UNIX tutorial

After logging on

Throughout the Unix Tutorial section we will use % to indicate the computer's ``ready'' prompt.

ls

Let's try a simple command in a command window. Type ls and press . ls is the program to list files in a directory. Right now you may or may not see any files-not seeing any files doesn't mean you don't have any! Just plain ls won't list hidden files (files whose names start with ``.'', like .login). Now try typing:

```
% ls -a
```

Don't actually type the % symbol! Remember, that's the computer's prompt which indicates it is ready to accept input. The spacing should be exactly as shown. ls followed by a space, followed by a -a. The -a is a ``flag'' which tells the ls program to list all files.

For more about command flags see below.

cd

Just for fun, let's look at the contents of another directory, one with lots of files. Directory names in Unix are straightforward. They are all arranged in a tree structure from the root directory ``/''.

For now, use cd to change your directory to the /bin directory. Type:

```
% cd /bin
```

and press <CR>. Now type ls again. You should see a long list of files-in fact, if you look carefully you will see files with the names of the commands we've been typing (like ls and cd). Note that the /bin in the command we typed above was not a flag to cd. It was a ``parameter.'' Flags tell ommands how to act, parameters tell them what to act on.

Now return to your login directory with:

```
% cd
```

Entering cd with no parameter returns you to your home directory. You can check to make sure that it worked by entering:

```
% pwd
```

which prints your current (or ``working'') directory. The computer should return a line of words separated by ``/'' symbols which should look something like:

```
/amethyst9/home/username
```

Whatever it returns, the list should end in your username.

Using the On-line Man Pages

Most Unix commands have very short and sometimes cryptic names like ls. This can make remembering them difficult. Fortunately there are on-line manual pages which allow you to display information on a specific program (to list all the flags of ls, for example) or list all the information available on a certain topic.

man

To investigate other flags to the ls command (such as which flags will display file size and ownership) you would type man ls.

Using man and more

Try it now. Use man ls to find out how to make the ls program print the sizes of your files as well as their names. After typing man ls and pressing , note how man displays a screenful of text and then waits with a prompt --More-- at the bottom of the screen.

What man is doing is sending everything it wants to display to the screen through a program known as a ``pager'' The pager program is called more. When you see --More-- (in inverse video) at the bottom of the screen, just press the space-bar to see the next screenful. Press <CR> to scroll a line at a time.

Have you found the flag yet? The -s flag should display the size in kilobytes. You don't need to continue paging onc e you have found the information you need. Press q and more will exit.

Listing File Sizes

Now type ls -as. You can stack flags together like this-this tells ls to list all files, even hidden files, and list their sizes in kilobytes.

Directory and File Structure

When you list files in Unix, it is very hard to tell what kind of files they are. The default behavior of the ls program is to list the names of all the files in the current directory without giving any additional information about whether they are text files, executable files or directories!

This is because the ``meaning'' of the contents of each file is imposed on it by how you use the file. To the operating system a file is just a collection of bytes.

There is a program file which will tell you information about a file (such as whether it contains binary data) and make a good guess about what created the file and what kind of file it is.

File Names

Unlike other operating systems, filenames are not broken into a name part and a type part. Names can be many characters long and can contain most characters. Some characters such as * and ! have special meaning to the shell. They should not be used in filenames. If you ever do need to use such a symbol from the shell, they must be specified sneakily, by ``escaping'' them with a backslash, for example \!.

Directories

Directories in Unix start at the root directory ``/''. Files are ``fully specified'' when you list each directory branch needed to get to them.

```
/usr/local/lib/news

/home/pamela/src/file.c
```

The ``File System'' Tree Structure

Usually disks are ``partitioned'' into smaller sized sections called partitions If one partition of the disk fills up the other partitions won't be
affected.

Only certain large directory points are partitions and the choice of these points can vary among system managers. Partitions are like the larger branches of a tree. Partitions will contain many smaller branches (directories) and leaves (files).

The df Program

To examine what disks and partitions exist and are mounted, you can type the df command at the % prompt. This should display partitions which have names like /dev/sd3g---3 for disk 3, g for partition g. It will also display the space used and available in kilobytes and the ``mount point'' or directory of the partition.

Disk Space Maintenance

It's important to keep track of how much disk space you are using. The command du displays the disk usage of the current directory and all of its subdirectories. It displays the usage, in kilobytes, for each directory-including any subdirectories it contains-and ends by displaying the total.

```
% du
    display disk usage of current directory
% du -s
    display only total disk usage
% du -s -k
    some versions of Unix need -k to report kilobytes
```

Your Login Directory

A login directory can always be specified with ~username (~ is commonly called ``twiddle," derived from proper term ``tilde.") If you needed to list files in someone else's login directory, you could do so by issuing the command:

```
    % ls ~username
```

substituting in their username. You can do the same with your own directory if you've cd'd elsewhere. Please note-many people would consider looking at their files an invasion of their privacy; even if the files are not protected! Just as some people leave their doors unlocked but do not expect random bypassers to walk in, other people leave their files unprotected.

Subdirectories

If you have many files or multiple things to work on, you probably want to create subdirectories in your login directory. This allows you to place files which belong together in one distinct place.

Creating Subdirectories

The program to make a subdirectory is mkdir. If you are in your login directory and wish to create a directory, type the command:

```
    % mkdir directory-name
```

Once this directory has been created you can copy or move files to it (with the cp or mv programs) or you can cd to the directory and start creating files there.

Copy a file from the current directory into the new subdirectory by typing:

```
cp filename directory-name/new-filename
    copy file, give it a new name

cp filename directory-name
    copy file, filename will be the same as original
```

Or cd into the new directory and move the file from elsewhere:

```
% cd directory-name
% cp ../filename .
```

copies the file from the directory above giving it the same filename: ``." means ``the current directory"

Specifying Files

There are two ways you can specify files. Fully, in which case the name of the file includes all of the root directories and starts with ``/", or relatively, in which case the filename starts with the name of a subdirectory or consists solely of its own name.

When Charlotte Lennox (username lennox) created her directory arabella, all of the following sets of commands could be used to display the same file:

```
   % more lennox/arabella/chapter1
or
   % cd lennox
   % more arabella/chapter1
or
   % cd lennox/arabella
   % more chapter1
```

The full file specification, beginning with a ``/" is very system dependent. On our machines, all your user directories are in the /amethyst9 partition.

Protecting Files and Directories

When created, all files have an owner and group associated with them. The owner is the same as the username of the person who created the files and the group is the name of the creator's default login group, such as users, system etc. Most users do not belong to a shared group on our systems. If the creator of the file belongs to more than one group (you can display the groups to which you belong with the groups command) then the creator can change the group of the file between these groups. Otherwise, only the root account can change the group of a file.

Only the root account can change the ownership of a file.

Displaying owner, group and protection

The command ls -lg filename will list the long directory list entry (which includes owner and protection bits) and the group of a file.

The display looks something like:

```
protection  owner      group     filename
-rw-r-----  hamilton   ug        munster_village
```

The Protection Bits

The first position (which is not set) specifies what type of file this is. If it were set, it would probably be a d (for directory) or l (for link). The next nine positions are divided into three sets of binary numbers and determine protection to three different sets of people.

```
 u    g    o
rw-  r--  ---
 6    4    0
```

The file has ``mode'' 640. The first bits, set to ``r + w'' (4+2) in our example, specify the protection for the user who owns the files (u). The user who owns the file can read or write (which includes delete) the file.

The next trio of bits, set to 4, or ``r,'' in our example, specify access to the file for other users in the same group as the group of the file. In this case the group is ug-all members of the ug group can read the file (print it out, copy it, or display it using more).

Finally, all other users are given no access to the file.

The one form of access which no one is given, even the owner, is ``x'' (for execute). This is because the file is not a program to be executed-it is probably a text file which would have no meaning to the computer. The x would appear in the 3rd position and have a value of 1.

Changing the Group and the Protection Bits

The group of a file can be changed with the chgrp command. Again, you can only change the group of a file to a group to which you belong. You would type as follows:

```
% chgrp groupname filename
```

You can change the protection mode of a file with the chmod command. This can be done relatively or absolutely. The file in the example above had the mode 640. If you wanted to make the file readable to all other users, you could type:

```
% chmod 644  filename
```
or
```
% chmod +4  filename    (since the current mode of the file was 640)
```

For more information see the man page for chmod.

Default Protections: Setting the umask

All files get assigned an initial protection. To set the default initial protection you must set the value of the variable umask. umask must be defined once per login (usually in the .profile file). Common umask values include 022, giving read and directory search but not write permission to the group and others and 077 giving no access to group or other users for all new files you create.

The Unix Shell Syntax

As mentioned earlier, user commands are parsed by the shell they run. There are many shells other than the Korn shell which allow different types of shortcuts. We will only discuss the Korn shell here, but some alternate shells include the C-shell, Bourne shell (/bin/sh), the Bourne-Again Shell (bash), zsh and tcsh (a C shell variant).

The Path

One of the most important elements of the shell is the path. Whenever you type something at the % prompt, the Korn shell first checks to see if this is an ``alias'' you have defined, and if not, searches all the directories in your path to determine the program to run.

The path is just a list of directories, searched in order. Your default .profile file in your home directory will have a path defined for you. If you want other directories (such as a directory of your own programs) to be searched for commands, add them to your path by editing your .profile file. This list of directories is stored in the PATH environment variable. We will discuss how to manipulate environment variables later.

Flags and Parameters

Most commands expect or allow parameters (usually files or directories for the command to operate on) and many provide option flags. A ``flag'' as we saw before, is a character or string with a - before it-like the -s we used with the ls command.

Some commands, such as cp and mv require file parameters. Not surprisingly, cp and mv (the copy and move commands) each require two! One for the original file and one for the new file or location.

It would seem logical that if ls by itself just lists the current directory then cp filename should copy a file to the current directory. This is logical-but wrong! Instead you must enter cp filename . where the ``.'' tells cp to place the file in the current directory. filename in this case would be a long filename with a complete directory specification.

Not surprisingly ls . and ls are almost the same.

Creating Files

The cat Program

cat is one of most versatile commands. The simplest use of cat:

```
    % cat .profile
```

displays your .profile file to the screen. Unix allows you to redirect output which would otherwise go to the screen by using a > and a filename. You could copy your .profile, for example, by typing:

```
    % cat .profile > temp
```

This would have the same effect as:

```
    % cp .profile temp
```

More usefully cat will append multiple files together.

```
    % cat .profile .login > temp
```

will place copies of your .profile and .kshrc into the same file. Warning! Be careful not to cat a file onto an existing file! The command:

```
    % cat .profile > .profile
```

will destroy the file .profile if it succeeds.

If you fail to give cat a filename to operate on, cat expects you to type in a file from the keyboard. You must end this with a <Ctrl>-D on a line by itself. <Ctrl>-D is the end-of-file character.

By combining these two-leaving off the name of a file to input to cat and telling cat to direct its output to a file with > filename, you can create files.

For example:

```
    % cat > temp

    ;klajs;dfkjaskj
    alskdj;kjdfskjdf
    <Ctrl>-D
    %
```

This will create a new file temp, containing the lines of garbage shown above. Note that this creates a new file-if you want to add things on to the end of an existing file you must use cat slightly differently. Instead of > you'd use >> which tells the shell to append any output to an already existing file. If you wanted to add a line onto your .profile, you could type

```
% cat >> .profile
echo "blah blah blah"
<Ctrl>-D
%
```

This would append the line echo "blah blah blah" onto your .profile. Using > here would be a bad idea-it might obliterate your original .profile file.

Text Editors

cat is fine for files which are small and never need to have real changes made to them, but a full fledged editor is necessary for typing in papers, programs and mail messages. Among the editors available vi and emacs. We will not discuss emacs here.

Be careful! Not all Unix editors keep backup copies of files when you edit them.

vi

vi is an editor which has a command mode and a typing mode. When you first startup vi (with the command vi filename) it expects you to enter commands. If you actually want to enter text into your file, you must type the insert command i. When you need to switch back to command mode, hit the escape key, usually in the upper left corner of your keyboard.

To move around you must be in command mode. You can use the arrow keys or use j, k, h, l to move down, up, left and right.

For more information type man vi.

On the Suns, there is also an interactive text editor called “textedit”. You can start it by using the pull-down menu, or by typing “textedit &” in a command shell.

Files as Output and Log Files

Ordinarily there are two types of output from commands: output to standard output (stdout) and to standard error (stderr). The > and >> examples above directed only standard output from programs into files. To send both the standard output and error to a file when using the C shell, you should type >& :

```
% command >& filename
```

Logging Your Actions to a File

Sometimes you may wish to log the output of a login session to a file so that you can show it to somebody or print it out. You can do this with the script command. When you wish to end the session logging, type exit.

When you start up you should see a message saying script started, file is typescript and when you finish the script, you should see the message script done. You may want to edit the typescript file-visible ^M's get placed at the end of each line because linebreaks require two control sequences for a terminal screen but only one in a file.

Comparing Files

The basic commands for comparing files are:

cmp
: states whether or not the files are the same

diff
: lists line-by-line differences

comm
: three column output displays lines in file 1 only, file 2 only, and both files

See the man pages on these for more information.

Searching Through Files

The grep program can be used to search a file for lines containing a certain string:

```
 % grep  string filename
% grep  -i  string filename       (case insensitive match)
```

or not containing a certain string:

```
% grep  -v string filename
```

See the man page for grep---it has many useful options.

more and the vi editor can also find strings in files. The command is the same in both-type a /string when at the --More-- prompt or in vi command mode. This will scroll through the file so that the line with ``string'' in it is placed at the top of the screen in more or move the cursor to the string desired in vi. Although vi is a text editor there is a version of vi, view, which lets you read through files but does not allow you to change them.

The System and Dealing with Multiple Users

Most Unix commands which return information about how much CPU-time you've used and how long you've been logged in use the following meanings for the words ``job'' and ``process.''

When you log in, you start an interactive ``job'' which lasts until you end it with the logout command. Using a shell like Korn shell which has ``job-control'' you can actually start jobs in addition to your login job. But for the purposes of the most information returning programs, job (as in the ``JCPU'' column) refers to your login session.

Processes, on the other hand, are much shorter-lived. Almost every time you type a command a new process is started. These processes stay ``attached'' to your terminal displaying output to the screen and, in some cases (interactive programs like text editors and mailers) accepting input from your keyboard.

Some processes last a very long time-for example the /bin/ksh (Korn shell) process, which gets started when you login, lasts until you logout.

Information about Your Processes

You can get information about your processes by typing the ps command.

```
    PID TT STAT  TIME COMMAND
   9980 s9 S     0:06 -ksh (ksh)
  12380 s9 R     0:01 ps
```

The processes executing above are the C shell process and the ps command. Note that both commands are attached to the same terminal (TT), have different process identification numbers (PID), and have different amounts of CPU-time (TIME), accumulated.

Information about Other People's Processes

who

The simplest and quickest information you can get about other people is a list of which users are logged in and at which ``terminals'' (terminal here is either a terminal device line or telnet or rlogin session). The command to do this is who and it responds quickest of all the commands discussed here because it simply examines a file which gets updated everytime someone logs in or out.

Be careful though! This file, utmp, can get out of date if someone's processes die unexpectedly on the system. Any program which uses utmp to report information may list users who are not really logged in!

w

The w command is slower than the who command because it returns more information such as details about what programs people are running. It also returns a line containing the number of users and the system load average. The load average is the average number of processes ready to be run by the CPU and is a rough way of estimating how busy a system is.

w also uses the utmp file mentioned above. It takes longer than who because it then looks around and collects more information about the users it finds in the utmp file.

ps

The ps command used earlier to list your own processes can be used to list other users' processes as well. who and w list logins-but not individual processes on the system. They don't list any of the running operating system processes which start when the computer is booted and which don't have logins.

Since ps doesn't use utmp it is the program to use when you really want to find out what processes you might have accidentally left on the system or if another user is running any processes. Note that although ps might report processes for a user, it might be because that user has left a ``background job'' executing. In this case you should see a ``?'' in the TT field and the user won't really be logged in.

To get this fuller listing, give the flags -aux to ps. For more information on the uses of ps, type man ps.

finger

The finger program returns information about other users on the system who may or may not be logged in. finger by itself returns yet another variation of the list of currently logged in users. finger followed by a username or an e-mail -style address will return information about one or more users, the last time they logged into the system where you are fingering them, their full name, whether or not they have unread mail and, finally, the contents of two files they may have created: .plan and .project

For more information about using finger or ways to provide information about yourself to others, type man finger.

Shortcuts

If you use certain command flags regularly (-lga for ls) you can alias them to shorter commands. You can use wildcard symbols to refer to files with very long names. You can easily repeat commands you have already executed or modify them slightly and re-execute them.

Aliases

As mentioned above, you can alias longer commands to shorter strings. For example, ls -F will list all the files in the current directory followed by a trailing symbol which indicates if they are executable commands (a *) or directories (a /). If you wanted this to be the default behavior of ls you could add the following command to your .kshrc file in your home directory:

```
    % alias ls ls -F
```

To list the aliases which are set for your current process, type:

```
    % alias
```

without any parameters.

Example for contents of the .kshrc file

```
#
# Environment file for Korn shell
#

# Environment variables

export  EDITOR=$(whence vi) \
#default editor

      HISTFILE=~/.sh_history \
#command history file location

      HISTSIZE=128 \
#max no. of command logged in history

      PWD=$HOME
#initializes the PWD variable

# Put info and history number into prompt + other great stuff

export MACN="$(uname -n)"

if [[ "$TERM" = "xterm" ]]
then
      export PS1=' ]2;$MACN     $PWD     "'$LOGNAME [!]$ "
else
      export PS1="$(uname -n) [!]$ "
```

```
fi


# Aliases

alias ll='ls -al'
#lists all files, verbose

alias h='history -50'
#shows last 50 lines of history stack

alias cls=clear
#clear screen

alias lsd='ls -F | grep "/"'
#lists directories only

alias lsdr='ls -FR | grep "/"'
#recursive list of all directories under current

alias lsr='ls -l | grep ">"'
#shows links

alias lss='ls -l | sort +3 | more'
#sorts a list by owner

alias lsx='ls -F | grep "*"'
#lists the executables
```

Wildcards

Wildcards are special symbols which allow you to specify matches to letters or letter sequences as part of a filename.

Some examples:

```
*
    The basic wildcard character. Beware rm *!!
    ls *.dat
        lists all files ending in .dat
    ls r*
        lists all files starting with r
?
    a one character wildcard.
    ls ?.dat
```

lists 5.dat, u.dat, but not 70.dat

```
[]
    limits a character to match one of the characters between the brakets
    ls *.[ch]
        lists all .h and .c files
    more [Rr][Ee][Aa][Dd][Mm][Ee]
        mores the files README, readme,ReadMe, and Readme, among others
```

Directory Specifications

You've already met the shortcut. The two other important directory symbols are ``." for the current directory and ``.." for the previous (parent) directory.

```
    % cd ..
```

moves you out of a subdirectory into its parent directory.

Environment Variables

Environment variables are pieces of information used by the shell and by other programs. One very important one is the PATH variable mentioned earlier. Other important variables you can set include:

```
    EDITOR
    TERM
    MAIL
```

To see what environment variables are set and what they are set to, type the command printenv. To set a variable, use the export command as in the example below.

```
    % TERM=vt100
    % EDITOR=vi
```

Many programs mention environment variables you may want to set for them in their man pages. Look at the ksh man page for some of the standard ones.

History

Most shells allow ``command line editing" of some form or another-editing one of the previous few lines you've typed in and executing the changed line. You can set a history ``environment variable" to determine how many previous command lines you will have access to with set history=40

History and command recall mechanisms

Recalled commands are retrieved from the history file starting from the most recent up to the oldest previously entered commands. Search strategies may follow vi or emacs syntax and semantics (only vi mode will be explained below). First of all, pressing the ESC key will make the Korn shell enter the vi command recall/editing mode. At that point, you may enter whatever vi searching command you like to look for any previously entered command. (You have to think about your history file as an edited file that will be scanned by vi or emacs commands in the reverse order, that is from the bottom of the file back to the top of it). The vi/emacs editor recalling command, which can be repeated, will in place display successive search results; if you hit carriage-return at the time the
displayed recalled command is the one you want, then that command will be executed.

Examples :

having pressed the ESC key, any - key hit will sequentially display in back order all the previously entered commands (that is, 1st -
will display the last but 1, 2nd - will display the last but 2, and so on).
having pressed the ESC key, and a number of times the - key, pressing the + key will return one step forwards towards the most
recent command.
let say that you have some time ago used a given command, and you want to recall it for execution. All what you have to do is to
make a search string operation through your history file. For example, to recall the last qsub command, you press ESC, then /qsub.
The vi / search string delimiter will make the Korn shell find the last entered qsub command in your history file.
to continue the search operation, still in the reverse order, press the n key. To continue the search towards the most recent, press N.
You may do this if you want to find another instance of the qsub command.
of course, you may recall a command by only entering one of its used parameter, as in :

```
ESC/-m
```

will retrieve for example the following command : qstat -m
if you want to recall a command searching only on its first few characters, you will do (the caret ^ character enforces the match to
begin at the beginning of the line) :

```
ESC/^string
```

as an alternative of the above, you can merely do

```
r string
```

This might be used to redo a command given its number, as it appeared in the prompt.

Example :

```
r 95
```

will redo the command that was executed at line 95

etc

Shell Vi Editing Mode Summary

This is our default mode to repeat or edit previous commands. The table below shows some of the most useful commands.

Key	Brief Description	Key	Brief Description
l	Moves forward one character	h	Moves back one character
^	Moves to the start of the line	$	Moves to the end of the line
x	Deletes the current character	dw	Deletes the current word
db	Deletes the previous word	~	Changes case of current character
d$	Deletes from to end of line	\	Do filename completion
[RETURN]	Executes the current line	k or -	Fetches the previous command
j or +	Fetches the next command line	v	Run full vi session on current line
A	Inserts text at end of line	i	Inserts text before current character
w	Moves forward one word	b	Moves back one word

Job Control

It is very easy to do many things at once with the Unix operating system. Since programs and commands execute as independent processes you can run them in the ``background'' and continue on in the foreground with more important tasks or tasks which require keyboard entry.

For example, you could set a program running in the background while you edit a file in the foreground.

The fg and bg Commands

When you type <Ctrl>-Z whatever you were doing will pause. If you want the job to go away without finishing, then you should kill it with the command kill %. If you don't want it paused but want it to continue in the foreground-that is, if you want it to be the primary process to which all the characters you type get delivered-type fg. If you want it to continue processing in the background while you work on something else, type bg.

You should not use bg on things which accept input such as text editors or on things which display copious output like more or ps.

What to Do When You've Suspended Multiple Jobs

If you've got several processes stopped-perhaps you are editing two files or you have multiple telnet or rlogin sessions to remote computers-you'll need some way of telling fg which job you want brought to the foreground.

By default fg will return you to the process you most recently suspended. If you wanted to switch processes you would have to identify it by its job number. This number can be displayed with the jobs command. For example:

```
% jobs
[1]        Stopped     vi .login
[2]   +     Stopped     rn
[3]        Running     cc -O -g test.c
%
```

The most recently suspended job is marked with a + symbol. If you wanted to return to job one instead, you would type:

```
% fg %1
```

You can type %1 as a shortcut.

Starting Jobs in the Background

Some jobs should start in the background and stay there-long running compilations or programs, for example. In this case you can direct them to the background when you start them rather than after they have already begun. To start a job in the background rather than the foreground, append an & symbol to the end of your command.

You should always run background processes at a lower priority by using the nice command. Non-interactive jobs are usually very good at getting all the resources they need. Running them at a lower priority doesn't hurt them much-but it really helps the interactive users-people running programs that display to terminal screens or that require input from the keyboard.

If you need to run CPU-intensive background jobs, learn about how to control the priority of your jobs by reading the manual pages (man nice and man renice).

Suspend, z and <Ctrl>-Z

Some programs provide you with special ways of suspending them. If you started another shell by using the ksh command, you would have to use the suspend command to suspend it.

If you wish to suspend a telnet or rlogin session you must first get past the current login to get the attention of the telnet or rlogin program.

Use (immediately after pressing a return) to get rlogin's attention. <Ctrl>-Z will suspend an rlogin session.

Use <Ctrl>-] to get telnet's attention <Ctrl>-]z will suspend a telnet session.

Some Common and Useful Unix Commands For Files

cp

The cp command allows you to create a new file from an existing file. The command line format is:

```
% cp input-file-spec output-file-spec
```

where input-file-spec and output-file-spec are valid Unix file specifications. The file specifications indicate the file(s) to copy from and the file or directory to copy to (output). Any part of the ilename may be replaced by a wildcard symbol (*) and you may specify either a filename or a directory for the output-file-spec. If you do not specify a directory, you should be careful that any wildcard used in the input-file-spec does not cause more than one file to get copied.

```
% cp new.c old.c
% cp new.* OLD (where OLD is a directory)
```

ls

command allows the user to get a list of files in the current default directory. The command line format is:

```
% ls file-spec-list
```

where file-spec-list is an optional parameter of zero or more Unix file specifications (separated by spaces). The file specification supplied (if any) indicates which directory is to be listed and the files within the directory to list.

lpr

The lpr command tells the system that one or more files are to be printed on the default printer. If the printer is busy with another user's file, an entry will be made in the printer queue and the file will be printed after other lpr requests have been satisfied. The command line format is:

BLOCKQUOTE> % lpr file-spec-list

where file-spec-list is one or more Unix files to be printed on the default printer. Any part of the filenames may be replaced by a wild card.

Here is more information about where the printers actually are and what kind of printers are available.

man

The man command is a tool that gives the user brief descriptions of Unix commands along with a list of all of the command flags that the command can use. To use man, try one of the following formats:

```
% man  command
% man  -k  topic
```

more

The more command will print the contents of one or more files on the user's terminal. The command line format is:

```
% more file-spec-list
```

more displays a page at a time, waiting for you to press the space-bar at the end of each screen. At any time you may type q to quit or h to get a list of other commands that more understands.

mv

The mv command is used to move files to different names or directories. The command line syntax is:

```
% mv input-file-spec output-file-spec
```

where input-file-spec is the file or files to be renamed or moved. As with cp, if you specify multiple input files, the output file should be a directory. Otherwise output-file-spec may specify the new name of the file. Any or all of the filename may be replaced by a wild card to abbreviate it or to allow more than one file to be moved. For example:

```
% mv data.dat ./research/datadat.old
```

will change the name of the file data.dat to datadat.old and place it in the subdirectory research. Be very careful when copying or moving multiple files.

rm

The rm command allows you to delete one or more files from a disk. The command line format is:

```
% rm file-spec-list
```

where file-spec-list is one or more Unix file specifications, separated by spaces, listing which files are to be deleted. Beware of rm *! For
example:

```
% rm *.dat able.txt
```

will delete the file able.txt and all files in your current working directory which end in .dat. Getting rid of unwanted subdirectories is a little more difficult. You can delete an empty directory with the command rmdir directory-name but you cannot use rmdir to delete a directory that still has files in it.

To delete a directory with files in it, use rm with the -r flag (for recursive).

Korn Shell Script Programming (ksh)

Overview

The UNIX operating system offers a number of interactive environments for users, known as shells. The Bourne shell (sh) represents the oldest and most commonly used shell. The Korn shell (ksh) is a modern shell with many advanced editing and programming features. Each shell offers a programming language with a unique set of semantic and syntactic features. This document provides a tutorial on shell programming with the Bourne and Korn shells. Since the Bourne shell programming language provides a subset of the Korn shell language, greater emphasis will be placed on ksh programming.

This document assumes that the reader is already familiar with non-programming aspects of the sh and ksh shells, such as: environment variables, history manipulation, command substitution, pattern matching and wild-cards, and resource files.

Script Basics

The word script is used to indicate a shell program. Shell scripts differ from other programming languages (such as C and Fortran) in that they are interpreted by a shell instead of being compiled into machine executable code.

A script is a plain text file which contains names of other programs to be executed and [optionally] shell directives which can affect the execution of programs. Scripts can be created with editors such as vi and emacs, both of which are commonly available on UNIX systems.

Below is an example of a simple script which, when invoked appropriately, executes the commands finger, date and w sequentially:

```
finger
date
w
```

We'll assume that this information has been stored in a file called fdw. This script has no shell specific commands in it, only program names. For this reason the fdw script can be executed by any shell, including sh, ksh, and csh (csh is known as the C Shell, and is also in common use on many UNIX installations).

We might expect that by typing fdw at our shell prompt the three commands contained therein would be executed. As yet this is not the
case:

```
sunrise$ dw

fdw: Permission denied.
```

Executing Scripts

There are two methods of executing a shell script:

1.The first method involves passing the script as an argument to a shell such as sh or ksh:

```
sunrise$ /bin/ksh fdw
```

Here the contents of the script are read in by the shell, interpreted and executed a line at a time. This method of script execution has one major drawback: It assumes the end user of the shell script knows which shell to execute the script with. Since our fdw script contains no shell specific syntax this is not an issue for this example.

2.The second, slightly more sophisticated method of executing a script involves changing the access permissions of the fdw script so that it is executable. This can be accomplished by way of the chmod command:

```
sunrise$ chmod u+x fdw
```

Now it is possible to type fdw at the shell prompt and see the expected output.

Comments

Adding comments to a shell script is achieved by placing a # character in front of the non-executable text. It is always a good idea to document the intent and purpose of any shell script you develop. Below is our fdw script with a comment added:

```
# This shell executes finger, date and w in order
finger
date
w
```

Shells also understand a special comment that specifies the path of the shell with which a script is to be executed. This comment must begin at row 1, column 1 of the script and starts with the #! characters followed by the path of the shell to be executed. For example, to ensure that the fdw script is executed by the Korn shell we would change fdw as shown below:

```
#!/bin/ksh
# This shell executes finger, date and w in order
finger
date
w
```

The importance of the #! comment will become evident when we introduce Korn shell specific syntax that is not understood by sh.

Ksh preparedness

Here are the most important things to know and do, before really getting serious about shellscripting.

All examples given should be put into some file, created by a text editor such as “textedit” on the sun. You can then run it with "ksh file".Or, do the more official way; Put the directions below, exactly as-is, into a file, and follow the directions in it.

```
#!/bin/ksh
# the above must always be the first line. But generally, lines
# starting with '#' are comments. They dont do anything.
# This is the only time I will put in the '#!/bin/ksh' bit. But
# EVERY EXAMPLE NEEDS IT, unless you want to run the examples with
#  'ksh filename' every time.
#
# If for some odd reason, you dont have ksh in /bin/ksh, change
# the path above, as appropriate.
#
# Then do  'chmod 0755 name-of-this-file'. After that,
# you will be able to use the filename directly like a command

echo Yeup, you got the script to work!
```

Understand variables

Hopefully, you already understand the concept of a variable. It is a place you can store a value to, and then do operations on "whatever is
in this place",vs the value directly.

In shellscripts, generally, a collection of letters and/or numbers [aka a 'string'] can be in a variable, as well as just plain numbers.

You set a variable by using

```
variablename="some string here"
  OR
variablename=1234
```

You access what is IN a variable, by putting a dollar-sign in front of it.

```
echo $variablename
  OR
echo ${variablename}
```

If you have JUST a number in a variable, you can do math operations on it. But that comes later on in this tutorial.

Put everything in appropriate variables

Well, okay, not EVERYTHING :-) But properly named variables make the script more easily readable. There isn't really a 'simple' example for this, since it is only "obvious" in large script. So either just take my word for it, or stop reading and go somewhere else now!

An example of "proper" variable naming practice:

```
#Okay, this script doesnt do anything useful, its just for demo purposes
INPUTFILE="$1"
USERLIST="$2"
OUTPUTFILE="$3"

count=0

while read username ; do
        grep $username $USERLIST >>$OUTPUTFILE
        count=$(($count+1))
done < $INPUTFILE
```

```
echo user count is $count
```

While the script may not be totally readable to you yet, I think you'll agree it is a LOT clearer than the following;

```
i=0
while read line ; do
        grep $line $2 >> $3
        i=$(($i+1))
done <$1
echo $i
```

Note that '$1' means the first argument to your script.
'$*' means "all the arguments together
'$#' means "how many arguments are there?"

Know your quotes

It is very important to know when, and what type, of quotes to use.
Quotes are generally used to group things together into a single entity.

Single-quotes are literal quotes.
Double-quotes can be expanded

```
 echo "$PWD"
```

prints out your current directory

```
 echo '$PWD'
```

prints out the string $PWD

```
 echo $PWDplusthis
```

prints out NOTHING. no such variable "PWDplusthis

```
 echo "$PWD"plusthis
```

prints out your current directory, and the string "plusthis" immediately following it. You could also accomplish this with the alternate form of
using variables,

```
 echo ${PWD}plusthis
```

Ksh basics

This is a quickie page to run through basic "program flow control" commands, if you are completely new to shell programming. The basic ways to shape a program, are loops, and conditionals. Conditionals say "run this command, IF some condition is true". Loops say "repeat
these commands" (usually, until some condition is met, and then you stop repeating.

Conditionals

IF

The basic type of condition is "if".

```
if [[ $? -eq 0 ]] ; then
        print we are okay
else
        print something failed
fi
```

IF the variable $? is equal to 0, THEN print out a message. Otherwise (else), print out a different message.

The final 'fi' is required. This is to allow you to group multiple things together. You can have multiple things between if and else, or between else and fi, or both. You can even skip the 'else' altogether!

```
if [[ $? -eq 0 ]] ; then
        print we are okay
        print We can do as much as we like here
fi
```

case

The case statement functions like 'switch' in some other languages. Given a particular variable, jump to a particular set of commands, based on the value of that variable.

While the syntax is similar to C on the surface, there are some major differences;

- The variable being checked can be a string, not just a number
- There is no "fall through". You hit only one set of commands
- To make up for no 'fall through', you can 'share' variable states
- You can use WILDCARDS to match strings

```
echo input yes or now
```

```
read  answer
case $answer in
        yes|Yes|y)
                echo got a positive answer
                # the following ';;' is mandatory for every set
                # of comparative xxx)  that you do
                ;;
        no)
                echo got a 'no'
                ;;
        *)
                echo This is the default clause. we are not sure why or
                echo what someone would be typing, but we could take
                echo action on it here
                ;;
esac
```

Loops

while

The basic loop is the 'while' loop; "while" something is true, keep looping.

There are two ways to stop the loop. The obvious way is when the 'something' is no longer true. The other way is with a 'break' command.

```
keeplooping=1;
while [[ $keeplooping -eq 1 ]] ; do
        read quitnow
        if [[ "$quitnow" = "yes" ]] ; then
                keeplooping=0
        fi
        if [[ "$quitnow" = "q" ]] ; then
                break;
        fi
done
```

until

The other kind of loop in ksh, is 'until'. The difference between them is that 'while' implies looping while something remains true. 'until', implies looping until something false, becomes true

```
until [[ $stopnow -eq 1 ]] ; do
        echo just run this once
        stopnow=1;
```

```
    echo we should not be here again.
done
```

for

A for loop, is a "limited loop". It loops a specific number of times, to match a specific number of items. Once you start the loop, the number of times you will repeat is fixed.

The basic syntax is

```
for var in one two three ; do
    echo $var
done
```

Whatever name you put in place of 'var', will be updated by each value following "in". So the above loop will print out

```
one
two
three
```

But you can also have variables defining the item list. They will be checked ONLY ONCE, when you start the loop.

```
list="one two three"
for var in $list ; do
    echo $var
    # Note: Changing this does NOT affect the loop items
    list="nolist"
done
```

The two things to note are:

1. It stills prints out "one" "two" "three"
2. Do NOT quote "$list", for multiple items.

If you used "$list", it would print out a SINGLE LINE, "one two three"