8 UNIX Shell Introduction

Known in Chapter 2, UNIX provides a text-based interface or a CUI interface (see Figure 2.1). This interface is a shell, which bridges between the UNIX kernel and users. In other words, when typing in a command line in a terminal system or a terminal window, users of UNIX work on one of the shells. When a user logs on and enters a terminal system, UNIX starts running a program that is a UNIX shell (Bach 2006; Miller et al 2000; Mohay et al 1997; Quarterman et al 1985; Ritchie et al 1974; Sarwar 2006). When the shell starts running, it gives a shell prompt ($ for the Bourne or Korn shell, or % for the C shell) and waits for the user to type in commands (Bourne 1978; Bourne 1983; Joy 1980; Korn 1983; Rosenblatt et al 2002). The UNIX shell executes commands that the user types on the keyboard.

The primary purpose of the shells is as the UNIX command interpreter to interpret commands that the user types in. UNIX shells can execute programs and commands, and control program and command input and output. They can also execute sequences of programs and commands. In this chapter, we will discuss a variety of UNIX shells, UNIX shell as the command interpreter, environment variables, switching between UNIX shells, and shell metacharacters.

UNIX shells are also high-level programming languages. For the kernel, the shells are just ordinary programs that can be called by a UNIX kernel or commands. They contain some characters (such as variables, control structures, and so on) that make it similar to a programming language. Users can use one of shells to write their own short programs, called shell scripts, to accomplish particular functions, and run them on that particular shell. We will learn how to program with Bourne shell in the next two chapters.

8.1 Variety of UNIX Shells

There are many kinds of UNIX shells, just as the situation that there are many versions of the UNIX operating system. Among them, Bourne, Korn, and C shells are the most popular ones. And there are also some else, such as the Bash, TC, and Z shells (Sarwar et al 2006). And users can add their own utilities to their shells and even create totally a different shell if they like and have enough competence.

8.1.1 Shell Evolution

Usually, in a UNIX operating system, there are several different shells existing, which are likely the Bourne, C, and Korn shells. As programs have been developed more and more with time to meet the needs of users, shell programs are evolving and growing.

The Bourne shell was developed for UNIX System versions by Stephen Bourne at AT&T Bell Labs (Bourne 1978), and it is the development fundamental of other shells else and the most basic shell. The C shell was developed for BSD by William N. Joy at the University of California, Berkeley (Joy 1980). The C shell is different from the Bourne shell and mostly, simulates the C language in some way. The Korn shell was developed by David Korn at AT&T Bell Labs (Korn 1983), which is compatible with the Bourne shell and have more functions than the Bourne shell.

In Linux, there is the Bash shell, which means Bourn Again Shell. The Bash shell evolves from the Bourne shell, but includes some features of the C and Korn shells. Most of the Bourne scripts can run in the Bash shell without any modification. There are also TC and Z shells for Linux systems (Sarwar et al 2006).

The simplified shell evolution is shown in Figure 8.1, which also displays the tendency that the functions are increasing – the shell at the right of the figure has more functions than the shell at the left usually. The Bourne shell is the ancestor for all the other shells, and the Korn shell is a relatively younger and has more functions. The Figure 8.1 is just illustrative and helps readers know the shell evolution process, so they may not follow the conventions rigorously. A solid line roughly represents the closer relationship between the right and left items than a dot line.

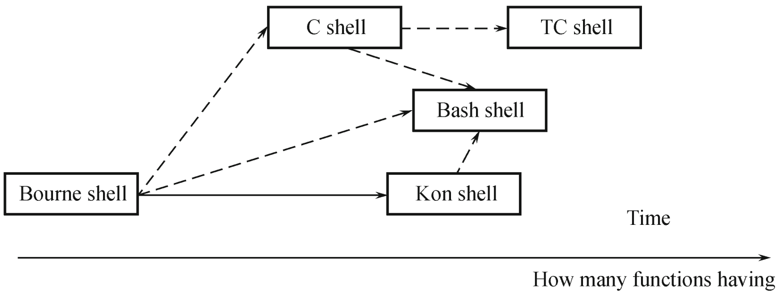

Fig. 8.1 The simplified shell evolution.

8.1.2 Login Shell

When logging on, only one shell launches. This shell is called the login shell, which is determined by the system administrator. Recall that in Section 6.1.1, the passwd file in /etc directory holds several lines of user information, each line for each user on the system. And the following is the format of the line: Login_name:password:user_ID:group_ID:user_info:home_directory:login_shell

The last field is a login_shell, which holds the absolute pathname for the user's login shell. When the user logs on, the system executes the command corresponding to the pathname in this field. The following is the example that was also given in Section 6.1.1.

```
wang:x:120:101:wang wei:/users/project/wang:/usr/bin/sh
```

where, the login shell is /usr/bin/sh, which is the Bourne shell.

To check out which shell is the login shell on the system, we can also use the echo $SHELL command just after logging in the system (see Section 6.3.1).

Usually, a shell is usually located in a corresponding directory in the file system. Table 8.1 shows a list of the most common shells, their pathnames on the file system, and the shell program names. Note: The pathnames shown here are typical for most systems, but they may be slightly different for different UNIX operating systems.

Table 8.1 Shell pathnames and program names

Shell	Pathname	Program Name
Bourne	/usr/bin/sh	sh
C	/usr/bin/csh	csh
Korn	/usr/bin/ksh	ksh
Bash	/bin/bash	bash
TC	/usr/bin/tcsh	tcsh
Z	/usr/local/bin/zsh	zsh

8.2 UNIX Shell as a Command Interpreter

When a user logs in UNIX, a login shell acts as an interface between the user and the UNIX kernel. In Chapter 2, it has been known that after logging on UNIX, some of the UNIX operating system directly provide a text-based interface or a CUI interface and some others bring the user into a GUI interface, but having a terminal window that functions like the text-based interface. In the latter situation, the user interacting environment is also in the login shell, which bridges between the UNIX kernel and the user. Therefore, when the user types in a command on a command line in a terminal system or terminal window, the user works on the login shell. When a shell starts running, it gives a shell prompt ($ for the Bourne or Korn shell, or % for the C shell) and waits for the user to type in commands. The UNIX shell executes the commands that the user types on the keyboard. From the knowledge in the previous several chapters, it is known that the user types a command and presses Enter after a shell prompt, and the shell interprets the command and executes it. That is why the shell is called the UNIX command interpreter.

8.2.1 Shell Internal and External Commands

Shell commands can be divided into two groups: the internal (built-in) commands and the external commands. As being also called built-in commands, the internal command code is part of the shell process. It means that they are part of a shell program, and the instruction is run without going out of the shell code. Usually, the commands that are the most useful and relatively shorter are embedded in the shell code.

The external commands are usually programs that are stored as binary executable files. When they are executed, the kernel needs the fork and execve system calls to create a child-process and do the execution (see Section 4.4).

When the shell is being executed, it can be terminated by the user's pressing CTRL-D on a new command line. When the shell receives this keystroke combination CTRL-D, it terminates itself and makes the user log off the UNIX. The system then prompts the login: again, and the user can use it to log in the system again.

The above is a typical whole process that a shell works.

8.2.2 Shell's Interpreting Function

For UNIX users, the shell functions like a user interface. All it needs is the ability to read from and write to the terminal, and to execute other programs.

In the kernel, the execution mechanisms for internal commands and external commands are different. For the internal commands, it is simple — just to let the instruction jump to the beginning of that part of code. For the external commands, it needs the fork and execve system calls. So the shell's interpreting function typically indicates the external command execution.

When the external command is executed, the shell interprets first the command. Typically, it treats the command line that the user just types in the following order:

- The first field, which contains the name of the executed command;
- The second field, which holds the options that are starting with a hyphen (-);
- The third field, which accommodates the command arguments.

This order is just the same as a shell program needs and following the UNIX command syntax shown in Section 2.4.1.

After reading the command line, the shell process determines whether the command is an internal or external command. It performs all internal commands by jumping to the corresponding code segments that are within its own code.

To execute an external command, the shell searches several directories in the file system structure, looking for a file that has the command name. The kernel then transfers control to an instruction located at the beginning of the file code and executes the code.

8.2.3 Searching Files Corresponding to External Commands

The pathnames of the directories that a shell uses to search the program file of an external command are called the search paths. The search paths are stored in the shell variable PATH (or path in the C shell, the former is capital-lettered, the latter is small-lettered).

To check out the PATH variable, use the echo command, which has been discussed in Section 6.3.1. The following displays the echo command again, but here with shell variables as its arguments.

```
$ echo $PATH
% echo $path
```

Function: to display shell variables on the display screen; the first one is used in Bourne, Korn, and Bash shells; the second one is used in C shell.

Note: echo can be used to send strings, file references, or shell variables to the standard output. As the search paths are discussed here, shell variables are focused on.

For example:

```
$ echo $PATH
/usr/bin:.:/users/project/wang/bin:/users/project/wang:/usr/include:
/usr/lib:/etc:/usr/etc:/usr/local/bin:/usr/local/lib:/bin
$
% echo $path
/usr/bin . /users/project/wang/bin /users/project/wang /usr/include
/usr/lib/etc /usr/etc /usr/local/bin /usr/local/lib /bin
%
```

The results of the two commands are different. The first command is for the Bourne, Korn, and Bash shells, and it separates pathnames by colons; the second command is in the C shell and separates pathnames by spaces.

8.3 Environment Variables

Some environment variables have been used in the previous chapters and sections, such as PATH, SHELL, and HOME. In this section, a list of environment variables is given and how to set them will be discussed, too.

8.3.1 Some Important Environment Variables

Different shells have their own environment variables to determine how the shells to act, how to execute commands, how to handle I/O, how to program, and the user environment. Table 8.2 lists some important environment variables for the popular shells.

Table 8.2 Some important environment variables of the Bourn, Korn and C shells

Bourne, Korn Shells	C Shell	Use
CDPATH	cdpath	Its value is used by the cd command as the pathnames to search for its argument directory; if it is not set, the cd command searches the working directory
ENV		Its value is used by UNIX as the pathname of the configuration files
EDITOR		Its value is used by some programs, such as the e-mail program and pine, as the default editor
HOME	home	Its value is the name of the user's home directory
MAIL	mail	Its value is the name of the system mailbox file
MAILCHECK		Its value is a period at which the shell should check user's mailbox for new mail and inform user
PATH	path	Its value is the search path that a shell uses to search for an external command or program
PPID		Its value is the process ID of the parent process
PS1	prompt	Its value is the shell prompt that appears on the command line. For Bourne or Korn shell, it is $; for C shell, it is %
PS2	prompt2	Its value is the secondary shell prompt displayed on second line of a command if the command is not finished. The default value is "> "(a greater-than symbol and a space). The user can continue to type in characters after the secondary shell prompt to finish the command
PWD	cwd	Its value is the name of the current directory
SHELL		Its value is the pathname for the current shell
TERM		Its value is the type of the user's console terminal

The environment variables in Table 8.2 are writable. There are also some other environment variables, such as the positional arguments, which are read-only. For the Bourne shell, Chapter 9 will give a detailed discussion.

In Section 2.6, the shell setup files have been discussed. They are some-hidden files in the home directory, such as .cshrc (for the C shell), .profile (for the Bourne or Korn shell), .login (for the C shell), .bashrc, .bash_profile, or .bash_login. They are shell setup files or configuration files. Shell setup files contain commands that are automatically executed when a new shell starts – especially when a user logs in. They also contain the initial settings of important environment variables for the shell and something else, for instance, .profile in System V and .login in BSD. And some hidden files for specific shells are executed when to start a particular shell, for example, .cshrc for C shell and .bashrc for Bash. Usually, it is the responsibility of the system administrator to modify the hidden files. But for readers, it has been introduced a little in Chapter 3 and will be discussed more in the following chapters how to change them.

In UNIX, there are also some other hidden files that are for other setup and configuration purposes rather than for shells. For all the hidden files, the same is that their names start with a dot (.).

8.3.2 How to Change Environment Variables

The environment variables can be set temporarily on the command line by using the set command and set permanently in some hidden files. In Chapter 9, detailed discussion on the set command will be given. Here, take the PATH variable as an example to practice setting the value of the environment variable.

In Table 8.2, the environment variable PATH (for Bourne or Korn shell) or path (for C shell) holds the search path that a shell uses to search for an external command or program and is set in the .profile (on System V) or .login (on BSD) file. To change the search path, set the new value of this variable.

To change the search path temporarily for the current session, just change the value of PATH at the command line. For example:

```
$ PATH=~/bin:$PATH:.
$
```

In this command, $PATH means remaining the default search path and ~/bin:$PATH:. means adding two directories (~/bin and .) into the current search path. Be careful! Don't lose the default search path settings.

For a permanent change, change the value of the PATH variable in the corresponding hidden file.

8.3.3 Displaying the Current Values of Environment Variables

To display the current values of environment variables, use the env command.
The syntax and function of env command are as follows.

```
$ env
```

Function: to display the current values of environment variables.
Common options: none.
Note: It depends on if or not the env command is available in the system. If not, the echo command can do the same job, even though the echo command shows one environment variable at a time.
For example:

```
$ env
EDITOR=/usr/ucb/vi
HOME=/users/project/wang
LOGNAME=wang
PATH=/usr/bin:.:/users/project/wang/bin:/users/project/wang:/bin
SHELL=/usr/bin/sh
......
$
```

The result of this command displays the environment variables and their values.

8.4 Switching Between UNIX Shells

Usually, in a UNIX operating system, there is more than one shell that is available. And most of the shells have some common facilities. As mentioned above, all a shell needs is the ability to read from and write to the terminal, and to execute other programs. They must have the ability to handle the I/O in order to interact with the user and the ability to execute the internal and external commands or programs. They also must be used to program in order to execute a sequence of programs and commands.

8.4.1 Why to Change Shell

Even though shells usually have some common functions mentioned above, some programs or external commands can be programmed in different shells. Hence, some of them can be executed in a specific shell while some others can be run in a different shell because not every shell is compatible to other shells well.

In fact, each shell has its own strengthens and shortcomings. For UNIX users, especially programmers, it is wise to learn the detail of some popular shells. If they learn more, they will realize the carefully learning can help them not only to know how to use the shells but also to understand how the shells to function, and even better, to design their own version of shell in the future. In addition, as there are so many shells, and programs or scripts that users get someday in the future may run in different shells, knowing the shells well can help the users execute the programs smoothly and well. By the way, it is common for UNIX users to interact with more than one shell during a session, especially among shell programmers.

8.4.2 How to Change Shell

There are two different methods to change the shell.

One way to change the shell is to execute some corresponding commands on the command line. But before changing the shell, check out which shell is the current one, use the echo command like this:

```
$ echo $SHELL
/usr/bin/sh
$
```

This command shows the pathname of the current shell.

To change from Bourne shell to C shell, use the following command:

```
$ csh
%
```

The prompt has changed from $ to %.

To change from C shell to Bourne shell, use the following command:

```
% sh
$
```

The prompt has turned from % to $.

As learned from the third example in Section 4.4.2, a parent process can fork and execute a child process for another shell that can be different from the parent process. Hence, the above method is just to create a child process that is running the new shell and the login shell is not changed, which is the parent process of the new shell.

To terminate this temporary shell and return to the original login shell, press CTRL-D on a new command line, like

```
% CTRL-D
$
```

On some systems, this method may not function well. If not, do the following:

```
% exit
$
```

The above two commands are used for the situation that the original login shell is Bourne shell. If the original login shell is C shell, the difference is the prompt changed from $ to %, not from % to $.

For other shells, the program names and pathnames can be seen in Table 8.1.

The other way to make the current shell change is to use the chsh command.

For some UNIX operating systems, the chsh command can do shell change. But it is not guaranteed for all UNIX operating systems. If it is available, the shell then prompts to type in the pathname of the new shell, which can be chosen from Table 8.1. For instance, the pathname of the C shell is /usr/bin/csh.

If the above methods cannot work well, the reasons can be one of the following: the C shell is not available on the system, it is not accessible, or the search path does not include /usr/bin. The final problem can be solved by using /usr/bin/csh instead of csh, or puting /usr/bin in the search path.

8.4.3 Searching for a Shell Program

For users, sometimes, it is necessary to know firstly whether a shell exists in the system and where it is located in the file system in order to change the current shell. To search for a program in a file system, use the whereis or which command. Here, try to use these commands to check out if or not a certain shell program is available on the system.

The syntax and function of whereis command are as follows.

```
$ whereis [option(s)] filename(s)
```

Function: to find and display on the screen the location of the binary, source, and man page files for a command.

Common options:

-b: to search only for binaries;

-m: to search only for manual sections;

-i: to search only for sources.

For example:

```
$ whereis csh
/usr/bin/csh
$
```

The result of this command is the absolute pathname of csh.

The which command can do the same job as the whereis command does.

The syntax and function of the which command are as follows.

```
$ which filename(s)
```

Function: to find and display on the screen the pathname or alias of a command.

Common options: none.

For example:

```
$ which csh
/usr/bin/csh
$
```

The result of this command is also the absolute pathname of csh.

8.5 Shell Metacharacters

In Section 6.4, three kinds of wildcards that are asterisk (*), question mark (?), and square brackets ([]) have been discussed. They can be used to save typing for a long filename or to choose many files at once. These wildcards belong to the shell metacharacters.

Shell metacharacters are some characters rather than letters and digits, and have special meaning to the shell. For their special uses, they cannot be used in filenames. As the wildcards, the shell metacharacters can also be used to specify many files in many directories in one command line. It will save users a lot of time in the future to know well how to use them in commands.

No space is required before or after a metacharacter when these characters are used in commands. However, it will be clear to use spaces before and after a shell metacharacter. Table 8.3 lists some shell metacharacters and their functions. Table 8.4 displays some shell metacharacters only used in C and Korn shells. In Table 8.3, most of the examples in the third column are the practical examples that have been used in the previous chapters. And some other examples can be used in the following chapters.

Table 8.3 Some shell metacharacters

Metacharacter	Function	Practical example
Enter	To end a command line and start a new line	
Space/ Tab	To separate elements on a command line	$ ls $HOME
#	To start a comment in a shell script	# This is a C shell script
$	To end line To substitute a shell variable	$ ls $HOME
&	To put a running command in background	$ sort longfile > longfile.sorted &
;	To separate commands in sequentially execution	$ (echo “Welcome!”; pwd); date
“ ”	To quote multiple characters To allow substituting a shell variable	$ grep -v “David” freshman $ ls “$dirct1”
‘ ’	To quote multiple characters	‘2008-8-21’
``	To substitute a command	cmd1=`ls -al`

Continued

Metacharacter	Function	Practical example
ˆ	To begin a line To negate the following characters	$ ls ∼/work/file[ˆ6]
( )	To execute a command list in a subshell	$ (echo "Welcome!"; pwd); date
{ }	To execute a command list in the current shell	$ {echo "Welcome!"; pwd}
[ ]	To surround a choice or a range of digits or letters	$ ls -i [a-zA-Z]?[1-7].html
*	To stand for any number of characters in a filename	$ more ∼/[ˆ0-9]*.[c,C]
?	To stand for a single character	$ ls -i [a-zA-Z]?[1-7].html
<	Input redirection operator	$ cat < gfreshman
>	Output redirection operator	$ date > td-time
|	To connect the output of one command to the input of another command	$ ls -la | grep "wang"
/	The root directory The separator symbol in a pathname	/usr/bin
\	To escape a single character To escape Enter character to allow continuation of a shell command on the next line	$ find. \\(-name file1 -o -name '*.c' \\) -print > myfilepn &

Table 8.4 Some shell metacharacters only for C and Korn shells

Metacharacter	Function	Practical example
%	The C shell prompt The starting character for specifying a job number	% or % 3
∼	The home directory	∼ /.profile

Except wildcards have been discussed in Chapter 6 and redirection operators and pipes have been introduced in Chapter 7, some other metacharacters are used in Chapter 4. Some new metacharacters are introduced here.

In the practical example column of Table 8.3, there is cmd1=`ls -al`. Here, the ls -al command is enclosed in the backquotes (` `). The backquote is the second-top most-left key on the keyboard.

Also in Table 8.3, the function of backslash (\) is to escape the single character after it. Usually, the following character is one of the metacharacters. As a metacharacter has its special meaning for the shell, to avoid confusing the shell and to use the character, put a back slash in front of the character to let the shell know that the character does not have the special meaning here. For example:

```
$ cat test\&1
```

```
......  The text of the test&1 is displayed here.
$
```

8.6 Summary

There are many kinds of UNIX shells. The most popular shells can be Bourne, Korn, and C shells. And there are also some else, such as Bash, TC, and Z shells.

When logging on the system, only one shell starts execution, which is called the login shell. When it starts running, a shell gives a shell prompt ($ for Bourne or Korn shell, or % for C shell) and waits for the user to type in commands. The UNIX shell interprets and executes the command.

Shell commands can be divided into two groups: the internal (built-in) commands and the external commands. The internal command code is part of the shell process. The external commands are usually programs that are stored as binary executable files. When the external commands are executed, the kernel needs the fork and execve system calls to create a child-process do the execution.

The pathnames of the directories that a shell uses to search the program file of an external command are called the search paths. The search paths are stored in the shell variable PATH (or path). Along with PATH, SHELL, and HOME, there are many environment variables. Different shells have their own environment variables to be used to customize for a user the environment, including how the shells to act, how to execute commands, how to handle I/O, and how to program. Shell setup files and some other hidden files, for instance, .profile in System V and. login in BSD, contain the initial settings of important environment variables for the shell and something else. The environment variables can be set temporarily on the command line by using the set command and set permanently in some hidden files. Using the env command can display the current values of environment variables.

Usually, in a UNIX operating system, there is more than one shell that is available. We can change the current shell with some methods: one is done by running some corresponding command on the command line; the other is by using the chsh command.

Shell metacharacters are some characters rather than letters and digits and have special meaning to the shell. As the wildcards, the shell metacharacters can also be used to specify many files in many directories in one command line.

Problems

Problem 8.1 What is a UNIX shell? Describe its main purposes and give some popular shell examples. What is a shell script for?

Problem 8.2 What is the login shell? How can you know which shell is your login shell? Give two ways of doing this and write down the login shell on your system?

Problem 8.3 How can you terminate the execution of a subshell? How about to terminate the execution of the login shell?

Problem 8.4 Why is a shell called as the UNIX command interpreter?

Problem 8.5 Give some examples of shell commands. How does the shell execute them? How can the shell execute an external command? Describe the process.

Problem 8.6 What is the search path for a shell? How can you find the search path? Check it out on your system and write down the displayed result.

Problem 8.7 What are environment variables of a shell for? What files can be used to set important environment variables for the shell initially and automatically? How can you display the current values of environment variables? Write down some of the environment variables' values on your system.

Problem 8.8 If we want to change the search path temporarily for the current session and remain the default search path, please recommend your solution to this problem.

Problem 8.9 At the shell prompt, type the set | more command, see what happens on the screen, and write down the displayed result.

Problem 8.10 At the shell prompt, type the csh command. Next, type the setenv | more command, see what happens on the screen, and write down the displayed result. Finally, press CTRL-D on a new line.

Problem 8.11 How can you change the current shell? Give your solutions for the temporary and permanent changes, respectively.

Problem 8.12 Using the whereis command, test the pathnames of the various shells listed in Table 8.1. Are all these shells available on your system? If they are, write down their pathnames.

Problem 8.13 What are the shell metacharacters? Give three examples of the shell metacharacters.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Bourne SR (1978) The UNIX shell. T Bell Syst Tech J, 57(6) Part 2: pp 1971 – 1990

Bourne SR (1983) The UNIX system. Addison-Wesley, Reading, Massachusetts

Joy WN (1980) An introduction to the C shell. UNIX Programmer's Manual, 4.2 Berkeley Software Distribution. Computer systems research group, Depat. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Aug 1980

Korn D (1983) KSH - a shell programming language. USENIX Conference Proceedings, Toronto, Ontario, June 1983, pp 191 – 202

Miller RC, Myers BA (2000) Integrating a command shell into a web browser. 2000 USENIX Annual Technical Conference, San Diego, California, 18 – 23 June 2000. http://www.usenix.org/events/usenix2000/general/generaltechnical. html. Accessed 24 Sep 2009

Mohay G, Zellers J (1997) Kernel and shell based applications integrity assurance. ACSAC'97: The IEEE 13th Annual Computer Security Applications Conference, 1997, pp 34 – 43

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Rosenblatt B, Robbins A (2002) Learning the Korn shell, 2nd edn. O'Reilly & Associates, Sebastopol

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing