Chapter 5: UNIX Commands and Concepts

This chapter introduces you to the UNIX command structure, and to many important commands and concepts. The features introduced in this chapter constitute the core of the UNIX operating system, and many of these tools are quite powerful and flexible. Some of the features are shell-specific, and we provide the distinctions where necessary.

5.1 Processing Environment

5.1.1 Programs, Commands and Processes

A *program* is an executable file. A program is invoked by entering its filename (which is the *command* associated with the executable), often followed by options, arguments, and/or parameters on the command line. The shell allows three types of commands:

- an executable file that contains object code produced by a compilation of source code
- an executable file that contains a sequence of shell command lines (a *shell script*)
- an internal shell command (*built-in* command)

The first two command types may include standard UNIX utilities, commercial products, and user-written programs. All the shells allow both *interactive* command entry in which the commands are typed at the keyboard and executed one by one, and *scripted* entry in which commands are put in a file, called a shell script, and executed sequentially when the script is run. See section 4.4 for a brief discussion of the uses of shell scripts and how to execute them.

Shells execute commands by means of *processes*. A process is an instance of a program in execution. A process can interact with the kernel by invoking a well defined set of *system calls*. The system calls instruct the kernel to perform particular operations for the calling program and they can exchange data between the kernel and the process. For example, a process can use system calls to create new processes and terminate running processes.

When a terminal session begins, the operating system starts a single *parent process*. Creating a new process from an existing process is called *forking*. This new process is called a *child process* or *subprocess*. Each process has a unique process identification number (PID). A subprocess can fork another process and become a parent. A process which is not receiving input from the terminal, either running or stopped, is said to be in the *background* (see section 5.5). The **`ps`** command can be used to print the status of active processes. See the man pages for information about its options.

When you give the shell a command associated with a compiled executable or shell script, the shell creates, or *forks*, a new process called a *subshell*. The new process runs the system call *exec* which invokes yet another program to execute the command in place of the current process (the subshell). Unless the subprocess runs in the background, the parent process remains dormant until its subprocess completes or is stopped. Then control returns to the parent.

To execute most built-in commands, the shell forks a subshell which executes the command directly (no *exec* system call). For the built-in commands **cd**, **set**, **alias** and **source**, the current shell executes the command; no subshell is forked. You can, however, cause the shell to fork a process by enclosing the command in parentheses. The following example illustrates this (use of the semicolon is described in section 2.5; and the commands **cd** (change directory)and **pwd** (print working directory) are described in section 6.5):

% **cd /dir1; pwd**	displays **/dir1** (no subshell is forked)
% **(cd /dir2; pwd)**	due to the parentheses, a subshell is forked, then the commands are issued; displays **/dir2**. Control then returns to the parent process.
% **pwd**	displays **/dir1** since the current process was unaffected by the previous command line.

Most built-in commands exist in all shells, but there may be differences regarding arguments, options, or output format between the shell-specific versions of each command. Some commands for a given shell are not available on all platforms. Refer to a UNIX text for lists of built-in commands.

You do not need to distinguish between built-in and other commands to execute them. However in order to find help in the man pages, you do need to know which is which. Help on shell commands is usually found under the shell name, for example under **man tcsh** or **man bash**. Some platforms provide man pages for built-in commands, however in general you may find it easier to look in a reference book! Help on other commands is found directly under **man** ***command***.

5.1.2 Command Interpretation by the Shell

When the shell receives a command, it interprets it in a series of three (for Bourne shell family) or four (for C shell family) passes. Naturally, if the command is an alias (see section 9.3), it requires an additional pass up front for substitution.

- The first pass for the C shell family looks for the **!** character, and replaces it with the previous command (see section 5.3 for information on command recall).
- The next pass (the first pass for the Bourne shell family) replaces wildcards (used in filename expansion, redirection, and regular expressions; see sections 6.2.2, 5.4.2, and 5.4.5, respectively).
- The next pass looks for the **$** character in order to replace variable names with their values (see section 9.1).
- The final pass splits the command line elements by whitespace to arrive at the final, literal command that the shell must execute.

There are ways to prevent interpretation of special characters in each of these passes. Preceding a character with a backslash (****) works for all special characters; wildcards can be enclosed in single or double quotes; variables can be enclosed in single quotes; and whitespace is ignored if the argument containing the whitespace is enclosed in single or double quotes.

To illustrate the operations that take place in each pass, the following table presents a series of three examples using the **echo** command and the same string, first in single quotes, then double quotes, and finally with no quotes. The **echo** command writes the string to standard output. Assume that the files that match **q*** are `qq` and `qqq`, and the value of the variable *a* is **foo**.

Command -->	**echo 'q* $a x'**	**echo "q* $a x"**	**echo q* $a x**

After first pass[a], only wildcards are interpreted.	**`echo 'q* $a x'`** (no wildcard expansion due to quotes)	**`echo "q* $a x"`** (no wildcard expansion due to quotes)	**`echo qq qqq $a x`** (unquoted wildcard is expanded)
After second pass, unquoted or double quoted variables are replaced by their values.	**`echo 'q* $a x'`** (no variable replacement due to quotes)	**`echo "q* foo x"`** (double-quoted variable **`$a`** replaced by value)	**`echo qq qqq foo x`** (unquoted variable **`$a`** replaced by value)
After final pass, command string is broken up according to whitespace. The separate elements are listed vertically.	**`echo`** **`'q* $a x'`** (string treated as one argument due to quotes)	**`echo`** **`q* foo x`** (string treated as one aregument due to double quotes)	**`echo`** **`qq`** **`qqq`** **`foo`** **`x`** (no quotes; each argument treated separately)
When you type in the original command, the system returns the string:	**`q* $a x`**	**`q* foo x`**	**`qq qqq foo x`**

a. This would be the *second* pass for C shell family; there were no **`!`** characters to replace.

5.2 Command Entry

A UNIX command is either a built-in command or the name of an executable file which the operating system will load and execute. When you see the prompt, you can enter a command by typing the command name, any options and arguments, followed by a carriage return.

Recall, the formats displayed in this manual use **`this font style`** to indicate characters to be typed as is, and ***this font style*** to indicate arguments to be substituted. Arguments enclosed in square brackets, **`[...]`**, are optional.

You should be aware that UNIX commands are not noted for their consistency of format. Furthermore, commands, formats, arguments, and options may vary slightly from one UNIX platform to another. In this manual, we attempt to be as generic as possible, and describe options that are widely available.

UNIX commands are described on-line in the man pages (see section 3.1).

5.2.1 Command Format

The basic format of UNIX commands is:

`%` ***command −option(s) argument(s)***

where:

`%` is the (default, non-FUE) **csh** prompt.[1]

1. **`$`** is the non-FUE default for Bourne shell.

command	is the UNIX command name of a utility or tool.
option(s)	modifies how the command runs; options are nearly always preceded by a dash and listed one after another. See example below.
argument(s)	specifies data or entities (usually files) on which the command is to operate; arguments are separated by blanks ("white space").

Remember, UNIX is case-sensitive. Therefore UNIX commands must be entered in the correct case. Most of the time commands are entered in lower case.

The components are separated by at least one blank space. If an argument contains a blank, enclose the argument in double quote marks. Normally, options can be grouped; e.g., the **`-lw`** and the **`-l -w`** option specifications are equivalent in the examples below (**`wc`** is a sample command; it lists line, word, and/or character count of one or more files.):

`%` **`wc -lw`** *file1 file2*

`%` **`wc -l -w`** *file1 file2*

Some options can have arguments, and there isn't consistency on whether there should be a blank space between the option and its argument. Check the man pages when you're not sure. In the next example which shows the FORTRAN command, ***outputfile*** is the argument of the option **`-o`**:

`%` **`f77 -o`** ***outputfile program*`.f`**

Looping and conditional commands are also supported. These are more advanced shell commands and are not covered in this manual. Consult a UNIX text for information on these.

5.2.2 Miscellaneous Command Line Features

- To correct typos you can use the erase key (Delete or Backspace) to erase character-by-character, or the Kill key to kill an entire line (see section 2.4).
- More than one command can be entered on a line if the commands are separated by semicolons. The commands will be executed sequentially. See section 2.5 for more information on using multiple commands on one line.
- If you need to continue a command to a new line, you can either keep on typing (without doing a carriage return), or enter a backslash (****) followed directly by a carriage return (no space in-between) and then continue typing on the next line. (Recall the backslash is used to prevent a special character's meaning to be interpreted by the shell. See section 2.5.)
- You can use parentheses to group commands. Since a subshell is created for each group, this can be used to prevent changing the current environment. It can also be used to redirect all output from the commands considered as a group (see section 5.4.2).
- Type ahead works, even if the characters get interspersed with output.

5.3 Command Recall

Command recall is quite different in each shell. One common feature for all shells that support command recall is the **history** mechanism. It maintains a list of commands that have been entered and allows them to be reexecuted. The *history* variable, set to some number at login time in the start-up files, determines the number of commands that are saved in the list. The *savehist* variable specifies how many commands are to be saved for your next session after you log out. The **`history`** command displays the list of saved commands:

% history

We discuss the following shells separately: **csh**, **tcsh**, and **bash/ksh**. There is no command recall facility for **sh**.

csh

There is no command line editing native to **csh**. Before describing the standard **csh** command recall facility, we should mention a Fermilab product called **cedit** that we recommend for use with **csh** instead. It was designed to mimic VMS line editing, and turns out to provide similar command recall and editing functionality to **tcsh**. To use **cedit**, you need to set it up initially. Enter:

% setup cedit

To execute it, type:

% m

followed by <Return>. **m** stands for *modify*. Use the up or down arrow keys to scroll to the desired command. The right and left arrow keys and your backspace key allow you to edit the command before reexecuting it. There are several control characters that perform functions within **cedit**. Typing <Ctrl-i> in **cedit** displays the available commands.

Recalling history commands using standard **csh** syntax is fairly easy. Use the commands listed below.

!!	Reexecute the previous command
!*n*	Reexecute command ***n*** from the history list
!*text*	Reexecute the most recent command beginning with ***text***
!?*text*?	Reexecute the most recent command containing ***text***

For example, to reexecute the 4th command from the history list, enter:

% !4

and to reexecute the last command starting with **ls**:

% !ls

The dollar sign (**$**) can be used to recall the last word of a command. **!$** causes substitution of the last word of the last command. For example, you can check the contents of `myfile.f` and then compile it using the following sequence of commands:

% less myfile.f

% f77 !$

A couple of nice features you can use with these reexecution commands are *preview* (**p**) and *substitute* (**s**). To substitute a string in the previous command and preview it before execution, use the syntax:

% !:p:s/*oldstring*/*newstring*

To do the same for the ***n***th command in the history list, use:

% !*n*:p:s/*oldstring*/*newstring*

To execute after previewing (and/or substituting), simply type:

```
% !!
```

tcsh

Recalling commands is easy if you are using **tcsh**. The up/down arrows on the keyboard can be used to recall commands and the left/right arrows can be used to move around within the command to edit it (VMS users will be familiar with this concept).

A command line correction algorithm is available in **tcsh**. To enable it, enter:

```
% set correct=all
```

This causes all words on the command line to be checked. If any part gets corrected, the system notifies you, and gives you a chance to accept or reject it. For example, say you type in:

```
% lz /usr/bin
```

The system will return with:

```
CORRECT ls /usr/bin (y|n|e|a|)?
```

Where `y`=yes, `n`=no, `e`=edit, and `a`=abort. You must provide one of these responses.

To turn off command line correction, enter:

```
% set correct=none
```

ksh

Two styles of command recall are supported; **emacs** and **vi**. The style is determined in one of two ways:

- include the line **`set -o`** ***editor*** in either your `.profile` or `.shrc` file, where ***editor*** is either **emacs** or **vi** (this takes precedence if variables below are set differently)
- set either the *EDITOR* or *VISUAL* environment variable to one of these editors

When set to **emacs**, use the usual **emacs** commands to display and modify previous commands, for instance <Ctrl-p> for previous line. When set to **vi**, command recall is initiated by typing the Escape (or <Ctrl-[>) key. Then all the standard **vi** commands can be used. Some of the basic **vi** and **emacs** commands are listed in section 11.3.

bash

Both **csh** and **ksh**-style recall are supported.

5.4 Important Concepts

This section attempts to provide an overview of a few of the important concepts in UNIX which are very different from other systems and may therefore be confusing to the novice user. In order to be able to make effective use of UNIX, these concepts need to be understood.

5.4.1 Path

When you issue a command, the shell program parses the command line and either processes it directly or searches for an executable file with that name in any of the directories specified in your *search path*, which is controlled by the variable *PATH*. If the file is not found in any of the directories in your search path, the shell reports that the command was not found. The file may well be on the disk somewhere, but it is **not in your path**. This concept will be familiar to users of MS-DOS.

FUE attempts to provide an appropriate path, and we recommend that you not change this basic path. However, feel free to *add* directories to it. For the **csh** family, your `.login` file contains a `set path` line for the shell variable *path*.[1] Uncomment this line (remove the `#`) and include additional directories in the shown format:

```
#set path=($path /dir1 /dir2... )
```

Or change the environment variable *PATH* (also in `.login`), as follows:

```
setenv PATH "${PATH}:/dir1:/dir2"
```

For the **sh** family, uncomment and add directories to the `PATH` line in your `.profile` file:

```
# PATH=$PATH:/dir1:/dir2...
```

See section 9.2 for information on the *PATH* variable.

As an aside, if you add an executable to one of the directories in your search path, it may be necessary for you to either log out and log back in, or to recreate the internal tables used by the shell with the **`rehash`** (**csh**) or **`hash`** (**sh**) command (see section 4.4).

5.4.2 Standard Input and Output Redirection

The shell and many UNIX commands take their input from *standard input* (`stdin`), write output to *standard output* (`stdout`), and write error output to *standard error* (`stderr`). By default, standard input is connected to the terminal keyboard and standard output and error to the terminal screen. VMS users would know these as the logical devices SYS$INPUT, SYS$OUTPUT, and SYS$ERROR.

The way of indicating an end-of-file on the default standard input, a terminal, is usually <Ctrl-d>.

Redirection of I/O, for example to a file, is accomplished by specifying the destination on the command line using a *redirection metacharacter* followed by the desired destination.

C Shell Family

Some of the forms of redirection for the C shell family are:

Character	Action
`>`	Redirect standard output
`>&`	Redirect standard output and standard error
`<`	Redirect standard input
`>!`	Redirect standard output; overwrite file if it exists

1. Shell versus environment variables are discussed in section 9.1.

<table>
<tr><th>Character</th><th>Action</th></tr>
<tr><td>>&!</td><td>Redirect standard output and standard error; overwrite file if it exists</td></tr>
<tr><td>|</td><td>Redirect standard output to another command (pipe)</td></tr>
<tr><td>>></td><td>Append standard output</td></tr>
<tr><td>>>&</td><td>Append standard output and standard error</td></tr>
</table>

The form of a command with standard input and output redirection is:

`%` ***command* –[*options*] [*arguments*] < *input file* > *output file***

If you are using **csh** and do not have the *noclobber* variable set (see section 9.2), using **>** and **>&** to redirect output will overwrite any existing file of that name. Setting *noclobber* prevents this. Using **>!** and **>&!** always forces the file to be overwritten. Use **>>** and **>>&** to append output to existing files.

Redirection may fail under some circumstances: 1) if you have the variable *noclobber* set and you attempt to redirect output to an existing file without forcing an overwrite, 2) if you redirect output to a file you don't have write access to, and 3) if you redirect output to a directory.

Examples:

`%` **who > names**	Redirect standard output to a file named `names`
`%` **(pwd; ls -l) > out**	Redirect output of both commands to a file named `out`
`%` **pwd; ls -l > out**	Redirect output of **ls** command only to a file named `out`

Input redirection can be useful, for example, if you have written a FORTRAN program which expects input from the terminal but you want it to read from a file. In the following example, `myprog`, which was written to read standard input and write standard output, is redirected to read `myin` and write `myout`:

`%` **myprog < myin > myout**

You can suppress redirected output and/or errors by sending it to the *null device*, **/dev/null**. The example shows redirection of both output and errors:

`%` **who >& /dev/null**

To redirect standard error and output to different files, you can use grouping:

`%` **(cat myfile > myout) >& myerror**

Bourne Shell Family

The Bourne shell uses a different format for redirection which includes numbers. The numbers refer to the file descriptor numbers (0 standard input, 1 standard output, 2 standard error). For example, **2>** redirects file descriptor 2, or standard error. **&*n*** is the syntax for redirecting to a specific open file. For example **2>&1** redirects 2 (standard error) to 1 (standard output); if 1 has been redirected to a file, 2 goes there too. Other file descriptor numbers are assigned sequentially to other open files, or can be explicitly referenced in the shell scripts. Some of the forms of redirection for the Bourne shell family are:

Character	Action
>	Redirect standard output
2>	Redirect standard error
2>&1	Redirect standard error to standard output
<	Redirect standard input
|	Pipe standard output to another command
>>	Append to standard output
2>&1|	Pipe standard output and standard error to another command

Note that **<** and **>** assume standard input and output, respectively, as the default, so the numbers 0 and 1 can be left off. The form of a command with standard input and output redirection is:

$ *command* –[*options*] [*arguments*] < *input file* > *output file*

Redirection may fail under some circumstances: 1) if you have the variable *noclobber* set and you attempt to redirect output to an existing file without forcing an overwrite, 2) if you redirect output to a file you don't have write access to, and 3) if you redirect output to a directory.

Examples:

$ who > names	Direct standard output to a file named `names`
$ (pwd; ls -l) > out	Direct output of both commands to a file named `out`
$ pwd; ls -l > out	Direct output of **ls** command only to a file named `out`

Input redirection can be useful if you have written a FORTRAN program which expects input from the terminal and you want to provide it from a file. In the following example, `myprog`, which was written to read standard input and write standard output, is redirected to read `myin` and write `myout`.

```
$ myprog < myin > myout
```

You can suppress redirected output and/or error by sending it to the *null device*, **/dev/null**. The example shows redirection of standard error only:

```
$ who 2> /dev/null
```

To redirect standard error and output to different files (note that grouping is not necessary in Bourne shell):

```
$ cat myfile > myout 2> myerror
```

5.4.3 Pipes

UNIX uses the concept of a *pipe* to connect the standard output of one program directly into the standard input of another program. This is specified by separating the two commands with the pipe operator, the vertical bar (**|**). The general format is:

`%` ***command1*** **|** ***command2*** **|** `. . .`

where, of course, each command can have options and arguments. To implement pipes of commands, the shell forks off multiple processes. For example if you run the command:

```
% history | more
```

the shell forks twice; the grandchild runs **`history`**, the child runs **`more`** (after hooking up the right file descriptors to the right pipe ends), and the parent shell waits for the process to finish. The **`history`** command, a built-in, is implemented in the grandchild shell process directly, while the **`more`** command requires an *exec* system call.

The **`tee`** command can be used to send output to a file as well as to another command.

```
% who | tee whoout | sort
```

This creates a file named `whoout` which contains the original **`who`** output. It also sorts the **`who`** output and sends it to standard output, the terminal screen. The following example sends the (unsorted) **`who`** output to the file and the screen:

```
% who | tee whoout
```

5.4.4 Filters

A *filter* is a command or program which gets its input from standard input, sends its output to standard output, and may be used anywhere in a pipeline. Examples of filters are the UNIX utilities:

- **more** (and **less**)
- **grep**
- **awk**
- **sort**

The combination of UNIX filters **grep**, **awk**, and **sort** and the use of pipes is very powerful.

more and less

The **more** filter allows you to display output on a terminal one screen at a time. You press Spacebar to move to the following screen, and **`q`** to quit.

less is a much more flexible variant of the standard UNIX utility **more** and is provided under FUE[1]. The command **`less`** lists the output (e.g., specified files) on the terminal screen by screen like the command **`more`**, but in addition allows backward movement in the file (press **`b`** to go back one full screen) as well as forward movement. You can also move a set number of lines instead of a whole page. To view a file with the **less** filter, enter:

`% less [`*options*`] [`*filename*`]...`

1. FUE sets your environment variable *PAGER* to the **less** filter.

The options and usage are described in the man pages for **more** and **less**.

After displaying a page of information, **more** and **less** display a colon prompt (**:**) at the bottom of the screen and wait for instructions.

```
     LESS(1)                 UNIX System V                 LESS(1)



     NAME
          less - opposite of more

     SYNOPSIS
          less [-[+]aABcCdeEimMnqQuUsw] [-bN] [-hN] [-xN] [-[z]N]
               [-P[mM=]string] [-[lL]logfile] [+cmd]
               [-ttag] [filename]...

     DESCRIPTION
          Less is a program similar to more (1), but which allows
          backwards movement in the file as well as forward movement.
          Also, less does not have to read the entire input file
          before starting, so with large input files it starts up
          faster than text editors like vi (1).  Less uses termcap (or
          terminfo on some systems), so it can run on a variety of
          terminals.  There is even limited support for hardcopy
 :
```

You can search for patterns in the file by entering ***/pattern*** at the **less** prompt. Continue to search for the same pattern using a slash (**/**). A further advantage is that **less** does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like **vi**.

grep

The **grep** filter searches the contents of one or more files for a pattern and displays only those lines matching that pattern. **grep** is described in Section 6.4.2.

awk

awk is much more than a filter; it is a powerful pattern scanning and processing language. Although you will need to spend a little time learning how to use **awk**, it is very well suited to data-manipulation tasks. It handles internally what you would have to handle laboriously in a language like C or FORTRAN. You can do in a few lines what would take many, many lines of FORTRAN.

awk works best when the data it operates on has some structure, for example a document with heading levels, or a table. In the case of a table, you can tell it the field separator (spaces, colons, commas, tabs) and it can align and interpret the contents of the field according to the way you use it. Or you can reorder the columns, or change rows into columns and vice-versa.

We present here some very basic information to get you acquainted with the concepts of **awk**, but you will need a more in-depth reference in order to use this utility. A widely-available book on **awk** is *The awk Programming Language* by Aho, Kernighan, and Weinberger, Addison-Wesley. Another good reference, from which much of the information in the present section is extracted, is *sed & awk* published by O’Reilly & Associates.

There are several versions of **awk**, and they differ from platform to platform. “Old” **awk** may be **awk** or **oawk**, “new” **awk** may be **nawk**. FUE provides a GNU version of **awk** called **gawk** as part of the **shells** product.

Some of the features of **awk** are:

- Ability to view a text file as made up of records and fields in a textual database
- Use of variables to manipulate the database
- Use of arithmetic and string operators
- Use of common programming constructs such as loops and conditionals
- Ability to generate formatted reports

With **nawk**, additional features make it easier to write larger scripts. Using **nawk** you can:

- Define functions
- Execute UNIX commands from a script
- Process the result of UNIX commands
- Process command-line arguments more gracefully
- Work more easily with multiple input streams

awk executes a set of instructions for each line of input. You can specify instructions on the command line or create a script file. Input is read a line at a time from one or more files or from standard input. The ***instructions*** must be enclosed in single quotes to protect them from the shell. (Instructions always contain curly braces which are interpreted as special characters by the shell.) We refer you to one of the books on **awk** for the available instructions. We'll use the instruction **`print`** in our examples.

For command lines, the syntax is:

`% `**`awk '`***instructions***`'`** ***files***

As an example, say that file `test` contains only the line `Hello, world`. The command:

```
% awk '{ print }' test
```

produces the output:

```
Hello, world
```

Multiple command lines can be entered by separating commands with semicolons or using the multi-line input capability of the Bourne shell. **awk** programs are usually placed in a file where they can be tested and modified. The syntax for invoking **awk** with a script file is:

`% `**`awk -f '`***script***`'`** ***files***

where **`-f`** indicates that the filename of a script follows.

awk interprets each line of the input data file(s) as a record, and each word on that line, delimited by blank spaces or tabs, as a field. You can reference these fields, either in patterns or procedures. **`$0`** represents the entire input line; **`$1`**, **`$2`**, ... refer to the position of individual fields on the input line. As an example, say that the file `personnel` contains a list of employees' first names, last names, and addresses. The command:

```
% awk '{ print $1}' personnel
```

would produce output of the type:

```
John
Alice
Mary
Eric
```

To use the pattern-matching features of **awk**, you need to be familiar with the metacharacters used in regular expressions (see section 5.4.5). A pattern is enclosed between forward slashes (**/**) on the command line or in a script. When **awk** reads an input line, it attempts to match each pattern-matching rule in a script. Only the lines matching the particular pattern are the object of an action. If no action is specified, the line that matches the pattern is printed (executing the **`print`** statement is the default action).

In our `personnel` example, let's assume that Alice is from Illinois (IL) and Eric is from Iowa (IA). To bring up the complete records with the pattern IL or IA, we could issue the command:

```
% awk '/I./{print}' personnel
```

where the metacharacter **.** matches any single character. We could more simply type:

```
% awk '/I./' personnel
```

and get the same result in either case:

```
Alice Jones 834 S. Jefferson St., Batavia, IL 60510
Eric Smith 24 Birch St., Albert City, IA 50510
```

In Appendix C we present **awk**'s programming model, which is beyond the scope of the present section. This model will help you understand the potential that **awk** offers the programmer.

sort

sort sorts the lines of the specified files, typically in alphabetical order. Using the **-m** option it can merge sorted input files. Its syntax is:

% **sort** [*options*] [*field-specifier*] [*filename(s)*]

For example, start with the `personnel` file contents:

```
John Smith 75 South Ave., Denver, CO 80145
Alice Jones 834 S. Jefferson St., Batavia, IL 60510
Mary Fahey 901 California St., San Francisco, CA 94121
Eric Smith 24 Birch St., Albert City, IA 50510
```

Run the command:

```
% sort personnel
```

to reorder the file contents as follows:

```
Alice Jones 834 S. Jefferson St., Batavia, IL 60510
Eric Smith 24 Birch St., Albert City, IA 50510
John Smith 75 South Ave., Denver, CO 80145
Mary Fahey 901 California St., San Francisco, CA 94121
```

sort is very easy to use. Read the man page for **sort** to see what the available options are and how to specify the sort fields. If a field is not specified, the sort key is the entire line. The sorted output goes to standard output by default.

5.4.5 Regular Expressions

A *regular expression* is a string composed of letters, numbers, and special symbols that defines one or more strings. They are used to specify text patterns for searching. This is similar to wildcards on VMS.

A regular expression is said to *match* any string it defines. The major capabilities include:

1) match single characters or strings of characters

2) match any arbitrary character

3) match classes of characters

4) match specified patterns only at the start or end of a line

5) match alternative patterns

Regular expressions are used by **vi**, **grep**, and **awk** (and at least a couple of utilities not covered in this manual, for instance **ed** and **sed**). **grep** in fact stands for **g**lobal **r**egular **e**xpression **p**rinter. For a complete discussion of regular expressions, refer to a UNIX text. To get you started, we include a table of special characters that can be used in expressions.

Note that regular expression special characters are different from those used in filename expansion.

`.`	Matches any single character Example: *.ing* matches all strings with any character preceding *ing*; sin*g*ing, *p*ing
`*`	Represents 0 or more occurrences of the preceding character Example: *ab*c* matches *a* followed by 0 or more *b*'s followed by *c*; *ac*, *abc*, *abbbbbc*
`.*`	Matches any string of characters (`.` matches any character, `*` matches any number of occurrences of the preceding regular expression)
`$`	Placed at the end of a regular expression, matches the end of a line Example: *ay$* matches *ay* at the end of a line; ... tod*ay*
`^`	Placed at the beginning of a regular expression, matches the beginning of a line Example: *^T* matches a *T* at the beginning of a line; *T*oday ...
`"`	Delimits operator characters to prevent interpretation
`\`	Turns off special meaning of the following single character (`\` is often called a *quote* character)
`[]`	Specifies character classes
`[...]`	Matches any one of the characters enclosed in square brackets Example: *[bB]ill* matches *bill* or *Bill*

There is an extended set of special characters available for full regular expressions, including for example `?` and `+`. These can be used in **egrep** and **awk**. Refer to a UNIX book for information.

5.5 Job Control

Any command you give to the shell (true for all shells except **sh**) is a *job* and is given a *job number*. A single command is the simplest job. A series of commands separated by semicolons, or commands piped together, create a single job. A script also creates a single job. A job may consist of many processes, because each command is a process.

The job stays with its environment, for example, the current directory. If you subsequently change directories after putting a job in the background and then resume the background job, you will be in the original directory again.

Job control allows you to work on several jobs at once, switching back and forth between them at will, and it allows you to stop, start, and kill them. When you start up a job interactively, it is by default in the *foreground* and attached to your terminal. You can move that job into the *background* so you can start up another job or observe another job that is already running. You can move any background job into the foreground so it is once again attached to your terminal. You can run any number of background jobs at any one time, but there can be only one foreground job. The use of multiple windows on an X terminal makes much of this transparent.

5.5.1 Priority

You can control the priority of a command or shell with the shell command **nice**:

% **nice [+*n* |−*n*] [*command*]**

n is the value by which you want to increase or decrease priority. Values range from 1 to 19, with the default at 10. The higher the **nice** value, the lower the priority of a process, and the slower it runs. (You are being *nicer* to other users!) If no number is specified, **nice** sets the priority to 4. If ***command*** is omitted, the priority is set for the current shell. If ***command*** is specified, it is run at the specified (or default) priority in a sub-shell. You can use **nice** to lower the priority of a command or shell that makes large demands on the system but isn't needed right away.

☞ Note that another **nice** command exists, /bin/nice. It is not a built-in shell command. If you do **man nice**, you will get information on this one. In order to get information on the shell command **nice**, do **man csh** (using **csh** as an example).

5.5.2 Background, Foreground, and Suspended Jobs

You run jobs in the *background* so that you can perform other tasks in the foreground (i.e., interactively). Jobs are always in one of three states: running in the foreground, running in the background, or suspended. Any job intended to run in the background should have its output and error redirected to a file.

There are two ways to put jobs into the background:

Using the & Metacharacter

One way to start a job in the background is to append the ampersand metacharacter (**&**) to the end of the command line. In the first example, the standard output is redirected to a file (in this case, the syntax is valid for both shell families):

% ***command* > *output-file* &**

Note that the parentheses are necessary in the next example in order to send both commands to the background:

`% (`*`command1; command2`*`) &`

The shell prints a line indicating the job number and process ID of its top-level commands, and the job is started as a background job.

Using the Suspend Control Character

The other way is to use the suspend control character, called *susp* or *swtch*, (see section 2.4) which is usually assigned to <Ctrl-z>. It stops or *suspends* the foreground (the currently running interactive) job, moving it to the background; it does not kill it.

After stopping a job, you can either resume it with the **fg** command or make it run in the background with the **bg** command (see below). You may want to stop a job temporarily to do another task and then return to it interactively, or you may want to stop it in order to let it finish as a background job.

When a background job terminates, this is reported just before the next prompt (so the message doesn't interrupt the current foreground job).

A background job will stop if it tries to read from the terminal. If output is not redirected, a background job can either (continue to) send output to the terminal or be stopped if it attempts to write to the terminal. The following command can be used to toggle this behavior:

`% stty [-]tostop`

The minus indicates negation, meaning that background jobs will continue to run even if they attempt to write output to the terminal and that the output will appear on the terminal screen. However, programs which attempt to interrogate or change the mode of the terminal will be blocked when they are not in the foreground whether or not *tostop* is set.

Listing Jobs

The **jobs** command lists your jobs:

`% jobs [-l]`

This command lists the background jobs, their job number, status, and the command being executed. A plus sign in the output means that job is *current* (in control of your terminal), a minus sign means that job is *next*. *Current* and *next* refer to its relation to the foreground (see **fg**). The **-l** option lists the process ID as well.

Commands Used for Controlling Jobs

There are a number of commands to control jobs: **fg**, **bg**, **stop**, **kill**. All of them can take an argument which specifies the particular job, or they can have no argument. The argument can take two basic forms: a simple process ID number (as displayed by **ps**) or a form prefixed with a percent sign (%). If no argument is given, the current job is acted upon.

The % form of the argument can be **%-** where **-** indicates the previous job, %***n*** where ***n*** is the job number as displayed by the **jobs** command, %***pref*** where ***pref*** is some unique prefix of the command name and arguments of one of the jobs, or %**?*str*** where ***str*** is some unique string in one of the jobs.

You can use the **fg** command to move a suspended or background job into the foreground:

`% [fg] %[`*`job`*`]`

The **fg** is not mandatory. If the job specification is omitted, the current job will be brought into the foreground, and the next job becomes current.

Examples (note that the first `%` on each line represents the default **csh** prompt):

`% fg %5`	Bring job number 5 into the foreground
`% %1`	Bring job number 1 into the foreground
`% %`	Bring the current job into the foreground

After stopping a foreground job, you can start it running in the background with the **`bg`** command. **`bg`** puts the current or specified jobs into the background, continuing them if they were stopped. In the following commands, ***job*** stands for job number.

`% bg %[`*job*`]`

We described above how to stop (suspend) a foreground job with the suspend control character (<Ctrl-z>). Similarly, you can suspend a background job with the **`stop`** command:

`% stop %`*job*

You can abort a suspended or background job with the **`kill`** command:

`% kill %`*job*

If you attempt to exit a shell (logout) when there are stopped jobs, you will get a warning message. A second **`logout`** will log you out if you choose not to see what jobs are stopped before you exit. In the C shell family, background jobs will continue running after you log out.

5.5.3 Scheduling Jobs: at and cron

UNIX provides two methods for running jobs at some specified time.

AFS

If AFS is installed on your system, there are Kerberos authentication problems with running programs that spawn jobs external to your login process group (Kerberos authentication is described in section 7.3). **at** and **cron** fall into this category. You can run the job, but it will not run with authentication, and most likely will not be able to write into `/afs` space. A work-around is available for executing an authenticated **cron** job (send mail to *helpdesk@fnal* requesting the full details of this procedure).

at

The first is the **at** utililty. This allows the user to queue a job for later execution.

The format of the **`at`** command is:

`% at` *time* `[`*date*`] [+`*increment*`]`

at reads the commands from standard input. Standard output and standard error output will be mailed to you unless they are redirected.

The shell saves the environment variables and the working directory that are in effect at the time you submit the job and makes them available when the job is executed.

- The ***time*** can include 1, 2, or 4 numbers. One or two digits is assumed to be hours, four digits to be hours and minutes. It can be specified as two numbers separated by a colon (hours:minutes), either in 24-hour format or with *am* or *pm* appended. The names *noon*, *midnight*, *now*, and *next* are recognized.
- The ***date*** is either a month name followed by a day number (and optionally a year number followed by an optional comma) or a day of the week (fully spelled out or abbreviated to three characters). The words *today* and *tomorrow* are known. If no date is given, *today* is assumed if the hour is greater than the current hour and *tomorrow* if it is less.

• The optional ***increment*** is a number suffixed by *minutes*, *hours*, *days*, *weeks*, or *years* in singular or plural form.

Examples:

% at 8

% at 0800

% at 8:00am Jan 24

% at now + 1 minute

at reads from standard input, meaning you type in the commands (there may or may not be a prompt). When you are finished, terminate input with <Ctrl-d> followed by a carriage return.

You can also redirect the input to a file of commands, for example:

% at now + 1 hour < myscript

at runs in the Bourne shell (**sh**) by default. If you need to force it to run in C shell, you can use the trick illustrated in the following interactive example:

% /bin/csh << xxxxx

? at now + 2 minutes

? source .cshrc

? alias > aout

? <Ctrl-d> (followed by carriage return)

The first line causes **csh** (C shell) to read the following lines up to **xxxxx** or to the end-of-file. There is no **xxxxx**, of course, so it reads until you give it the <Ctrl-d>. The third line runs your `.cshrc`. It is an illegal Bourne shell command, therefore you can tell **at** ran in the C shell and that your `.cshrc` file was executed. You will receive a message similar to the following, and the results will be mailed to you (alas, **at** will say it's using `/bin/sh` even if you've "tricked" it):

```
warning: commands will be executed using /bin/sh
job 826157640.a at Wed Mar  6 18:14:00 1996
```

After 2 minutes, `aout` is mailed to you. It contains a list of all the aliases defined in your `.cshrc` file. If you are running a script using **at** then the script will be run under whatever shell you specify in the script.

For example, say you run:

% at now + 2 minutes

? ***script***

? <Ctrl-d> (followed by carriage return)

where ***script*** is a file that contains the line `#!/bin/csh` at the beginning. The commands in the script will execute under **csh**.

cron

The second method for running jobs at some specified time is the **`crontab`** command. It is designed for jobs that need to be run on a regular basis, e.g., once a night, or once per week. Note that **cron**, like **at**, uses the Bourne shell so that output redirection must be specified using Bourne shell syntax. Scripts will be run under what ever shell is specified in the script. If no shell is specified then Bourne shell is used.

The format of the command is:

`% crontab [`*filename*`]`

`% crontab [`*options*`]`

where ***filename*** is the name of a file containing the commands that you want to have executed. If you do not specify a file, then **`crontab`** will read commands from standard input as you type them, ending with <Ctrl-d>, and the commands will be run in Bourne shell. The system utility **cron** reads the crontab file and runs the commands. Standard output and standard error will be mailed to you unless they are redirected.

The command can also take the following options:

`-r`	remove crontab file
`-l`	list contents of crontab file

A **cron** file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

- minute (0-59)
- hour (0-23)
- day of the month (1-31)
- month of the year (1-12)
- day of the week (0-6 with 0=Sunday)

If an asterisk appears in a field instead of a number, **cron** interprets that as a wildcard for all possible values. The sixth field of a line in a **cron** file is a string that is executed by the shell at the specified times.

Examples:

The user creates a **cron** file `myfile`, and runs **`crontab`**:

```
#Myfile
# Run script that archives to 8mm tape for backup.
# Monday-Thursday at 2200 backup everything that has been
# changed.  Every Friday at 2200 backup everything.
0 22 * * 1-4 /usr/buckley/daily  1>>/usr/buckley/cron/backup.log 2>&1
0 22 * * 5   /usr/buckley/weekly 1>>/usr/buckley/cron/backup.log 2>&1
```

`% crontab myfile`

This command will perform an incremental backup at 10pm Monday to Thursday and a full backup at 10pm on Friday.