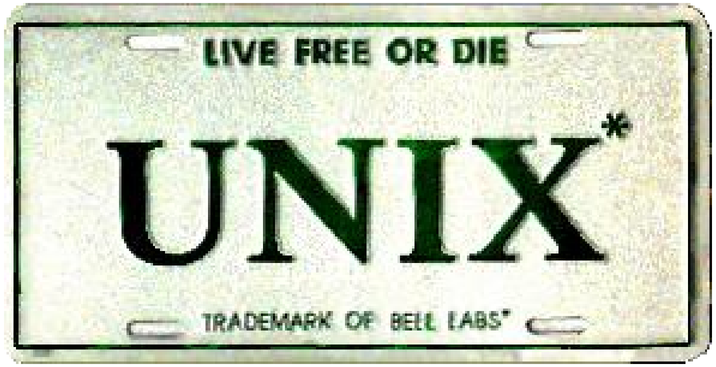


UNIX for Programmers and Users

- **PIPES**

  - Shells allow you to use the standard output of one process as the standard input of another process by connecting the processes together using the pipe | metacharacter.

  - The sequence

    ```
    $ command1 | command2
    ```

    causes the standard output of command1 to “flow through” to the standard input of command2.

  - Any number of commands may be connected by pipes.

  - A list of commands in this way is called a *pipeline*.

  - Based on one of the basic UNIX philosophies: large problems can often be solved by a chain of smaller processes

- Example, pipe the output of the **ls** utility to the input of the **wc** utility in order to count the number of files in the current directory.

```
$ ls                    ---> list the current directory.
a.c    b.c      cc.c    dir1    dir2

$ ls | wc -w
    5

$ ls -l | awk '{ print $1 }' | sort          ---> example
```

- **COMMAND SUBSTITUTION**

A command surrounded by grave accents (`) - back quote - is executed, and its standard output is inserted in the command’s place in the entire command line.

Any new lines in the output are replaced by spaces.

For example:

```
$ echo  the date today is `date`
the date today is Wednesday August 24 11:40:55 2016
$ _
```

- By piping the output of who to the wc utility,
  it’s possible to count the number of users on the system:

```
$ who    ---> look at the output of who.
posey        ttyp0        Jan   22   15:31  ( blackfoot:0.0 )
glass        ttyp3        Feb    3   00:41  ( bridge05.utdalla )
huynh        ttyp5        Jan   10   10:39  ( atlas.utdallas.e )

$ echo  there  are  `who  |  wc -l`   users on the system
there are 3 users on the system
$ _
```

- **SEQUENCES**

If you enter a series of simple commands or pipelines separated by semicolons, the shell will execute them in sequence, from left to right.

This facility is useful for type-ahead(and think-ahead) addicts who like to specify an entire sequence of actions at once.

Here's an example:

```
 $ date;  pwd;   ls     ---> execute three commands in sequence.
Wednesday August 24 11:40:55 2016
 /home/glass/wild
 a.c    b.c    cc.c        dir1        dir2
 $ _
```

- Each command in a sequence may be individually I/O redirected as well:

```
  $ date > date.txt;  ls;   pwd   > pwd.txt
  a.c         b.c         cc.c         date.txt         dir1         dir2

  $ cat  date.txt
 Wednesday August 24 11:40:55 2016

  $ cat   pwd.txt         ---> look at output of pwd.
  /home/glass
  $ _
```

- **Conditional Sequences**

- Every UNIX process terminates with an exit value.
  an exit value of 0 --> process completed successfully
  a nonzero exit value --> failure

- All built-in shell commands return a value of 1 if they fail.

1) Commands are separated by “&&” tokens

2) Commands are separated by “||” tokens

- For example,
  if the C compiler cc compiles a program without fatal errors,
  it creates an executable program called “a.out” and returns an exit code of 0;
  otherwise, it returns a nonzero exit code.

```
$ cc  myprog.c  &&  ./a.out
```

- The following conditional sequence compiles a program called “myprog.c” and displays an error message if the compilation fails:

```
$ cc  myprog.c  ||  echo  compilation  failed.
```

- **GROUPING COMMANDS**

  - Commands may be grouped by placing them between parentheses.

  - The group of commands shares the same standard input, standard output, and standard error channels and may be redirected and piped as if it were a simple command.

  - Here are some examples:

```
$ date; ls; pwd > out.txt                    ---> execute a sequence.
Wednesday August 24 11:40:55 2016             ---> output from date.
a.c          b.c                                     ---> output from ls.
$ cat   out.txt                                    ---> only pwd was redirected.
 /home/glass

$ ( date; ls; pwd ) > out.txt                ---> group and then redirect.
$ cat out.txt                                       ---> all output was redirected.
Wednesday August 24 11:40:55 2016
a.c                b.c
/home/glass
$ _
```

- **Background Processing**

```
$ find   .    -name     a.c      --->search for “a.c”
./wild/a.c
./reverse/tmp/a.c

$ find   .    -name   b.c &     --->search in the background.
27174                          --->process ID number.

$ date                                 -->run “date” in the foreground.
./wild/b.c                             -->output from background “find”.
Wed, Aug 24, 2016  11:59:08 AM -->output from date.

$ ./reverse/tmp/b.c        -->more output from background “find”
                           -->came after we got the shell prompt,
                           --> but we don’t get another prompt.
```

- **Background Processing**

- Several background commands may be specified on a single line by separating each command by an ampersand.

```
$ date & pwd &          ---> create two background processes.
27310                     ---> process ID of “date”.
27311                     ---> process ID of “pwd”.
/home/glass             ---> output from “pwd”.
Wed, Aug 24, 2016  6:59:08 AM     ---> output from “date”.
$ _
```

• REDIRECTING OUTPUT OF BACKGROUND PROCESSES

To prevent the output from a background process from arriving to the terminal, redirect its output to a file.

```
$ find  .   -name  a.c  > find.txt  &
27188                                   ---> process ID of “find”.

$ ls -l  find.txt                      ---> look at “find.txt”.
-rw-r--r--   1   glass     0  Aug   23   18:11   find.txt

$ ls -l  find.txt                      ---> watch it grow.
-rw-r--r--   1   glass    29  Aug   23   18:11   find.txt

$ cat  find.txt                        ---> list “find.txt”.
./wild/a.c
./reverse/tmp/a.c
$ _
```

SUBSHELLS

1) When a grouped command such as (ls; pwd; date) is executed

If the command is not executed in the background, the parent shell sleeps until the child shell terminates.

2) When a script is executed

If the script is not executed in the background, the parent shell sleeps until the child shell terminates.

3) When a background job is executed

The parent shell continues to run concurrently with the child shell.

- A child shell is called a *subshell*.

- *cd* commands executed in a subshell do not affect the working directory of the parent shell:

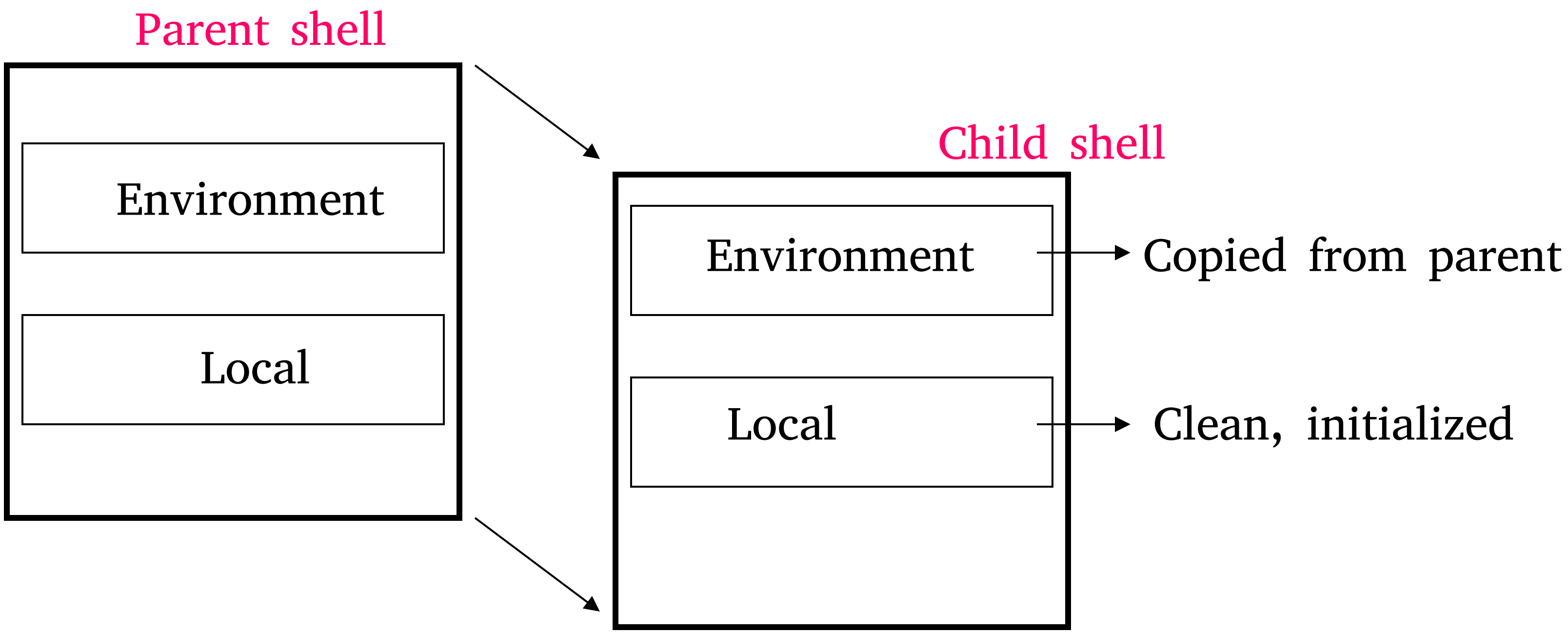
```
$ pwd                  ---> display my login shell's current directory.
/home/glass

$ ( cd /; pwd ) ---> the subshell moves and executes pwd.
/                          ---> output comes from the subshell.

$ pwd                   ---> my login shell never moved.
/home/glass
$ -
```

- Every shell contains two data areas:
  an environment space and a local-variable space.

  A child shell inherits a copy of its parent's environment space and a clean local-variable space:



Environment variables are therefore used for transmitting useful information between parent shells and their children.

- **VARIABLES**

- Here is a list of the predefined environment variables that are common to all shells:

Name	Meaning
$HOME	the full pathname of your home directory
$PATH	a list of directories to search for commands
$USER	your username
$SHELL	the full pathname of your login shell
$TERM	the type of your terminal

- **VARIABLES**

- the syntax for assigning a variable is as follows:

```
variableName=value        ---> place no spaces around the value
or
variableName=“ value ”   ---> here, spacing doesn’t matter.
```

```
$ echo $HOME
/home/SRD

$ HOME=UG1

$ echo $HOME
UG1
$
```

- **VARIABLES**

- The next example illustrates the difference between local and environment variables.

  In the following, we assign values to two local variables and then make one of them an environment variable by using the Bourne shell *export* command.

  Note that the value of the environment variable is copied into the child shell, but the value of the local variable is not.

  Finally, we press *Control-D* to terminate the child shell and restart the parent shell, and then display the original variables:

```
$ firstname="Shiv Ram"                    ---> set a local variable.
$ lastname=Dubey                          ---> set another local variable.
$ echo $firstname  $lastname  ---> display their values.
Shiv Ram Dubey

$ export lastname                         ---> make “lastname” an
                                          ---> environment variable.
$ bash                       ---> start a child shell; the parent sleeps.

$ echo  $firstname   $lastname            ---> display values again.
Dubey                        ---> note that firstname was’t copied.

$ ∧D
$ echo  $firstname  $lastname ---> they remain unchanged.
Shiv Ram Dubey
$ _
```

Warning !

- The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.

```
count=0
count=Sunday
```

- Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it.

- So it is recommended to use a variable for only a single TYPE of data in a script.

- **QUOTING**

- There are often times when you want to inhibit the shell's wildcard-replacement, variable-substitution, and/or command-substitution mechanisms.

  The shell's quoting system allows you to do just that.

- Here's the way that it works:

  1) Single quotes(') inhibit wildcard replacement, variable substitution, and command substitution.

  2) Double quotes(") inhibit wildcard replacement only.

  3) When quotes are nested, it's only the outer quotes that have any effect.

- **QUOTING**

-Examples:

```
$ echo  3 * 4 = 12     ---> remember, * is a wildcard.
   3   a.c    b    b.c    c.c    4 = 12
$ echo  “3 * 4 = 12” ---> double quotes inhibit wildcards.
   3 * 4 = 12
$ echo  ‘3 * 4 = 12’  ---> single quotes inhibit wildcards.
   3 * 4 = 12

$ name=Graham
$ echo  ‘my name is $name - date is `date`’
   my name is $name - date is ‘date’

$ echo  “my name is $name - date is `date`”
   my name is Graham - date is Wed, Aug 24, 2016  7:38:55 AM
$ -
```

- **QUOTING**

-Examples:

```
$ echo  3 * 4 = 12 $USER `date`

$ echo  “3 * 4 = 12 $USER `date`”

$ echo  ‘3 * 4 = 12 $USER `date`’

$ echo  ‘3 * 4 = 12 “$USER” `date`’

$ echo  “3 * 4 = 12 ‘$USER’ `date`”
```

Command Substitution

- The backquote “`” is different from the single quote “´”. It is used for command substitution:

```
$ LIST=`ls`
$ echo $LIST
hello.bash read.bash

$ PS1=“`pwd`---->”
/home/SRD---->
```

- We can also perform the command substitution by means of $(command)

```
$ LIST=$(ls)
$ echo $LIST
hello.bash read.bash
```

- **JOB CONTROL**

- Convenient multitasking is one of UNIX's best features, so it's important to be able to obtain a ***listing of the current processes*** and to ***control their behavior***.

1) **ps**, which generates a list of processes and their attributes, including their names, process ID numbers, controlling terminals and owner.

2) **kill**, which allows to terminate a process based on its ID number.

Utility : ps

ps -e

ps generates a listing of process-status information.

The -e option instructs ps to include all running processes.

Utility : sleep seconds

The sleep utility sleeps for the specified number of seconds and then terminates.

```
$ ( sleep 10; echo done ) &      ---> delayed echo in background.
27387                                       ---> the process ID number.
$ ps
PID     TTY    TIME  CMD
27355   pts/3   0:00   bash                 ---> the long shell.
27387   pts/3   0:00   bash                 ---> the subshell.
27388   pts/3   0:00   sleep 10         ---> the sleep.
27389   pts/3   0:00   ps                 ---> the ps command itself!
$ done                              ---> the output from the background process.
```

The meaning of the common column headings of ps output:

Column	Meaning
PID	the ID of the process
TTY	the controlling terminal
TIME	Amount of CPU Time
CMD	the name of the command

- **Signaling Processes: kill**

- *kill* command terminates a process before it completes.

```
kill [-signalId] {pid}
kill -l
```

- kill sends the signal with code signalId to the list of processes.

- signalId may be the number or name of a signal.

- By default, *kill* sends a TERM signal (number **15**), which causes the receiving processes to terminate.

- To send a signal to a process, you must either own it or be a super-user.

- To ensure a kill (forcefully), send signal number **9**.

```
$ sleep 1000 & sleep 1000 & sleep 1000 & ---> create three processes.
[1] 16245
[2] 16246
[3] 16247

$ ps
PID    TTY      TIME      CMD
15705  pts/4    00:00:00  bash
16245  pts/4    00:00:00  sleep
16246  pts/4    00:00:00  sleep
16247  pts/4    00:00:00  sleep
16249  pts/4    00:00:00  ps

$ kill  16245                     ---> kill first sleep.
$ ps
PID    TTY      TIME      CMD
15705  pts/4    00:00:00  bash
16246  pts/4    00:00:00  sleep
16247  pts/4    00:00:00  sleep
16265  pts/4    00:00:00  ps
[1]    Terminated               sleep 1000
```

- **OVERLOADING STANDARD UTILITIES**

```
$ cat > ls            ---> create a script called “ls”.
echo my ls
 ^D                     ---> end of input.
$ chmod +x  ls  ---> make it executable.

$ echo $PATH            ---> look at the current PATH setting.
/bin:/usr/bin:/usr/sbin
$ echo $HOME            ---> get pathname of my home directory.
/home/UG1

$ PATH=/home/UG1:$PATH     ---> update.
$ ls         --->  call “ls”.
my ls      ---> my own version overrides “/bin/ls”.
$ _
```

Note that only this shell and its child shells would be affected by the change to PATH; all other shells would be unaffected.

- **TERMINATION AND EXIT CODES**

In the Bash, Bourne and Korn shells, the special shell variable **$?** always contains the value of the previous command’s exit code.

In the C shell, the $status variable holds the exit code.

-In the following example, the date utility succeeded, whereas the cc utility failed:

```
$ date                  ---> date succeeds.
Mon  29  8  22:13:38  2016
$ echo $?               ---> display its exit value.
0                          ---> indicates success.

$ cc prog.c             ---> compile a nonexistent program.
cpp: Unable to open source file ‘prog.c’.
$ echo $?               ---> display its exit value.
1                          ---> indicates failure.
$ _
```

- *Eval BUILT-IN COMMAND*

The *eval* shell command executes the output of a command as a regular shell command.

It is useful for processing the output of utilities that generate shell commands.

-Example: execute the result of *echo* command:

```
$ echo  x=5                    ---> first execute an echo directly.
x=5

$ echo $x

$ eval  `echo x=5`             ---> execute the result of the echo.

$ echo $x
5
$ _
```