CHAPTER 2

- Commands—Are These Real Words?
- User Configuration—What Environment Am I In?
- Shells
- Fun Commands
- AIX Commands

Using AIX

Commands—Are These Real Words?

The basic AIX commands (and all UNIX system commands) are, for the most part, very short, cryptic, two-letter command names. Imagine back years ago, when computers had only very slow teletype keyboards and paper "displays." (Some of us aren't imagining, we're remembering!) Imagine also, people who didn't like typing long commands because there was such a long delay between commands and the computer response. If there were any mistakes, the user had to retype the whole thing (especially aggravating for folks that type with only two fingers!).

Also, some UNIX commands came from university students and researchers who weren't bound by usability standards (no rules, merely peer pressure). They could write a very useful, clever command and name it anything—their own initials, for example (`awk` by Aho, Weinberger, and Kernighan), or an acronym (`yacc`, Yet Another Compiler-Compiler).

User Configuration—What Environment Am I In?

There are some setup files that create an environment when you log in. These files are executed automatically to give you a customized system. You can use what came in the box, or change it to your special way of working.

The files `/etc/profile` and `/etc/environment` are executed for all users of the system and the file `.profile` in your home directory (a hidden file) is executed when you log into the system (except for C shell users).

Usually these files set environment variables such as "PATH" which lists all the directories you want AIX to look into for the command you type in. It can also set your prompt, default editor, and mail message when new mail is received.

You can add AIX commands in your `.profile` that you want executed every time you log in (see `cron` for more information about executing commands automatically at other times of the day or night). My favorite command to put in my `.profile` file is `/usr/games/fortune` (this command is in fileset `bos.games`).

The `.dtprofile` is used by the common desktop for initial configuration information. The `.mh_profile` is used by the mail handler, MH, for user customization. The `.mwmrc` file is used by the motif window manager.

By convention the `.kshrc` file is executed by the Korn shell when you log in (the line `export ENV=$HOME/.kshrc` must be in your `.profile` file). Alias definitions are usually stored in the `.kshrc` file. The `.login` and `.cshrc` files in your home directory are executed by the C shell.

A file named `/etc/motd` (Message Of The Day) is displayed when users log into the system. It doesn't really set anything up but it shows up during your login. You can use this message for daily reminders. If multiple people share the system, it can be used to announce system changes, or more importantly, birthdays!

Shells

UNIX systems offer a shell around the system that allows you to execute commands and write programs with those commands. No compiler is needed. The shell interprets the logic.

Over the years, several shell versions have evolved and AIX provides the ones our customers want. The shells available in AIX systems are the Bourne shell, the C shell, the Korn shell, variations on these shells, the Restricted shell, and the Trusted shell.

The first shell, the Bourne shell (`/usr/bin/bsh`), was created by Steven Bourne at Bell Labs. The Bourne shell is a popular shell to use when writing "shell scripts", or text files that use shell programming constructs to carry out some task.

The C shell (`/usr/bin/csh`) was created by Bill Joy at the University of California at Berkeley. It uses a format similar to the C programming language. It lets you recall and employ previously used commands, define and use aliases, and do job control. The phrase "job control" here is not related to empowerment, but rather the ability to suspend currently running programs, resume them, and/or move them to/from the background/foreground.

The Korn shell (`/usr/bin/ksh` and `/usr/bin/sh`) was developed by David Korn at AT&T Bell Labs. Although it is based on the Bourne shell, it also draws upon functionality found in the C shell, and thus supports recalled commands, aliases, and job control.

The Restricted shell (`/usr/bin/Rsh`) is based on the Bourne shell and prevents users from changing directories, changing the PATH and SHELL environment variables, executing programs not specified in the current PATH, modifying or viewing any files not in the current directory, and redirecting output. This may sound pretty restrictive, but to be truly restricted, the system administrator should ensure that the user does not have access to any program such as `vi` in the PATH that could be used to overcome these limitations.

The Trusted shell (`/usr/bin/tsh`) is based on the Korn shell and gives the user a slightly restricted environment intended to protect from `trojan horses` (a user modified command that masquerades as a real command), computer viruses, and other undesirables. The Trusted shell does not read in user profile files such as `.profile`, but rather only reads the `/etc/tsh_profile` file. It will not support user function and alias definitions or command history, and uses several additional built-in commands (`logout`, `shell`, and `su`) to ensure that the intended command is executed. This book does not go into detail about computer security. That's a whole book in itself!

The Bourne shell was the default for AIX Version 3 and the Korn shell is the Version 4 default. That is, on AIX Version 3, `/usr/bin/sh` and `/usr/bin/bsh` are the same, and on AIX Version 4, `/usr/bin/sh` and `/usr/bin/ksh` are the same. The Korn shell has always been recommended as the default login shell on AIX. It was made the default system shell in AIX Version 4 since the AIX Version is standards compliant (XPG4 and POSIX).

There are several important shell concepts that all users should be familiar with. These include environment variables, aliases, I/O redirection, command history, file globbing, and regular expressions. Each of the shells supports programming constructs which allow users to quickly write powerful "shell scripts" which enable automation of nearly anything you can do on the command line. One of the great things about shell programming is that you can edit the shell program and run it without the compiling step; that makes it much faster. However, we will not cover shell programming here. If you are interested in that, or just want to know more about shells, you may wish to check out one of the following books:

1. Morris Bolsky and David Korn, *The New Korn Shell Command and Programming Language,* New Jersey: Prentice-Hall, 1995.
2. Daniel Gilly, *Unix in a Nutshell,* Sebastopol, CA: O'Reilly & Associates, 1992.
3. Bill Rosenblatt, *Learning the Korn Shell,* Sebastopol, CA: O'Reilly & Associates, 1994.
4. Stephan Kochan and Patrick Wood, *Unix Shell Programming,* New Jersey: Hayden Books, 1985.

5. Gary Anderson, *The Unix C Shell Field Guide,* New Jersey: Prentice-Hall, 1986.

6. Martin Arick, *Unix C Shell Desk Reference,* QED Technical Publishing Group, 1992.

Environment Variables

The environment is the system "sky" all around you when you're using a UNIX system. There are all kinds of things floating around out there (some you see, some you don't) that affect the way the system acts. There are predefined environment variables that contain information your applications may want to use. You can create variables for your own purposes.

Although each shell supports environment variables, they each tend to have a different method to change them. To set environment variable FOO to AIX, on Bourne shell or Korn shell, type `FOO=AIX; export FOO`. When using the C shell, type `setenv FOO AIX`. You can change environment variables at the command line, and the change will be effective until you log off. If you want to permanently change your environment then you will have to edit the appropriate configuration file (`.profile` for the Korn shell and Bourne shell, and `.login` for the C shell). To view a single environment variable such as FOO, type `echo $FOO`. If you want to see all of your environment variables, then type `env` and you will see something like this:

```
MANPATH=/afs/austin/common/usr/man
LANG=En_US
LOGIN=carolynj
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
VISUAL=vi
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin:.
HISTFILE=.sh_history
LOGNAME=carolynj
MAIL=/usr/spool/mail/carolynj
LOCPATH=/usr/lib/nls/loc
USER=carolynj
SHELL=/bin/ksh
HOME=/afs/austin/u/carolynj
CMVC_FAMILY=aix
TERM=dtterm
MAILMSG=[YOU HAVE NEW MAIL]
```

```
PWD=/afs/austin/u/carolynj
TZ=CST6CDT
```

Generally environment variables serve to communicate configuration information to either the shell or other system programs. Other environment variables convey information from the shell to the user or other programs. Some of the common environment variables that are supported in all shells include:

- **HOME** This environment variable refers to your home directory. This is the most frequently used variable.
- **LANG** This environment variable refers to the current locale. If you set this environment variable to 'C' you will get the best performance. By setting it to other locales, you can make your system support a variety of national languages.
- **MANPATH** This environment variable specifies a list of paths to search for when looking for man (manual publications) pages with the `man` command.
- **NLSPATH** This environment variable specifies where files of translated messages called "message catalogs" can be found.
- **PATH** This environment variable specifies which directories the shell should check and in what order when looking for a command that you type in.
- **LOCPATH** This environment variable specifies which directories the system should check when looking for the locale specified by the LANG environment variable.
- **USER** The shell sets this to your user ID.
- **SHELL** This environment variable specifies which shell you want to use when executing a shell from another program.
- **TERM** This variable specifies your terminal type (aixterm, dtterm, xterm, and vt100, are popular values depending on what you use).
- **TZ** This variable specifies your time zone, along with some interesting information such as whether or not your computer should follow daylight savings time, and so on.

Aliases

Many command line users like to set up aliases for long or complex commands to save typing (or they may have a bad memory). This is especially helpful while a person is learning or if they are used to different commands that map into AIX commands. Frequently typed commands are good candidates.

If you type the `alias` command it will show you all your current aliases:

```
ls='/usr/bin/ls -F'
stop='kill -STOP'
suspend='kill -STOP $$'
myfiles='ls /u/home/trent'
```

To create an alias `ls` that replaces the `ls` command with a customized option, type `alias ls='ls -F'` on the Korn shell. To do the same on the C shell type `alias ls 'ls -F'`. (Notice that the C shell does not like the equal sign.) Bourne shell does not support aliases.

I/O Redirection and Piping

Once you get used to redirection and piping, it will be your favorite and frequent sequence of commands. There is no limit to the number of combinations of commands you can put together to get something done. It's easy and quick.

In its simplest form, I/O redirection can be used to prevent output from a command from going to the screen, and instead send it to a file. Likewise, many commands will take input from a file if it is specified with I/O redirection.

For example, you can redirect *output* from the `ls` command to a file like this:

```
$ ls
Mail a.out hello.c todo
$ ls > output
$ cat output
Mail
a.out
hello.c
output
todo
```

You can also redirect *input* to a command such as `cat`. (I know this is sort of trivial, since the `cat` command will take input from a file anyway!)

```
$ cat < hello.c
main()
{
printf("Hello world\n");
}
$
```

More interesting is the ability to "pipe" the output from one UNIX command to another UNIX command. For example, suppose you have a command that generates a lot of output, such as the `ls -al` command on a large directory. You

can pipe its output to other commands that process it. For instance, you could count the number of lines to get an idea of how many files are in the directory with the `wc -l` command like this:

```
$ ls -al /usr/bin | wc -l
     647
```

Now we know that there are 647 files in the directory `/usr/bin`.

Piping from one command to another, and maybe another after that, saves you disk space. You don't have to save the output from each command, just the final output.

Command History

Wouldn't you like to reuse the command you just typed, instead of retyping the whole thing? Or would you like to see a list of the commands you just executed to see how you got into a mess? You'll learn how in this section.

Since command history is not supported on the Bourne shell, we will describe how it works on the Korn shell and the C shell. By default the Korn shell keeps track of the last 128 commands. You can see the last 15 of them by simply typing `history`. You can then execute a specific command again by typing `r` followed by the number of that command, or you could type `r l` to rerun the last command that started with the letter `l`. You can also just use `r` by itself to rerun the last command:

```
$ history
1       ls
2       vi hello.c
3       echo Hi Mom!
4       history
$ r 3
echo Hi Mom!
Hi Mom!
$ r l
ls
a        b       file
```

To use command history with the C shell, you will need to ensure that you are using AIX Version 4.1.4 or later, and that the following two commands are in your `.cshrc` file:

```
set savehist=100
set history=100
```

Then, similar to the Korn shell, you can list recent commands with the history command, and can rerun commands by using the exclamation point (!) character. To repeat the previous command, use `!!`:

```
$ history
1       ls
2       vi hello.c
3       echo Hi Mom!
4       history
$ ! 3
Hi Mom!
$ ! l
a        b       file
$ !!
a        b       file
```

You can recall a recent command, edit it, and then execute the modified command. To do this, you'll have to learn the `vi` editor (see Chapter 4, Editors) if you don't already know it. You really need to learn `vi` for many reasons, but I'll give that pep talk later. To set up the ability to edit the recalled commands, type:

```
set -o vi
```

Press `<Esc> k` to recall the last command, then keep pressing the "k" key to get the commands before that until you get to the one you want. Then use the `vi` editor commands to change the command. Then press `Enter` and execute it.

File Globbing Goes Wild!

File "globbing" is the technical UNIX way to refer to the use of wild cards in filenames to specify a group of files. This can also be called "Pattern Matching Notation for Files", but I think that "globbing" sounds more interesting. Most users are familiar with the use of "a*" to refer to all files in the current directory that start with the letter "a". Wherever you can type a file name in a UNIX command, you can generally use file globbing.

This section looks scary because of all the special characters, but don't be intimidated. Read through it and pick out a couple things to try and use. You'll be surprised how useful it will be!

File globbing consists of the use of a number of metacharacters that can be used as a form of shorthand to match other characters and files. There are only three main kinds of file globbing characters. The simplest type of file globbing is the question mark (?), which will match ANY single character. The question mark (?)

will match “a” or “q” or “z” or “w” or any other single character. Typing `ls ?` would list all single character files in the current directory.

The asterisk (*) will match any number of characters including NO characters. Some examples are listed here:

- `*` matches all files.
- `*a*` matches any file that has an “a” in it somewhere.
- `a*` matches any file that starts with an “a”.
- `*a` matches any file that ends with an “a”.
- `a*a` matches any file that starts and ends with a different “a”, including “abracadabra”, and “aa”, however it will not match “a”, since each “a” in the pattern has to match to a unique “a” in the file name.

The most interesting (and complicated) type of file globbing is the bracket expression. Like the question mark, it only matches a single character. There are three main types of bracket expressions. A simple bracket expression like `[abc]` will match a single “a” or a single “b” or a single “c”, but nothing else. Bracket expressions can also contain ranges; `[a-z]` will match any single lower case character in the En_US locale. This form of bracket expression should actually be avoided since the characters in the range from “a” to “z” will depend on the locale and code set being used. Rather, the preferred way to use a bracket expression to match something like “all lowercase letters” is to use a “character class expression,” such as `[[:lower:]]`, which will match any single lowercase character no matter what locale is being used. The following character classes are available for use in bracket expressions in all locales:

- `[[:alnum:]]` Match a single alpha-numeric character.
- `[[:alpha:]]` Match a single alphabetic character.
- `[[:blank:]]` Match a single blank character (space or tab).
- `[[:cntrl:]]` Match a single control character.
- `[[:digit:]]` Match a single numeric character.
- `[[:graph:]]` Match a single graph character (all printable characters minus the space character).
- `[[:lower:]]` Match a single lowercase character.
- `[[:print:]]` Match a single printable character (including space).
- `[[:punct:]]` Match a single punctuation character.
- `[[:space:]]` Match a single white space character.

- `[[:upper:]]` Match a single uppercase character.
- `[[:xdigit:]]` Match a single hexadecimal numeric character.

One nifty feature of bracket expressions is the ability to specify a "nonmatching list" in which the first character in the bracket expression is the exclamation point (!). When this happens, the bracket expression will match any character that is NOT inside the brackets. For example, `[!abc]` will NOT match "a", "b", or "c", but will match "d", "e", "f", and so on. If the exclamation point is not the first character in the expression then it loses its special meaning. For example, `[a!b]` will match "a", "!", or "b". Bracket expressions can also match collating symbols and equivalence classes.

Of course, all three of these types of expressions can be combined to form complex and useful patterns. You could use an expression such as `version1.[abc]*.c` to make a list of all files that start with `version1.a` or `version1.b` or `version1.c` and end in `.c`. Or if you want to see a file that is called either `bob` or `Bob` or `8ob`, you could use the pattern `?ob`. These few constructs allow for a wide variety of file-matching possibilities.

Here are a couple of down-to-earth examples of how I use file globbing:

- `rm *.tmp` (remove temporary files in directory)
- `cc *.c` (compile all c programs in directory)
- `cat prog[123]?.c` (`cat` certain c programs)
- `ls [adt]*` (look at all files or directories starting in `a`, `d`, or `t`)

What do you do if you want to actually use one of these special characters in a file name? You can use metacharacters such as *?[] in file names by simply quoting the file name with either single or double quotes. For example, `cat '*'` would `cat` a file called '*' rather than `cat` all files in the directory.

Regular Expressions

Regular expressions are related to, but have significant differences from, file globbing. There are two types of regular expressions: Basic Regular Expressions (BRE) and Extended Regular Expressions (ERE). For the most part, simple BREs will work on programs expecting EREs for input. The type of regular expression a program expects depends on the program. For example, `grep` expects a BRE as the matching pattern, and `egrep` expects an ERE as the matching pattern. Although we will cover all major features of regular expressions, we will not detail every nuance. The best source for more information is either IEEE's POSIX or X/Open's XPG4 standards documents. Other useful sources include InfoExplorer and the O'Reilly book, *Sed & awk*.

- BRE Wildcard Metacharacter (.)

In a BRE, the period matches any single character. It is similar to the question mark in file globbing.

- BRE Interval Metacharacter (*)

The asterisk matches zero or more occurrences of the previous element. So "a*" will match one or more "a"s. If you wish to make an expression that will match anything, then ".*" is the correct expression rather than "*".

- BRE Beginning of Line Anchor Metacharacter (^)

The circumflex will match the beginning of a line if it occurs as the first character in the pattern. However, it matches itself if it is not the first character in the pattern. (Circumflex has other meanings within bracket expressions.) The command `grep ^a` will match all lines that start with the letter "a", and `grep a^` will match all lines that contain the string "a^". Use of the circumflex to match the beginning of a line is referred to as anchoring, since it anchors the pattern to the beginning of the line.

- BRE End of Line Anchor Metacharacter ($)

The dollar sign will match the end of a line if it occurs as the last character in the pattern. However, it matches itself if it is not the last character in the pattern. The string `grep a$` will match all lines that end with the letter "a", and `grep $a` will match all lines that contain the string "$a". Use of the dollar sign to match the end of a line is referred to as anchoring, since it anchors the pattern to the end of the line.

The $ and ^ metacharacters can be combined to match an entire line. For instance, the pattern "^apple$", will match only a line that contains only the string "apple", and "^$" will match only empty lines.

- BRE Subexpressions

Any valid BRE can be enclosed in escaped parenthesis to form a subexpression. For example, "\(apple\)" will match the string "apple", as will the pattern "\(app\)\(le\)", as will the pattern "\(a\(ppl\)e\)".

- BRE Backreferences

It is possible to repeat a subexpression with the metacharacter string "\#", where # is a digit between 1 and 9. The string "\1" refers to the first started subexpression, "\2" refers to the second started subexpression, and so on. The pattern "\(apple\)\1" will match the string "appleapple".

- BRE Duplication

Any character, metacharacter, or subexpression can be duplicated a specified number of times. The construct "\{m\}" will match exactly m occurrences of the

preceding character, metacharacter, or subexpression, "\{m,\}" will match m or more occurrences, and "\{m,n\}" will match between m and n occurrences inclusive. Examples, "a\{3\}" will match "aaa", "a\{3,\}" will match "aaa", and "aaaaa", but not "aa", "a\{2,3\}" will only match either "aa" or "aaa". Additionally, ".\{3\}" will match any three characters, and "\(apple\)\{2\}" will match "appleapple".

- BRE Bracket Expressions

Similar to file globbing, BREs support bracket expressions that match a single character. Bracket expressions can contain a list of characters to match (for example, "[abc]" still matches either "a" or "b" or "c"), a range expression (for example, "[a–c]" will match "a" or "b" or "c" in the En_US locale), and character class expressions such as "[[:upper:]]" match any single uppercase character. Like file globbing, bracket expressions can also match collating symbols and equivalence classes. Unlike file globbing "nonmatching lists" are specified by using a circumflex as the first character in a bracket expression, rather than the exclamation point. For example "[^abc]" will NOT match "a", "b", or "c", but it will match "d", "e", "f", and so on. If the circumflex is not the first character in the expression, then it loses its special meaning. For example "[a^b]" will match "a", "^", or "b".

- BRE Bracket Expressions

As with file globbing, these expressions and characters can be combined to arbitrary complexity. For example "[abc]\{2,3\}\([[:upper:]]\)\1..foo$" will match a string that has two or three characters from the set "abc", followed by any uppercase character repeated twice, followed by any two characters, followed by the string "foo" that must be at the end of the line.

That pretty much wraps up a simple introduction of Basic Regular Expressions. Extended Regular Expressions provide for several more features than BREs, and in some cases provide for a cleaner pattern, since parenthesis and braces are not escaped as in BREs. I will describe the main differences between BREs and EREs here.

- ERE Subexpressions vs. BRE Subexpressions

With EREs you do not need to precede parenthesis with backslashes. So the BRE "\(abc\)" should be written as "(abc)" as an ERE.

- ERE Brace Range Intervals vs. BRE Brace Range Intervals

As with subexpressions, with EREs you do not need to precede braces with backslashes. So the BRE "a\{5\}" should be written as "a{5}" as an ERE.

- ERE Anchoring with ^ and $ vs. BRE Anchoring with ^ and $

The main difference in anchoring between ERE and BRE is how the metacharacters ^ and $ are treated if they're not at the beginning and end (respectively) of the pattern. With BREs they lose their special meaning and get treated as the characters themselves, however with EREs they are assumed to retain their special anchor meaning. So an ERE like this "a$b" will try to match an "a" at the end of a line, followed by a "b" on the next line. However, most programs that do regular expression matching look at data line by line, and thus this particular line spanning pattern would never match anything.

- New ERE Interval Metacharacter (+)

The new + metacharacter matches one or more occurrences of the previous element. For example, the pattern "a+" will match one or more occurrences of the letter "a". The "+" character is equivalent to the ERE expression "{1,}" or the BRE expression "\{1,\}".

- New ERE Interval Metacharacter (?)

The new ? metacharacter matches zero or one occurrences of the previous element. For example, the pattern "a?" will match zero or one occurrences of the letter "a". The "?" character is equivalent to the ERE expression "{0,1}" or the BRE expression "\{0,1\}."

- New ERE Alternation Metacharacter (|).

The new | metacharacter allows the matching of either the element that precedes it or the element that follows it. For example "a|b|c" will match either "a" or "b" or "c". Of course, this particular example is equivalent to the simpler pattern "[abc]". Alternation can also be used on subexpressions or longer patterns, for example, "apple|orange" will match either "apple" or "orange".

- ERE Backreferences

EREs do not support backreferences such as "\(a\)\1" as used in BREs.

File and User Permissions—What's a Root?

The word "root" in AIX (and all UNIX systems), means two things: the root user is the superuser and the root directory is the top of the directory tree.

Root User. Users have a name in the system. You define your user environment, home directory, and user login name. The system comes with one predefined user, the root. The root user has a password also. The root user can read and write into any directory. (See the file attributes section for more information.) The system is usually protected so that only the root user can change system operating characteristics. To change from your user ID to the root, use the

su (superuser) command. To go back to your user ID from the root, press *<Ctrl> d*. If you can't remember which ID you are currently using, type `whoami`.

Root Directory. The top of the file system at "/" is known as the root directory. If you picture a tree upside down, it looks like the file system structure with the "/" as the tree root. Although some system files are at the root, user files should not be stored at the root directory. This tree is a family tree. The directory above your current directory is the "parent" directory. The directories below are child directories. This analogy is also used for process hierarchy. Commands (processes) that invoke another command are "parent" processes with child processes.

User Authorities

There are 3 entities concerning file permission authority:

- owner
- group
- other

Each file has an owner, the creator of the file. Each user belongs to one or more groups that are defined in the system. The system comes with some predefined groups, such as "staff" and "system". The designation of "other" allows anyone the specified access to the files.

Each directory and file have permissions set individually for the file owner, specified group, and other:

- read
- write
- execute

The read permission allows users to look at the contents of a file. Write permission allows modification and deletion of a file, and execution permission allows a program file to execute.

There are additional special bits for advanced users:

- set user id
- set group id
- sticky bit

When a file has the "set user id" attribute, it executes as the owner of the file regardless of who actually typed the command. The "set group id" attribute is sim-

ilar, but for the group instead of the user. The "sticky bit" attribute changes the way the link permission acts; only the owner of the file can delete it. (Traditionally, the sticky bit was used to request that a program stick around in memory after being accessed.)

To view the permissions on a file, go to the Desktop File Manager, click a file, elect `selected`, and `change permissions`, then you will see something like Figure 2–1.

To view the permissions from the command line, type `ls -l` and you'll see something like this:

```
ls -l
total 33
-rw-r--r--    1 carolynj system    3212 Aug 28 21:15 chapt5
-rw-r--r--    1 carolynj system    1773 Aug 28 21:02 chapt6b
-rw-r--r--    1 carolynj system    5337 Aug 20 15:01 chapt8
-rw-r--r--    1 carolynj system    5337 Aug 28 21:02 chapt8b
```

The files all have read and write (`rw`) permission for the owner, carolynj. The middle set of three characters (`r--`) define the read permission of the group "system". The last set of three characters (`r--`) allow read permission for others.

To change the permissions on a file, use the desktop file manager or the `chmod` command. The `umask` command sets the default for newly created files. The `chown` command changes ownership of a file (only the root user can do this).

Figure 2–1 CDE File Manager—Permissions Screen

Process Control—Do I Have Daemons Running?

Each command is at least one process while it is executing. Some processes run in the background where they are invisible to the user. Others run in the foreground where you can see them running. Some processes run forever; they are called "daemons". OK, UNIX can't spell "demon" right, but you can get the idea!

Here is a quick list of actions you can perform on processes:

`ps`	Process Status
`kill`	Terminate the process
`&`	Use "&" after a command to execute it in the background
`<Ctrl> z`	Pause a command (system is waiting on you)
`bg`	Put the command in the BackGround
`fg`	Put the command in the ForeGround

To get a list of the processes running in the system, use the `ps` process status command. If you run `ps` without any parameters, you'll get a list of the processes running for just you, in that terminal, for example:

```
> ps
   PID TTY   TIME CMD
 10868 pts/0 0:00 ksh
 11640 pts/0 0:00 ps
```

You can get a very long list of all the processes running in the foreground and background for all users by running `ps -a` (process status for all). The list is too long to show here and will depend on your system setup. The `ps -l` will show the long status:

```
> ps -l
   F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
200001 A 0 10868 8772 1 60 20 c9e 152     pts/0 0:00 ksh
200001 A 0 11644 10868 9 64 20 857 148    pts/0 0:00 ps
```

There's a lot of information in this read out that you won't usually need, but it's there if you do. I usually use `ps` or `ps -e | pg` to find the process ID, especially the processes that run in the background.

You can kill processes with the `kill` command. You'll need the process identifier (PID). You'll need this command if you have a runaway command you wish you hadn't started. If you execute `kill <pid>`, the system will ask the process to die nicely. If it refuses, you'll have to use stronger measures! The `kill -9 <pid>` command kills the process without giving it a chance to clean up before its end. Use `kill -9` as a last resort.

Advanced users can explore the system resource controller (SRC), which provides commands and library calls to start, stop, and get status on processes. AIX uses the SRC to start many of the daemons at boot time. The actions taken by the system when it first starts up is partially defined by the file `/etc/rc`, other `/etc/rc*` files, and `/etc/inittab`.

I'd Rather Call it Sam—Links

You may have a file that belongs in more than one directory. But it is too much trouble (and it would take up more disk space) to copy it over every time you change the file.

Use a symbolic link and the system will give you one file with multiple names or directory paths. Link from the real file to the link, for example:

```
ln -s memo jan.memo
ln -s memo /u/guest/memo
```

Now there will be three names for one file which all appear and act like three different files in different places. But there is only one real file. How can you tell if the file is unique, or a link?

```
$ ls -l
lrwxrwxrwx 1 carolynj system 4 Aug 28 21:35 jan.memo->memo
-rw-r--r--   1 carolynj system 0 Aug 28 21:34 memo
```

The linked file shows up with an arrow pointing to the real file.

Paper Dolls—Cutting and Pasting

Cutting and pasting is a function that is very useful and easy. You can cut text to and from files or onto the command line itself.

Let's say you are looking at a memo your boss sent you and you would like to take part of it and put it into a different memo. There are several methods you could use. It's as easy as 1, 2, 3:

1. Bring up your favorite editor for the new memo.
2. Cut: Use the mouse to highlight the parts of your boss's memo that you want (just hold down the left mouse button while moving it across the text).
3. Paste: Move your mouse pointer over to the new memo and click with the middle mouse button (or both mouse buttons, if you have only two).

Now you can quote your boss in your new memo, you can add your own text, save the file, print it, whatever.

If you're not a great typist, cutting commands and long file names will save a lot of aggravation. Here's an example:

```
$pwd
/afs/austin/projects/framebook/chapt5
```

Now cut the path name using the left mouse button. Type part of the command you want to execute, paste, type the rest:

```
ln -s <paste-it> <type-the-rest>
```

Where Did My Files Go?

Some users get lost and confused by the hierarchical file structure. There are a lot of system files and directories that come with the system, all nested within each other.

Here is a snapshot of many of the AIX directories that come in the system:

`/bin, /usr/bin, /sbin`	Binary commands
`/lpp, /usr/lpp`	Licensed program product applications
`/etc`	System configuration files
`/home, /u`	User directories (linked directories)
`/dev, /cdrom`	Device system information
`/lib, /usr/lib`	Libraries for program linkage
`/tmp`	Temporary files stored here

If you name your directories in a simple organized manner, it helps. Usually boring file names are better than too much creativity. Use a descriptive name for the contents of the file. I wouldn't use `Hawaii` for the file name of my financial report. Keep track mentally of where you are, or change your command prompt to show what directory you are currently in.

If you use the Korn shell, you can add this to your `.profile` file:

```
export PS1=' $PWD: '
```

The desktop file manager always shows the directory path of where you are (see the chapter on CDE, the Common Desktop Environment).

If you type in `cd` on the command line with nothing else, you will go to your "HOME" directory. This is the directory where your files are stored by default. The path is usually `/home` followed by your login name. There is a "HOME" environment variable stored in your profile (the `.profile` file in your home directory) which tells the system the name of your home.

If you lose track of where you put a file, you can search for it in a couple of ways:

- Using the desktop file manager, select `File` then `Find`

- Using a command, type `find <starting point> -name <filename> -print`

  For example: `find /home/sue -name memo -print`
- This command lists all files and directories under the home directory recursively.

  `ls -R $HOME | more`

Fun Commands

The following commands are all shipped with AIX in the `bos.games` fileset:

`/usr/games/arithmetic`	Quizzes you on simple arithmetic
`/usr/games/back`	The game of backgammon
`/usr/games/bj`	The game of blackjack
`/usr/games/craps`	The game of craps
`/usr/games/fish`	The game of go fish
`/usr/games/fortune`	Displays amusing statements
`/usr/games/hangman`	The game of hangman
`/usr/games/moo`	Guess four-digit number
`/usr/games/number`	Displays numbers you type in English
`/usr/games/quiz`	Quizzes you on a variety of subjects
`/usr/games/ttt`	Tic-Tac-Toe
`/usr/games/wump`	Find the Wumpus!!!!

Well, obviously, you didn't purchase your computer to run these programs. But they can lighten up your day for awhile.

AIX Commands

Now we get into the commands that you can use to get work done from the command line. Commands are very flexible and powerful! The list that follows is printed in the book twice so that you can cut out one of the pages and put it next to your computer or in a handy place, as a cheatsheet.

You can get your work done using the desktop functions, but there is a lot more you can do from the command line. If you want to be more of a guru, learn some commands. We grouped similar commands together and capitalized letters in the description that make up the command to help you remember. We also added some of our favorite command combinations that may come in handy!

Commands have two parts: the action and the parameters. UNIX systems care about upper and lowercase. If you type CP, it's not the same as cp, theoretically there could be two separate commands. Parameters are usually prefaced by a dash "-" and are one character long. In the example that follows, the parameters are shown in brackets "[ ]". Some commands require file names or other user specified information, such as the cp command to copy from the infile to the outfile.

```
cp [-p -r –]... infile [indir] outfile [outdir]
```

Directory Commands

- `mkdir` MaKe a new DIRectory
- `cd` Change to new Directory
- `pwd` Print Working Directory (where am I?)
- `rmdir` ReMove DIRectory
- `mvdir` MoVe DIRectory under different directory or rename
- `ls, li` LiSt, LIst the files in a directory
- `.` Abbreviation for the current directory
- `..` Abbreviation for the parent directory
- `df` Disk Free space status
- `whereis` Show directories WHERE command IS stored

File Commands

- `cp` CoPy a file
- `mv` MoVe a file to a different directory or rename it
- `rm` ReMove a file
- `ln` LiNk a filename to another file
- `cat` Show file contents, conCATenate a file
- `pg` Show contents of a file, one PaGe at a time
- `more` Show MORE file contents
- `enq` ENQueue a file to print
- `chmod` CHange read / write / execute MODe of file
- `cmp` CoMPare binary files
- `diff` Show DIFFerence in text files
- `grep` Search for text in a file
- `join` JOIN two files based on first column
- `sort` SORT contents of a file

Backup Commands

- `compress` COMPRESS files to take up less backup space
- `uncompress` UNCOMPRESS the compressed file
- `cpio` CoPy Input / Output
- `tar` Tape ARchiver
- `pax` Portable Archive interchange(X)
- `dd` Device to Device copy
- `backup` BACKUP files (use with restore command)
- `restore` RESTORE files from backup format
- `mksysb` MAKe SYStem Backup for rootvg volume group

Miscellaneous Commands

- `crontab` Set commands to run at a specified time / date (CRONological TABle)
- `clear` CLEAR the screen
- `lpstat` Check Line Printer STATus
- `date` Current DATE and time
- `hostname` Local system HOST NAME
- `whoami` Current login name
- `su` Change to another user login (Default Super User)
- `wall message` Broadcast a message to all logged-on users (Warn ALL)
- `banner "text"` Screen BANNER for text

Help Commands

- `<cmd> -?` List parameters for command
- `<cmd> -help` Short help for command
- `man <cmd>` On-line MANual pages for command
- `info` INFOrmation from the on-line help
- `learn` Lessons for a subject or command

Handy Commands

- `ls -lRs` Long, recursive sorted file list (sorted by modification date)
- `echo "text" > file` Create a file with one line in it
- `echo "End" >> file` Puts a final line on the file
- `ls | more` List files and pause at each page
- `sed "s/|/ /g" file` Change all pipe symbols to blanks
- `aixterm -fn Rom22 &` Use very big font for command line
- `sort file | uniq | more` Show unique sorted list
- `mail user@location < file` Send a file as mail
- `enscript -q -p - -r -f Courier8` Print with small font, landscape (to postscript printer)
- `fgrep <string> file | cut -c1-80 > newfile` Filter out lines without the string, cut all but columns 1-80, put in `newfile`
- `tee <file> <cmd>` Write command output to two places, standard out (typically the screen) and a file
- `wc -l file` Count number of lines in file
- `du | sort -rn | pg` Disk usage sorted for largest first
- `sdiff file1|file2` Compare two files side by side
- `for i in *.txt; do lpr $i; done` Print all files in current directory that end in `.txt`.
- `find . -print |xargs grep pattern` Find all files that contain pattern in the current directory hierarchy

Directory Commands

- `mkdir` MaKe a new DIRectory
- `cd` Change to new Directory
- `pwd` Print Working Directory (where am I?)
- `rmdir` Remove Directory
- `mvdir` MoVe DIRectory under different directory or rename
- `ls, li` LIst, LiSt the files in a directory
- `.` Abbreviation for the current directory
- `..` Abbreviation for the parent directory
- `df` Disk Free space status
- `whereis` Show directories WHERE command IS stored

File Commands

- `cp` CoPy a file
- `mv` MoVe a file to a different directory or rename it
- `rm` ReMove a file
- `ln` LiNk a filename to another file
- `cat` Show file contents, conCATenate a file
- `pg` Show contents of a file, one PaGe at a time
- `more` Show MORE file contents
- `enq` ENQueue a file to print
- `chmod` CHange read / write / execute MODe of file
- `cmp` CoMPare binary files
- `diff` Show DIFFerence in text files
- `grep` Search for text in a file
- `join` JOIN two files based on first column
- `sort` SORT contents of a file

Backup Commands

- `compress` COMPRESS files to take up less backup space
- `uncompress` UNCOMPRESS the compressed file
- `cpio` CoPy Input / Output
- `tar` Tape ARchiver
- `pax` Portable Archive interchange(X)
- `dd` Device to Device copy
- `backup` BACKUP files (use with restore command)
- `restore` RESTORE files from backup format
- `mksysb` MAKe SYStem Backup for rootvg volume group

Miscellaneous Commands

- `crontab` Set commands to run at a specified time / date (CRONological TABle)
- `clear` CLEAR the screen
- `lpstat` Check Line Printer STATus
- `date` Current DATE and time
- `hostname` Local system HOST NAME
- `whoami` Current login name
- `su` Change to another user login (Default Super User)
- `wall message` Broadcast a message to all logged-on users (Warn ALL)
- `banner "text"` Screen BANNER for text

Help Commands

- `<cmd> -?` List parameters for command
- `<cmd> -help` Short help for command
- `man <cmd>` On-line MANual pages for command
- `info` INFOrmation from the on-line help
- `learn` Lessons for a subject or command

Handy Commands

- `ls -lRs` Long, recursive sorted file list (sorted by modification date)

  `echo "text" > file` Create a file with one line in it
- `echo "End" >> file` Puts a final line on the file
- `ls | more` List files and pause at each page
- `sed "s/|/ /g" file` Change all pipe symbols to blanks
- `aixterm -fn Rom22 &` Use very big font for command line
- `sort file | uniq | more` Show unique sorted list
- `mail user@location < file` Send a file as mail
- `enscript -q -p - -r -f Courier8 <file>` Print with small font, landscape (to a postscript printer)
- `fgrep <string> file | cut -c1-80 > newfile` Filter out lines without the string, cut all but columns 1-80, put in `newfile`
- `tee <file> <cmd>` Write command output to two places, standard out (typically the screen) and a file
- `wc -l file` Count number of lines in file
- `du | sort -rn | pg` Disk usage sorted for largest first
- `sdiff file1|file2` Compare two files side by side
- `for i in *.txt; do lpr $i; done` Print all files in current directory that end in `.txt`.
- `find . -print |xargs grep pattern` Find all files that contain pattern in the current directory hierarchy

AIX Command Details

You'll need details on some of the AIX commands. We've included some of the commonly used commands, but not all the commands, because there are so many. See the appendix for more details. There are whole reference books for all the commands and libraries that you can get if you're a real guru, guru wanna-be, or writing your own programs.

This list of reference books will get you started:

RISC System/6000 Technology

Elements of AIX Security

Printing for Fun and Profit Under AIX

AIX Performance Monitoring and Tuning Guide

AIXtra Magazine

AIXpert Magazine

Life with Unix, Ressler Libes

The Design of the Unix Operating System, Back, Prentice-Hall

Unix in a Nutshell, Billy, O'Reilly & Associates

Unix Power Tools, Peek, Loukides, O'Reilly

POSIX Programmer's `Guide`, Lewine

Practical C Programming, Oualline

Using C on the UNIX System, Curry

UNIX for FORTRAN Programmers, Loukides

Software Portability with imake, DuBois

AIX Operating System Technical Reference