
# Advanced UNIX

Grut Computing Systems, Inc.
333 Melrose Drive, #10B
Richardson, TX 75080
(940) 894-6623

**Advanced UNIX** was developed and written by Grut Computing Systems, Inc.



Grut Computing Systems, Inc.
333 Melrose Drive, # 10B
Richardson, TX 75080
(940) 894-6623

**Important Notice:**

The purpose of this student manual is to serve as a supplement to the instructor-based technical presentation in the classroom. The work is not intended to be used as a self-contained reference manual on its own.

Revised November, 2015

**Advanced UNIX**

## COURSE ABSTRACT

The Advanced UNIX course extends the Unix Shell Scripting class into the world of advanced UNIX system tools and programming. This class will focus on those tools that extend the UNIX shell to its limit. Advanced UNIX commands, constructs, advanced sed and awk, and extended shell facilities will be explored to their limit. Also, techniques, error and signal trapping, and extended shell expressions will be covered.

## AUDIENCE

This course is designed for UNIX and Linux users, administrators, and programmers.

## DURATION

4 Days

## LEARNING OBJECTIVES

**Upon completion of this course, the participant will be able to:**

- Use the full complement of regular expression metacharacters for pattern matching
- Use the full complement of options with fgrep, grep, and egrep
- Create portable and generic system tools using Bash scripting
- Create and interpret complex Bash (and Korn) Shell script files
- Employ secure UNIX practices within shell scripts

## PREREQUISITES

Introductory programming skills using UNIX is recommended. Familiarity with basic UNIX commands, basic Bash or Korn Shell Scripting and familiarity with any UNIX text editor (emacs, vi, nano, etc.) is required.

# Table Of Contents

# Foundational Review

## The Process

A process is two things; an environment and a program executing in that environment. Each command the user enters, by default, causes a new process to be created. Each process has associated with it an environment and user information. Each process has an id that is unique to that process.

A process that spawns another process is called a parent process. Its spawned process is called a child process.

A parent process creates a child process through the *fork* system call. A *fork* creates a new process identical to the parent process with a new process id. This child process inherits from its parent, among other things, the working directory, permission settings, open files, file creation size limits, memory restrictions, and exported variables.

When a child process terminates, control is returned to the parent process.

(Note: An *exec* system call can be made to overlay the current running process with a new process, i.e., no child process is spawned.)

## Foundational Review (continued)

### Logging In

A Linux process called *init* is always running. One of its jobs is to launch processes which open ports for access. These ports may allow access to both secure and non-secure sessions. A secure shell server, which may actually be launched through other processes, waits for a user to access that port (usually port 22.) Once the user enters their user-id and the <Enter> key, Linux *exec*s a login process which takes the submitted user id as its argument. The login process requests the user's password. Once the password has been entered, the password is encrypted and compared it with that user's current encrypted password which is kept in a disk file *(/etc/passwd or /etc/shadow*). If the two encrypted passwords match, the user is allowed into the system.

The user's startup routine, usually a shell, is then executed, followed by the commands in the system login procedure *(/etc/profile*) and the user's login procedure(s) execute. The order of execution for a *bash* login is normally as follows:

if *.bash_profile* exists it will execute. If it does not exist, but *.bash_login* exists, it will execute. If it does not exist, but *.profile* exists, it will execute.

After the above, if a *.bashrc* file exists it is usually invoked from .bash_profile. (Also, a *.bashrc* file will re-execute anytime a new, interactive Bash is spawned.)

The shell, which is a running process, spawns child process' at the user's request and terminates when the user requests it.

#### Logging Out

The user is logged out when their highest parent process (their login shell) is terminated. In Bash, if a *.bash_logout* file exists, it will execute before logging out.

A running shell is terminated with a <ctrl> d or by typing, within the running shell, the *exit* command.

Foundational Review (continued)

## A Few Commands

cd change directory

Examples:

```
cd  /etc      # Current directory now /etc
cd  ..        # Current directory is my former parent directory
cd  -         # Current directory is my prior directory
cd            # Curent directory is my home directory
```

echo print arguments to the terminal

Examples:

```
echo “This is a single argument”
echo These are multiple arguments
echo “My home directory is $HOME”
```

exit terminate the current shell

Examples:

```
exit     # Kill this shell
exit 0   # Kill this shell, setting this shell’s return code to 0
```

ls list files

Examples:

```
ls            #  List all non-hidden files, by name, in current
              # directory
ls  –a        #   List all files, by name, in current directory
ls  –l        #  List all non-hidden files, in “long” format, in current
              # directory.
ls  –al  /etc # List all files, in “long” format, in /etc directory
```

## Foundational Review (continued)

**man** on-line manual pages (documentation). Piped into the *more* command, by default.

Examples:

```
man ls          # Show the ls man pages
man man         # Show the man man pages
```

**passwd** change passwords

Examples:

```
passwd          #  Change my password
passwd elvis    #  Change elvis' password, if I have root privileges
```

**ps** report process status

Examples:

```
ps              # Show my processes attached to this login
ps –e           # Show all processes, mine, and others
ps –u elvis     # Show all elvis' processes
ps –u elvis –f  # Show all elvis' processes, in "full" format
ps –e –l        # Show all processes in "long" format
```

**touch** create a file or update an existing file's timestamps

Examples:

```
touch fruit             # Create fruit, or, if fruit already exists,
                        # update its last accessed, modified, and
                        # change timestamps to now.
touch –a fruit          # Update only last accessed time to now.
touch –c fruit          # Update timestamp only, but do not create
touch –r veges fruit        # Update fruit's accessed and
                            # modified timestamps to match veges
                            # timestamps.
touch –t 200709151200.00 fruit
                        # Update accessed and modified timestamps
                        # to Sep 15, 2007 at 12 noon.
```

Foundational Review (continued)

Lab 1

List out the contents of /etc/profile. Examine its contents.
List out the contents of your .bash_profile. Examine its contents.
List out the contents of your .bashrc file. Examine its contents.
List out the contents of /etc/passwd. Examine its contents.

Ensure you are comfortable with the general concepts of all three files.

## Foundational Review (continued)

## The File System

The UNIX file system is similar to the Windows file system. Structurally, both descend from a mandatory root directory into other sub-directories. The root directory and sub-directory delimiter in UNIX is the forward slash ( / ). (Not to be confused with the Windows root directory and sub-directory delimiter which is the backwards slash ( \ ).)

However, UNIX does not have drive designators such as C:, D:, etc. Instead, UNIX uses mount points. A mount point is nothing more than a conceptual D: drive traversed to whenever you change directories to the D drive's mount point (we'll see mount points later on.) It's usually transparent to the user when one traverses to a different drive, but sometimes it's important

Also, the mounted file system may be a temporary file-system, an in-memory file-system, or a file-system on a different machine. Again, this is usually transparent to the user.

Keep in mind that within your UNIX session, each running shell will have its own current directory. For example, if you change directories within a shell script, it will have no affect on the current directory of the launching shell. That's a good thing.

## Foundational Review (continued)

### Which command?

When you submit a command to the shell, either interactively or through a shell script, the shell has a standard set of rules for locating the command.

First, the shell determines what the command is based on whatever is typed before the first white space character.

And then, for *bash*, the following sequence is followed:

1. If the command is fully qualified (the file system location of the command is specified), the command is loaded and run.

2. If not fully qualified, the shell determines if the command is an alias.

3. If not an alias, the shell determines if the command is a keyword (reserved word, usually a programming construct word) such as *if* or *for*. If it is, it's executed.

4. If not a keyword, the shell determines if the command is built-in.

5. If not a built-in, the shell determines if the command is a function.

6. If not a function, the shell looks at the contents of the environment's PATH variable. It will look for the command, residing on disk, in each directory specified in the PATH variable. If it finds it, it will execute it. If not, an error message is written to your terminal.

Note: If running a *Korn* shell, flip items 2 and 3 above.

## Foundational Review (continued)

### Shell Metacharacters

Metacharacters are characters that mean something special to the shell. The Bash and Korn shells have several metacharacters which follow:

### Filename Matching Wild Cards

| | |
|---|---|
| ? | Any single character. |
| [xyz] | x, y, or z. |
| [!xyz] | Not x or y or z |
| * | Any combination of characters. |

## Foundational Review (continued)

## Other Shell Metacharacters

| | |
|---|---|
| \ | Literal value of the following character. |
| & | Process command in background. |
| ; | Separate multiple commands on one command line. |
| $ | The value stored in the suffixed variable. |
| (commands) | Execute commands in an anonymous sub-shell. |
| {commands} | Execute commands in this shell. |
| . command | Execute command as part of current shell. |
| ~ | My home directory. |
| ` command` | Output of command. (reverse tick) |
| ‘item’ | Treat item as literal. (single quote) |
| “item” | Treat item as literal, except $ and reverse ticks, preserve spaces, newlines, and tabs. (double quote) |
| command1 \| command2 | Use output from command1 as input to command2. |
| > | Redirect Output |
| < | Redirect Input |
| >> | Append Output |

## Foundational Review (continued)

### A Few commands

alias — Rename a command.

Example:
```
alias psu="ps –u napoleon"
alias grep="grep –i"
alias c=clear
```

command — Show how a command would be executed or bypass function lookup. What does UNIX actually see once the shell is finished massaging the invocation.

Example:
```
command -v psu
command -v ls
```

env — Print the current environment. Your shell's environment consists of those variable that have been *export*ed. An *export*ed variable has a copy of itself sent down to any sub-shells you spawn (either explicitly, or via the default invocation of a shell script.

sleep — Suspend execution for *x* seconds.

Example:
```
sleep 10
```

type — What type of command is this?

Example:
```
type cd
type if
type ls
type functions
```

unalias — Remove an existing alias.

Example:
```
unalias ls
unalias c
unalias grep
```

## Foundational Review (continued)

whence — Show how a command would be executed. (Similar to *command –v*, but built in to the shell.)

which — For commands stored as utilities somewhere on the file-system, in which directory will they be found? This command actually looks through your current PATH setting to find the first occurrence of the command you passed as which's argument.

Example:

```
which ls
which grep
```

### Lab 2

What directory does the *touch* command reside in?
What type of command is *cd*?
What type of command is *case*?
Create an alias named *psu*. Its functionality is “ps –u $LOGNAME”.

## Foundational Review (continued)

### Variable Creation and Assignment

To create a shell variable, simply assign a value to your variable. There must not be any spaces on either side of the =.

### Variable Naming Convention

All variable names must begin with a letter (or an underscore), followed by zero or more alpha-numeric chars or underscores.

### Retrieving a Variable's Value

To retrieve a variable's value, simply prefix the variable's name with a $. You have the option of enclosing the variable's name in { }. The { } surrounding the variable protects the variable's name from any other valid characters that might be interpreted as part of the variable name. Also, variables that the shell sets (those beginning with a numeric or punctuation character) are assumed to be one character long (unless protected by { } .) If other characters are appended to the variable name, the shell would consider them as being literals.

Example:

Input:

```
MYVAR=HELLO
echo $MYVAR, World      #  Yields:  HELLO, World
echo ${MYVAR}, World    #  Yields:  HELLO, World

X=hot
echo $Xdog              #  Yields:  (the contents of
                        #  variable Xdog, which is probably nothing.)

echo ${X}dog            #  Yields:  hotdog

echo $1                 #  Yields the contents of variable 1

echo $10                #  Yields the contents of variable 1, followed by
                        #  a zero

echo ${10}              #  Yields the contents of variable 10
```

When in doubt, wrap { } around the variable's name. (But, again, most of the time it is strictly optional.)

## Foundational Review (continued)

### Capturing Command Output

The output of commands may be captured in environment variables. The method of doing this is to enclose the command in reverse tiks ( ` ).
Example:

```
LSLIST=`ls`              #  reverse tiks
LSLIST=$(ls)             #  same thing, different syntax

echo $LSLIST             #  Whitespace runs are translated to a
                         #  single space when not protect by " "

Yields:                  file1   file2   file3

echo "$LSLIST"           #  Whitespace is protected

Yields:                  file1
                         file2
                         file3
```

2 Rules for variable assignment:

Rule 1: On any assignment, there must not be any white space on either side of the =.

Rule 2: You can only assign 1 “thing” to a variable.

## Foundational Review (continued)

### Exporting Variables

If you are interested in making a copy of your variables available to any sub-shells you spawn (including shell scripts, Perl scripts, etc.) you must export those variables.

Example:

Input:

```
X=50
echo The Value Is $X
```

Output:

```
The Value Is 50
```

Input:

```
bash                    # spawn a sub-shell (simulates the first step in the
                        # running of a shell script.)

echo The Value Is $X
```

Output:

```
The Value Is
```

Input:

```
exit                    # kill the sub-shell, back to our original shell
                        # (simulates the last step in the running of a shell
                        # script.)
export X
bash
echo The Value Is $X
```

Output:

```
The Value Is 50
```

Input:

```
X=100
exit                    #kill the sub-shell, back to our original shell.
echo The Value Is $X
```

Output:

```
The Value Is 50
```

## Foundational Review (continued)

### Quoting Rules

The shell has several ways to treat quoted strings that differ from unquoted strings. Commands, wildcards, and environment variables expand differently depending on the quotes used.

Quoting symbols do not imply anything about data typing, e.g. string data vs. numeric data. They only indicate to the shell how to treat metacharacters (a character that has a special meaning to the shell.) That's all that 1) a lack of quotes, 2) single quotes, or 3) double quotes do.

Rules:

| | |
|---|---|
| No Quotes ( ) | Variables expand<br>Wildcards expand<br>Commands expand<br>Duplicate white space is removed |
| Double Quotes ( " ) | Variables expand<br>Wildcards do not expand<br>Commands expand<br>Duplicate white space is preserved<br>Protect everything else from metacharacter expansion.<br>Turn many things into one thing |
| Single Quotes ( ' ) | Variables do not expand<br>Wildcards do not expand<br>Commands do not expand<br>Duplicate white space is preserved<br>Protect everything from the shell<br>Turn many things into one thing<br>( Pretty much, what you see is what you get.) |

## Foundational Review (continued)

**Example Quoting Rules Session:**

(Assume we have 2 files in our directory, file1, file2)

(Assume the following variable assignment)

TESTVAR=”HELLO, MY NAME IS KIP”

**No Quote Example:**

Input:

```
echo  *  $TESTVAR `ls`
```

Output:

```
file1 file2 HELLO, MY NAME IS KIP file1 file2
```

**Double Quote Example:**

Input:

```
echo “*   $TESTVAR `ls` ”
```

Output:

```
* HELLO, MY NAME IS KIP
file1
file2
```

Single Quote Example:

Input:

```
echo ‘*   $TESTVAR `ls`’
```

Output:

```
*   $TESTVAR `ls`
```

## Foundational Review (continued)

### Other Metacharacter Rules

You can always protect any single metacharacter from shell interpretation by prefixing the metacharacter with a backward slash ( \ ).

For Example:

Input:

```
echo I once had a $100 bill.
```

Output:

```
I once had a 00 bill        #  $100 translates into the value of $1 followed
                            #  by 00
```

Input:

```
echo I once had a \$100 bill
```

Output:

```
I once had a $100 bill
```

### Lab 3

Type your name into the variable associated with a *read* statement. Put a tab between your first and last name. Print your variable in a manner that will reduce the tab to 1 space and then, in a manner that will preserve the tab.

Assign your name into a variable with a newline between the first and last name. Print your variable in a manner that will reduce the newline to a single space and then, in a manner that will preserve the newline.

Link to the instructors *fruit* file. Ask the instructor if you are not sure how. Store the contents of *fruit* in a variable named FRUIT. Print the contents of FRUIT as a single record and, then, as multiple records.

## Foundational Review (continued)

When included within either single or double quotes, the following “escaped” characters have special meaning:

| | |
|---|---|
| \a | Beep |
| \b | Backspace |
| \c | Suppress Newline |
| \f | Form Feed |
| \n | Newline |
| \r | Carriage Return |
| \t | Tab |
| \v | Vertical Tab |
| \\ | Backward Slash |
| \N | N is a 1, 2, or 3 digit octal number |

## Foundational Review (continued)

### The File System

The UNIX file system is an hierarchical file system with all directories descending from the root ( / ) directory. All UNIX file systems have a similar, predefined directory structure which may be added to by the UNIX users.

### A Typical UNIX Directory Structure

/

**bin dev etc home lib opt proc sbin tmp usr var**

| | |
|---|---|
| bin | User executable command files |
| dev | Device driver files |
| etc | Administrative files |
| home | May be aliased to /export/home |
| lib | Library files |
| opt | Third-party package installation directory |
| proc | Processes and kernel data structures (in memory) |
| sbin | Administrative executable command files |
| tmp | User temporary files (sometimes, in memory) |
| usr | Permanent non-changing files |
| var | Permanent changing files |

## Foundational Review (continued)

### The File Inode

Each file consumes 1 file inode. The file inode contains information about each file, such as its type, permissions, time stamps, size, links, ownership, name, and physical disk location. In order to access some of the inode information, the user may use the *ls -l* command:

Example:

```
ls   -l

-rw-rw-rw-  1  kip  staff  5432  May 1   10:00  bigfile
drwxr-xr-x  2  kip  staff  1024  May 15  12:30  mydir
```

| | |
|---|---|
| Field 1 | File type and permissions (- is normal file. d is directory) |
| Field 2 | File links or directory block sizes |
| Field 3 | File owner |
| Field 4 | Group owner |
| Field 5 | File size, in bytes |
| Field 6 | Last modified time stamp |
| Field 7 | File name |

## Foundational Review (continued)

### File Type and Permissions

Consider the following inode information:

```
-rw-r-----  1  kip  staff  5432  May  1  10:00  myfile
```

The first field contains the information on file types and permissions. It may be viewed as 10 columns of information as follows:

| Column 1 | Columns 2-4 | Columns 5-7 | Columns 8-10 |
|---|---|---|---|
| File Type | User Perms | Group Perms | Other Perms |
| - | rw- | r-- | --- |

In this example, the user (owner) has read and write permissions, the group members have read permission, and all others have no permissions.

## Foundational Review (continued)

### Changing File Permissions With chmod

You can change the permission of a file if you are its owner. You use the *chmod* command with either a symbolic or octal argument.

Method 1:

u = user
g = group
o = other
a = all

+ add privileges
- remove privileges
= assign privileges, regardless of current settings

r = read
w = write
x = execute

Examples:

| | |
|---|---|
| chmod u+x myfile | (add execute privileges for user) |
| chmod g-r myfile | (remove read privileges for group) |
| chmod o-w myfile | (remove write privileges for other) |
| chmod u+x,g-r,o-w myfile | (all with one command) |
| chmod +x myfile | (add execute privileges for all) |
| chmod u=rwx,g=rx,o= myfile | (assign read, write, and execute for user, read and write for group, and nothing for other) |

## Foundational Review (continued)

### Changing File Permissions

Method 2:

read = 4
write = 2
execute = 1

Examples:

chmod 740 myfile (user gets rwx, group gets r, other gets nothing)

chmod 777 myfile (all get everything)

chmod 700 myfile (user gets everything, all others get nothing)

## What Privileges Mean?

For a normal file:

| | |
|---|---|
| r | May read the file |
| w | May modify or delete the file |
| x | May run the file as a command |

For a directory file:

| | |
|---|---|
| r | May look at the contents of the directory |
| w | May create a file in the directory |
| x | May attach to the directory |

## Foundational Review (continued)

### Setting Default File Permissions With umask

You may use the *umask* command to set up default file permissions at file creation time. Your *umask* value is a bit mask used whenever a file is created. *umask* without an argument will show you the current value.

When setting your *umask* value, pass it a 3-digit argument (there are couple of exceptions to this.) The argument represents the permissions NOT given at file creation time for owner, group members, and others.

The maximum privileges for a normal file are read and write. Execute must be given manually, after creation, with *chmod*.

The result is the octal permissions set at creation time.

Example:

umask 027 (Set the umask value to 027)

touch newfile (Will have privileges of 640 or rw-r----- )

```
  666
- 027
  640
```

To check the current *umask* value, type:

umask

mkdir newdir (Will have privileges of 750 or rwxr-x---)

```
  777
- 027
  750
```

## Foundational Review (continued)

### Copying, Moving, and Renaming Files

cp

A copy of a file may be made with the *cp* command. Both a source and destination name is required. If multiple files are to be copied with one *cp* command, the destination must be a directory. (UNIX may not warn you that you are overwriting an existing file) Check the *man* pages for different option settings.

Example:

```
cp   goodfile   goodfile.bak #make a backup copy of my file

cp   /etc/passwd   .         #copy a file into my current directory.

cp  /etc/*   .               #copy all from /etc into my current directory.
```

mv

A file may be renamed or moved with the *mv* command. The *mv* command, within the same file system, will always do a rename. Across file systems, a physical move will take place. Both a source and destination name is required. If multiple files are to be moved with one *mv* command, the destination must be a directory. (UNIX may not warn you that you are renaming over an existing file.) Check the *man* pages for different option settings.

Example:

```
mv   newfile   oldfile       #rename newfile to oldfile

mv   /tmp/logfile   .        #move /tmp/logfile to current directory.

mv  /tmp/*   .               #move all from /tmp into my current directory.
```

## Foundational Review (continued)

### Deleting and Linking Files

rm

To delete or remove a file or files, use the *rm* command. Multiple files may be removed by using wildcards. (UNIX may not warn you if you are attempting to remove all files. UNIX should warn you if you do not have write permissions to the file, but own it and still want to remove it.) Check the *man* pages for different option settings.

Example:

```
rm   oldfile  oldfile2  #delete 2 files

rm   *          #remove all files.  Be careful.  No undelete command.

rm –i  f*       # prompt before each removal

rm –r g* z*     # recurse through sub-directories

rm –f  abc??  #  force delete on owned files that have no write privilege

rm –rf  *       #  recurse and force deletion
```

## Foundational Review (continued)

### Links

ln (Add Hard and Create Symbolic Links)

You may get a logical view of a file by linking to the file. A link gives you the perception of creating a new file but you are only giving the file a second name. You are still bound by the permissions of the original file. When you remove your link (with the *rm* command) you are only breaking your link, not deleting the file. Only when the *rm* command is executed with no additional links to the file is the file actually destroyed.

There are two kinds of links; hard and symbolic.

Hard links are additional directory entries pointing to an existing inode. There is no difference between the new link and the file linked to. If the original file is deleted, the directory entry created by the hard link is still active and the file is still accessible. The file contents are inaccessible only when the last link to the file's inode is deleted.

With a hard link, if the file linked to is deleted and then recreated, the hard link is still pointing to the old version of the file. This may cause a problem. (The /etc/passwd file, for example, often gets deleted, then recreated.)

Symbolic links are directory entries that point to a file name instead of an existing inode number. A symbolic link can point to a file name that does not even exist. When a symbolically linked-to file is deleted, then recreated, the symbolic link points to the new file, which is probably what you want.

Note: Since each file system maintains its own inode table, you cannot create a hard link across file systems. You can, however, create a symbolic link across file systems. Check the *man* pages for different option settings.

Examples:

```
ln   /etc/passwd   mypass          # Hard link to /etc/passwd

ln –s /etc/hosts       myhosts     # Symbolic link to /etc/hosts

rm   mypass myhosts       #       Remove both links.  No affect on
                          #     /etc/passwd or /etc/hosts
```

## Foundational Review (continued)

### Creating and Deleting Directories

### mkdir

A directory may be created with the *mkdir* command. You must have write and execute permissions in the directory you are creating the subdirectory in.

Example:

```
mkdir newdir #make new directory in current directory

mkdir /home/kip/newdir2    #make another directory.
```

### rmdir

A directory may be removed with the *rmdir* command. You must have write and execute permissions in the directory you are removing, the directory must be empty, and no one must be currently attached to that directory.

Example:

```
rmdir newdir2          #delete a directory in current directory

rmdir /home/kip/newdir       #remove another directory.
```

Lab 4

Create a link from the /etc/passwd file to a file in your current directory named “p”. Should the link be a hard or symbolic link?

Create a symbolic link from the /etc/bigfoot file to a file in your current directory named “bf”. Type out the contents of the bf file. Does the bf link really exist? Does /etc/bigfoot exist?

## Foundational Review (continued)

### Finding Files

find

Files may be located within the file system by using the *find* command. We must specify a starting point in the file system, one or more pieces of criteria which must be met, and an action to take once we find a file fitting the specified criteria.

The only argument required to find is one or more starting points in the directory heirarchy. After that, the criteria arguments must be in the form of –criteria one_value –criteria one_value –criteria one_value.

The criteria to identify the file may be one of the following (among others):

| | |
|---|---|
| file name | -name (file's name) |
| file owner | -user (user id or name) |
| size | -size (in blocks) |
| type | -type (f for file, d for directory) |
| permissions | -perm (octal permissions. See *chmod*.) |
| inode number | -inum (inode number) |
| last modification time | -mtime (in days) |
| last accessed time | -atime (in days) |
| last status change time | -ctime (in days) |

See the man pages for more options.

Once a file is located matching the specified criteria, we may:

| | |
|---|---|
| print the file name | -print |
| execute a command | -exec x {} \; |
| execute a command if we ok it | -ok x {} \; |

Note: With the –exec and –ok options, x represents the command to execute, {} is replaced with the file name found, and the protected semi-colon is required for shell interpretation purposes (there must be a space before the \;).

## Foundational Review (continued)

### Finding Files (continued)

Examples:

#find all files, starting at the root ( / ) directory, with a name of #core. Once you've found it, print its fully qualified name.

```
find / -name core -print
```

#find all files, starting at my current ( . ) directory, with a name #beginning with xyz. Once you've found it, list it in long format.

```
find . –name "xyz*" -exec ls -l {} \;
```

#find all files, starting at the /usr directory, which belong to user #lyle, and have a last modified date of greater than 5 days. Print its #name.

```
find /usr -user lyle –mtime +5 -print
```

#find all files belonging to kip or napoleon in the full file system. #Once you've found them, delete them if I give you the OK.

```
find / -user kip -o -user napoleon -ok rm {} \;
```

Note: When specifying wildcards as arguments to your criteria, protect the wildcard with a quoting symbol. If you don't, you may get wrong results (or, maybe the correct results…or, maybe, a syntax error. )

Note: Start as low in the directory hierarchy as is practical to limit the number of files searched. As an example, if you are looking for hard links, start on the mount point where the physical file system where the file resides (as hard links cannot span file systems.)

Note: When specifying multiple criteria, AND is assumed. Or must be specified with the –o conjunction.

Note: You can negate criteria by placing a ! before the specified criteria.

## Foundational Review (continued)

### Directory shortcuts

There are, however, several shorthand methods of referencing directories. In particular:

. represents the current directory

.. represents the parent directory

To use your current directory as an argument:

cp /tmp/temp.file .

In order to move up one level:

cd ..

Just keep in mind that . and .. change as you change directories.

### Absolute vs. Relative Path Names

We can always refer to a file or directory by its absolute path name. For example, our favorite directory may be /usr/lib/terminfo. So, we can always get to this directory by:

cd /usr/lib/terminfo (Use absolute path name)

However, if you were currently attached to /usr you could more easily use relative path naming:

cd lib/terminfo (Use relative path name)

Rule: Whenever you refer to a file name beginning with a /, you must supply the full path name of the file. Otherwise, UNIX considers that the file you are referring to is relative to our current directory.

## Foundational Review (continued)

### File System Command Summary

| | |
|---|---|
| chmod | Change the access permissions of files |
| cp | Make copies of files |
| df | Summarize information on disk free space |
| find | Find files, somewhere in the file system, meeting some criteria |
| ln | Create a hard or symbolic link between files |
| mkdir | Create new directories |
| mv | Move or rename files |
| rm | Remove files |
| rmdir | Remove a directories |
| umask | Set initial file creation permissions mask |

### Lab 5

In your .profile, set your *umask* value to 027. At your command prompt, set your *umask* value to 027.

Find all files in /etc and below whose name begins with *mo* and is a normal type file. Write these file names into a file called *mofiles.*

Foundational Review (continued)

Lab 6

Find all files, starting at the */usr* directory, which are owned by root and are greater than 500 blocks in size. Disregard all errors. Eliminate any records that contain the pattern “mail”. Send your output to file *nomail.fil.*

Do the same again but accept only those that do contain the pattern mail. Send your output to file *onlymail.fil.*

## Foundational Review (continued)

### Overview

An understanding of your current environment is crucial to effectively operating at the shell level. In this section we will explore the current environment and how we can modify it.

### /etc/profile and .bash_profile

Certain modifications to your environment will set by the system administrator and will be kept in the */etc/profile* file. These commands and settings are invoked immediately at login time. You, as a casual user, will not have the ability to modify this file.

Any settings or commands that you would like invoked after the system profile executes should be kept in your *.bash_profile* file in your home directory. *.bash_profile* is a hidden file similar to an MS-DOS AUTOEXEC.BAT file. In many cases, you may want to reset default settings given to you in */etc/profile*. Put these changes in your *.bash_profile*.

After you login, keep in mind that *.bash_profile* settings apply to your login shell's environment. As you move to sub-shells, *.bash_profile* will not automatically reinvoke itself. Therefore, in a sub-shell, your environment will not necessarily be the same as your parent shell's environment. You, with a few adjustments to your login shell's environment, can replicate these settings to sub-shells.

## Foundational Review (continued)

### System Environment Variables

| | | |
|---|---|---|
| HISTFILE | - | The file containing your command history. |
| HOME | - | Your home directory. |
| IFS | - | Internal Field Separator. |
| LOGNAME | - | Your login user name. |
| LPDEST | - | Override of system default printer. |
| MAIL | - | Your mailbox file. |
| MAILCHECK | - | How often between new mail checks. |
| PAGER | - | Command for navigating thru the man pages. |
| PATH | - | Your command search path. |
| PS1 | - | Primary prompt value. |
| PS2 | - | Secondary prompt value. |
| PWD | - | This shell's current directory. |
| SHELL | - | Used by some applications to determine the type of shell to spawn to (vi, mail, etc.) |
| SHLVL | - | How many levels deep your shell is. |
| TERM | - | Type of terminal or emulation. |

## Foundational Review (continued)

Keep in mind, some shell variables have a direct influence on the shell's behavior, e.g. the ones listed above. Some shell variables are for your use and you can use them to store whatever values you like, e.g. X, Y, Z.

Remember, an *export*ed variable has a copy sent down to sub-shells. Un*export*ed variables do not. Therefore, if a variable has been *export*ed and the variable has a special effect on your shell, that effect will be redefined down at the sub-shell level. Also, the *export*ed status flows down to subsequent sub-shells.

### readonly

You may execute the *readonly* command against any variable. For the life of that shell, the variable may not be modified.

Example:

```
readonly   LOGNAME
```

### Shell options

```
set –o                  # Show my current shell option settings, for this
                        # environment

set –o option           # Turn on this option, in this environment

set +o option           # Turn off this option, in this environment
```

Example:

```
set –o noclobber        # Disallow redirection clobbering at this shell

set +o braceexpand      # Turn off curly brace expansion


shopt                   # show more bash specific options

shopt –s xpg_echo       #  Turn on echo escaped-character expansion

shopt –u huponexit      #  Disable hangup on exit
```

## Foundational Review (continued)

### aliases

| | |
|---|---|
| alias | show all my alias', in this environment |
| alias x="y –z" | create a new alias, in this environment |
| unalias x | delete alias x, in this environment |

### functions

*function*s are similar to *alias*es in two respects. First, they live in the shell's environment where they were defined. Second, they define commands to execute.

*function*s differ in two respects to *alias*es. Whereas *alias*es usually have a one-to-one correspondence to a command and its new name, *function*s usually represent a series of commands. Also, *function*s have a lower precedence than *alias*es (*alias*es are located before keywords, *function*s are located after keywords.)

To see a list of this shell's functions, the command is:

| | |
|---|---|
| functions | # (an alias for typeset –f.) |

## Foundational Review (continued)

**function Example:**

```
function info {

        echo The current date is $(date)
        echo Your current process id is $$
        echo You are logged in as $LOGNAME
}

or

 info() {

        echo The current date is $(date)
        echo Your current process id is $$
        echo You are logged in as $LOGNAME
}
```

### Lab 7

Inside of your *.bashrc* file put an echo “Hello from .bashrc”.
Also, define a function *myinfo* that will display who you are, show the current date and time, and show your home directory.
Put your *psu* alias at the end of *.bashrc*.

Spawn a bash. Ensure that your psu alias works. Ensure that you received the hello message. Log out and then back in. Ensure that you received the hello message and your psu alias works.

Foundational Review (continued)

# Communicating With Processes and Users

In this section we will look at communicating with running process', logged in users, and user accounts.

## Communicating with processes

Generally, when we run a program a child process is *fork*ed and the program is initiated in that process. This new process has its own process id and environment. In this section, we will look at several methods of invoking programs, and communicating with these new processes.

### exec

You may use the *exec* command to replace your currently running program with a new program running in the same process. You may want to do this, for example, when changing shells (from a Bourne to a Korn, for example):

ksh

will spawn a child Korn shell with a new process id and environment.

exec ksh

will replace the existing shell with a Korn shell. No new process is spawned.

### &

To submit a job in background, append an & to the end of your command. Keep in mind, that this new background process is still a child, and the termination of the parent will terminate all children.

## Foundational Review (continued)

### fg

Used to bring a background job into the foreground.

### bg

Used to send a foreground job to the background. You will have to pause the current foreground job with a <ctrl> z.

### jobs

Used to show all processes in a background or suspended state. The entry with the + is the current job. The entry with a – is the next most current job.

### nohup

Submit a command, which is not dependent on its parent's survival for its survival. Used to submit jobs remotely or jobs which may run for a long time. Will allow logging off without the job aborting.

```
nohup   bigsort   &   (Now logoff, and go home)
```

### nice

Submit a job at lesser priority than normal. (By default, 10 units less, whatever a unit is.)

```
nice   bigsort   &
```

Or to *nohup* and *nice* the job, at the same time:

```
nohup   nice   bigsort   &
```

## Foundational Review (continued)

### . (dot)

Submit a command, which will run as part of the current process. Used when, for example, you want to set environment variables. If you submit a job to set environment variables without the ., a sub-shell will be spawned and the variables will be set in the sub-shell. When the program terminates, the sub-shell will be destroyed, along with the set variables. (In the following example, setvars is a fictitious shell script which sets variables.)

```
setvars       (Variables will be short lived.)

. setvars     (Variables will be accessible after job completes.)
```

## Foundational Review (continued)

### wait

Wait for process termination. Use this command when you want to ensure that a particular job is finished before you start a subsequent job.

```
sort   bigfile   >   bigfile.out   &

wait              # Ensure sort is finished before printing.

lp   bigfile.out
```

### kill

There are times when a job needs to be terminated prematurely. Possibly the job was submitted by mistake, it's hanging up a terminal, or it's performing faulty logic. You may use the *kill* command to accomplish this:

```
bigsort   &

        [  12345  -  bigsort running  ]

kill   12345    #  send the default signal 15 to terminate the process

kill    %1      #  kill job number 1 (in your job's list)
```

If it fails to terminate, you might try:

```
kill   -9   12345
```

-9 indicates a sure kill. Should work most of the time.

You can also use the mnemonic for the option, as in:

```
kill  –TERM  %3
```

## Foundational Review (continued)

Actually, kill just sends a signal to a process. You may want to do something other than kill the process. To list out all of the available signals, use:

kill -l (ell)

### trap

For this shell, you might want to modify the behavior of a signal. trap indicates what action to take whenever this process receives a signal (INTR, STOP, CONT, TERM, etc.) All signals can be trapped except for signal 9 (KILL).

Example:

trap “cleanup” 15

This notation says that when this process receives signal 15 (TERM), instead of terminating, execute the command *cleanup* (*cleanup* may be a UNIX utility, an alias, a function, etc.) Now, whenever we receive a signal 15, instead of terminating cleanup will execute that might make an entry in a log file, route an e-mail message to the owner of the process, and then terminate.

Example:

trap “” 15

Whenever we receive signal 15, do nothing. Essentially, for this process, we have said signal 15 is dead.

Example:

trap 15

Put signal 15 back to its default behavior.

## Foundational Review (continued)

### Communicating with users

You may send a message directly to another user's terminal. You may send it selectively to a particular user or to all currently logged in users.

### write

Write a message to a particular logged-in user. If the user is not logged in, *write* will not be invoked. To terminate your message, type <ctrl> d on a line by itself:

```
write   ecartman

        Eric.  Call me on my extension (234) when you get time.
        Stan and Kyle have a project for us.
        Thanks.

        Butters.
        <ctrl> d
```

## Foundational Review (continued)

### mesg

To keep messages from obliterating your screen, you might want to refuse *write* messages. You may issue the following to refuse all messages from all users except root:

```
mesg   n
```

At a later time, to start accepting messages again:

```
mesg   y
```

### wall

There are times when a message should be broadcast to all users, such as “The system is going down in 3 seconds – Please log off now”. You might want to write to all users:

```
wall

The system will be unavailable from noon to 1:00 PM today.
Please ensure your work is saved and your session is terminated
by noon.   Admin.
<ctrl> d
```

## Foundational Review (continued)

### mail

A widely used communications facility is the UNIX *mail* facility. *mail* is used not only for sending messages to logged-on and logged-off users but also for sending mail to oneself. (For example, at job completion, the status of the job.)

To read your mail:

```
mail
```

To send mail:

```
mail   smarsh

Hey Stan.  Tell Butters to hold off on the project.  Mr.
Mackey doesn't think it's such a good idea.
<ctrl> d.
```

If you're attached to the internet:

```
mail   chef@southpark.com
```

If your mail message is in a file:

```
mail   wendy   <   my.message

         or

cat   my.message   |   mail  wendy
```

Foundational Review (continued)

# File Processing

When working with UNIX, we often have the need to work with files; looking at them, filtering them, modifying them, interrogating them, creating them, etc. UNIX has a vast array of tools and techniques for managing your files.

## Standard Devices

UNIX has 3 standard logical devices for fetching and displaying information. They are:

| | | | |
|---|---|---|---|
| Device 0 | - | Standard Input | (Your terminal keyboard) |
| Device 1 | - | Standard Output | (Your terminal screen) |
| Device 2 | - | Standard Error | (Your terminal screen) |

By default, UNIX commands expect their input to come from device 0, the keyboard. By default, UNIX commands send their output to device 1, the terminal screen. By default, UNIX commands send their error output to device 2, the terminal screen.

UNIX supplies a mechanism for modifying your 3 devices and they are:

| | |
|---|---|
| < | Change device 0 |
| > | Change device 1 |
| >> | Change and append device1 |
| 2> | Change device 2 |
| 2>> | Change and append device 2 |

## Foundational Review (continued)

### A Few Basic Commands

#### cat

Use the *cat* command to type out or con*cat*enate the contents of a file or files.

Example: cat .profile

#### more

Use the *more* command to type out the contents of a file or files a page at a time. Space bar will present the next page of output.

Example: more /etc/profile

#### less

Use the *less* command to type out the contents of a file or files a page at a time. May go backwards as well as forwards. Enter key will present the next page of output. (less may not be available, depending on your variant of UNIX.)

Example: pg /etc/profile

## Foundational Review (continued)

### head

Type out the first x (10 by default) lines of a file or input stream.

Example:

| | |
|---|---|
| head /etc/passwd | (First 10 lines of /etc/passwd) |
| head -5 /etc/passwd | (First 5 lines of /etc/passwd) |

### tail

Type out the last x (10 by default) lines of a file or input stream.

Example:

| | |
|---|---|
| tail /etc/passwd | (Last 10 lines of /etc/passwd) |
| tail -n 5 /etc/passwd | (Last 5 lines of /etc/passwd) |
| tail -n +5 /etc/passwd | (Starting on line 5, type the rest of /etc/passwd.) |

tail –f log.file (Type any new entries to the log.file)

### file

Determine, as best UNIX can, what type of file are we dealing with. Useful to prevent typing the contents of binary files:

Example: file /etc/passwd
/etc/passwd: ASCII text (Output)

## Foundational Review (continued)

### strings

Find ASCII strings within binary files. Useful for getting information on executable files before execution:

```
strings   /bin/ping
```

### Redirection Examples

Example:

```
echo     Hello, World   >   hello
```

#The echo command normally sends its output to the screen, device 1. In the preceding example, we've created a new disk file called *hello* which contains the text "Hello, World".

Note: If a file called *hello* was in existence, it would have been wiped out and recreated with the new information.

Example:

```
ls   >   my_ls_list

ls   aaabbb   2>   my_errors
```

In the first *ls* example, the output from the *ls* command is redirected into a file called *my_ls_list*.

In the second *ls* example, the error output is redirected into a file called *my_errors*. (Unless you actually did have a file named *aaabbb*.)

## Foundational Review (continued)

Example: cat file1 file2 > output.fil

A new file is created which contains the contents of *file1* followed by *file2*.

Example: cat file3 file4 >> output.fil

Add *file3* and *file4* to the end of *output.fil*

### Lab 8

Echo a message into your mailbox. Redirect your entry from the output of the *who* command into your mailbox. Redirect the entire contents of your *.profile* into your mailbox. Redirect the last line in the output of the *df* command into your mailbox. Check your mail. Delete your mail messages.

### Lab 9

Identify the 4th line of output from the *ps –e* command. Write a construct that will print that line plus the next 5 lines of output.

### Lab 10

Run, in background, a sleep command for 1,000 seconds. Using the *kill* command, put the process in a suspended state. Ensure that the command is, indeed, stopped. Then, using the *kill* command, put the process back in a run state.

## Foundational Review (continued)

## Regular Expressions

“Regular expression” is the term used for describing text patterns. In addition to standard ASCII characters, regular expressions may contain wildcard type metacharacters, which are NOT to be confused with shell metacharacters.

Regular expressions are used with many of the pattern matching tools available in UNIX, such as *grep*, *sed*, *vi*, and *awk*.

### Regular Expression Metacharacters

| | | |
|---|---|---|
| . | - | Represents any one character. |
| ^ | - | Represents the beginning of a line. |
| $ | - | Represents the ending of a line. |
| [xyz] | - | Represents one of the characters in the list. |
| * | - | Represents zero or more occurrences of the preceding character. |
| \ | - | Protect the next character from metacharacter interpretation. |

## Foundational Review (continued)

### Sample *vi* Session, Using Regular Expressions

We know we can search forward for patterns in *vi* with the / operation. What follows the / is actually a regular expression. To this point we have only searched for static text. In this section we will use some of the regular expression metacharacters to expand the power of *vi*.

Assuming you are in a *vi* session, consider the following searches:

| | |
|---|---|
| /^THE/ | Search for “THE” at the beginning of a line. |
| /.$/ | Search for any character at the end of a line. |
| /\.$/ | Search for a period at the end of a line. |
| /^$/ | Search for an empty line. |
| /^THE.*\.$/ | Search for a line with “THE” at the beginning and a period at the end. (Anything in the middle.) |
| /^THE…END/ | Search for a line with “THE” at the beginning, followed by any 3 characters, followed by “END”. |

## Foundational Review (continued)

### The grep family (grep, fgrep, egrep)

The *grep* family of commands is used to locate patterns (regular expressions) in files or input streams.

*grep* is the most commonly used and evaluates the standard set of regular expression metacharacters.

*fgrep* (fixed grep) treats metacharacters as static text.

*egrep* (extended grep) allows more sophisticated pattern matching, such as multiple pattern searching and extended regular expressions.

Revisiting the /etc/passwd file

The */etc/passwd* file contains information for all users of the particular UNIX system. The file consists of seven fields, each field delimited by a colon ( : ). The fields are:

Field 1 - User Name
Field 2 - Encrypted Password
Field 3 - User ID
Field 4 - Group ID
Field 5 - Comments
Field 6 - User's Home Directory
Field 7 - User's Startup Program

We will be using the */etc/passwd* file for much of our file processing. Take a quick look at it:

```
cat   /etc/passwd
```

## Foundational Review (continued)

### grep Family Examples:

Example:

grep “^m” /etc/passwd

(Return all lines from /etc/passwd which start with “m”.

fgrep “lib” /etc/passwd

(Return all lines from */etc/passwd* which contain “lib”.

egrep “root|uucp” /etc/passwd

(Return all lines from */etc/passwd* which contain “root” or “uucp”.

grep -v bin /etc/passwd

(Return all lines from */etc/passwd* that do NOT contain “bin”

grep -i train /etc/passwd

(Return all lines from */etc/passwd* that contain train, disregarding case sensitivity.)

Check the *man* pages for grep to see all of the cool options.

## Foundational Review (continued)

### Piping and Filters

Many times in UNIX we want to take the output from one command and use it as input into another command. This technique is called piping. Commands that can take both standard input and produce standard output are called filters.

Piping is accomplished by specifying a broken bar ( | ) after a command that produces standard output and before a command which accepts standard input.

Example:

cat fruit* | grep banana

Find any records in all files beginning with fruit which contain the pattern banana.

cat /etc/passwd | grep "^m.*bash$"

Functionally the same as: grep "^m.*bash$" /etc/passwd
(All records from /etc/passwd that start with "m" and end with "bash")

ps -e | grep "bash"

Show the process information on all running Bash shells.

ps -e | grep "bash" > bash.file

Same as prior example, but saving output in file *bash.file*.

ps -e | grep -iv "bash" > bash.file

Same as prior example, but only records that don't contain bash, ignoring case.

## Foundational Review (continued)

### sed

*sed* is a cousin to *grep*. *sed* looks for patterns in files or input streams, but, if it finds a pattern match, will act on that line of input. In the case where there is no pattern match, the line will be passed through to the output stream intact.

Keep in mind, that only the output stream is being modified. If the output needs to be preserved, redirect the output to a disk file.

Example:

sed ‘s/tina/deb/’ /etc/passwd

Change the first occurrence of “tina” to “deb” on each line of the */etc/passwd* file.

sed ‘s/rico/tina/g’ /etc/passwd

Change all occurrences of “rico” to “tina” on each line of the */etc/passwd* file.

sed ‘/pedro/d’ /etc/passwd

Delete all lines of /etc/passwd which contain “pedro”

sed ‘/pedro/d’ /etc/passwd | sed ‘s/:/?/g’ > final.fil.

Delete all lines of */etc/passwd* which contain “pedro”. On those remaining lines, change all colons ( : ) to question marks ( ? ). Write the output to *final.fil.*

## Foundational Review (continued)

### tr

*tr* is a cousin to *sed*. *tr* performs a one character to one character translation of an input stream. tr cannot read from files. tr can only read from STDIN, i.e., either pipe in your records or redirect in your records.

Example:

tr “:” “\011” < /etc/passwd

Change all colons ( : ) to tabs (octal 011) in */etc/passwd*.

tr “[a-z]” “[A-Z]” < /etc/passwd

Change all lowercase characters to uppercase characters in */etc/passwd.*

tr -s “-“ < my.stuff

Squeeze out multiple occurences of dash.

e.g, a------b-------c becomes a-b-c

tr -s “,” “:” < my.stuff

e.g, a,,,,,,b,,,,,,c becomes a:b:c

Squeeze out multiple occurrences of commas, replacing the single occurrences of commas to colons.

tr –d "?" < my.stuff

Delete all question marks from my input.

tr -c "a-zA-Z" "?" < my.stuff

Change all non-alpha characters to question marks.

## Foundational Review (continued)

### cut

*cut* extracts indicated characters or fields from an input stream. Extract by field with variable length field records with a common delimiting character. Extract by character position with fixed length fields.

When you cut by field, you must tell cut what the field delimiter is.

Examples:

cut -f 1 –d : /etc/passwd

Extract field 1 (The user name) from */etc/passwd.* The fields are delimited with a colon ( : )

cut -f 1,3 -d : /etc/passwd

Same as above, but fields 1 and 3.

cut -c 1-5 /etc/passwd

Extract columns 1 through 5 from */etc/passwd.*

ls -l | cut -c 16-24

Show only the owner name of all files in directory.

## Foundational Review (continued)

### sort

*sort* takes an input stream, file, or group of files and sorts it.

Examples:

sort file.in

Sort *file.in*, starting at column 1, ascending

sort file.in > file.out

Same as prior, but write output to *file.out,* not screen.

sort -o file.out file.in

Better method, same functionality of prior.

sort -t : +2 -n /etc/passwd

Sort the */etc/passwd* file, third field (+2), numerically (-n), with the delimiter between fields of : ( -t : )

sort -t : +2 -n -r /etc/passwd

Same as the last, except descending sequence ( -r )

## Foundational Review (continued)

### uniq

Remove unique, adjacent lines from file(s) or input stream.

Examples:

cut -f 4 -d ":" /etc/passwd | sort -n | uniq

Show a list of used group ids.

### Special Logical Devices

In UNIX, there are a couple of special device names which you will need to use, on occasion:

| | | |
|---|---|---|
| /dev/tty | - | Always refers to your terminal. |
| /dev/null | - | Bit bucket, throw-away area |

Examples:

find / -name core -print 2> /dev/null

Throw away those error messages that we expect from *find*.

Foundational Review (continued)

Lab 11

Create a list of sorted list of users currently logged on. Ensure there are no duplicates. Make sure your username is excluded.

Lab 12

What is the highest userid (numeric) currently in use on this UNIX box?
What is the lowest? What are the highest and lowest of those users with a home directory immediately beneath /home?

Lab 13

Is the cron daemon currently running? How about the xinet daemon? Generate a two record list of these processes and their process id.

Lab 14

Link to the instructor's script file named *adduser*. Execute the script file and answer the questions with real, or imaginary, information. This will add a record to the instructor's data file named *user.list.* You can run the command as many times as you like, with an additional record being added each time.

Foundational Review (continued)

Lab 15

Link to the instructor's file named *user.list*. From that list extract out your most current record, based on your user-id. (Use either $LOGNAME or `whoami` to identify who you are.) Your most current record will be the record showing up the latest in the file. Then from that record, cut out just your first name. After you have finished the construct, the instructor will guide you through putting your construct in a script file named *firstname*.

## Foundational Review (continued)

## Miscellaneous Commands

This section will explore some other UNIX commands. These commands don't fit well into any category but are important and quite often used.

**cal** Display a calendar for a month or year. Examples:

```
cal    4    1996

cal    1997
```

**clear** Clear your screen. Example:

```
clear
```

**date** Display the system date in different formats. Examples:

```
date                #current date and time
date   +%D          #current date
date   +%T          #current time
date   +%y          #current year (2 digit)
date   +%Y          #current year (4 digit)
date    +%m         #current month
date   +%d          #current day of month
date   +%w          #current numeric day of week
date   +"%d %m %Y"
                    #British format.  Use quotes when including
                    spaces in your format.
```

(See other format specifiers in the *man* pages.)

**df** Display free disk space statistics.

```
df                  #show for all file systems
df   /dev/hd1       #show for a particular file system
```

## Foundational Review (continued)

**expr** Perform integer mathematics, comparative logic, string operations, and regex processing:

```
expr   5   +   10                  #Addition - Prints 15
expr   20   -   7                  #Subtraction - Prints 13
expr   40   /   3                  #Division – Prints 13
expr   33   \*   4                 #Multiplication – Prints 132
expr   40  %  3                    #Modulus – Prints 1

expr  10 \< 20                     #True – Prints 1
expr 20 \< 10                      #False – Prints 0

expr match Dynamite 'D.*m'         # Prints 5, number of characters
                                   # matched

expr substr Napoleon 3 4           # Prints pole

expr index Napoleon pole           # Prints 3

expr length Napoleon               # Prints 8
```

Example:

```
X=50
Y=30
expr   $X   +   $Y                 # Prints 80
Z=` expr   $X   -   $Y`
echo   $Z                          # Prints 20
```

**fold** Wrap lines into a specified width.

```
fold   -w 5   /etc/passwd      # Print passwd 5 characters per line.
```

## Foundational Review (continued)

**split** Split a file into multiple pieces.

```
split  -5  /etc/passwd  out.   # Creates as many files
                               # as needed, at 5 lines per file.
                               #  Files will be named out.aa,
                               # out.ab, out.ac, etc.
```

**su** Invoke a shell with a substitute user and group ID

```
su  kenny          #must know kenny's password.
su                 #must know root's password.
```

**tee** Split output into a file and standard output:

```
cat  /etc/passwd  |  tee  my.password.file
                           # shows passwd on screen and creates
                           # copy of file /etc/passwd.
```

**time** Measures system time requirements for executing a command. This is a decent tool for testing program performance.

```
time  ps  -el          #After ps command, stats are printed.
```

**wc** Counts number of characters, words, and / or lines in a file or input stream.

```
wc   *          #Show c, w, and l for all files

wc   -l   *     #Show line count only for all files

wc  -c  *       #Show character count only for all files.

wc   -w   *     #Show word count only for all files.
```

## Foundational Review (continued)

| | |
|---|---|
| who | Who is currently logged in?<br><br>who |
| who am i | What is my user-id, terminal-id and login-time?:<br>who am i |
| whoami | What is my current identity?<br><br>whoami |
| xargs | Feed standard input to command item(s) one at a time:<br><br>cat files_to_compile \| xargs cc<br><br>cat filelist \| xargs ls -l |

Foundational Review (continued)

Lab 16

Write the command to show how many users are currently logged on. The output should look like:

There are currently *xx* users logged on.

xx should be replaced by the number of users currently on. (Think about the *who* and *wc* commands.)

Lab 17

Write the command to produce the current day of week, followed by the current date, followed by the user's name. The output should look like:

The current date is Thursday, 05/29/87, Mark

Think about command substitution, different formats for the *date* command, and the *echo* command.

Lab 18

How many currently running processes does root own? Display it like this:

Root owns 122 processes.

Foundational Review (continued)

Lab 19

Without using a text editor, I want to insert a new fruit in my fruit file. I want a pomegranate as the 3rd fruit. How can I do this? It may take a couple of steps.

## Q



## R



## S



## T



## U



## V



## W



## X