sun® microsystems

Doing More with SunOS™: Beginner's Guide

Part Number: 800-1710-10
Revision A, of 9 May 1988

sun®
microsystems

Doing More with SunOS™: Beginner's Guide

Part Number: 800-1710-10
Revision A, of 9 May 1988

Contents

Tables

Figures

Preface

This manual describes some of the more sophisticated features SunOS provides, and how to use them to simplify complicated tasks.

Chapter 1 is a brief introduction.

Chapter 2 provides details about files, their attributes, filename substitution, and searching through text files.

Chapter 3 describes how to use commands as building blocks for complicated tasks.

Chapter 4 provides an overview of the C shell and its timesaving features.

Chapter 5 describes processes and their behind-the-scenes role in providing balanced service to concurrent tasks.

Chapter 6 introduces tools for sophisticated file management.

Chapter 7 describes the printer queue, how to select a printer, printing preformatted files, and printing graphics from the workstation screen.

In addition to a glossary, command summary, and quick reference, there are appendices that describe details about the C shell, such as special characters and scripts.

Prerequisite Documents

Getting Started with SunOS: Beginner's Guide
Mail and Messages: Beginner's Guide
Using the Network: Beginner's Guide
Setting Up Your SunOS Environment: Beginner's Guide

If you are using SunView, the Sun windows system, you should also read this manual first:

SunView 1 Beginner's Guide

Companion Documents

Self-Help with Problems: Beginner's Guide
SunOS Reference Manual

For Sun386i users, this manual augments the material in the *Sun386i Advanced Skills* manual.

1

Introduction

1

Introduction

SunOS provides you with features that are powerful, flexible, and adaptable. This means that there is quite a lot that the system can do for you, and there is quite a lot to learn. The power and richness of the commands make for limitless possibilities. In fact, one of the main advantages of the SunOS system design is its open-ended nature.

Everyone goes through several stages when learning to use SunOS effectively, including:

a) learning the basics

You are here. ⇒

b) learning enough to get curious

c) experimenting with the various features and commands

d) educated experimentation and writing simple shell scripts

e) digging deeper into the system and its internal workings.

This manual is intended to help satisfy your curiosity with an overview of features that give you major productivity gains.

Previous manuals in this series, such as *Getting Started with SunOS: Beginner's Guide Setting Up Your SunOS Environment: Beginner's Guide* and *Mail and Messages: Beginner's Guide* gave you a basic familiarity with SunOS, but may not have answered questions about *why* the system works the way it does, or *how* to get more out of it. Hopefully, this one does.

Companion manuals, such as *Using the Network: Beginner's Guide* will tell you about more specialized topics.

Why and How

SunOS is based on the UNIX operating system developed at Bell Laboratories; it is an enhanced version, incorporating many of the additions developed at the University of California, Berkeley.

From its origins as a simple research project, the UNIX system evolved into a powerful, flexible and popular computer operating system, and a major influence in the industry. It was designed to accommodate this evolution by providing a simple model for storing and transferring information, called a *file*, a collection of simple commands to operate on files, and a straightforward method for combining commands to perform more complicated tasks. Because the UNIX system grew out of a computer science research environment, the terminology and command names are oriented toward professionals in that field, as are many of the tools.

Commands are terse to save keystrokes. They are usually suggestive of the simple function they perform. Unless you are already familiar with those sorts of functions, the names may seem cryptic. The more you learn, the more sensible things will begin to seem. So, rather than being put off by it, get familiar with the jargon! You'll learn a lot more about computers than just how to use one.

Try it Yourself!

When learning more about SunOS, there is no substitute for experimenting on your own. To really grasp what a command does, you simply have to try it. So, as you go through this, and the remaining beginner's guides in this series, try out the examples. Then try out variations of your own design.

Play it Safe!

Whenever you experiment with SunOS it is important to set up a safe place in which to do so. Never experiment with an unfamiliar command on valuable data. Instead, make a copy and place it in a directory where the data is known to be dispensable. Always run your tests in this directory to avoid the risk of corrupting previous work. Once you have tested the command and have seen what it does, only then should you apply it to files that you care about.

Make a directory, `test`, in your home directory, as follows:

```
venus% cd
venus% mkdir test
```

Consider everything in this directory to be expendable, and never place anything there that you intend to keep.

Hang in There!

Because the UNIX system was developed to support programming research, many of SunOS's standard features are oriented toward the programming professional. This is one reason why the system is so powerful, and also why some features seem a bit abstract at first. In most cases, their power and flexibility make this an easy thing to get used to.

SunOS is designed to be general in scope. It can support a wide variety of applications, and work well within a broad range of situations. The information in this manual should help you to take this general and flexible, but somewhat abstract system, and use it to meet your specific needs and working style.

2

More About Files

2

More About Files

2.1. Filename Substitution

As you learned in *Getting Started with SunOS: Beginner's Guide* filename wild cards can save you time and keystrokes. The system replaces, or *substitutes* characters from filenames for the wild card symbols.

In addition to the wild cards, `*`, and `?`, SunOS provides more sophisticated ways of specifying a set of files on the command line.

Single-Character Matching with `[` and `]`

You can use *brackets* instead of a `?`, to match a single character. Within the brackets you can specify a list of characters to match against. For instance,

```
[ab]*
```

matches all filenames that begin with a lower-case `a` or `b`. You can also specify a *range* of characters to match against. Thus,

```
[A-Z]*
```

matches all filenames that begin with an upper-case alphabetical character.

Listing Hidden Files with `ls -a`

Filenames that begin with a dot (`.`) are a special case. They aren't matched unless you specify a dot in the first character. However, the name `.` stands for the current directory, and `..` stands for the parent directory. So, although the command

```
ls .*
```

does list hidden files, it *also* lists all the other files in the directory (matching `./*`), and the parent directory (matching `../*`).[1]

To list hidden files along with the others, use the command:

```
ls -a
```

[1] To make matters worse, it also lists the contents of any directory, in the current directory, which starts with a dot.

String Matching with { and }

You can use *braces* instead of `*`, to match specific character strings of any length. Within the braces, strings are separated by commas. For instance,

```
{uranus,sygnus,x}*
```

matches any filenames beginning with `uranus`, `sygnus` or `x`.

Within braces, `*`, and `?`, are legal. You can nest braces within strings for interesting results. For instance, `{{ura,syg}nus,x}*` is another way to match filenames beginning with `uranus`, `sygnus` or `x`.

2.2. Properties of Files

As your skill with the system grows, you will encounter situations in which a prior understanding of files and their properties, especially file *ownership* and *permissions*, will be of immense help.

You can think of a file as a named location from which information can be obtained or to which data can be sent. SunOS uses the notion of a file as a general model for all sources (input) or destinations (output) of data operated on by commands. The system treats terminals, printers, tape drives, and other such devices for putting information into, or getting information out of the system, as if they too were *files*.

Commands and programs don't need to know whether the data they use comes from (or goes to) a terminal, disk file, printer (or even another program). Just like any other file, each device has a pathname. The `tty` command tells you the pathname of your terminal or window.

Figure 2-1 *The* `tty` *Command*

```
venus% tty
/dev/ttyp1
venus%
```

In addition to having a *name*, and *contents*, a file under SunOS has other important properties that you can examine with options to `ls`. (Refer to `ls` in the *SunOS Reference Manual* for a complete list of these options.) The `-l` options shows a more detailed (long) list of the files:

Figure 2-2 *The* `ls -l` Command

```
venus% ls -l
total 112
-rw-rw-r--  1 sam        77293 Jun 27 15:36 csh.1
-rw-rw-r--  1 wild       27492 Jul  9 21:14 csh.blt
-rw-rw-r--  1 ames        6550 Jul  9 21:02 csh.new
-rw-rw-r--  1 root       14492 Jul 12 17:07 csh.spc
-rw-r--r--  1 sam         2884 Jul 17 18:24 files
-r-xr-xr-x  1 sam         1381 Jul 12 15:50 script
venus%
```

The top line tells you how many blocks (units of space on the disk), are occupied by files in the directory. The remaining lines are composed of columns that describe specific properties of each file:

Figure 2-3 *Information Displayed By* `ls -l`

```
permissions      links owner size   modification time   filename
-rw-rw-r--         1   sam  77293   Jun 27 15:36        csh.1
```

The leftmost column shows the *permissions* for each file. Permissions are explained in detail below. The second column shows the number of *links*, to it. Links are also described later on.

The third column shows each file's *owner*. Normally, the owner of a file is the person who created it, although the operator of your system can change this. Not shown here is the file's *group* ownership.

The fourth column shows the file's *size* in bytes. The size of the file often changes when you edit it.

The next three columns show the date and time when the file was last modified (*modification time*). This also changes whenever you edit the file. If a file hasn't been modified in six months, they display the year and date instead.

The rightmost column shows the *filename*.

2.3. Permissions

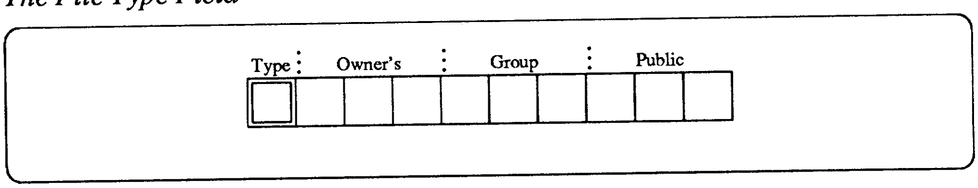
Every file has a set of access modes or *permissions* that determine which users have access to read, write, or *execute* its contents.

Like devices, programs are treated as files. When you enter a command, SunOS looks up a file by that name among the directories listed in the PATH environment variable, and performs the instructions contained in that file.

The *permissions* column consists of ten characters as shown in Figure 2-3, above. The leftmost character shows the type of file (regular, directory or device). The next triplet of characters displays access modes for the owner. The second triplet shows those for the group, and the last, those for the public.

File Type

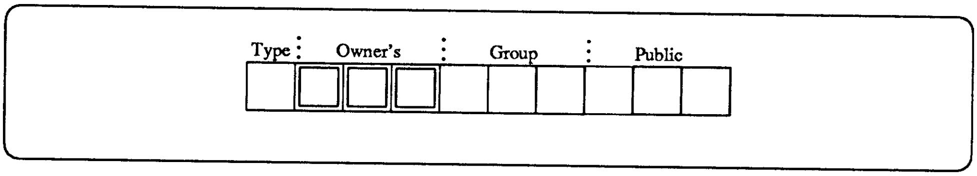
Figure 2-4 *The File Type Field*



A d in the leftmost character indicates that the file is a directory. A – indicates a standard file. A b, or c indicates that the file is a *device*. An s, indicates that the file is a *socket* for communication between two running programs. An l indicates that the filename is a *symbolic link* that refers to the name of another file.

Owner's Permissions

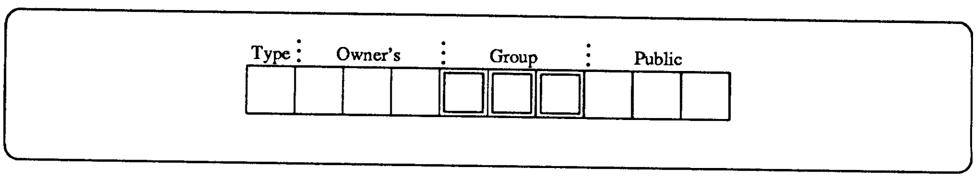
Figure 2-5 *Owner's Permissions Field*



In the listing of Figure 2-3, sam is the owner of the file csh.1. An r as the first character in this triplet indicates that the owner has permission to read the file. A – indicates that the permission does not apply. A w as the second character indicates that the owner can write on (modify, add to, or remove) the file. An x as the third character indicates that the owner can execute the file (use it as if it were a command[2]). As Figure 2-3 shows, sam can read and write on, but not execute the file csh.1.

You can change the *access* privileges for a file with the command chmod (described further on). However, only a system administrator can change the file's *ownership* (with the chown command.)

Figure 2-6 *Group Permissions Field*



To see which *group* the file belongs to, use the –lg option of ls.

[2] Of course, unless the file is either a program or list of shell commands, executing it doesn't make any sense.

Figure 2-7 *The* `ls -lg` *Command*

```
venus% ls -lg
total 112
-rw-rw-r--  1 sam    wheel   77293 Jun 27 15:36 csh.1
-rw-rw-r--  1 wild   wheel   27492 Jul  9 21:14 csh.blt
-rw-rw-r--  1 ames   wheel    6550 Jul  9 21:02 csh.new
-rw-rw-r--  1 root   wheel   14492 Jul 12 17:07 csh.spc
-rw-r--r--  1 sam    wheel    2884 Jul 17 18:24 files
-r-xr-xr-x  1 sam    wheel    1381 Jul 12 15:50 script
venus%
```

In this case, all files belong to the group `wheel`. The files `csh.1` through `csh.spc` can be read and written on by any member of the group. The file `script` can be executed and read, but not written on.

You can change the group ownership of a file or directory by using `chgrp`; you must be a member of the group to which you're reassigning it (and you have to own the file). For more information on `chgrp`, see the *SunOS Reference Manual* or type `man chgrp`.

How do you know which group or groups you belong to? Simple—with the `groups` command, as follows:

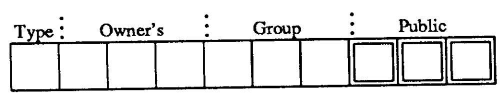
Figure 2-8 *The* `groups` *Command*

```
venus%  groups
wheel        staff
venus%
```

This tells you that you belong to the groups *wheel* and *staff*.

Public Permissions

Figure 2-9 *Public Permissions Field*



All files in the above list can be read by anyone. The `x` in the rightmost character for `script` indicates that anyone can use it as a command.

Permissions of Directories

With directories, the access modes have a slightly different meaning. To check the permissions of the current directory, use the `-ld` option of `ls`.

Figure 2-10 *Checking Directory Permissions*

```
venus% ls -ld
drwxrwxr-x  3 sam          512 Jul 16 23:10 .
venus%
```

An `r` indicates that the directory can be *read*. You must have read access to a directory before you can list its contents.

A `w` indicates that files can be added or removed from the directory.

An `x` indicates that the directory can be *searched* (that you can list its contents). The directory must have search permissions turned on for you to `cd` into it, or for you to add or delete files, or even to list its contents. (This is because if the system can't search the directory's contents, it can't know what's there to retrieve, or overwrite, etc.)

You can remove any file in a directory for which you have write permission, regardless of who owns that file.[3] If you do not have write permission for the file itself, the system asks you for confirmation before removing it.

In the directory shown above, the owner (`sam`) can read, search, and add or delete files, as can the group. The public can read and search, but cannot add or delete files.

2.4. Changing Permissions with `chmod`

From time to time you may want to change the access modes of files that you own, either to restrict or to allow access to it. In most cases, restricting access to a file is sufficient to protect it from tampering or unwarranted reading. Even so, you should be aware that the operator of your system has unlimited access to any file. Because SunOS evolved in a relatively friendly research-and-development setting, the file system provides adequate, but not unbreakable, security between users.[4]

You can use an argument to `chmod` to specify the access mode for each class of user (owner, group, or public), or to indicate how the mode is to be changed. An argument is composed of one or more classes, an operation, and one or more permissions from the chart below:

[3] The exception to this is for directories with the "sticky bit" set; then only the superuser or the file's owner can remove it.

[4] No computer system provides unbreakable security between authorized users. Also note that the system administrator can read any file on the system. If you want to protect your files from unauthorized reading, you can *encrypt* them. See Section 2.10 below, for details.

Table 2-1 chmod *Command Syntax Diagram*

chmod [*class(es)*] *operation permission(s)* [, ...] *filename* ...

where *class(es)*, *operation* and *permission(s)* can be selected from:

class		*operation*		*permission*	
u	user (owner)	=	set permission	r	read
g	group	–	remove access	w	write
o	others (public)	+	give access	x	execute
a	all				

For example, the command

Figure 2-11 *Using the* chmod Command

```
venus% chmod o-r,a+x,g=rw csh.1
venus%
```

a) removes read permission for the public (others),

b) adds execute permission for all three classes, and

c) sets access to read and write for the group

for the file csh.1.

If you omit *class*, the new setting is applied to all three.

chmod can also use a digit from zero to seven to represent each triplet in the permissions column, as follows:

chmod [*o*[*g*]]*p*

where *o* is a digit representing the owner's permissions, *g* is a digit representing the group permissions, and *p* is a digit representing permissions for the public. The value of each digit is the sum of the permission values as in the following chart.

Table 2-2 *Chart of* chmod *Numeric Arguments*

value	*permission*	*explanation*
4	r	read
2	w	write
1	x	execute

To figure each digit, add up the values corresponding to each permission setting in the triplet. For read, write and execute permission, the value is 7. All values, and the permissions they correspond to, are shown below:

Table 2-3 *Values and Permissions*

value	*permissions*	*explanation*
7	`rwx`	read, write, and execute
6	`rw-`	read and write
5	`r-x`	read and execute
4	`r--`	read only
3	`-wx`	write and execute
2	`-w-`	write only
1	`--x`	execute only
0	`---`	no access whatsoever

The command

Figure 2-12 *Giving Open Permissions to Everybody*

```
venus% chmod 777 csh.1
venus%
```

gives read, write and execute access to `csh.1` to the owner, the group, and the public.

On the other hand, the command

Figure 2-13 *An Unlikely Permission Setting*

```
venus% chmod 7 csh.1
venus%
```

gives the public read and write access, and denies all access to the owner and the group. So, although they aren't required, it's a good idea always to use all three digits.[5]

2.5. Setting Default Permissions with `umask`

When you create a new file or directory, the system automatically assigns permissions. The default setting for new files is

```
-rw-r--r--
```

or `644`. For new directories, the default is

```
drwxr-xr-x
```

or `755`.

You can change the default permission setting for the current session with the `umask` command:

[5] There is also a fourth digit, one that is used to allow certain programs to assume another user ID or group ID while running, or to to remain in memory even when stopped. Unless you are writing programs like that, you will have little occasion to use the fourth digit.

```
umask [o[g]]p
```

o, *g* and *p* are digits corresponding to the owner's, group, and public permission masks, respectively.

You can change the permissions for all sessions by placing a `umask` command in your `.cshrc` file.

Like `chmod`, `umask` uses three digits to determine the permissions. Unlike `chmod`, it computes the permissions according to the following table:

Table 2-4 *Values and Permissions for New Files*

Files		*Directories*	
value	*permissions*	*value*	*permissions*
0	`rw-`	0	`rwx`
1	`rw-`	1	`rw-`
2	`r--`	2	`r-x`
3	`r--`	3	`r--`
4	`-w-`	4	`-wx`
5	`-w-`	5	`-w-`
6	`---`	6	`--x`
7	`---`	7	`---`

`umask` does not activate *execute* permission for files.

So, the command

```
umask 2
```

or

```
umask 002
```

yields permissions of `-rw-rw-r--` for files, and `drwxrwxr-x` for directories.

The command

```
umask 22
```

yields permissions of `-rw-r--r--` for files and `drwxr-xr-x` for directories.

2.6. Ownership

Only the owner[6] of a file can change its permissions. To find out how to change the ownership or group ownership of files, refer to *Using the Network: Beginner's Guide*.

2.7. Modification Time

The modification time indicates the most recent time that the file has been edited, or appended to. You can change a file's modification time, without affecting its contents, with the `touch` command.

`touch` *filename*

Touch does not alter the contents of *filename*, but rather, resets the modification time to the current date and time. If the file does not exist already, `touch`

[6] or the *superuser*, described in Section 5.4.

creates it. `touch` is useful when you want to create empty files (say, for a test) or when you want to update a file when using `Make`.

2.8. Making Links

A *link* is a name associated with a file. SunOS allows several links to a file at any one time, so the same file can have more than one name. This is useful when you want to get at a file quickly from within different directories. Moreover, you can keep a link to a file in a restricted directory, thus allowing people access to the file without giving them access to the forbidden directory. When you create a file, the system makes the first link, or filename, for you. To make an additional link, use the `ln` command.

`ln` *oldname newname*

If you attempt to make a link to a file in a directory that is on a different disk or disk partition than that of *oldname*, you will get an error message of the form:

newname`: Cross-device link`

In this case, you can use the `-s` option of `ln` to make a *symbolic* link to the file.

`ln -s` *oldname newname*

A symbolic link is an entry in the directory that points to the *name* of another file, rather than the file itself. A symbolic link can be made across devices, and can be made even when *oldname* does not exist. Because a symbolic link refers to another file's name, rather than the file itself, it may be to your advantage to use a symbolic link instead of a regular link when you want to specify an alternate pathname to the same file.

Both regular (hard) and symbolic links allow you to use *newname* instead of *oldname* to gain permitted access to a file. But, neither a regular (hard) link nor a symbolic link changes the ownership, group, or permissions of a file. So, although you can make a link to a file that you can't read, you still won't be able to read its contents, whichever name you use.

2.9. Seeing File Types with `ls -F`

The `-F` option of `ls` appends a character to the end of each filename to indicate what type of file it is, as follows:

Table 2-5 `ls -F` *File Type Indicators*

tag	*type of File*
(*none*)	normal file
/	directory
*	execute access allowed
@	symbolic link

You may find it useful to place an alias in your `.cshrc` so that `ls` is replaced with `ls -F`:

```
alias ls 'ls -F'
```

2.10. Encrypting Files

You can use `crypt`[7] to encode the contents of confidential files. To encode a file named `secret.plans`, use the following command:

Figure 2-14 *Making a File Secret*

```
venus% crypt < secret.plans > crypt.plans
```

The *angle brackets* are required. The > should be familiar to you. The < is explained in Chapter 3.

Remember to remove the unencrypted version, or your secrets may not keep!

`crypt` then asks you for an encryption *key*. A key is some memorable, but unlikely, word, the longer and odder the better. This key is necessary for `crypt` to do its work, and like your password, you must remember it if you want to read your file once again.

```
Key:
```

You can also use `crypt` to decode a file:

Figure 2-15 *Decoding a file*

```
venus% crypt < crypt.plans > decoy.plans
Key:
```

`decoy.plans` will contain the text you started out with.

If you want to look at the decoded contents, a command of the form:

`crypt <` *cryptfile* `| more`

will, after asking for the key, display them on the screen.

You can edit the contents of an encrypted file using the `-x` option of `vi`.

Figure 2-16 *Using* `vi` on and Encrypted File

```
venus% vi -x crypt.plans
Enter key:
```

Whenever you issue the `w`, or *write*, command, `vi` runs the file through `crypt`.

2.11. Searching Through a File with `more`

There are times when you need to look up something in a long file, but `grep` won't do because you need to see a whole paragraph or screenfull of information, rather than just one line. If the file is very long, stepping through it a screenfull at a time with `more` may take too much time. So, `more` allows you to search for a string within a file. Instead of typing a [SPACE] to see the next page, or a [Return] to see the next line, you can type in a slash (`/`), followed by a *string*, and `more` will skip ahead to a screenfull containing *string*.

[7] SunOS encryption facilities are only available to customers within the United States of America.

Figure 2-17 *Using* `more`

```
venus% more decoy.plans
...
...more 5%...
/picnic
Skipping ...
...
up to the cabin, where we will
have a picnic lunch.
Afterward we could take a swim, and then sip
some sangria.
...
...more 85%...
```

To skip to the next occurrence of that same string, use `n`.

When using `more` to look at several files, the command `:n` will skip to the next file.

2.12. Using `pushd`, `popd` and `dirs` to Change Directories

Sometimes, when you are traveling through a variety of directories, you may find that you want to backtrack. Of course, `cd`, doesn't remember where you've been. So, unless *you* do, backtracking can be painful. `pushd`, `popd` and `dirs` allow you to stack up a list of directories to revisit.[8] When you are in a directory you'll want to return to, type

`pushd` *directory*

where *directory* is the name of the directory you want to switch to. (Unlike `cd`, you must always specify a destination *directory*, even when changing to your home directory.) `pushd` changes to the new *directory*, while keeping track of the directory you changed from and to.

If you want to jump back to a previous directory, type

`popd`

to work your way back.

If it's been a while since you last did a `pushd` or `popd`, and you want to see the list of directories you've stacked up, the

`dirs`

command will show it to you. (Note that `pushd` and `popd` will also display the directory stack, with the current directory at the left.)

[8] These commands only work with the C shell. Refer to Chapter 4, The C Shell, for more information.

Figure 2-18 pushd, popd, *and* dirs

```
venus% pwd
/var/spool/mail
venus% pushd ~
~ /var/spool/mail
venus% pushd /etc
/etc ~ /var/spool/mail
venus% dirs
/etc ~ /var/spool/mail
venus% popd
~ /var/spool/mail
```

dirs, with the -l option, displays the full pathnames of stacked directories:

Figure 2-19 dirs with Full Pathnames

```
venus% dirs -l
/home/medici/cosimo /var/spool/mail
```

3

More About Commands

3

More About Commands

3.1. Redirecting Output, Redirecting Input, and Pipes

Commands perform actions, typically on data contained in a file. Unless you indicate otherwise, they normally display their results on the terminal screen. The terminal is known as the command's *standard output.*

Because SunOS commands treat files and devices in a uniform way, you can direct the output of a command to any file or device that you choose. You can also use the output of one command as direct input to another, using a special connection symbol called a *pipe.*

Unless you indicate otherwise, commands normally operate on data as you type it in from the keyboard; so the terminal is known as the command's *standard input.*

Redirecting Output

As you learned in *Getting Started with SunOS: Beginner's Guide*, a right *angle-bracket* (>)[9] on the command line indicates that the next word is the name of a file or device in which to place, or *redirect* the output of a command. For instance, the command line:

Figure 3-1 *Redirecting Output*

```
venus% ls -la > list
```

places the output of the `ls -la` command (a detailed list of all files, including hidden files) in a file named `list`.

CAUTION **If a file by that name already exists, any previous contents are deleted *before* the command is performed.**

So, the command

```
cat will.be.empty > will.be.empty
```

removes all existing contents from the file `will.be.empty` before the `cat` command is executed.

To avoid writing over existing files, add a line with the command

[9] may be pronounced as "into"

```
set noclobber
```

to your `.cshrc` file if one isn't there already.[10] Then type in the command:[11]

Figure 3-2 *The* `source` *Command*

```
venus% source .cshrc
venus%
```

When you are certain that you want to overwrite the previous contents of a file, using a `>!` overrides this file protection.

You can *append*, or 'add to the end of' a file using a *double-right-angle-bracket* (>>).[12] Thus, the command[13]

Figure 3-3 *Appending to an Existing File*

```
venus% ls >> list
```

adds a second version of output from `ls` (containing just the names of nonhidden files) onto the end of `list`.

Redirecting Input

Just as you can redirect the output of a command, you can also specify a file (or device) from which that command obtains its *input*.

You can use a *left* angle-bracket (<)[14] to redirect the standard input of a command. For instance, the following command prints the contents of the file `list`.

[10] Refer to *Setting Up Your SunOS Environment: Beginner's Guide* for more information about this file.

[11] If using windows, type this `source` command in each `shelltool` or `cmdtool` window, so that the change will take effect in the C shell running within each.

[12] may be pronounced as "onto"

[13] With `noclobber` set, a file must already exist before the standard output can be appended to it. Using a `>>!` overrides this.

[14] may be pronounced as "from"

Figure 3-4 *Redirecting Standard Input*

```
venus% cat < list
drwxr-xr-x  3 sam     512 Jul 29 23:11 ./
drwxrwxrwx  4 sam     512 Jul 19 12:17 ../
drwxrwxrwx  2 sam     512 Jul 26 18:52 SCCS/
-rw-r--r--  1 sam   77293 Jun 27 15:36 csh.1
-r--r--r--  1 sam   21773 Jul 24 16:43 files
-rw-r--r--  1 sam       0 Jul 29 23:11 list
lrwxrwxrwx  1 sam       8 Jul  8 16:40 outline -> ../wwu.b
-rw-r--r--  1 sam    3557 Jul 12 18:59 philos
-rw-r--r--  1 sam      82 Jul 24 16:43 pic.src
-r--r--r--  1 sam    1381 Jul 12 15:50 preface
SCCS/
csh.1
files
list
outline@
philos
pic.src
preface
```

Most commands allow the input file to be specified as an argument. You could, for example, produce the same display with the command:

Figure 3-5 *Another Way to Display a File*

```
venus% cat list
```

However, other commands, such as `crypt`, only read from the standard input, and thus require use of <, the input redirection symbol.

Pipes and Pipelines

The output of one command can be fed in directly as input to another. A set of commands strung together in this way is called a *pipeline*, and the symbol for this input/output (I/O) connection is a vertical bar (|),[15] called a *pipe*. Pipes and pipelines have a wide variety of uses.

For example, suppose you wanted only to list symbolic links in the directory. You can combine `ls` and `grep` to get the result you want. The pipeline

```
ls -l | grep lrwx
```

will do the trick. But it will also list any files with the unlikely combination *lrwx* in their filenames, so, just to be sure, you might want to try this pipeline:

A less efficient way to accomplish would be to use a temporary file:

```
ls -l >filename
grep lrwx <filename
rm filename
```

```
ls -F | grep @
```

(The `ls -F` command shows links with the `@` symbol.)

[15] may be pronounced as "through"

There is no filename following `grep` because the pipe symbol indicates that `grep` is to search through its standard input, which in this case is the output of `ls`.

You can connect several commands to make longer pipelines. For instance, the command line:

Figure 3-6 *Using Pipes*

```
venus% ls -l | grep lrwx | wc
       1      10      65
```

uses `wc` (word count) to display the number of lines, words, and characters, respectively, in the list of symbolic links culled from the output of `ls` by `grep`. Since `wc` received only one line from `grep`, there was only one symbolic link in the directory.

The ability to 'cook up' intricate commands on the spot is a very special feature of the SunOS system, and one that becomes increasingly useful as you continue to experiment and learn.

Filters

Commands like `grep` are called *filters*. They accept text as input, transform it in a straightforward way, and produce text as output. Although often used as commands in their own right, filters are especially useful in pipelines.

`ls` is not a filter, because it doesn't accept data from the standard input. Neither is `date`. As you might expect, the command `ls | date` produces *only* the date, since `date` ignores its standard input. What does `date | ls` produce?

`more` is another type of filter. It transforms the data by breaking it up into screen-sized chunks. Some other interesting filters are:

`head` -*n* displays the first *n* lines of a file. With no -*n* argument, it displays the first ten lines.

`tail` -*n* displays the last *n* lines. With no -*n* argument, it displays the last ten.

`tail` +*n* skips to line *n* and displays that line through the end of the file.

`more` +/*pattern*
like `tail`, this command begins printing two lines before the first match for *pattern*, which can be either a string or a `grep` search pattern (described below under `grep` *and* `grep` *Search Patterns*).

`cat -v` translates nonprinting characters into strings of regular characters of the form ^*c* (for control characters), or `M-`*c* (for 8-bit characters).

`sort` display the line in alphanumeric order, or according to an order you specify. Refer to `sort` in the *SunOS Reference Manual* for more information.

`sort -n` sort in numerical order.

`fmt` does rudimentary formatting of text.

`rev` reverses the order of characters within each line.

`pr -t -n` breaks up the output into *n* columns. The `-t` option suppresses a heading that would otherwise appear.

`spell` produces a list of possibly-misspelled words.

The command

`look` *string*

looks up words (in the system dictionary) whose leftmost characters match *string*. The command

`look a`

will display all words starting with `a`. To further restrict the search, add more characters.

`sed` performs simple edits on a line-by-line basis. For instance, the alias:

```
alias grep 'grep \!* | sed "s/:/:   /"'
```

Improves the appearance of `grep` output by substituting a "colon-plus-three-spaces" for the first "colon" on a line (if any). Compare:

Figure 3-7 `grep` *without* `sed`

```
venus% grep "H C" *
c.shell:.H C "The C Shell"
commands:.H C "More About Commands"
files:.H C "More About Files"
intro:.H C "Introduction"
manag:.H C "Managing Your Files"
preface:.UH C "Preface"
printr:.H C "More About Printing"
proc:.H C "Processes and Other Users"
```

with:

Figure 3-8 `grep` *with* `sed`

```
venus% alias grep 'grep \!* | sed "s/:/:   /"'
venus% grep "H C" *
c.shell:   .H C "The C Shell"
commands:   .H C "More About Commands"
files:   .H C "More About Files"
intro:   .H C "Introduction"
manag:   .H C "Managing Your Files"
preface:   .UH C "Preface"
printr:   .H C "More About Printing"
proc:   .H C "Processes and Other Users"
```

Or you can use a Tab rather than three spaces for better alignment. Refer to *Editing Text Files* for more on `sed`.

Example of Filters in Action

One clever trick is to create a rhyming dictionary of words using filters and the system dictionary:

Figure 3-9 *Creating a Rhyming Dictionary*

```
venus% rev /usr/dict/words | sort | rev | pr -t -3 | more
St:            UK             Elba
NCAA           BTL            alba
FAA            TTL            samba
NOAA           SIAM           marimba
ABA            IBM            Zomba
MBA            ACM            Manitoba
YMCA           CACM           Cuba
RCA            JACM           Hecuba
YWCA           SCM            scuba
FDA            FM             Aruba
ERDA           GM             tuba
USDA           NM             catawba
CIA            PM             Ithaca
USIA           RPM            portulaca
UCLA           ASTM           Dacca
AMA            CERN           Decca
BEMA           USN            Mecca
more...
```

As noted above, `rev` reverses the character order of each line. Since each word appears on a line by itself in the system dictionary, `rev` reverses the order of characters in each word. `sort` then sorts the words in order of (what was) their last character. A second pass through `rev` reverses the characters in each word a second time so that they read correctly, and you have the makings of a rhyming dictionary! Piping this through `pr` and `more`, yields a more readable display.

Using the `tee` Command

Suppose that you want to send duplicate output both to the terminal screen, and to a file for future reference. When placed in a pipeline, the `tee` command lets you direct output to more than one destination. For example, the pipeline

Figure 3-10 *Using* `tee` *to Get Dual Output*

```
venus% ls -l | grep lrwx | tee newlist
```

displays the list of symbolic links on the screen and creates a file `newlist` that contains a copy of this information as well.

With the `-a` option, `tee` appends the data onto named files that already exist. So the command:

Figure 3-11 *Using* `tee` *to Append Output*

```
venus% ls -l | grep lrwx | tee -a newlist
```

adds this information to `newlist` once again (displaying it on your screen as well).

Redirecting the Standard Error

When a command performs without problems, it produces results on its standard output. When that command encounters a problem, however, it uses a different channel to send error messages, or *diagnostic output*, to the terminal. This second channel, called the *standard error*, can also be redirected.

You can redirect the standard error to the same destination as the standard output by appending an ampersand (`&`) to the output redirection symbol.

`>&` sends both standard and diagnostic output to a destination file.[16] `>>&` appends the output to the file. `|&` includes both types of output as input to the next command in the pipeline.

If you want a command to perform silently, that is, to display no output of either kind, you can redirect its output to `/dev/null`, the system ''wastebasket.''

command `>& /dev/null`

To separate the standard error from the standard output, use a command line of the form:[17]

`(`*command* `>` *outfile*`) >&` *errorfile*

When you want to force output to appear on the terminal, you can redirect it to `/dev/tty`, (a synonym for) the name of the terminal.

command `>& /dev/tty`

So, the command

Figure 3-12 *Redirecting Standard Error*

```
venus% (nroff /usr/dict/words > /dev/null ) >& /dev/tty
```

throws away any formatted output and displays only the error messages produced by `nroff` (if any). This construction can save you time when testing long-running commands.

3.2. Escape Character, Quotes, Separation and Continuation Symbols

To indicate that a special character or symbol is to be taken as literal text, precede it with a backslash (\). By prepending the backslash, you *escape* the special meaning of the symbol.

You can use double quotes (") to surround text that you want to be interpreted as one word. For example, if you want to use `grep` to search all files for the phrase *roger, good buddy*, you would type

Figure 3-13 *Double Quotes as Escape Characters*

```
venus% grep "roger, good buddy" *
```

16 The Bourne shell uses the symbols: `2>&1` to accomplish this.

17 In the Bourne shell:
command `>` *outfile* `2>` *errorfile*

Single quotes (') also group multi-word phrases into single units. Single quotes also make sure that certain characters, such as $, are interpreted literally. (The history metacharacter, ! is always interpreted as such, unless you escape it with a backslash.) In any case, it is a good idea to escape characters such as &, !, $, ?, ., ;, and \ when you want them taken as ordinary typographical characters.

To place more than one command on a single command line, separate them with a semicolon (;). For instance, this command changes you to your home directory and then lists its contents:

Figure 3-14 *Multiple Commands on a Single Line*

```
venus% cd; ls
```

To continue a command onto the next line, use a backslash to escape the Return key.

Figure 3-15 *Commands on Two Lines*

```
venus% rev /usr/dict/words | \
sort | rev > rhymes
```

produces the rhyming dictionary described above. The terminal displays the carriage return, but the system ignores it.

3.3. grep and grep Search Patterns

You can use grep to search for *patterns* much like those you are familiar with from *Filename Substitution*.

Although the action is similar to that of filename substitution, the way you specify search patterns is different. Because they search through lines of text, grep search patterns, or *regular expressions*[18] cover a broader range of text patterns than those for filename substitution, and they have a different *syntax*.[19] Some characters with special meaning to grep also have special meaning to the system and need to be quoted or escaped. So, whenever you use a grep regular expression on the command line, surround it with quotes, or escape such characters as &, !, ., *, $, ?, and especially \, with a backslash.

Within a regular expression, dot (.) matches any single character (like ? in filename substitution). So the command

Figure 3-16 *The . Metacharacter*

```
venus% grep '.b' list
```

matches all lines in which b is preceded by a character. In effect, this matches all lines containing b, except when b is the first character on the line.

[18] The name grep is derived from the ed search and print command: g/*regular-expression*/p

[19] Although not a formal definition, you can think of the *syntax* of a command or argument as a rule for typing it in correctly.

A caret (^) anchors the pattern to the beginning of the line. So the command

Figure 3-17 *The ^ Metacharacter*

```
venus% grep '^b' list
```

matches any line starting with b. A dollar-sign ($) anchors the pattern to the end of the line. The command

Figure 3-18 *The $ Metacharacter*

```
venus% grep '^b$' list
```

matches any line in which b is the only character.

Bracketed lists and ranges work just as they do for filename substitution, but the asterisk (*) doesn't. When the asterisk follows a character, grep interprets it as 'zero or more instances of that character'. When the asterisk follows a regular expression, grep interprets it as 'zero or more instances of characters matching the pattern'. To match zero or more occurrences of any character, use

```
.*
```

Suppose you want to find lines in the text that have a period in them. Preceding the dot in the regular expression with a backslash (\) tells grep to ignore (*escape*) its special meaning. The expression

```
^\.
```

matches lines starting with a period, and is especially useful when searching for nroff formatting requests.

Table 3-1 grep *Search Pattern Elements*

character	*matches:*
^	The beginning of a text line.
$	The end of a text line.
.	Any single character (like ? in filename substitution).
[...]	Any single character in the bracketed list or range.
[^...]	Any character not in the list or range.
*	Zero or more occurrences of the *preceding character*or *regular expression.* (Not like filename substitution.)
.*	Zero or more occurrences of any single character. Equivalent to '*' in filename substitution.
\	Escapes special meaning of next character.

Going back to the rhyming dictionary, we can now use grep to produce an alliterative list of rhyming words starting with a:

Figure 3-19 *Putting it All Together*

```
venus% rev /usr/dict/words | sort | rev | grep "^a" \
| pr -t -3 | more
a               anthropomorphic    apocalyptic
amoeba          anorthic           antagonistic
alba            acyclic            anachronistic
armada          angelic            autistic
addenda         alcoholic          atavistic
agenda          apostolic          agnostic
anaconda        acrylic            acoustic
althea          aerodynamic        attic
azalea          academic           aeronautic
area            algorithmic        astronautic
alfalfa         astronomic         analytic
alga            autonomic          arc
more...
```

Refer to `grep` in the *SunOS Reference Manual* for more information about regular expressions and the `grep` family of commands.

4

The C Shell

4

The C Shell

4.1. Overview

When you type in a command, you can expect certain things to happen. By now you know that if you misspell a command the system replies with an error message. You then get a new prompt so that you can try again. When you type in the command correctly, the system waits for it to finish before giving you another prompt (unless you put it in the background with an `&`).

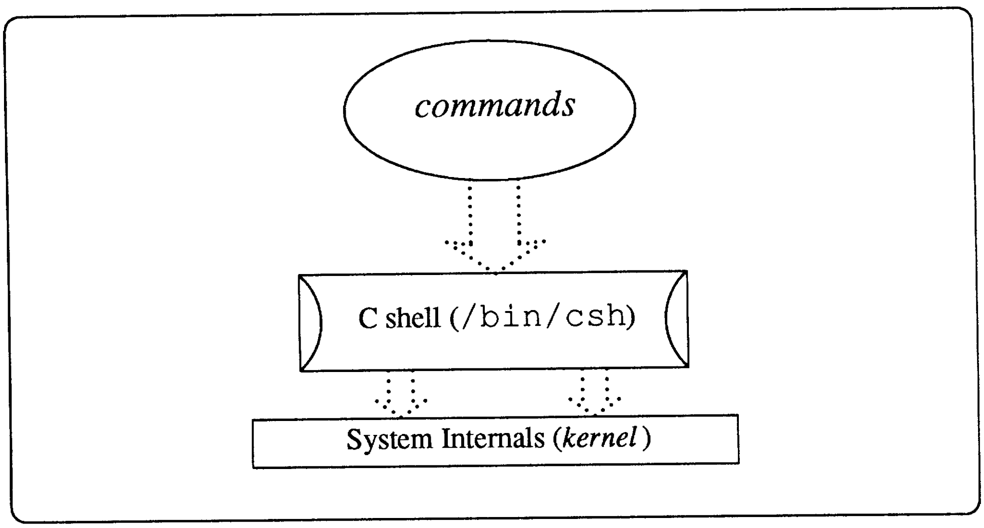
Of course, these things don't just happen by magic. A program, called a *shell*, accepts and interprets what you type, passes your interpreted commands on to be performed, and waits for each to finish before proceeding to the next.

Although the shell waits before issuing a prompt, the terminal allows you to type ahead. That is, the terminal displays what you type and passes each line along when the shell (or interactive program like `vi`) is ready for it.

There are two shells available on the Sun Workstation, the C shell, and the Bourne shell. The C shell has convenient features for interactive use, and we assume that you are using it for this purpose. The Bourne shell has fewer conveniences, but runs faster, and has a simpler syntax for writing command routines, called *scripts*.

The system starts a shell whenever you log in or create a terminal with `shelltool`. Technically speaking, the *C shell* is known as a *command interpreter*. You can think of the C shell as a layer of software between you and the system's internal workings.

Figure 4-1 *Where the C Shell Sits*



Filename substitution is one example of how the C shell interprets what you type. When you use the `*` wild card, the C shell compares it against entries in the directory and builds a list of filenames that match. It then replaces the wild card with the list, sending this expanded version of the command you typed on to the control of the system's internal scheduling mechanisms.

The way the C shell performs *alias substitution* is another example. When you type in an alias, the C shell recognizes it as such, and replaces it with the more complex command or, *expansion* that you have assigned to it.

A *shell* is an interactive program just as `Mail` and `vi` are. You can switch to a new C shell, just as you can switch to `vi`, by typing in the `csh` command. To escape such a *subshell* use [Ctrl-D] or `exit`.

You can run a command within a *noninteractive* C shell by placing it within parentheses on the command line. You have already seen an example of this in *More About Commands*, where a *subshell* is used to separate the standard output from the standard error:

(*command* > *outfile*) >& *errorfile*

The C shell provides features that you can use to further simplify entering of commands. In addition to repeating previous commands, you can use the `history` mechanism to modify them. You can put "placeholders" within alias definitions to simplify complicated commands and pipelines. And, you can define *variables* to stand for long strings or lists of words.

These and other features make the C shell easy to work with and easy to customize.

4.2. Filename Completion

Currently, filename completion will not work in SunView command or text windows unless scrolling in that window is disabled. See the *SunView 1 Beginner's Guide* on how to enable and disable scrolling. Filename completion will work in `shelltool` windows.

In addition to the wild card characters `?` and `*`, the C shell provides a *filename completion* utility which fills in the rest of a filename after you type in just the first few characters. Suppose you want to look at the file `alaska` in the directory `united.states`, which itself is the only subdirectory of directory `north.america` in directory `hinterland` in the home directory `/home/medici`.[20] (Whew!) You *could* type in:

`cat ~/hinterland/north.america/united.states/alaska`

but you'd soon get quite tired of that.

SunOS provides you with a file name completion feature. By including the line

```
set filec
```

in your `.cshrc` file, you can type the first letter, or first few letters, of a file's name and let the C shell fill in the rest.[21]

[20] This filesystem is diagrammed in the "Abbreviations for Special Directory Pathnames" section of the manual *Getting Started with SunOS: Beginner's Guide*.

[21] See the manual *Setting Up Your SunOS Environment: Beginner's Guide* for more on `.cshrc` files.

With `filec` set, type

`cat ~/h` [Esc]

and the C shell will fill in the rest of the letters of the directory name `hinterland` and leave you on the same line, ready to type in more. This is what you see:

Figure 4-2 *Using Filename Completion: I*

```
venus%  cat ~/hinterland█
```

just as though you had typed all of *hinterland* in yourself. Note that SunOS *does not process the command until you hit* [Return]; instead, it returns you to the end of the line you're typing in. Then type a / and the letter *n* followed by the [Esc]. Like so:

Figure 4-3 *Using Filename Completion: II*

```
venus%  cat ~/hinterland/n [Esc]
```

and SunOS completes the filename *north.america*:

Figure 4-4 *Using Filename Completion: III*

```
venus%  cat ~/hinterland/north.america█
```

You can do the same with the directory `united.states`. In fact, you don't have to type any of the letters *united.states* because it's the only thing in its directory. The C shell completes its name when just [Esc] is typed.

Listing Matching Files

Now suppose that in the directory `united.states` there are several files: `alaska`, `alaska.wilderness`, `alaska.urban`, and `hawaii`. You (and the SunOS file completion feature) have typed in

Figure 4-5 *Using Filename Completion: IV*

```
venus%  cat ~/hinterland/north.america/united.states/█
```

and now you want to look at `alaska.urban`.

First, type the letter *a*. This eliminates the file `hawaii` from being name-completed. Then, by typing [Ctrl-D], you make SunOS show you all the possible files and directories which start with *a*. (You could have typed in `al` or `ala` or `alaska`, the principle is the same.) It's as though you did an `ls a*` and retyped the `cat` command line. This is what you see:

Figure 4-6 *Listing Matching Files*

```
venus%  cat ~/hinterland/north.america/united.states/a [Ctrl-D]
alaska          alaska.urban          alaska.wilderness
venus%  cat ~/hinterland/north.america/united.states/a█
```

Again, you're returned to the end of the command line you're typing in. There is more than one file which starts with the letter *a*, so if you type [Esc] now, SunOS flashes the screen to indicate that it doesn't know how to finish off the file name. It fills in as much as the three choices share in common—in this case, the first six letters *alaska*—and then waits. By adding a `.` and a `u`, you make the file name unambiguous, and now your [Esc] completes `alaska.urban` and you're left with the completed command line

Figure 4-7 *A Completed Line*

```
venus%  cat ~/hinterland/north.america/united.states/alaska.urban
```

Then hit [Return] and you're done—and you've saved thirty-eight keystrokes.

(As a side note, you could do much of the above using the wild card character `*`. But file name completion with [Esc] is less ambiguous because you can see exactly what characters are being substituted, and because it displays a range of choices with [Ctrl-D]. The `*` character is better suited for working with *groups* of files.)

4.3. History Substitution and Command-Line Editing

Add this command to your `.cshrc` file if it isn't already there.

The C shell keeps a list of previous commands that you have typed in. The `history` variable determines the length of this list.

To set or change this variable, use a command of the form:

`set history=`*n*

where *n* is the number of commands to remember.

Reviewing Commands

To see the list of previous *events*, or command lines, type `history` after the prompt.

Figure 4-8 *The* `history` *Command*

```
venus% history
   1  ls
   2  cd
   3  grep -v done tasklist
   4  history
```

Repeating Commands

As you learned in *Getting Started with SunOS: Beginner's Guide*, you can repeat the most recent event by typing in two exclamation points (`!!`). The history mechanism lets you repeat any command in the events list by typing an exclamation point, followed by its command line number,

`!`*n*

for example:

Figure 4-9 *The* ! *Metacharacter*

```
venus% !3
grep -v done tasklist
...
```

You can specify the *n*'th command back,

`!-`*n*

as in:

Figure 4-10 *Backing Up*

```
venus% !-3
cd
```

You can repeat an event by typing an exclamation point, followed by the first few characters that match it,

`!`*str*

The history mechanism performs the first match it encounters. You may have to add a few characters to get the desired event. In this example the user wants to repeat the `clear` (to clear the screen) command:

Figure 4-11 *Repeating a Matching Command (An Error)*

```
venus% history
...
11   ls -l
12   clear
13   cp *.dit /tmp
14   history
venus% !c
cp *.dit /tmp
^C
```

Because the user typed in too few characters to specify the event precisely, `!c` matched the most recent event beginning with *c*, namely `cp`, even though this wasn't the event desired. The observant user interrupts it, and then types in `!cl` to match the desired event:

Figure 4-12 *Repeating a Matching Command (Correctly)*

```
venus% !cl
clear
...
```

Sometimes it's easier to match against a string of characters *embedded* within the event. To repeat a command in this way, use:

!?*str*?

where *str* is the embedded string to search for. For example:

Figure 4-13 *Matching Embedded Strings in Commands*

```
venus% !?tmp?
cp *.dit /tmp
```

Command Line Editing

A word on the command line that begins with an exclamation is referred to as an *event designator*. An event designator can stand for a previous command, or selected words from a previous command line.

You have already seen how to edit the previous command using quick substitution (^*old*^*new*^). And, you have seen how to repeat the last word of the previous command (!$). The history mechanism provides you with the means to select any word from any event in the history list, and to modify it. In some cases, it can be easier just to type the new command directly. But in many cases, command line editing can save you time and keystrokes.

You can place a :p on the end of an event designator or quick substitution to prevent the expanded command from being performed. The shell interprets the command, echos it, and places it in the history list. This gives you a chance to look at the expanded version before actually running it. If it checks out, you can use !! to run it. Otherwise you can do successive edits using

^*old*^*new*^:p

until you get it just right.

Suppose that you want to apply several commands to a long list of files, and you don't want to have to retype the list every time. !* repeats all arguments to the previous command (all but the first word of the command line). !^ expands to the first argument. If the last command was

```
echo first
```

!^ would expand to first. !:*n* expands to the *n*'th argument (*n*+1'th word).

!:0 expands to the zero-th argument, which in SunOS is the command itself. So, for example, if you type

more ***file1***

you can then type

! : 0 ***file2***

which expands to

`more` *file2*

Selecting Words Within Events

You can select a specific word from a specific event by appending a *word designator* to its event designator. A word designator has the form of a colon, followed by a character. `:*` expands to all arguments in the event. Using the history list above,

```
mv !?tmp?:*
```

expands to

```
mv *.dit /tmp
```

`:$` expands to the last argument of the selected event. `:^` expands to the first argument. `:`*n* expands to the *n*'th argument. `:0` expands to the command itself, in this case, `mv`.

Modifying Selected Words and Events

You can edit the text of an event or word by appending an *event modifier* to it. A modifier starts with a colon, followed by one or more characters that indicate the actions to perform. `:s/`*old*`/`*new*`/` substitutes *new* for *old* in the first word where there is a match for *old*. When inserted between the colon and the modifier, a `g`

indicates that the modifier applies to all designated words, not just the first. So

```
mv !?tmp?:*:gs/dit/dot/
```

expands to

```
mv *.dot /tmp
```

As mentioned above, `:p` indicates that the event or word is to be expanded and echoed, but not performed. You can place several modifiers in an event or word designator. For instance:

```
mv !?tmp?:*:gs/dot/dit/:p
```

is echoed as

```
mv *.dit /tmp
```

but not performed.

For more information about event designators, word designators, and event modifiers, refer to Appendix C, C Shell Special Characters.

4.4. Amazing Aliases

You can use *escaped* event and word designators within alias definitions to create aliases for complicated commands and pipelines. When you use the alias as a command, the escaped event designator (such as `\!*`) is replaced by command line arguments that you then type in. For instance, you might want to create an alias for a pipeline to format and then print a file.

An `alias` for `nroff` with the proper options is easy, because no characters follow the arguments you supply when using it:

Figure 4-14 *Aliases*

```
venus% alias format 'nroff -ms'
venus% format file1 file2

(formatted text appears)
```

But if you want to get the formatted output to the printer with the same command, you must supply a pipe symbol, followed by `lpr`. Rather than having to type these characters in every time, you can use the event designator `\!*` within the definition to stand for all arguments to `nroff`. When you actually run the command, the C shell replaces the event designator with any words that follow `print` on the command line.

Figure 4-15 *Event Designators in Aliases*

```
venus% alias print 'nroff -ms \!* | lpr &'
venus% print file1 file2
[1] 2832
(printed output comes out of the printer later on)
```

The `&` at the end of the line makes both `nroff` and `lpr` run in the *background*; that is, out of sight so that you can continue to type commands in while the command line is processed. (Running things in the background is explained later on in this chapter.)

You can also use the command-separation symbol `;` to create aliases that perform several commands in succession.

Figure 4-16 *Making Multi-Command Aliases*

An event designator can be used more than once within an `alias` definition.

```
venus% alias rw 'chmod +rw \!* ; ls -l \!*'
venus% rw file1 file2
-rw-rw-rw-  1 user         1699 Jul 23 13:32 file1
-rw-rw-rw-  1 user         1023 Jul 20 10:18 file2
```

Another alias that is quite useful tells you which directory you've changed to whenever you use `cd`:[22]

```
alias cd 'cd \!* ; pwd'
```

[22] Although you could use `\!:1` instead of `\!:*` (since `cd` gives an error message when used with more

To see what aliases you have, just type `alias`; to see a particular alias, type `alias` followed by the command you want to see:

Figure 4-17 *Seeing Current Aliases*

```
venus% alias rm
rm -i
venus% alias
a       alias
h       history
j       jobs -l
ls      ls -F
mv      mv -i
rm      rm -i
venus%
```

Escaping an Alias

To run the unaliased version of a command, precede the name of that command with a backslash. Here, `rm` is aliased to confirm file deletions, but in its escaped form it removes the file without checking first.

Figure 4-18 *Escaping an Alias*

```
venus alias rm
rm -i
venus% rm test
rm: remove test? n
venus% \rm test
venus%
```

Some C shell builtin commands, such as `cd` and `pushd`, cannot be escaped with a backslash. To escape these commands, put the *null string* before the command. The null string is represented by a set of empty double quotes:

Figure 4-19 *Escaping Aliases on Builtins*

```
venus% pwd
/home/venus/medici/other.directory
venus% alias cd 'echo yow'
venus% cd ; pwd
yow
/home/venus/medici/other.directory
venus% ""cd ; pwd
/home/venus/medici
venus%
```

(The semicolon separates two commands, as discussed in section 3.2.)

than one argument), it is simpler to figure out what is going on if your aliases preserve, as closely as possible, the original behavior of commands they replace.

4.5. Unaliasing an Alias

To remove an alias, simply use the `unalias`command:

Figure 4-20 *Unaliasing*

```
venus% alias rm
rm -i
venus% unalias rm
venus% alias rm
venus%
```

4.6. Variable Substitution

A *variable* is a named location in which to store text that you'd like the C shell to remember for you. You can use the `set` command to associate a variable name with a word to remember. A placeholder, composed of a dollar-sign (`$`), followed by the name of a variable, is replaced with the contents of that variable by the C shell. Thus, you can use a variable name, preceded by a `$`, as an abbreviation for its contents.

To assign a value to a variable, type in a command like:

Figure 4-21 *Setting Variable Values*

```
venus% set testdir = ~/programs/test
```

To display that variable's contents:

Figure 4-22 *Displaying a Variable's Contents*

```
venus% echo $testdir
~/programs/test
```

Suppose that you are working with files in two directories, each with very long, and very different pathnames:

```
/home/sam/sources/gfx/lines/module3
/home/bin/c/gfx/lines/module3
```

You can abbreviate these pathnames as follows:

```
set src = /home/sam/sources/gfx/lines/module3
set bin = /home/bin/c/gfx/lines/module3
```

Then, when you want to perform commands on files in these directories, you can use `$src` instead of `/home/sam/sources/gfx/lines/module3`, and `$bin` instead of `/home/bin/c/gfx/lines/module3` on the command line:

Figure 4-23 *Directories as Variables*

```
venus% cd $bin;pwd
/home/bin/c/gfx/lines/module3
venus% cd $src;pwd
/home/sam/sources/gfx/lines/module3
```

The `set` command with no arguments prints a list of all C shell variables and their current values. To see the value of a single variable, use a command of the form:

`echo $`*variable*

Storing Lists in C Shell Variables

In addition to single words, you can store a list of words in a C shell variable by enclosing the list in parentheses when you use the `set` command. One example of this is the `path` variable that you set in your `.cshrc` file. Another might be:

Figure 4-24 *Multiple-Word Variables*

```
venus% set mdirs = (/home/dakota/kitchen /home/dakota/gym)
venus% ls $mdirs
/home/dakota/gym:

aerobics        basketball      cars            dance

/home/dakota/kitchen:

anchovies       bagel           cabbages        doughnuts
venus%
```

Suppose that you just want to list those files in these directories which start with the letter *b*:

Figure 4-25 *Variables in Commands*

```
venus% ls $mdirs/b*
/home/dakota/gym/basketball

/home/dakota/kitchen:

anchovies       bagel           cabbages        doughnuts
venus%
```

This failed: `ls` lists the files starting with *b* in `/home/dakota/gym`, and *all* the files in `/home/dakota/kitchen`. This is because the */b** got appended to `mdirs` as a whole, and not to to each individual part of the variable. So typing

```
ls $mdirs/b*
```

is equivalent to typing

```
ls /home/dakota/kitchen /home/dakota/gym/b*
```

(You can operate on each member of a variable list by using the `foreach` command, described in the next section.)

You can select a specific word from the list by appending an *index* to the *call*[23] to the variable as follows:

`$`*`var`*`[`*`n`*`]`

where *var* is the name of the variable, and *n* is a number indicating the position of the word within the list. Using the above example, the word `/home/dakota/gym` is the second word in the list. So the command:

```
echo $mdirs[2]
```

displays the value

```
/home/dakota/gym
```

You can also specify a range:

Figure 4-26 *Specifying a Range for Variables*

```
venus% echo $mdirs[1-2]
/home/dakota/kitchen  /home/dakota/gym
venus%
```

But if you enclose a number in the braces that is higher than the count of words in the variable, you will get an error message. You can use filename substitution to simplify entering a list. The command:

```
set man = (/usr/man/{man,cat}?)
```

yields the following value:

Figure 4-27 *Metacharacters in Variables*

```
venus% echo $man
/usr/man/man1 /usr/man/man2 /usr/man/man3 /usr/man/man4
/usr/man/man5 /usr/man/man6 /usr/man/man7 /usr/man/man8
/usr/man/cat1 /usr/man/cat2 /usr/man/cat3 /usr/man/cat4
/usr/man/cat5 /usr/man/cat6 /usr/man/cat7 /usr/man/cat8
```

which is a complete list of all the directories containing Manual Page sources and formatted files.

[23] A call to a variable is the string you use to indicate that what you really want is the value it contains, in this case the name of the variable preceded by a dollar-sign.

Processing Lists with `foreach`

The `foreach` command provides a means to apply a set of commands successively for every word in a list. It prompts you for a set of commands, uses an *index* variable to store the current word while executing each pass through the commands, and repeats the list of commands once for each word in the list.

The syntax of the `foreach` command is:

`foreach` *index* (*list*)

where *index* is the name of the variable, and *list* is a list of words. After you type in the [Return], `foreach` prompts for a command with a question mark. It continues to prompt for commands until you type the command `end` by itself after the question mark. This signifies the end of the loop.[24] In Figure 4-25 we tried unsuccessfully to list all the files beginning with the letter *b* in the directories contained in the variable `$mdirs`. `foreach` allows you to do this:

Figure 4-28 *Using* `foreach`

```
venus% foreach i ($mdirs)
? ls $i/b*
? end
basketball
bagel
venus%
```

Here's another example. In this example, `*` is the filename metacharacter which represents all the files in a directory, and the `-n` option to `echo` is used to put all the output on the same line:

Figure 4-29 *Listing Files with* `foreach`

```
venus% foreach file (*)
? echo -n $file
? echo -n ", "
? end
```

The result is like using `ls`, except the files all appear on the same line, with a comma we specifically provided:

```
... file1, file2, file3, file4, ...
```

You can use variable substitution, as well as filename substitution symbols within the list.[25] Using the variable `man` defined above, the following `foreach` loop gives you a count of the source files and then the formatted files within each section of the Manual Pages. As the loop proceeds, the value of the index variable (written as `$dir`) changes with each pass.

---

[24] A *loop* is a set of commands to repeated successively.

[25] This also works with the `set` command.

Figure 4-30 *Listing Directories with* `foreach`

```
venus% foreach dir ($man)
? echo -n $dir
? ls $dir | wc -l
? end
/usr/man/man1      264
/usr/man/man2      118
/usr/man/man3      155
/usr/man/man4       47
/usr/man/man5       49
/usr/man/man6       36
/usr/man/man7        8
/usr/man/man8      108
/usr/man/cat1      264
/usr/man/cat2       94
/usr/man/cat3      154
/usr/man/cat4       47
/usr/man/cat5       49
/usr/man/cat6       36
/usr/man/cat7        8
/usr/man/cat8      108
```

Predefined Variables

The C shell maintains a set of predefined variables. Some of these, like `noclobber`, are used by the C shell to affect the way it behaves. Others keep track of information that the C shell needs to know about. `home`, for instance, keeps a record of your home directory. If you change the value of `home`, and then use `cd` with no argument, the C shell attempts to change directories to that new value.

Figure 4-31 *Predefined Variables*

```
venus% set home=/
venus% cd;pwd
/
venus% set home=nonesuch
venus% cd;pwd
cd: Can't change to home directory.
venus% echo $home
nonesuch
venus% cd ~
nonesuch: No such file or directory
```

Environment Variables

The C shell also maintains a set of variables, called *environment* variables; you should be familiar with them from reading *Setting Up Your SunOS Environment: Beginner's Guide* . Environment variables are passed along to any commands or subshells. They are created and modified using the `setenv` command, which has a different syntax than that of `set`.

`setenv` *name value*

There is no equal sign between the name of the variable and its value, as there is with `set`. And, only one word (or string within quotes) can be assigned to an environment variable.

Environment variables are passed to all commands and programs run from within the current shell. C shell variables are only effective within the *current* shell.

Typically, the names of environment variables are given in all capitals. In some cases, there is a lower-case equivalent used by the C shell.

Others include: `user` and `USER`, `term` and `TERM`, `shell` and `SHELL`, and `path` and `PATH`

The environment variable `HOME` is such a case. When you use the `set` command to change the value of the (`home`) shell variable, the equivalent environment variable is also changed. When you use `setenv` to change the environment variable, however, the value of the `home` shell variable is not affected:

Figure 4-32 *Exporting Variable Values*

```
venus% set home=bogus
venus% echo $home
bogus
venus% echo $HOME
bogus
venus% setenv HOME /home/sam
venus% echo $home
bogus
venus% echo $HOME
/home/sam
venus% set home=$HOME
venus% echo $home
/home/sam
venus%
```

To get a list of all environment variable and their current values, use the command `printenv`.

4.7. Output Substitution

Output substitution allows you to use the output of other commands as arguments on the command line.

When you surround a command with backquotes (` ) anywhere on the command line, the C shell starts a subshell, executes the commands within the backquotes, and substitutes the resulting output for the backquoted text. Suppose, for example, that you have a list of names in a file called `namelist`. The following command automatically mails the file `message` to each person in `namelist`.

Figure 4-33 *Output Substitution*

`echo` is a useful command for testing the results of filename, variable, and command substitution.

```
venus% echo `cat namelist`
drew@plasma
casey@bat
rvalens@labamba
loeb@leopold
venus% mail `cat namelist` < message
```

4.8. Job Control

SunOS is a *multitasking* operating system. This means that it can keep track of several users and their commands simultaneously. The system also allows you to run several commands at once by placing them in the background. The C shell provides you with the means to inquire about, stop, or bring to the foreground any job started through it.

Because each window runs with a different shell, you can't use job control to inquire about jobs started from different windows.

To see how job control works, start a background job that won't finish until you tell it to:

Figure 4-34 *A Background Job*

```
venus% vi test &
[1] 4001
```

The `[1]` is the *job* number. The `4001` is a *process number* that you can ignore for now.[26] In this case, number 1, running `vi`, is the only job that is either stopped or running in the background. When `vi` attempts to write its startup message to the terminal, it does not succeed because control of the terminal belongs to the C shell. So, `vi` stops, and waits for you to give it access to the terminal. The C shell reports any change in the status of jobs under its control, so you see a message that looks like:

```
[1] + Stopped (tty output) vi test
```

when the C shell issues the next prompt. Notice the plus sign. This indicates that the job is *current*, meaning that it is the most recent job to have stopped. A minus sign indicates that a job is *next*. When the current job is finished, a job so marked will become current.

To give a job access to the terminal, or 'bring it into the *foreground*', type in

`%`***n***

where ***n*** is the job number. If you omit the job number, the C shell brings the current job forward. When you stop an interactive program like `vi`, it waits, under job control, for you to start it running again. So, if you want to stop in the middle of `vi` without losing your place, you can type a [Ctrl-Z]. `vi` stops, and the C shell resumes control of the terminal until you type in a `%`.

[26] Processes are described in Chapter 5, Processes and Other Users.

Figure 4-35 *Moving a Job from the Background*

```
venus% %1
(the same vi screen comes up)
```

To stop the job once again, type in a [Ctrl-Z].

Figure 4-36 *Stopping a Job*

```
(vi screen)
^Z
Stopped
venus%
```

Stopping a job and resuming it can be useful when you have large programs (such as `nroff`) running, and you need to do something quickly. Rather than opening a new `shelltool` or `cmdtool`, or waiting for the big program to finish, you can stop (or *suspend*) it temporarily, perform your urgent task, and then resume the big program from where it left off.

To see what jobs are either stopped or running in the background, type in `jobs`.

To indicate that a stopped job should continue to run in the background, type in

`%`*n* `&`

where *n* is the number of the stopped job.

Figure 4-37 *Restarting Jobs in the Background*

```
venus% nroff -ms hugefile vastfile | lpr
^Z
Stopped
venus% jobs
[1] - Stopped (tty output) vi test
[2] + Stopped              nroff -ms hugefile vastfile
venus% %2 &
[2]   nroff -ms hugefile vastfile | lpr &
venus%
```

To abort a background job, use a command of the form:

`kill %`*job*

where *job* is the number of the job to kill.

Figure 4-38 *Killing Jobs*

```
venus% kill %1
[1]   Terminated          vi test
```

Exiting With Stopped Jobs

If you try to exit a shell while a job is stopped, you get the warning message:

```
There are stopped jobs.
```

A second `logout` will then log you out (but its a good idea to see what jobs are stopped with `jobs` before you exit).

`bg` and `fg`

The C shell has two builtin commands, `bg` and `fg`, which can be used to put jobs in the background or foreground. See the *SunOS Reference Manual* under `csh`.

5

Processes and Other Users

5

Processes and Other Users

5.1. Processes

After each command is interpreted by the C shell, SunOS creates an independent *process*, with a unique process ID number (PID), to perform it.[27]

The system juggles its time and *resources* amongst the various processes currently running, and uses the PID to track the progress, current status, the amount of time and the percentage of available memory each process uses.

The C shell passes its environment variables[28] (created by the `setenv` command) and their values along to the processes it starts. These are known as *child* processes. A child process may also create new children of its own.[29] In general, when a process creates a child, it waits for the child to finish before proceeding with its own tasks. As each child process completes its work, it sends an exit status number, or *return code* to its parent process. Most programs that finish normally exit with a return code of `0`. Programs that encounter errors typically exit with a status of `1` (or some other number).

To see what processes you have running, use the `ps` command. In addition to showing the PID for each process you own (created as a result of a command you typed in), `ps` also shows you the terminal from it was started, its current status (or *state*), the cpu time it has used so far, and the command it is performing.

Figure 5-1 `ps`

```
venus% ps
  PID TT STAT  TIME COMMAND
 2649 co IW    0:23 sunview
 2650 p0 IW    1:12 shelltool -C
 2651 p0 IW    0:06 -bin/csh (csh)
 6006 p1 R     0:02 ps
 2655 p2 S    34:32 shelltool
 2659 p2 IW    0:50 -bin/csh (csh)
 6000 p2 R     0:05 vi proc
```

---

27 Technically speaking, a process is an area in memory that contains a copy of the *program* indicated by the command you typed in, along with any data from the files you supplied as arguments (or from your terminal).

28 It does not pass along shell variables (created by `set`).

29 The parent is said to `fork` a child process.

The table below should help decipher the display.

Table 5-1 *Information Displayed By* `ps`

Column	*Symbol*	*Meaning*
`PID`		process ID number
`TT`		terminal:
	`co`	`/dev/console`
	mn	`/dev/tty`*mn*
`STAT`		state of the process:
	`R`	runnable (running)
	`T`	stopped
	`P`	paging
	`D`	waiting on disk
	`S`	sleeping (less than 20 seconds)
	`I`	idle (more than 20 seconds)
	`Z`	terminated, control passing to parent
	`W`	swapped out[31]
	`>`	exceeded soft memory limit
	`N`	priority was reduced
	`<`	priority was raised
`TIME`		processing time (so far)
`COMMAND`		command being performed

Terminating a Process with `kill`

`kill` provides you with a direct way to stop commands that you no longer want, even from a shell running on another terminal or from another window. This is particularly useful when you make a mistake typing in a command that takes a long time to run, such as `troff`.[32]

To terminate a process, type `ps` to find out the process ID.

You can pipe `ps` output through `grep`:
`ps | grep` *command-name*

When you see which process or processes to terminate, type in `kill` followed by the PIDs for those processes.

[31] Of the various states in the `STAT` column, `IW` can be an indication that a process is in trouble. If you find a process in this state, and if in 5 minutes or so it is still in that state, it is probably a good idea to terminate it and run the command again (checking to be sure that the command line makes sense and is typed in correctly).

[32] `troff` is a powerful text formatter that can prepare typeset-quality documents like this one.

Figure 5-2 *Terminating a Process*

Note that in Figure 5-1 `grep` reports *two* processes with the word *troff* in them.

```
venus% troff -Tlp -ms much.too.big.doc
^Z
Stopped
venus% ps | grep troff
6788 p2 S    34:32 troff -Tlp -ms much.too.big.doc
4811 p1 S     0:00 grep troff
venus% kill 6788
[1]   Terminated       troff -Tlp -ms much.too.big.doc
venus%
```

Use `kill -9` *PID* to forcefully terminate a process.

`kill` will accept either a PID number, or a job number preceded with a `%` (`%1`, for instance) as an argument.[33] You can, however, set up an alias that will search for a command by name and terminate the first process it finds running that command:[34]

Figure 5-3 `slay`

```
alias slay 'set p=`ps|grep \!*|head -1`; echo $p; kill -9 $p[1]'
```

The first part of this alias (up to the semicolon) searches for the command that you supply as an argument, strips off all but the first occurrence and stores the output line in the variable *p*. The second part displays which process it is about to kill. The third part selects the first word in the variable *p* (the PID), and kills the process with that number. Here's how `slay` works (`view` is a version of the `vi` editor):

Figure 5-4 *Using* `slay`

```
venus% view &
[1] + Stopped (tty output) view
venus% slay view
1154 p3 T 0:00 view
venus%
```

33 When run from the C shell, not the Bourne shell.

34 When you desire functions that are more complex than this, such as performing steps repeatedly or making use of more than one variable, you should consider writing a shell script to perform it. See Appendix D for information about writing Bourne shell scripts, or Appendix B for information about C shell scripts.

Timing Processes

To keep track of the system resources used by a particular command, type in `time`, followed by the command:

Figure 5-5 *The* `time` *Command*

```
venus% time wc file
58       57       536 file
0.0u 0.2s 0:01 24% 1+1k 6+0io 0pf+0w
venus%
```

`time` displays statistics about the command as follows:

Table 5-2 *Information Displayed By* `time`

Column	*Explanation*
`_ . _u`	user time
`_ . _s`	system time
`_ : _ _`	elapsed time
`_ _%`	cpu time as a percentage of elapsed time
`_+_k`	average shared memory, plus average unshared memory (kilobytes)
`_+_io`	number of block input operations, plus block output operations
`_pf+`	page faults
`_w`	swaps

When a command runs for longer than a certain number of cpu seconds (determined by the `time` C shell variable), these statistics are displayed automatically.

5.2. Running Commands Automatically

`at` and `batch`

You can take advantage of hours when the system is not heavily used to run large jobs that require a large amount of system time or memory (like formatting large documents with `troff`).

First, create a file containing the command line(s) you wish to run later on:

Figure 5-6 *Creating an* `at` *File*

```
venus% cat > atfile:
troff -ms much.too.large.document
^D
venus%
```

Then type in `at`, followed by the time you wish to run the job, and the name of the file containing the command line(s).

Figure 5-7 *Using an* `at` *File*

```
venus% at 2a atfile
venus%
```

This command tells the system to start formatting and printing the large document at 2:00am. You can use up to four digits to specify the time in hours and minutes, followed by an `a` for am, or `p` for pm.

`batch` is similar to `at` except that, instead of running a job or bunch of jobs at a time you choose, `batch` *sends* the jobs off immediately to be executed, but waits until the system load level is low before actually *running* them.

There are two files, `at.allow` and `at.deny`, which regulate who can use the `at` command. For more on `at` and `batch`, see the *SunOS Reference Manual* or type **`man at`**.

Running Commands Periodically — `crontab`

`at` and `batch` are useful for running jobs on a one-time basis. To run commands periodically, use the `crontab` command. For example, you can use it to clean out your `/tmp` directory on the fifteenth of each month, or reset your clock every day.

`crontab` is a program which edits the file `/var/spool/cron/crontabs/`*username*, where *username* is the your login name. This file contains a number of commands to execute. Each command is preceded by the time (and date, if needed) the command is to be run. The command may be an actual shell command or it may be the name of an executable file, such as a shell script.

In earlier versions of SunOs, each machine had a single `crontab` file, which everyone using the machine shared, and the user had to become superuser (`root`) in order to modify it.

Each person on a machine has his or her own `crontab` file, including `root`. Commands which only the superuser can execute — such as `rdate` and `sa`, described below — must be in `root`'s `crontab` file, while other commands or files to execute can be in your own `crontab` file.[35]

A `crontab` file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1=Monday). Each of these patterns may contain a number in the range above; two numbers separated by a dash meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is the command or file to be executed.

Here are some example lines from a `crontab` file, to give you a better sense of the file's format:

[35] See Section 5.4 for information on becoming `root`, your machine's *superuser*.

Figure 5-8 *Some Typical* crontab *Entries*

```
15 0 * * * /usr/etc/sa -s >/dev/null
0,20,40 * * * * /usr/ucb/rdate chiqui
0 1 15 * * /home/titan/medici/mail/.mailrun
```

For more on rdate and sa, see the *SunOS Reference Manual.*

- The first line says to run the sa command at fifteen minutes past midnight, every day. (sa is a program run by root to maintain system accounting files; in this case, it does some maintenance and produces a summary report which is automatically sent to the "garbage" directory /dev/null.)
- The second line says to run rdate every twenty minutes. rdate gets the time and date from the machine chiqui. (rdate also is run by root.)
- The third says to run the script .mailrun at one A.M. on the fifteenth of every month; .mailrun is some script, probably to manage mail files, written by user medici; it does not have to go in root's crontab file.

To use the crontab program, type

Figure 5-9 crontab

```
venus% crontab -e
```

A *daemon* is a program which the system runs to do certain housekeeping chores. A print daemon, for example, might queue up and print files, while a mailer daemon takes care of the details of sending electronic mail. Most daemons are invisible to the user.

This starts you editing your crontab file (it creates one if none exists). Type in the commands you want, in the format given above. A permanent process, the cron daemon, examines it and executes its commands at appropriate times.

You must pay attention to file permissions when using crontab. Make sure that any files you want cron to run are marked as executable. Moreover, because the cron daemon is owned by the system and not by you, you must make sure that any files you want it to erase have their permissions set to allow this. Likewise, if cron is creating files for you to access, you must have cron set open permissions on them for you. For more information on permissions, see Section 2.3.

As with at, there are two files, cron.allow and cron.deny, which regulate who can make or modify crontab files on your machine. For more on cron and crontab, see the *SunOS Reference Manual.*

5.3. Other Users

By now you've realized that to the system you're not just another pretty face. From the system's standpoint, every user has a login name, a password, an identification number, or *userid*, a group membership, a user's name or other pertinent data, a home directory, and a default shell. This information is kept in the file /etc/passwd. To find out who can log in to your system, look in this file.

Figure 5-10 *The* `/etc/passwd` *File*

```
root:0XtYHFnkYou3Y:0:10:Operator:/:/bin/csh
daemon:*:1:1::/:
uucp:eXs0qzRjUOS8Y:4:4::/var/spool/uucppublic:
cindy:Lu8UBYYbPNEpw:26:20:Cindy Smith:/home/cyndi:/bin/csh
carter:SQxRMoQbqQ0Hk:612:20:Jamie Carter:/home/carter:/bin/csh
jimg:lUvG9UKY0uE/A:1131:60:Julie Gomez:/home/jimg:/bin/csh
ben:bAwVM.A6LiXFo:1132:30:Ben Benson:/home/ben:/bin/csh
karla:mceur1TqKdcDQ:1172:30:Karla Caracas:/home/karla:/bin/csh
```

Fields corresponding to the above categories are separated by colons, and described in the following table (using the last line above as a sample entry).

Table 5-3 *Information Contained in* `/etc/passwd`

Field	*Sample*
login name	karla
encrypted password	mceur1TqKdcDQ
user ID number	1172
group ID number	30
commentary	Karla Caracas
home directory	/home/karla
login shell	/bin/csh

The first line of this file contains an entry for `root`, the operator of the system. When logged in as `root`, the operator can access any file or device on the system, perform system maintenance, and edit system files such as this. (For more on `root`, see Section 5.4.) The next two entries allow for certain networking functions to be performed, and the subsequent lines correspond to individual users.

If you are using the Yellow Pages, then a single plus sign (+) on a line by itself in `/etc/passwd` gives login privileges on your machine to anyone in the Yellow Pages directory. To find out more about the Yellow Pages, and users with access over the network, refer to *Using the Network: Beginner's Guide* or *System and Network Administration*.

For a more complete treatment of `/etc/passwd`, see the *SunOS Reference Manual* or type **`man 5 passwd`**.

Users Currently Logged In

The system tries to provide equivalent performance to everyone using it. To find out who is logged in, type `who`.

Figure 5-11 `who`

```
venus% who
landon     tty0c    Aug 30 11:35
wilkie     tty17    Aug 30 09:02
dewey      tty09    Aug 30 08:44
goldwatr   tty19    Aug 28 16:04
ford       tty16    Aug 29 15:06
bush       ttyp0    Aug 30 12:50    (iowa)
venus%
```

`who` shows you the login-name of each user on the system, the terminal that person is using, when they logged in, and, if logged in from a remote machine, the name of that machine.[36]

From time to time, you may want to see what others are doing. The `w` command tells you what command is running on each user's terminal. In addition, it shows you the amount of time since the user last typed something in (idle), the total CPU time spent by each user so far (JCPU), the CPU time spent by the command now running (PCPU).

Figure 5-12 `w`

```
venus% w
  1:18pm  up 4 days,  2:51,  14 users,  load average:
0.32, 0.20, 0.00
User      tty       login@  idle   JCPU   PCPU  what
landon    tty0c   11:35am    12     54     14   -csh
wilkie    tty17    9:02am          6:20    26   mail
dewey     tty09    8:44am          1:57     8   -csh
goldwatr  tty19   16:04am  3:10     22      4   mail
ford      tty16   15:06am  1:40     18      4   -csh
bush      ttyp0   12:50pm    13      7      4   -csh
venus%
```

To get a detailed list of everyone's processes, use the command

```
ps -au
```

[36] See *Using the Network: Beginner's Guide* for more information about using remote machines.

Figure 5-13 `ps -au`

```
venus% ps -au
USER       PID %CPU %MEM   SZ  RSS TT STAT  TIME COMMAND
landon   19755 49.8 10.0  212  140 0c R     0:03 ps -au
wilkie   19751 42.4 15.8  366  226 17 S     0:12 vi mail.record
dewey    19754  4.8  8.3  232  114 09 S     0:02 /usr/lib/sendmail -bm c2
goldwatr 18732  0.0  0.0  186    0 19 IW    0:44 mail
ford     19752  0.0  2.2   70   24 16 S     0:00 pmsg
bush     18085  0.0  0.0  300   86 p0 IW    0:10 vi eco
venus%
```

The `-a` option tells `ps` to show you information about all processes, not just your own. The `-u` option gives a more detailed display that includes the name of the user who owns the process. The `-au` option is simply the combination of these two.[37] For information about the remaining columns, refer to `ps` in the *SunOS Reference Manual.*

Changing Identity with `su`

If you know someone else's password, you can temporarily assume that person's system identity by using the `su` (*superuser*) command. A common reason for doing so is to get access to files that you don't own. Suppose that a colleague has moved a file into one of your directories that you want to edit:

Figure 5-14 *An Alien File*

It is usually better to copy such a file yourself, since you often don't know the password of another user.

```
venus% ls -l
total 34
-r--r--r--  1 sam      1697 Aug  2 13:35 env.b
-r--r--r--  1 sam      1244 Aug  2 13:50 chapter.1
-r--r--r--  1 jd       3623 Aug  2 13:50 program.source
```

First, use `cp` to make a copy of the file. You will own the copy, and can edit it. To get rid of the version you don't own, switch your userid and delete it:

Figure 5-15 *Using* `su`

```
venus% cp program.source my.source
venus% su jd
Password: ...
venus% rm program.source
venus%
```

To revert to your previous ID, enter a [Ctrl-D] (or the command `exit`).

If, after switching userids, you want to find out what your effective login identity is, type `whoami`:

37 Single-letter options that can be combined like this are sometimes referred to as *flags*.

Figure 5-16 `whoami`

```
venus% whoami
jd
venus% ^D
venus% whoami
sam
```

The command

`who am i`

reveals your original login identity when you use `su` to temporarily become someone else. For more on `who am i`, see *Using the Network: Beginner's Guide.*

5.4. Becoming `root`, the superuser

Each machine has a *superuser*, a user who has powers and permissions quite above and beyond those of mortal users. This superuser is often known as `root`. A person with superuser status can edit files which are off-limits to ordinary users, such as `/etc/passwd`, the password file, or `/etc/hosts.equiv`, the list of other machines on a network which your machine trusts. `root` can also use some restricted commands, such as `mount` or `reboot`.

Originally, the UNIX operating system, on which SunOS is based, was designed for many users to be working on a single, more-or-less centralized machine. One person, the System Administrator, was in charge of maintaining, configuring, and upgrading the system — hence the name *superuser*.[38]

With a network of independent workstations like Suns, however, each person may have the ability to become `root` on his or her own machine, and take care of many of the tasks which were formerly the province of the superuser, such as making connections to printers or mounting remote filesystems. In a workstation environment, then, a superuser and a System Administrator are not necessarily the same thing: a System Administrator would be someone who maintains *shared* machines and networks.

For example, suppose you are a diskless client of the server `chiqui`. That means that you have your own workstation — call it `venus` — and you keep your files on the machine `chiqui`. On your own machine, `venus`, you can become superuser. But maintenance and configuration of `chiqui` is left to your System Administrator. On the other hand, if you are running a standalone system (one with a disk), then you are the System Administrator, and you become `root` to carry out all System Administrator tasks.

If you type `su` with no name, it attempts to switch you to `root`, also referred to as the *superuser*. When you become the superuser, the last character of the prompt changes from a percent sign (`%`) to a pound sign (#).

---

[38] This is still the setup for people using "dumb" terminals.

Figure 5-17 *Becoming* `root`

```
venus% su
Password: ...
venus#
...
^D
venus%
```

As `root`, you can kill any process running on your machine. You have read and write privileges on every file on your machine's disk (or disk partition) and you can change the ownership of these files.[39] Additionally, there are a number of commands, such as `mount` and `reboot`, which require that you be superuser to use them.

You must become `root` to perform system maintenance tasks such as adding new users, adding new terminals or printers, etc. Refer to *System and Network Administration* for more information on performing these tasks.

Sun386i users can use `SNAP` instead of `su`; see the *Sun386i SNAP Administration* manual.

[39] Files mounted from a remote host belong to that machine. You must be logged in as `root` on the remote host to get superuser privileges for files that reside on it. Refer to *Using the Network: Beginner's Guide* to find out more about remote hosts and mounted file systems.

6

Managing Your Files

6

Managing Your Files

SunOS has good facilities to help you locate files, monitor changes to important files, and manage your space on the disk.

6.1. Locating Files

To locate a file in the file system hierarchy, you may need to know its absolute pathname. When trying to locate a file, chances are that you are either looking for the pathname of a particular command, or you are looking for a certain text file. SunOS provides several ways to locate commands. These are presented first, followed by methods for locating text files.

Looking Up a Command with `whereis` and `which`

To find the pathname of a standard SunOS command, type in `whereis` followed by the command name. (`whereis` also displays the pathname of the `man` entry.)

Figure 6-1 `whereis`

```
venus% whereis csh
csh: /bin/csh /usr/man/man1/csh.1
```

You can also use `which` to look up a command. This is useful when you have commands that are aliased, or if your system contains commands in addition to the standard set. If the command is an alias, `which` shows you its definition. If the command is in a directory listed in your `path` variable, `which` displays its pathname. If there is more than one version of a command in those directories, `which` displays the version that the system finds first. This is the same version that the system performs when you type the command in.

Figure 6-2 `which`

```
venus% which ls
ls:      aliased to ls -F

venus% which chesstool
/usr/games/chesstool
```

Looking Up a Command's Description with `whatis`

`whatis`, followed by the name of a command, will give you a brief description of what that command does.

Figure 6-3 `whatis`

```
venus% whatis whatis
whatis (1)              - describe what a command is
```

Looking Up Files with `find`

Starting with a named directory,[40] `find` searches for files that meet conditions you specify. A condition could be that the filename match a certain pattern, that the file is owned by a certain user (or belong to a certain group), or that the file has been modified within a certain timeframe.

Unlike most SunOS commands, `find` options are several characters long, and the name of the starting directory must precede them on the command line.

`find` *directory options*

Each option describes a criterion for selecting a file. A file must meet all criteria to be selected. So, the more options you apply, the narrower the field becomes. The `-print` indicates that you want the results to be displayed. (As described later on, you can use `find` to run commands. You may want `find` to omit the display of selected files in that case.)

The `-name` *filename* option tells find to select files that match *filename*. Here *filename* is taken to be the rightmost component of a file's full pathname. For example, the rightmost component of the file `/usr/lib/calendar` is `calendar`. This portion of a file's name is often called the *basename*. To see which files within the current directory and its subdirectories end in `s`, type in:

Figure 6-4 `find`

```
venus% find . -name '*s' -print
./programs
./programs/graphics
./programs/graphics/gks
./src/gks
...
venus%
```

Other options include:

`-name` *filename*	select files whose rightmost component matches *filename*. Surround *filename* with single quotes if it includes filename substitution patterns.
`-user` *userid*	select files owned by *userid*. *userid* can be either a login name or user ID number.

[40] You must supply a name.

`-group` *group*	select files belonging to *group*.
`-mtime` *n*	select files that have been modified within *n* days.
`-newer` *checkfile*	select files modified more recently than *checkfile*.

You can combine options within (escaped) parentheses (`\(...\)`) to specify an order of precedence for criteria. Within escaped parentheses, you can use the `-o` flag between options to indicate that `find` should select files that qualify under either category, rather than just those files that qualify under both.

Figure 6-5 *The* `-o` *Option to* `find`

```
venus% find . \( -name AAA -o -name BBB \) -print
./AAA
./BBB
```

You can invert the sense of an option by prepending an escaped exclamation point. `find` then selects files for which the option does *not* apply.

Figure 6-6 *Reversing a* `find` *Option*

```
venus% find . \!-name BBB -print
./AAA
```

Running Commands with `find`

You can also use `find` to apply commands to the files it selects with the

`-exec` *command* `'{}' \;`

option. This option is terminated with an escaped semicolon (`\;`). The quoted braces are replaced with the filenames that `find` selects.

You can use `find` to automatically remove temporary work files. If you name your temporary files consistently, you can use `find` to seek them out and destroy them wherever they lurk.[41] For example, if you name your temporary files *test* or *dummy*, this command will find them and remove them:

```
find . \(-name test -o -name dummy \) -exec rm '{}' \;
```

Looking at File Types with `file`

Sometimes you want to see what sort of data a file contains without having to look at its contents. In particular, if the file is a compiled program (*object-file*), trying to display its contents can produce spectacular and disconcerting results on your screen. `file` quickly tells you whether a file contains, for example, plain text, `troff` sources, C program sources, executable files, or tape-format archives. (There are a number of kinds of files; see under *file* in the *SunOS Reference Manual*.

[41] For good housekeeping, you may want to get rid of such files on a regular basis without having to think about it. If you put a command like this in your `.logout` file, then whenever you log out, the system will clean up unwanted files for you.

Figure 6-7 *What Kind of File Am I?* `file`

```
venus% file *
AAA:  empty
document:  nroff, troff, or eqn input test
troff.output:  troff (CAT) output
program:   demand paged pure executable
scratch:  ascii text
```

6.2. Looking at Differences Between Files with `diff`

It often happens that different people with access to a file make copies of it and then edit their copies. `diff` will show you the specific differences between versions of a file and provide you with an indication of how the contents of one can be edited to produce the other. The command

`diff` *leftfile rightfile*

scans each line in *leftfile* and *rightfile* looking for differences. When it finds a line (or lines) that differ, it determines whether the difference is the result of an addition, a deletion, or a change to the line, and how many lines are affected. It tells you the respective line number(s) in each file, followed by the relevant text from each.

If the difference is the result of an addition `diff` displays a line of the form

l[,*l*] **a** *r*[,*r*]

where *l* is a line number in *leftfile* and *r* is a line number in *rightfile*. If the difference is the result of a deletion, `diff` uses a `d` in place of `a`; if it is the result of a change on the line, `diff` uses a `c`.

The relevant lines from both files immediately follow. Text from *leftfile* is preceded by a *left* angle-bracket (<). Text from *rightfile* is preceded by a *right* angle-bracket (>). This example shows two sample files, followed by their `diff` output.

Figure 6-8 *Two Sample Files and* `diff` *Output*

```
venus% cat sched.7.15
Week of 7/15

Day:  Time:       Action Item:       Details:

T     10:00       Hardware mtg.      every other week
W     1:30        Software mtg.
T     3:00        Docs. mtg.
F     1:00        Interview
venus% cat sched.7.22
Week of 7/22

Day:  Time:       Action Item:       Details:

M     8:30        Staff mtg.         all day
T     10:00       Hardware mtg.      every other week
W     1:30        Software mtg.
T     3:00        Docs. mtg.
venus% diff sched.7.15 sched.7.22
1c1
< Week of 7/15
---
> Week of 7/22
4a5
> M     8:30        Staff mtg.         all day
8d8
< F     1:00        Interview
```

6.3. Monitor Changes with `sccs`

When you want to protect an ASCII file from accidental deletion, keep track of changes to it, or allow more than one person to modify it, you can monitor the file using `sccs`. `sccs`, or "source code control system" is a utility program that protects important files by allowing only one person at a time to make changes, by maintaining a record of those changes, and by rebuilding the current (or any previous) version upon request.

Putting a File Under `sccs` Control (`sccs create`)

To put a file under `sccs` control, perform the following steps:

1. `cd` to the directory containing the file(s) to be protected. If a subdirectory name `SCCS` is not already present, create it. If you want to allow other users access to the files, change the permissions of the current directory and those of the `SCCS` subdirectory to `775`.[42]

[42] Unless you are sure that you do *not* want them to have access, it is normally a good idea to change permissions of both directories to allow it, at least for other members of your user group.

Figure 6-9 *Putting Files under* `sccs`

```
venus% cd project
venus% mkdir SCCS
venus% chmod 775 . SCCS
```

2. Type in a command of the form:

   `sccs create` *filename* ...

   filename is the name of a file or files to monitor. This is how you would put all you files under SCCS:

Figure 6-10 `sccs create`

```
venus% sccs create *
```

For each file that you indicate on the command line, `sccs` produces a special file called a *history* file, and puts it in the `SCCS` subdirectory. The history file has a name of the form:

`s.`*filename*[43]

and contains a complete record of all lines changed throughout the life of the file. `sccs` maintains a checksum on all history files, so *do not* edit them!

`sccs` may respond with the warning:

`No id keywords (cm7)`

This message can safely be ignored when you are auditing your own files.

When working with files that are part of a large project, `sccs` ID keywords can be important. Refer to *Programming Utilities for the Sun Workstation* for more information about `sccs` as a tool for managing large programming projects.

3. Remove the backup file(s) that `sccs` leaves behind. These files are created by `sccs` as a safety precaution, and are no longer necessary once the `create` operation is complete. Names of these backup files begin with a comma (`,`).

Figure 6-11 *Removing Backup Originals*

```
venus% rm ,*
```

Once under `sccs` control, you have to check a file out before you can make changes to it. Files that aren't checked out through `sccs` have permissions set to read-only for everyone (`444`).

[43] History files are also referred to as "s.files."

Which Files are Checked Out? (`sccs info`)

To see which files in the working directory are checked out, use the `sccs info` command. If no files are checked out, `sccs` responds with the message:

```
Nothing being edited
```

If there are files checked out, it lists those that are, the current version number of each, the version number each will have when checked in again, the name of the user who checked out each and the date and time of check-out:

```
csh.1: being edited: 1.4 1.5 sam 85/09/04 16:32:15
```

Recovering the Current Version (`sccs get`)

Because several people may have write access to the directory, it is possible that a file in the working directory may be deleted accidentally. Files that *aren't* under `sccs` control are gone for good once they are removed, but you can easily restore files under `sccs` from their history-files using the `sccs get` command:

`sccs get` *filename*

If you want to recover the current version of all files in the directory, use the command:

```
sccs get SCCS
```

Checking a File Out (`sccs edit`)

Only one person at a time can check a file out. This assures you that changes won't be lost, garbled, or intermixed between the edits of different users. To check out a file, type in `sccs edit` followed by the file or files you wish to check out. `sccs` will respond with the current version number, the new version (delta) number, and the number of lines in the file.

Figure 6-12 *Checking a File Out*

```
venus% sccs edit program
1.1
new delta 1.2
220 lines
venus%
```

Once checked out, you can edit the file using `vi`, or an editor of your choice.

When you check out a file, `sccs` changes the ownership of the file to you, gives you write permission (owner only), and places a *lock* file containing your userid, the version number, and other information in the `SCCS` directory.[44] When you check the file back in, the lock file is removed and the permissions are set to read only, but you retain ownership of the file.

[44] The lock file has a name of the form: `p.`*filename*, and referred to as a "p-file."

Looking at Current Changes (`sccs diffs`)

While still checked out, you may want to review the changes you have made so far. To do so, type in:

`sccs diffs` *filename*

`sccs` responds with standard `diff` output, using `sccs`'s current version as the "leftfile" and the *filename* as the "rightfile." (See section 6.2.)

Checking a File In (`sccs delget`)

When you are done making changes you can check in the new version of the file by typing in the nonintuitive command:

`sccs delget` *filename*

`delget` is a contraction for `delta`, the command to incorporate a new version into the history file, and `get`, the command to recover the newest version (that you are just now checking in).[45]

When you use `delget` (or `delta`) to check in the file, `sccs` asks you for a line of comments. These comments are included in the history file, and should briefly summarize the changes you have made. After adding your comments and pressing [Return], `sccs` responds with the new version number, the number of lines inserted, deleted and unchanged, and the total number of lines.

Figure 6-13 *Checking a File In*

```
venus% sccs delget program
comments? added remarks for more readable code
1.2
43 inserted
18 deleted
287 unchanged
1.2
348 lines
```

A replaced line shows up as an insertion and deletion.

Backing Out With No Changes (`sccs unedit`)

To check a file back in without any changes, type in:

`sccs unedit` *filename*

Looking at the File's History (`sccs prt`)

To review a file's history, use the command:

`sccs prt` *filename*

This command shows you the version number, comment lines, date checked in, and user responsible for each version of the file.

[45] If `sccs` responds with an error message, it does not perform the `get` action, and you may have to recover files using `sccs get SCCS`.

Figure 6-14 `sccs prt`

```
venus% sccs prt program
SCCS/s.program:

D 1.2 85/09/04 12:51:07 sam 2 1 00042/00008/00357
MRs:
COMMENTS:
added remarks for more readable code

D 1.1 85/08/30 16:54:57 sam 1 0 00365/00000/00000
MRs:
COMMENTS:
date and time created 85/08/30 16:54:57 by sam
```

Comparing Versions (`sccs sccsdiff`)

To compare previous versions of a file, use the command

`sccs sccsdiff -r`*x.y* `-r`*m.n* *filename*

Where *x.y* and *m.n* are version numbers to be compared. This command produces standard `diff` output.

Restoring a Previous Version (`sccs get -r`)

If you want to back out a version of the file that is already checked in, you must perform the following steps:

1. Recover the previous version. You can look up its number using `sccs prt` *filename*. To rebuild the previous version, type in a command of the form:

   `sccs get -r`*x.y* *filename*

   where *x.y* is the desired version number.

2. Rename the recovered version of the file

   `mv` *filename* `temp`

3. Check the file out with `sccs edit`.
4. Replace the checked-out version with the old version:

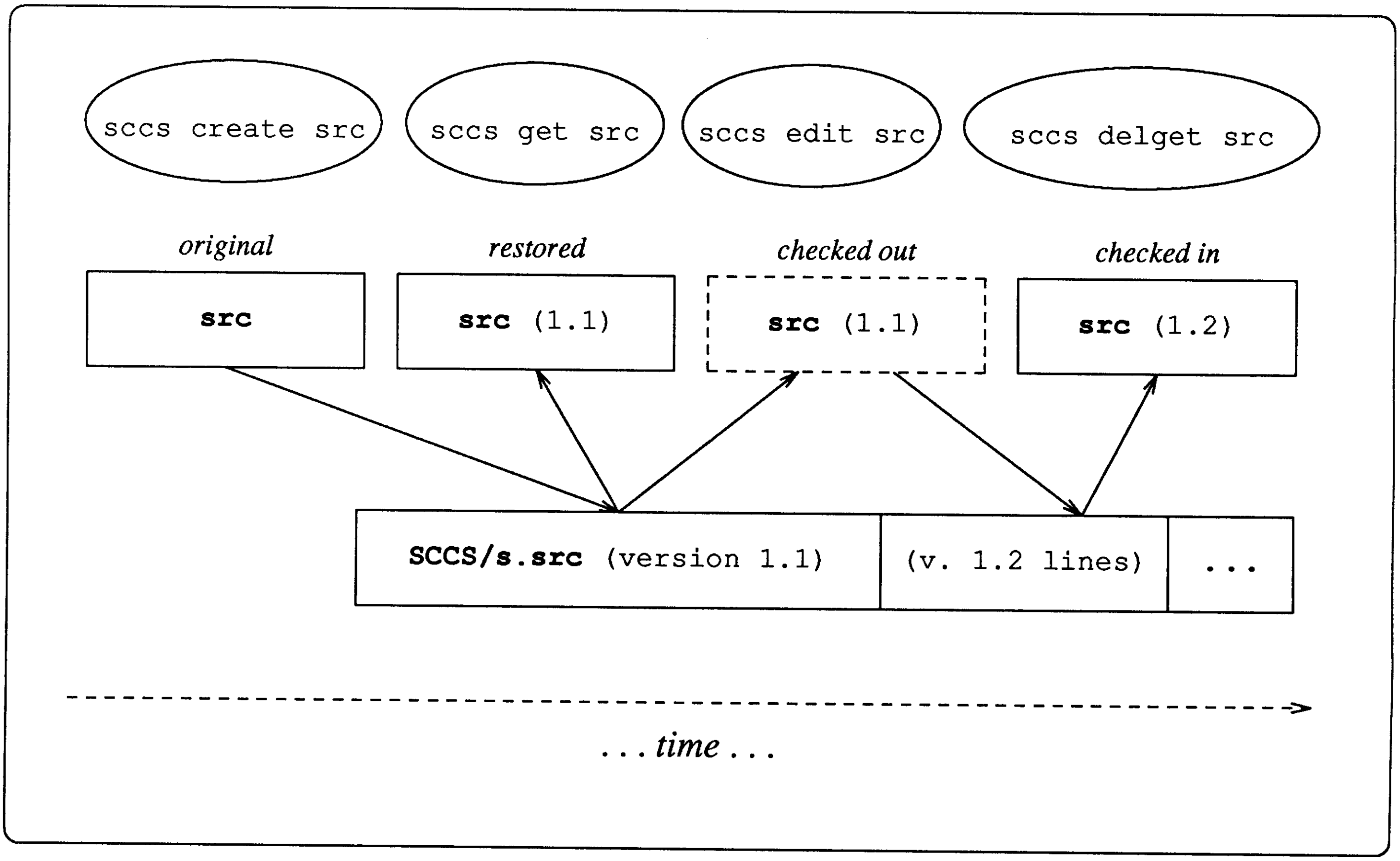
   `mv temp` *filename*

5. Check the file back in with `sccs delget`.

To assure that it all worked properly, compare the latest version with the desired previous version using `sccs sccsdiff`.

The typical flow of events when making changes to a file under `sccs` control is:

Figure 6-15 *Flow of Events with* `sccs`*-Controlled Files*



Solving Problems with `sccs`

`sccs` is a complicated and verbose utility. There may be times when it responds with an error message even though things worked properly. Its error messages are sometimes difficult to interpret. If you are not sure that `sccs` succeeded in doing what you asked, you can take certain steps to verify whether it has:

Are Files Under `sccs` *Control?*

`ls -l SCCS`
: will show an s.file for each file under `sccs` control.

Is the File Checked Out?

`sccs info`
: will show which files are checked out, and to whom.

Was the File Checked In?

`sccs prt`*filename*
: will show your comments in the first three lines when you have checked in a file successfully.

What If I Can't Check the File Out?

If you attempt to check a file out and you get the message:

`ERROR [SCCS/s.`*filename*`]: writable '`*filename*`' exists (ge4)`

this usually means that someone has the file checked out already. You can verify this using `sccs info`. If `sccs info` does not list the file as being edited, then the lock file in the SCCS directory has been deleted. When this happens `sccs` will not allow anyone to check the file either in or out.

To correct this problem, first run `sccs diffs` on the file to see if it differs from the version last checked in. If so, it is a good idea to contact the file's owner to find out if the changes made should be kept. If so, then copy the file to a new filename, remove the writable original, and check the file out using `sccs edit`. Then move the new filename back to the original name (overwriting the checked out version), and check the new version back in using `sccs delget`.

If the changes need not be saved, you can correct the problem by simply removing the writable file, restoring the current version using `sccs get` and then checking it out using `sccs edit`.

6.4. Automating Complicated Tasks with `make`

Performing complicated tasks, such as producing object code for programs or formatting large documents involves processing different files through various programs at the proper times and in the proper order. This can be a lot to remember. `make` simplifies these complications by following a record of the steps involved, called a *makefile*, that you create.

The makefile contains a list of the steps called *targets;* each target contains a list of SunOS commands. A target can be qualified by a list of other targets upon which it depends. One target is said to depend on another if the latter must be be completed before the former can be performed successfully. The latter target is called a *dependency.*

For example, an SCCS subdirectory must be created before you can put files under `sccs`. And, you must put a file under `sccs` with `sccs create` before you can check that file out. So the command `sccs edit` depends in practice on the commands `mkdir SCCS` and `sccs create` for its own success.

`make` uses the list of targets as a recipe to produce a desired program, document, or other object file called a *target file,* or simply *target.*

`make` performs only those steps that are required to bring the target files up to date. The makefile lists the various steps involved, and how they depend on one another, and `make` examines the list to see which target files are outdated.

A target is considered to be outdated when a source file used to produce it has changed since the target file itself was last produced. `make` then performs only those steps required to replace any outdated target files.

`make` has a facility to perform *macro substitution.*[46] This allows you to abbreviate long lists, and to predefine parameters that often change, so that with a few simple edits the same procedure can be used to produce other, similar objects.

[46] Like an alias, a *macro* is a string of text that is replaced by its definition, or *expansion* when encountered in an input file (or command line).

Makefiles

Like a recipe card, a makefile is composed of two sections. The first section is a list of macro definitions. These are described in detail later on. The second section outlines steps in the procedure and their relationships to one another. In `make` parlance, each step is called a *target*.

Each target has a name. If that target's function is to produce an object file of some sort, then the name of the target should be the same as the name of the file it produces. If the target performs some sort of housekeeping step, then it can have any name you like.

A target may also have a list of *dependencies*, or targets it depends on, associated with it. `make` uses this list to determine whether files produced by the target are up to date.

Finally, each target has a list of SunOS commands to perform. When performing a step, `make` performs each command in turn, starting a Bourne shell[47] for each command line.[48]

The following is an example of a makefile to put the contents of a directory under `sccs` control. The file consists of just three targets, and no macro definitions:

Figure 6-16 *Sample Makefile to Put Files under* `sccs`

```
# makefile: for putting files under sccs

#         no macro definitions

#         target definitions

put.under:  SCCS
	# these lines begin with a required tab character
	-sccs create *
	-rm ,*
	-sccs get SCCS

SCCS:
	-mkdir SCCS
	-chmod 775 SCCS .
```

The targets are `put.under` and `SCCS`. The target `put.under` depends on the target `SCCS`. If the `SCCS` directory is not already present and up to date (directories always are), `make` performs the commands listed under `SCCS` first.

The format of each target is significant. The name of the target must be followed by a colon and the list of dependencies, if any. (If this list is longer than one line,

[47] Because it runs a Bourne shell, certain C shell constructs, such as `foreach`, don't work. Refer to `sh` in the *SunOS Reference Manual* for more information about the Bourne shell.

[48] Since each command line is executed in its own shell, you must use the command separation character ; , and the command-line continuation character \ [Return] to build command *routines*.

you can split it in two by leaving a backslash (\) at the end of the first line.) The list of commands immediately follows the target name, and each command line begins with a [Tab].

Comments begin with a #, and can be placed to the right of commands on any line (not ending in a backslash). At least one blank line separates target definitions from one another.

When you prepend a – to a command, `make` ignores a nonzero (error) return code from that command. Normally, `make` halts whenever a command it runs exits with a nonzero status. Adding the dashes in this case tells `make` to continue putting new files under `sccs` control, even though it may encounter older files already there.

Because `make` checks for dependencies, you can write makefiles in a top-down fashion. The step that produces the final output should appear first. Steps that it depends upon can appear next, followed by steps that *they* depend on.

Running `make`

When the makefile is ready, simply type in `make`.

`make` looks for a file in the working directory named `makefile`, or `Makefile`,[49] checks for dependencies, beginning with the first target it encounters, and then performs commands in their proper order.

Figure 6-17 *Running* `make`

```
venus% make
mkdir SCCS
chmod 775 SCCS .
sccs create *

SCCS:
ERROR: directory 'SCCS' specified as 'i' keyletter value (ad29)

makefile:
No id keywords (cm7)
[messages from sccs]
rm ,*
sccs get SCCS
[messages from sccs]
venus%
```

The error message

```
ERROR: directory 'SCCS' specified as 'i' ...
```

indicates that `sccs` attempted to create a history file for the directory `SCCS`. Because we used a dash as the first character of the command line, `make` continued processing.

[49] You can specify the name of some other makefile, using the `-f` *filename* option, as in `make -f buildit`, where `buildit` is a different Makefile.

Testing Makefiles

Most makefiles take a bit of debugging. To find out what commands `make` will perform without actually running them, use the `-n` option.

Figure 6-18 *The* `make -n` *Option*

```
venus% make -n
sccs create *
rm ,*
sccs get *
```

In the above makefile, `put.under` depends upon `SCCS`. When you ran `make` the first time, the `SCCS` directory was created. When you ran `make -n` subsequently, `make` did not indicate that it would perform that step (since it was up-to-date anyway). If you were to remove the `SCCS` directory, and then run `make`, it would perform commands in the `SCCS` target once again.

Defining Macros in the Makefile

The next example is a makefile used to format and print a document made up of several source files. With macro substitution, copies of a makefile such as this can be used for different documents:

Figure 6-19 *Sample Makefile for Printing a Document*

```
# Makefile: for printing a document

#         macro definitions

SOURCES = title intro tutorial reference appendix
PRINTER = Plw
MACROS  = ms

#         target definitions

print: troff.output
        lpr -$(PRINTER) -t troff.output &

troff.output: $(SOURCES)
        tbl $(SOURCES) | eqn | troff -t -$(MACROS) > troff.output
```

A change to the list of sources, the printer, or the macro package can be made in one place and take effect throughout the makefile. For large and complex procedures, this is a big advantage.

By placing the `troff` output in an intermediate file,[50] you can avoid having to reformat the document every time you want to print a copy. By making `print` depend upon the file `troff.output`, you can be sure that you always get the latest formatted version.

[50] `troff` intermediate output files are *not* text files. They will produce strange results if you try to look at them on the screen, and they should *not* be placed under `sccs`. It would be a good idea to put the source files under `sccs` instead.

By making `troff.output` depend on the list of sources (the expansion of the `$(SOURCES)` macro), you can be sure that when you change any one of the sources, `make` will rebuild `troff.output`, and the change will be reflected when you print the document.

Selecting A Target

You can select any target in the makefile by specifying it as an argument to `make` on the command line. If a target does not appear in the list of dependencies for the target you select (or the first target by default) `make` will not perform it. So, you can record several independent procedures within the same makefile. For example, this makefile can be used either to put new source files under `sccs`, or to print a finished document.

Figure 6-20 *A Makefile with Independent Procedures*

```
# Makefile: for printing a document
#           and putting sources under SCCS

#         macro definitions

SOURCES = title intro tutorial reference appendix
PRINTER = Plw
MACROS  = ms

#         target definitions

print: troff.output
        lpr -$(PRINTER) -t troff.output &

troff.output: $(SOURCES)
        tbl $(SOURCES) | eqn | troff -t -$(MACROS) > troff.output

# -----------------------------------------------------------

put.under: SCCS
# the next three lines begin with a tab
        -sccs create `ls | grep -v troff.output`
        -rm ,*
        -sccs get *

SCCS:
        mkdir SCCS
        chmod 775 SCCS .
```

Using this makefile, if you type in `make` (or `make print`), you will get the document (typing `make` does everything in the Makefile). If you type in

```
make put.under
```

your sources will be put under `sccs`.

6.5. Managing Disk Storage

Space on the disk is a limited resource. So, it is a good idea to keep track of how much space you use, especially if your system is running with disk quotas.[51]

SunOS provides facilities to monitor your disk usage and locate big directories that are candidates for housekeeping. Even so, it can be unwise to delete old files willy-nilly. You never know what gems you may have socked away there. So, the system also provides a facility to make tape archives of important files. Tape archives are especially good for large files that you need to keep but don't often use. If you make a tape archive before cleaning house, you can be sure that you won't lose anything important. You can use `df`, `du` and `ls -l` to locate such files, and you can use `tar` to move them onto a tape for storage offline, as described in the following sections.

Looking at Disk Usage with `df`

`df` shows you the amount of space used up on each disk that is *mounted* (directly accessible) to your system. It is very simple to use, just type

```
df
```

to see the capacity of each disk mounted on your system, the amount available, and the percentage of space already used up.

Figure 6-21 `df`

```
venus% df
Filesystem            kbytes    used   avail capacity  Mounted on
waldo:/export/root/donkey
                       75651   17302   50783    25%    /
waldo:/usr            106303   54120   41552    57%    /usr
krakow:/usr/krakow    547697  374274  118653    76%    /home/krakow
athens:/usr/athens    266107  212150   27346    89%    /home/athens
toupee:/usr/view      326031  213711   79716    73%    /home/view
salsa:/usr/salsa      352022  299302   72344    75%    /home/salsa
waldo:/home/waldo     371967  327280    7490    78%    /home/waldo
waldo:/export/share   106303   54120   41552    57%    /usr/share
popeye:/usr/games      54387   48649     299    89%    /usr/games
krakow:/usr/tools      39095   31396    3789    89%    /usr/tools
croaker:/usr/demo      62855   50736    5833    90%    /usr/demo
athens:/usr/doc       105843   87253    8005    82%    /usr/doc
```

Filesystems at or above 90% of capacity should be cleansed of unnecessary files. You can do this either by moving them to a disk or tape that is less full using, `cp`, and then remove them with `rm`. Or you can simply remove them outright. Of course, you should only perform housekeeping chores on files that you own.

[51] A disk quota is a limit on the amount of space (information) a user is allowed to use on the disk at any one time.

Directory Usage and `du`

You can use `du` to display the usage of a directory and all its subdirectories (in kilobytes).

`du` shows you the disk usage in each subdirectory. To get a list of subdirectories in a filesystem (disk), `cd` to the pathname associated with that filesystem, and run the following pipeline:

`du | sort -r -n`

For instance:

Figure 6-22 du

```
venus% du | sort -r -n
5314    .
1155    ./Documents.new
818     ./SCCS
234     ./Programs.new
230     ./Reference.new
204     ./Reference.old
123     ./Library.new
89      ./Library.old
87      ./Users.Guide.old
49      ./Reports.old
27      ./Documents.old
5       ./Programs.old
```

This pipeline, which uses the *reverse* and *numeric* options of `sort`, pinpoints large directories. Use `ls -l` to look at the size (in bytes), and modification times of files within each directory. Old files, or text files over 100K bytes, often warrant storage *off-line*.

6.6. Making a Tape Archive with `tar`

Sun386i users should refer to the *Sun386i SNAP Administration* manual for information on the SNAP backup procedure.

The simplest and most complete method to make a tape archive is to:

1. Mount a fresh tape on the tape drive. If you don't know how to do this, see your System Administrator or consult *System Administration for the Sun Workstation* for details.
2. `cd` to a directory you wish to archive. If you wish to archive an entire hierarchy of files, `cd` to the topmost directory in that hierarchy. `tar` will archive the directory and all its subdirectories.
3. Type in the `tar` command as follows:

   `tar -cvf` *drive*

   The `-c` option tells `tar` to *create* a new tape archive and overwrite the previous contents of the tape. The `v` stands for *verbose*. `tar` tells you everything that it is doing. The `f` tells `tar` to put the archive on the file (tape drive) *drive*. Your System Administrator can tell you the name of a tape drive to use.

Tapes can be reused. If you do not wish to overwrite the previous contents, you can use `-r` rather than `-c`. With `-r`, `tar` skips to the end of the previous archive, and then adds files onto the end. If you want to conserve space on the tape, you can use `-u`.[52] With `-u`, `tar` replaces files whose contents have changed with their newest version, adds new files onto the end, and leaves untouched files alone.

drive can be a diskfile. Since `tar` output takes up less space than do text files, a tape archive on disk can provide some space savings and a bit more convenience than using an actual tape. For even more space reduction, run the tape archive file, or *tarfile* through `compact`.[53]

Looking at the Contents of a Tape Archive

To examine the contents of a `tar` tape archive, use the `-t` option:

`tar -tvf` *drive*

To search for a specific file on the tape, pipe the output of `tar -t` through `grep`.

Extracting Files From a Tape Archive

To extract files from a tape archive, `cd` to the directory in which to place the file, mount the tape, and then use the `tar -x` option:

`tar -xvf` *drive filename ...*

If you omit filename, `tar` extracts the contents of the entire tape. If you specify a *filename*, or a list of filenames, `tar` extracts the named file(s).

tar -xvf /dev/rst0 ./letters/little

---

[52] The `-r` and `-u` options do not work with quarter-inch cassettes. They only work with half-inch tape drives. See the `mt` command for quarter-inch tapes.

[53] The command `uncompact` restores the tarfile to its original state, and you can then use `tar` to retrieve files from within the tarfile just like you would from a tape drive.

7

More About Printing

7

More About Printing

In *Getting Started with SunOS: Beginner's Guide* you learned how to print a file. Printers are often in high demand, and are normally shared by a number of people. To keep things running smoothly, the system feeds each request to the printer on a first-come first-served basis. Requests that are waiting are kept in the print *queue*.

7.1. Looking at the Queue with `lpq`

To look at the queue on the printer you normally use, type in

```
lpq
```

(short for "line printer queue"). If the queue is empty, `lpq` will respond with:

```
no entries
```

If there are some entries, `lpq` will list them for you and indicate which one is currently being printed.

Figure 7-1 *The* `lpq` *Command*

```
venus% lpq
Rank   Owner      Job  Files                    Total Size
active sam        18   standard input           39668 bytes
1st    sam        19   document                443820 bytes
2nd    joe        20   program.listing          32833 bytes
...
```

`lpq` will also tell you if there's some problem printing out your file. Some of the error messages `lpq` gives are described in the *SunOS Reference Manual*, or you can type **`man lpq`** for the information.

7.2. Removing Printer Jobs with `lprm`

If you decide not to print a job after all, you can remove it from the queue by typing in `lprm` followed by the job number, as shown by `lpq`:

Figure 7-2 *Removing Files with* `lprm`

```
venus% lprm 19
dfA019venus dequeued
cfA019venus dequeued
venus% lpq
Rank   Owner      Job  Files                    Total Size
active sam        18   standard input           39668 bytes
1st    joe        20   program.listing          32833 bytes
```

To remove all your jobs from the queue, use the – option:

```
lprm -
```

7.3. Selecting a Printer `lpr -P`

If the line for the printer is too long and there is another printer available to your system, you can direct jobs to that other printer with the –P*printer* option of `lpr`. Your System Administrator can tell you the names of other printers that you can use. `lpq` and `lprm` also accept this argument.

Figure 7-3 *Selecting a Printer*

```
venus% lpr -Plaserwriter memo
venus%  lpq -Plaserwriter
Rank   Owner      Job  Files                    Total Size
active jd          98  standard input          559668 bytes
active jenny       99  memo                      2077 bytes
active louisf     100  letter                   57320 bytes
active sam        115  document                621633 bytes
venus% lprm -Plaserwriter 115
lrhost: dfA115venus dequeued
lrhost: cfA115venus dequeued
```

7.4. Printing `troff` Output Files with `lpr -t`

To print `troff` output files, use the `-t` option of `lpr`.

```
lpr -t troff.output
```

7.5. Printing Screen Dumps

If you want to capture an image of the workstation screen on paper, use the following pipeline:

```
screendump | rastrepl | lpr -v &
```

`screendump` captures the image dot-for-dot, `rastrepl` increases its size, and the `-v` option of `lpr` prints the resulting image. There is significant computation involved in each of these steps, so be sure to run this pipeline in the background.

7.6. Printing Other Graphics Displays

`lpr` will print out a variety of graphics displays, depending upon the capabilities of the printer you use. For more information, consult the *SunOS Reference Manual*, and your System Administrator.

A

Glossary

A

Glossary

angle-brackets
Term for the characters < and >.

append
To add text or data onto the end of a file.

archive
A copy of a file or set of files, usually on tape, made for historical purposes or for long-term storage.

background
A process that is running, but does not have control of the terminal from which it was started, is said to be running in the background.

braces
Term for the characters `{` and `}`.

brackets
Term for the characters `[` and `]`.

builtin
Adjective for a command that is part of a particular shell; it is literally "built in" to the shell software. Such commands are only available when using the particular shell that supports them. Contrast this with such commands as `ls`, which are available for use with either shell.

C shell
A command interpreter for SunOS that provides filename substitution, alias substitution, a history mechanism, variable substitution, command (output) substitution, and job control. The C shell can interpret commands directly from the terminal, or from command files with a syntax modeled after the C programming language.

child process
A process started from within a shell or another process.

contents
The text or data contained in a file.

daemon
A process which runs in the background, usually invisible to the user. Daemons perform routine maintenance and low-level functions, such as

queueing up files for the printer and sending mail.

default
An assumed value, or an action taken when you omit an argument, command, or value.

dependency
A step within a procedure upon which a subsequent step depends. The step must be completed before the latter can be performed properly. `make` uses this notion to organize sets of SunOS commands, and do the minimum amount of work required to perform a task or bring a set of object-files up to date.

device
Typically a hardware peripheral supported by the system, and the software that controls it. May also be a specialized software program. SunOS treats a device as if it were a file. The programs that operate peripheral devices reside in the directory `/dev`.

directory
A type of file that contains names and access information about other files, including other directories. Directories are organized in a hierarchy, the root of which is named `/`.

drive
(*tape drive* or *disk drive*). The hardware that performs the physical transfer of data from the system onto a tape or disk, and vice-versa.

embedded
Contained within a file, within a line of text, or within a word. Usually applied to commands or symbols that are surrounded by ordinary characters.

encrypt
To encode or scramble data to prevent unauthorized reading.

environment
General: to the extent that an interactive program can be customized, the values of the various options, settings, and variables that are currently in effect. Technical: the set of data inherited from the parent process and/or passed along to child processes.

escape
A character, usually a backslash, indicating that the character following it is to be interpreted as plain text, rather than as a symbol having special meaning.

event
In history substitution: the text of a command-line contained in the history list.

execute
To perform a set of instructions or program.

expansion
The value of a variable or macro. For instance, in the C shell the expansion

of the character ~ is the pathname of the user's home directory.

filename
The name of a file, directory, or device.

file
A portion of a mass-storage memory device, typically a disk, containing a specific, named set of data. Generalized to include any source from which data can be received or transferred within the system.

file type
A field in the permissions column of the `ls -l` display that indicates whether the file is a plain disk file, a directory, a device, or a symbolic link.

filter
A command or program that accepts text from the standard input, applies a transformation rule (or rules) to that text, and produces text on the standard output.

foreground
The process that has control of the terminal is said to be running in the foreground. Processes that do not control the terminal are said to be running in the background, or they may be suspended.

fork
By a shell or command: to start a new process and wait for it to finish before proceeding.

group
A subset of users with access to the system. Members of a group may be granted more complete access to files than the public at large. The permissions that control group access to files.

job
A background process, running or stopped, under the control of the C shell.

key
A character string used to encode or decode a file by `crypt`.

link
A filename, or entry in a directory corresponding to a file. A *hard* link is a direct entry. A *symbolic* link is a string that contains the name of the file it is associated with.

macro
A string of text that is replaced by another, typically much longer, string when interpreted by a shell or program.

makefile
A file containing instructions for `make`. Typically named `makefile` or `Makefile`.

modification time
The date and time at which a file was last changed. A field in the directory entry for a file that can be altered directly using the `touch` command.

monitor (v.)
To maintain a record of changes to a file, to assure that only one user at a time can make changes, and to assure that the most recent version of a file can quickly be restored.

multitasking
Performing multiple tasks at once. The ability of the system to handle the work of several simultaneous users or windows.

noninteractive
A program that accepts no input from, and displays no output on, the terminal.

object-file
A file containing the output, typically not text, of a compiler, plotting program, or other such program.

off-line
Disconnected from the system.

operation
The action of the system or program to accept input, transform data, and produce output.

owner
The user to whom a file belongs, who can alter its name, access permissions, and other attributes.

pattern
A string that includes special characters that, when interpreted, correspond to a set of possible text strings.

parent directory
A directory containing the current directory, or directory of interest.

parent process
A process, from which the current process of interest was started.

permissions
Attributes of a file that determine whether a specific user has access to read, write on (or delete), or execute (use as a command), a file.

pipe
The vertical bar character |. The mechanism by which the system passes the output of one command as direct input to another command.

pipeline
A set of commands connected by pipes. The intermediate commands are typically *filters*.

process
General: A command that is being performed by the system. Each process has a unique number. The mechanism by which the system keeps track of a single task among the many requested of it at any given time. Technical: a set of instructions and data under the control of the system's memory

management facilities.

public
The entire set of users who have access to the system. The permissions that control public access to files.

range
A set of characters specified by the first and items in a list. For instance, the entire upper-case alphabet can be specified as: `A-Z`.

redirect
The standard input, standard output, and standard error output of a command is normally received by, or sent to, the terminal. To explicitly indicate a file from which, or to which the command is to send or received data using symbols such as > and <.

regular expression
The method for specifying search patterns for `grep`, and editors such as `vi`.

resources
Refers to the computation capacity and speed, available memory, (and sometimes the peripheral devices) available to the system.

return code
The value returned (to its parent) by a process upon completion.

robust
Programs: Able to perform reliably under a variety of conditions, or with a variety of (possibly unexpected) data. Syntax: The degree to which a set of rules allows for expression of a wide range of information.

routine
A set of commands or instructions that together perform a complete task.

s-file
An `sccs` history file in the `SCCS` subdirectory.

shell
A programmable command interpreter.

size
The number of characters in a text file.

standard error
The channel through which a command sends diagnostic messages.

standard input
The channel through which a command receives data.

standard output
The channel through which a command sends results.

state
The current condition of a process.

string
A set of characters terminated on either end by a tab, space, newline, or

other delimiting character.

subdirectory
A directory that resides within another. For instance, `/usr` is a subdirectory of `/`.

subshell
A shell invoked from within another shell or program.

superuser
A mnemonic for the `su` command, which allows a user to temporarily adopt the ID of another user on the system, typically `root`. Also a term for the Operator or System Administrator's userid, `root`.

rule
A list of SunOS commands for `make` to perform in order to complete a step, or produce a target file.

syntax
General: the format for a legal command and its arguments. Technical: the rules by which input is interpreted.

target
An object file to be produced, or label for a list of SunOS commands to be performed, by `make`.

user
A person with an account on the system who can log in, issue commands, and create files.

userid
The login name, or ID number assigned to each user by the system administrator.

variable
A named location in which a data value (or list of values) is temporarily stored in memory.

B

C Shell Scripts

B

C Shell Scripts

You can put a sequence of SunOS commands in a file called a *script*. By using the `source` *filename* command, or by setting the execute permissions and typing in the filename as if it were a command, you can tell the C shell to read and perform commands in the file.

NOTE *We recommend that you use the Bourne shell for writing shell scripts. The Bourne shell has a simpler command syntax, faster execution time, and provides better security. Refer to Appendix D for information about writing Bourne shell scripts.*

This appendix outlines features that you can use when writing scripts for the C shell.

C Shell Invocation

C shell scripts do not serve the same function as `make`, which is useful for consistently performing a set of operations on related files. While scripts can be written to do this, the C shell is more general in scope. Scripts do not check for dependencies, for instance. And, there are many things that you can do with scripts, such as prompting for input from the terminal, that are not practical using `make`.

When a script is invoked by name, the system looks at the very first line of the file to decide how to run it:

- If the first line of the script starts with a `#!`, followed by the name of a program, the system uses that program to perform commands in the script.
- If the first line starts with a # (hash sign), the system uses the C shell to run the script.
- If the first line does *not* start with a # (hash sign), the system uses the Bourne shell to run the script.

To run a script with no C shell startup processing, the first line should be of the form:

Figure B-1 *Starting a C Shell Script*

```
#! csh -f script
```

Command-Line Arguments in Scripts

To pass command-line arguments as parameters to a script, type its name, followed by any arguments you wish. The C shell places words following the name in the variable `argv`, the *arguments list*. Command-line arguments are treated as words contained in this variable, or you can use the equivalent variables: `$1` through `$`*n* where *n* is the number of arguments in the list.

Variables in Scripts

A number of notations are available for accessing words in variables, and other variable attributes. The notation:

```
$?name
```

expands to `1` if a named variable exists (using the `set` command), or to `0` otherwise.

Figure B-2 *The* `$?` *Notation*

```
venus% set var=(a b c)
venus% echo $?var
1
venus% unset var
venus% echo $?var
0
```

All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the count of words in the variable *name*:

Figure B-3 *The* `$#` *Notation*

```
venus% set var=(a b c)
venus% echo $#var
3
venus% unset var
venus% echo $#var
var: Undefined variable.
```

There is a special C shell variable, `$$`, which represents the process number of the shell itself. Why would you want a variable like this? Because the shell's process number is unique on the system, you can use it as part of a file's name if you want to create unique temporary files from inside the shell. Part of your script might create a file called `/tmp.$$`, for example; that file will not be confused with any other which already exists.

The redirection characters:

```
$<
```

indicate that a line is to be read from the terminal. To write out the prompt `yes or no?` without a newline and then read the answer into the variable `a`:

```
echo -n "yes or no?"
set a=($<)
```

In this case `$#a` would be `0` if either a blank line or [Ctrl-D] were typed in response.

A minor difference between `$`*n* and `$argv[`*n* `]` is that `$argv[`*n* `]` yields an error if *n* is larger than the word count `$#argv`, while `$n` never yields a subscript-out-of-range error. This is for compatibility with older shells.

It is never an error to give a subrange of the form *var*`[`*n*–`]`. If there are less than *n* words in the given variable, then no words are selected.

A range of the form *var*`[`*m*–*n*`]` likewise returns a value without an error, even when *m* exceeds the number of words, provided that *n* is in range.

Expressions

All of the arithmetic operations of the C language are available in the C shell with the same precedence that they have in C. These operations are useful for evaluating expressions in branches and loops. The operations == and != compare strings, and the operators `&&` and `||` implement the logical *and* and *or* operations, respectively. The operators =~ and !~ are similar to == and !=, allowing for pattern matching as with filename substitution.

File Enquiries

The expression:

`-e` *filename*

returns `1` if the file exists, and `0` otherwise. Similar primitives provide other tests:

`-r` `1` if read-access is allowed for the user running the script.

`-w` `1` if write-access is allowed for the user.

`-x` `1` if execute-access is allowed.

`-o` `1` if the user owns the file.

`-z` `1` if the file has zero length.

`-f` `1` if a plain file.

`-d` `1` if a directory.

Pathname Processing Primitives

There are also primitives to apply to pathnames to strip off unneeded components:

`:t` (*tail*) removes all but the rightmost component (or *basename*) of the pathname.

`:r` (*root*) removes suffixes beginning with a dot (`.`).

`:e` (*end*) removes prefixes ending with a dot.

`:h` (*head*) removes the last component, leaving the pathname of the directory in which the file resides.

Here's an example of how these apply to a file:

If you had a file called `/usr/include/sys/types.h`, then `:t` would remove all but `types.h`; `:r` would leave you with `/usr/include/sys/types`; `:e` would leave you with just `h`; and `:h` would give you `/usr/include/sys`.

Return Codes

It is possible to test whether a command terminates normally by using a primitive of the form { *command* }, which returns 1 if the command exits normally (with exit status 0), or 0 if the command terminates abnormally (with a nonzero return code).

If more detailed information about the status of a command is required, it can be executed and the variable `status` examined in the next command. Since every command returns a value to `status`, you must save values of interest on the very next line of the script:

`set` *checkpoint*`=$status`

where *checkpoint* is a suitable variable name.

Sample C Shell Script

The following script, `copyc`, copies files named as arguments into a backup directory:

Figure B-4 *A Sample C Shell Script*

```
#
# copyc copies files named on the command line
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup. . . not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

Basic Control Structures: `if` and `foreach`

This script uses the `foreach` command, which causes the C shell to execute the commands between it and the corresponding `end` with the named variable taking on each of the values given between (and). The named variable — in this case `i` — is set to successive words in the list. Within this loop you can use the `break` command to stop executing the loop and `continue` to terminate one iteration and begin the next. After the `foreach` loop, the iteration variable (`i` in this case) has the value it had during the last iteration.

The variable `noglob` is set to prevent filename expansion from being performed on members of `argv`. This is a good idea, in general, if the arguments to a C shell script are filenames that have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a `$` variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form:

```
if ( expression ) then
command
...
endif
```

The placement of the keywords here is *not* flexible. The word `then` *must* appear on the same line as `if`, when used with a block of commands.

The C shell does *not* accept the formats:

```
if ( expression )
then
```

or

```
if ( expression ) then command endif
```

For individual conditional commands, the C shell has another form of the `if` statement:

```
if ( expression ) command
```

which can also be written as

```
if ( expression ) \
     command
```

The newline is escaped here for the sake of appearance. The command must not involve `|` , `&` or `;` and must not be another control command. The final \ must immediately precede the end-of-line. This is the only form of the `if` command that can be used within an alias definition.

The more general `if` statement also admits a sequence of `else-if` pairs followed by a single `else` and an `endif`.

```
if ( expression ) then
     commands
else if ( expression ) then
     commands
...
else
     commands
endif
```

Introducing Comments with

The character # introduces a C shell comment in a script (but not from the terminal), and the C shell ignores all subsequent characters the line.

Other C Shell Control Structures

The C shell also has the control structures `while` and `switch` that are similar to those in C.

```
while ( expression )
    commands
end
```

and

```
switch ( word )

case str_1:
    commands
    breaksw

  . . .

case str_n:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

See the *csh* manual page for details. C programmers should note that `breaksw` exits from a `switch`, while `break` exits a `while` or `foreach` loop.

Finally, `csh` allows a `goto` statement, with labels looking as they do in C, that is:

```
loop:
    commands
    goto loop
```

Here Documents

A *here document* is a special notation used to pass instruction along to commands that normally run interactively. The here document begins with a <<*eot* and ends with a line containing *eot* by itself. *eot* can be any string.

Here is a script that runs `ed` to delete leading blanks from every line in each file in the argument list. In this case, the *eot* string is "woof":

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'woof'
1,$s/^[  ]*//
w
q
'woof'
end
```

(The brackets in the script contain a tab and a space.)

The notation << `'woof'` means that the standard input for the `ed` command is the text in the C shell script file up to the next line consisting of exactly `'woof'`. The fact that the `woof` is enclosed in quote characters prevents the C shell from substituting variables on the intervening lines. In general, the C shell uses the word following << to terminate the text to be given to the command. If any part of the word following the << is quoted, these substitutions are not performed. In this case, since the form `1,$` was used in the editor script, you needed to ensure that the `$` is not variable-substituted. You can also ensure this by preceding the `$` here with a \, for instance:

```
1,\$s/^[ ]*//
```

but quoting the `woof` terminator is a more reliable way of achieving the same effect.

Catching Interrupts with `onintr`

If your script creates temporary files, you can use `onintr` to catch interrupts, so that the script can delete them before halting.

`onintr` *label*

where *label* is a label in your program that is followed by your housekeeping commands. If the C shell receives an interrupt, it performs a `goto` *label*, and executes those commands.

Exit

You can also use the `exit` command (which is built in to the C shell) to terminate the script. If you wish to exit with a nonzero status, do the following:

`exit(` *status* `)`

where *status* is the status you want to exit with.

C

C Shell Special Characters

C

C Shell Special Characters

Characters with special meaning to the C shell:

?	Single character wild card.
*	String wild card, zero or more characters.
.	Abbreviation for current working directory.
..	Abbreviation for the parent of the current directory.
~	Abbreviation for your home directory.
~ *user*	Abbreviation for the home directory of *user*.
[...]	Matches any single character listed within the brackets.
[*x* – *y*]	Matches any character within the range of *x* and *y*.
{*str*, ...}	*Grouping*. Matches each *str* successively. Filename substitution is applied to each *str* before matching occurs. Thus, `{x,*y*,?z*}` matches a filename `x`, all filenames containing the letter `y`, and all filenames having `z` as the second character. Groups enclosed with braces can be nested.
&	Places the command in the background.
Ctrl-Z	Stops the foreground job, placing it stopped in the background.
%[*n*]	Brings the current (stopped) job, or the specified background job to the foreground.
%[*n*] &	Continues, in the background, the current or specified stopped job.

\>*filename*

Redirects the standard output to *filename*. If *filename* already exists, its previous contents are lost. When set, the shell variable `noclobber` prevents redirection to existing files or character special devices.

\>! *filename*

Forces the standard output to *filename*, even when `noclobber` is set.

>& *filename*
Routes diagnostic (standard error) output to *filename*, along with the standard output.

>& ! *filename*
Forces diagnostic and standard output to *filename*.

>> *filename*
Appends the standard output to *filename*. When `noclobber` is set, the file must already exist.

>> ! *filename*
Forces the standard output to *filename*, even when `noclobber` is set. Creates a new file if necessary.

>>& *filename*
Appends the diagnostic as well as standard output to *filename*. When `noclobber` is set, the file must already exist.

>>& ! *filename*
Forces appending of diagnostic and standard output to *filename*, even when `noclobber` is set.

cmd **|** *cmd*
Pipe. Uses the standard output of the left-hand *cmd* as standard input for the right-hand *cmd*.

cmd **|&** *cmd*
Uses both standard and diagnostic output of the left-hand *cmd* as standard input for the right-hand *cmd*.

(...) *Command grouping*. Commands and pipelines surrounded by parentheses are executed in a subshell and treated as a unit by the current C shell.

(...) >& *filename*
Redirects the standard output (if any) and the diagnostic output of the enclosed command(s) to *filename*. This is especially useful if the enclosed commands redirect the standard output to a file (thus sending the standard output and the standard error to separate destinations).

< *filename*
Opens *filename* as the standard input.

cmd **<<** *word*
(*Here document*). Indicates that a command (typically interactive) is to accept *its* commands from the same device or file (usually a script) as the shell. *word* is interpreted literally as the *end-of-input* mark for the command. The C shell parses, but does not execute, each text line between the here document and a line containing *word* by itself. After applying command, filename, and variable substitution, the C shell passes each line on to *cmd*. To suppress all substitution, include a \, ", or ′ in *word*.

;	Separates commands on one input line.
\	At the end of a line, escapes the newline character and continues the command to the next input line.
\	Escape the special meaning of the character it precedes.
´. . . ´	The C shell treats the enclosed text as one word, preventing history and variable substitution.
". . . "	The C shell treats the enclosed text as one word, breaking words only at enclosed newlines.[54] History and variable substitution is performed *before* escape characters are interpreted.
\`*command* \`	Replaces the backquoted command or pipeline (including the backquote marks) with its output. Output is broken into words at blanks, tabs and newlines, except for the final newline. Unless the right-hand backquote is followed by a space, the last word of the substitution is prepended to the following word on the command line.

Escaped history substitution event designators and word designators (described below) can be used to indicate command line arguments within an alias definition.

^*l*^*r*[^]	Substitutes the string *r* for the string *l* in the previous command line. The final ^ is required only if history substitution modifiers are appended.
!	Begins a history substitution. To escape its special meaning, precede the ! with a backslash (\). A ! is also escaped when followed by a blank, tab, newline, (or =.

The following designators select an event (command line) from the history list. Word designators and modifiers can be appended for command-line editing.

!!	The previous command.
!*n*	Command line number *n*.
! −*n*	Selects the event whose number is *n* less than the current one.
!*str*	The most recent command beginning with *str*.
!?*str*[?]	The most recent command containing *str*. The closing question mark is only required when word designators or modifiers are appended.
!*	All arguments from the previous command, but not argument zero (the command name).

[54] An *enclosed newline* is a carriage return within quotes; ie., an *escaped* newline.

`!^`	The first *argument* from the previous command. If, for instance, the command was `echo first`, then `!^` would expand to `first`.
`!$`	The last argument from the previous command.
`!:`*n*	The *n*'th argument from the previous command.
`!#`	The contents of the *current* command line typed in so far.
`!{`*str*`}` ...	Restrict the event designation to *str*; text following the brackets is appended to the last word of the expansion *after* substitution takes place.

Word designators can be appended to the history substitution character (`!` for the previous event) to a quick substitution, or to an event designator.

`:*`	All arguments, except argument zero.
`:^`	The first argument .
`:$`	The last argument.
`:`*n*	The *n*'th argument.
`:%`	The word matched by most recent `!?` search.
`:`*x*–*y*	Argument *x* through argument *y*.
`:-`*y*	abbreviates `:0-`*y*.
`:`*x*`*`	Argument *x* through the last argument.
`:`*x*`-`	Argument *x* through the next-to-last argument.
`:#`	The contents of the *current* command line typed in so far.

The following modifiers can be used in any sequence to modify a selected event or word. A colon is required to separate modifier(s) from event or word designators.

[:]**p**	Prints the new command but does not execute it.
[:]**h**	Removes a trailing pathname component, leaving the head.
[:]**t**	Removes all leading pathname components, leaving the tail.
[:]**r**	Removes a filename extension (*.xxx*).
[:]**e**	Removes all but the extension.
[:]**s**/*l*/*r*/	Substitutes *r* for *l*. *l* is a literal string, not a regular expression. Any character may be used as the delimiter in place of `/`. The character `&` in the right hand side is replaced by the left hand string. A null *l* uses the previous string either from a *l* or from a `?` event search.
[:]**&**	Repeats the previous substitution.
[:]**q**	Quotes the substituted words, preventing further substitutions.

[:]x	Like `:q`, but breaks words at blanks, tabs and newlines.
:g*m*...	*Global prefix.* When prefixed any of the above modifiers, *m*, the modifier(s) apply to all words in the specified event. Normally, each word must be modified separately.

After the input line is aliased and parsed, and before each command is executed, the C shell performs variable substitution on words that start with an unescaped `$`, according to the list below. A `$` is escaped by preceding it with a backslash (`\`), or when followed by a blank, tab, or end-of-line.

Shell variables have names consisting of up to 20 letters, digits and underscore characters, starting with a letter.

Environment variables can be expanded but not modified.

$*var* Is replaced with the value of *var*.

${*var***}** ... The brackets indicate that the enclosed string is the variable name. The value of the named variable is prepended to the text that follows on the command line.

${*var***[***selector***]}**

Select words from within *var*. *selector* can be one of:

n	a number.
x – *y*	two numbers separated by a – to specify a range.
x –	Word *x* through the last word.
– *y*	The first word through word *y*.
*****	all words in the value.
$*var*	the value of another variable, in which case variable substitution is applied to the *selector* first, and then to the entire word.

$#*var*	The number of words in the variable.
${#*var***}**	Same as `$#`*var*
$0	The name of the file from which command input is being read. An error occurs if the name is not known.
$*n*	The *n*'th word in the argument list; equivalent to `$argv[`*n*`]`.
${*n***}**	Same as `$`*n*.
$*	All words in the argument list; equivalent to `$argv[*]`.
$?*var*	
${?*var***}**	replaced with `1` if *var* is set, or `0` if not.
$?0	replaced with `1` if the current input filename is known, `0`, otherwise.
$$	Is replaced with the process ID (PID) of the (parent) shell.

$< replaced with text taken from the standard input, with no further interpretation. Used to read from the keyboard in a C shell script.

The modifiers `[:]h`, `[:]t`, `[:]r`, `[:]q`, and `[:]x` can be applied to the substitutions above. See *Modifiers* under *History Substitution*, above, for a description.

If braces `{...}` appear in the variable substitution, modifiers must be enclosed within them.

The current implementation allows only one modifier within each variable substitution.

The following variable substitutions can not be modified: **$?**, **$$**, and **$<**.

Expressions appear within the `@`, `exit`, `if`, and `while` builtin commands.

Null or missing terms are interpreted as `0`.

Results of all expressions are *strings* that represent decimal numbers. Results of logical expressions are `1` (for true) or `0` (for false).

(...) Parentheses indicate grouping of operators and terms within an expression, overriding the standard precedence of operators.

== True if the string on the left is equal to the string on the right (after all substitutions are performed).

!= True if the string on the left is not equal to the string on the right.

=~ True if the string on the left is matched by the pattern on the right.

!~ True if the string on the left is not matched by the pattern on the right.

< True if the number on the left is less than the number on the right.

<= True if the number on the left is less than or equal to the number on the right.

> True if the number on the left is greater than the number on the right.

>= True if the number on the left is greater than or equal to the number on the right.

|| Logical *or* connective.

&& Logical *and* connective.

{...} *Command successful.* True if the command surrounded by brackets exits with status code `0`.

An operator of the form

flag filename
is true if the attribute *flag* applies to *filename*, with respect to the current user. *flag* can be one of:

-r read access

`-w`	write access
`-x`	execute access
`-e`	existence
`-o`	ownership
`-z`	zero size
`-f`	plain file
`-d`	directory
`!` *flag*	true if *flag* does *not* apply.

If the file does not exist, or is inaccessible, then all inquiries yield false as a result.

`+`	Addition.
`–`	Subtraction.
`*`	Multiplication.
`/`	Division.
`%`	Remainder after division.
`0`*str*	A string with a leading zero is interpreted as an octal numeral.
`<<`	Bitwise *shift left* operator.
`>>`	Bitwise *shift right* operator.
`|`	Bitwise *or* operator.
`^`	Bitwise *exclusive or* operator.
`&`	Bitwise *and* operator.

D

Bourne Shell Scripts

D

Bourne Shell Scripts

You can use the Bourne shell to perform a set of SunOS commands contained in a file called a *script.*

Bourne shell scripts do not serve the same function as `make`, which is useful for consistently performing a set of operations on related files. While scripts can be written to do this, the Bourne shell is more general in scope. Scripts do not check for dependencies, for instance. And, there are many things that you can do with scripts, such as prompting for input from the terminal, that are not practical using `make`.

To run a Bourne shell script (for which you have execute permission), type in its filename as if it were a command. When you do, the system looks at the very first line of the file to decide which Shell should run the script:

- If the first line does *not* start with a # (hash sign), the system uses the Bourne shell to run the script.
- If the first line starts with a # (hash sign) and is *not* followed by a `!` (exclamation mark), the system uses the C shell to run the script.
- Finally, if the first line of the Shell script starts with a `#!` combination and is followed immediately by a name, the system looks for a program of that name to run the Shell script. If you supply arguments on the command line, these are passed along to variables in the Bourne shell called *arguments.* The first argument after the name of the script is placed in variable `1`. The second is placed in variable `2`, and so forth.

NOTE *You can often simplify testing of Bourne shell scripts (or commands to run within them) by using the Bourne shell interactively. To do so, type in the command* `/bin/sh`, *and enter commands as described in this Appendix. Use* [Ctrl-D] *to exit and return to the C shell. Most of the examples below make use of the Bourne shell interactively, as well as within scripts.*

Bourne Shell Variables

The Bourne shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. You may assign values to variables by writing the variables name, an equal sign, and a value (with no spaces between). For example:

```
$ user=fred box=m000 acct=mh0000
```

assigns values to the variables *user, box* and *acct.* To set a variable to the null string, you can say:

```
$ cheese=
```

The value of a variable is substituted by preceding its name with $ — for example:

```
$ name=fred
$ echo $name
fred
$
```

You can use variables to provide abbreviations for strings that are used frequently throughout a script. A script containing the following lines

```
b=/home/fred/bin
...
mv pgm $b
```

moves the file *pgm* from the current directory to the directory */home/fred/bin*. A more general notation is available for parameter (or variable) substitution, as in:

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed immediately by a letter or digit:

```
tmp=/tmp/ps
ps >${tmp}a
```

directs the output of `ps` to the file `/tmp/psa`.

Variables can be concatenated onto each other. If the variable `x` is set to *hello*, then `$x.foo` will be equal to *hello.foo*.

Bourne Shell Initial Variables

Except for `$?`, the variables defined in table D-1 are set initially by the Bourne shell. `$?` is set after executing each command.

Table D-1 *Variables Initialized by the Bourne Shell*

Variable	***Explanation***
`$?`	The exit status (return code) of the last command executed, as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned.
`$#`	The number of arguments(in decimal).
`$$`	The process number of this Shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary filenames. For example, `tmp.$$` will not be confused with any other file.
`$ !`	The process number of the last process run in the background (in decimal).
`$-`	The current Bourne shell flags, such as `-x` and `-v` .

Variables with Special Meaning to the Bourne Shell

Some variables have a special meaning to the Bourne shell; avoid them in general use.

`$MAIL` When the Bourne shell is used interactively, it looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at, the Bourne shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file `.profile` in your home directory. For example:

The file `.profile` in your home directory is the setup file for the Bourne shell — equivalent to the combination of the `.cshrc` and `.login` files for the C shell.

```
MAIL=/var/spool/mail/fred
```

`$HOME` Your home directory; this variable is also typically set in `.profile`.

`$PATH` A list of directories that contain commands (the *search path*). Each time the Bourne shell executes a command, a list of directories is searched for an executable file by that name. If `PATH` is not set, then the current directory, `/bin`, and `/usr/bin` are searched by default. `$PATH` consists of directory names separated by `:`. For example,

```
PATH=:/home/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first `:`), `/home/fred/bin`, `/bin`, and `/usr/bin` are to be searched in that order. This allows you to have your own private commands accessible independently of the current directory. If the command name contains a `/`, then this directory search is not used.

`$PS1` The primary Bourne shell prompt string, by default, '`$ `'.

`$PS2` The Bourne shell prompt when further input is needed, by default, '`> `'.

`$IFS` The set of characters to be interpreted as blanks when parsing command lines.

The `test` Command

Although the `test` command is not part of the Bourne shell, scripts frequently use it. `test` can be used to check on the status of files, to compare strings and algebraic expressions, and to perform integer calculations. For instance:

`test -f` *file*

returns zero exit status if *file* exists and non-zero exit status otherwise. In general `test` evaluates a predicate and returns the result as its exit status. Here is the list of things you can test for.

`-b` *file*	true if *file* exists and is a block special device.
`-c` *file*	true if *file* exists and is a character special device.
`-d` *file*	true if *file* exists exists and is a directory.
`-f` *file*	true if *file* exists and is not a directory.
`-g` *file*	true if *file* exists and is setgid.
`-h` *file*	true if *file* exists and is a symbolic link.
`-k` *file*	true if *file* exists and is sticky.
`-l` *string*	the length of *string*.
`-n` *string*	true if the length of *string* is nonzero.
`-r` *file*	true if *file* exists and is readable.
`-s` *file*	true if *file* exists and has a size greater than zero.
`-t [`*fildes*`]`	true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
`-w` *file*	true if *file* exists and is writable.
`-x` *file*	true if *file* exists and is executable.
`-z` *string*	true if the length of *string* is zero.
string-1 = *string-2*	true if the strings *string-1* and *string-2* are equal.
string-1 != *string-2*	true if the strings *string-1* and *string-2* are not equal.
string	true if *string* is not the null string.
n1 `-eq` *n2*	true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons `-ne`, `-gt`, `-ge`, `-lt`, or `-le` may be used in place of `-eq`, where `ne` means "not equal," `-ge` means "greater than or equal to," `-lt` means "less than," and so on.

`[ ... ]` *alternative form of the* `test` *command*

You can also call `test` by surrounding the expression to be tested with brackets (`[ ]`). (The left bracket is a command name, the right bracket is a argument signifying the end of the expression.) This form is most often used with the `if` command described later on.

Getting Started — A Simple Procedure

Here is a very simple Bourne shell script to look up names in a list of names and telephone numbers contained in a file called `names.list`. Let's call the lookup script `name`:

```
$ cat name
#! /bin/sh
grep -i $1 names.list
$
```

This is about as simple as you can get. Let's run the `name` procedure looking for people called *Ted*:

```
$ name ted
Ted Applehead            teda@seeds              7534
Ted Monsterpie           random@house            7256
$
```

Later on we will show a more sophisticated version of `name`, and expand on this procedure to demonstrate other features of the Bourne shell.

Control Flow in the Bourne Shell — `for`

A frequent use of Bourne shell procedures is to loop through the arguments (`$1, $2, . . .`) executing commands once for each argument. Here's an expanded version of the `name` procedure from above. The original version of `name` can only look for one person's name. Now we want to expand it to look for more than one name at a time. Let's look at the new version:

```
$ cat name
#! /bin/sh
for person
    do grep -i $person names.list
done
$
```

Here we set a variable called `person` to the value of each argument, one at a time, then we call out the value of `person` in the `grep` command. Now we can look for more than one name at a time:

```
$ name ben madge
Ben Tortcake             tort@icky               7258
Madge Hittite            celtics@garden          7214
$
```

General form of the `for` loop

The `for` loop notation is recognized by the Bourne shell and has the general form

`for` *name* **in** *w1 w2* ...
`do` *command-list*
`done`

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like do and done are only recognized following a newline or semicolon. *Name* is a Bourne shell variable that is set to the words *w1 w2* ... in turn each time the *command-list* following do is executed. If in *w1 w2* ... is omitted, then the loop is executed once for each argument; that is, in $* is assumed.

An example of the use of the for loop is the *create* command whose text is

```
for i do >$i; done
```

(Remember that cat >*filename* creates a file where none exists.)[55] The command:

```
$ create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. Use the notation >*file* on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before done.

Control Flow in the Bourne Shell — case

The case notation provides a multi-way branch.For example, suppose you wrote a script called append which contained the following lines:

```
case $# in
     1)  cat >>$1 ;;
     2)  cat >>$2 <$1 ;;
     *)  echo 'usage: append [ from ] to' ;;
esac
```

esac, you may have noticed, is *case* backwards.

When called with one argument as

```
$ append file
```

$# is the string "1" and the standard input is copied onto the end of *file* using the *cat* command. To append the contents of *file1* onto *file2*, say:

```
$ append file1 file2
$
```

If the number of arguments supplied to *append* is other than 1 or 2, a message is displayed indicating proper usage.

The general form of the case command is:

[55] In fact, in the Bourne shell, you don't need cat; typing >*filename* by itself creates a file.

```
case word in
     pattern-1) command-list-1;;
     pattern-2) command-list-2;;
     . . .
esac
```

The Bourne shell attempts to match *word* with each *pattern*, in the order in which the patterns appear. If a match is found the associated *command-list* is executed, and execution of the `case` is complete. Since * is the pattern that matches any string, you can use it for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example the commands following the second * are never executed.

```
case $# in
     *) . . . ;;
     *) . . . ;;
esac
```

Another example of the use of the `case` construction is to distinguish between different forms of an argument. The following example is a fragment of a `cc` (C compiler) command:

```
for i
do case $i in
     -[ocs]) . . . ;;
     -*) echo 'unknown flag $i' ;;
     *.c)    /lib/c0 $i . . . ;;
     *)  echo 'unexpected argument $i' ;;
     esac
done
```

What does this do? It checks for the options (or *flags*) `-o`, `-c`, or `-s`; if it gets some other flag, it reports it as unknown. It checks to see if it gets a file ending in `.c` and processes it when it does; if it gets anything else it reports an unexpected argument.

Matching Multiple Patterns in One Case

To allow the same commands to be associated with more than one pattern the `case` command provides for alternative patterns separated by a '|'. For example:

```
case $i in
     -x|-y)  . . .
esac
```

is equivalent to

```
case $i in
     -[xy])  . . .
esac
```

The usual quoting conventions apply, so that

```
case $i in
     \?)    . . .
```

will match the character `?`.

Here Documents in the Bourne Shell

Sometimes a Shell procedure requires data. Instead of having the data in some file somewhere in the system, the data can be included as part of the Shell procedure. Such a collection of data is called a *here document* — the data (document) is right *here* in the Shell procedure. One advantage of a here document is that Shell parameters can be substituted in the document as the Shell is reading the data.

The general form of a here document is like this:

```
        lines of Shell commands
            . . .
command-name << end-marker
lines of data
belonging to the
here document
            . . .
end-marker
            . . .
        more lines of Shell commands
```

The `name` Command Using Here Document

Let's revisit the `name` procedure discussed in earlier sections. Instead of having the names and numbers in one file and the Shell procedure in another file, you can keep both the procedure and the list in the same file — that is, in the procedure. Here's another version of the `name` command:

```
$ cat name
#! /bin/sh
grep -i $1 <<woof
Ted Applehead           teda@seeds          7534
Bernice Barns           boat@carib          7441
    . . .
    more names
    . . .
David Smiter            acme@nadir          7435
Ben Tortcake            tort@icky           7258
Dave von Noknock        dave@dove           7296
woof
$
```

In this example the Bourne shell takes the lines between `<<woof` and `woof` as the standard input for *grep*. The string `woof` is arbitrary, the document being terminated by a line that consists of the string following <<.

Now you'll notice that in *this* version of `name` we're back to being able to only look up one name at a time. We *could* combine the multiple-name version with the here-document version:

```
$ cat name
#! /bin/sh
for person
    do grep -i $person <<woof
Ted Applehead             teda@seeds              7534
Bernice Barns             boat@carib              7441
    . . .
    more names
    . . .
David Smiter              acme@nadir              7435
Ben Tortcake              tort@icky               7258
Dave von Noknock          dave@dove               7296
woof
done
$
```

The problem with this approach is that the Shell reads up the list of names every time around the `for` loop. This could become excruciatingly slow. In a later section we show another version of `name` using temporary files for faster performance.

Parameter Substitution in Here Documents

Parameters are substituted in the here document before it is made available to whatever command as illustrated by the following procedure called `edg` (`ed` globally).

```
ed $3 <<woof
g/$1/s//$2/g
w
woof
```

Then the command line:

```
$ edg string1 string2 file
```

is equivalent to the command:

```
$ ed file
g/string1/s//string2/g
w
```

and changes all occurrences of *string1* in *file* to *string2*. You can prevent substitution by using \ to quote the special character `$` as in

```
ed $3 <<woof
1,\$s/$1/$2/g
w
woof
```

This version of *edg* is equivalent to the first except that *ed* displays a `?` if there are no occurrences of the string `$1`. Quoting the terminating string prevents substitution entirely within a *here* document, for example:

```
grep $i <<\#
. . .
#
```

In this case the shell does not try to replace the # with anything.

The document is presented without modification to `grep`. If parameter substitution is not required in a *here* document, this latter form is more efficient.

Control Flow in the Bourne Shell — `while`

The actions of the `for` loop and the `case` branch are determined by data available to the Bourne shell. A `while` or `until` loop and an `if then else` branch are also provided whose actions are determined by the exit status returned by commands. A `while` loop has the general form

```
while command-list-1
do command-list-2
done
```

The value tested by the `while` command is the exit status of the last simple command following `while`. Each time round the loop *command-list-1* is executed; if a zero exit status is returned then *command-list-2* is executed; otherwise, the loop terminates. For example,

```
while test $1
do . . .
    shift
done
```

is equivalent to

```
for i
do . . .
done
```

`shift` is a Bourne shell command that renames the arguments `$2, $3, . . .` as `$1, $2, . . .` and discards `$1`.

Another kind of use for the `while/until` loop is to wait until some external event occurs and then run some commands. In an `until` loop the termination condition is reversed. For example,

```
until test -f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. Presumably another process will eventually create the file.

Control Flow in the Bourne Shell — `if`

A general conditional branch of the form

```
if command-list
then    command-list
else    command-list
fi
```

is also available to test the value returned by the last simple command following `if`.

We can illustrate a very simple use of the `if` command by expanding on our `name` procedure from before. The relevant change is in the first few lines (remember that `-lt` means *less than*):

```
$ cat name
#! /bin/sh
if test $# -lt 1
then
        echo Usage: $cmd name ...
        exit 1
fi
grep -i $1 <<woof
Ted Applehead           teda@seeds              7534
Bernice Barns           boat@carib              7441
    . . .
    more names
    . . .
David Smiter            acme@nadir              7435
Ben Tortcake            tort@icky               7258
Dave von Noknock        dave@dove               7296
woof
$
```

The change here is the `if` command — the original version of the procedure didn't check that the user supplied any parameters at all. This version checks the number of parameters (`$#`) using the `test` command, and displays a *usage* message if there are no parameters to remind the user of the correct way to use the procedure.

We mentioned earlier that the `test` command can also be written as `[`. Here is the first couple of lines of the `name` procedure above rewritten in that way:

```
$ cat name
#! /bin/sh
if [ $# -lt 1 ]; then
        echo Usage: $cmd name ...
        exit 1
fi
grep -i $1 <<woof
    . . .
woof
$
```

The `if` command may also be used in conjunction with the *test* command to test for the existence of a file as in

```
if test -f file
then    process file
else    do something else
fi
```

Here is an example of the `test` command in action. This is an extract from the `diff3` Shell procedure:

```
$ cat -n /usr/bin/diff3
     1  #! /bin/sh
     2  e=
     3  case $1 in
     4  -*)
     5    e=$1
     6    shift;;
     7  esac
     8  if test $# = 3 -a -f $1 -a -f $2 -a -f $3
     9  then
    10    :
    11  else
    12    echo usage: diff3 file1 file2 file3 1>&2
    13    exit
    14  fi
    15  trap "rm -f /tmp/d3[ab]$$" 0 1 2 13 15
    16  diff $1 $3 >/tmp/d3a$$
    17  diff $2 $3 >/tmp/d3b$$
    18  /usr/lib/diff3 $e /tmp/d3[ab]$$ $1 $2 $3
```

The relevant line is number 8, which reads

```
if test $# = 3 -a -f $1 -a -f $2 -a -f $3
```

This says that if the number of parameters (`$#`) is equal to 3, and all three parameters are files, the procedure can continue, otherwise the procedure displays an error message and stops. (The `-a` is a *logical and operator*; it joins statements just like the word *and*.)

`elif`: Multiple-Test Version of `if`

A multiple-test `if` command of the form

```
if . . .
then    . . .
else    if . . .
    then    . . .
    else    if . . .
          . . .
        fi
    fi
fi
```

may be written using an extension of the `if` notation:

```
if condition-1
then    actions-1
elif    condition-2
then    actions-2
elif    condition-3
. . .
fi
```

The sequence

```
if command-1
then    command-2
fi
```

may be written this way (the `&&` is a logical *and*):

```
command-1 && command-2
```

Conversely,

```
command-1 || command-2
```

executes *command-2* only if *command-1* fails (the `||` is a logical *or*). In each case the value returned is that of the last simple command executed.

Command Grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

In the first, *command-list* is simply executed. (The semi-colon is necessary to indicate the end of *command-list*.) The second form executes *command-list* as a separate process. For example,

```
$ (cd x; rm junk )
$
```

executes `rm junk` in the directory `x` without changing the current directory of the invoking Shell.

The commands

```
$ cd x; rm junk
$
```

have the same effect but leave the invoking Shell in the directory *x*.

Debugging Bourne Shell Procedures

The Bourne shell provides two tracing mechanisms to help in debugging Shell procedures. The first is invoked within a procedure as

```
set -v
```

(`v` for *verbose*) and displays lines of the procedure as they are read. It is useful to help isolate syntax errors. It may be invoked within a script, or when the procedure is run, as here: procedure, by saying

```
$ sh -v proc . . .
$
```

where *proc* is the name of a Bourne shell procedure. This flag may be used in conjunction with the `-n` flag which prevents execution of subsequent commands. `-n` serves as a *breakpoint*, allowing you to stop a script at a convenient point in the debugging, instead of having the whole script execute. Note that saying `set -n` at a terminal will render the terminal useless until an end-of-file is typed.

The command

```
set -x
```

produces an execution trace. Following parameter substitution, each command is displayed as it is executed. The `-v` and `-x` flags are similar; `-x` puts a + sign in front of the line shown being executed, and it only displays executing lines, not control lines. This means that a `for` or `while` loop line will be displayed with `-v` but not with `-x`. The following shows the difference:

```
$ cat test
echo hello
for i in one two
do echo $i
done
$  sh -v test
echo hello
hello
for i in one two
do echo $i
done
one
two
$  sh -x test
+ echo hello
hello
+ echo one
one
+ echo two
two
$
```

Notice how, in the second example, *one* and *two* are substituted in for `$i`. Both flags may be turned off by saying

```
set -
```

and the current setting of the Bourne shell flags is available as `$-`.

Keyword Parameters in the Bourne Shell

Bourne shell variables may be given values by assignment or when a Shell procedure is invoked. An argument to a Bourne shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking Shell is not affected. For example,

```
$ user=fred command
```

executes *command* with **user** set to *fred*. The `-k` flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as arguments `$1, $2, . . ..`

You can also use the *set* command to set arguments from within a procedure. For example,

```
set - *
```

sets `$1` to the first filename in the current directory, `$2` to the next, and so on. Note that the first argument, -, ensures correct treatment when the first filename begins with a -.

Parameter Transmission in the Bourne shell

When a Bourne shell procedure is called, both arguments and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a Bourne shell procedure by specifying in advance that such parameters are to be *exported*. For example,

```
export user box
```

marks the variables *user* and *box* for export to procedures. When a Shell procedure is called, copies are made of all exported variables for use within the invoked procedure. For example:

```
$ name=fred
$ export name
$ cat test
echo $name
$ sh test
fred
$
```

Modification of such variables within the procedure does not affect the values in the calling Shell. (It is generally true of a Bourne shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. Shared file descriptors are an exception to this rule.)

Names whose values are intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name . . .
```

Subsequent attempts to set readonly variables are illegal.

Parameter Substitution in the Bourne Shell

If a Bourne shell parameter is not set, the null string is substituted for it. For example, if the variable d is not set

```
$ echo $d
```

or

```
$ echo ${d}
```

will echo nothing. A default string may be given as in

```
$ echo ${d-.}
```

which will echo the value of the variable d if it is set and '.' otherwise. The default string is evaluated using the usual quoting conventions so that

```
$ echo ${d-'*'}
```

will echo * if the variable d is not set. Similarly

```
$ echo ${d-$1}
```

will echo the value of d if it is set and the value (if any) of $1 otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d-.}
```

and if d was not previously set then it is now set to the string '.'. The notation ${...=...} is not available for arguments.

```
echo ${d?message}
```

echos the value of the variable d if it has one; otherwise the Bourne shell prints *message*, if the Shell is interactive, and stops executing the procedure. If *message* is absent, then a standard message is printed. A Bourne shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
. . .
```

Colon (:) is a command that is built in to the Bourne shell and does nothing once its arguments have been evaluated. If any of the variables **user, acct** or **bin** are not set, and the Shell is not interactive, the Shell stops executing the procedure.

Command Substitution in the Bourne Shell

In a similar way, you can substitute the standard output from a command as the value of a parameter. The command *pwd* displays on its standard output the name of the current directory. For example, if the current directory is */home/fred/bin* then the command

```
d=`pwd`
```

is equivalent to

```
d=/home/fred/bin
```

The entire string between grave accents[56] (`...`) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ` must be escaped using a \ . For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents) and the treatment of the resulting text is the same in both cases. This mechanism allows use of string processing commands within Bourne shell procedures. An example of such a command is *basename*, which removes a specified suffix and the pathname's prefix from a string. For example,

```
basename /home/fred/main.c .c
```

displays the string *main.* The following fragment from a *cc* command illustrates its use:

```
case $A in
    . . .
    *.c)    B=`basename $A .c`
    . . .
esac
```

that sets `B` to the part of `$A` with the pathname and suffix `.c` stripped.

Here are some composite examples.

- `for i in `ls -t`; do . . .`
  The variable `i` is set to the names of files in time order, most recent first.

- `set `date`; echo $6 $2 $3, $4`
  will print, for instance, `1977 Nov 1, 23:59:59`

Evaluation and Quoting in the Bourne Shell

The Bourne shell is a macro processor that provides parameter substitution, command substitution and filename generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in the 'Grammar' section. Before a command is executed, the following substitutions occur.

[56] Often called backquotes.

- Parameter substitution, such as `$user`
- Command substitution, such as `` `pwd` ``

  Only one evaluation of a variable occurs. For example, if the value of the variable `y` is *hello*, so that

  ```
  echo $y
  ```

  yields `hello`, and we set the variable `X` to *$y*, then

  ```
  echo $X
  ```

  yields `$y` and not `hello`.
- Blank interpretation

  Following the above substitutions, the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string `$IFS`. By default, this string consists of blank, tab and newline. The null string is not regarded as a word unless it is quoted. For example,

  ```
  echo ''
  ```

  will pass on the null string as the first argument to *echo*, whereas

  ```
  echo $null
  ```

  will call *echo* with no arguments if the variable `null` is not set or set to the null string with `null=''`.
- Filename generation

  Each word is then scanned for the file pattern characters `*`, `?`, and `[. . .]`, and an alphabetical list of filenames is generated to replace the word. Each such filename is a separate argument.

The evaluations just described also occur in the list of words associated with a `for` loop. Only parameter and command substitution occurs in the *word* used for a `case` branch.

As well as the quoting mechanisms described earlier using and `'. . .'`, a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occur, but filename generation and the interpretation of blanks does not. The following characters have special meanings within double quotes and may be quoted using \.

Character	*Meaning*
$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

passes the value of the variable x as a single argument to *echo*. Similarly,

```
echo "$*"
```

passes the argument as a single argument and is equivalent to

```
echo "$1 $2 . . ."
```

The notation $@ is the same as $* except when it is quoted.

```
echo "$@"
```

passes the arguments, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" . . .
```

The following table gives, for each quoting mechanism, the Bourne shell metacharacters that are evaluated.

Table D-2 *Quoting Mechanisms*

Quoting Character	*Metacharacter*					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

Where t=terminator, y=interpreted, and n=not interpreted

In cases where more than one evaluation of a string is required, use the built-in command *eval*. For example, if the variable X has the value *$y* and y has the value *pqr*, then

```
eval echo $X
```

echos the string *pqr*.

In general, the *eval* command evaluates its arguments (as do all commands) and treats the result as input to the Bourne shell. The input is read and the resulting command(s) are executed. For example,

```
wg='eval who|grep'
$wg fred
```

is equivalent to

```
who|grep fred
```

In this example, *eval* is required since there is no interpretation of metacharacters, such as |, following substitution.

Error Handling in the Bourne Shell

The treatment of errors detected by the Bourne shell depends on the type of error and on whether the Bourne shell is being used interactively. A Bourne shell invoked with the `-i` flag is deemed to be interactive.

Execution of a command (see also 'Command Execution') may fail for any of the following reasons.

- Input/output redirection may fail, for example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a 'bus error' or 'memory fault.' See table D-3 for a complete list of SunOS signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the Bourne shell goes on to execute the next command. Except for the last case, the Bourne shell displays an error message. All remaining errors cause the Bourne shell to exit from a command procedure. An interactive Bourne shell will return to read another command from the terminal. Such errors include the following:

- Syntax errors such as, if ... then ... done
- A signal such as an interrupt. The Bourne shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as *cd*.

The Bourne shell flag `-e` terminates the Bourne shell if any error is detected.

Table D-3 *SunOS Signals*

Signal Name	*Signal Number*	*Notes*	*Description*
SIGHUP	1		hangup
SIGINT	2		interrupt
SIGQUIT	3	*	quit
SIGILL	4	*	illegal instruction
SIGTRAP	5	*	trace trap
SIGIOT	6	*	IOT instruction
SIGEMT	7	*	EMT instruction
SIGFPE	8	*	floating point exception
SIGKILL	9		kill — cannot be caught, blocked, or ignored
SIGBUS	10	*	bus error
SIGSEGV	11	*	segmentation violation
SIGSYS	12	*	bad argument to system call
SIGPIPE	13		write on a pipe with no one to read it
SIGALRM	14		alarm clock
SIGTERM	15		software termination signal from kill
SIGURG	16		urgent condition on IO channel
SIGSTOP	17	†	stop — cannot be caught, blocked, or ignored
SIGTSTP	18	†	stop signal from tty
SIGCONT	19	•	continue after a stop — cannot be blocked
SIGCHLD	20	•	to parent on child stop or exit
SIGTTIN	21	†	background read attempted from control terminal
SIGTTOU	22	†	background write attempted from control terminal
SIGIO	23		input/output possible signal *
SIGXCPU	24		exceeded CPU time limit
SIGXFSZ	25		exceeded file size limit
SIGVTALRM	26		virtual time alarm
SIGPROF	27		profiling time alarm
SIGWINCH	28	•	window changed

Notes on the Signals

* These signals normally create a memory image of the terminated process ("core dumped").

• These signals are discarded if the signal action is `SIG_DFL`.

† These signals normally stop the process.

The Bourne shell itself ignores quit, which is the only external signal that can cause a dump. The signals in this list of potential interest to Bourne shell programs are 1, 2, 3, 14 and 15.

Fault Handling in the Bourne Shell

Bourne shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received it executes the commands

```
rm /tmp/ps$$; exit
```

Exit is another built-in command that terminates execution of a Bourne shell procedure. The *exit* is required; otherwise, after the trap has been taken, the Bourne shell will resume executing the procedure at the place where it was interrupted.

SunOS signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without its having to take any further action. If a signal is being ignored, on entry to the Bourne shell procedure, for example, by invoking it in the background (see 'Command Execution'), then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the `name` command. You'll recall that the version of the `name` command shown using a *here* document would only look for one name at a time and that if we modified it to look for multiple names, the *here* document would be read every time around the `for` loop. Here is a version that copies the *here* document into a temporary file. The name of the temporary file is derived from the process ID of this command. When the procedure terminates, the `trap` is called to remove the temporary file. Let's take a look at this version of the `name` command (note that script creates a temporary file using the `$$` variable):

```
#! /bin/sh -u
if [ $# -lt 1 ]; then
    echo Usage: name person ...
    exit 1
fi
junk=/tmp/$cmd.$$
trap "rm -f $junk; exit" 0 1 2 15
cat > $junk <<woof
Ted Applehead          teda@seeds          7534
Bernice Barns          boat@carib          7441
    . . .
    more names
    . . .
David Smiter           acme@nadir          7435
Ben Tortcake           tort@icky           7258
Dave von Noknock       dave@dove           7296
woof
for person
    do grep -i $person $junk
done
```

The `trap` command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in SunOS, the Bourne shell uses it to indicate the commands to be executed on exit from the Bourne shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the *nohup* command:

```
trap '' 1 2 3 15
```

which causes both the procedure and the invoked commands to ignore the *hangup*, *interrupt*,and *kill* signals.

Traps may be reset by saying:

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing:

```
trap
```

The `scan` *Command*

The `scan` procedure shown below is an example of the use of `trap` where there is no exit in the `trap` command. `scan` takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when `scan` is waiting for input.

```
d=`pwd`
for i in *
do if test -d $d/$i
    then cd $d/$i
        while echo "$i:"
            trap exit 2
            read x
        do trap : 2; eval $x; done
    fi
done
```

`read` is a built-in command that reads one line from the standard input and places the result in the variable which is its argument. `read` returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

Here is an example of the `scan` command in action:

```
$ scan
bin:
ls
diffmark     henry.pct    lifescreen    scan.sh
bin:
^D
experiments:
ls
Makefile         doctools         macro.packages   test.bs
Old.Stuff        ellipse.ps       macros           test.pages
diffs            junk             new.macros       tmac.ex
experiments:
rm junk
experiments:
^D
misc:
ls -CF
addresses/       memos/            squash/
henry.raving/    quotes/           status.reports/
howto/           ski.cabins/       stoneman/
jokes/           solar/            sugfest/
letters/         sources/          sun.board
misc:
^D
system.v.book:
ls
Makefile            intro.mexp         shell.mexp
book.mss            login.mexp         shex1.mss
docprep.mexp        mail.mexp          shex2.mss
ed.and.sed.mexp     manpage.mss        softtool.mexp
ex.mexp             misc               stdio.mexp
filesystem.mexp     preface.mexp       system.admin.mexp
headex.mss          roman.mss          tablex.mss
system.v.book:
^D
$
```

Command Execution in the Bourne Shell

To run a command (other than a built-in), the Bourne shell first creates a new process using the *fork* system call. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no fork is required and simply replaces the Bourne shell with a new command. For example, a simple version of the *nohup* command looks like:

```
trap '' 1 2 3 15
exec $*
```

The *trap* turns off the specified signals so that they are ignored by subsequently created commands and *exec* replaces the Shell by the command specified.

Most forms of input/output redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No filename generation or blank interpretation takes place so that, for example,

```
echo . . . >*.c
```

writes its output into a file whose name is `*.c`. Input/output specifications are evaluated left to right as they appear in the command.

A *file descriptor* is a number assigned to a file when the file is opened for reading and/or writing. File descriptors 0, 1, and 2 refer to the *standard input*, *standard output*, and *standard error* (error messages) respectively.

> *word*	The standard output (file descriptor 1) is sent to the file *word*, which is created if it does not already exist.
>> *word*	The standard output is sent to file *word*. If the file exists, then output is appended (by seeking to the end); otherwise the file is created.
< *word*	The standard input (file descriptor 0) is taken from the file *word*.
<< *word*	The standard input is taken from the lines of Bourne shell input that follow, up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted, then parameter and command substitution occur, and \ is used to quote the characters \ $ ` and the first character of *word*. In the latter case newline quoted with backslashes are ignored (c.f. quoted strings).
>& *digit*	The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
<& *digit*	The standard input is duplicated from file descriptor *digit*.
<&-	The standard input is closed.
>&-	The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
. . . 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
. . . 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the Shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
$ ed file &
```

would allow both the editor and the Shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that the command ignores them. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason the SunOS convention for a signal is that if it is set to 1 (ignored), then it is never changed, even for a short time. Note that the Bourne shell command *trap* has no effect for an ignored signal.

Calling the Bourne Shell

The Bourne shell interprets the following flags when it is called. If the first character of argument zero is a minus—that is, the command itself starts with a minus—then commands are read from the file *.profile*.

`-c` *string*
: If the `-c` flag is present, commands are read from *string*.

`-s` If the `-s` flag is present or if no arguments remain, commands are read from the standard input. Bourne shell output is written to file descriptor 2.

`-i` If the `-i` flag is present or if the Bourne shell input and output are attached to a terminal (as determined by *gtty*), then this Bourne Shell is *interactive*. In this case TERMINATE is ignored (so that `kill 0` does not kill an interactive Bourne Shell), and INTERRUPT is caught and ignored (so that `wait` is interruptable). In all cases, the Shell ignores QUIT.

Bourne Shell Grammar

Commands are parsed initially according to the following grammar.

item: *word*
    input-output
    name = value

simple-command: item
    simple-command item

command: simple-command
    (*command-list*)
    { *command-list* }
    `for` *name* `do` *command-list* `done`
    `for` *name* `in` *word* ... `do` *command-list* `done`
    `while` *command-list* `do` *command-list* `done`
    `until` *command-list* `do` *command-list* `done`
    `case` *word* `in` *case-part* ... `esac`
    `if` *command-list* `then` *command-list else-part* `fi`

pipeline: *command*
    pipeline | *command*

andor: *pipeline*
    andor && *pipeline*
    andor || *pipeline*

command-list: *andor*
command-list ;
command-list &
command-list ; *andor*
command-list & *andor*

input-output: > *file*
< *file*
>> *word*
<< *word*

file: *word*
& *digit*
& -

case-part: *pattern*) *command-list* ;;

pattern: *word*
pattern | *word*

else-part: `elif` *command-list* `then` *command-list else-part*
`else` *command-list*
empty

empty:

word: *a sequence of non-blank characters*

name: *a sequence of letters, digits or underscores starting with a letter*

digit: `0 1 2 3 4 5 6 7 8 9`

Bourne Shell Metacharacters and Reserved Words

Syntactic

`|`	pipe symbol
`&&`	'andf' symbol
`||`	'orf' symbol
`;`	command separator
`;;`	case delimiter
`&`	background commands
`( )`	command grouping
`<`	input redirection
`<<`	input from a here document
`>`	output creation
`>>`	output append

Patterns	`*`	match any character(s) including none
	`?`	match any single character
	`[. . .]`	match any of the enclosed characters
Substitution	`${. . .}`	substitute Shell variable
	`` `. . .` ``	substitute command output
Quoting	`\`	quote the next character
	`'. . .'`	quote the enclosed characters except for '
	`". . ."`	quote the enclosed characters except for `$ ` \ "`

Reserved Words

```
if  then  else  elif  fi
case  in  esac
for  while  until  do  done
{  }
read
```

E

Command Summary

E

Command Summary

Filename Substitution

[*range*]
Match characters in a list or range.

[ab]*
matches filenames starting with a or b.

[a-Z1-0]*
matches filenames starting with any alphanumeric character.

{*string*, *string*}
Match enclosed strings.

{venus,mars}
matches the filenames venus and mars.

File Properties

chmod *arg filename*
change permissions. *arg* is one of:

ddd
where *d* is a digit from 0 to 7.

class op perm ...
where *class*, *op* and *perm*, are taken from:

class		*op*		*perm*	
u	user (owner)	=	set permission	r	read
g	group	−	remove access	w	write
o	others (public)	+	give access	x	execute
a	all				

crypt [*key*] *filename*
encrypt a file using *key* as the encryption key. To edit an encrypted file, use vi -x.

ln [−s] *oldname newname*
make a link to *oldname* called *newname*. With −s, make a symbolic link.

ls *option*
List files and selected properties. *option* can be one or more of:

`-a` list hidden files.

`-l` long listing. Shows permissions, links, owner, modification time, and name.

`-lg` groups. Shows group ownership in addition to above properties.

`-ld` directory. Shows `-l` listing for a directory itself, rather than the files it contains.

`-F` Append a tag indicating the file type:

- `*` execute permission is set.
- `/` directory.
- `@` symbolic link.

`pushd`, `popd` and `dirs`
use the directory stack to remember and revisit directories.

`touch` *filename*
change a file's modification time to the current time. Create a file if *filename* doesn't exist.

`tty`
display the filename of the terminal.

`umask` *ddd*
set initial permissions mask for new files according to the table below. The default mask is `022`.

Files		*Directories*	
value	*permissions*	*value*	*permissions*
0	`rw-`	0	`rwx`
1	`rw-`	1	`rw-`
2	`r--`	2	`r-x`
3	`r--`	3	`r--`
4	`-w-`	4	`-wx`
5	`-w-`	5	`-w-`
6	`---`	6	`--x`
7	`---`	7	`---`

I/O Redirection

`>` redirect the standard output.

`>!` force redirection, even if the file exists.

`>>` append the standard ouput to the file.

`>>!` append the standard output, creating the file if necessary.

`>&` redirect both the standard output and the standard error.

`>>&` append both the standard output and the standard error.

`<` redirect the standard input.

| pipe. Use the standard output of the command on the left as the standard input for the command on the right.

|& Use both the standard output and standard error of the command on the left as input for the command on the right.

`/dev/null`
: The system wastebasket. Unwanted output can be redirected to this file.

`/dev/tty`
: The terminal. Output from commands in scripts and subshells can redirected to the screen using this filename.

`set noclobber`
: This command prevents files from accidental overwrites. Include it in your `.cshrc` file.

`| tee` *filename*
: When placed on the end of a pipeline, the standard output is both redirected to *filename* and echoed on the screen.

Command-Line Special Characters

& run the command in the background.

`\`*c* escape character. Interpret *c* as text with no special meaning.

" double-quote. Interpret characters enclosed by double-quotes as a single word.

' quote. Interpret characters enclosed by quotes as a single word, and do not perform substitutions. (Special characters must still be escaped to be ignored.)

; command separator. Commands separated by semicolons are performed sequentially.

Filters

`cat` *filename ...*
concatenate and print one or several files.

`fmt` *filename*
simple file formatter.

`grep` *"reg_exp" filename ...*
search for a regular expression in a file or files. *reg_exp* is a combination of text, escaped characters, and *grep* special characters from the following table:

character	*matches:*
`^`	The beginning of a text line.
`$`	The end of a text line.
`.`	Any single character (like `?` in filename substitution).
`[...]`	Any single character in the bracketed list or range.
`[^...]`	Any character not in the list or range.
`*`	Zero or more occurrences of the *preceding character* or *regular expression*. (Not like filename substitution.)
`.*`	Zero or more occurrences of any single character. Equivalent to '`*`' in filename substitution.
`\`	Escapes special meaning of next character.

`head [` *−n* `]` *filename*
Display the first *n* lines of a file.

`look` *str*
look up words beginning with *str* in the system dictionary.

`more`
page through a file. The subcommand:

/string skips to a screenful containing *string*.

`nroff` *−mac filename*
format a file using the *mac* macro package.

`pr -t` *−n filename*
print a file in *n* column format. The `-t` option suppresses a heading that would otherwise appear.

`rev` *filename*
reverse the order of characters in each line of a file.

`spell` *filename*
check for misspelled words.

`sort` *filename*
put lines of a file in order.

`tail` *option filename*
display the last several lines of a file, as determined by *option*:

-n display the last *n* lines.

+n skip to line number *n*, and display the remaining lines.

`wc` *filename*
display the number of lines, words and characters in a file.

Job Control

`%` [*n*]
bring job *n*, or the current job, to the foreground.

`%` [*n*] `&`
resume processing stopped job *n*, or the current job, in the background.

`jobs`
display the list of background jobs.

Process Control

`kill` *PID*
terminate process number *PID*.

`ps [ -au ]`
display the list of processes. With the `-au` option, display the list of processes owned by all users.

User Activity

`grep` *userid* `/etc/passwd`
search for *userid* in file containing the list of local users.

`su` [*userid*]
switch userid to *userid*, or `root` (the superuser), when *userid* is omitted.

`w` display a detailed list of users currently logged in.

`who`
display a brief list of users currently logged in.

`who am i`
display the *userid*, terminal name, date and time.

`whoami`
display the userid only.

Managing Files

`diff` *leftfile rightfile*
show differences between two files.

`df` show disk space utilization on each disk as a percentage of capacity.

`du` show disk space utilization in the current directory.

`find`

`find` *pathname options*
locate files that meet the conditions specified in *options*, within the directory *pathname*, and its subdirectories. *options* can be:

`\ !` *option*	invert the meaning of *option*. (Select files for which the option doesn't apply.)
`\ (` *option* ... `\ )`	group a set of options into one condition.

`-exec` *command* `'{}' \;`
perform *command* on the located files.

`-group` *group* — locate files belonging to *group*.

`-mtime` *n* — select files modified within *n* days.

`-name` *filename* — locate files that match *filename* after filename substitution.

`-newer` *checkfile* — locate files modified more recently than *checkfile*.

`-o` — within an option group of the form:

`\(` *-option* `-o` *option* `\)`

select files for which either option applies. Normally, a file is selected only when all options apply.

`-print` — print the list of selected files.

`-user` *userid* — select files owned by *userid*.

`file` *filename* — determine the type of device, or type of data contained in, *filename*.

`make`

`make [ -n ] [ -f` *makefile* `]`
perform the procedure described in *makefile*. With the `-n` option, `make` echoes the commands it will perform, without performing them.

makefile is composed of *macro definitions* and *target definitions*:

macro definition
a line of the form:

macro = *expansion*

macro
is a character string.

expansion
is the remainder of the text on the line.

Once defined, macros are called as:

`$(`*macro*`)`

throughout the file.

target definitions
a set of lines of the form:

target: *dependency* ...
command line[57]

[57] starts with a Tab

...

target
a filename produced by, or logical label for, the step.

dependency
the name of another target upon which this one depends.

command line
a SunOS command line, beginning with a tab character. (If the tab is followed immediately by a dash (–) then return codes from commands on that line are ignored. Comment lines are introduced with a #).

`sccs`

`sccs` *subcommand filename*

use a feature of the source code control system. *subcommand* is one of:

`create`	put a file under `sccs` control by creating a history file in the `SCCS` subdirectory.
`info`	report any files checked out (omit *filename* in this case).
`edit`	check out a file.
`diffs`	contrast the edited version with the most recent checked in version.
`delget`	check in a new version to the history file and replace the existing version of the text.
`delta`	check in a new version to the history file.
`get`	rebuild the current checked in version.
`prt`	examine the summary comments for all versions in the history file.

`sccsdiff -r`*x.y* `-r`*m.n*
contrast previous versions *x.y* and *m.n*.

`tar`

`tar` *option filename*
tape archive program. *option* is one of:

`-cvf` *drive*	create an archive on *drive*.
`–xvf` *drive*	extract files from an archive on *drive*.
`–tvf` *drive*	display the files in the archive on *drive*.

Locating Commands

`whatis` *command*
give one-line description of a command.

`whereis` *command*
search the standard directories for the pathname of a command.

`which` *command*
search directories in the user's `path` variable for *command.*

Line Printer Commands

`lpr` [–P*printer*] *filename*
select a *printer* to print a file.

`lpq` [–P*printer*] *filename*
display the queue for *printer*.

`lprm` [–P*printer*] *job*
remove *job* from the queue for *printer*.

`troff –t` *options filename* ... > *output.file*
place typesetter-formatted output in an intermediate (binary) *output.file* for later printing.

`lpr –t` *output.file*
print a `troff` output file.

`screendump | rastrepl | lpr -v`
print the workstation screen display.

Misc. Commands

`chesstool`
window-based chess-playing program.

`csh`
the C shell command.

`date`
display the date and time.

`echo`
display the arguments on the terminal.

`printenv`
display the list of environment variables and values.

`set` *var* [=*value*]
create, or assign a value to, a C shell variable.

`sh` the Bourne shell command.

`source` *filename*
read and perform commands in *filename*.

`time` *command*
report statistics for *command*.

Index

Q

R

S

T

U

Notes

Notes

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 287815

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Sun House
31-41 Pembroke Broadway
Camberley
Surrey GU15 3XD
England
0276 62111
TLX 859017

Australia: 61-2-436-4699
Canada: 416 477-6745
France: (1) 46 30 23 24
Germany: (089) 95094-0
Japan: (03) 221-7021
The Netherlands: 02155 24888
UK: 0276 691297

Europe, Middle East, and Africa, call European Headquarters:
0276 62111

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales