CSE 178

Fall Semester 1995

Unix and Shell Programming Basics

Table of Contents

Introduction and Basic Unix Utilities

This Document

In addition to being a set of class notes, these notes also exist as an interactive hypertext document. This means there are embedded links to other documents and sites, such as John Smith's extensive UNIXhelp for Users. Browsing these links, using a WWW browser, is not only fun, but educational!

Also, don't miss out on using Usenet News and Electronic Mail to help you learn *Unix and Shell Programming Basics*.

Operating System

Definition: A program which acts as an interface between the computer hardware and the user of the computer.

1. Provides an environment in which a user may execute programs.
2. Controls the resources of a computer and allocates them among the users of the computer.

Goals:

1. Make the computer easier to use.
2. Efficient use of the computer's resources.
   - Manage memory.
   - Manage processes and CPU time and usage.
   - Control peripheral devices (e.g. disks, printers, terminals, modems)
   - Provide a file system that manages the long term storage of information (e.g. programs,data,documents).

Unix

The name Unix refers to a family of computer operating systems (e.g. Ultrix, IBM Unix (AIX), AT&T System V UNIX (UNIX=registered trademark of AT&T and USL (Unix Systems Lab)), Berkeley Unix).

Definition: An interactive multiuser operating system.

History: Developed at AT&T Bell Labs in Murray Hill, New Jersey in 1969 by Ken Thompson. Originally written in PDP-7 assembly language. Written in C in 1973.

Advantages:

1. Portability - Since most of Unix is written in the language C, it is quite portable; it runs on micros to mainframes. This gives Unix a strong commercial advantage.
2. Easily modified -The source code is available and written in a high level language (C) so the system is easy to adapt to different requirements.
3. Provides a rich and productive environment.
4. Supports multi-tasking (asynchronous processes).

General Structure:

[[graphics - no textual representation.]]

The kernel controls the resources of the computer and allocates these resources among the users of the computer.

Unix Philosophy: The power of a software system comes as much from the relationships among programs as it does from the programs themselves.

1. On their own many Unix tools perform very simple operations. (Tools philosophy: separation of different functions into different programs.)

2. Simple tools can be easily combined to create powerful, general-purpose tools.

Unix Utilities

passwd

As soon as you log in to the Unix system for the first time change your password using this command. You will be prompted for all needed information.

---

vi

The standard full-screen editor available under Unix.

```
Using Command mode
ZZ         - save file and exit
:q!        - exit without save
:w         - write buffer to disk
h (left)   j (down)   k (up)   l (right)
ndw        - delete n words (default for n = 1)
nx         - delete n characters (default for n = 1)
ndd        - delete n lines (default for n = 1)
-          - go to start of previous line
^          - go to first non-blank character of current line
0          - go to first column of current line
$          - go to end of current line
nG         - go to nth line of file (default for n = 1)
nyy        - yank and save n lines
p          - put down lines saved

Using INSERT MODE
<esc>      - exit INSERT mode
i          - enter INSERT mode ; insert to left of cursor
a          - enter INSERT mode ; insert to right of cursor
A          - enter INSERT mode ; add at end of current line
o          - enter INSERT mode ; add a new line after current line
O          - enter INSERT mode ; add a new line before current line
/<pat>     - search from cursor down for pattern <pat>
?<pat>     - search from cursor up for pattern <pat>
n          - go to next occurrence of last pattern indicated
N          - go to previous occurrence of last pattern indicated
```

[[graphics - no textual representation.]]

OPTIONAL

---

stty

sets I/O options on current output terminal. With no arg, reports terminal speed and settings different than defaults. Some options are :

1. all : print all normally used option settings.
2. everything : print all information known to stty.
3. raw : raw mode input (no input processing (e.g. erase, kill,interrupt).
4. -raw : negate the raw mode.
5. echo : echo back each character typed.
6. -echo : do not echo characters.
7. erase c : set erase character to c (default='#').
8. kill c : set kill character to c (default='@').
9. intr c : set interrupt character to c (default=ctrl ?).
10. start c : set start character to c (default=ctrl q).
11. stop c : set stop character to c (default=ctrl s).

Example: `stty eof '^d' erase '^h' kill '^u' intr '^c'`

These are the typical settings. Note the differences from the defaults given.

---

control characters

1. ctrl-w : deletes the current command line word you are typing (word=any sequence of non-blank characters).
2. ctrl-d : end-of-file character.
3. ctrl-c : interrupts current foreground process (In mail two ctrl-c characters kill the mail message and save it in dead.letter).
4. ctrl-s : suspends output being printed to screen.
5. ctrl-q : resumes printing of suspended output.
6. ctrl-r : moves to next line and redraws command line.
7. ctrl-h : erases character just typed on command line.

---

cat

displays the contents of a text file. This command is actually used to concatenate two or more files. Since the output goes by default to the standard output, cat can be used to display at the terminal screen the contents of a single file (i.e. a single file with nothing concatenated to it).

1. -e : marks the ends of lines of input files with dollar signs.
2. -s : suppresses the warning message when cat can't find an input file.
3. -t : marks each tab in input files with ^I.
4. -v : cat displays nonprinting characters (other than tabs and newlines) found in input files.
5. -n : prepend line number to output line

Syntax: `cat <file names>`

Examples:

```
$ cat prog.c
$ cat file1 file2 file3
$ cat -v -t file1
```

---

ls

with no arguments prints a list of the names of the files in the current working directory. With directory arguments a list of files in each directory is printed. With a file argument that filename is listed. Some options are:

1. -a : displays all entries including those that begin with a period.
2. -c : sorts entries by time of last inode modification.
3. -d : displays directories only.
4. -F : marks entries in list (directory/ , socket= , symbolic link@ , executable_file*)
5. -l : lists mode, number of links, owner, size(bytes), time of last modification. Special files have size field replaced with major and minor device numbers. (mode first character: d=directory, b=block-type special file, c=character type special file,l=symbolic link,s=socket,-=plain file) (mode permissions:user-group-other)
6. -s : displays size (in blocks) of each file (given as first entry)
7. -r : sorts entries in reverse alphabetic or time order

8. -i : prints the inode number of each file
9. -u : sorts by time of last access (read or executed)
10. -t : sort (Example: Sort by last inode modification time = ls -ctl)
11. -1 : print in a single column (*Examples:* `ls -1` or `ls -ct1`)

Syntax: `ls <file names>`

Examples:

```
        $ ls -l draw
        -rwx------   1 jpl   24576 Oct 18 12:45 draw

                            permissions = -rwx------
                            number of links = 1
        meaning ->          owner userid = jpl
                            size in bytes = 24576
                            date of last modification  = Oct 18 12:45
                            filename = draw
```

Three times are recorded and available when using ls.

1. Inode modification time (c)
2. File contents modification time. (default)
3. Access time (u)

---

rm

removes the named files.

1. -i : ask before removing each file listed (interactive mode).
2. -r : remove listed directories and all their subdirectories
3. -f : silently remove files for which one does not have write permissions.

Syntax: `rm <file names>`

Examples:

```
      $ rm file1 file2
      $ rm prog.c
      $ rm -ri workdir
```

---

more

displays the named files one screenful at a time.

1. +n : starts displaying first file at line number n
2. -n : specifies number of lines to display on screen. (The default is obtained from variable TERM and info in /etc/termcap.

Syntax: `more <file names>`

Examples:

```
      $ more file1 file2
      $ more +25 prog.c
```

---

rmdir

removes directories (if empty)

Syntax: `rmdir <directory names>`

Example:

```
$ rmdir dir1 dir2 dir3
```

---

mkdir

makes a directory (needs write permission in parent directory)

Syntax: `mkdir <directory name>`

Example:

```
$ mkdir artdir
```

---

man

displays pages from the on-line system documentation/manuals.

Syntax: `man <utility name>`

Examples:

```
$ man more
$ man rm
$ man man
```

---

echo

prints all of its arguments to the terminal screen.

1. -n : Prevents a newline from being added to the output. (C Shell)

   Syntax: `echo <argument list>`

   Example:

   ```
   $ echo Hello world.
   Hello world.
   $ echo -n Hello world.
   Hello world.$
   ```

2. \c : Prevents a newline from being added to the output. (Bourne)

---

cp

makes a copy of a file. Any file can be copied (e.g. text or executable files). Be careful: cp may overwrite an existing file without warning.

1. -p : give the copied file the same modification times and access permissions as the source file.
2. -i : if a file will be overwritten, ask before destroying the old contents.

3. -r : if a source file is a directory, copy it and its contents and all of its subdirectories (if any exist) into the directory named as a last argument.

Syntax:

```
cp <source filename> <new filename>
cp <source file list> <directory>
```

Example:

```
$ ls
file1
$ cp file1 file2
$ ls
file1 file2

$ cp file1 file2 /csarea/csefac/jpl/recdir

$ cp -r dir1 dir2
```

---

mv

renames a file.

1. -f : move the named files regardless of their access permissions.
2. -i : ask before overwriting an existing file.

Syntax:

```
mv <old file> <new file>
mv <file list> <directory>
```

Example:

```
$ ls
file1
$ mv file1 file2
$ ls
file2

$ mv file1 file2 /csarea/csefac/jpl/recdir

$ mv -i file1 file2
```

---

date

displays the current date and time. (OPTIONAL)

Syntax:

```
  date
  date +<format>
          <format> may contain the following
          a       abbreviated day (Sun to Sat)
          d       day of the month
          D       date in month/day/year format
```

```
h       abbreviated month (Jan to Dec)
H       hour (00 to 23)
j       day of the year (001 to 366)
m       month of the year (01 to 12)
M       minutes (00-59)
r       time in A.M./P.M. notation
y       last two digits of year (00 to 99)
S       seconds (00 to 59)
T       time in hours:minutes:seconds format
w       day of the week (0 to 7, Sun=0)
n       newline
t       tab
```

Example:

```
$ date
Wed Aug  7 10:44:34 MST 1991

$ date '+%a %t %h %t %y'
Tue       Dec      93

$ date '+%d%t%D%t%H'
07        12/07/93                09

$ date '+%r%n%T'
09:35:16 AM
09:35:16
```

---

grep

searches the files named for lines that match the pattern given. The lines that contain the given pattern are printed. (grep=Global Regular Expression Print) Some of the options available are:

1. -v : all lines except those matched are printed.
2. -i : ignore uppercase/lowercase distinctions in matching.
3. -c : only a count of matching lines is printed.
4. -l : only the names of files with matching lines is printed. (Each filename is printed once and separated with a newline.)
5. -n : each line is preceded by its relative line number in the file.

Syntax: `grep [-vicln] <pattern> <file names>`

Example:

```
$ cat file1
azz
bzz
czz
dzz
$ grep azz file1
azz
$ grep -v azz file1
bzz
czz
dzz
```

---

head

prints a portion of the beginning of a file.

Syntax: `head [-number]`

Examples:

```
$ head file1      # prints default of first 10 lines
$ head -2 file1   # prints first two lines of file1
```

---

tail

prints a portion of the end of a file.

1. + : display blocks,characters, or lines from start
2. -b : count by blocks
3. -c : count by characters
4. -l : count by lines (default)
5. -f : copy last line and enter infinite loop; wait and print added lines if file grows
6. -r : print lines in reverse order

Syntax: `tail [-|+ [number] options] [file]`

```
$ tail -8c file1          # prints last 8 characters
                  # newline is invisible
$ tail +8 file1   # prints file starting at line 8

$ make accts > accts.out &
$ tail -f accts.out
```

The above example allows you to save the output of make while you watch its output on the screen. If need be one can break out of the tail and do something else in the foreground.

---

sort

sorts lines of all named input files and then writes the sorted result to the standard output by default. (standard input = default input when no input files are named) (default sort key = entire line) (if filename = "-" it is stdin)

1. -r : order items in decreasing order
2. -u : omit all but one duplicate lines
3. -c : only produce output if files named are not sorted

Syntax: `sort [-o outfile] [-rcu] <file list>`

There are many other options. For example, a file can be sorted on arbitrary keys. NOTE: Defaulting to the standard input is seen in many Unix tools. Most/many Unix tools also send their output to the standard output by default.

Examples:

```
$ sort -o info.out myinfo
$ sort -u myinfo  # sort and discard equal lines
```

---

uniq

prints a file skipping adjacent duplicate lines. The default input is stdin and the default output is stdout. Initial fields and characters can be skipped in the comparisons but these features are not covered here.

1. -c precede each line with #occurrences
2. -d print only lines that are repeated
3. -u print only lines that are not repeated

Syntax: `uniq [options] [input file] [output file]`

Examples:

```
$ uniq file1 file1  #Oops, you lost contents of file1.
$ cat file2
a
a
b
c
c
c
$ uniq file2
a
b
c
$ uniq -c file2
2 a
1 b
3 c
$ uniq -u file2
b
```

---

diff

tells what actions need to be taken in order to convert the first file named into the second file. If one of the files is "-" then it is the stdin. The "<" names lines from the first file and the ">" lines from the second file. Numbers before a letter command refer to first file lines while numbers after refer to second file lines. (a=add, d=delete, c=change)

1. -b trailing spaces and tabs are ignored
2. -e produces a script of commands for line editor "ed" that will produce file2 from file1

Syntax: `diff [-eb] <file1> <file2>`

```
(NOTE: diff used without -e does not produce editor script format.)

status codes:
   0=identical files
   1=different files
   2=error condition

format explained:

  n1 a n3 - add line n3 (file2) after line n1 (file1)
  n1 a n3,n4 - add lines n3 to n4 from file2 after
               line n1 in file1
  n1,n2 d n3 - delete lines n1 to n2 from file1. (To
               make file2->file1 add n1 to n2 of file1
               after line n3 in file2.)
  n1,n2 c n3,n4 - change lines n1 to n2 of file1 to
                  lines n3 to n4 of file2
```

Example:

```
$ cat f1
```

```
a
b
c
$ cat f2
a
c
c
$ diff f1 f2
2c2 # change line 2 of file1 into line 2 of file2
<b
---
>c
$ diff -e f1 f2
2c
c
.
```

NOTE: Keeping an original file and a chain of version to version scripts made by diff could be the basis of a version control system.

---

cmp

compare two files. Makes no comment if files are identical. If different only gives byte and line number at which the first difference occurred.

---

who

prints a list of the users currently logged in. Information given is user name, terminal that user is using, date and time that user logged in.

Example:

```
$ who
jpl   ttyp4   Aug 12 09::40 (fremont.cse.nau.)
```

who am i : gives "who" info for yourself.

---

wc

counts words, lines, and characters.

1. -c : display the number of characters in the input file(s)
2. -l : display the number of lines in the input file(s)
3. -w : display the number of words in the input file(s)

Examples:

```
$ wc -l t1 t2
4 t1
3 t2
7 total

$ wc -lc t1 t2
4       20 t1
3       15 t2
7       35 total
```

Using the Unix Mail Facility.

The mail utility (mail) allows you to send and receive electronic mail (email). A mail message has two parts, a header and a body. A header is composed of four parts as shown below.

```
        [[graphics - no textual representation.]]
```

Command Line Options of the Mail Facility

```
mail                    Read any mail residing in your system mailbox.

mail -f <filename>      Read any mail residing in file <filename> .

mail -f                 Read any mail residing in the file ~/mbox.

mail -v <address list>  Create a mail message and send it to the given
                        addresses and watch each one being delivered.

mail -s <text> <address list>   Send mail to the given addresses using
                                <text> for the subject line.

mail <pathname> Create a mail message in a file given by <pathname>
                The <pathname> must have a slash (/) within it.

mail '|<cmd>'   Create a mail message and send it to the process
                created when command <cmd> is executed. Since <cmd>
                may not contain spaces, an alias in .mailrc may have to
                be used.
```

Other Useful Commands Related to the Mail Facility

```
from            Print a summary of all the mail in your system mailbox.

biff y          Give notice of new mail and print a summary of its message.
```

A Summary of Commands Available When Reading Mail

```
h(eaders)               Display the headers of pending messages.

+ , n(ext)              Display the next message.

-                       Display the previous message.

e(dit)   <msg#> Edit message specified. The current message is the default.

q(uit)                  Exit the mail program retaining all changes made.

x                       Exit the mail program. Do not retain any changes.

w(rite) <file>  Write the body of the current message to <file>.

s(ave) <file>   Write header and body of the current message to <file>.

<number>                Display the message whose number is specified.

d(elete) <msg#> Delete the message specified. Current message = default.

u(delete) <msg#> Restore the message specified. Current message=default.

dp                      Delete the current message and display the next message.

m <userid>              Create a new mail message and send it to <userid> .
```

As indicated above, commands can be applied to a default message, which is usually the current message, and commands can be applied to a specified message. It is also possible to specify a list of messages using the notation given below.

Individual Message and Message List Notation

.	The current message.
<number>	The message number specified.
n-m	The list of messages from message #n to message #m
^	The first message pending.
$	The last message pending.
*	All messages pending.
<userid>	All messages sent by <userid> .
/<pattern>	All messages with <pattern> in the subject line.
:n	All new messages pending.
:r	All messages already read.
:u	All messages not yet read.

Examples:

```
p 2-4          Print messages 2, 3 and 4.
w * temp       Write messages 2 through 4 to file temp.
e ^            Edit the first message pending.
```

Whenever the mail facility is executed it in turn executes the .mailrc file if it exists. Special variables can be set and aliases can be defined in the .mailrc file. An alias is a name given to a command or list of commands. These can be useful when using the command line option that allows mail messages to be sent to commands. Special variables can be used to adjust the behavior of the mail facility. Given below are a few of the variables that can be used to set up the mail facility

Mail Facility Variables

ask	Prompt for the subject line.
askcc	Prompt for the copy line.
crt=<number>	Set the number of lines to be viewed per screen for long messages.

The Information in a message header.

```
The <userid> of the sender      of the message.
The date and time that the message was received .
The size of the message.
As much of the subject line as will fit into the header.
```

Examples:

```
jpl@pine.cse.nau.edu    Jan  6  11:52  (121 li) ACM Lecture Visit
```

Unix Files and the Unix File System

The Unix File Structure

The Unix file system is so important that understanding it is a prerequisite to using Unix well.

A file system in Unix is an organization of files on a disk. It is basically the collection of the files on a disk together with a catalog of the files. This catalog occurs at a fixed location on the disk, is called the ilist of the file system, and is composed of a number of entries called inodes.

A regular file is simply a file that contains a program or data.

The file system is hierarchical. It organizes files into one large tree structure. This allows users to group related files together in a simple way.

```
        [[graphics - no textual representation.]]
```

Note the following:

1. All nodes are files (A directory is just a special file.)
2. There is exactly one root node (/) .
3. Every node has 0 or more dependents.
4. Every node has one parent except the root (/) node.

Because the file system is hierarchical, files can have the same name with no conflict if they are located in different directories.

Compatibility of file, device, and interprocess I/O (output serves as input to another program) is an example of how an abstract concept can identify common aspects of diverse software components.

```
        [[graphics - no textual representation.]]
```

Unix does not distinguish between these three possible data sources and three possible destinations. A user can specify, using the same command syntax for naming files, devices, and processes, at execution time which of each to use for I/O. In general, a file is defined as simply a sequence of bytes.

1. No structure is imposed on a file by the system.
2. No meaning is attached to the contents of a file. A program using a file determines the meaning of that file.

Every Unix file has the following information associated with it :

1. A position in the file hierarchy.
2. A file name.
3. A file size.
4. The blocks that contain a file's data.
5. A set of permissions.
6. Last modification, access, and inode change times. For example, the command find affects

access time while the command chmod affects inode change time.
7. Other administrative and system information.

An inode defines a file. Remember that an inode is an entry in the ilist of a file system which is itself the catalog of files that exist on that disk.

1. An inode number is the system's internal name for a file. (i-number)
2. Inode number of file = number of inode holding file's information.
   Example: `ls -i file1 ---> 38541 file1`
3. The first part of each directory entry (the inode number of a file) is the only connection between the name of a file and its contents.
4. A filename is called a "link" because it links a name in the directory hierarchy to an inode and in this way connects that name to all the information the file represents.
5. An inode contains the following information:
   - Whether the file is a directory, Unix special file, or Unix ordinary file.
   - What physical device the file resides on.
   - Who owns the file.
   - Who can access the file.
   - If the file resides on a disk, what blocks on the disk constitute the file and in what order.

Directories are files that have a special format.

1. Each entry defines a file. (It begins with the file's inode number.)
2. The first entry of every directory is the file dot ("."). This represents the entry for the directory itself. This entry allows a directory to be accessed without knowing its complete (absolute) pathname. (e.g. in a shell script you can give a relative pathname: ls -d .)
3. The second entry of every directory is file dot-dot (".."). This is the entry for the parent of the directory.
4. You may examine directory and regular file formats using the "od" command. (octal dump)
   - -c option interprets bytes as characters
   - -cb option also gives octal numbers

```
$ cat file1
Hello
T
$od -cb file1
0000000     H   e   l   l   o   \n   T   \n
         110 145 154 154 157 012  124 012
```

OPTIONAL

Brief sketch of file system layout :

- Block 0 : This first block is the boot block. It contains a short bootstrap program. This may read in a larger bootstrap program or it may read in the Unix system kernel itself. This is very system dependent. For a system not using bootstrapping this block is usually not used.
- Block 1 : This second block is the file system header (the super block) This block contains the file system size, the number of inodes used, and information about the free blocks list.

1. When a file system is mounted an entry is made in the Unix kernel's mount table and that file system's super block is read into a large internal buffer.
2. The super blocks of all the mounted file systems are accessible to the kernel which needs

the information to access files and inodes in a file system.

- Block 2 : This third block begins the ilist (file system inode list). Inodes are fixed in size and numbered consecutively from 0. Since the beginning of the ilist is fixed any inode can be located given its number.

Try df -i. (df = disk free information; -i is the inode option.)

OPTIONAL

Layout of a Unix file system on a device. [[graphics - no textual representation.]]

Some directories to be familiar with :

1. /usr/bin : (binaries) standard Unix utility programs are found here (e.g. who,write,rm) Prior to System V Rlease 4, /bin contained some of these utilities.
2. /dev : (devices) special files called device files are found here (all files representing peripheral devices).
3. /etc : (et cetera) contains administrative files and tables (e.g. passwd which contains all the login information about each user.
4. /lib : (libraries) libraries of subroutines for programmers using high level languages like C and other special purpose programming libraries.
5. /tmp : used for temporary files made during program execution
6. /src : sometimes holds full/partial set of Unix C language source code listings. (/usr/src may also hold such files)
7. /usr : will find many subdirectories here including bin , lib, and src. Typically /usr/bin holds off-the-shelf software packages (executable modules) that are widely used.
8. /usr/include : contains header files for C programs. Each of these files contains a ".h" suffix.
9. /usr/man : contains on-line manual pages for Unix commands, system calls, and library functions.

Some useful terminology :

1. home directory - directory entered when you log in.
2. working directory - the directory you are currently working in (also called the current directory or the current working directory).
3. / - the root directory.
4. pathname - the name of the path from one directory to another file or directory. If the pathname begins with the root directory it is called the absolute pathname. (Example: rm /csarea/csefac/jpl/project1/file1) Otherwise it is called a relative pathname. (Example:: Assume you are in /csarea/csefac/jpl/project2 : rm ../project1/file1)
5. super-user - Unix system administrator who may read and modify any file in the system no matter what the file's permissions are.
6. login id - the name you log in with.
7. uid - the internal number that the Unix system actually recognizes a user with.

Some useful commands :

1. pwd - print pathname of the working directory.
2. cd - change to the home directory
3. cd <directory> - change to the directory named.

File permissions : Every file (and so every directory also) has a set of permissions associated with it. These permissions allow various types of protection to be given to a file/directory. Two forms of permissions specifications can be used:

1. symbolic rwxrwxrwx 3 groups: user,group,others r = read permission granted w = write permission granted x = execute permission granted
2. absolute 3 numbers can represent a file's permissions where r=4, w=2, x=1, and 0 = none are set.

The setting of the permission specification will determine your privacy/protection. Recall that:

```
        $ ls -l <file names>   # reports file permissions
        $ ls -ld <directory>   # reports directory perms
```

Examples:

```
         rw-rw-r--      user and group may read and write
                        to file, others may only read file

         rwxr--r--      user may read, write, and execute
                        file; group and others may only
                        read the file
      Misc info:

        4000 - set user ID on execution.
        2000 - set group id on execution.
        1000 - sticky bit (keep in the swap space)
        0400 - read by user granted
        0200 - write by user granted
        0100 - execute by user granted
        etc.
```

Directory permissions :

```
        - A directory can never be executed.

        - A directory can be read from and written to by all three types of user
                (u=owner , g=group , o=all others)

        - Read permission allows you to read the entires in a directory.

        - Write permission allows you to make changes to a directory.
                (e.g. create, move, copy, remove, etc. files)

        - Execute permission (x) for a directory has the following semantics:
                - You can use (e.g. execute) the entries in a directory.
                  You can also cd to directory and check file sizes.

        Example:
                wx = Allows one to access and modify but not read a directory.
                (e.g. Can remove a file if you know it is in that directory
                        but you cannot read (ls) the directory.)
```

Permissions are set by using the chmod command.

Syntax:

```
      chmod [augo]+ optr sym_perm {, optr sym_perm} <files>
      chmod absolute_perm <files>

      where: a=all, u=user, g=group. o=other
```

```
              optr = ( + | - | = )
              sym_perm = (r | w | x)
```

Examples:

```
        $ chmod 666 file1   # sets permissions to rw-rw-rw-
        $ chmod a+x file1   # all 3 groups get execute perms
        $ chmod a-w file1   # no write permission for all
        $ chmod o-w file1   # deny others write permission
        $ chmod g+w,o-r,u+x file1 # write to group, deny
                                  # read to others, exec
                                  # for user

        $ chmod go=u file1  # sets groups,others = user
        $ chmod go-rw file1 # takes away read and write
                            # perms from groups and others
        $ chmod a=rw file1  # sets all groups perms to rw
        $ chmod go-rwx file1 #no perms for group and others
```

Basic information about Unix I/O and I/O devices.

1. Each process that executes under Unix has: A. a source from which it expects to receive input : a standard input. B. two destinations to which it can send output (if any) : standard output (normal output) and standard error.
2. A process may have other files and devices added to this minimal set of three basic I/O devices, but when a process first begins execution these three devices come by default: A. keyboard (the default standard input device) B. terminal screen (the default standard output device) C. terminal screen (the default standard error device)
3. I/O devices include terminal screens, keyboards, printers, tape drives, disk drives, etc.
4. A special file is a file that is associated with an I/O device rather than a region on some storage medium. By convention, all special files are located in the directory /dev. There exists a special file for each I/O device supported by your system.
5. There are two types of special files (try ls -l /dev) : A. block special file: performs I/O in blocks (e.g. 512 or 1024) (perms start with b). B. character special file: performs I/O on a character by character basis (perms start with a c).
6. Each special file has a major and a minor number. A. major device number: encodes the type of device. B. minor device number: gives the instance of the device of a certain type.

When you log in, you get a terminal line and , therefore, a file in /dev through which the characters you type and receive are sent.

Try command "tty". It prints the special filename of the terminal attached to your standard output.

Bourne and C Shell Basics, Redirection, Pipes, Processes

The Unix Shell

The shell is a command language. A command language is a program that interprets your requests to run programs. The shell is first invoked as a user's initial process by the login process. We will learn many basic features of two different shells commonly used on Unix systems: the C shell and the Bourne Shell.

```
C Shell - developed at UCB

Bourne Shell - written by S.R. Bourne

(Korn Shell - enhanced Bourne Shell)
```

The shell is a C program which:

1. Prompts the user for a command line.
2. Reads a command line from a file (/dev/tty or Unix system file).
3. Performs interpretation of a command line.
4. Sends processed command line to the Unix operating system for execution. (The resulting line is executed using system primitives.)

The shell command line structure :

1. The simplest command is a single word which usually names a file for execution.
2. Either the newline or the semicolon terminate a command.

   Examples: `1. date 2. ls; 3. ls;date`

3. In general, a simple shell command is a sequence of non-blank arguments separated by blanks or tabs. The first argument (numbered 0) usually specifies the name of the command to be executed. Example: lpr f1 f2 f3
4. If the first argument of a command line is a compiled executable file then the shell spawns a process that immediately executes the program.
5. If the first argument of a command line is executable but is not a compiled file (i.e. it is an ascii file), it is assumed to be a shell procedure/script which is a file of ordinary text of shell command lines. In this case the shell spawns another shell (sub-shell) to read the file and execute the commands in it.

The shell looks for commands in a specific way. Commands that begin with / , ./ , ../ are simply looked for in the directory indicated.

Examples: `1. /bin/date 2. ../myprog 3. ../mydir/myprog`

Other commands are searched for in each directory listed in the shell variable PATH. The value of PATH is a list of directory names which are searched in order (left-to-right).

We will begin by introducing some basic attributes of the C Shell because this shell has many facilities that will be of help to you in your daily work with Unix. (Your login shell is specified in /etc/passwd as the C Shell.) After this introduction to the C shell, however, we will concentrate on the Bourne Shell since the Bourne Shell is generally the shell of choice when writing shell programming.

The C Shell allows one to create variables (shell variables) that can be used on the current command line, in shell programs, and to control properties of the shell. Some C Shell variables are used like switches which cause certain properties to be exhibited by the shell if they are defined. In other words, defining one of these variables is like turning on the property represented by that variable. Some of these variables are given below.

On/Off switch-type variables. (Turn on/off with set/unset commands.)

1. ignoreeof If defined forces one to use "logout" command to logout.
2. noclobber If defined won't allow redirected output to overwrite an existing file.
3. noglob If defined treats * , ? , [,] , ~ as literal characters.
4. filec If defined enables filename completion using the escape key.

Another type of C Shell variable is used to contain or hold local information that the user can utilize when desired. Some of these variables are given below.

Information containing variables.

1. history the size of the command/event history list.
2. cwd the pathname of the current working directory.
3. home the pathname of your login directory.
4. prompt the prompt string. To display an event number within a prompt use an exclamation point. (e.g. set prompt = 'jpl [\!]%') Note that you must escape the exclamation point.
5. savehist the number of history lines to save at logout.
6. status the status of the last command executed.
7. argv the command line arguments of the current shell (an array)
8. #argv the number of arguments to the shell (not counting arg[0])
9. shell the pathname of the shell
10. path the list of directories to be searched by the shell to find commands
11. $$ the pid (process id) of the current shell
12. user

Yet another type of C Shell variable is used to contain or hold information that can be exported to other shell processes. These variables are called environment variables and by convention utilize capital letters. Some of these variables are given below.

1. EDITOR the pathname of the default text editor
2. LOGNAME the userid of the person who is currently logged in
3. HOME the pathname of login directory (used by cd and to expand ~)
4. TERM the type of terminal being used
5. PATH the list of directories to be searched by the shell to find commands
6. SHELL the pathname of the shell
7. USER

Three shell variables (term, path, and user) have corresponding environment variables (TERM, PATH, USER). If one of these shell variables is modified, its corresponding environment variable is modified automatically. Changing one of the three environment variables mentioned above, however, does not automatically affect its corresponding shell variable.

The set , setenv, and @ commands are used to assign values to C Shell variables. The set command assigns values to shell (local) variables. The setenv command assigns values to environment variables, and the @ command is used in assignment to numeric variables.

In the example below, a local shell variable word is assigned the string value hello. Note that spaces may surround the equal sign. (In the Bourne Shell spaces may not surround the equal sign in an assignment.) Immediately after the assignment the value of variable word is printed with the echo command. The dollar sign ($) is the dereferencing operator in both the C and Bourne Shells.

```
(1) % set word = hello
(2) % echo $word
```

```
        hello
```

The use of command set with no arguments will result in a list of all the shell variables and their values. As mentioned earlier, the set command can also be used to "turn on" switch variables.

```
        (3) % set ignoreeof
```

Array values can also be assigned to C Shell variables as shown below

```
        (4) % set cars = (ford toyota chevy)
        (5) % echo $cars
        ford toyota chevy
        (6) % echo $cars[2]
        toyota
        (7) % echo $cars[2-3]
        toyota chevy
```

Most of the arithmetic operators available in the C language are also available in the C Shell when one uses the @ command as shown below.

```
        (8) % @ count = 0
        (9) % echo $count
        0
        (10) % @ count++
        (11) % echo $count
        1
        (12) % @ count = 5 + 1
        (13) % echo $count
        6
        (14) % @ count = (3 * 4)
        (15) % echo $count
        12
        (16) %  @ count += 3
        (17) % echo $count
        15
```

Page 315 in your textbook gives a complete listing of the arithmetic operators supported by the C Shell.

The C Shell also supports the use of curley braces to clearly denote the names of variables during dereferencing.

```
        (18) % set x = John
        (19) % echo ${x}son
        Johnson
```

The History Facility and History Substitution

The C Shell saves several of the last commands (called events) that it was issued in what is called the history list. The size of this list is determined by the value of the shell variable history. You can print out the value of this variable or you can modify its value to alter the size of your history list.

There is also a C Shell command called history. When issued with no arguments this command simply displays the contents of the history list. When a numeric argument is provided, the history command displays the number of most recent events in the history list indicated by that argument. For example, the following shell command prints the last 5 events that occurred which includes the current history command itself.

```
(20) % history 5
16 @ count += 3
17 echo $count
18 set x = John
19 echo ${x}son
20 history 5
```

Use of the -r option causes the most recent events to be printed first.

```
(21) % history -r 5
21 history -r 5
20 history 5
19 echo ${x}son
18 set x = John
17 echo $count
```

Note that the value of variable savehist is the number of most recent commands that are to be saved across login sessions.

There are three ways in which to reference a past event. These are illustrated below.

1. Give the absolute number of an event (i.e. the number an event is associated with in the history list). For example, to execute the event numbered 19 in the history list type the following command.

   ```
   (22) % !19
   echo ${x}son
   Johnson
   ```

2. Give the number relative to the current event. For example, to execute the second preceding event, event 21, type the command

   ```
   (23) % !-2
   history -r 5
   <output not shown>
   ```

3. Give text that forms the prefix of a previous event. For example, to execute the most recent echo command type the following command.

   ```
   (24) % !e
   echo ${x}son
   Johnson
   ```

(Note: If a command that begins with an "e" other than the echo command had been executed more recently than the last echo command it would be executed again.)

(!! is a special command that causes the last event to be executed again just as it was previously.) To execute a previous command that contained text embedded anywhere within it use question marks. In the following command the last event that contained the text son is executed.

```
(25) !?son?
echo ${x}son
Johnson
```

The above techniques for referring to previous events can be used to augment a command. For example, if one wanted to add the argument jumps to the echo command representing event 25 above, the following command would do the trick.

```
(26) % !25 jumps
echo ${x}son jumps
Johnson jumps
```

Replacement or substitution of text in the last command executed (the last event) is easy to do in the C Shell. Consider the command given below which replaces the text jumps with the text hops in the last command executed (event 26).

```
(27) % ^jumps^hops
echo ${x}son hops
Johnson hops
```

Replacement or substitution of text in any event recorded in the history list can also be done in the C Shell. Consider the command given below which replaces the text son found in event 19 with the text ston.

```
(28) % !19:s/son/ston
echo ${x}ston
Johnston
```

Because the exclamation mark (!) is a special character in the C Shell, you may have been wondering how to simply print the symbol ! on the terminal screen. The following command does just that by "escaping" the special meaning of the ! with a backslash (\).

```
(29) % echo \!
!
```

A useful shortcut is to use the two symbols !* to represent all the arguments that were used in the previous event. For example, assume that you had just executed the following command.

```
(30) % echo a b c
a b c
```

Now to access all of the arguments used in the last event (event 30 above) you could simply use the symbols !* as shown below.

```
(31) % echo !* !*
echo a b c a b c
a b c a b c
```

The following example shows how options are treated when using the symbols !* for argument access.

```
(32) % ls -l file1 file2
-rw-------      1 jpl          faculty 64      Dec 18  14:21 file1
-rw-------      1 jpl          faculty 3443    Oct 29   17:42 file2
(33) % echo !*
echo  -l file1 file2
-l file1 file2
```

The following examples are meant to demonstrate other features of command line manipulation and event access supported by the C Shell.

```
(34) % echo one two three four
one two three four
```

Select the second argument from event 34

```
(35) % echo !34:2
echo two
two
```

Select the first argument from event 34 in two different ways.

```
(36) % echo !34:1
echo one
one
(37) % echo !34:^
echo one
one
```

Select the last argument from event 34 in two different ways.

```
(38) % echo !34:4
echo four
four
(39) % echo !34:$
echo four
four
```

Select the command from event 34.

```
(40) % echo !34:0
echo echo
echo
```

Select the second through the fourth argument from event 34.

```
(41) % echo !34:2-4
echo two three four
two three four
```

Some letters can also be used to represent operations in expressions of the type shown above. The following examples illustrate this.

```
(42) % echo /home/carmel/xxx/file1  /home/carmel/yyy/file2
 /home/carmel/xxx/file1  /home/carmel/yyy/file2
```

Remove the last element of the pathname given by the first argument of the previous event.

```
(43) % !!:h
echo /home/carmel/xxx/file1  /home/carmel/yyy/file2
/home/carmel/xxx  /home/carmel/yyy/file2
```

Print the pathname given as the second argument of event 42 after removing the last element of that pathname.

```
(44) % echo !42:2:h
echo  /home/carmel/yyy
/home/carmel/yyy
```

You might want to experiment with the letter t which causes the removal of all elements from a pathname except the last.

Command Aliasing

alias = a name given to a command or list or commands

To create an alias use the alias command.

Syntax: `alias [ <name> [ <command(s)> ] ]`

In the example below, the command ls -ls is given the alias ll.

```
        (45) % alias ll 'ls -ls'
```

It is now possible to invoke the command ls -ls using only the alias ll . Arguments are simply added to the end of the alias as shown below.

```
        (46) % ll file1 file2
```

Consider the following alias that returns the number of persons currently logged onto the system.

```
        (47) % alias c  'who | wc -l'
        (48) % c
             1
```

When the alias command is used with a name only, the value of that name is returned.

```
        (49) % alias c
        who | wc -l
```

The alias command used with no arguments returns the values of all aliases. The unalias command deletes the alias named as its argument.

```
        (50) % unalias c
```

The symbols !* can be used to refer to all arguments that are passed to an alias (i.e. to the current command). In the following definition of alias s, the arguments used with alias s will be passed as arguments to the cat command within the alias definition.

```
        (51) % alias s 'cat \!* | sort'
```

In the following example, file1 and file2 will both be passed as arguments to the cat command. In other words, files file1 and file2 will both be concatenated and then this concatenated result will be read by the sort command.

```
        (52) % s file1 file2
```

In an alias definition, if the ! symbol is used alone (i.e. not followed by an *) then it refers to the current event. This allows the same mechanism for command line argument substitution explained previously for the history mechanism to be used when creating an alias. Consider the examples given below.

```
        (53) % alias lastarg  echo \!:$
        (54) % lastarg one two three four
        four

        (55) % alias second  echo \!:2
        (56) % lastarg one two three four
        two
```

Files that are used for initialization and termination.

The files .cshrc , .login , and .logout are run automatically by the C Shell. These files generally contain commands that are used to set up the working environment of a shell. Aliases can be defined, environment variables can be given values, and other tasks can be performed that are needed to set up the environment of a shell.

Every time a new shell is created, the commands in the .cshrc file are executed within that new shell automatically. New shells are created when logging onto a system and when running shell scripts. (Shell scripts are programs composed of shell commands.) The commands in the .login file, however, are only executed once when logging onto a system. These commands are executed just after the commands in the .cshrc file. The commands in the .logout file also are executed only once, this time when logging out from a system.

We will now begin to focus more on the Bourne Shell but will also discuss facilities which are found in both the C and the Bourne Shell. When a facility is limited to a certain shell that fact will be noted.

The Bourne Shell also allows one to create variables that can be used on the current command line directly or in shell scripts. Some of the properties of the shell are controlled by shell variables. As with other types of variables, shell variables can be read and modified.

1. Some standard Bourne Shell variables:

```
A.    HOME - user login directory.

B.    MAIL - names the standard file where your mail is kept.

C.    PATH - where the shell looks for commands.

D.    PS1 - primary prompt string ("$")

E.    PS2 - secondary prompt string (">")
```

2. The Bourne Shell command set prints a list of shell and environment variables and their values.
3. Bourne Shell variable names must begin with a letter or underscore and may contain letters, digits, and underscores.
4. Bourne Shell variable values that contain spaces or tabs must be quoted. Example : WORDS="this is a sentence"
5. Shell variables are local to the current shell unless you explicitly specify otherwise. Shell files (programs) normally run in a subshell and cannot change the value of a variable in the parent shell.

NOTE: To dereference a shell or environment variable precede its name with a dollar sign. You may also delimit the name with curley braces to avoid any ambiquity.

Information about Bourne Shell variables.

Some special Bourne Shell variables that are automatically set by the shell are:

1. $? - value returned by the last executed command. ($status in the C-shell.)
2. $$ - process number of the current shell.

3. $! - process number of last background process shell invoked.
4. $# - number of arguments to the shell. (#arg in C-shell)
5. $* - all arguments to the shell. (argv in C-shell)

Bourne Shell variables are easy to create, assign to, and reference.

```
$ color=blue
$ echo $color
blue
$ echo $colors

$ echo ${color}s
blues
```

Note that in the C Shell the first line above would be replaced with "set color=blue" or if exportation is desired "setenv color blue". To export a shell variable in a Bourne Shell to a subshell one must use the export command: export color.

You can remove the value that has been assigned to a variable simply by assigning it the empty string.

```
$ color=
$ echo ${color}

$
```

A variable can be declared to be read-only by using the readonly command. Once this is done the variable may not be changed.

```
$ max=10
$ echo ${max}
10
$ max=11
$ echo ${max}
11
$ readonly max
$ max=12
max: is read only
$
```

You should assign a value to a variable before declaring it to be read-only. Since the default value of a variable is the empty string, if you declare a new variable to be read-only before that variable has been assigned a value, it will be automatically assigned the empty string as its read-only value.

The Bourne Shell read command allows you to assign to shell variables information that is provided in the standard input. The read command reads one line from the standard input and assigns the first word (nonblank sequence of characters) to the first variable, the second word to the second variable, etc. If more words appear on the input line than variables in the read command, then all extra words are also assigned to the last variable.

```
$ cat fun
echo "What are your two favorite pets?  \c"
read pets
echo "Your favorite two pets are :  ${pets}"
$ fun
What are your two favorite pets?  dog and cat
Your favorite two pets are : dog and cat
$

$ cat cmd
```

```
echo "What is your favorite Unix command?   \c"
read nice_cmd
$nice_cmd
echo "I like the command $nice_cmd also."
$

$ cmd
What is your favorite Unix command ?   date
Tue Aug 24 10:26:29 MST 1993
I like the command date also.
$
```

Consider this last example that used the read command to assigns words to several shell variables.

```
$ read w1 w2 w3 w4
This is a very good example.
$ echo $w1
This
$ echo $w2
is
$ echo $w3
a
$ echo $w4
very good example.
```

Meta-characters in the Bourne and C Shells. The Bourne and C Shells both recognize several characters as standing for something other than their literal self. This allows the user to use regular expressions in shell commands. * matches any string of zero or more characters.

Examples:

```
$ echo * # prints names of all files
         # in working directory except
         # those starting with "."

$ rm *

$ ls -l *.c

$ more ch*.tex
```

? matches any single character.

Example:

```
$ echo c?e3?? # matches cse320 and cse396
              # but not cse486 or cse172

$ ls ??   # list all filenames in the
          # working directory that
          # contain exactly two
          # characters
```

Pattern matching in the shell is a bit different than it is in the editor. Patterns are anchored. Thus the symbol that matches any single character (?) in the shell matches only single character filenames. (In "ed" or "vi" the dot matches every non-blank line.)

'...' single quotes protect metacharacters from being interpreted. When quoted, metacharacters simply represent their literal values.

Examples:

```
$ echo '*'
*

$ echo x '*' y
x * y

$ echo '?'A
?A
```

Notes : Arguments to echo separated by multiple spaces or tabs are printed with a single space separating them.

```
\   the backslash is an escape character which turns
    off the special meaning of a metacharacter.
```

Example:

```
$ echo \*
*
```

Note that a backslash at the end of a line causes the line to be continued.

Example:

```
$ echo abc\
    def\
    ghi
    abcdefghi
```

\`...\` this is called command line substitution. The output of the command represented by ... is substituted for the command itself.

Example:

```
$ echo The date is: `date`
The date is: Thu Aug 8 15:48:01 MST 1991
```

When interpreting output in backquotes as argu- ments, the shell treats newlines as word separators.

Example:

```
$ cat file1
one
two
$ echo `cat file1`
one two
```

"..." double quotes protect all metacharacters except $, \`...\` , \

Examples:

```
                    $ A=10

                    $ echo "$A"
                    10

                    $ echo '$A'
                    $A

                    $ echo "`date`"
                    Thu Aug 8 15:48:01 MST 1991

                    $ echo "\$A"
                    $A
```

# comment delimiter. Everything after the # and up to the newline is ignored. (Bourne Shell)

Example:

```
          $ ls # hello

     [...]   matches any single character in the brackets.
             This is called a character class.

          - to get "]" inside, put it first

          - ranges may be given, some examples are:
               [0-9],[a-z],[A-Z],[0-9f-v]

          - ^ as a first element is the negation operator
```

Example: `[^0-9] = any character not a digit`

Example:

```
               echo c?e[34][0-9]*  # matches cse320,cse486,
                                   # cxe39xxf, cse396
```

Shell script parameters. It is possible to write programs that consist only of shell commands. These are called shell scripts or shell programs or shell files. We will write shell programs using Bourne Shell commands. All discussions concerning shell programs should assume that the programming is being done using Bourne Shell attributes and facilities.

When you invoke a shell script you can pass parameters to the shell program. These shell command line arguments are called positional parameters.

The name of the nth positional parameter is $n. Consider the following invocation of shell script "sh_cmd" and its two arguments f1 and f2:

```
     sh_cmd f1 f2
```

Note that the same command line syntax applies as we discussed before. Now consider:

```
     f1 is the first positional parameter. Inside the
     shell script (sh_cmd) it is referred to as $1.

     f2 is the second positional parameter. Inside the shell
     script (sh_cmd) it is referred to as $2.
```

```
The positional parameter named $0 is the name of the
shell script itself.
```

Example:

```
$ cat sh_cmd
echo "0th positional parameter = $0"
echo "1st positional parameter = $1"
echo "2nd positional parameter = $2"
$ sh_cmd bill 10
0th positional parameter = sh_cmd
1st positional parameter = bill
2nd positional parameter = 10
```

In the example above, "sh_cmd" is simply an ascii file that could have been created with any editor. Also note that after "sh_cmd" was created it needed to be given executable permissions. This is not shown in the above example. Now consider the following example that makes use of two of the special variables set by the Bourne Shell.

Example:

```
$cat sh_file2
echo "All arguments = $*"
echo "Number of arguments = $#"
$sh_file2 1 sam "hi there"
All arguments = 1 sam hi there
Number of arguments = 3
```

You may also explicitly set the shell (positional) parameters of any Bourne Shell by using the set command with arguments. Each of the arguments to the "set" command will become positional parameters of the current shell after the "set" command is executed.

Example:

```
$ set 2 sam "hi there"
$ echo $1
2
$ echo $2
sam
$ echo $3
hi there
```

Example:

```
$ echo $#
0
$ set a b c
$ echo $#
3
```

Note that in the C Shell the set command is used for assignment of local shell variables and the setenv command is used for assignment to shell variables that are to be exported to subshells.

Example:

```
(19)% setenv g "g value"
(20)% set t="t value"
(21)% cat sh_file
echo "g = $g"
```

```
          echo "t = $t"
          (22)% sh_file
          g = g value
          t =
```

Input/output redirection (Bourne and C Shells)

Every program has three default files created when it starts execution (e.g. at login). These are:

1. standard input (stdin) is opened first
2. standard output (stdout) is opened second
3. standard error (stderr) is opened third

Each file is numbered by small integers called file descriptors. File descriptors are generally integers between 0 and 19 (the limit is system dependent) which may be used to reference files that have been opened for input and/or output. File descriptors are offsets into the Unix System Process File Descriptor Table.

1. standard input (stdin) has file descriptor 0
2. standard output (stdout) has file descriptor 1
3. standard error (stderr) has file descriptor 2

Each individual Unix process has its own File Descriptor Table. This table contains pointers to information about each open file. This information is available to its associated process.

I/O redirection is accomplished as follows:

1. A standard file (input,output,err) is closed.
2. The file named for redirection is opened.
3. A new file pointer replaces the closed File Descriptor Table entry.

The symbols < , > , and >> have special meaning on a shell command line and are used to specify redirection :

```
       <  redirects current input to the named file
```

Example: `wc -l < file1.c` In this example, the input is initially stdin. > redirects output to the named file

Example: `ls > list.out` In this example, the output is initially stdout. (wc -l >&file1.c # csh stderr/stdout redirection) >> redirects output to the named file but does so as an append operation

Example: `ls >> oldlist.out`

Examples:

```
       $ who > temp
       $ wc -l < temp   # prints the number of users logged in

       who >temp
       grep mary <temp # Is mary logged in ?
```

It is important to understand that the interpretation of < , > , and >> is being done by the shell. This allows you to use these special symbols with any program.

```
sort < temp   In this example sort is used with no
              arguments and so it defaults to
              reading its standard input. Its
              standard input is, however, redirected
              so that it comes from file "temp" and
              not the terminal. This redirection is
              taken care of by the shell. Utility
              sort knows nothing about a file called
              temp; it simply reads from
              stdin.

sort temp     Here sort sees the filename "temp" as
              an argument. It must open this file
              called "temp", read it, and sort it.

sort          Utility sort now waits for the user to
              type in the input since the default
              standard input is the terminal keyboard
```

The following examples should give you some more ideas about what is going on underneath in the operating system internals when redirection is done.

Example:

```
cat file.old > file.new
```

The above example results in the following:

1. The default standard output file is closed.
2. file.new is opened for writing and becomes the new standard output.

Example:

```
cat > file1        FILE DESCRIPTOR TABLE
                0  13579
                1  13589
                2  13593

The default standard output gets closed.
file1 gets opened and the pointer 57924 is returned.

                   FILE DESCRIPTOR TABLE
                0  13579
                1  57924
                2  13593
```

Dereferencing shell variables and command line substitution (`...`) can be used when specifying filenames for use in redirection but stay away from using other metacharacters for this purpose.

Examples:

```
$ wc -l < *.c         NOT GOOD

$ wc -l < "${dest}"   OK

$ a=defs.h
$ cat < $a            OK

$ cat aa
defs.h
$ cat < `cat aa`      OK
```

The Bourne Shell supports redirection in these additional ways:

```
echo hello      is equivalent to echo hello >&1

2 > &1          put stderr on same stream as stdout

1 >&2           put stdout on same stream as stderr

2 >file         redirect stderr into file

2 >>file        append stderr onto file

1 >>file

1>file

test.err 2>&1 >err.file
   stderr -> stdout
   stdout -> err.file  # &1 and &2 are now different

test.err >err.file 2>&1
   stdout -> err.file
   stderr -> stdout=err.file # &1 and &2 are same now

sh <numusers  --> stdin=numusers
sh numusers   --> stdin=terminal; sh runs numusers
```

The C Shell allows one to redirect standard out and standard err to the same file by using an ampersand as shown below.

```
cat file1 file2 >& file.out
```

Pipelines and Processes (Bourne and C Shells)

Pipes. The pipe facility was a fundamental contribution of the Unix system. Pipes often can free one from using temporary files.

```
pipe      connects the output of one program to the
          input of another without using a temporary
          file

pipeline  a connection of two or more programs
          through pipes
```

Examples:

```
who | sort

who | wc -l  # number of users logged in.

ls | wc -l   # number of files in current dir

who | grep jpl  # Is jpl logged in ?
```

Any program that reads from stdin can read from a pipe instead.

Any program that writes to stdout can write to a pipe instead.

This is the importance of the convention that programs write by default to stdout and read by default from stdin.

Programs in a pipeline all run at the same time; the kernel schedules and synchronizes them.

The exit status of a pipeline is non-zero if the exit status of the last command in the pipeline is non-zero.

Example:

```
$ ls
file1 file2
$ cat file1
1
2
3
$ cat file1 | grep xxx
$ echo $?
1
```

filter - a program designed to fit as a stage in a pipeline.

Characteristics of a filter :

1. Takes input from stdin by default (needs no filename).
2. Sends output to stdout (needs no filename).
3. Performs well defined transformation on input.
4. Produces output with no header/trailer info etc.
5. All of input is simply data.
6. Generally is not interactive.
7. Error/diagnostic output goes to stderr (not to stdout).

Processes. Processes have a hierarchical structure. Each process has a parent and may have children. Your shell was created by a process associated with whatever terminal line connects you to the system. As you run commands those processes are the direct children of your shell. A child process inherits user/group ids, open I/O channels, and environment variables.

```
process     - an instance of a running program
            - the execution of an image
            - compiled program + process environment

image        a computer execution environment,
             including contents of memory, register
             values, name of current directory, status
             of open files, info recorded at login
             time, etc.

process-id   processes are identified internally by an
             id number

&            allows a command to be started and run in
             the background

             wc bigfiles* > bigcount.out &

             Note that & applies to commands and since
             pipelines are commands you don't need
             parentheses with pipelines:

             pr file | lpr &
```

```
;           command separator

            who ; date # commands are run in sequence

            date ; who | wc --> date;(who|wc)
               # a pipeline is one command

            (date;who) | cat
               # all output goes into pipe


kill [ -signal] <process-ids>  default signal is
                               15 (terminate)


kill 0   kill all processes resulting from current
         login (except the shell)

kill -9 <process-ids>  a sure kill since it can't be caught

ps      - displays status of current processes
        - lists processes you are running in
          foreground and background.
     -a = all processes most frequently requested,
     -e = every process now running
     -l = long listing
```

Examples:

```
$ wait     # wait until all processes started with
             & have finished

$ pr ch* | lpr -Prp66 &       # pr formats and prints file
6951       # pid is printed

$ kill 6951
```

The following C Shell commands are often useful.

```
jobs      - displays existing stopped/suspended jobs

fg <job_id>  - can use %<job_number> , %+ , %- ,
              %<command_name_prefix> , or <text>
              <text> is a prefix of a job's command.

%+                - most recently suspended job

%-                - next most recently suspended job

ctrl-z    - stops the current job
```

Miscellaneous Commands.

```
tee        saves data flowing through a pipe into a
           file (Example: echo hi|tee f1 f2 f3|cat)

tee -a     appends input to existing files

sleep n    wait n seconds
grouping   (sleep 5;date) , (sleep 5;date)&
```

Shell Programming in the Bourne Shell

Creating new commands

```
$ cd
$ echo 'who | wc -l' > numusers
$ chmod +x numusers
$ mkdir bin   # do this if you have no bin already
$ mv numusers bin

Now change your PATH shell variable in your .login
file to include your bin ($HOME/bin) if it does not
include it already.

Note: If a file is executable and it also contains
text, then it is assumed by the shell to be a file
of shell commands.

A shell script will be run in the Bourne Shell unless
a "#" character is placed at the very start of that
file.

Note: It is a good idea to make the first line of
    every shell program  #! /bin/sh  or whatever the
     desired shell is.
```

Control Structures (for the Bourne Shell)

```
for <variable> in <a list>
do
   <command list>
done

<a list>   a list of blank separated strings
<variable> takes on value from each string in
           <a list>
```

Examples:

```
for i in $*    =  for i
do                do
    echo $i          echo $i
done              done

for i in *
do
    echo "`ls -l $i` `file $i`"
done

for i in *
do
    echo "The type of the file named $i:"
    echo
    file $i
done


for i in *
do
    cat $i
done | grep smith >smith.out #note where loop
                              output goes
```

Examples of some for loop header lines:

```
      for i in $*
      for i in `cat file.lst`
      for i in 2 4 6
      for i in file1 file2 file3
```

Example using expr :

```
   set 1
   for i in *.c
   do
      echo "### File $1   ###"
      cat $i
      set `expr $1 + 1`
   done
```

Note that "expr" can perform arithmetic operations on its args. But many operators (e.g. parentheses, ampersands,asterisks) are shell metacharacters and so must be escaped.

```
   count=10
   count=`expr $count + 1`
   expr 4 + 5
   expr 5 \* 4
```

Exit status.

Every command returns an exit status. An exit status is a value returned to the shell to indicate success or failure.

```
    0 = success/true

    nonzero = not successful/false
```

Example:

```
 grep returns 0 if a match occurred
                               1 if no match occurred
                               2 if an error occurred
                                 in pattern or filenames
                 $ grep hello xxx    # there is no xxx
                 $ echo $?
                 2
```

The test command.

"Test" is a useful command to use with the "if" statement. The sole purpose of "test" is to return an exit status.

```
test options for files :

   -r     true if file is readable
   -w     true if file is writable
   -x     true if file is executable
   -s     true if file size > 0 and is a file or
          directory (false on device files etc.)
   -d     true if file is a directory
   -f     true if file exists and is not a directory


test options for strings:
```

```
-z        true if string is length 0
-n        true if string length > 0
s1 = s2  true if string s1 equals string s2
s1 != s2 true if string s1 does not equal sting s2
s1        true if s1 is not the empty string
```

Example:

```
$ test ""
$ echo $?
1
```

test options for numeric operators: -eq, -ne, -gt, -ge, -lt, -le test options for logical operators: -a and operator -o or operator Notes: The test command does not allow concatenation of options. For example, the command test -rw file is illegal; you must use the command test -r -w file In the command test "3"="4" , "3"="4" is seen as one string. To get a string comparison you must use spaces as shown here test "3" = "4" . Selection statements. if <command list1> then <command list2> fi if <command list1> then <command list2> else <command list3> fi if <command list1> then <command list2> elif <command list3> then <command list4> else <command list5> fi Here control flow is based on the success or failure of the executing programs. True means that the last command in the command list had an exit status of 0. (An exit status indicates success or failure.) The exit status of the "if" statement is the exit status of the last command in any "then" or "else" clause. Also note that "if", "then" ,and "else" are recognized only after a newline or semicolon. if test -f shl.in; then echo hi; fi ! is used for negation (if test ! -w $fname ...) Note: the space after the "!" is important.

Examples:

```
if test -r $filename
then echo "$filename is readable."
fi

if test 4 -lt 5 -a 5 -lt 6
then echo yes
fi

if test \(3 -gt 2 \) -a \(4 -eq 4 \)
then rm oldfile
fi

if cp foo1 foo1.cpy
   cp foo2 foo2.cpy
   test -f foo3
then
  echo YES
else
  echo NO
fi


for next in *
   do
      if test -f $next
      then
          echo "$next is a file."
      elif test -d $next
      then
          echo "$next is a directory."
      else
          echo "$next is neither."
      fi
   done
```

```
test -f "$fn" is equivalent to [ -f "$fn" ]
where the spaces are important

if [ -f sfle ]
   then echo yes
fi
```

Examples with strings:

```
   ST1=hi
   ST2=ho
   if test $ST1 = $ST2
   then echo no
   fi
```

While Loops

```
while <command_list>
do
        <command_list>
done
```

Example 1:

```
$ ls
tst1  tst2   tst3
$ while test -f tst3
> do
>       if test -f tst1
>       then
>               rm tst1
>               echo STEP 1 DONE
>       else
>               rm tst2
>               rm tst3
>               echo STEP 2 DONE
>       fi
> done
STEP 1 DONE
STEP 2 DONE
$
```

Example 2:

```
$ cnt=1
$ while test $cnt -ne 5
> do
>       echo $cnt
>       cnt = `expr $cnt + 1`
> done
1
2
3
4
```

Until Loops

```
until <command_list>
do
        <command_list>
done
```

Example 1:

```
$ cnt=1
$ until test $cnt -eq 5
> do
>       echo $cnt
>       cnt=`expr $cnt + 1`
> done
1
2
3
4
```

Example 2:

```
$ cat mailwhenlogson
until  who|grep "${1}" > /dev/null
do
        sleep 300
done
mail "${1}" << end
Glad to see that you have logged on.
end
$ mailwhenlogson jpl &
19875
$ You have new mail.
```

The break and the continue commands. The break and the continue commands work as expected with the for, while, and until loops.

```
cnt=0                                            OUTPUT:
while true
do                                               5
        cnt=`expr $cnt + 1`                      6
        if test $cnt -eq 10                      7
        then                                     8
                break                            9
        elif test $cnt -lt 5
        then
                continue
        fi
        echo $cnt
done
```

The shift command

Using the positional parameters ($1, $2, ... , $9) only allows you to access the first nine command line arguments to a shell script. The shift command allows you to access any number of command line arguments. Each time the shift command is executed, the second positional parameter ($2) becomes the first positional parameter ($1) , the third ($3) becomes the second ($2) , etc. The argument that was the first positional parameter before the shift command was executed is lost (made inaccessible).

Example:

```
$ cat printargs
cnt=1
while test "$1" != ""
do
        echo "Argument $cnt = ${1}."
```

```
                cnt=`expr $cnt + 1`
                shift
        done
        $ printargs 1 2 3 4 5 6 7 8 9 10 11 12
        Argument 1 = 1.
        Argument 2 = 2.
        Argument 3 = 3.
        Argument 4 = 4.
        Argument 5 = 5.
        Argument 6 = 6.
        Argument 7 = 7.
        Argument 8 = 8.
        Argument 9 = 9.
        Argument 10 = 10.
        Argument 11 = 11.
        Argument 12 = 12.
```

The shell logical operators

```
||   logical OR

     test -f file1 || echo "file1: not a simple file"

     The command to the left is executed. If its exit
     status is zero, the command to the right is
     ignored. Otherwise the command to the right is
     executed and its exit status is returned.
     (short circuit evaluation)

&&   logical AND

     && executes its right operand only if its left
     one succeeds (i.e. returns 0 exit status)

NOTE: The following may be helpful.
   true
   false
   :   (returns true --> while :)
```

Example:

```
$ false
$ echo $?
255
$ true
$ echo $?
0
$
```

Miscellaneous Examples

```
cc prog.c lib.c & cd /$HOME/bill ; ls -l

   The shell creates a new process and executes cc
   in that process. Without waiting for cc to end the
   shell sequentially executes the cd and ls commands

test -d  ../tools && cd ../tools

   If ../tools exists and is a directory, then make
   it the current working directory.

test -d ../tools || mkdir ../tools
```

```
        If directory ../tools does not exist, then create
        it.
```

The case statement

```
case <word> in

<pattern> ) <command_list> ; ;

<pattern> ) <command_list> ; ;

  . . .

esac
```

The <word> is compared to the patterns from top to bottom. The first pattern matched (and only the first pattern matched) causes the command list associated with that pattern to be executed.

In the following shell program, which is a modified version of an example given in The UNIX Programming Environment by Kernighan and Pike, the case statement is used to make the cal command more user friendly.

```
case $# in

0)      set `date`; m=$2; y=$6;;  # No args given. Give data from today's date.

1)      m=$1; set `date`; y=$6;;  # One arg given. Use this year.

*)      m=$1;  y=$2 ;;             # Two args given (month and year).

esac

case $m in

[jJ]an*)                m=1 ;;

[fF]eb*)                m=2 ;;

[mM]ar*)                m=3 ;;

[aA]pr*)                m=4 ;;

[mM]ay*)                m=5 ;;

[jJ]un*)                m=6 ;;

[jJ]ul*)                m=7 ;;

[aA]ug*)                m=8 ;;

[sS]ep*)                m=9 ;;

[oO]ct*)                m=10 ;;

[nN]ov*)                m=11 ;;

[dD]ec*)                m=12 ;;

[1-9]|10|11|12)  ;;      # A numeric month was given. No need to convert.

*)                      y=$m;  m=""  ;; # Just the year was given.

esac

/usr/bin/cal $m $y              # Call the system cal command.
```

Here Documents

A here document is found within a shell program and has the following form.

```
<command> << <end_word>
<Text of here document>
<end_word>
```

The symbols << mark the beginning of the here document. The user-defined <end_word> can contain one or more characters and is used to mark the end of the here document when it is placed on a line by itself. The command itself reads from the standard input which has been redirected so that the command actually is reading from the text of the here document. All characters found within the text of the here document are interpreted literally (i.e. none are treated as metacharacters).

Example 1:

```
$ mail jpl << end
>This is a test.
>1 2 3.
>end
$
```

Example 2.

```
$ mail jpl << end
>This is a test.
> 1 end 2 3
>end
$
```

Example 3.

```
$ cat << end
> This is fun.
> 1 2 3
> end end end
>end
This is fun.
1 2 3
end end end
$
```

Example 4.

```
$ cat phonenums
grep -i  $1  << endnums
Sally Bumkin    669-6671
Bob Smith               667-4352
Fred Zilensky   667-3455
endnums
$ phonenums Smith
Bob Smith               667-4352
$ phonenums 667-
Bob Smith               667-4352
Fred Zilensky   667-3455
```

The shell trap command

The shell trap command provides for a sequence of commands to be executed when a signal is detected. When a signal is sent to a running program, that program is terminated unless it includes explicit actions for handling that signal.

```
trap  '<command list>'  <list of signal numbers>
```

Shell Signal Numbers

0	Shell exit for any reason including the end of file.
1	Hangup.
2	Interrupt. (ctl-c)
3	Quit (ctl-\). This causes the program to create a core dump.
9	Kill. (Cannot be caught or ignored.)
15	Terminate. (This is the default signal generated by kill.)

The following program will allow you to experiment with trapping different type of signals.

```
$ cat mytrap
trap 'echo "Shell exit for any reason."; echo; exit 0' 0
trap 'echo "Hangup signal."; exit 1' 1
trap 'echo "Interrupt signal."; exit 2' 2
trap 'echo "Quit, ctl-\\, signal."; exit 3' 3
trap 'echo "Kill that cannot be caught or ignored."; exit 9' 9
trap 'echo "Terminate signal kill."; exit 15' 15
cnt=1
while test ${cnt} -lt 1000
do
        cnt=`expr ${cnt} + 1`
done
$ tst
^CInterrupt signal.
Shell exit for any reason,

$ tst
^\Quit, ctl-\, signal.
Shell exit for any reason.

$ tst &
25204
$ kill 25204
$ Terminate signal - kill.
Shell exit for any reason.

$ tst &
25356
$ kill -9
$
```

Misc:

```
$ at 9:05am
at> <cmd1>
at> <cmd2>
^D
$
```

Using the Internet

The Internet is almost a mythical creature at this point. The tools which you should learn in the **TCP/IP** suite:

- e-mail
- telnet
- ftp
- lynx
- netscape

There are other tools, but these are sufficient to get by with considering the importance of the World Wide Web.

An online course (one of many) can be found at the URL (Universal Resource Locator): http://www.brandonu.ca/~ennsnr/Resources/Roadmap/

A more interesting, and challenging, Internet experience is to partake in the Internet Hunt.