ОП.14 **Основы функционирования UNIX - систем**

ЗАНЯТИЕ 14

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Система UNIX, как и любая другая развитая современная операционная система, имеет огромное количество команд.

Они позволяют осуществлять множество операций с объектами системы, такими как пользователи, файлы, устройства, управлять их поведением и настраивать систему по своему усмотрению (в определенных пределах).

Ранее мы уже использовали многие команды UNIX, предполагая, что все они вводятся из так называемой командной строки.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Интуитивно этот процесс понятен: из приглашения командной строки вводится команда и на экране консоли отображается или не отображается результат ее выполнения.

С их помощью можно выполнять большинство действий по настройке, диагностированию и управлению системой.

Команды операционной системы являются "кирпичиками", из которых строятся любые, часто очень сложные и изощренные программы.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Далее мы рассмотрим более подробно, как выполняются команды операционной системы и как на их основе создавать программы, которые чаще всего называются ***командными файлами*** или ***командными сценариями***.

Если провести аналогию с операционной системой MS-DOS, то командные файлы там имеют расширение bat и состоят из команд, интерпретируемых ко мандным процессором command.com определенным образом.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Команды UNIX не выполняются сами по себе, а только в контексте определенной командной оболочки, которую называют ***интерпретатором команд***.

Интерпретатор команд проверяет и анализирует введенные команды и их аргументы, анализирует их синтаксис, корректность введенных ключей и т. д.

После успешного завершения проверки интерпретатор запускает соответствующую программу, т. е. создает в UNIX процесс и передает ему управление.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Командные интерпретаторы имеют общее название `shell`.

Помимо исполнения команд, интерпретатор `shell` выполняет и другие операции:

- конвейеризацию команд;
- переназначение ввода/вывода;
- генерацию имен файлов;
- контроль среды окружения.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Значение интерпретатора для пользователей UNIX трудно переоценить.

Командный интерпретатор позволяет выполнять не только отдельные команды, но и решать намного более сложные задачи, которые реализуются на основе командных файлов.

В дальнейшем выражения "командный файл", "командный скрипт" и "командный сценарий" будут использоваться как синонимы и обозначать исполняемый файл, состоящий из команд в виде текстовых строк.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

В настоящее время наиболее часто используются четыре основные разновидности **shell**:

- **Bourne shell** (sh) — является оригинальным командным интерпретатором и включается во все без исключения дистрибутивы операционной системы UNIX;
- **C shell** (csh) — разработан в Калифорнийском университете (г. Беркли). Особенностью этого интерпретатора является возможность интерактивной обработки **shell**-окружения;
- **Korn shell** (ksh) — разработан Дэвидом Корном и включает целый ряд дополнительных возможностей по сравнению с **Bourne shell**;

Взаимодействие пользователя с операционной системой: командные интерпретаторы

- **Bourne Again shell** (bash) — разработан Фондом свободно распространяемых программных продуктов (Free Software Foundation) аккумулирует в себе возможности оболочек C shell и Korn shell, что обусловило его широкую популярность среди системных администраторов.

Он является наиболее продвинутым интерпретатором по сравнению с остальными и служит очень мощным инструментом программирования задач системного администрирования.

В этом разделе мы будем подробно рассматривать именно возможности bash, хотя большинство командных файлов без каких-либо изменений будут работать и в других оболочках.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Наряду с термином "интерпретатор" мы будем использовать другое определение — "командная оболочка".

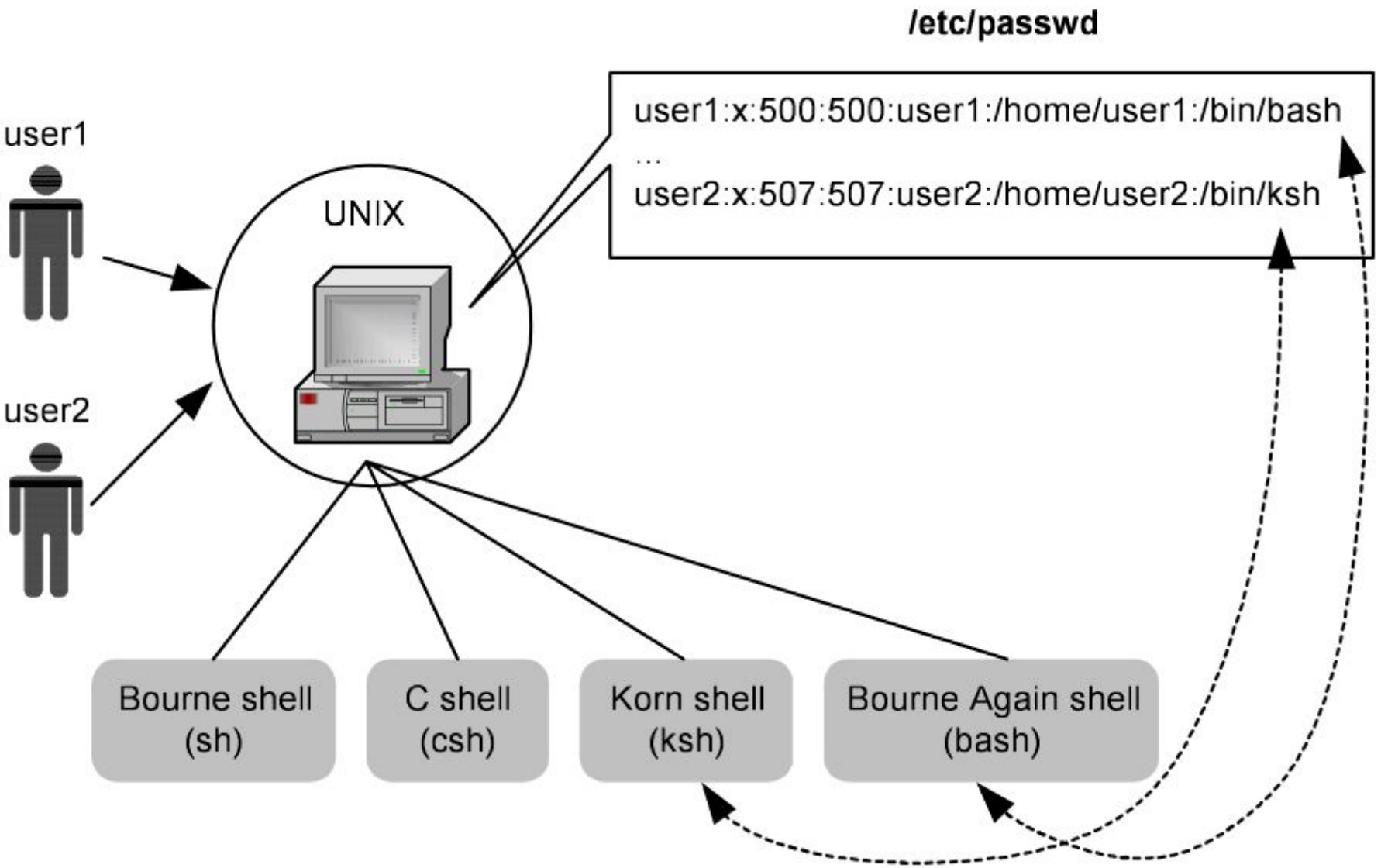
Оба эти выражения являются синонимами и употребляются в одинаковом контексте.

Одна и та же операционная система позволяет работать сразу с несколькими командными оболочками и переходить от одной командной оболочки к другой.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Возможна и одновременная работа пользователя в нескольких экранах со своими командными оболочками.

Любой пользователь, работающий в операционной системе UNIX, оперирует командами того интерпретатора, который для него установлен либо в файлах инициализации, или по умолчанию в файле `/etc/passwd`.


/etc/passwd
user1:x:500:500:user1:/home/user1:/bin/bash
...
user2:x:507:507:user2:/home/user2:/bin/ksh
user1
user2
UNIX
Bourne shell
(sh)
C shell
(csh)
Korn shell
(ksh)
Bourne Again shell
(bash)

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Как видно из рисунка, для пользователя `user1` при входе в систему по умолчанию будет определен интерпретатор `bash`, в то время как для пользователя `user2` интерпретатором по умолчанию будет `ksh`.

Если нужно изменить тип командного интерпретатора, то следует модифицировать соответствующим образом файлы инициализации пользователей `user1` и `user2`.

В практическом аспекте командный интерпретатор `shell` является языком программирования очень высокого уровня.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Работу с командным интерпретатором можно начинать после входа пользователя в систему, при этом оболочка отображает приглашение (**prompt**) к вводу команды в виде одного из символов:

```
$,

#,

>
```

или др., за которым обычно следует пробел.

Команды пользователя вводятся сразу же после этого пробела.

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Анализ особенностей командного интерпретатора проведем на примерах, разработанных в командном интерпретаторе `bash`.

Исходные тексты командных файлов, представленных здесь, легко адаптируются для работы в любой из версий UNIX.

Элементы языка shell

Элементы языка shell

Как и любой язык высокого уровня, командный интерпретатор `shell` имеет определенный синтаксис.

В состав языка входит целый ряд операторов и символов, имеющих специальное значение.

В следующей таблице приводится полный перечень всех специальных символов, используемых в командных интерпретаторах.

Символ	Функция
*	Используется в шаблонах поиска файлов
?	Используется в шаблонах поиска файлов
[ ]	Используется в шаблонах поиска файлов
&	Использование символа амперсанда после вводимой команды позволяет выполнить команду в фоновом режиме, освободив терминал для выполнения других программ
;	Точка с запятой служит разделителем команд, вводимых в одной строке
\	Обратный слэш отменяет назначение специальных символов *, ?, [ ], &, ;, >, < и |
''	Используемый в паре прямой апостроф отменяет значение пробела как разделителя символов и назначение специальных символов
#	Указывает на то, что следующие за ним символы являются обычным текстом (комментарием)

Символ	Функция
` `	Используемый в паре обратный апостроф замещает вывод команды
" "	Кавычки отменяют значение пробела как разделителя символов и назначение специальных символов, кроме $ и `
>	Переназначает вывод команды в файл, при этом предыдущее содержимое файла уничтожается
<	Символ переназначения ввода позволяет команде читать ввод с файла
>>	Позволяет добавить вывод команды в конец файла
|	Символ программного канала переназначает вывод одной команды на вход другой в конвейере команд
$	Символ доллара используется в позиционных параметрах *(см. далее)* и в переменных, определенных пользователем. Кроме этого, данный символ обозначает приглашение к вводу командного интерпретатора

Элементы языка shell

Рассмотрим использование специальных символов на практических примерах.

Следующая командная строка демонстрирует использование точки с запятой:

```
# who;ls -l
root      :0           Sep 29 09:32
root     pts/0        Sep 29 09:57  (:0.0)
total 34
drwxr-xr-x 8 root root 4096 Oct 11 23:43 tmp
-rw-r—r— 1 root root 20633 Sep 3 12:50 install. log
```

Элементы языка shell

```
-rw-r—r— 1 root root 20633 Sep 3 12:50 install
.log.syslog
-rw-r—r— 1 root root 141 Sep 2 14:13 textfile
```

В этом примере последовательно выполняются две команды — `who` и `ls -l`.

Первая команда отображает на экране пользователей, работающих в системе.

Вторая команда отображает список файлов и каталогов.

Элементы языка shell

Последовательность символов `&&` (двойной амперсанд) читает код завершения первой команды, и если он равен **0** (успешное завершение), то будет выполнена вторая команда.

Например:

```
# test -d ./tmp && echo tmp is a directory

tmp is a directory
```

При этом сообщение выводится только в том случае, если объект файловой системы `/tmp` является каталогом (в данном случае это так).

Элементы языка shell

Если изменить условие и выполнить проверку объекта `tmp` как файла, то команда и результат будут другими:

```
# test -f ./tmp && echo tmp is a directory
```

Никакого вывода на экран консоли в этом случае мы не получим.

Это происходит потому что результатом выполнения команды `test` является число, отличное от нуля.

Так как `/tmp` не является файлом.

Элементы языка shell

Если в командной строке набрать команду `test` с опцией `-e` (выполняет проверку на существование файла):

```
# test -e install.log && echo File install.log exists
File install.log exists
```

Так как было выведено сообщение:

`File install.log exists (файл install.log существует)`

то это означает, что после выполнения проверки оказалось, что файл `install.log` действительно существует в текущем каталоге.

Элементы языка shell

Следующая команда выводит на экран содержимое файла `textfile` только в том случае, если этот файл имеет установленный атрибут для чтения:

```
# test -r textfile && cat textfile
```

Если атрибут для чтения установлен, то на экран выводится содержимое файла `textfile`.

Если атрибут для чтения не установлен, то на экран ничего не выводится.

Элементы языка shell

Результат, противоположный тому, который получается при использовании `&&`, дает оператор `||`.

Если из двух команд, разделенных этим оператором, первая команда завершается с **ненулевым** результатом (т. е. **неудачно**), то выполняется вторая команда.

Успешно выполненная первая команды (с **нулевым** кодом завершения) запретит выполнение второй. Например:

```
# test -f /.tmp || echo tmp is a directory
```

Здесь на экран консоли будет выведена строка

```
tmp is a directory
```

Элементы языка shell

Аналогично, команда

```
# test -x textfile ||cat textfile
```

отображает содержимое файла `textfile`, поскольку для этого файла не установлен атрибут исполнения (смотри предыдущий пример, где для этого файла установлен атрибут чтения).

Следовательно, результат выполнения команды `test -x` будет **ненулевым**.

Далее на экране появится содержимое файла `textfile`.

Элементы языка shell

Комбинация операторов `&&` и `||` позволяет создавать довольно замысловатые алгоритмы обработки, например:

```
# test -e textfile && test -r textfile && cat textfile
```

Эта команда выводит на экран содержимое файла `textfile` (то есть выполняется команда `cat textfile`) только при одновременном выполнении двух условий:

- файл существует (`test -e`);
- файл имеет доступ по чтению (`test -r`).

Элементы языка shell

Альтернативой приведенной может служить команда:

```
# test -e textfile && test -x textfile || cat textfile
```

Поскольку команда `test -e` завершается успешно, то будет выполнена следующая за ней команда `test -x`, которая завершается с **ненулевым** кодом (так как у файла существует атрибут для чтения, а не на выполнение).

Результат с **ненулевым** кодом разрешает выполнение команде `cat textfile`.

Далее на экране появится содержимое файла `textfile`.

Элементы языка shell

Следующая командная строка ничего на экран не выводит:

```
# test -e textfile && test -x textfile && cat textfile
```

В этой цепочке вторая команда `test -x` завершится **неудачно**.

Это не позволяя выполниться команде `cat textfile`.

Создавая такие программные конструкции, следует помнить, что конечный результат зависит от результатов выполнения каждой из команд такой цепочки.

Элементы языка shell

Если команда должна выполняться как фоновый процесс, то в ее конце нужно указать символ амперсанда `&`.

Обычно запуск в фоновом режиме нужен для того, чтобы освободить консоль для запуска других программ, особенно если запускаемая программа выполняется достаточно длительное время.

Выполнение команды в фоновом режиме сопровождается выводом на экран идентификатора (**PID**) процесса, соответствующего выполняемой команде, при этом система, запустив фоновый процесс, вновь выходит на диалог с пользователем.

Допускается запускать в фоновом режиме несколько команд, разделенных точкой с запятой.

Например:

```
# who;ls -l;pwd&
root :0 Jan 17 09:32
root pts/0 Jan 17 09:57 (:0.0)
 [1]+ Done
total 56
-rw-r--r-- 1 root root 1544 Dec 1 12:54 install.log
-rw-r--r-- 1 root root 479 Jan 17 09:40 textfile
 [1] 5211
```

После выполнения указанной цепочки команд в системе будет работать фоновый процесс с идентификатором 5211.

Элементы языка shell

Очень мощным средством командного интерпретатора `shell` является переназначение ввода/вывода, расширяющее возможности стандартного ввода/вывода.

Стандартный ввод обозначается в операционной системе UNIX как `stdin` (standard input) и осуществляет операцию ввода данных с клавиатуры терминала, а стандартный вывод `stdout` (standard output) выполняет вывод данных на экран терминала.

Кроме того, диагностические сообщения и сообщения об ошибках направляются в стандартное устройство ошибок, обозначаемое как `stderr` (standard error).

Элементы языка shell

Операционная система UNIX обладает механизмами, позволяющими перенаправить результаты работы любой команды, предназначенные для стандартного ввода/вывода, на другое устройство или в файл.

Вначале рассмотрим операцию ***переназначения вывода***.

Поскольку любые устройства операционной системы UNIX являются файлами, то принято говорить о переназначении вывода в файл, для чего служит оператор >.

Элементы языка shell

Если, например, нужно записать содержимое текущего каталога в файл с именем `list_dir`, то следует ввести команду

```
# ls -l > list_dir
```

выполняющую два действия:

- создание, если не существует, и открытие файла с именем `list_dir` в текущем каталоге процесса;
- запись выводимых командой `ls` данных в файл `list_dir`, при этом предыдущее содержимое файла (если он был непустой) затирается.

После выполнения команды файл `list_dir` будет содержать список файлов и каталогов, находящихся в текущем каталоге.

Элементы языка shell

Переназначение вывода используется очень часто.

Вот несколько типичных примеров применения этой операции:

- запись текстовых строк в файл;
- операции архивирования данных.

Для записи текста в файл используется команда `echo`, например:

```
# echo String to be written in file > textfile
```

Элементы языка shell

Переназначение вывода используется очень часто.

Вот несколько типичных примеров применения этой операции:

- запись текстовых строк в файл;
- операции архивирования данных.

Для записи текста в файл используется команда `echo`, например:

```
# echo String to be written in file > textfile
```

При использовании `>`, файл будет перезаписан.

Элементы языка shell

Пример создания архива:

```
# find tmp -print I cpio -ovB > tmp_arch
```

Здесь команда `find` передает по программному каналу данные программе `cpio`, которая записывает их в архивный файл `tmp_arch`, используя переназначение вывода.

Еще один пример переназначения стандартного вывода:

```
# cpio -it -I tmp_arch > list_arch
```

Здесь команда `cpio` записывает список файлов, содержащихся в архиве `tmp_arch`, в файл `list_arch`.

Элементы языка shell

К операторам переназначения вывода относится и >>, как и оператор >, он используется для добавления данных в уже существующий файл.

При этом информация записывается в конец файла, не затирая предыдущую информацию.

С помощью этого оператора можно добавить в файл textfile, созданный в одном из предыдущих примеров командой echo, следующую строку:

```
# echo String to be added to the  file >> textfile
```

Оператор переназначения >> можно использовать и как замену >.

Элементы языка shell

К операторам переназначения вывода относится и >>, как и оператор >, он используется для добавления данных в уже существующий файл.

При этом информация записывается в конец файла, не затирая предыдущую информацию.

С помощью этого оператора можно добавить в файл textfile, созданный в одном из предыдущих примеров командой echo, следующую строку:

```
# echo String to be added to the  file >> textfile
```

Оператор переназначения >> можно использовать и как замену >.

Элементы языка shell

Вывод на стандартное устройство ошибок также можно переназначить, но при этом следует использовать выражение `2>`.

Рассмотрим пример. Попытаемся выполнить команду `ls` с несуществующей опцией `-y`, перенаправив при этом вывод в файл `ls.LOG`:

```
# Is -y 2> ls.LOG
```

После выполнения команды файл `ls.LOG` может содержать примерно следующее:

```
# cat ls.LOG
Is: invalid option -- y
Try `ls --help` for more information.
```

Элементы языка shell

Нужно всегда помнить, что если файл, в который переназначается вывод, уже существует, то его предыдущее содержимое теряется.

Если нужно сохранить содержимое файла, лучше ис пользовать оператор >>.

Программы, использующие для получения данных стандартный ввод, могут принимать их из файла через оператор переназначения ввода <.

Например, для подсчета количества строк, слов и символов в файле `textfile` можно применить команду:

```
# wc < textfile
```

Элементы языка shell

Операция переназначения ввода используется и в более сложных операциях, например, разархивирования данных:

```
# cpio -ivB < tmp_arch
```

Здесь команда `cpio` получает ввод из файла архива, созданного в одном из предыдущих примеров командой `find`, восстанавливая полный путь к файлу в процессе разархивирования.

Элементы языка shell

Ввод и вывод одной и той же команды могут быть переназначены одновременно, как в этом примере:

```
# echo String > output
```

```
# cat < output > input
```

Команда `echo` записывает текстовую строку в файл `output`.

В следующей строке команда `cat` получает данные из файла `output` и записывает их в файл `input`.

После выполнения этих двух команд файл `input` будет содержать строку `String`.

Элементы языка shell

Другими, очень мощными средствами командного интерпретатора `shell` являются фильтры и ***конвейеры команд*** (***программные каналы***, `pipes`).

Фильтрами называются программы или команды, которые выполняют чтение со стандартного ввода и записывают результат в стандартный вывод.

Многие команды операционной системы UNIX реализованы в виде фильтров.

Фильтры допускают объединение, используя механизм конвейеризации, позволяющий объединять стандартный вывод одной команды со стандартным вводом другой.

Элементы языка shell

Оператор, реализующий механизм программных каналов, обозначается вертикальной чертой `|`.

Командную строку, состоящую из нескольких команд (прог рамм), работающих в конвейере, можно представить следующим образом:

программа1 `|` *программа2* `|` *программа3*`...`

Первая команда конвейера создает поток выходных данных, принимаемых на входе *программы2*.

программа2, в свою очередь, создает выходной поток для *программы3* и т. д.

Элементы языка shell

Вернемся к функционированию программных каналов в интерпретаторе `shell`.

Пусть требуется определить количество файлов, содержащихся в текущем каталоге.

Если использовать переназначение ввода/вывода, то данная задача решается посредством двух команд:

```
# ls -l > list_files
# wc -l < list_files
```

Здесь используются операторы `>` и `<` переназначения ввода/вывода, а промежуточный результат хранится в файле `list_files`. Команда `wc -l` считает количество строк в документе.

Элементы языка shell

Также операция с использованием программного канала решается одной командой:

```
#  ls -l | wc -l
```

Нужно четко представлять себе, когда лучше использовать переназначение ввода/вывода, а когда конвейер команд.

Оптимальное сочетание того и другого механизмов позволяет реализовать довольно сложные и эффективные алгоритмы обработки данных.

Так, например, если необходимо сохранить список файлов, имена которых начинаются с "`text`", то это можно сделать одной командой:

```
# ls -l | grep text > text_files
```

Элементы языка shell

Командный интерпретатор `shell`, подобно языкам высокого уровня, использует переменные.

Имя переменной должно начинаться с буквы или с символа подчеркивания, например:

```
string_1

_InputVal
```

Элементы языка shell

Имена переменных наподобие:

`7n`

`my var`

`.var1`

использовать нельзя.

Имя `7n` начинается с цифры,

`my var` содержит пробел,

а `.var1` — недопустимый символ точки.

Элементы языка shell

Переменным можно присвоить определенные значения с помощью оператора присваивания =:

```
string_1="String 1"

Var1=1
```

Здесь переменной `string_1` присвоена строка символов `"String 1"`, а переменной `Var1` — значение 1.

Обратите внимание на то, что строковое значение заключается в кавычки.

Элементы языка shell

В отличие от языков программирования высокого уровня, таких, например, как C («си»), переменные командного интерпретатора `shell` не связаны с определенным типом данных, поэтому любое присвоенное им значение интерпретируется как строка символов.

Для доступа к переменной нужно непосредственно перед ее именем установить знак доллара `$`.

Следующий пример демонстрирует вывод содержимого переменной `myvar` на экран:

```
# myvar=10
# echo  $myvar
10
```

Элементы языка shell

Точно так же на экране отображается и строковое значение, присвоенное переменной:

```
# message="String"
# echo $message
String
```

Везде, где встречается символ `$`, предшествующий имени переменной, он заменяется ее значением, например:

```
# myvar=100
# echo myvar = $myvar
myvar = 100
```

Элементы языка shell

Если имя переменной заключить в фигурные скобки, то отображаемое значение будет содержать символы (если есть), находящиеся за фигурными скобками, например:

```
# myvar="String"
# echo ${myvar}100
String100
```

Важное замечание: при записи операции присваивания переменная, оператор = и присваиваемое значение должны быть записаны без пробелов.

Элементы языка shell

Переменной может быть присвоен результат выполнения команды, для чего используются обратные апострофы, например:

```
# DATE=`date`
```

Здесь переменная `DATE` получает результат вывода команды `date`.

Элементы языка shell

Если команда заключена в обратные апострофы, то интерпретатор обязательно выполнит ее, поместив результат на то место в командной строке, где эта команда указана:

```
# users=`who | wc -l`
# echo $users users logged in.
3 users logged in.
```

Элементы языка shell

Для работы со строками, файлами и каталогами в интерпретаторе `shell` очень часто используются специальные символы (метасимволы):

- `*` — произвольная последовательность символов, может не содержать ни одного символа;
- `?` — один произвольный символ;
- `[... ]` — любой из символов в указанном диапазоне, либо указанный в перечислении.

Элементы языка shell

Например, команда:

```
# ls -l r*
```

отображает все файлы каталога, начинающиеся с литеры `r`.

Команда:

```
# ls -l *rt*
```

отображает на консоли все файлы, содержащие в имени `rt`. Команда:

```
#  ls -l t.???
```

выводит файлы текущего каталога, начинающиеся с литеры `t` и имеющие 3-буквенные расширения, например, `tx.doc` и `tx.out`.

Элементы языка shell

Команда:

```
# ls -l [a-d]*
```

выводит на экран список файлов, начинающихся с `a`, `b`, `c`, `d`.

Аналогичный результат можно получить с помощью одной из команд:

```
# cat [abed]*
# cat [bdac]*
```

Элементы языка shell

Символ * можно использовать и самостоятельно.

Например, команда:

```
# echo *
```

отображает имена всех файлов текущего каталога на экране.

Элементы языка shell

Кроме перечисленных метасимволов, командный интерпретатор `shell` использует еще несколько символов, имеющих специальное назначение:

- двойные кавычки `"`;
- апостроф `‘`;
- символ обратной наклонной черты `\`.

Элементы языка shell

Предположим, что текущий каталог содержит файлы `data1`, `prog1` и `text1`.

Введем две команды `echo`, аргументы которых содержат `*`, и сравним результаты выполнения:

```
# echo *
data1 prog1 text1
# echo "*"
*
```

В первом случае интерпретатор `shell` воспринимает `*` как список содержимого текущего каталога и отображает список файлов, а во втором двойные кавычки отменяют значение символа `*`.

Элементы языка shell

В этом и заключается смысл использования двойных кавычек.

Любой символ, имеющий специальное значение (например, `*`, `?`, `>`, `>>` , `||`), утрачивает свой специальный статус.

Исключениями из этих случаев являются:

- символ `$`,
- обратный апостроф (`` ` ``),
- обратный слэш (`\`),

если они предшествуют специальному символу.

Элементы языка shell

Смысл одиночных апострофов демонстрирует следующая команда:

```
# echo '$text'
$text
```

Как это следует из результата, если переменная заключена в апострофы, то ее значение не подставляется при вычислении выражения.

Элементы языка shell

Разницу между двойными кавычками, апострофами и их отсутствием можно увидеть, выполнив следующие команды:

```
# echo *
data1 prog1 text1

# echo * "*" '*'
data1 prog1 text1 * *

# var1=test
# echo $var1 "$var1" '$var1'
test test $var1
#
```

Элементы языка shell

Обратная наклонная черта перед символом ликвидирует его специальное значение:

```
# echo \$text
$text

# echo \\
\
```

Элементы языка shell

Кратко рассмотрим смысл основных системных переменных: • `PATH` — указывает, в каких каталогах искать выполняемые файлы;

- `HOME` — это путь к домашнему каталогу пользователя;
- `MAIL` — имя файла электронной почты пользователя;
- `SHELL` — указывает оболочку, в которой работает пользователь.

Элементы языка shell

Эти переменные можно использовать в выражениях и командах интерпретатора `shell` обычным образом, так же, как и другие переменные.

Большинство командных интерпретаторов в общем случае не предназначены для выполнения сложных математических вычислений, хотя имеют встроенные средства для арифметических операций.

Приятным исключением является командная оболочка `bash`, в которой весьма существенно расширены возможности по обработке математических величин.

Элементы языка shell

Например, командный интерпретатор выполняет приведенные далее операции следующим образом:

```
# n=0
# n=$n+1
# echo $n
0+1
```

Арифметические операции, которые можно выполнить в `shell`, мы рассмотрим далее при анализе командных файлов.

Командные файлы

Командные файлы

Командный интерпретатор `shell` наиболее эффективен, когда используется для запуска так называемых командных файлов (командных сценариев, скриптов).

Командные файлы представляют собой обычные текстовые файлы, содержащие последовательность команд.

В командных файлах, что очень существенно, можно использовать логические конструкции, подобные тем, что применяются в языках высокого уровня:

- `while`,
- `if`,
- `for`.

Командные файлы

Такие логические структуры позволяют реализовывать довольно сложные алгоритмы обработки данных и управления системой, которые невозможно реализовать из командной строки.

Хотя в командной строке и можно обеспечить некоторую гибкость выполнения за счет использования:

- команд `test`,
- операторов `&&`,
- операторов `||`.

Командные файлы

Вот простой пример командного файла с именем `dl`, созданного с помощью редактора `vi`:

```
pwd
ls
echo This is the end of the shell script
```

Для запуска на выполнение файла `dl` введем строку:

```
# sh dl
```

Командные файлы

Если текущим каталогом является, например, `/home/user1`, то результат может выглядеть так:

```
# sh dl
/home/user1
file1
file2
file3
…
This is the end of the shell script
```

Командные файлы

Текстовый файл можно сделать выполняемым (командным) при помощи команды `chmod`:

```
# chmod +x  имя_файла
```

Здесь *имя_файла* — имя текстового файла, подлежащего выполнению.

Данная команда устанавливает для файла атрибут исполнения (в символьной форме `+x`).

Указывая имя файла, помните, что UNIX различает строчные и прописные литеры.

Командные файлы

При создании командных файлов следует учитывать несколько важных моментов:

• не начинайте командный файл символом `#`, если планируется выполнение этого файла в командном интерпретаторе `csh` (`C shell`);

• ни в коем случае не присваивайте командному файлу имя, совпадающее с именем одной из системных команд, таких, например, как `ls`, `rm` или `mv`, иначе вместо командного файла UNIX выполнит системную команду.

Объясняется это просто — система ищет выполняемую команду в каталогах, определяемых системной переменной PATH, и только потом в каталоге пользователя.

Командные файлы

В командных файлах широко используются так называемые позиционные параметры, задающие аргументы командной строки для выполняемого файла.

Позиционный параметр — это число, перед которым расположен знак доллара, например, `$1`, `$2`, `$3` и т. д.

При запуске командного файла параметр `$1` соответствует первому по порядку аргументу после имени командного файла, параметр `$2` — второму и т. д.

Допускается использовать до девяти позиционных параметров.

Командные файлы

Командный файл (предположим, он называется `test_parms`), содержащий операторы:

```
echo 1-st parameter – $1
echo 2-nd parameter – $2
echo 3-d parameter – $3
```

после запуска с параметрами `1`, `2`, `3` выводит на консоль строки:

```
# ./test_parms 1 2 3
1-st parameter – 1
2-nd parameter – 2
3-d parameter – 3
```

Командные файлы

Позиционные параметры позволяют задавать параметры командной строки в командных файлах, включая применение в конвейере команд.

Приведем некоторые примеры использования позиционных параметров.

Пусть командный файл содержит строку:

```
ls -l | grep $1
```

Предположим, что файл назван `list_file`, тогда его выполнение с параметром `text` может дать, например, такой результат (если файл `text` существует):

```
# ./list_file text
-rw-rw-r— 1 root root 19 Sep 8 12:18 text
```

Командные файлы

Командная строка может содержать до 128 аргументов.

Однако командный файл может обрабатывать одновременно только девять параметров.

Во многих случаях требуется вводить данные с клавиатуры, присваивая их значения переменным.

Это обеспечивает команда `read`, которой часто предшествует команда `echo`.

Командные файлы

Команда `echo`, отображающая на экране приглашение к вводу, например:

```
echo "Enter Integer:"
read x
```

Данный фрагмент командного файла после вывода на экран сообщения:

```
Enter integer:
```

ожидает ввода значения с клавиатуры.

Командные файлы

Одна команда `read` позволяет присвоить значения сразу нескольким переменным, и если параметров команды `read` больше, чем их было введено, то оставшимся переменным присваивается пустая строка.

Если количество введенных значений больше, чем параметров в команде `read`, то лишние значения игнорируются.

Командные файлы

Следующий пример демонстрирует использование команды `read`:

```
echo Enter string:

read x y

echo You entered: $x + $y
```

Один из возможных результатов работы командного файла показан далее:

```
Enter string:

37 51

You entered: 37 + 51
```

Командные файлы

Командный интерпретатор `shell` обрабатывает и два так называемых специальных параметра — `$#` и `$*`.

Параметр `$#` указывает на общее количество параметров выполняемого командного файла.

Если сохранить строку

```
echo The number of arguments is $#
```

в файле `arg_num` и выполнить его, то получим, например, такой результат:

```
# arg_num param1 param2 param3
The number of arguments is 3
```

Командные файлы

Параметр `$*` представляет собой строку, содержащую все аргументы командного файла.

Если сохраним строку:

```
echo The parameters are: $*
```

в файле `arg_content` и запустим его с параметрами `Three arguments here`, то получим такой результат:

```
# arg_content Three arguments here
The parameters for this command are: Three arguments here
```

Командные файлы

Командные файлы, в общем случае, не так часто используют какие-либо математические вычисления.

Тем не менее, для обработки целых чисел (если такое необходимо) можно воспользоваться командой `expr`.

Использование этой команды лучше всего показать на нескольких примерах.

Командные файлы

Первый пример демонстрирует вывод разности двух целых чисел на дисплей:

```
echo Enter two integers

read x y

echo The difference = `expr $x - $y`
```

Обратите внимание на запись выражения справа от команды `expr`.

Чтобы получить правильный результат, необходим пробел между переменными и знаком операции.

Командные файлы

Команда `expr`, кроме сложения и вычитания, позволяет выполнять умножение и деление чисел.

В следующем примере командного файла находится произведение двух переменных, значения которых введены с клавиатуры:

```
echo Enter two numbers:
read x y
echo 1-st num: $x
echo 2-nd num: $y
echo Multiplying $x and $y = `expr $x '*' $y`
```

Командные файлы

Обратите внимание на оператор:

```
`expr $x '*' $y`
```

Здесь для выполнения операции умножения символ * следует взять в апострофы, иначе командный интерпретатор выдаст ошибку.

Операцию деления двух целых чисел можно выполнить, если в рассмотренном выше примере заменить символ * символом /.

Командные файлы

Вот пример более сложных вычислений:

```
echo Enter two numbers:
read x y
echo First number: $x
echo Second number: $y
echo $x '*' $y + $x '/' $y = `expr $x '*' $y + $x '/' $y`
```

Командные файлы

Если сохранить указанные команды в файле с именем `multi_ops` и запустить его на выполнение, то можно получить результат такого вида:

```
# ./multi_ops
Enter two numbers:
5 3
First number: 5
Second number: 3
5*3 + 5/3 = 16
```

Командные файлы

Нередко требуется передавать значения переменных, используемых в одном процессе, другому процессу.

Все переменные какого-либо процесса по умолчанию являются локальными для него и недоступными остальным процессам.

Командный интерпретатор `shell` имеет механизм, позволяющий данные одного процесса передавать другому процессу.

Этот механизм нарывается экспортированием.

Он реализован при помощи команды `export`.

Командные файлы

Команды `export` имеет такой синтаксис:

```
# export список_переменных
```

Здесь *`список_переменных`* представляет собой список переменных, разделенных пробелами, причем символ `$` перед именами переменных не ставится.

Любой процесс, который выполняется после этой команды, получает доступ к экспортированным переменным, но изменить их значений он не может.

Командные файлы

Запишем в командный файл строку:

```
echo   $x
```

и сохраним его под именем `demo_exp`.

Этот командный файл при выполнении ничего не выводит на экран.

Это происходит потому что команда `echo` ничего "не знает" о переменной `x`.

Командные файлы

Присвоим переменной `x` значение 777 и экспортируем ее:

```
# x=777
# export x
```

Повторно запустим командный файл `demo_exp` .

Теперь на экране появится значение **777**.

Командные файлы

Механизм экспортирования обладает одной весьма полезной особенностью.

Если присвоить переменной `x` другое значение, то это значение будет доступно другим процессам без повторно го выполнения команды `export`.

Список литературы:

1. Юрий Магда. UNIX для студентов, Санкт-Петербург «БХВ-Петербург», 2007.
2. Unix и Linux: руководство системного администратора, 4-е издание, 2012, Э. Немет, Г. Снайдер, Т. Хейн, Б. Уэйли
3. Организация UNIX систем и ОС Solaris 9, Торчинский Ф.И., Ильин Е.С., 2-е издание, исправленное, 2016.

Спасибо за внимание!

Преподаватель: Солодухин Андрей Геннадьевич

Электронная почта: asoloduhin@kait20.ru