UNIT 3

3. The Shell, The Process, Customizing the environment 7 Hours

Text Book

3. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

The Shell

Introduction

In this chapter we will look at one of the major component of UNIX architecture – The Shell. Shell acts as both a command interpreter as well as a programming facility. We will look at the interpretive nature of the shell in this chapter.

Objectives

- The Shell and its interpretive cycle
- Pattern Matching – The wild-cards
- Escaping and Quoting
- Redirection – The three standard files
- Filters – Using both standard input and standard output
- /dev/null and /dev/tty – The two special files
- Pipes
- tee – Creating a tee
- Command Substitution
- Shell Variables

1. The shell and its interpretive cycle

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to ***command.com*** in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files. The original shell was the Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.

Numerous other shells are available. Some of the more well known of these may be on your Unix system: the Korn shell, ***ksh***, by David Korn, C shell, ***csh***, by Bill Joy and the Bourne Again SHell, ***bash***, from the Free Software Foundations GNU project, both based on ***sh***, the T-C shell, ***tcsh***, and the extended C shell, ***cshe***, both based on ***csh***.

Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment you key in something.

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and waits for you to enter a command.
- After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like the * in rm *) to recreate a simplified command line.
- It then passes on the command line to the kernel for execution.
- The shell waits for the command to complete and normally can't do any work while the command is running.

- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another command.

2. Pattern Matching – The Wild-Cards

A pattern is framed using ordinary characters and a metacharacter (like *) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards. The following table lists them:

Wild-Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and x
[!ijk]	A single character that is not an i,j or k (Not in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and x (Not in C Shell)
{pat1,pat2…}	Pat1, pat2, etc. (Not in Bourne shell)

Examples:

To list all files that begin with *chap*, use

```
$ ls chap*
```

To list all files whose filenames are six character long and start with chap, use

```
$ ls chap??
```

Note: Both * and ? operate with some restrictions. for example, the * doesn't match all files beginning with a . (dot) ot the / of a pathname. If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

```
$ ls .???*
```

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.

Similarly, these characters don't match the / in a pathname. So, you cannot use

```
$ cd /usr?local          to change to /usr/local.
```

The character class

You can frame more restrictive patterns with the character class. The character class comprises a set of characters enclosed by the rectangular brackets, [and], but it matches a single character in the class. The pattern [abd] is character class, and it matches a single character – an a,b or d.

Examples:

```
$ls chap0[124]          Matches chap01, chap02, chap04 and lists if found.
$ ls chap[x-z]          Matches chapx, chapy, chapz and lists if found.
```

You can negate a character class to reverse a matching criteria. For example,

- To match all filenames with a single-character extension but not the .c ot .o files, use *.[!co]
- To match all filenames that don't begin with an alphabetic character, use [!a-zA-Z]*

Matching totally dissimilar patterns

This feature is not available in the Bourne shell. To copy all the C and Java source programs from another directory, we can delimit the patterns with a comma and then put curly braces around them.

```
$ cp $HOME/prog_sources/*.{c,java} .
```

The Bourne shell requires two separate invocations of cp to do this job.

```
$ cp /home/srm/{project,html,scripts/* .
```

The above command copies all files from three directories (project, html and scripts) to the current directory.

3. Escaping and Quoting

Escaping is providing a \ (backslash) before the wild-card to remove (escape) its special meaning.

For instance, if we have a file whose filename is chap* (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as *rm chap**, as it will remove all files beginning with chap. Hence to suppress the special meaning of *, use the command *rm chap**

To list the contents of the file chap0[1-3], use

```
$ cat chap0\[1-3\]
```

A filename can contain a whitespace character also. Hence to remove a file named My Documend.doc, which has a space embedded, a similar reasoning should be followed:

```
$ rm My\ Document.doc
```

Quoting is enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted.

When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

Examples:

$ rm 'chap*'	Removes fil *chap**
$ rm "My Document.doc"	Removes file *My Document.doc*

4. Redirection : The three standard files

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device:

Standard input: The file (stream) representing input, connected to the keyboard.

Standard output: The file (stream) representing output, connected to the display.

Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.

The standard input can represent three input sources:
The keyboard, the default source.
A file using redirection with the < symbol.
Another program using a pipeline.

The standard output can represent three possible destinations:
The terminal, the default destination.
A file using the redirection symbols > and >>.
As input to another program using a pipeline.

A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input
1 – Standard output
2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.
Examples:
Assuming file2 doesn't exist, the following command redirects the standard output to file *myOutput* and the standard error to file *myError*.

```
$ ls –l file1 file2 1>myOutput 2>myError
```

To redirect both standard output and standard error to a single file use:

```
$ ls –l file1 file2 1>| myOutput 2>| myError  OR
$ ls –l file1 file2 1> myOutput 2>& 1
```

5. Filters: Using both standard input and standard output

UNIX commands can be grouped into four categories viz.,

1. Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2. Commands like ls, pwd, who etc. don't read standard input but they write to standard output.
3. Commands like lp that read standard input but don't write to standard output.
4. Commands like cat, wc, cmp etc. that use both standard input and standard output.

Commands in the fourth category are called filters. Note that filters can also read directly from files whose names are provided as arguments.
Example: To perform arithmetic calculations that are specified as expressions in input file calc.txt and redirect the output to a file result.txt, use

```
$ bc < calc.txt   > result.txt
```

6. /dev/null and /dev/tty : Two special files

/dev/null: If you would like to execute a command but don't like to see its contents on the screen, you may wish to redirect the output to a file called /dev/null. It is a special file that can accept any stream without growing in size. It's size is always zero.

/dev/tty: This file indicates one's terminal. In a shell script, if you wish to redirect the output of some select statements explicitly to the terminal. In such cases you can redirect these explicitly to /dev/tty inside the script.

7. Pipes

With piping, the output of a command can be used as input (piped) to a subsequent command.

```
$ command1 | command2
```

Output from command1 is piped into input for command2.

This is equivalent to, but more efficient than:

```
$ command1 > temp
$ command2 < temp
$ rm temp
```

Examples

```
$ ls -al | more
$ who | sort | lpr
```

When a command needs to be ignorant of its source

If we wish to find total size of all C programs contained in the working directory, we can use the command,

```
$ wc –c *.c
```

However, it also shows the usage for each file(size of each file). We are not interested in individual statistics, but a single figure representing the total size. To be able to do that, we must make wc ignorant of its input source. We can do that by feeding the concatenated output stream of all the .c files to wc –c as its input:

```
$ cat *.c | wc –c
```

8. Creating a tee

tee is an external command that handles a character stream by duplicating its input. It saves one copy in a file and writes the other to standard output. It is also a filter and hence can be placed anywhere in a pipeline.

Example: The following command sequence uses tee to display the output of who and saves this output in a file as well.

```
$ who | tee users.lst
```

9. Command substitution

The shell enables the connecting of two commands in yet another way. While a pipe enables a command to obtain its standard input from the standard output of another command, the shell enables one or more command arguments to be obtained from the standard output of another command. This feature is called command substitution.
Example:

```
$ echo Current date and time is `date`
```

Observe the use of backquotes around date in the above command. Here the output of the command execution of date is taken as argument of echo. The shell executes the enclosed command and replaces the enclosed command line with the output of the command.

Similarly the following command displays the total number of files in the working directory.

```
$ echo "There are 'ls | wc –l' files in the current directory"
```

Observe the use of double quotes around the argument of echo. If you use single quotes, the backquote is not interpreted by the shell if enclosed in single quotes.

10. Shell variables

Environmental variables are used to provide information to the programs you use. You can have both global environment and local shell variables. Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell. Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

To declare a local shell variable we use the form *variable=value* (no spaces around =) and its evaluation requires the $ as a prefix to the variable.
Example:

```
$  count=5
$  echo $count
5
```

A variable can be removed with **unset** and protected from reassignment by **readonly**. Both are shell internal commands.

Note: In C shell, we use **set** statement to set variables. Here, there either has to be whitespace on both sides of the = or none at all.

```
$ set count=5
$ set size = 10
```

Uses of local shell variables

1. Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
2. Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.
3. Concatenating variables and strings: Two variables can be concatenated to form a new variable.
   Example: $ base=foo ; ext=.c

```
$ file=$base$ext
$ echo $file        // prints foo.c
```

Conclusion

In this chapter we saw the major interpretive features of the shell. The following is a summary of activities that the shell performs when a command line is encountered at the prompt.

- Parsing: The shell first breaks up the command line into words using spaces and tabs as delimiters, unless quoted. All consecutive occurrences of a space or tab are replaced with a single space.
- Variable evaluation: All $-prefixed strings are evaluated as variables, unless quoted or escaped.
- Command substitution: Any command surrounded by backquotes is executed by the shell, which then replaces the standard output of the command into the command line.
- Redirection: The shell then looks for the characters >, < and >> to open the files they point to.
- Wild-card interpretation: The shell then scans the command line for wild-cards (the characters *, ?, [and]). Any word containing a wild-card is replaced by a sorted list of filenames that match the pattern. The list of these filenames then forms the arguments to the command.
- PATH evaluation: It finally looks for the PATH variable to determine the sequence of directories it has to search in order to find the associated binary.

The Process

Introduction

A process is an OS abstraction that enables us to look at files and programs as their time image. This chapter discusses processes, the mechanism of creating a process, different states of a process and also the ps command with its different options. A discussion on creating and controlling background jobs will be made next. We also look at three commands viz., at, batch and cron for scheduling jobs. This chapter also looks at nice command for specifying job priority, signals and time command for getting execution time usage statistics of a command.

Objectives

- Process Basics
- ps: Process Status
- Mechanism of Process Creation
- Internal and External Commands
- Process States and Zombies
- Background Jobs
- nice: Assigning execution priority
- Processes and Signals
- job Control
- at and batch: Execute Later
- cron command: Running Jobs Periodically
- time: Timing Usage Statistics at process runtime

1. Process Basics

UNIX is a multiuser and multitasking operating system. *Multiuser* means that several people can use the computer system simultaneously (unlike a single-user operating system, such as MS-DOS). *Multitasking* means that UNIX, like Windows NT, can work on several tasks concurrently; it can begin work on one task and take up another before the first task is finished.

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system. Stated in other words, a process is created. A process is a program in execution. A process is said to be born when the program starts execution and remains alive as long as the program is active. After execution is complete, the process is said to die.

The kernel is responsible for the management of the processes. It determines the time and priorities that are allocated to processes so that more than one process can share the CPU resources.

Just as files have attributes, so have processes. These attributes are maintained by the kernel in a data structure known as process table. Two important attributes of a process are:

1. The Process-Id (PID): Each process is uniquely identified by a unique integer called the PID, that is allocated by the kernel when the process is born. The PID can be used to control a process.
2. The Parent PID (PPID): The PID of the parent is available as a process attribute.

There are three types of processes viz.,

1. Interactive: Initiated by a shell and running in the foreground or background
2. batch: Typically a series of processes scheduled for execution at a specified point in time
3. daemon: Typically initiated at boot time to perform operating system functions on demand, such as LPD, NFS, and DNS

The Shell Process

As soon as you log in, a process is set up by the kernel. This process represents the login shell, which can be either sh(Bourne Shell), ksh(korn Shell), bash(Bourne Again Shell) or csh(C Shell).

Parents and Children

When you enter an external command at the prompt, the shell acts as the parent process, which in turn starts the process representing the command entered. Since every parent has a parent, the ultimate ancestry of any process can be traced back to the first process (PID 0) that is set up when the system is booted. It is analogous to the root directory of the file system. A process can have only one parent. However, a process can spawn multiple child processes.

Wait or not Wait?

A parent process can have two approaches for its child:

- It may wait for the child to die so that it can spawn the next process. The death of the child is intimated to the parent by the kernel. Shell is an example of a parent that waits for the child to terminate. However, the shell can be told not to wait for the child to terminate.
- It may not wait for the child to terminate and may continue to spawn other processes. init process is an example of such a parent process.

2. ps: Process Status

Because processes are so important to getting things done, UNIX has several commands that enable you to examine processes and modify their state. The most frequently used command is ps, which prints out the process status for processes running on your system. Each system has a slightly different version of the ps command, but there are two main variants, the System V version (POSIX) and the Berkeley version. The following table shows the options available with ps command.

POSIX	BSD	Significance
-f	f	Full listing showing PPID of each process
-e or –A	aux	All processes (user and system) processes

-u *user*	U *user*	Processes of user *user only*
-a		Processes of all users excluding processes not associated with terminal
-l	l	Long listing showing memory related information
-t *term*	t *term*	Processes running on the terminal *term*

Examples

```
$ ps
PID  TTY   TIME     CMD
4245 pts/7 00:00:00 bash
5314 pts/7 00:00:00 ps
```

The output shows the header specifying the PID, the terminal (TTY), the cumulative processor time (TIME) that has been consumed since the process was started, and the process name (CMD).

```
$ ps -f
UID    PID   PPID  C STIME    TTY   TIME COMMAND
root   14931 136   0 08:37:48 ttys0 0:00 rlogind
sartin 14932 14931 0 08:37:50 ttys0 0:00 -sh
sartin 15339 14932 7 16:32:29 ttys0 0:00 ps –f
```

The header includes the following information:

UID – Login name of the user
PID – Process ID
PPID – Parent process ID
C – An index of recent processor utilization, used by kernel for scheduling
STIME – Starting time of the process in hours, minutes and seconds
TTY – Terminal ID number
TIME – Cumulative CPU time consumed by the process
CMD – The name of the command being executed

System processes (-e or –A)

Apart from the processes a user generates, a number of system processes keep running all the time. Most of them are not associated with any controlling terminal.
They are spawned during system startup and some of them start when the system goes into multiuser mode. These processes are known as daemons because they are called without a specific request from a user. To list them use,

```
$ ps –e
PID    TTY      TIME  CMD
0      ?        0:34  sched
1      ?        41:55 init
23274  Console  0:03  sh
272    ?        2:47  cron
7015   term/12  20:04 vi
```

3. Mechanism of Process Creation

There are three distinct phases in the creation of a process and uses three important system calls viz., fork, *exec*, and wait. The three phases are discussed below:

- Fork: A process in UNIX is created with the fork system call, which creates a copy of the process that invokes it. The process image is identical to that of the calling process, except for a few parameters like the PID. The child gets a new PID.
- Exec: The forked child overwrites its own image with the code and data of the new program. This mechanism is called exec, and the child process is said to *exec* a new program, using one of the family of exec system calls. The PID and PPID of the exec'd process remain unchanged.
- Wait: The parent then executes the wait system call to *wait* for the child to complete. It picks up the exit status of the child and continues with its other functions. Note that a parent need not decide to wait for the child to terminate.

To get a better idea of this, let us explain with an example. When you enter ls to look at the contents of your current working directory, UNIX does a series of things to create an environment for ls and the run it:

- The shell has UNIX perform a fork. This creates a new process that the shell will use to run the ls program.
- The shell has UNIX perform an exec of the ls program. This replaces the shell program and data with the program and data for ls and then starts running that new program.
- The ls program is loaded into the new process context, replacing the text and data of the shell.
- The ls program performs its task, listing the contents of the current directory. In the meanwhile, the shell executes wait system call for ls to complete.

When a process is forked, the child has a different PID and PPID from its parent. However, it inherits most of the attributes of the parent. The important attributes that are inherited are:

- User name of the real and effective user (RUID and EUID): the owner of the process. The real owner is the user issuing the command, the effective user is the one determining access to system resources. RUID and EUID are usually the same, and the process has the same access rights the issuing user would have.
- Real and effective group owner (RGID and EGID): The real group owner of a process is the primary group of the user who started the process. The effective group owner is usually the same, except when SGID access mode has been applied to a file.
- The current directory from where the process was run.
- The file descriptors of all files opened by the parent process.
- Environment variables like HOME, PATH.

The inheritance here means that the child has its own copy of these parameters and thus can alter the environment it has inherited. But the modified environment is not available to the parent process.

How the Shell is created?

- When the system moves to multiuser mode, **init** forks and execs a **getty** for every active communication port.
- Each one of these **getty**'s prints the login prompt on the respective terminal and then goes off to sleep.
- When a user tries to log in, **getty** wakes up and fork-execs the **login** program to verify login name and password entered.
- On successful login, **login** for-execs the process representing the login shell.
- **init** goes off to sleep, waiting for the children to terminate. The processes **getty** and **login** overlay themselves.
- When the user logs out, it is intimated to **init**, which then wakes up and spawns another **getty** for that line to monitor the next login.

4. Internal and External Commands

From the process viewpoint, the shell recognizes three types of commands:

1. External commands: Commonly used commands like **cat**, **ls** etc. The shell creates a process for each of these commands while remaining their parent.
2. Shell scripts: The shell executes these scripts by spawning another shell, which then executes the commands listed in the script. The child shell becomes the parent of the commands that feature in the shell.
3. Internal commands: When an internal command is entered, it is directly executed by the shell. Similarly, variable assignment like x=5, doesn't generate a process either.

Note: Because the child process inherits the current working directory from its parent as one of the environmental parameters, it is necessary for the cd command not to spawn a child to achieve a change of directory. If this is allowed, after the child dies, control would revert to the parent and the original directory would be restored. Hence, cd is implemented as an internal command.

5. Process States and Zombies

At any instance of time, a process is in a particular state. A process after creation is in the *runnable* state. Once it starts running, it is in the *running* state. When a process requests for a resource (like disk I/O), it may have to wait. The process is said to be in *waiting* or *sleeping* state. A process can also be *suspended* by pressing a key (usually *Ctrl-z).*

When a process terminates, the kernel performs clean-up, assigns any children of the exiting process to be adopted by **init**, and sends the death of a child signal to the parent process, and converts the process into the zombie state.

A process in zombie state is not alive; it does not use any resources nor does any work. But it is not allowed to die until the exit is acknowledged by the parent process.

It is possible for the parent itself to die before the child dies. In such case, the child becomes an **orphan** and the kernel makes **init** the parent of the orphan. When this adopted child dies, **init** waits for its death.

6. Running Jobs in Background

The basic idea of a background job is simple. It's a program that can run without prompts or other manual interaction and can run in parallel with other active processes.

Interactive processes are initialized and controlled through a terminal session. In other words, there has to be someone connected to the system to start these processes; they are not started automatically as part of the system functions. These processes can run in the foreground, occupying the terminal that started the program, and you can't start other applications as long as this process is running in the foreground.

There are two ways of starting a job in the background – with the shell's & operator and the **nohup** command.

&: No Logging out

Ordinarily, when the shell runs a command for you, it waits until the command is completed. During this time, you cannot communicate with the shell. You can run a command that takes a long time to finish as a background job, so that you can be doing something else. To do this, use the & symbol at the end of the command line to direct the shell to execute the command in the background.

```
$ sort –o emp.dat emp.dat &
[1] 1413                    The job's PID
```

Note:

1. Observe that the shell acknowledges the background command with two numbers. First number [1] is the *job ID* of this command. The other number 1413 is the PID.
2. When you specify a command line in a pipeline to run in the background, all the commands are run in the background, not just the last command.
3. The shell remains the parent of the background process.

nohup: Log out Safely

A background job executed using & operator ceases to run when a user logs out. This is because, when you logout, the shell is killed and hence its children are also killed. The UNIX system provides nohup statement which when prefixed to a command, permits execution of the process even after the user has logged out. You must use the & with it as well.

The syntax for the nohup command is as follows:

```
nohup command-string [input-file] output-file &
```

If you try to run a command with nohup and haven't redirected the standard error, UNIX automatically places any error messages in a file named nohup.out in the directory from which the command was run.

In the following command, the sorted file and any error messages are placed in the file nohup.out.

```
$ nohup sort sales.dat &
1252
Sending output to nohup.out
```

Note that the shell has returned the PID (1252) of the process.
When the user logs out, the child turns into an orphan. The kernel handles such situations by reassigning the PPID of the orphan to the system's init process (PID 1) - the parent of all shells. When the user logs out, init takes over the parentage of any process run with nohup. In this way, you can kill a parent (the shell) without killing its child.

Additional Points
When you run a command in the background, the shell disconnects the standard input from the keyboard, but does not disconnect its standard output from the screen. So, output from the command, whenever it occurs, shows up on screen. It can be confusing if you are entering another command or using another program. Hence, make sure that both standard output and standard error are redirected suitably.

```
      $ find . –name "*.log" –print> log_file 2> err.dat &
OR    $ find . –name "*.log" –print> log_file 2> /dev/null &
```

Important:
1. You should relegate time-consuming or low-priority jobs to the background.
2. If you log out while a background job is running, it will be terminated.

7. nice: Job Execution with Low Priority

Processes in UNIX are sequentially assigned resources for execution. The kernel assigns the CPU to a process for a time slice; when the time elapses, the process is places in a queue. How the execution is scheduled depends on the priority assigned to the process.

The *nice* command is used to control background process dispatch priority.

The idea behind *nice* is that background jobs should demand less attention from the system than interactive processes.
Background jobs execute without a terminal attached and are usually run in the background for two reasons:
1. the job is expected to take a relatively long time to finish, and
2. the job's results are not needed immediately.

Interactive processes, however, are usually shells where the speed of execution is critical because it directly affects the system's apparent response time. It would therefore be nice for everyone (others as well as you) to let interactive processes have priority over background work.
nice values are system dependent and typically range from 1 to 19.

A high *nice* value implies a lower priority. A program with a high nice number is friendly to other programs, other users and the system; it is not an important job. The lower the nice number, the more important a job is and the more resources it will take without sharing them.
Example:

```
$ nice wc –l hugefile.txt
```

OR `$ nice wc –l hugefile.txt &`

The default nice value is set to 10.
We can specify the nice value explicitly with –n *number* option where *number* is an offset to the default. If the –n *number* argument is present, the priority is incremented by that amount up to a limit of 20.
Example: `$ nice –n 5 wc –l hugefile.txt &`

8. Killing Processes with Signals

When you execute a command, one thing to keep in mind is that commands do not run in a vacuum. Many things can happen during a command execution that are not under the control of the command. The user of the command may press the interrupt key or send a kill command to the process, or the controlling terminal may become disconnected from the system. In UNIX, any of these events can cause a **signal** to be sent to the process. The default action when a process receives a signal is to terminate.

When a process ends normally, the program returns its *exit status* to the parent. This exit status is a number returned by the program providing the results of the program's execution.
Sometimes, you want or need to terminate a process.
The following are some reasons for stopping a process:

- It’s using too much CPU time.
- It’s running too long without producing the expected output.
- It’s producing too much output to the screen or to a disk file.
- It appears to have locked a terminal or some other session.
- It’s using the wrong files for input or output because of an operator or programming error.
- It’s no longer useful.

If the process to be stopped is a background process, use the kill command to get out of these situations. To stop a command that isn’t in the background, press <ctrl-c>.

To use kill, use either of these forms:
kill PID(s) OR kill –s NUMBER PID(s)
To kill a process whose PID is 123 use,

```
$ kill 123
```

To kill several processes whose PIDs are 123, 342, and 73 use,

```
$ kill 123 342 73
```

Issuing the kill command sends a signal to a process. The default signal is SIGTERM signal (15). UNIX programs can send or receive more than 20 signals, each of which is represented by a number. (Use kill –l to list all signal names and numbers)

If the process ignores the signal SIGTERM, you can kill it with SIGKILL signal (9) as,

$ kill -9 123 OR $ kill –s KILL 123

The system variable $! stores the PID of the last background job. You can kill the last background job without knowing its PID by specifying $ kill $!

Note: You can kill only those processes that you own; You can't kill processes of other users. To kill all background jobs, enter kill 0.

9. Job Control

A job is a name given to a group of processes that is typically created by piping a series of commands using pipeline character. You can use job control facilities to manipulate jobs. You can use job control facilities to,

1. Relegate a job to the background (bg)
2. Bring it back to the foreground (fg)
3. List the active jobs (jobs)
4. Suspend a foreground job (*[Ctrl-z]*)
5. Kill a job (kill)

The following examples demonstrate the different job control facilities.

Assume a process is taking a long time. You can suspend it by pressing *[Ctrl-z]*.

[1] + Suspended wc –l hugefile.txt

A suspended job is not terminated. You can now relegate it to background by,

$ bg

You can start more jobs in the background any time:

$ sort employee.dat > sortedlist.dat &

[2] 530

$ grep 'director' emp.dat &

[3] 540

You can see a listing of these jobs using jobs command,

$ jobs

[3] + Running grep 'director' emp.dat &

[2] - Running sort employee.dat > sortedlist.dat &

[1] Suspended wc –l hugefile.txt

You can bring a job to foreground using fg %jobno OR fg %jobname as,

$ fg %2 OR $ fg %sort

10. at And batch: Execute Later

UNIX provides facilities to schedule a job to run at a specified time of day. If the system load varies greatly throughout the day, it makes sense to schedule less important jobs at a time when the system load is low. The at and batch commands make such job scheduling possible.

at: One-Time Execution

To schedule one or more commands for a specified time, use the at command. With this command, you can specify a time, a date, or both.
For example,

```
$ at 14:23 Friday
at> lp /usr/sales/reports/*
at> echo “Files printed, Boss!” | mail -s”Job done” boss
[Ctrl-d]
commands will be executed using /usr/bin/bash
job 1041198880.a at Fri Oct 12 14:23:00 2007
```

The above job prints all files in the directory /usr/sales/reports and sends a user named boss some mail announcing that the print job was done.
All at jobs go into a queue known as at queue.at shows the job number, the date and time of scheduled execution. This job number is derived from the number of seconds elapsed since the Epoch. A user should remember this job number to control the job.

```
$ at 1 pm today
at> echo “^G^GLunch with Director at 1 PM^G^G” >
/dev/term/43
```

The above job will display the following message on your screen (/dev/term/43) at 1:00 PM, along with two beeps(^G^G).

Lunch with Director at 1 PM

To see which jobs you scheduled with at, enter at -l. Working with the preceding examples, you may see the following results:

job 756603300.a at Tue Sep 11 01:00:00 2007
job 756604200.a at Fri Sep 14 14:23:00 2007

The following forms show some of the keywords and operations permissible with at command:

at hh:mm	Schedules job at the hour (*hh*) and minute (*mm*) specified, using a 24-hour clock
at hh:mm month day year	Schedules job at the hour (*hh*), minute (*mm*), month, day, and year specified
at -l	Lists scheduled jobs
at now +count time-units	Schedules the job right now plus *count* number of *timeunits*; time units can be minutes, hours, days, or weeks
at –r job_id	Cancels the job with the job number matching *job_id*

batch: Execute in Batch Queue

The batch command lets the operating system decide an appropriate time to run a process. When you schedule a job with batch, UNIX starts and works on the process whenever the system load isn’t too great.

To sort a collection of files, print the results, and notify the user named boss that the job is done, enter the following commands:
$ batch
sort /usr/sales/reports/* | lp
echo “Files printed, Boss!” | mailx -s”Job done” boss
The system returns the following response:
job 7789001234.b at Fri Sep 7 11:43:09 2007
The date and time listed are the date and time you pressed <Ctrl-d> to complete the batch command. When the job is complete, check your mail; anything that the commands normally display is mailed to you. Note that any job scheduled with batch command goes into a special at queue.

11. cron: Running jobs periodically

cron program is a daemon which is responsible for running repetitive tasks on a regular schedule. It is a perfect tool for running system administration tasks such as backup and system logfile maintenance. It can also be useful for ordinary users to schedule regular tasks including calendar reminders and report generation.

Both *at* and *batch* schedule commands on a one-time basis. To schedule commands or processes on a regular basis, you use the cron (short for *chronograph*) program. You specify the times and dates you want to run a command in crontab files. Times can be specified in terms of minutes, hours, days of the month, months of the year, or days of the week.

cron is listed in a shell script as one of the commands to run during a system boot-up sequence. Individual users don’t have permission to run cron directly.

If there’s nothing to do, cron “goes to sleep” and becomes inactive; it “wakes up” every minute, however, to see if there are commands to run.
cron looks for instructions to be performed in a control file in
/var/spool/cron/crontabs
After executing them, it goes back to sleep, only to wake up the next minute.

To a create a crontab file,
First use an editor to create a crontab file say cron.txt
Next use crontab command to place the file in the directory containing crontab files. crontab will create a file with filename same as user name and places it in /var/spool/cron/crontabs directory.

Alternately you can use crontab with –e option.

You can see the contents of your crontab file with crontab –l and remove them with crontab –r.

The cron system is managed by the cron daemon. It gets information about which programs and when they should run from the system's and users' crontab entries. The

crontab files are stored in the file /var/spool/cron/crontabs/<user> where <user> is the login-id of the user. Only the root user has access to the system crontabs, while each user should only have access to his own crontabs.

A typical entry in crontab file

A typical entry in the crontab file of a user will have the following format.

minute hour day-of-month month-of-year day-of-week command

where, Time-Field Options are as follows:

Field	Range	
minute	00 through 59	Number of minutes after the hour
hour	00 through 23	(midnight is 00)
day-of-month	01 through 31	
month-of-year	01 through 12	
day-of-week	01 through 07	(Monday is 01, Sunday is 07)

The first five fields are time option fields. You must specify all five of these fields. Use an asterisk (*) in a field if you want to ignore that field.

Examples:

00-10 17 * 3.6.9.12 5 find / -newer .last_time –print >backuplist

In the above entry, the find command will be executed every minute in the first 10 minutes after 5 p.m. every Friday of the months March, June, September and December of every year.

30 07 * * 01 sort /usr/wwr/sales/weekly |mail -s”Weekly Sales” srm

In the above entry, the sort command will be executed with /usr/www/sales/weekly as argument and the output is mailed to a user named srm at 7:30 a.m. each Monday.

12. time: Timing Processes

The time command executes the specified command and displays the time usage on the terminal.

Example: You can find out the time taken to perform a sorting operation by preceding the sort command with time.

```
$ time sort employee.dat > sortedlist.dat
real    0m29.811s
user    0m1.370s
sys     0m9.990s
```

where,

the *real* time is the clock elapsed from the invocation of the command until its termination.

the *user* time shows the time spent by the program in executing itself.

the *sys* time indicates the time used by the kernel in doing work on behalf of a user process.

The sum of user time and sys time actually represents the CPU time. This could be significantly less than the real time on a heavily loaded system.

Conclusion

In this chapter, we saw an important abstraction of the UNIX operating system viz., processes. We also saw the mechanism of process creation, the attributes inherited by the child from the parent process as well as the shell's behavior when it encounters internal commands, external commands and shell scripts. This chapter also discussed background jobs, creation and controlling jobs as well as controlling processes using signals. We finally described three commands viz., at, batch and cron for process scheduling, with a discussion of time command for obtaining time usage statistics of process execution.

Customizing the Environment

Introduction

The UNIX environment can be highly customized by manipulating the settings of the shell. Commands can be made to change their default behavior, environment variables can be redefined, the initialization scripts can be altered to obtain a required shell environment. This chapter discusses different ways and approaches for customizing the environment.

Objectives

- The Shell
- Environment Variables
- Common Environment Variables
- Command Aliases (bash and korn)
- Command History Facility (bash and korn)
- In-Line Command Editing (bash and korn)
- Miscellaneous Features (bash and korn)
- The Initialization Scripts

The Shell

The UNIX shell is both an interpreter as well as a scripting language. An interactive shell turns noninteractive when it executes a script.

Bourne Shell – This shell was developed by Steve Bourne. It is the original UNIX shell. It has strong programming features, but it is a weak interpreter.

C Shell – This shell was developed by Bill Joy. It has improved interpretive features, but it wasn't suitable for programming.

Korn Shell – This shell was developed by David Korn. It combines best features of the bourne and C shells. It has features like aliases, command history. But it lacks some features of the C shell.

Bash Shell – This was developed by GNU. It can be considered as a superset that combined the features of Korn and C Shells. More importantly, it conforms to POSIX shell specification.

Environment Variables

We already mentioned a couple of environment variables, such as `PATH` and `HOME`. Until now, we only saw examples in which they serve a certain purpose to the shell. But there are many other UNIX utilities that need information about you in order to do a good job.

What other information do programs need apart from paths and home directories? A lot of programs want to know about the kind of terminal you are using; this information is stored in the `TERM` variable. The shell you are using is stored in the `SHELL` variable, the operating system type in `OS` and so on. A list of all variables currently defined for your session can be viewed entering the **env** command.

The environment variables are managed by the shell. As opposed to regular shell variables, environment variables are inherited by any program you start, including another shell. New processes are assigned a copy of these variables, which they can read, modify and pass on in turn to their own child processes.
The set statement display all variables available in the current shell, but env command displays only environment variables. Note than env is an external command and runs in a child process.
There is nothing special about the environment variable names. The convention is to use uppercase letters for naming one.

The Common Environment Variables

The following table shows some of the common environment variables.

Variable name	Stored information
HISTSIZE	size of the shell history file in number of lines
HOME	path to your home directory
HOSTNAME	local host name
LOGNAME	login name
MAIL	location of your incoming mail folder
MANPATH	paths to search for man pages
PATH	search paths for commands
PS1	primary prompt
PS2	secondary prompt
PWD	present working directory
SHELL	current shell
TERM	terminal type
UID	user ID
USER	Login name of user
MAILCHECK	Mail checking interval for incoming mail
CDPATH	List of directories searched by cd when used with a non-absolute pathname

We will now describe some of the more common ones.

The command search path (PATH): The PATH variable instructs the shell about the route it should follow to locate any executable command.
Your home directory (HOME): When you log in, UNIX normally places you in a directory named after your login name. This is called the home directory or login directory. The home directory for a user is set by the system administrator while creating users (using useradd command).
mailbox location and checking (MAIL and MAILCHECK): The incoming mails for a user are generally stored at /var/mail or /var/spool/mail and this location is available in the environment variable MAIL. MAILCHECK determines how often the shell checks the file for arrival of new mail.

The prompt strings (PS1, PS2): The prompt that you normally see (the $ prompt) is the shell's primary prompt specified by PS1. PS2 specifies the secondary prompt (>). You can change the prompt by assigning a new value to these environment variables.
Shell used by the commands with shell escapes (SHELL): This environment variable specifies the login shell as well as the shell that interprets the command if preceded with a shell escape.

Variables used in Bash and Korn

The Bash and korn prompt can do much more than displaying such simple information as your user name, the name of your machine and some indication about the present working directory. Some examples are demonstrated next.

```
$ PS1='[PWD] '
[/home/srm] cd progs
[/home/srm/progs] _
```

Bash and Korn also support a *history* facility that treats a previous command as an *event* and associates it with a number. This event number is represented as !.

```
$ PS1='[!] '                $ PS1='[! $PWD] '
[42] _                      [42 /home/srm/progs] _
```

```
$ PS1="\h> "          // Host name of the machine
saturn> _
```

Aliases

Bash and korn support the use of aliases that let you assign shorthand names to frequently used commands. Aliases are defined using the alias command. Here are some typical aliases that one may like to use:

```
alias lx='/usr/bin/ls -lt'
alias l='/usr/bin/ls -l'
```

You can also use aliasing to redefine an existing command so it is always invoked with certain options. For example:

alias cp="cp –i"
alias rm="rm –i"

Note that to execute the original external command, you have to precede the command with a \. This means that you have to use \cp file1 file2 to override the alias.
The alias command with a argument displays its alias definition, if defined. The same command without any arguments displays all aliases and to unset an alias use unalias statement. To unset the alias cp, use unalias cp

Command History

Bash and Korn support a history feature that treats a previous command as an event and associates it with an event number. Using this number you can recall previous commands, edit them if required and reexecute them.

The history command displays the history list showing the event number of every previously executed command. With bash, the complete history list is displayed, while with korn, the last 16 commands. You can specify a numeric argument to specify the number of previous commands to display, as in, history 5 (in bash) or history -5 (korn).

By default, bash stores all previous commands in $HOME/.bash_history and korn stores them in $HOME/.sh_history. When a command is entered and executed, it is appended to the list maintained in the file.

Accessing previous commands by Event Numbers (! and r)

The ! symbol (r in korn) is used to repeat previous commands. The following examples demonstrate the use of this symbol with corresponding description.

$!38 The command with event number 38 is displayed and executed (Use r 38 in korn)
$!38:p The command is displayed. You can edit and execute it
$!! Repeats previous command (Use r in korn)
$!-2 Executes command prior to the previous one (r -2 in korn)

Executing previous commands by Context

When you don't remember the event number of a command but know that the command started with a specific letter of a string, you can use the history facility with context.
Example: **$!v** Repeats the last command beginning with v (**r v** in korn)

Substitution in previous commands

If you wish to execute a previous command after some changes, you can substitute the old string with new one by substitution.

If a previous command cp progs/*.doc backup is to be executed again with doc replaced with txt,

$!cp:s/doc/txt in bash
$ r cp doc=txt in korn

$_ is a shorthand feature to represent the directory used by the previous command.

$ mkdir progs
Now, instead of using cd progs, you can use,
$ cd $_

The History Variables

The command history will be maintained in default history files viz.,
.bash_history in Bash
.sh_history in Korn

Variable HISTFILE determines the filename that saves the history list. Bash uses two variables HISTSIZE for setting the size of the history list in memory and HISTFILESIZE for setting the size of disk file. Korn uses HISTSIZE for both the purposes.

In-Line Command Editing

One of the most attractive aspects of bash and korn shells is their treatment of command line editing. In addition to viewing your previous commands and reexecuting them, these shells let you edit your current command line, or any of the commands in your history list, using a special command line version of vi text editor. We have already seen the features of vi as a text editor and these features can be used on the current command line, by making the following setting:

set –o vi

Command line editing features greatly enhance the value of the history list. You can use them to correct command line errors and to save time and effort in entering commands by modifying previous commands. It also makes it much easier to search through your command history list, because you can use the same search commands you use in vi.

Miscellaneous Features (bash and korn)

1. Using set –o

The set statement by default displays the variables in the current shell, but in Bash and Korn, it can make several environment settings with –o option.

File Overwriting(noclobber): The shell's > symbol overwrites (clobbers) an existing file, and o prevent such accidental overwriting, use the noclobber argument:

set –o noclobber

Now, if you redirect output of a command to an existing file, the shell will respond with a message that says it "cannot overwrite existing file" or "file already exists". To override this protection, use the | after the > as in,

head –n 5 emp.dat >| file1

Accidental Logging out (ignoreeof): The [Ctrl-d] key combination has the effect of terminating the standard input as well as logging out of the system. In case you accidentally pressed [Ctrl-d] twice while terminating the standard input, it will log you off! The ignoreeof keyword offers protection from accidental logging out:

set –o ignoreeof

But note that you can logout only by using exit command.

A set option is turned off with set +o *keyword*. To reverse the noclobber feature, use

set +o noclobber

2. Tilde Substitution

The ~ acts as a shorthand representation for the home directory. A configuration file like .profile that exists in the home directory can be referred to both as $HOME/.profile and ~/.profile.

You can also toggle between the directory you switched to most recently and your current directory. This is done with the ~- symbols (or simply -, a hyphen). For example, either of the following commands change to your previous directory:

cd ~- OR cd –

The Initialization Scripts

The effect of assigning values to variables, defining aliases and using set options is applicable only for the login session; they revert to their default values when the user logs

out. To make them permanent, use certain startup scripts. The startup scripts are executed when the user logs in. The initialization scripts in different shells are listed below:

.profile (Bourne shell)
.profile and .kshrc (Korn shell)
.bash_profile (or .bash_login) and .bashrc (Bash)
.login and .cshrc (C shell)

The Profile

When logging into an interactive login shell, login will do the authentication, set the environment and start your shell. In the case of bash, the next step is reading the general profile from /etc, if that file exists. bash then looks for ~/.bash_profile, ~/.bash_login and ~/.profile, in that order, and reads and executes commands from the first one that exists and is readable. If none exists, /etc/bashrc is applied.

When a login shell exits, bash reads and executes commands from the file, ~/.bash_logout, if it exists.

The profile contains commands that are meant to be executed only once in a session. It can also be used to customize the operating environment to suit user requirements. Every time you change the profile file, you should either log out and log in again or You can execute it by using a special command (called dot).

```
$ . .profile
```

The rc File

Normally the profiles are executed only once, upon login. The rc files are designed to be executed every time a separate shell is created. There is no rc file in Bourne, but bash and korn use one. This file is defined by an environment variable BASH_ENV in Bash and ENV in Korn.

```
export BASH_ENV=$HOME/.bashrc
export ENV=$HOME/.kshrc
```

Korn automatically executes .kshrc during login if ENV is defined. Bash merely ensures that a sub-shell executes this file. If the login shell also has to execute this file then a separate entry must be added in the profile:

```
. ~/.bashrc
```

The rc file is used to define command aliases, variable settings, and shell options. Some sample entries of an rc file are

```
alias cp="cp –i"
alias rm="rm –i"
set –o noclobber
set –o ignoreeof
set –o vi
```

The rc file will be executed after the profile. However, if the BASH_ENV or ENV variables are not set, the shell executes only the profile.

Conclusion

In this chapter, we looked at the environment-related features of the shells, and found weaknesses in the Bourne shell. Knowledge of Bash and Korn only supplements your knowledge of Bourne and doesn't take anything away. It is always advisable to use Bash or korn as your default login shell as it results in a more fruitful experience, with their rich features in the form of aliases, history features and in-line command editing features.