в Linux версии `cp` есть параметр `-d`, который делает то же, что и `-P`, но кроме этого сохраняет жесткие ссылки. Разумеется только в том случае, если все жесткие ссылки копируются одной командой.

Пример

```
$ touch t
$ ln t t2
$ ls -l
```

```
total 0
-rw-r--r-- 2 monk users 0 Oct 11 15:45 t
-rw-r--r-- 2 monk users 0 Oct 11 15:45 t2
```

Видите, что в количестве ссылок указано число 2.

```
$ mkdir a
$ cp -d t t2 a
$ ls -l a
```

```
total 0
-rw-r--r- 2 monk users 0 Oct 11 17:37 t
-rw-r--r- 2 monk users 0 Oct 11 17:37 t2
```

Все нормально скопировалось. Теперь очистим каталог `a` и скопируем файлы поодиночке

```
$ rm a/*
$ cp -d t a
$ cp -d t2 a
$ ls -l a
```

```
total 0
-rw-r--r- 1 monk users 0 Oct 11 17:38 t
-rw-r--r- 1 monk users 0 Oct 11 17:38 t2
```

Теперь количество ссылок равно 1, так как `cp` не знает, что уже скопированный `t` и копируемый `t2` должны быть жесткими ссылками.

mv: переместить файл

Разумеется переместить файл можно и последовательными командами `"cp file1 file2; rm file1"`. Но при этом возникают все вышеперечисленные проблемы, причем проблема жестких ссылок не решается принципиально: если есть жесткие ссылки на `file1`, то на `file2` уже жестких ссылок не будет. Поэтому существует отдельная команда `mv`. Её формат аналогичен команде `cp`, но ключ `-t` и его производные отсутствуют, так как `mv` всегда перемещает каталоги со всем содержимым и ссылки переименовывает в ссылки. Более того, если файл-источник и файл-приемник находятся на одном и том же физическом устройстве, то команда `mv` заменяет только имя файла не меняя все остальные данные. В частности, это означает что все времена доступа не изменятся и жесткие ссылки останутся работоспособными.

mount: подключение дополнительных дисков

Изначально первая версия UNIX грузилась с двух лент. Так как каждая лента с точки зрения системы является отдельным устройством, то пришлось реализовать метод подключения второй ленты так, чтобы в результате получилось единое дерево. (то что корень должен быть один уже было решено). Тогда в системные вызовы добавили вызов `mount`, который позволял к любому существующему каталогу подключить произвольную файловую систему. Второй файловой системой тогда являлся `/usr` (а третьей `/usr/local`) именно отсюда копирование структуры корневой системы.

Реализуется эта функциональность с точки зрения пользователя командой `mount`. Если её вызвать без параметров, то увидите список существующих разделов и к каким каталогам эти устройства подключены. Как правило, настраивается подключение файловых систем в файле `/etc/fstab`, затем просто все подключаются

```
$ mount -a
```

Исключения составляют съемные носители и сетевые диски. Съемные носители прописываются в `/etc/fstab`, но, собственно, подключение производится пользователем, так как съемные носители могут меняться в работающей системе. Рассмотрим работу с дискетой. Сначала производится команда

```
$ mount /mnt/floppy
```

(если эта команда выдает ошибку, значит нет соответствующей строки в `/etc/fstab`)

После этой команды все содержимое дискеты оказывается подключено к каталогу `/mnt/floppy`. По окончании работы с дискетой перед тем как её вытащить \bf обязательно необходимо выполнить команду

```
$ umount /mnt/floppy
```


чтобы отключить дискету от файловой системы и были завершены все отложенные операции записи, иначе можете испортить данные на дискете. Аналогично производится работа с CD-дисками, за исключением того, что пока диск не отключен CD-дисковод блокируется и вытащить его нельзя. Если нет желания прописывать устройство в `/etc/fstab`, то можно всё равно его подключить (необходимо выполнять от пользователя `root`):

```
# mount /dev/fd0 /mnt/floppy
```

То что подключил `root`, может отключить только он.

Без параметров `mount` выводит список подключенных файловых систем.

Место на диске

Разумеется при работе с данными первый вопрос, который приходит в голову: хватит ли места? И если ответ, на первый вопрос "нет", то необходимо как-то узнать, куда это место ушло.

На первый вопрос нам ответит команда `df` (от `disk free`). Без параметров она выводит данные о том, какие устройства есть в системе, куда подключены, сколько на них места всего и сколько используется:

```
$ df
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/hda3	19466372	16098680	2378844	88%	/
/dev/hda5	7801752	7309112	492640	94%	/mnt/win
/dev/hda1	18104872	15947252	2157620	89%	/mnt/ntfs
none	257732	0	257732	0%	/dev/shm

Как правило нас интересует не всё свободное место, а доступное нам. С учетом того, что могут использоваться символические ссылки только из имени каталога и таблицы подключений устройств нельзя выяснить на каком устройстве находится текущий каталог. Но это можно узнать при помощи той же самой команды `df`, если ей передать имя каталога в качестве параметра

```
$ df .
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/hda3	19466372	16098680	2378844	88%	/

Теперь предположим, что места нам не хватает. Используемое каждым каталогом место покажет команда `du` (от `disk usage`). Без параметров она показывает объем каждого подкаталога рекурсивно в текущем каталоге, то есть объем все подкаталогов и их подкаталогов и т.д. Таким образом, после первого же запуска можно прекратить распределение места. Если это поведение нежелательно, то ключ `-s` позволяет показывать только объем тех каталогов, которые переданы в качестве параметров (или текущего, если параметров нет). Так же есть ключ `-h`, который заставляет эту команду выдавать результаты в килобайтах, мегабайтах и т.д., а не всегда в килобайтах. Как правило, используется в форме

```
$ du -sh *
```

что значит "показать объем всех файлов и каталогов в текущем".

Полезные возможности

В UNIX в отличие от операционных систем Microsoft файловые системы не подвержены фрагментации, кроме того, благодаря разделению понятий файл и Inode возможно обновление на ходу любых программ, установленных в системе. Дело в том, что в UNIX после операции "открытие файла" процесс работает именно с соответствующим блоком Inode. Если файл удален или переписан поверх новой версией, то данные удаленного файла не стираются с диска до тех пор пока этот файл не будет закрыт всеми процессами, которые его используют. Например, если будут обновлены системные библиотеки, то все программы запущенные после обновления будут использовать новые библиотеки, но и старые процессы будут работать, так как библиотеки будут храниться на диске (хоть и без имени) до тех пор, пока хоть один процесс эту библиотеку использует (и как следствие, держит её открытой).

Ещё один вариант использования этой возможности - создание временных файлов без имени. Программа создает файл, открывает его (всё одним вызовом `open`), затем удаляет созданный файл и этот файл будет окончательно удален с диска после окончания программы (даже аварийного). Это позволяет с одной стороны, не мусорить временными файлами (бывает, что процесс создает временный файл, а потом забывает его стереть), с другой стороны содержимое такого файла гораздо сложнее прочесть другим процессам (что повышает безопасность).

Благодаря наличию символических ссылок можно переносить каталоги на другие физические устройства так, чтобы данные реально находились на другом устройстве, но с точки зрения системы не перемещались (например так можно переместить каталог `/usr/share/doc` на сетевой диск). Фактически эту же возможность предлагает команда `mount`, но символическими ссылками можно всё настроить гораздо тоньше. Также при

этом появляется возможность реально держать файлы относящиеся к разным пакетам в различных папках, а в /bin, /lib и т.д. сложить только символические ссылки на нужные файлы.

Поиск файлов

find: искать по критериям

В предыдущем разделе было показано как можно что-либо сделать с группой файлов со сходными именами (операция подстановки по маске). Но ведь иногда требуется найти, например, все файлы, которые не читались в течение месяца. Для такого рода задач используется команда `find`. В простейшем варианте (без параметров) она выдает список всех файлов, каталогов и их подкаталогов и т.д. от текущего каталога. Первый параметр у `find` (если есть) - это каталог от которого искать. Список выводится так же как и подстановка, то есть, если

```
$ echo ./*
$ find .
```

также подставляет путь указанный в параметре ("`find .`" и "`find`" выводят одно и то же). Остальные параметры команды `find` используются через ключи: ключ `-name` позволяет сделать поиск по имени. Например, если хотите найти на диске

```
$ find / -name '*find*'
```

Одинарные кавычки используются для того, чтобы shell не интерпретировал `*find*` как подстановку в текущем каталоге. Если ввели эту команду и не хотите ждать её окончания (полный поиск по диску может быть достаточно долгим), можете нажать `Ctrl+C` (далее `^C`).

Если нужен поиск вне зависимости от регистра строки, то вместо `-name` надо использовать `-iname`. Кроме того есть множество дополнительных параметров поиска: `-type` - поиск по типу файла (`f` - обычный файл, `d` - каталог и т.д.), `-user` - поиск по владельцу, `-atime` - поиск по последнему времени доступа (параметр число, `5` - доступ `5` дней назад, `+5` - больше `5`, `-5` - меньше `5`) и некоторые другие возможности (полный список смотрите в `man find`). При этом различные условия можно писать либо подряд (тогда они должны выполняться все), либо ставя между ними ключ `-o` (он работает как слово ИЛИ). Также перед условием можно написать `-not`, тогда смысл условия меняется на противоположный. Затем после ключей условий могут идти ключи действий. По-умолчанию действие - вывод имени найденного файла, но можно его заменить. Например для удаления всех найденных файлов, можно ввести

```
$ find / -name '*.bak' -exec rm {} \;
```

`\;` используется как признак окончания команды, а заменяется на найденное имя файла. Если вместо `-exec` использовать `-ok`, то на каждый файл будет запрашиваться подтверждение выполнения команды.

xargs: оптимизируем обработку большого числа файлов

Возьмём пример из предыдущего раздела: удалить все найденные файлы. Очевидно, что команда `rm` будет вызываться на каждый файл отдельно, несмотря на то, что она может работать не с одним файлом, а с несколькими. Разумеется можно сделать что-то вроде

```
$ rm `find / -name '*.bak'`
```

Тогда весь вывод команды `find` будет вставлен в аргументы `rm`. Но это не всегда работает, так как общая длина командной строки ограничена. Поэтому идеологически правильным вариантом будет

```
$ find / -name '*.bak' | xargs rm
```

Это будет работать с любым количеством файлов, но помните, что мы говорили про специальные символы в именах файлов? Так как `find` не пишет обратный слэш (`\`) перед специальными символами, то `xargs` будет их подставлять как есть, что может привести к неверному результату. Для файлов со специальными символами команда должна выглядеть как

```
$ find / -name '*.bak' -print0 | xargs -0 rm
```

`-print0` - действие, вывод имени файла с нулевым символом в конце, а параметр `-0` у `xargs` позволяет ему воспринимать строки заканчивающиеся нулевым символом и трактовать все остальные символы буквально.

locate: очень быстрый поиск

Чаще всего делается поиск по имени, поэтому была создана специальная система, которая хранит все имена на диске в файле и ищет затем по этому файлу, что гораздо быстрее.

Выглядит это так:

```
$ locate myfile
```

С точки зрения результата это эквивалентно

```
$ find / | grep myfile
```

которая делает образ файловой системы в файле называется updatedb. Как правило она выполняется автоматически раз в сутки, но если, например, надо сделать несколько поисков, а содержимое диска сильно изменилось, то можно её запустить вручную (пользователем root).

Поиск помощи

Я уже рассказывал про команду `man`, которая может дать краткое описание любой команды. Но что делать, если точное имя команды неизвестно? Для этого существует команда `apropos`. Например, Вы хотите найти команду `tm`, но не можете вспомнить её имя, хотя знаете, что оно должно уметь удалять файлы. Вводите

```
$ apropos "remove files"
```

or directories Если искомая фраза находится в нескольких заголовках файлов помощи, они будут выведены все. Если же надо искать не по заголовкам, а по содержимому файлов помощи, то используйте команду `man` с ключом `-K`

```
$ man -K "строка поиска"
```

Обработка текста

Все данные в UNIX хранятся в файлах, а большинство файлов (в частности все файлы настроек и журналы событий) являются текстовыми. Поэтому здесь существуют достаточно мощные средства обработки текстовых файлов.

grep: искать в файле

Более точно будет сказать, что `grep` позволяет провести фильтрацию строк файла по определенному шаблону. Например, нужно выяснить, какие разделы на `ext3` есть в системе. Самый простой способ это выяснить

```
$ grep ext3 /etc/fstab
```

```
/dev/hda3 / ext3 noatime 0 0
```

В данном случае команда `grep` выводит все строки в файле `/etc/fstab`, содержащие `ext3`. Если имя файла не указывать, то `grep` будет работать со стандартным входным потоком (как `cat`). Например, получим ту же информацию из команды `mount`:

```
$ mount | grep ext3
```

```
/dev/hda3 on / type ext3 (rw,noatime)
```

Также очень удобно использовать `grep` совместно с `find`. Например: найти все файлы, если в имени файла или каталога есть слово `Distrib`

```
$ find | grep Distrib
```

Или можно использовать `grep` для поиска по содержимому:

```
$ find /etc -print0 | xargs -0 grep ext3
```

найдет все файлы в каталоге `/etc`, содержащие слово `ext3` и выведет строки с этим словом. Если использовать ключ `-l` у `grep`, то выведены будут только имена файлов, в которых встречается искомая строка. Таким образом можно использовать `grep` как еще одно условие в команде `find`.

Команда `grep` также может искать по шаблону, но её шаблоны отличаются от шаблонов `shell`. В `grep` символ `"*"` означает повторение предыдущего символа любое число раз (включая 0), `"+"` - также, но от 1 раза. Например `"n+"` эквивалентно `"nn*"`. `"."` означает любой символ. Таким образом то, что в `shell` было `"*"` в `grep` будет `"*"`, а вместо `"?"` будет `"."`. Если необходимо ставить просто точку, то, как всегда, перед ней ставится обратный слэш `"\"`. Кроме того есть ещё два специальных символа `"^"` - начало строки и `"$"` - конец строки. Таким образом маска `shell` `"*.bak"` в `grep` будет выглядеть как `"^*.bak$"`. Шаблоны `grep` называются *регулярными выражениями*.

sed: обработка строк

В самом примитивном случае `grep` тоже обрабатывает строки: либо выводит строку, либо нет. Но часто необходимо выведенную строку привести к удобному виду. В приведенном выше примере с командой `mount`, предположим, что нам нужны только точки монтирования. Разумеется, если просмотр делает человек, то он и так увидит всё что ему надо, но, предположим, что результат нам нужно положить в параметры какого-нибудь скрипта. Если попытаться описать человеческим языком, что нужно сделать, чтоб из строки `"/dev/hda3 on / type ext3 (rw,noatime)"` получить только точку монтирования, то получится что-то вроде:

- удалить всё, что, до символов `"on "`
- удалить всё, что, после символов `" type"`

Соответственно команда выглядеть будет так:

```
$ mount | grep ext3 | sed 's/.*on //' | sed 's/ type.*//'
```

Команда `s//` у `sed` имеет формат `s/шаблон/строка/` и заменяет то, что попадает под шаблон на строку. В приведенном примере она просто удаляет найденное регулярное выражение. Если шаблон встречается в строке несколько раз и надо заменить все экземпляры, то в команде `s//` добавьте букву `g` в конец. Рекомендуется команды `sed` брать в одинарные кавычки (апострофы), чтобы `shell` не пытался интерпретировать символы.

awk: работаем с таблицами

Разумеется речь в этом подразделе пойдет о текстовых таблицах. Например, такую таблицу выдают команды `"ls -l"`, `mount` и многие другие. Можно считать, что строки состоят из нескольких полей, разделенных пробельными символами (последовательностью пробелов и табов). Тогда в строке `"/dev/hda3 on / type ext3 (rw,noatime)"` точку монтирования можно считать третьим полем. И вот окончательное решение: третьим полем. И вот окончательное решение:

```
$ mount | grep ext3 | awk '{print $3}'
```

Общий формат команды для `awk` выглядит как `"условие {действие}"`, причем поля соответствующей строки заносятся в нужные переменные. Условие может быть пустым (как в вышеуказанном примере), может быть регулярным выражением. Абсолютно эквивалентный вариант предыдущей команды:

```
$ mount | awk '/ext3/{print $3}'
```

И условие может быть практически любым условием с переменной поля. Например в нашем примере можно заметить, что `ext3` должно быть именно в 5-ом поле (иначе будут показываться, например, также точки монтирования со с названиями типа `text3`, так как `ext3` ищется во всей строке). Так вот, чтобы получить именно то, что надо, выполните команду

```
$ mount | awk 'NF==5{print $3}'
```

`awk` также может работать с полями, разделёнными не пробельными символами, а любыми другими. Для этого у него есть параметр `-F`. Например, чтобы вывести все имена (первое поле) из `/etc/passwd` (поля разделены двоеточиями), нужно выполнить команду

```
$ awk -F ':' '{print $1}' /etc/passwd
```

Сортировка строк

Часто бывает нужно не только вывести в нужном формате, но и отсортировать по нужным полям. В той же команде `mount` можно сортировать по точке подключения или по имени устройства. Эту функциональность обеспечивает команда `sort`. В простейшем варианте она сортирует всё, что ей приходит на стандартный вход как строки и выводит на стандартный вывод. Для сортировки по какому либо полю используется параметр `-k` затем пишутся через запятую начальное и конечное поля, по которым нужно сортировать. Этот ключ можно использовать несколько раз. Если надо изменить разделитель поля, то используйте ключ `-t`. Для численной (а не строковой) сортировки используется ключ `-n`. Например, если надо вывести файл `/etc/passwd` (разделитель `:`) отсортированным по номерам пользователей (3-е поле), то нужно ввести команду:

```
$ sort -n -t : -k 3,3 < /etc/passwd
```

Еще полезной команда `sort` оказывается при использовании `du` для выяснения куда уходит больше всего места.

```
$ du -s * | sort -n
```

позволит вывести список файлов и каталогов с размерами отсортированный по размеру. Ключ `-h` у `du` использовать нельзя, иначе не удастся сортировка.

В некоторых случаях кроме сортировки нужно убрать повторяющиеся строки. Для этого используется команда `uniq`. Она так же, как и `sort` берет строки со стандартного ввода, а результат выдает на стандартный вывод. Для удаления дубликатов строки должны быть предварительно отсортированы.

head и tail: быстрый обзор файла

Как уже было замечено, для просмотра файла есть команда `cat`, но она очень неудобная, если нужно просмотреть большой файл, например те же системные журналы, длина которых может достигать нескольких тысяч строк. Если нужно взглянуть только на начало файла (например, там часто бывает информация о том, что в нем находится) есть команда `head`.

```
$ cat /etc/passwd | head
```

```
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
lp:x:4:7:lp:/var/spool/lpd:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/bin/false
news:x:9:13:news:/usr/lib/news:/bin/false
```

По умолчанию `head` выводит первые десять строк, если надо больше или меньше задайте ключ вида `-n количество_строк`. Например, `head -n 5` выведет 5 строк. Обычно `head` используют совместно с `sort`, например, чтобы вывести 10 самых больших каталогов (упражнение).

Аналогично команда `tail` выводит последние строки. Особенно полезна она для просмотра журналов системных событий. Также можно с её помощью смотреть за системными событиями в реальном времени. Например

```
# tail -f /var/log/messages
```

будет показывать новые записи в файле `/var/log/message` по мере их появления. Прервать выполнение этой команды можно как обычно, нажав `^C`.

Кроме просмотра эту команду можно использовать, чтобы вести журнал событий в двух местах, например

```
# nohup tail -f /var/log/messages > /my/copy &
```

позволит всегда иметь копию системного журнала, даже если файл `/var/log/messages` будет испорчен (по вине аппаратуры или злоумышленника). Но учтите, что таким образом можно получить копию только тех данных, которые дописаны в конец, поэтому не следует пытаться таким образом получить копию, например, базы данных.

less: просмотр текстовых файлов

И всё таки, что делать, если необходимо просмотреть весь файл, а не его первые или последние строки и файл очень большой. Для этого в Linux есть программа просмотра текста под названием `less`. Если Вы хоть раз уже использовали команду `man` (см. `man`), то Вы с ней уже сталкивались. Дело в том, что `man` его использует для отображения документации. Для того, чтобы посмотреть произвольный файл, можете просто набрать "`less имя_файла`". Кроме того, можно просматривать сразу результат вывода команды. Например,

```
$ find /etc | less
```

вывода. При просмотре используются обычные клавиши со стрелками, `PgUp`, `PgDown`. Кроме того, если Вы, например, хотите использовать `less` при доступе через `telnet` (он позволяет использовать только алфавитно-цифровые клавиши), то двигаться по тексту можно вводя следующие латинские символы:

- `j` - вниз
- `k` - вверх
- `f` - `PgDown`

- b - PgUp

Помощь по всем доступным символам доступна при нажатии кнопки "h". Также перед командами j и k можно вводить произвольное число и тогда передвижение будет происходить на соответствующее количество строк. Например, чтобы передвинуться на 30 строк вниз надо набрать "30j". Кроме того есть возможность перейти на определенную строку: "30g" или "30G" перейдут на строку 30. Без параметра g переходит на начало, а G на конец файла. Команда "=" показывает, где сейчас находимся (строки и байты). Из полезных команд хотелось бы упомянуть две. Команда "/" позволяет найти строку в файле и перейти на неё. Например, по команде "/find" less переходит на ближайшее слово "find" в открытом файле. Если "/" используется без параметра, то происходит поиск того, что только что ищлось (как правило используется, чтоб найти следующее такое же слово). Вторая полезная команда: F. Она переводит less в режим аналогичный tail -f. Прервать этот режим можно командой ^C. Любую команду less можно передать через параметр "+". Например

```
# less +F /var/log/messages
```

позволит также следить за журналами, но при помощи less.

vi: редактор текста

Благодаря вышеупомянутым командам обработки текста Вы теперь можете делать простое редактирование файлов, например, добавить в конец файла строку или изменить какую-нибудь настройку по её имени. Просмотр текста любого объема достаточно удобно выполняется при помощи less. Но часто требуется провести достаточно сложное редактирование текста. Для этого в UNIX есть редактор vi. В Linux используется его вариация под названием vim (Vi IMproved). Запуск производится очевидным образом

```
$ vi file.txt
```

или

```
$ vim file.txt
```

Основным принципом vi является наличие двух режимов работы: командный режим и режим ввода текста. В режиме ввода любая нажатая клавиша непосредственно вводится в текст, а в командном режиме они воспринимаются как команды. Если Вы находитесь в режиме ввода, то перейти в командный режим можно нажав кнопку Esc. Введенный символ как правило можно отменить, нажав Backspace. Кроме этого в vim (но не в vi) в режиме редактирования работают клавиши перемещения по тексту и кнопка Del. В командном режиме, очевидно, самая важная команда: перейти в режим ввода. Для этого нажмите или "i" (от слова insert) или "a" (от слова append). Соответственно ввод начнется или перед символом, на котором был курсор или после этого символа.

Стрелки могут не работать даже в командном режиме (например, при входе по telnet). Тогда перемещаться можно почти как в less:

- j - вниз
- k - вверх
- h - влево
- l - вправо

Клавиши выбраны так, чтобы находиться под рукой при десятипальцевом методе печати.

Перед любой командой так же как и в less можно вводить число, тогда команда будет выполнена нужное число раз. Например введя "80i*<Esc>" Вы получите строку из 80 звездочек. Отменить последнее действие можно командой u (в vim можно отменять сколько угодно раз). Вновь вернуть командой ^R. Сохранить и выйти можно нажав дважды Z (обязательно большую).

Нажав ":" можно перейти в режим многосимвольных команд. Например в этом режиме можно вводить команды вида s/aa/bb как в sed. По-умолчанию эта команда действует только на текущую строку. Если надо применить замену к нескольким строкам, то надо вводить <n>,<m>s/aa/bb где <n> - номер первой строки, где нужно провести замену, а <m> - номер последней строки. Последняя строка файла обозначается символом \$, текущая строка - ".", также можно задавать относительные номера строк. Например, +5 значит на 5 строк

вперед от текущей строки, аналогично -5 - 5 строк назад. Команда :help позволяет получить помощь по vim (очень подробную, включающую в себя учебник). :q! позволяет выйти без сохранения. Также есть клавиши быстрого перемещения по тексту:

- \ - начало строки
- \$ - конец строки
- w - слово вперед
- b - слово назад
- % - перейти на парную скобку (очень полезно при редактировании программ)

Напоследок хотелось бы заметить, что несмотря на сложность в изучении работа с vim очень удобна, потому что человек мыслит объектами, а vi/vim позволяет редактировать в тех командах, которыми человек думает: "удалить следующие два слова, перейти на 5 строк вниз, заменить Иван, на Петр", кроме того, команды расположены удобно при десятипальцевом методе печати. Также в vim есть подсветка синтаксиса, автоматическая расстановка отступов в программах, переход по функциям в C/C++/Java и многое другое. Есть лишь один редактор, который превосходит его по гибкости. Он называется emacs. Разумеется в Linux он также доступен, но не всегда есть в минимальной установке (в отличие от vim).

Управление задачами

UNIX является многозадачной системой. Это позволяет выполнять команды и при этом одновременно продолжать работать или напротив, дать команду на сервере посредством дистанционного доступа, затем отключиться и вернувшись увидеть результат выполнения команды.

Для выполнения команды в фоновом режиме после неё надо написать знак &. Например, если надо найти все файлы *.html на диске, то процесс это долгий и, чтобы команда работала и не мешала можно её выполнить так:

```
$ find / -name '*.html' > result 2>&1 &
```

Перенаправление вывода делать практически обязательно, если Вы не хотите, чтобы программа писала результат на экран, мешая работать. На экран вывелся номер задачи в квадратных скобках и номер процесса. После этого, если ввести команду

```
$ jobs
```

то получим список запущенных работ. Если работа завершена, то вместо Running будет Done, а при повторном запуске команды jobs её в списке уже не будет.

Если надо завершить фоновую задачу, то можно воспользоваться командой kill

```
$ kill %1
```

После знака % должен быть номер задачи, какой он был в команде jobs. После остановки задачи будет выведено сообщение (обычно достаточно нажать ещё раз Enter)

```
$
```

Для того, чтобы процесс не прерывался после выхода из терминала начинайте его с команды nohup

```
$ nohup find / -name '*.html' > result 2>&1 &
```

В этом случае Вы сможете вновь войти в систему позже и посмотреть результаты.

Если Вы уже запустили длительную команду и хотите её приостановить, нажмите ^Z. Тогда команда приостановится и Вы получите доступ к командной строки. Для продолжения выполнения команды в обычном режиме наберите

```
$ fg %1
```

Или можете перевести её в фоновый режим:

```
$ bg %1
```


Если надо посмотреть процессы не принадлежащие к текущему терминалу (например, созданные указанным выше образом), то можно воспользоваться командой `ps`. Без параметров она показывает только процессы принадлежащие текущему терминалу

```
$ ps
```

```
PID TTY      TIME CMD
9329 pts/0    00:00:00 bash
9964 pts/0    00:00:00 find
9965 pts/0    00:00:00 ps
```

В первой колонке номер процесса, его так называемый PID, в последней - имя команды, которая запущена в этом процессе. Если нужно полное имя команды, то нужно добавить несколько параметров

```
$ ps -o pid,args
```

```
PID COMMAND
9329 -bash
10030 find / -name *.html
10069 ps -o pid,args
```

Ключ `-o` позволяет изменять формат вывода. Для вывода всех процессов, добавьте ключ `-e`, а если хотите получить только свои процессы выполните команду

```
$ ps -u $USER
```

Ключ `-u` позволяет фильтровать вывод по пользователю, а переменная `USER` хранит имя текущего пользователя.

Для того, чтобы остановить процесс, полученный таким образом, пользуйтесь той же командой `kill`, но в качестве параметра подставляйте номер процесса (без символа %).

```
$ kill 10030
```

Если процесс завис, то он может не отвечать на сигнал `kill`. Тогда можно сделать принудительное прерывание процесса той же командой `kill`, но с параметром `-KILL`.

```
$ kill -KILL 10030
```

Но запомните, что команда `kill` с ключом `-KILL` не позволяет приложению завершиться корректно и может привести к потере данных. Если программа не завершилась после такой команды, значит зависание произошло в одной из функций ядра. Тут уже ничем не поможешь, разве что можно отправить сообщение об ошибке (bugreport) разработчикам ядра.

Ещё раз про подход UNIX

Как Вы, очевидно, уже заметили, в UNIX для решения любой задачи есть несколько путей. Например для обработки файлов можно использовать как параметр `-xexs`, так и `xags`. Функциональность `awk` и `grep` также перекрывается (`awk` шире, но медленнее и сложнее). Общий принцип этого подхода: у каждой задачи должен быть свой наиболее подходящий инструмент. Разнообразие этих инструментов в Linux ещё больше: `perl`, `python`, `tcl`, не считая огромного количества специализированных утилит. Рекомендую постараться если не изучить их все, то хотя бы выяснить для каких задач что лучше всего подходит и изучать по мере необходимости.

Второй принцип UNIX: команда не должна запрашивать подтверждения и не должна ничего сообщать при успехе, но обязательно должна сообщать об ошибке. В некоторых случаях этот принцип кажется не к месту, например, в командах типа `rm` или форматирования диска. Именно поэтому надо всегда помнить о таком "недружелюбном" поведении системы.

Третий принцип: любая задача разбивается на маленькие кусочки, затем для каждого кусочка пишется максимально общая программа (пример: задача - фильтрация строк и `grep`, умеющий обрабатывать произвольные регулярные выражения). Затем всё это склеивается при помощи `shell` (конвейеры, подстановка аргументов, временные файлы, и т.д.). Побочным эффектом данного подхода является то, что практически все компоненты UNIX являются заменимыми. Например, Linux является по сути UNIX, но с другим ядром (полностью совместимым). Аналогично, `grep` в Linux написан группой программистов GNU, но это не мешает использовать те же скрипты, что и на других UNIX системах или даже поставить GNU `grep` на любой другой UNIX.