```
[root@dev1 ~]# ps
  PID TTY          TIME CMD
15844 pts/0    00:00:00 bash
15868 pts/0    00:00:00 ps
[root@dev1 ~]# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             9.4G  7.1G  1.8G  80% /
none                  129M     0  129M   0% /dev/shm
[root@dev1 ~]# who
root     pts/0        Nov 28 11:13 (cpe-173-172-107-98.austin.res.rr.com)
[root@dev1 ~]# ls /etc
DIR_COLORS         dev.d          inputrc          makedev.d        php.d          rc4.d          shells
DIR_COLORS.xterm   environment    iproute2         man.config       php.ini        rc5.d          skel
X11                exports        issue            mime.types       pki            rc6.d          smart
adjtime            filesystems    issue.net        mke2fs.conf      pm             redhat-release ssh
aliases            fstab          krb5.conf        modprobe.conf    postfix        resolv.conf    subversion
aliases.db         gpm-root.conf  krb5.conf.rpmnew modprobe.d       ppp            rpc            sudoers
alternatives       group          ld.so.cache      motd             prelink.conf   rpm            sysconfig
apt                group-         ld.so.conf       mtab             printcap       rwtab          sysctl.conf
bashrc             gshadow        ld.so.conf.d     multipath.conf   profile        rwtab.d        syslog.conf
blkid              gshadow-       ld.so.conf.rpmnew my.cnf          profile.d      sasl2          termcap
cron.d             hal            libaudit.conf    netplug          protocols      screenrc       udev
cron.daily         host.conf      libuser.conf     netplug.d        rc             scsi_id.config updatedb.conf
cron.deny          hosts          localtime        nsswitch.conf    rc.d           securetty      vimrc
cron.weekly        hosts.allow    localtime.rpmnew nsswitch.conf.rpmnew rc.local    security       virc
csh.cshrc          hosts.deny     login.defs       openldap         rc.sysinit     selinux        wgetrc
csh.login          httpd          logrotate.d      opt              rc0.d          services       xinetd.d
dbus-1             init.d         lvm              pam.d            rc1.d          sestatus.conf  yum
default            initlog.conf   mail.rc          passwd           rc2.d          shadow         yum.conf
depmod.d           inittab        mailcap          passwd-          rc3.d          shadow-        yum.repos.d
[root@dev1 ~]#
```

# SHELLS

*Yeah, the hard part of Linux*

# THE BIG LOOP

- In order to master the shell, you have to understand it's inner workings
- The first concept is *The Big Loop*

# THE BIG LOOP

1. Print prompt, await user input
2. Parse and verify input; on error, loop
3. Perform requested operation ( execute command, built-in )
4. Loop

# MORE ON STEP 2

- Step 2: parse and verify input
- Very important step, includes:
- Syntax checking, command identification, metacharacter substitutions and operations

# SYNTAX

- `<command> [options] [arguments]`
- Everything is separated with white space
- Options are just a special interpretation of arguments, generally identified with a prefixed hyphen
- POSIX options ( or long options ) use a double hyphen prefix, and often spell out the option with a word rather than just a letter ( --verbose instead of -v )

# QUOTING

- Generally, arguments are separated with whitespace, but sometimes whitespace needs to be part of the argument itself ( spaces in filenames, for example ). Consider:
  - `command filename with spaces`
  - Without any guidance, the shell will interpret this input as a command with 3 arguments.
- Quoting is the easiest way to guide the shell in this matter. There are two forms...

# SINGLE QUOTES

- Single quotes are the simplest to use:
  - `command ‘filename with spaces’`
- The quotes let the shell know where an argument starts and stops ( quotes not included ), and it doesn’t bother with what’s between the markers - it is interpreted strictly as data
- Hence, this line would be interpreted as a command with one argument, `filename with spaces`

# DOUBLE QUOTES

- Double quotes follow single syntax, but interpret differently:
  - `command “filename with spaces”`
- The quotes let the shell know where an argument starts and stops, but the data in between is loosely examined for *metacharacters*. More on that in a minute.
- So, this line would also be interpreted as a command with one argument, `filename with spaces`

# METACHARACTERS

- A metacharacter is any character that has more than one meaning or interpretation.

- For example, you just learned about two of them: the single and double quotes.  In normal context, they denote endpoints for arguments, not *actual* quote characters

- But what if you need a quote in your argument value, say a filename with a single quote like: `smith's`

# ESCAPING

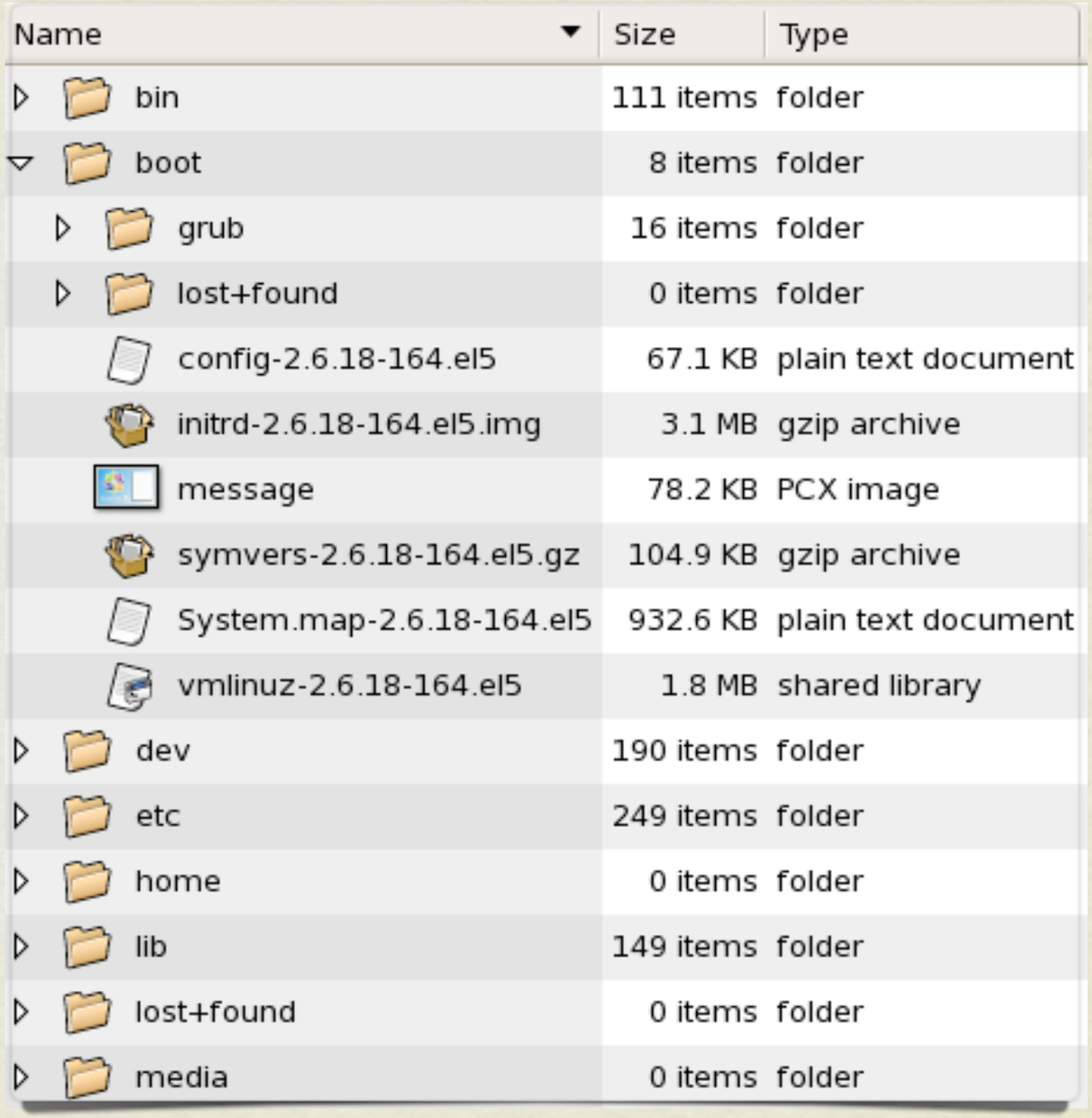
- The quick and simple way to do that is with the escape metacharacter, the backslash: \
  - `command smith\'s`
- The escape character tells the shell to interpret the character following the backslash as a normal character, rather than a metacharacter
- This allows you to use metacharacters as regular characters

# BASIC COMMANDS

- `who`: Lists currently logged in users
- `uptime`: Statistics about machine usage and run time
- `echo`: Prints the given arguments to the screen
- `date`: Print current date and time
- `exit`: Terminate current shell session
- `reset`: Reset terminal state to default settings

# HIERARCHIES

- Data is stored in files
- Files are grouped and organized in Directories, creating a tree structure
- The filesystem begins at root, represented as: /
- The Standard Hierarchy provides basic organization



# WORKING DIRECTORY

- Operations within the shell generally gather input from files and output information to files, so the shell tracks a "working directory" to ease the file specifications, and have a default location to output files if one is not provided

- `pwd`: Print Working Directory

- `cd`: Change [working] Directory

# PATHNAMES

- A pathname specifies the exact location of a file or directory within the filesystem.
- Understanding pathnames is critical to a happy shell life
- There are two types of pathnames: absolute and relative

# ABSOLUTE PATHNAME

- An absolute pathname uses the root of the filesystem to fix the starting location for the path search.
  - `/etc/passwd`
- Starting from `/`, descend into the `etc` folder, then locate the file named `passwd`
- The key is the leading slash - exactly fixing the starting point

# RELATIVE PATHNAME

- Relative pathnames only specify a file's location with respect to a working directory. The path is *relative* to the current working directory. Relative pathnames never start with a /.
  - `memos/january.txt`
- From within the current directory ( see? the starting point is the current directory - not always / like for absolute ), descend into the `memos` folder and locate the file `january.txt`

# COMPARISON

Absolute Pathnames

- *Always start with a /*
- Search starts from /
- Always refers to exactly one file

Relative Pathnames

- *Never start with a /*
- Search starts from CWD
- Can refer to any number of files ( dependent on CWD )

# BASIC COMMANDS

- `mkdir`: Create a new directory
- `touch`: Update modification and access times of given file
- `spell`: Spell check given file ( or input on stdin )
- `mv`: Move a file from one location to another ( rename )
- `cp`: Copy a file to another location
- `rm`: Remove ( delete ) a file
- `ls`: Display listing ( contents ) of a directory

# WILDCARDS

- Wildcards are another set of metacharacters which provide a shorthand notation for specifying large groups of files
- There are 3 basic pathname wildcards:
  - *
  - ?
  - [set]

# WILDCARD: *

- The * wildcard is the easiest to understand, and most common
- Definition: Match 0 or more characters.  Any characters.
- Examples:
  - `*`
  - `a*`
  - `*.txt`

# WILDCARD: ?

- The ? wildcard comes in handy now and again
- Definition: Matches exactly 1 character.  Can be any character, but there must be exactly 1.
- Examples:
  - `file?.txt`
  - `log-????`
  - `????*`

# WILDCARD: [SET]

- The bracketed set wildcard can be very useful when filenames are following a specific pattern
- Definition: Match exactly 1 character, character must be from the set. Great flexibility in specifying the set
- Examples:
  - `log-2009-1[012]-*`
  - `[a-zA-Z]*`

# WILDCARD: [SET]

- Each desired character can be directly typed into the set:
  - `[012345]`
- Ranges are acceptable. Starting point must be “less” than ending point. Starting/ending case must match for letters:
  - `[0-5]`
  - `[d-h]`
  - `[N-Z]`

# WILDCARD: [SET]

- Mix and match:
  - `[0-9a-zA-Z]`
  - `[c-fikmp]`
- If a hyphen is needed to be part of the set, specify it first:
  - `[-acg0-4]`

# WILDCARD: [SET]

- You can also specify an “anti” set.  Anything listed in the set will **not** match.  Simply start set with !
  - `[!0-9]`
- If an exclamation mark is needed in a set, specify it anywhere after the first character:
  - `[0-9!bkg-i]`

# ENVIRONMENTS

- Every piece of running software (a process - more on that later) has it's own environment
- The environment is simply a collection of key->value pairs
- The key is [traditionally capitalized] letters, numbers and symbols to uniquely identify the variable
- The value is a string

# ENVIRONMENTS

- Examples:
  - `PATH=/usr/local/bin:/usr/bin:/bin:/sbin`
  - `HOME=/home/bob`
  - `TOTAL=348`

# ENVIRONMENTS

- To create a new variable ( or change an existing one ):
  - `TOTAL=100`
- You type the name of the variable, an equals sign, and the value.  Don't forget about quoting if needed!

# ENVIRONMENTS

- Once a variable is created, you can view it's value with the $ metacharacter. The easiest way is to use `echo`:
  - `echo $TOTAL`
- The $ metacharacter asks the shell to look up the value for the named variable, and replace everything with that value.
- So after parsing, the above command becomes:
  - `echo 100`

# ENVIRONMENTS

- Environment variables are local to the containing process, but you can mark variables as “exported”, which allows them to be passed down to subprocesses ( child processes )

- Once a variable is created, to mark it exported:
  - `export TOTAL`
- Note the **lack** of the $ metacharacter!
- To stop exporting: `export -n TOTAL`

# ENVIRONMENTS

- `set`: Displays all environment variables and values
- `env`: Displays exported environment variables and values
- To remove a variable completely:
  - `unset TOTAL`
- A note about the $ metacharacter: if the variable does not exist, the entire statement evaluates to the empty string

# MAN PAGES

- Man pages, short for Manual Pages, represent the online help system in the Linux environment

- Simple interface:

  - `man <command>`

  - `man <library>`

  - `man <function>`

  - `man <file>`

# MAN COMMAND

- The `man` command locates the requested manpage and formats it for display
- Manpages can be written to cover any topic, but generally are available for commands, libraries, function calls, kernel modules and configuration files.
- For example, to learn more about the `who` command:
  - `man who`

# MANPAGES

- Follow fairly standard format: Name, synopsis, description, examples, see also.  Additional parts include author, copyright, bugs and more.
- Manpages are organized into “sections”, grouping user commands into one section, system libraries in another, and so forth.
- The See Also section is invaluable!

# INFOPAGES

- There is some movement to convert the aging manpage system into a newer format, the infopage system.
- The info system provides a more advanced interface, supporting links, split windows and more.  Accessing infopages is the same:
  - `info <topic>`
- Once within the info system, type ? for help on the interface
- The conversion is still in it's infancy

# EXERCISES

- In your home directory, create a directory called ‘test’.
- Read the man page on man.
- List all files in your home directory that start with an ‘a’.
- Display your PATH environment variable and explain it’s purpose.

# INPUT AND OUTPUT


- This is the “normal” flow of data

# REDIRECTION

- Changing the standard flow of input and output
- Output redirection sends one or more of the output streams to files on disk
- Input redirection feeds a file from disk as the input to a process

# OUTPUT REDIRECTION

```
who > who.out
```


- Simple output redirection.  Creates/overwrites file.

# OUTPUT REDIRECTION

```
who 2> who.err
```


- Simple stderr output redirection.  Creates/overwrites file.

# OUTPUT REDIRECTION

```
who > who.out 2> who.err
```


- Combined out & err redirection.  Creates/overwrites files.
- File names must be different!

# OUTPUT REDIRECTION

```
who > who.all 2>&1
```


- Combined out & err redirection.  Creates/overwrites files.
- Only one file name, used for both output streams

# OUTPUT REDIRECTION

- All of the previous examples would create the output file if it did not exist, and if it did, would completely overwrite the existing file with the output of the command.

- Adding an extra > would turn the redirection functions into appending mode:

  - who >> who.out

  - who 2>> who.err

  - who >> who.all 2>&1

# OUTPUT REDIRECTION SUMMARY

| | |
|---|---|
| `> file` | • capture stdout to `file`<br>• overwrites<br>• > is equivalent to `1>` |
| `2> file` | • capture stderr to `file`<br>• overwrites |
| `> file 2> file2` | • capture stdout to `file`<br>• capture stderr to `file2`<br>• overwrites |

# OUTPUT REDIRECTION SUMMARY

| | |
|---|---|
| `>> file` | • capture stdout to `file`<br>• appends<br>• >> is equivalent to `1>>` |
| `2>> file` | • capture stderr to `file`<br>• appends |
| `>> file 2>> file2` | • capture stdout to `file`<br>• capture stderr to `file2`<br>• appends |

# OUTPUT REDIRECTION SUMMARY

| | |
|---|---|
| `> file 2>&1` | • capture stdout to `file`<br>• capture stderr to `file`<br>• overwrites |
| `>> file 2>&1` | • capture stdout to `file`<br>• capture stderr to `file`<br>• appends |

# INPUT REDIRECTION

```
cat < who.all
```


- Simple input redirection

# REDIRECTION

- Input redirection isn't common anymore, now that most commands can handle their own file I/O
- Input and output redirection can be combined:
  - `cat < who.all > cat.who.all`
  - `cat < who.all 2> cat.who.all.err`
  - `cat < who.all > cat.who.all.all 2>&1`

# EXERCISES

- From your home directory, use echo and output redirection to create a file in the ‘test’ folder called ‘file1’ with the contents ‘helllo’.  Use a **relative** pathname.

- Use input redirection and the spell command to spell check ‘file1’.

- Spell check ‘file1’ again, saving the output to a file using redirection.

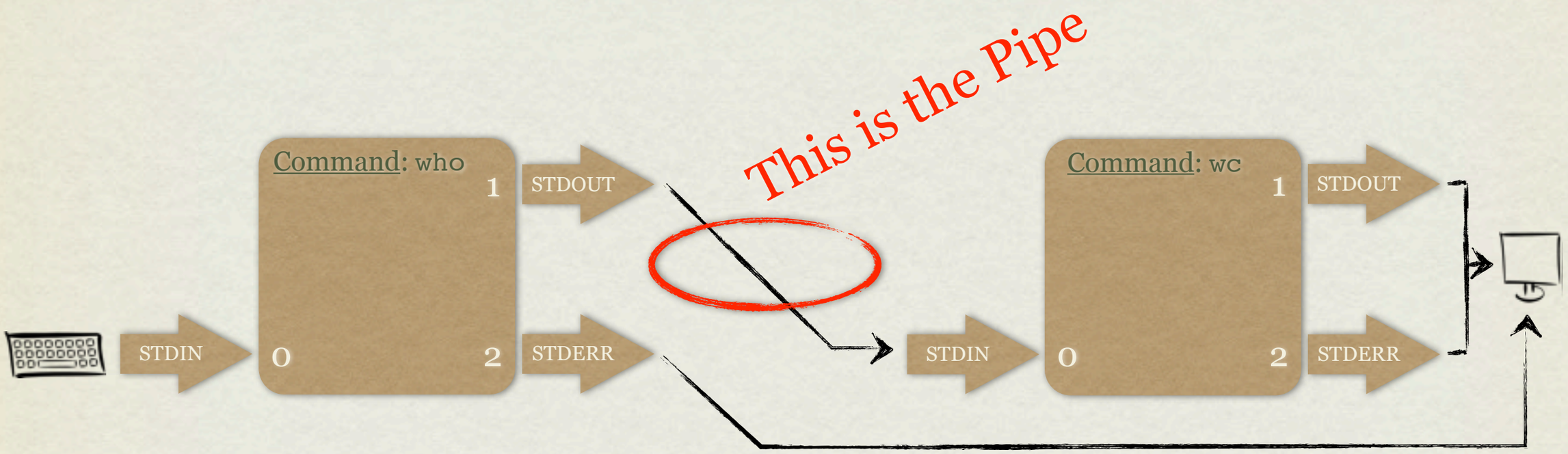
- What is the absolute pathname for ‘file1’?

# PIPES

- Sweet, beautiful, powerful pipes!  My favorite shell feature!
- In concept, pipes are very, very simple
- A pipe operates on two commands, connecting stdout of the command on the left to stdin of the command on the right
  - `who | wc -l`
- Let's look at a picture of this...

# PIPES

```
who | wc -l
```




- The output of `who` is piped into the input of `wc -l`
- This produces a count of the current user sessions

# PIPES

- Pipes can be chained as long as needed, and can also be combined with redirection:
  - `who | fgrep bob | wc -l > bob.sessions`
- It's even possible to intermix pipes and redirection! Just keep your streams straight in your head:
  - `who 2> who.errors | fgrep bob 2>&1 | wc -l`
- Try to diagram the previous command!

# TEE

- A very useful tool when working with pipes is `tee`
- `tee` takes one argument, a filename, and will feed all input from stdin to the file, *while simultaneously feeding the output to stdout*
- In effect, `tee` forks its input stream, sending one copy to a file on disk, and another copy to stdout
- Very useful tool!

# EXERCISES

- Spell check ‘file1’ and, using tee, output the results to the screen and a file on disk.

- Read the man page on wc.  Use this information to count the number of misspelled words in /etc/nsswitch.conf

- Use echo and redirection to append a few more lines to ‘file1’ with information about yourself.

```
slideshow.end();
```