Introduction to Shell and Common Unix Commands

Introduction to Unix Shell

- A shell executes all of the commands that you enter. Some popular shells:
  - Bourne shell (Bourne Again shell in Linux)
  - Korn shell
  - C shell (TC shell in Linux)
- All of these shells share a similar set of core functionality, together with some specialized properties.

Shell (con't)

- Each shell has its own programming language.
- Why would you write a program in a shell language rather than a language like C or Java?
  - The answer is that shell languages are tailored to manipulating files and processes in the UNIX system, which makes them more convenient in many situations.

Shell Operations

- A typical shell operation:
  - Shell starts up and initializes itself
  - Shell types a prompt character (percent or dollar sign) and waits for user to type a command line
  - User types a command line
  - Shell extracts first word and assumes the name of a program to be run
  - Shell searches for this program
    - If it finds it, runs the program
  - Shell suspends itself until the program terminates
  - Shell waits for another command

Shell Operations

- When a shell is invoked automatically during a login (or manually from a keyboard or script), it follows a preset sequence:
  1. It reads a special start-up file, typically located in the user's home directory (e.g. ~/.cshrc), that contains some initialization information. Each shell's start-up sequence is different.
  2. It displays a prompt and waits for a user command.
  3. If the user enters a Control-D character on a line of its own, this is interpreted by the shell as meaning "end of input" and causes the shell to terminate; otherwise, the shell executes the user's command and returns to step 2.

Shell Operations

- Shell is an ordinary user program – needs the ability to read from and write to the terminal, and execute other programs
- Commands may take arguments, which are passed to the called program as character strings
- Example
  - ***cp src dest***
- Shell accepts magic characters, called wild cards
  - Example: ***ls *.c***

Shell Operations

- Commands range from simple utility invocations, such as

  cuse93:> ls

  to complex-looking pipeline sequences, such as

  cuse93:> ps -ef | sort | ul -tdumb | lp

Shell Operations

- If you ever need to enter a command that is longer than a line on your terminal, you may terminate a portion of the command with a backslash (\) character, and the shell then allows you to continue the command on the next line:

  cuse93:> *echo this is a very long shell command and needs to *
  *be extended with the line continuation character. Note *
  that a single command may be extended for several lines.

  this is a very long shell command and needs to be extended with the line continuation character. Note that a single command may be extended for several lines.

  cuse93:> _

Executable Files Versus Built-in Commands

- Most UNIX commands invoke utility programs that are stored in the directory hierarchy. Utilities are stored in files that have execute permission. For example, when you type
  cuse93:> ls
  the shell locates the executable program called “ls,” which is typically found in the *“/bin”* directory, and executes it.
- In addition to its ability to locate and execute utilities, the shell contains several built-in commands, such as *echo* and *cd*, which it recognizes and executes internally.

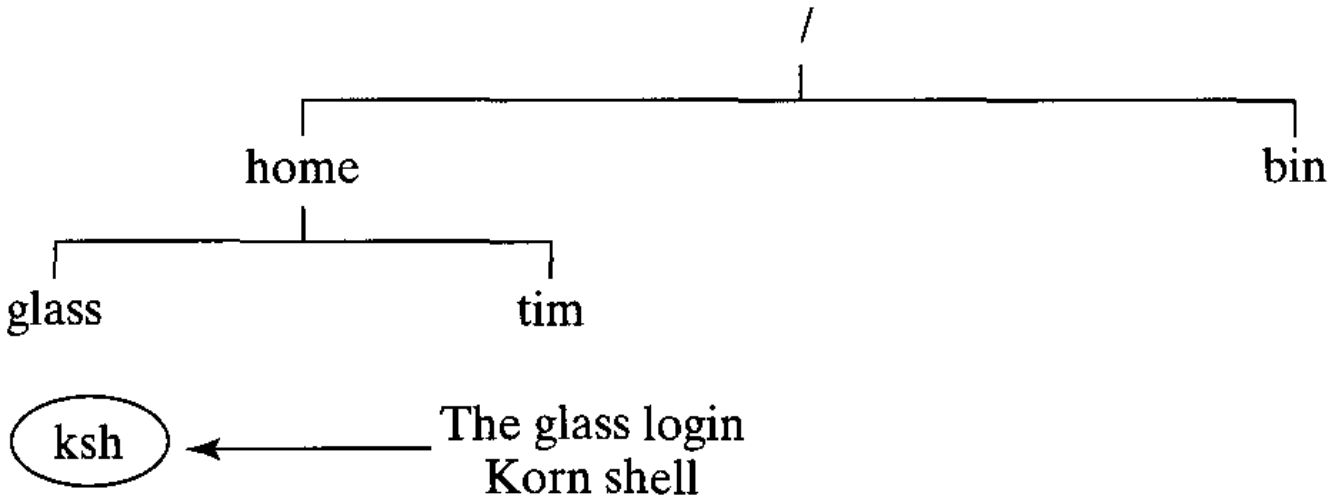
Showing Your Shell's Current Working Directory: pwd

- Every UNIX process has a location in the directory hierarchy, termed its *current working directory.*
- When you log into a UNIX system, your shell starts off in a particular directory called your *home directory*.
- In general, every user has a different home directory assigned by the system administrator.
- To show your shell's current working directory, use the **pwd** utility.

```
UNIX(r) System V Release 4.0
login: glass
Password:                    . . .secret
cuse93:> pwd
/home/glass
cuse93:>
```

Absolute And Relative Pathnames

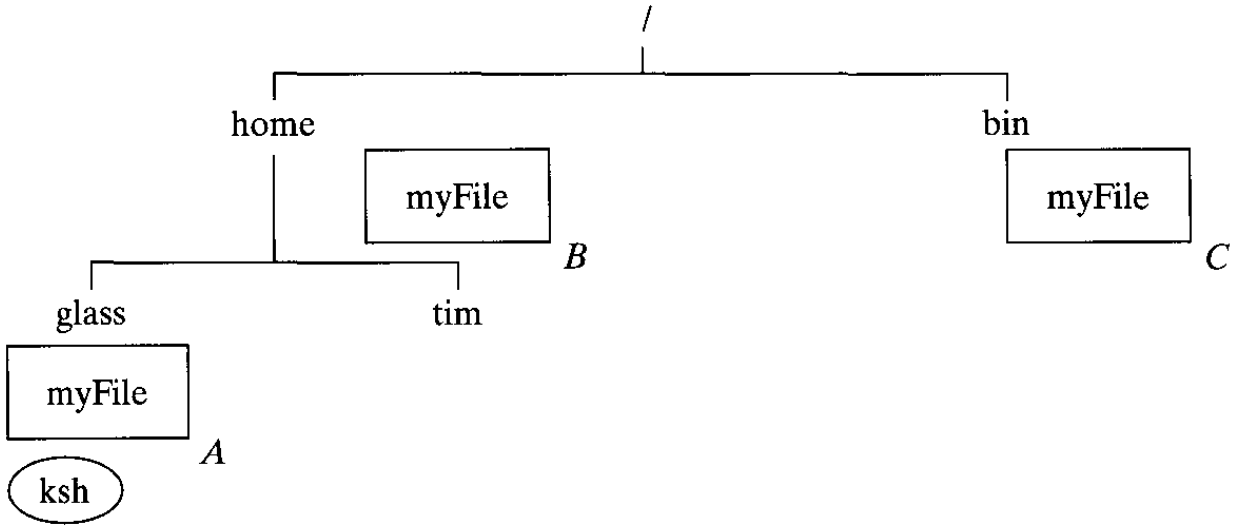
The following diagram indicates that the location of my login Korn shell is in the directory hierarchy.



The login shell starts at the user's home directory.

Absolute And Relative Pathnames

- Two files in the same directory may not have the same name, although it's perfectly OK for several files in *different* directories to have the same name.
- For example, this diagram shows a small hierarchy that contains a "ksh" process and three files called "myFile".



Different files may have the same name.

Absolute And Relative Pathnames (con't)

- Although these files have the same name, they may be unambiguously specified by their *pathname* relative to "/", the root of the directory hierarchy.
- A pathname is a sequence of directory names that leads you through the hierarchy from a starting directory to a target file.
- A pathname relative to the root directory is often termed an absolute or full pathname. The table shows the absolute pathnames of the "A," "B," and "C" instances of "myFile".

File	Absolute PathName
A	/home/glass/myFile
B	/home/myFile
C	/bin/myFile

Absolute And Relative Pathnames (con't)

- We can specify a pathname relative to the *home directory* of an account using the character "~"

Pathname	Meaning
~wlam	home directory of the account wlam
~/	home directory of the current user account
~/distribute/	"distribute" directory under the home Directory of the current user account

Absolute And Relative Pathnames (con't)

- A pathname can be specified *relative* to its current working directory.
- The UNIX file system supports the special fields shown below that may be used when supplying a relative pathname.

Field	Meaning
.	current directory
..	parent directory

Absolute And Relative Pathnames (con't)

- For example, the following table shows the pathnames of the three instances of "myFile" relative to the "ksh" process located in the "/home/glass" directory.
- Note that the pathname "myFile" is equivalent to "./myFile," although the second form is rarely used because the leading "." is redundant.

File	Relative PathName
A	myFile
B	../myFile
C	../../bin/myFile

Unix Programmer Support

- UNIX caters very well to programmers.
- Unix is an example of an “open” system
  - the internal software architecture is well documented and available in source code form, either free of charge or for a relatively small fee.
- The features of UNIX, are all easily accessible from a programming language such as C via a set of library routines known as “system calls.”
  - such as parallel processing, inter-process communication and file handling

End Of Input : Control-D

- Many UNIX utilities take their input from either a file or the keyboard. If you instruct a utility to do the latter, you must tell the utility when the input from the keyboard is finished. To do this, type *Control-D* on a line of its own after the last line of input.
- *Control-D* means “end of input.” For example, the **mail** utility allows you to send mail from the keyboard to a named user:

cuse93:> *mail tim* ...send mail to my friend Tim.
Hi Tim ...input is entered from the keyboard.
I hope you get this piece of mail. How about building a country one of these days?
- with best wishes from Graham
^D ...tell the terminal that there’s no more input.
cuse93:> _

Unix - Listing The Contents Of A Directory: ls

- Suppose that we use an editor to edit a file named “heart” with the following content

> *I hear her breathing,*
> *I’m surrounded by the sound.*
> *Floating in this secret place,*
> *I never shall be found.*

- We can use the **ls** utility to list the name and other information about a file or a directory.

```
cuse93:> ls                                  ... list all files in current directory
heart
cuse93:>  ls -l heart                        ... long listing of “heart”
-rw-r--r--  1  glass  106   Jan 30 19:46   heart
```

Unix - Listing The Contents Of A Directory: ls (con't)

- With no arguments at all, **ls** lists all of the files in the current working directory in alphabetical order, excluding files whose name starts with a period.
- To obtain a listing of directories other than the current directory, place their names after the options. To obtain listings of specific files, place their names after the options.
- The -l option generates a long listing, including file type, permission flags, the file's owner, and the last time the file was modified.

Unix - Listing The Contents Of A Directory: ls (con't)

Field #	Field value	Meaning
1	-rw-r--r--	the type and permission mode of the file, which indicates who can read, write, and execute the file
2	1	the hard link count
3	glass	the username of the owner of the file
4	106	the size of the file, in bytes
5	Jan 30 19:46	the last time the file was modified
6	heart	the name of the file

Unix - Listing A File: cat and more

- We can use the **cat** utility to display the content of a text file.
- An example:

```
cuse93:>  cat heart                    ... list the contents of the "heart" file.
I hear her breathing,
I'm surrounded by the sound.
Floating in this secret place,
I never shall be found.
cuse93:>
```

- **cat** is good for listing small files, but doesn't pause between full screens of output.
- The **more** utility is better suited for larger files and contains advanced facilities such as the ability to scroll backward through a file.

Listing A File: cat/more/page/head/tail

- To check the contents of the “heart” file that I had created in my home directory, “/home/glass”, I directed its contents to the screen using the **cat** utility. Notice that I supplied cat with the name of the file that I wanted to display:

```
cuse93:>  cat heart                    ... list the contents of the “heart” file.
I hear her breathing,
I’m surrounded by the sound.
Floating in this secret place,
I never shall be found.
cuse93:>
```

- **cat** can actually take any number of files as arguments, in which case they are listed together, one following the other. **cat** is good for listing small files, but doesn’t pause between full screens of output.
- The **more** and **page** utilities are better suited for larger files and contain advanced facilities such as the ability to scroll backward through a file.

Listing A File: cat/more/page/head/tail (con't)

- *Utility:* **more -f** [+*lineNumber*] {*fileName*}*
- The **more** utility allows you to scroll through a list of files, one page at a time. By default, each file is displayed starting at line 1, although the + option may be used to specify the starting line number. The **-f** option tells **more** not to fold long lines. After each page is displayed, **more** displays the message "--More--" to indicate that it's waiting for a command. To list the next page, press the space bar. To list the next line, press the *Enter* key. To quit **more,** press the q key. To obtain help on the multitude of other commands, press the h key.

- *Utility:* **page** -f [+*lineNumber*] {*fileName*}*
- The **page** utility works just like **more,** except that it clears the screen before displaying each page. This sometimes makes the listing display a little more quickly. While we're on the topic of listing files, there are a couple of handy utilities called **head** and tail that allow you to peek at the start and end of a file, respectively.

Listing A File: cat/more/page/head/tail (con't)

- *Utility:* **head** -n {*fileName*}*
- The **head** utility displays the first *n* lines of a file. If *n* is not specified, **head** defaults to 10. If more than one file is specified, a small header identifying each file is displayed before its contents are.

- *Utility:* **tail -n** {*fileName*}*
- The **tail** utility displays the last *n* lines of a file. If *n* is not specified, tail defaults to 10. If more than one file is specified, a small header identifying each file is displayed before its contents are. In the following example, I displayed the first two lines and last two lines of my "heart" file:

```
cuse93:> head -2 heart                ... list the first two lines
I hear her breathing,
I'm surrounded by the sound.
cuse93:> tail -2 heart                ... list the last two lines
Floating in this secret place,
I never shall be found.
cuse93:>
```

Filename Substitution (Wildcards)

- All shells support a wildcard facility that allows you to select files from the file system that satisfy a particular name pattern.
- Any word on the command line that contains at least one of the wildcard metacharacters is treated as a pattern and is replaced by an alphabetically sorted list of all the matching filenames.
- This act of pattern replacement is called globbing.

Filename Substitution (Wildcards)

- The wildcards and their meaning are shown in the following table

Wildcard	Meaning
*	Matches any string, including the empty string.
?	Matches any single character.
[..]	Matches any one of the characters between the brackets. A range of characters may be specified by separating a pair of characters by a dash.

Filename Substitution (Wildcards)

- Some examples of wildcard usage

cuse93:> *ls *.c*

file1.c axe.c go.c queue.c KIT.c on.c

cuse93:> *ls ??.c*

go.c on.c

cuse93:> *ls [ac]** … *any pattern beginning with a or c*

axe.c c.txt abc

cuse93:> *ls [a-z]** … *any pattern beginning with small letter*

file1.c axe.c c.txt abc axe.c go.c on.c

Renaming A File: mv

- Now that I'd created the first draft of my lyrics, I wanted to create a few more experimental versions. To indicate that the file "heart" was really the first generation of many versions to come, I decided to rename it "heart.ver1" by using the **mv** utility.
- Here's how I renamed the file using the first form of the **mv** utility:

```
cuse93:> mv heart heart.ver1          ...rename to "heart.ver1".
cuse93:> ls
heart.ver1
cuse93:>
```

Renaming A File: mv (con't)

- *Utility:* **mv** -i *oldFileName newFileName*
  mv -i {*fileName*}* *directoryName*
  mv -i *oldDirectoryName newDirectoryName*
- The first form of **mv** renames *oldFileName* as *newFileName.* If the label *newFileName* already exists, it is replaced.
- The second form allows you to move a collection of files to a directory.
- The third allows you to move an entire directory None of these options actually moves the physical contents of a file if the destination location is within the same file system as the original; instead, they just move labels around the hierarchy.
- **mv** is therefore a very fast utility. The -i option prompts you for confirmation if *newFileName* already exists.

Making A Directory : mkdir

- Rather than clog up my home directory with the many versions of “heart,” I decided to create a subdirectory called “lyrics” in which to keep them all. To do this, I used the mkdir utility. Here’s how I did it:

```
cuse93:> mkdir lyrics                ...create a directory called “lyrics”
cuse93:> ls  -lF                     ...confirm.
-rw-r--r--      1     glass     106   Jan 30    23:28    heart.ver1
drwxr-xr-x      2     glass     512   Jan 30    19:49    lyrics/
cuse93:>
```

- *Utility:* **mkdir** [-p] *newDirectoryName*
- The mkdir utility creates a directory. The -p option creates any parent directories in the *newDirectoryName* pathname that do not already exist. If *newDirectoryName* already exists, an error message is displayed and the existing file is not altered in any way.

Making A Directory : mkdir (con't)

- The letter "d" at the start of the permission flags of "lyrics" indicates that it's a directory file.
- In general, you should keep related files in their own separate directory. If you name your directories sensibly, it'll make it easy to track down files weeks, or even years, after you create them.
- Once the "lyrics" directory was created, the next step was to move the "heart.ver1" into its new location. To do this, I used **mv** and confirmed the operation by using ls:

```
cuse93:> mv   heart.ver1  lyrics        ...move into "lyrics"
cuse93:> ls                             ... list the current directory.
lyrics                                  ..."heart.ver1" has gone.
cuse93:> ls  lyrics                     ...list the "lyrics" directory.
heart.ver1                             ..."heart.ver1" has moved.
cuse93:>
```

Moving To A Directory : cd

- Although I could remain in my home directory and access the various versions of my lyric files by preceding them with the prefix “lyrics”, doing this would be rather inconvenient. For example, to edit the file “heart.ver1” with the UNIX vim editor, I’d have to do the following:

```
cuse93:> vim  lyrics/heart.ver1          …invoke the vim editor
```

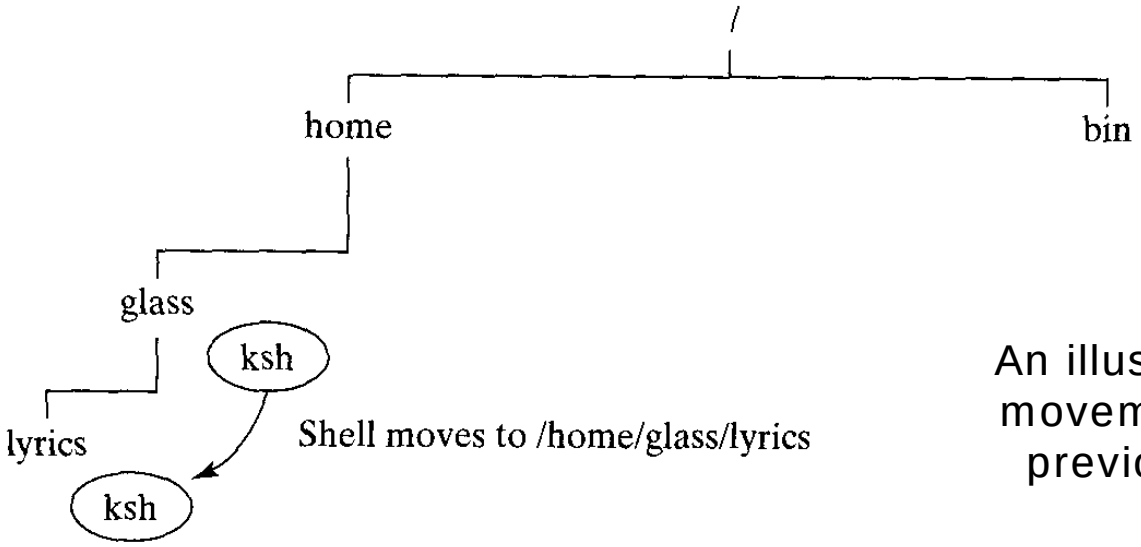
- In general, it’s a good idea to move your shell into a directory if you intend to do a lot of work there. To do this, use the *cd* command.
- *cd* isn’t actually a UNIX utility, but instead is an example of a shell built-in command. Your shell recognizes *cd* as a special keyword and executes it directly.

- *Shell Command:* **cd** [*directoryName*]
- The *cd* (change directory) shell command changes a shell’s current working directory to *directoryName.* If the *directoryName* argument is omitted, the shell is moved to its owner’s home directory.

Moving To A Directory : cd (con't)

- The following example shows how I moved into the "lyrics" directory and confirmed my new location by using **pwd:**

```
cuse93:> pwd                    ... display where I am.
/home/glass
cuse93:> cd lyrics              ...move into the "lyrics" directory.
cuse93:> pwd                    ...display where I am now.
/home/glass/lyrics
cuse93:>
```



An illustration of the shell movement caused by the previous *cd* command.

Moving To A Directory : cd (con't)

- Since "." and ".." refer to your shell's current working directory and parent directory, respectively, you may move up one directory level by typing "cd ..". Here's an example:

```
cuse93:> pwd                    ... display current position.
/home/glass/lyrics
cuse93:> cd ..                  ...  move up one level.
cuse93:> pwd                    ... display new current position.
/home/glass
cuse93:>
```

Copying A File : cp

- After moving into the “lyrics” directory, I decided to work on a second version of my lyrics. I wanted to keep the first version for posterity, so I copied “heart.ver1” into a new file called “heart.ver2” and then edited the new file. To copy the file, I used the **cp** utility, which works as shown in.

- *Utility:* **cp** -i *oldFileName newFileName*
  cp -ir {*fileName*}* *directoryName*

- The first form of **cp** copies *oldFileName* to *newFileName.* If the label *newFileName* already exists, it is replaced. The **-i** option prompts you for confirmation if *newFileName* already exists.
- The second form of cp copies a list of files into *directoryName.* The -r option causes any source files that are directories to be recursively copied, thus copying the entire directory structure.

Copying A File : cp (con't)

- cp actually does two things:
  - It makes a physical copy of the original file's contents.
  - In the directory hierarchy, it creates a new label that points to the copied file.

- The new copy of the original file can therefore be edited, removed, and otherwise manipulated without having any effect on the original file. Here's how I copied the "heart.ver1" file:

```
cuse93:> cp  heart.ver1  heart.ver2          ...copy to "heart.ver2".
cuse93:> ls -l  heart.ver1  heart.ver2       ... confirm.
-rw-r--r--   1  glass  106          Jan 30 23:28    heart.ver1
-rw-r--r--   1  glass  106          Jan 31 00:12   heart.ver2
cuse93:>
```

Copying A File : cp (con't)

- After creating five versions of my song's lyrics, my work was done. I moved back to my directory and created a subdirectory called "lyrics.final" in which to store the final version of the lyrics. I also renamed the original "lyrics" directory to "lyrics.draft" which I felt was a better name. The commands to do all this are as follows:

```
cuse93:> cd                          ...move back to my home directory.
cuse93:> mkdir  lyrics.final    ...make the final lyrics directory.
cuse93:> mv  lyrics  lyrics.draft     ...rename the old lyrics dir.
```

- The final version of my lyrics was stored in a file called "heart.ver5" in the "lyrics.draft" directory, which I then copied into a file called "heart.final" in the "lyrics.final" directory:

```
cuse93:> cp  lyrics.draft/heart.ver5  lyrics.final/heart.final
cuse93:>
```

Deleting A Directory : rmdir

- I decided to remove the “lyrics.draft” directory, to avoid exceeding my modest disk quota. Before I removed it, though, I hived its contents using the **pico** utility. To remove the directory, I used the rmdir utility.
- *Utility:* **rmdir** {*directoryName*}+
  The **rmdir** utility removes all of the directories from the list of directory names. A directory must be empty before it can be removed. To remove a directory and all of *its* contents recursively, use the **rm** utility with the -r option (described shortly).
- I tried to remove the “lyrics.draft” directory while it still contained the draft versions, and I received the following error message:

```
cuse93:> rmdir lyrics.draft
rmdir: lyrics.draft: Directory not empty
cuse93:>
```

- To remove the files from the “lyrics.draft” directory, I made use of the rm utility.

Deleting A File : rm

- The **rm** utility allows you to remove a file's label from the hierarchy. When no more labels reference a file, UNIX removes the file itself. In most cases, every file has only one label, so the act of removing the label causes the file's physical contents to be deallocated.
- However, in some occasions wherein a single file has more than one label. In these cases, a label may be removed without affecting the file that it refers to.

Deleting A File : rm (con't)

- *Utility:* **rm** -fir {*fileName*}*
- The **rm** utility removes a file's label from the directory hierarchy. If the filename doesn't exist, an error message is displayed. The **-i** option prompts the user for confirmation before deleting a filename; press y to confirm your request and n otherwise. If *fileName* is a directory, the -r option causes all of its contents, including subdirectories, to be recursively deleted. The **-f** option inhibits all error messages and prompts.
- To remove every file in the "lyrics.draft" directory, I moved into the "lyrics.draft" directory and used **rm:**

```
cuse93:> cd lyrics.draft                    ...move to "lyrics.draft" directory
cuse93:> rm heart.ver1  heart.ver2  heart.ver3  heart.ver4  heart.ver5
cuse93:> ls                                 ...nothing remains.
cuse93:>
```

Deleting A File : rm (con't)

- Now that all the files were erased, I moved back to my home directory and erased the draft directory:

```
cuse93:> cd                          ...move to my home directory.
cuse93:> rmdir  lyrics.draft         ...this time it works.
cuse93:>
```

- I could have written the following instead:

```
cuse93:> cd  lyrics.draft   ...move into "lyrics.draft" directory.
cuse93:> rm  *              ...erase all files in current directory.
```

- Even better, I could have used the more advanced -r option of rm to delete the "lyrics.draft" directory and all of its contents with just one command:

```
cuse93:> cd                          ...move to my home directory.
cuse93:> rm  -r  lyrics.draft        ...recursively delete directory.
cuse93:>
```

Printing A File: lpr/lpq/lprm

- The commands above are on most UNIX systems and come from the System V family of UNIX. BSD UNIX provided its own print commands, which are still supported in many versions of UNIX with a strong BSD background as well as in Linux. These commands cover the same basic functions of printing, checking the queue, and canceling the job, but have different names and arguments.
- To print my file by using such a system, I employ the **lpr** utility. **lpr** causes a numbered print job to be started for the specified files. You may find the status of a particular job or printer by using the **lpq** utility.

Printing A File: lpr/lpq/lprm (con't)

- *Utility:* **lpr** -m [*-Pprinter*] [*-#copies*] {*fileName*}*
- **lpr** prints the named files to the printer specified by the -P option. If no printer is specified, the printer in the environment variable $PRINTER is used. If no files are specified, standard input is printed instead. By default, one copy of each file is printed, although this may be overridden using the -# option. The -m option causes mail to be sent to you when printing is complete.

Printing A File: lpr/lpq/lprm (con't)

- *Utility:* **lpq** -l [*-Pprinter*] {*job#*}* {*userId*}*
- **lpq** displays the status of jobs on the printer specified by the -P option. If no printer is specified, the printer in the environment variable $PRINTER is used. **lpq** displays information pertaining to the specified jobs or the jobs of the specified users. If no jobs or users are specified, the statuses of all jobs on the specified printer are displayed. The -l option generates extra information.
- If, for some reason, you wish to cancel a print job, you may do so by using the **lprm** utility. You may obtain a list of the printers on your system from your system administrator.

Printing A File: lpr/lpq/lprm (con't)

- *Utility:* **lprm** [*-Pprinter*] [-] {*job#*}* {*userid*}*
- **lprm** cancels all of the specified jobs on the printer specified by the -P option. If no printer is specified, the printer in the environment variable $PRINTER is used. The
  — option cancels all of the print jobs started by you. If you are the superuser, you may cancel all of the jobs owned by a particular individual by specifying his or her user id.

Printing A File: lpr/lpq/lprm (con't)

- As in our previous example, I started by ordering a printout of "heart.final" from the "lwcs" printer. I then decided to order two more copies and obtained a printer status. Finally, I changed my mind and canceled the last print job. The commands are as follows:

```
cuse93:> lpr –Plwcs  heart.final          ...order a printout.
cuse93:> lpq -Plwcs glass              ... look at the printer status.
lwcs is ready and printing
Rank Owner Job Files                        Total Size
active glass 731 heart.final                213 bytes
cuse93:> lpr -#2 -Plwcs heart.final      ...order two more copies.
cuse93:> lpq -Plwcs glass       ...look at the printer status again.
lwcs is ready and printing
Rank Owner Job Files                        Total Size
active glass 731 heart.final                 213 bytes
active glass 732 heart.final                 426 bytes
cuse93:> lprm -Plwcs 732               ....remove the last job.
centaur: dfA732vanguard dequeued
centaur: cfA732vanguard. utdallas.edu dequeued
cuse93:>
```

Printing A File: lpr/lpq/lprm (con't)

- In the next example, I used the keyboard to compose a quick message for the printer and requested that I be notified by e-mail upon completion of the job:

```
cuse93:> lpr -m -Plwcs                    ...print from standard input.
Hi there,
This is a test of the print facility.
- Graham
^D                                        ...end of input.
cuse93:>
```

Counting The Words In A File : wc

- I was quite interested in finding out how many characters, words, and lines were in the “heart.final” file (even though printing it gave me a byte count). To do this, I used the **wc** utility. Here’s an example of **wc**:

```
cuse93:> wc heart.final                    …obtain a word count.
                    9      43     213 heart.final
cuse93:> _
```

Utility: **wc** -lwc { *fileName* } *

The **wc** utility counts the lines, words, or characters in a list of files. If no files are specified, standard input is used instead. The **-l** option requests a line count, the -w option requests a word count, and the **-c** option requests a character count. If no options are specified, then all three counts are displayed. A word is defined by a sequence of characters surrounded by tabs, spaces, or newlines.

Determining Your Terminal's Type: tset

- Several UNIX utilities, including the two standard editors **vi** (or **vim**) and **emacs,** need "know" what kind of terminal you're using so that they can control the screen correctly.
- The type of your terminal is stored by your shell in something called an *environment variable.*
- You may think of environment variables as being global variables for the shell and they can hold strings.

Determining Your Terminal's Type: tset (con't)

- Before **vi** (or **vim**) or **emacs** can work correctly, your shell's TERM environment variable must be set to your terminal type. Common settings for this variable include "vtl00" and "vt52." There are several ways to set TERM:
  - Your shell start-up file can set TERM directly by containing a line of the form 'setenv TERM vtl00' (C shell) or 'TERM=vtl00; export TERM' (for Bourne, Korn, and Bash shells). This method of setting TERM is practical only if you know the type of your terminal in advance and you always log into the same terminal.
  - Your shell start-up file can invoke the **tset** utility, which looks at the communications port that you're connected to and then set TERM accordingly. Consult the online manual for **tset**.
  - You can manually set TERM from a shell.

Editing A File: pico (nano in Linux)

- General Command
  - Write editor contents to a file — [Ctrl] o
  - Save the file and exit pico — [Ctrl] x
  - Spell Check — [Ctrl] t
  - Justify the text — [Ctrl] j
- Moving around in your file
  - Move one character to the right — [Ctrl] f or right arrow key
  - Move one character to the left — [Ctrl] b or left arrow key
  - Move up one line — [Ctrl] p or up arrow key
  - Move down one line — [Ctrl] n or down arrow key