

Platform: Unix

Level of Difficulty: Advanced

This document will go over more advanced topics of the Unix operating system including shells, i/o redirection, and process manipulation. It will also overview common commands and shell scripting.

Introduction: The Shell.

The shell is a special program used as an interface between the user and the heart of the Unix operating system, a program called the kernel.

The kernel is loaded into memory at boot-time and manages the system until shutdown. It creates and controls processes, manages memory, file systems, communications and so forth. All other programs, including shell programs, reside out on the disk. The kernel loads these programs into memory, executes them, and cleans up the system when they terminate. The shell is a utility program that opens when you log on. It allows users to interact with the kernel by interpreting commands that are typed at the command line or in a script file.

When you log on, an interactive shell starts up and prompts you for input. After you type a command it is the responsibility of the shell to:

1. Read input and parse the command line
2. Evaluate special characters
3. Set up pipes, background processing, and redirection.
4. Handle signals
5. Set up programs for execution.

When a command is typed at the prompt, the shell reads the line of input and parses the command line, breaking the line into words called tokens. Spaces and tabs separate tokens and a new line terminates the command line.

All Unix commands have the following syntax:

command [-flags] [arg1] [arg2] [arg3]...

The shell checks to see if the first word is a command built into the shell or located elsewhere on the disk. If it is built in, the shell will execute the command internally. If not, it searches through the paths listed in the path variable until it finds the command.

The shell will then start a sub-shell to run the command, and when finished, report the output or errors.

The order of processing the command line is:

1. History substitution. (if applicable)
2. Command line is broken up into tokens or words
3. History is updated (if applicable)
4. Quotes are processed
5. Alias substitution and variables are defined (if applicable)
6. Redirection, background, and pipes are set up.
7. Variable substitution is performed
8. Command substitution is performed.
9. Globbing, i.e. filename substitution.
10. Program execution.

Diffrent shell types:

There are 5 major shell types installed on most Rutgers servers. Tcsh is the default. To change it use the *chsh* command.

sh – the Bourne shell, the oldest of the commonly used shells. It is primitive and lacks job control features and other frills, but is the best shell for shell scripting and programming.

csh – the C shell, a shell with a C-like syntax. It also provides job control and history aliasing. Unfortunately this shell has a few bugs.

ksh – the korn shell, a Bourne-like shell with some of the features of the C shell, such as job control and history , and also allows history editing.

bash – the Bourne again shell, an improvement on the Korn shell, with a easier syntax, plus built in help commands.

tcsh – the extended C shell, this shell improves on the C shell fixing most of the bugs and adds history editing and other features.

The most commonly used shells are the bash and tcsh shells, which have the least bugs and the most features.

Processes

A process is a program in execution and can be identified by its unique PID (process identification) number. The kernel manages and controls processes. A process consists of the executable program, its data and stack, registers, and all the other information needed for the program to run. When you start the shell, it is a process. The shell belongs to a process group identified by the group's PID. Only one process group has control of

the terminal at a time and it is said to be running in the foreground. When you log on, your shell is in control of the terminal and waits for you to type a command at the prompt.

The shell can spawn other processes. When you enter a command from the prompt, the shell finds the command on the disk and arranges for the command to be executed. This is done with calls to the kernel, called system calls. A system call is a request for the kernels services and is the only way a process can access system hardware.

There are four main system calls:

Fork: the fork system call creates a duplicate of the original process. The original is called the parent and the new process is called the child. When you type a command, if it is not built into the shell, the shell invokes the fork system call to create a child shell to find the program and set it up for execution. While the child shell works, the parent usually sleeps.

Wait: the wait system call causes the parent process to suspend until one of its children terminates. If wait is successful, it returns the PID of the child that died and its exit status. If the parent does not wait and the child exits, the child is put into a zombie state until the parent calls it or the parent dies. If the parent dies, the child is inherited by the init process. The wait system call then is not just used to put the parent to sleep, but to ensure the process terminates gracefully.

Exec: after you enter a command at the terminal, the shell normally forks off a new child process. As mentioned earlier, the child is responsible for executing the program, it does this by calling the exec system call. Commands are really just executable program. When the child locates the appropriate executable, it sends the exec call. The kernel loads this program into memory in place of the shell that called it. The new program becomes the child process and starts executing.

Exit: the new program can terminate at any time by sending the exit system call. When a child process terminates, it sends a signal, and waits for the parent to accept its exit status. The exit status is a number between 0 and 255. An exit status of 0 means the program executed successfully. A non-zero exit status means that the program failed in some way.

The ps command:

The ps (processes) command displays current processes. `-f` is full format, and `-u` will take a username as an argument. For example:

```
eden>ps
  PID TTY  TIME CMD
 22047 pts/102  0:00 emacs
 18428 pts/102  0:00 bash
eden>ps -f
```

```

      UID  PID  PPID  C  STIME  TTY  TIME  CMD
nieradka 22047 18428 0 16:09:34 pts/102 0:00 emacs
nieradka 18428 18426 0 15:33:15 pts/102 0:00 -bash
eden>ps -fu nieradka
      UID  PID  PPID  C  STIME  TTY  TIME  CMD
nieradka 22047 18428 0 16:09:34 pts/102 0:00 emacs
nieradka 22048 22047 0 16:09:34 ?      0:00 /usr/local/gnu/lib/emacs/19.28/sp
arc-sun-solaris2.4/wakeup 60
nieradka 18428 18426 0 15:33:15 pts/102 0:00 -bash
eden>ps -fu paulmca
      UID  PID  PPID  C  STIME  TTY  TIME  CMD
paulmca 18701 18699 0 10:42:48 pts/72 0:00 -tcsh
eden>

```

UID is user ID, PID is process ID, PPID is parent process ID, STIME is start time, TTY is terminal type, TIME is the amount of time running, and CMD is the command that started the process.

The Environment and Inheritance

When you log on, the shell starts up and inherits a number of variables, I/O streams, and process characteristics from the */bin/login* program that started it. If a sub-shell or a child shell is created, it will inherit some of the environment from the parent, including such things as ownership, permissions, the working directory, file creation mask, special variables, open files, and signals.

Ownership

When you log on, the shell is given two identities: a user ID (UID), and one or more group IDs (GID). The UID and GID are from the */etc/passwd* file. They are integers associated with your username and group.

File permissions and file creation mask

The file creation mask determines what permissions a file will have when created. The mask can be edited with the *umask* command. The umask is a three digit octal number which is subtracted from 777 to determine the permissions in a file.

Permission Modes

<i>Octal</i>	<i>Binary</i>	<i>Permissions</i>
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

The `chmod` command changes permissions on a file. The symbolic notation is r = read, w = write, x = execute, u = user, g = group, o = others. It has three syntaxs; the shortest is by using the octal notation. The first number is the user permissions, the second the group, the third the world. Therefore:

```
chmod 755 filename
```

Results in rwx for the user, r-x for the group, and r-x for the world.

To set this to be the default permissions, use the `umask` command in your initialization files to 022 ($777 - 022 = 755$). 077 is perhaps the wisest choice, for it leaves your files unreadable and unwritable by default.

Shell Variables

There are two types of variables that can be defined in the shell, local and environmental. Some variables are created by the user and others are special shell variables.

Local Variables

Local variables are given values that are known to the shell in which they are created. Variable names must begin with a letter or underscore, and contain letters, numbers, and underscore characters. In setting a variable there can be no white space around the equal sign.

To set a variable:

```
eden>apple=orange
eden>echo $apple
orange
eden>
```

Note that the \$ indicates the beginning of a variable name. Otherwise:

```
eden>apple=orange
eden>echo apple
apple
eden>
```

Environment Variables:

Environment variables are available to the shell that they were created in, and any of its sub-shells and child processes. Some of your environment variables are created by the login process, or in your login script such as `.login` or `.profile`.

To get a list of currently defined environment variables, use the command `env`.

```

eden>env
HOSTNAME=er3.rutgers.edu
LOGNAME=nieradka
MAIL=/var/mail/nieradka
MACHTYPE=sparc-sun-solaris2.5
TERM=vt100
HOSTTYPE=sparc
PATH=/usr/local/X11R5/bin:/usr/local/X11R4/bin:/usr/openwin/bin:/usr/local/bin:/usr/ccs/bin:/usr/bin:/opt/SUNWspro/bin:/usr/ucb:/usr/local/gnu/bin:/usr/local/netpbm:/usr/ccs/bin:/eden/u7/nieradka:/eden/u7/nieradka/bin:/eden/u18/paulmca/bin:/usr/sbin
HOME=/eden/u7/nieradka
SHELL=/bin/bash
PS1=eden>
HZ=100
PS2=~~
USER=nieradka
MANPATH=/usr/openwin/man:/usr/dt/man:/usr/local/X11R5/man:/usr/local/man:/usr/local/gnu/man:/usr/man:/opt/SUNWspro/man
OSTYPE=solaris2.5
SHLVL=1
EDITOR=/usr/local/bin/emacs
TZ=US/Eastern
_=/usr/bin/env
_1411_GNU_nonoption_argv_flags_=0
eden>

```

HOSTNAME -- the machine you are currently logged into.
LOGNAME – username of the current user
MAIL – location of inbox
MACHTYPE – type of machine you are logged into.
TERM – Terminal type
PATH – the paths searched to find a command. Unix searches from the first directory in the path variable to the last, and uses the first executable it finds.
HOME – the home directory.
SHELL – the default shell to run.
PS1 – the primary prompt.
PS2 – the sub-shell prompt.
USER – the current user.
MANPATH – the paths to search through for man pages.
SHLVL – the level of shells, or how many generations of shells from login to current.
EDITOR – the default editor
TZ – the time zone.
PWD – the current working directory.

To edit your environmental variables, or to define more, edit your `.login` (csh and tcsh) or `.profile` (sh, ksh, or bash) file.

Other special variables:

`$$` the PID of the current shell

- \$- the shell options currently set.
- \$? the exit value of the last executed command.
- \$! the PID of the last job put in the background.

File Descriptors and I/O redirection.

All input/output including files, pipes, sockets, are handled by the kernel via a mechanism called a file descriptor. A file descriptor is a small unsigned integer, an index into a file descriptor table maintained by the kernel. Each process inherits its file descriptor table from its parent.

0, 1, and 2 are assigned to your terminal. 0 is the standard input (*stdin*), 1 is the standard output (*stdout*), and 2 is the standard error (*stderr*). When a file is opened, the next available file descriptor is assigned to the file, therefore the first file opened would have a file descriptor of 3. The *stdin* is usually your keyboard, the *stdout* the screen, and *stderr* the screen.

Input output Redirection

When a file descriptor is assign to something other than a terminal it is called input/output redirection. The shell performs redirection of output to a file by closing the standard output, file descriptor 1, and then assigning that descriptor to a file.

On the command line, to redirect input/output:

- > Redirect the output to a file
- < Redirect the input from a file
- | (Pipe) Use the output of one command as the input to the second.

For example:

```
ls -al > file1
```

This command would take the output from the command `ls -al` and put it in a file called `file1` instead of to the screen.

```
mail bob < file2
```

This would mail the user bob the contents of `file2`.

```
man ls | lpr -Pcaclp4
```

This would take the output of the `man ls` command and feed it to the `lpr` command. Effectively this would print out the manual of `ls` to `caclp4`.

Other special command characters

Command Forms:

<code>cmd &</code>	Execute <code>cmd</code> in the background.
<code>cmd1 ; cmd2</code>	Command sequence, multiple commands on the same line.
<code>(cmd1 ; cmd2)</code>	Subshell, treat <code>cmd1</code> and <code>cmd2</code> as the same command group.
<code>cmd1 cmd2</code>	Pipe; use the output of <code>cmd1</code> as the input for <code>cmd2</code> .
<code>cmd1 `cmd2`</code>	Command substitution. Use the output of <code>cmd2</code> as the arguments to <code>cmd1</code> .
<code>cmd1 && cmd2</code>	And; execute <code>cmd2</code> if <code>cmd1</code> succeeds.
<code>cmd1 cmd2</code>	Or; execute <code>cmd2</code> if <code>cmd1</code> fails.
<code>{cmd1 ; cmd2}</code>	Execute commands in current shell.

Redirection:

<code>cmd > file</code>	Write output of command to file.
<code>cmd >> file</code>	Append output of command to end of file.
<code>cmd < file</code>	Take input for <code>cmd</code> from file.
<code>cmd << text</code>	Read standard input up into a line identical to text.

/dev/null

A useful tool with redirection is the device `/dev/null`. `/dev/null` is not a normal device, but a special device that eats anything given to it and returns “end of file.”

This is a useful tool, as in the following examples:

```
cp /dev/null file1
```

Empties a file, but does not delete it.

```
program1 > /dev/null/ &
```

Squelches output on `program1` running in the background.

Sending Signals: Inter-process communication

Signals are the way one program sends a message to another program. There are only 32 signals that are defined and can be sent to another process or the kernel. The important ones are:

Signal Name	Number	Description
HUP	1	Hang up. Stop running. Sent when you disconnect from a modem.
INT	2	Interrupt. Stop running. <code><Ctrl>c</code>
QUIT	3	Quit. Stop running and dump core. <code><Ctrl>/</code>
KILL	9	Kill. Stop immediately and with prejudice.

TERM	15	Terminate. Terminate gracefully if possible. kill command.
TSTP	18	Stop executing, suspends. <Ctrl>z
CONT	19	Continue executing.
EOF		End of File. <Ctrl>d.

Kill command

The kill command sends signals to other processes. Its default signal is 15 (the terminate signal) but any of the above signals can be sent. The syntax is:

```
kill -[#] [pid]
```

For example:

```
eden>cat /dev/zero &
[1] 9797
eden>ps
  PID TTY  TIME CMD
  9725 pts/47  0:00 bash
  9797 pts/47  0:00 cat
eden>kill -2 9797
eden>
[1]+  Interrupt          cat /dev/zero
eden>
```

Foreground and Background Jobs

The shell keeps a table of current jobs, printed by the jobs command. When a job is started in the background with &, the shell prints a line which looks like: [1] 1234 indicating that the job was started as background job number 1 and has a process id of 1234. There are several ways to refer to jobs in the shell. To refer to a job with job number 1, you can refer to it with %1. Just naming a job brings it to the foreground; therefore, either %1 or fg %1 will bring job 1 into the foreground. Similarly saying %1 & returns job 1 to the background; this also may be done with bg %1.

The command *jobs* has a notation for current and previous background jobs. The current background job is marked with a '+' and the previous jobs with a '-'. %+ starts the current job and %- starts a previous job.

```
-----
eden>cat /dev/zero
ð^Z
[1]+  Stopped          cat /dev/zero
```

```
eden>jobs
[1]+  Stopped                  cat /dev/zero
eden>fg %1
cat /dev/zero
```

When you try to leave the shell while jobs are stopped, you will be warned that you have stopped jobs. If you immediately try to exit the shell again, the shell will not warn you a second time, and the suspended jobs will be terminated. You may use the jobs command to see what those jobs are and if you want to terminate them.

You can terminate a background job using the kill command. kill %1 will terminate job number 1 and kill 1234 will kill the job with a process id of 1234.

If you try to logout and there are suspended jobs, you will be warned of this and the logout will be aborted. If you immediately try to logout again these jobs will be killed and you will be logged out. If you must start a process that is to continue after you logout, precede the command for that process with the nohup command and be sure you have provided for all of its input and output to go to files.

Other Job Control Commands

batch

Execute commands on the standard input. End with end of file. Runs a series of commands in order, waiting for the first to finish before continuing. Useful for running several memory hungry programs in the background.

at

at executes a command entered on the standard input at a later time and date. It allows many different ways of entering the intended time of execution. For example the following are acceptable syntaxs:

```
at 1:32pm Nov 11
at 17:40 Oct 30
at now + 5 min
at noon next day
```

See the man pages for more details.

Wildcards

Wildcards are the shell's way of abbreviating filenames. A wildcard allows you to select a group of files, or type in an incomplete name. The common wildcards are:

- * Match zero or more characters. Therefore a* matches ab ac abc acb.4 etc.
- ? Match exactly one character. Therefore a? matches ab ac but not abc
- [a..2..c] Match any of the indicated characters. Therefore, a[a..b..2] matches aa ab a2.
- [1-4] Matches any character that is between the two characters. Therefore a[1-3] will match a1 a2 a3.

For example:

```
eden>ls
bob bob.3 bob.4 bob.z boring file.4 file1 file2 file3
eden>ls b*
bob bob.3 bob.4 bob.z boring
eden>ls bob.?
bob.3 bob.4 bob.z
eden>ls file[1-4]
file1 file2 file3
eden>ls f*[2-4]
file.4 file2 file3
eden>ls ?*.4
bob.4 file.4
eden>rm *.*
eden>ls
bob boring file1 file2 file3
eden>rm ?????
eden>ls
bob boring
eden>
```

For more help, contact the consultant on duty, or mail *help@server*. (i.e. help@eden, help@remus, help@rci, etc.)