Table of Contents

This is a collection of notes taken from the recommended books, slides, and lectures.

Linux Shell Overview

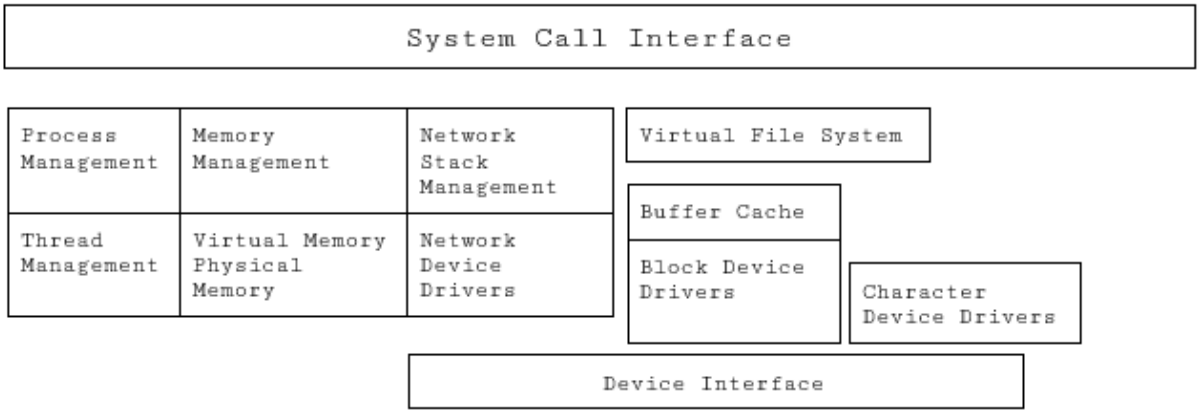
- Linux is an operating system, like Windows (and its sub-types e.g Windows XP, 8, and so on..).
- An operating system manages program execution and users, allocates storage, manages file systems, and handles interfaces to peripheral hardware (keyboards, printers, mouse).
- Peripheral - A device that connects to a computer but isn't part of the computer's main components.

Linux has three layers:

User Space:

shells	compilers	X Server	utilities	libraries	applications

Kernel:



Hardware Space:

CPU	Memory	Network Interfaces	Storage Devices	Other Devices

- user space: Linux shells, compilers, utilities, libraries, UI, applications.
- kernel: Manages processes and threads, memory, network stacks, file systems, and users.
- hardware space: CPU, Memory, network interfaces, storage devices, mouse, keyboard, terminal.

Why Linux?

- Not proprietary, portable to most hardware (Linux was written in C), inexpensive due to being open source, supports many simultaneous users, simple utilities (find,grep,sed,awk) that can be put together (with shells and pipes) for more complex capabilities. Easy integration with system programs and libraries, everything represented as a file (i.e terminals, block devices, network interfaces, processes, etc).

Unix Shell

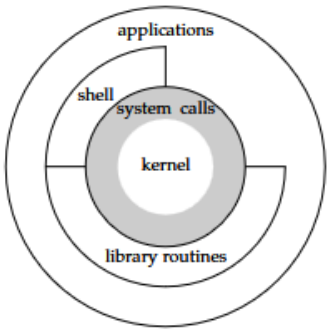
- Unix Shell is a command interpreter and a high-level programming language. It can be used at login, providing interactive and non-interactive capabilities. There are many different shell dialects including the bash and tcsh shell. Bourne Again Shell (BASH) is based on the Bourne Shell (an og UNIX shell). The TC Shell (tcsh) is an expanded version of the C shell (csh).
- Debian Almquist Shell (dash) is a stripped down version of bash used mostly by system and startup scripts due to performance. They don't need the fancy features we see in bash.
- tcsh and bash both provide up and down arrows.
- tcsh to see previously entered commands and gives auto completion of filenames and wildcard matching of filenames.
- bash for auto completion of filenames and wildcard matching of filenames as well.
- bash is the default Linux shell.
- you can tell which outer command shell you're executing by running:

```
$ echo $0
-tcsh
```

```
$ -> shell prompt
black text -> text user types
entry text -> when a file needs to be created/edited this shows the text lines.
green text -> system response
italics text -> a parameter (e.g filename)
```

UNIX Architecture

- OS system -> software that controls the hardware resources of the computer and provides an environment where programs can run.
- The software in this is referred to as the kernel.
- System calls -> The interface to the kernel, made up of a layer of software.



Architecture of the UNIX OS

- Libraries of common functions are built on top of the system call interface, but applications are free to use both.
- The shell is a special application that provides an interface for running other applications.
- Basically an OS consists of the kernel and software that makes a computer useful. This other software includes system utilities, applications, shells, libraries of common functions, etc.

Unix Shell processing steps:

1. Reads an input line from a file or the user.
2. Breaks the input line into tokens based on spaces (understands quotes and escapes, expands aliases).
3. Parses the tokens into simple and compound commands (separated by "|", ";", "&&", "||").
4. Performs expansions (e.g file name wildcards, home directory)
5. Performs redirections and removes the redirection tokens from the command argument list.
6. Executes the command, passing the expanded parameter list.
7. Optionally waits for command completion and exit status.

Unix Command Syntax:

In the Unix shell, spaces separate tokens. The first space is used to delimit the command name.

`~` home directory
`.` current directory
`..` up one level from the current directory
`/` separates directory names in paths

Linux uses a hierarchical file structure (i.e a tree structure) which begins with the root.

Important directories:

`/` -> Root of the tree. Every file in this file system starts within the root directory.

`/home` -> Contains user home directories (e.g /home/abc123).

`/bin` -> Common Linux commands that are binary executables used by all users (e.g ls, cp, rm, mkdir, cat, chmod)

`/sbin` -> Similar to /bin but contains commands that are used by system administrators (e.g fdisk, mkfs)

/etc -> contains configuration files including startup and shutdown scripts

/usr/bin > Contains binaries, libraries, documentation, and source code for slightly less critical programs (e.g ssh, perl, python3, scp)

/usr/lib -> Contains libraries supporting /usr/bin

/lib -> Contains libraries supporting the executables in /bin and /sbin

/mnt -> Contained mounted file systems from various storage devices

/dev -> Device files

Book Notes (UNIX):

The UNIX file system is a hierarchical arrangement of directories and files. Everything starts in the directory called *root*, whose name is the single character /.
A *directory* is a file that contains directory entries. The *stat* and *fstat* functions return a structure of information containing all the attributes of a file.
The names in the directory are called *filenames*.The only two characters that cannot appear in a filename are "/" and the null character. The / separates the filenames that form a pathname and the null character terminates a pathname.
Two filenames are automatically created whenever a new directory is created: "." (refers to the current directory) and ".." (refers to the parent directory). In the root, ".." is the same as "."

☆ stat is a command used directly in the shell to get file information. (operates on a file path)

☆ fstat is a system call used in C programs to get file information using a file descriptor. (operates on a file descriptor)

Pathname:
A sequence of one or more filenames, separated by slashes and optionally starting with a slash, forms a pathname. A pathname that begins with a slash is called an *absolute pathname*, otherwise it's called a *relative pathname*. Relative pathnames refer to files relative to the current directory. The name for the root of the file system "/" is a special-case absolute pathname that has no filename component.

stat example in bash:

$ **stat myfile.txt**

output:

```
File: myfile.txt
Size: 1234       Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 1234567     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  user)   Gid: ( 1000/  user)
Access: 2023-10-01 12:34:56.000000000 +0000
Modify: 2023-10-01 12:34:56.000000000 +0000
Change: 2023-10-01 12:34:56.000000000 +0000
Birth: -
```

fstat example in C:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
        int fd;
        struct stat statbuf;

        // Open the file
        fd = open("myfile.txt", O_RDONLY);
        if (fd == -1) {
        perror("open");
        return 1;
        }

        // Get file status using fstat
        if (fstat(fd, &statbuf) == -1) {
        perror("fstat");
        close(fd);
        return 1;
        }

        // Print some information from the stat structure
        printf("File size: %ld bytes\n", statbuf.st_size);
        printf("Number of links: %ld\n", statbuf.st_nlink);
        printf("File inode: %ld\n", statbuf.st_ino);

        // Close the file
        close(fd);
        return 0;
}
```

C program example that lists all the files in a directory:

```
#include "apue.h"
#include <dirent.h>
int
main(int argc, char *argv[])
{
DIR *dp;
struct dirent *dirp;

if (argc != 2)
err_quit("usage: ls directory_name");

if ((dp = opendir(argv[1])) == NULL)
err_sys("can't open %s", argv[1]);

while ((dirp = readdir(dp)) != NULL)
printf("%s\n", dirp->d_name);

closedir(dp);
exit(0);
}
```

This is a basic implementation of the ls(1) command.
☆ the format is ls[options][directory]. The (1) in ls(1) refers to the section of the manual where the command is documented. In this case, section 1 is for user commands. So it's a specific entry in the UNIX manual pages. The UNIX manual is a collection of documentation files installed on your system. The man pages are stored as files on your system, usually in directories like /usr/share/man or /usr/local/man. Each section has its own subdirectory (e.g., man1/ for section 1, man2/ for section 2, etc.).
How to see the manual pages for the ls command:

man 1 ls

or

man -s1 ls

The above example is a program that prints the name of every file in a directory and nothing else. If the source file is named myls.c, we compile it into the default a.out executable file via:

cc myls.c

Some sample output:

```
$ ./a.out /dev
.
..
cdrom
stderr
stdout
stdin
fd
sda4
sda3
sda2
sda1..
```

In regards to the C program example above:

- The headers: "apue.h" custom header, "dirent.h" is a standard header file for directory operations. It defines the DIR and struct dirent types used in the program (opendir and readdir). This is because the format of directory entries varies between UNIX systems, so we use the functions opendir/readdir/closedir to manipulate the directory.

- opendir returns a pointer to a DIR structure, and the pointer is passed to the readdir function. readdir is called in a loop to read each directory entry and returns a pointer to a dirent structure. Or, when it's finished with the directory, a null pointer. All we examine in the dirent structure is the name of each directory entry (d_name), not the contents.

- The main function declaration "main(int argc, char *argv[])" is expecting command line arguments.

- `if (argc != 2)`
  `err_quit("usage: ls directory_name");`
  Is where the user expects one argument, and if not provided, the program exits with error message err_quit.

- `if ((dp = opendir(argv[1])) == NULL)`
  `err_sys("can't open %s", argv[1]);`
  The program will try to open the directory specified by the first command-line argument (argv[1]) using opendir. If opendir fails, the program exits with an error message err_sys.

- `while ((dirp = readdir(dp)) != NULL)`
  `printf("%s\n", dirp->d_name);`
  `readdir` is used to read each entry in the directory, returning a pointer to a `struct dirent`, which contains info about the directory entry. The `d_name` field of `struct dirent` holds the name of the file/subdirectory, then the program prints each name to a standard output.

- When the program is done, it calls the function "exit" with an argument of 0. The exit function terminates a program. An argument of 0 means success, whereas an argument between 1 and 255 means that an error occurred.

Book Notes (LINUX):

When you enter this:

```
$ echo $0
```

The shell will display the name of the shell you're working with. This works because the shell expands $0 to the name of the program you're running. This means the shell replaces $0 with its actual value, in this case it's a special variable that holds the name of the program/script that's currently being executed. If you're running a shell script, $0 will contain the name of the script. If it's directly in the shell, $0 will contain the name of the shell itself.

A local system might display output like this:

```
$ echo $0
/bin/bash
```

Instead of the book's output:

```
$ echo $0
-bash
```

Example in a script:

If you create a script called myscript.sh with the content:

```
#!/bin/bash
echo "The name of this script is: $0"
```

And then run it:

```
./myscript.sh
```

The output will be:

```
The name of this script is: ./myscript.sh
```

If you change myscript.sh's contents to:

```
#!/bin/bash
echo "The name of this script is: $0"
echo "The contents of this script are:"
cat $0
```

The output will be:

```
The name of this script is: ./myscript.sh
The contents of this script are:
```

Other:

$ **stty**

Is a command that is used to display or change terminal line settings.
For example, to display all terminal settings in a human readable format:

$ **stty --all**

Version information: $ **stty --version**

To delete a word -> CTRL-W
Suspend a program -> CTRL-Z
Line kill key -> CTRL-U/CTRL-X
Terminate running program -> CTRL-C
Quit signal -> CTRL-\
KILL signal (last resort) -> $ **kill -TERM %1**
Repeat previous command -> UP ARROW key
Browse through the command lines -> DOWN ARROW key
Repeat previous command -> !!
ex: su -c "!!" (runs previous command with root privileges)
The command ^old^new^ reruns the previous command, substituting the first occurrence of the string old with new.
The shell also replaces the characters !$ with the last token word on the previous command line.

Example: #after testing it, lp works on WSL

$ **cat meno**

cat: meno: No such file or directory

$ **^n^m^**

cat memo
This is the memo file.

$ **lpr !$**

lpr memo

1. We try to display the contents of a file named meno using the cat command. The file meno doesn't exist, so the shell returns an error.
2. We use history substitution to modify and rerun the previous command ($ **cat meno**).
   old = n
   new = m
3. The shell then runs the modified command: cat **memo**.
4. Since memo exists, its contents (This is the memo file) are displayed.
5. We use history expansion via **!$** to refer to the last argument of the previous command. In the previous command (cat memo), the last argument is (memo).
6. The shell then runs the command: lpr memo. This sends the file memo to the printer using the lpr command.

Root Privileges:

- You can gain root privileges in 2 ways: Logging in as the user named root (so you are working with root privileges until you log out), or use su(substitute user) utility to execute a single command with root privileges or to gain root privileges temporarily to execute several commands. Both require the root password.

```
$ ls -l /lost+found
ls: cannot open directory /lost+found: Permission denied
$ su -c 'ls -l /lost+found'
Password:        Enter the root password
total 0
```

- Without any arguments, su spawns a new shell running root privileges. Typically a # will display when you're working with root privileges.

```
$ su
Password:
# ls -l /lost+found
total 0
# exit
exit
```

- Some distributions (like Ubuntu) ship with the root account locked, so the sudo utility requires you to enter your password to gain root privileges.

```
$ sudo ls -l /lost+found
[sudo] password for joe:
total 0
```

- With argument -s sudo spawns a new shell running with root privileges:

```
$ sudo -s
[sudo] password for joe:
# ls -l /lost+found
total 0
# exit
logout
```

LINUX's Manual Pages:

1. User Commands
2. System calls
3. Subroutines
4. Devices
5. File Formats
6. Games
7. Miscellaneous
8. System Administration
9. Kernel
10. New

Display the man page for the password utility from section 1 of the sys manual:

```
$ man passwd
```

To see the man page for the passwd file from section 5:

```
$ man 5 passwd
```

To view all man pages for passwd:

```
$ man -a passwd
```

Special characters:

& ; | * ? ' " ‘ [ ] ( ) $ < > { } # / \ ! ~

Professor Notes:
Directory Commands:

ls -al -> lists contents of a directory showing long details and including all files (even hidden . files)

cd ~ -> change directory to the user's home directory (/home/abc123)

cd .. -> change directory up one level

cd dirNm -> change directory to the directory named dirNm which must be in the current dir

mkdir dirNm -> make a new directory in the current dir and call it dirNm

rm -r -> recursively removes the specified directory and its contents

pwd -> print working directory, shows the full path for the current directory.

```
#makes a directory named "tryit" in your home directory
$ mkdir ~/tryit

#change to that directory and make a directory in it called "language"
$ cd ~/tryit
$ mkdir language

#we can combine the above into a more simple command by doing:
$ mkdir ~/tryit/language
#here we are not having to use mkdir repeatedly

# change to the language directory
$ cd ~/tryit/language

# see where we are
$ pwd
/home/abc123/tryit/language
```

Examples:

fox01: ~$ **\ls -la**

- directory listing where -l gives long-form listing
- "-a" gives hidden files

a lot of single letter utilities can be combined
so instead of -l -a, we can do -la or -al
"-" is short option
"--" is long option

```
$ echo $0 #shows what shell you're using
-bash
$ which sh #shows you the pash to the sh shell
/bin/sh
$ which bash #shows you the path to the bash shell
/bin/bash
$ bash #starts a new instance of the bash shell

$ tcsh #starts the tcsh shell
> echo $0 #shows the shell you're in (tcsh)
tcsh
> exit #exiting from tcsh shell

$ zsh #same stuff but with zsh shell
% echo $0
zsh
% exit
```

- "." and ".." is universal to every file system (hard links)
- "." refers to the current directory. self-referential.
- ".." refers to the parent directory, useful to move things up without knowing the exact names.

```
fox01:~$ mv file.txt .. #moves file to the parent
fox01:~$ mv file.txt ../.. #moves to the parent of the parent
```

```
$ echo -e "line one\n\n\nline two"
  line one
  first \n
  second \n
  third \n
  line two
```

"~" has a special meaning (home directory)

```
  fox01: ~$ #indicates you're in the home directory
  $ pwd
  /home/ssilvestro
```

"echo" is a built-in command and every shell supports it

```
  $ echo Hello, world
  Hello, world
  #prints out the input from echo
  fox01:~$ echo ~
  /home/ssilvestro
  #taking the ~ and replacing it with the $HOME value of the variable
```

if you want to print the actual ~ you can cancel it via blackslash

```
fox01:~$ echo ~
/home/ssilvestro
fox01:~$ echo \~
~
#putting it as “~” or ‘~’ will also bypass expansion.

fox01:~$ bash echoargs.sh this is a test
arg 1 = this
arg 2 = is
arg 3 = a
arg 4 = test
fox01:~$ bash echoargs.sh “this is” “a test”
arg 1 = this is
arg 2 a test
fox01:~$ bash echoargs.sh “this is a test”
arg 1 = this is a test
```

If you use filename completion in the shell, for spaces it’ll just “escape the spaces”. if you escape the spaces, they’re no longer used to tokenize the command line/break it apart.
So for the above example you can get the same thing by:

```
fox01:~$ bash echoargs.sh this\ is\ a\ test
arg 1 = this is a test
```

stdin/stdout/stderr:

- Many commands in Linux receive their input from stdin (standard input) and write to stdout (standard output).
- File descriptors are small non-negative integers that the kernel uses to identify the files accessed by a process. Whenever it opens an existing file or creates a new file, the kernel returns a file descriptor that we use when we want to read or write the file. The shell automatically opens stdin/stdout/stderr.
- file descriptor is how the OS identifies open files.
- Recall the **FILE *** descriptor declaration in C: **FILE *** is a pointer type used to represent a file stream. It's declared as **FILE *file_descriptor;** where **file_descriptor** is the name you give to the pointer variable. This declaration doesn't open a file, but it creates a pointer that can hold the address of a FILE structure. This structure contains information about the file, such as its location in memory and current position. The file descriptor is how the OS identifies open files.

  Example:

```
int main() {
FILE *fp;
fp = fopen("my_file.txt", "r");
. . .
```

- For interactive shell execution, stdin is defaulted to input from the terminal and stdout is defaulted to display on the terminal. Input can be terminated via CTRL-D.
- In Linux shell, **stdin** can be redirected to *come from a file* by specifying < followed by a filename.
- **stdout** can be redirected to *write to a file* by specifying > followed by a filename.
- When you direct stdout to a file in a command, it'll truncate (remove the contents of) that file.

  Example:

  ls > file.list

  Standard output is redirected to file.list

  Example:

  #redirect output to the file named "hello"

  $ **cat > hello**

  #After running cat > hello, the shell is waiting for you to type input. each line is written to the file "hello"

  hi
  hola
  guten tag

  #after typing your input, CTRL-D signals to the shell that you're done providing input.

  CTRL-D #file "hello" is now saved with the contents you typed.
- CTRL-D sends the end of file byte to the remote host.

  Example:

  This will read from stdin (keyboard) and print to stdout (computer screen)

  fox01:~$ **cat**
  line one
  line one
  line two
  line two
  CTRL-D
- CTRL-D recognizes that there's no more data to read, flushes its buffer, and exits. (Voluntary)
- CTRL-C sends a signal to terminate the process. (Forceful)

Example:

#redirect input from the file named "hello"

$ **cat < hello**

hi
hola
guten tag

$

#this reads from a file and displays its contents

Example:

To copy the file named "hello" to a new file named "greeting"

$ **cat < hello > greeting**

#The contents of the file "hello" are read by cat. The output of cat (the contents of hello) is written to the file "greeting"

```
fox01:~$ cat > mytext.txt
#cat will read from the stdin (my keyboard) and secretly write to
stdout. It parses the file before recognizing whether or not it already
exists, so it'll create the file or obliterate an already existing one.
this is a test
this is line two
fox01:~$ cat mytext.txt
this is a test
this is line two
fox01:~$ cat > mytext.txt
this is a NEW test
this is a new line two
final line.
fox01:~$ cat mytext.txt
this is a NEW test
this is a new line two
final line.
```

- Things are only displayed when stdout goes to your terminal. If it's redirected to the file, it'll go to your file.
- To avoid file obliteration and append, we use ">>"

```
fox01:~$ ./stderr
This is standard output
This is standard error
fox01:~$ ./stderr > stdout.txt
This is standard error
fox01:~$ cat stdout.txt
This is standard output
fox01:~$ ./stderr 2> stderr.txt
This is standard output
fox01:~$ cat stderr.txt
This is standard error
fox01:~$ ./stderr &> stdall.txt
fox01:~$ cat stdall.txt
This is standard error
This is standard output
fox01:~$ ./stderr 1> stdout.txt
This is standard error
#This is pointless; just to show using the file descriptor numbers.
```

If you use filename completion in the shell, for spaces it'll just "escape the spaces". if you escape the spaces, they're no longer used to tokenize the command line/break it apart.
So for the above example you can get the same thing by:

```
fox01:~$ bash echoargs.sh this\ is\ a\ test
arg 1 = this is a test
```

```
fox01:~$ cp cs34_ #The professor has 3 different files beginning with cs3423. by hitting tab, it auto completes the “cs3423”. after specifying further via: cs3423_su and hitting tab here again, it fully completes the name of the file.
fox01:~$ cp cs3423_su20_assign2-solvn-v1.0.4.zip .. #copying this to the parent directory
using mv will move it to another directory such as:
fox01:~$ mv cs3423_su20_assign2-solvn-v1.0.4.zip ../..
-e means “escape sequence”
fox01:~$ echo “a\n\n\nb”
a\n\n\nb
fox01:~$ echo -e “a\n\n\nb”
a
#first \n
#second \n
b
Line Suppression:
fox01:~$ echo hello world
hello world
fox01:~$ echo -n hello world
hello worldfox01:~/courses/cs/3423$
```

more fileNm (pager utility)
Lets you view a long file in your scrollback buffer. Using it will show you one line at a time each time you hit ENTER, or SPACE for one page at a time.

less fileNm
Same as above, but extra features including going between beginning and end, searching, up/down arrow keys.
When using \ls we see output of all our files in columns, but using ls -l displays everything in rows with extra metadata.

Example:
-rw------- 1 namehere faculty 596 Jan 22 13:14 test.c
- The hyphen in “-rw-...” represents the type of file it is. If it's a hyphen it’s a regular file. There is “s” for sockets, “p” for pipes, etc. The next line characters after are the traditional Linux file permissions. Determines who can read, write, and execute this file.
- “1” is the hard-link account that represents the number of directory entries that point to this same file’s data.
Most regular files will have 1, Directories will have atleast 2.
“namehere” is the owner, “faculty” is the owning group, “596” is the file size, “Jan 22 13:14” is the timestamp.

```
fox01:` $ vi whoson.bash
#creates/opens a file named whoson
file: whoson.bash
#!/bin/bash
date
echo “who is on?”
who

fox01:` $ whoson.bash #this searches a bunch of different directories
-bash: ./whoson.bash: Permission denied
fox01:` $ ./whoson.bash
#says look for a file in my current directory named whoson.bash
fox01:` $ chmod u+x whoson.bash
#allows you to change permissions so this says allow “user” + “who owns it”
to execute whoson.bash
fox01:` $ ls whoson.bash
-rwx---- 1 namehere faculty etc…
#now ./ or simply typing it will work too.
```

To redirect stdout and stderr to the same file in bash, use:
$ **command args &> outFile**
To do so in tcsh:
$ **command args >& outFile**

File Redirection

Action	BASH	TCSH
redirect stdin	< filename	< filename
redirect stdout	> filename	> filename
redirect stderr	2> filename	n/a
append to redirected stdout	>> filename	>> filename
redirect stdout and stderr to the same file	&> filename	>& filename
throwing away stdout	> /dev/null	> /dev/null

Manipulating files

`cp source-file destination-file` -> copies source-file creating destination-file

`rm fileNm` -> removes the file fileNm

`mv fileFrom fileTo` -> moves(renames) the file
mv will move the file to another folder if the fileTo is just an existing folder name. If fileTo is just a filename, it renames it. If fileTo is a folder path with a file name, it moves it and renames it.

`cp -R fileListFrom targetDir` -> will copy the files in the fileListFrom including directories and create corresponding directories in the targetDir.

`cat fileNm` -> view file, but scrolls quickly

`more fileNM` -> view file one page at a time

`less fileNm` -> view file similar to more, but with extra features

`vi fileNm` -> edit file using the vi editor

`cat > fileNm` -> creates a file named fileNm using input from the terminal. Multiple lines with ENTER are ok. It writes out when you press CTRL-D.

Listing contents of a directory using -l switch

Shows the details about the files in the directory.

`file type` -> for directories or links to directories, it's a 'd'

`file permissions` -> The read, write, and execute permissions. This is 3 sets of 3 characters.

`links` -> The number of references to this file.

*Note that each directory will have at least 2 (the reference from the parent directory and the . directory link within the directory).

`size` -> the file's size in bytes

`modification date` -> the date (and possibly time) of the last modification

`filename` -> the name of the file

Example:

```
#Show the contents of the language directory
$ ls -al
output:
drwx------ 2 rslavin faculty 4096 Apr 21 2015 .
drwx------ 4 rslavin faculty 4096 Aug 13 2015 ..
-rw------- 1 rslavin faculty 18 Apr 21 2015 hello
```

Example:

```
$ ls
memo
$ cp memo memo.copy
$ ls
memo memo.copy
```

In the above example, **ls** shows that memo is the only file in the directory. Then the **cp** command copies the file named memo to memo.copy
The period is part of the filename, not the file type. After that, a second **ls** command shows two files in the directory: memo and memo.copy

```
$ cp memo memo.0130
```

.0130 is including the date into the name of a copy of a file, in this example it's January 30.

If the destination-file exists before you give a **cp** command, **cp** overwrites it. This is because it overwrites and destroys the contents of an existing destination-file without warning.
Using **-i** prompts you before it overwrites a file.

```
$ cp -i orange orange.2
cp: overwrite ‘orange.2’? y #user answers y for ‘yes’
#in this example it assumes the file named orange.2 exists already
```

Example:

```
#mv existing-filename new-filename to rename a file without making a copy of it.
$ ls
memo
$ mv memo memo.0130
$ ls
memo.0130 #this changes the name of the file memo to memo.0130
```

! mv also has the -i option

lpr (line printer) utility places one or more files in a print queue for literal printing.
lpstat -p -> display a list of available printers
-P option -> instruct lpr to place the file in the queue for a specific printer

```
$ lpr report
#because this command doesn’t specify a printer, the output goes to the default printer.

$ lpr -P mailroom report
# prints the same file on the printer named mailroom

$ lpr -P laser1 05.txt 108.txt 12.txt
#sends multiple files to be printed. this one sends 3 different txt files to be printed by the printer laser1

$ lpq #same as lpstat -o, shows print queue
lp is ready and printing
Rank Owner Job Files Total Size
active max 86 (standard input) 954061 bytes
```

Creating a simple shell script:

bash scriptFilename -> starts a new shell session

date -> displays the date

echo -> displays text (variables can be referenced by using $variableName). newline characters can be embedded with -e option

who -> lists the users who are logged in

Example:

```
#creates a simple shell script named “whoson”
```

```
$ cat > whoson
date
echo “who is on?”
who
CTRL-D

#attempt to execute ./whoson
$ ./whoson
./whoson: Permission denied

#execute the script using bash
$ bash whoson
Wed May 17 16:57:22 CDT 2017
who is on?
maynard pts/1 2017-05-16 13:28 (172.24.136.111)
rslavin pts/2 2017-05-17 16:58 (172.24.136.180)
```

Changing Access Permissions:

The chmod command is used to change mode (i.e change file access permissions). In Unix, there are three user classes for file access permissions:

```
user -> owner
group -> users who are members of the same group (e.g. faculty)
others -> users are neither the owner nor members of the owner’s group

Modes:
r -> read a file/directory
w -> write (modify/delete) a file/directory
x -> execute a file (or recurse a directory)
syntax:
chmod permissions files
There are 2 different syntaxes for the permissions: symbolic and octal.
```

Example:

```
$ ls -l whoson #show the permissions for whoson
-rw------- 1 rslavin faculty 27 Apr 23 1:29 whoson
$ chmod u+x whoson #make “whoson” executable
$ ls -l whoson
-rwx------ 1 rslavin faculty 27 Apr 23 1:29 whoson
$ ./whoson #execute ./whoson
Wed May 17 16:58:35 CDT 2017
who is on?
maynard pts/1 2017-05-16 13:28 (172.24.136.111)
rslavin pts/2 2017-05-17 16:58 (172.24.136.180)
$ chmod a+rx whoson #make “whoson” readable and executable by all users
$ ls -l whoson
-rwxr-xr-x 1 rslavin faculty 27 Apr 23 1:29 whoson
```

chmod permissions using Symbolic Modes

syntax: userClass operator modes

- userClass -> one or more of u (user), g (group), o (other), and a (all)
- operator -> One of + (add), - (remove), = (set the exact modes for the specified user classes)
- modes -> one or more of r (read), w (write), and/or x (execute)

Example:

```
$ chmod a-x whoson #remove execute from all users for whoson command file
$ ls -l whoson
-rw-r--r-- 1 rslavin faculty 27 Apr 23 1:30 whoson
$ chmod g=rwx whoson #set group to be rwx for the whoson command file
$ ls -l whoson
-rw-rwxr-- 1 rslavin faculty 27 Apr 23 1:31 whoson
$ cat > names #create the names file
joe king
may king
jor king
CTRL-D
$ ls -l names
-rw------- 1 rslavin faculty 27 Apr 23 1:32 names
$ chmod g+rw name #add read and write for group to names
$ ls -l name
-rw-rw---- 1 rslavin faculty 27 Apr 23 1:33 names
```

chmod permissions using octal notation

syntax: userOctal groupOctal otherOctal

- each octal digit contains 3 bits: 4 - read, 2 - write, 1 - execute
- userOctal -> the first octal digit (not bit) sets the mode for user
- groupOctal -> the second octal digit sets the mode for group
- otherOctal -> the third octal digit sets the mode for others

syntax:

u - owner
g - owning group
o - all others
then an operator +, -, =
then r, w, or x

- **User classes**
  - **User** - owner
  - **Group** - users who are members of the same group
  - **Others** - users are neither the owner nor members of the owner's group

- **Modes:**
  - "r" - read a file
  - "w" - write, modify, or delete a file
  - "x" - execute a file

octal	read	write	execute
7	1	1	1
6	1	1	0
5	1	0	1
4	1	0	0
3	0	1	1
2	0	1	0
1	0	0	1
0	0	0	0

-rw-----
each of these can be associated with an octal number which goes 1-7
read is worth 4
write is worth 2
execute is worth 1

Example:
cchmod 660 fruits
660 -> (4+2)(4+2)(0)
So we have read+write, read+write, zero
(110) (110) (0)

444 is read-read-read
(100) (100) (100)

Examples:

```
fox01:~$ chmod u+x,go= whoson.bash
#changes execute for the owner, stripping permissions
fox01:~$ chmod a-x whoson.bash #removes permissions for all
                                #a-x same as ugo
```

make sure you're prefixing with a leading 0

```
fox01:~$ chmod 600 whoson.bash
#if you just gave it 600 it interprets it as a decimal number in
languages like C
fox01:~$ chmod 0600 whoson.bash
fox01:~$ chmod u+x whoson.bash
-rwx------ 1 namehere faculty 27 Jan 27 13.38
fox01:~$ chmod a+rx whoson.bash
#execute for everybody
-rwxr-xr-x 1 namehere faculty 27 Jan 27 13.38
fox01:~$ ls -d ~
drwx---- 16 namehere faculty 560k Jan 29 09:51 /home/namehere
fox01:~$ chmod g+rw name
$ ls -l name
-rw-rw-- 1 namehere faculty 27 Apr 23 1:33 names
#this is for if we want read-write permissions for the group
```

Example:

```
#create a fruits file
$ cat > fruits
apple
orange
dog
CTRL-D

#make the owner and group have rw mode for the fruits file
$ chmod 660 fruits
$ ls -l fruits
-rw-rw---- 1 rslavin faculty 27 Apr 23 1:34 fruits

#make all users have r mode for the whoson file using octal
$ chmod 444 whoson
$ ls -l whoson
-r--r--r-- 1 rslavin faculty 27 Apr 23 1:35 whoson

#make whoson rwx for all users
$ chmod 777 whoson
$ ls -l whoson
-rwxrwxrwx 1 rslavin faculty 27 Apr 23 1:36 whoson
```

Aliases

An alias lets you create a shortcut name for a command, file name, or any shell text

bash:

```
alias aliasName='value'
alias aliasName="value"
```

tcsh:

```
alias aliasName `value'
alias aliasName "value"
```

☆ Double quotation marks cause any variable references to be substituted when the alias is created.
☆ Single quotation marks - Any embedded variables would be substituted when the alias is referenced.
☆ The alias command without arguments will list defined aliases.

Note: that surrounding the value with double quotation marks causes any variable references to be substituted when the alias is created. With single quotation marks, any embedded variables would be substituted when the alias is referenced. If you have space around this, the shell's going to think you're trying to run a command named greet with two parameters 'equals' and 'hello'

Example:

```
fox01:~$ greet = hello
No command ‘greet’ found, did you mean:
fox01:~$ greet=hello
fox01:~$ echo $greet
hello
fox01:~$ alias eek=”echo $greet”
#Remembering our 7 steps, the shell won’t even attempt to resolve
whether or not ‘alias’ even exists until the parsing process finishes.
It first sees $greet within the double quotes, and take that current
value of $greet, put as a string literal in the definition of alias
fox01:~$ alias eek
alias eek=’echo hello’
fox01:~$ eek
hello
fox01:~$ greet=xyz
fox01:~$ eek
hello
#doesn’t matter if you change the value of greet
fox01:~$ alias eek=’echo $greet’
#single quotes specify a string literal
fox01:~$ alias eek
alias eek=’echo $greet’
#variable reference is now embedded in it

fox01:~$ xyz
#if you press enter, what will the shell do?
do you think it’s going to check the current directory to see if
there’s a program called xyz?
What if there’s an xyz system utility?
What if there’s 100 executables named xyz spread across the file
system? which of the 100 will it run?
that’s what the Shell steps aim to clarify
Output:
No command ‘xyz’ found, did you mean:
. . .

fox01:~$ /bin/echo hello world
#the shell knows I want to run hello world in /bin/echo
fox01:~$ ../echo hello world
#this also works because we’re telling it to look in the parent
directory

fox01:~$ alias hw=”/bin/echo hello world”
fox01:~$ hw
#replaces the entire command line with “/bin/echo hello world”
#if it’s an alias expanded, you start over again with the steps
hello world
```

If you want to see if something is a built-in function you can use the ‘type’ command

```
fox01:~$ type echo
echo is a shell builtin
fox01:~$ type source
source is a shell builtin
fox01:~$ type ls
ls is an aliased to ‘ls -larth --color –-group-directories-first’
fox01:~$ type cat
cat is hashed (/bin/cat)

fox01:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:..etc
```

Goes through everything to see if there’s an executable named xyz
You need to include the current directory at the very end
“which” locates the paths of executable files

```
fox01:~$ which cat
/bin/cat
fox01:~$ which ls
/bin/ls
fox01:~$ which echo
/bin/echo
```

Example:

```
# tcsh examples
$ alias ll 'ls -l'
$ ll
-rw------- 1 rslavin faculty 13030 Dec 19 19:20 cs2123p1Driver.c
-rw------- 1 rslavin faculty 15232 Jan 7 12:48 cs2123p1Driver.o
-rw------- 1 rslavin faculty 3425 Oct 11 2016 cs2123p1.h
-rw------- 1 rslavin faculty 320 Jan 6 12:18 Makefile
-rwx------ 1 rslavin faculty 20070 Jan 7 12:48 p1
-rw------- 1 rslavin faculty 3403 Jan 6 12:31 p1abc123.c
-rw------- 1 rslavin faculty 7232 Jan 7 12:48 p1abc123.o
-rw------- 1 rslavin faculty 663 Jan 6 12:35 p1Extra.txt
-rw------- 1 rslavin faculty 532 Jan 6 12:36 p1Input.txt
-rw------- 1 rslavin faculty 1702 Jan 7 12:50 p1OutExtra.txt
-rw------- 1 rslavin faculty 1308 Jan 7 12:49 p1Out.txt
# an example with a variable
$ set greet=hello
$ alias eek "echo $greet"
$ alias
eek    echo hello
ll     ls -l
$ eek
hello
$ alias eek 'echo $greet'
$ alias
eek    echo $greet
ll     ls -l
$ eek
hello
$ set greet=boo
$ eek
boo
```

How does the shell resolve what's being executed?

A particular command name could be defined in multiple directories or that name could be an alias.

1. If the command name contains slashes, it assumes you are telling the shell where to find it.
2. It checks for a defined alias, if found, it's substituted for the command.
3. It checks for a shell built-in function.
4. The shell searches the PATH environment variable.

```
#create a user defined version of the "cat" command
$ cat > cat
echo "meow"
cat
CTRL-D
$
#make cat rwx for all users
$ chmod 777 cat
#make an alias for cat
$ alias tiger=cat
#make an alias for my version of cat
$ alias myCat=./cat
```

Example:

In this example we have a script named cat as well

```
fox01:~$ vi cat
fox01:~$ alias tiger=cat
fox01:~$ alias myCat=./cat
fox01:~$ alias myCat tiger
alias myCat='./cat'
alias tiger='cat'
fox01:~$ cat cat
echo "meow"
cat
fox01:~$ cat < numbers.txt
one hundred one
one hundred two
one hundred three
. . .
fox01:~$ which cat
/bin/cat
fox01:~$ ./cat < numbers.txt
#this stops at step 1 because of the slash
meow
one hundred one
one hundred two
. . .
#Runs both versions of cat
fox01:~$ ./cat
#This runs in the current directory, but will block and wait for input
meow
line one
line two
. . .
```

In the script if we did

```
echo "meow"
./cat
```

It'll create a recursive loop (bad)

Professor:

Shell Processing Steps:

1. Reads an input line
   fox01: ~$ **echo hello world**
   hello world
   #it reads the line after pressing enter
2. Breaks the input line into tokens based on spaces
   It begins to tokenize that line, so any amount of any white space counts as a separator.

   Step 2a. Understands quotes and escapes:
   ex:
   fox01:~$ **echo hello world**
   hello world
   Despite the spaces it still has the same output because echo is token 1, hello is token 2, and world is token 3
   fox01:~$ **echo “hello world”**
   hello world

   Step 2b. Expands aliases
   fox01:~$ **alias hw=”echo ‘Hello, world’”**
   fox01:~$ **alias hw**
   Hello, world

3. Parses the tokens into simple and compound commands (separated by “|”, “;”, “&&”, “||”)
   fox01:~$ **echo A; sleep 2; echo B; sleep 2; echo C**
   A
   B
   C
4. Performs expansions
5. Performs redirections and removes the redirection tokens from the command argument list.
   If they were left in there, the program would be confused.
   fox01:~$ **cat < test.c**
   #this prints test.c, but the program removes < because cat doesn’t know what that is and would try to open it. This is because < is meaningful to bash, not to cat.
6. Executes the command, passing the expanded parameter list.
   fox01:~$ **slklskdfksd $USER < test.c**
   We could do steps 1 - 5, and still not know whether or not the command will execute. We could do the above and and then fail because there is no command ‘slklskdfksd’
   output would be:

```
slklskdfksd: command not found
```

7. Optionally waits for command completion and exit status

```
fox01:~$ sleep 5.
#the command prompt would wait 5 seconds before spawning another
fox01:~$

fox01:~$ sleep 5. & #gives you your prompt back immediately, & is also a command separator.
output:
[1] 1015
#bash is saying you have [1] process running behind the scenes and its process ID is 1015
Whenever you hit enter after 5 seconds, bash will notify you that the process completed successfully
output:
[1]+ Done                    sleep 5
fox01:~$ jobs #will show any other processes
```

Pipelining

Pipelining lets you take the output of one command and use it as the input for another command. The | symbol connects them so you can chain commands together.

Unix: Pipelines do not work with OS X resource forks; only data forks.

A Pipeline takes the output of one utility and sends that output as input to another utility (or takes standard output of one process & redirects it to become standard input of another process).

Utility head can accept input from a file named on the command line or via a pipeline, from standard input.

Syntax: `command_a [arguments] | command_b [arguments]`

Pipeline syntax is easier to type and is more efficient.

Given:

```
command_a [arguments] > temp
command_b [arguments] < temp
rm temp
```

The preceding syntax is literally this, but does not create an unnecessary file that will be deleted at the end (temp).

There is also: |& (short for 2>&1)

command1 |& command2 is equivalent to command1 2>&1 | command2 where both standard output and standard error is piped.

You can include any utility in a pipeline that accepts input from either a specified file or from standard input, as well as utilities that accept input only from standard input.

Ex: `tr string1 string2` #tr takes its input from standard input only.
#looks for chars that match one of the chars in string1.

Example:

```
$ cat abstract
I took a cab today!
$ cat abstract | tr abc ABC
I took A CAB todAy!
$ tr abc ABC < abstract
I took A CAB todAy!
```

Differences: With pipelining(|), 'cat' reads the file & outputs its contents, then it's passed to 'tr' via the pipe. With Input Redirection (<) the shell directly feeds the contents of the file abstract to fr without needing 'cat'. *tr doesn't change the contents of the original file.

Example:

```
$ who > temp
$ sort < temp
max pts/4 2013-03-24 12:23
max pts/5 2013-03-24 12:33
sam tty1 2013-03-24 05:00
zach pts/7 2013-03-23 08:45
$ rm temp
#displaying the efficiency of pipelining
$ who | sort
max pts/4 2013-03-24 12:23
max pts/5 2013-03-24 12:33
sam tty1 2013-03-24 05:00
zach pts/7 2013-03-23 08:45
```

Scenario: If multiple people are using the system you can access information about only one of them by sending the output from who to grep. The grep utility displays the line containing the string you specify.

```
$ who | grep sam
sam        tt1        2013-03-2 05:00
#This is the same as:
$ who |
> grep 'sam'
```

Because | implies continuation:
When a command line ends with | it indicates that the output of the command is being passed as input to another command. But the shell expects more input to complete the command pipeline and how it handles that situation depends on whether it's running interactively or within a script.
In an interactive environment (directly to terminal): It issues a secondary prompt

```
$ echo "Hello" |
>
```

In a script: Processes the next line as a continuation

```
echo "Hello" |
wc -c
```

```
fox04:~$ ls | more
```

| -> input/output redirection
In the process that'll eventually run ls, it secretly redefines standard output such that instead of going to the screen; it goes to the pipe which is just a piece of memory in the kernel.
In the other process that's going to run the more command; its file descriptor table is modified so when more reads from standard input, it's secretly reading from the same pipe.

```
fox01:~$ who | sort
#This will list who's on and sort them
anl176 pts/13 timestamp
gjh148 pts/2 timestamp
jaq088 pts/9 timestamp
jsherett pts/3 timestamp
nsd690 pts/10 timestamp
ssilvestro pts/14 timestamp
fox01:~$ who | sort -r
#This will do the above, but reverse the list
fox01:~$ who | sort | tac | \grep -v ssilvestro
#This does the above, tac also prints files in reverse as -r does, and it removes the instance of Silvestro in the list printed.
fox01:~$ who | sort | tac \grep -v ssilvestro | wc -l
10
#This is "how many users are online besides myself"
```

Example:

```
$ sort months | head -4
Apr
Aug
Dec
Feb
```

Displays number of files in a directory: The wc (word count) utility with the -w (words) option displays the number of words in its standard input/file you would specify.

```
$ sort ls | wc -w
14
```

Example:

```
$ ls
memo memo.0714 practice
$ echo Hi
Hi
$ echo This is a sentence.
This is a sentence.
$ echo star: *
star: memo memo.0714 practice
```

*Random: $ **tail months | lpr** #sends output of program to a printer

---

Filters

A filter is a command that processes an input stream of data to produce an output stream of data while utilizing | to connect the stdout of one command, to the stdin of the filter, to the stdin of another command.
☆Not all utilities can be used as filters

```
$ who | tee who.out | grep sam
sam         tt1         2013-03-24 05:00
$ cat who.out
sam         tt1         2013-03-24 05:00
max         pts/4       2013-03-24 12:23
max         pts/5       2013-03-24 12:33
zach        pts/7       2013-03-24 08:45
```

☆ (tee) copies its stdin both to a file and to stdout.
The output of who is sent via | to stdin of 'tee'. 'tee' then saves a copy of stdin in a file named who.out and also sends a copy to stdout. stdout of 'tee' goes via | to stdin of grep, which displays only those lines containing the string 'sam'.
☆ Using 'tee' with '-a' (append) allows it to append a file instead of overwriting it.

Lists

A list is 1 or more |, each separated from the next by one of the control operators: ';', '&', '&&', '||'
'&&' and '||' have equal precedence. ';' and '&' have lower precedence.

- AND list:
  pipeline1 && pipeline2
  where pipeline2 is executed (if and only if) pipeline1 returns a true/zero exit status.
  Example:
  ```
  $ mkdir /newdir && cd /newdir
  mkdir: cannot create directory '/newdir/: Permission denied
  ```
- OR list:
  pipeline1 || pipeline2
  where pipeline2 is executed (if and only if) pipeline1 returns a false/nonzero exit status.
  Example:
  ```
  $ ping -cl station &>/dev/null || echo "station is down"
  station is down
  ```

Foreground - All commands up to this point have been run in the foreground (shell waits for it to finish before displaying another prompt & continuing) vs running a command in the background and not having to wait.
Job - Another name for a process running a pipeline. You can have 1 foreground on a screen & many background jobs.
Job/PID number - Type & just before the return that ends the command line to run it in the background > shell assigns a number to the job & displays this job number (smaller number) between brackets > the shell proceeds to display the process identification number (larger number) assigned to the OS > each number identifies the command running in the background.

Example:

```
$ ls -l | lpr &
[1] 22092 #[1]->job num, 22092->PID
[1]+ Done       ls -l | lpr #msg when job completes execution
```

File Name Patterns

?

Matches any single character. For example, p? which matches p1, p2, and p3, but would not match p1.h

p? specifically means you're looking for a two character name, so p* or p- would work, but this is why p1.h wouldn't.

"*"

Matches from zero to many of any characters. For example, p* would match p1, p1.h, p1main.c

[list] (character class)

Matches one character to any of the characters listed within the brackets. For convenience, range abbreviations can be used (e.g [a-f],[0-9]) We can also do [a-f,A-F,0-9], [aeiouy], [0-9xyz]. We do have to specify if we want lower or upper cases. So for both, we'd put [aeiouyAEIOUY].

If we want to include a hyphen it has to be at the beginning or end like: [-a-z] or [a-z-], brackets []a-z].

[^list]

This matches one character if it isn't listed within the brackets.

[^a-z]

Example:

```
$ ls
cs2123p1Driver.c  Makefile       p1abc123.o   p1OutExtra.txt
cs2123p1Driver.o  p1             p1Extra.txt  p1Out.txt
cs2123p1.h        p1abc123.c p1Input.txt
$ ls p? #looks for files that starts w/ 2 chars, first has to be p other has
p1       to be anything else
$ ls p*
p1 p1abc123.c p1abc123.o pe1Extra.c p1Input.txt p1OutExtra.txt
$ ls [a-c]*
cs2123p1Driver.c cs2123p1Driver.o cs2123p1.h
$ ls [^a-c]*
Makefile p1abc123.c p1Extra.txt
p1      p1abc123.o p1Input.txt
#echo doesn't see the pattern, only the matches.
```

Example:

```
fox01:~$ ls {p1I,p10}*
This is going to look for all the files that start with p1I or p10
->It creates two patterns: ls p1I* p10*
fox01:~$ echo x{red,blue}y
xredy xbluey
fox01:~$ echo x{red,blue}{car,truck}y
xredcary xredtrucky xbluecary xbluetrucky
Basically it distributes across both sets
fox01:~$ ls {p1I,p10}* #ls p1[IO] is the same (simpler)
-rw---- 1 namehere faculty 0 Aug 1 2024 p1Out.txt
-rw---- 1 namehere faculty 0 Aug 1 2024 p1Input.txt
-rw---- 1 namehere faculty 0 Aug 1 2024 p1OutExtra.txt
```

```
fox01:~$ ls [a-z][0-9][abc]*.o
-rw---- 1 namehere faculty 0 Aug 1 2024 p1abc123.o
#search for 3 characters, first has to be a letter, 2nd has to be a
digit, third has to be a letter
```

echo p1*.*

Matches anything that has p1 and a “.”
like p1test.txt.bak.final.raw.bin
Just saying there has to be a dot in the rest of the file, and will match all dots.

echo p1*.?

Looks for a filename that begins with p1, dot, and ends with one character as the filetype.
So p1abc123.o or p1abc123.c would be the output.

These are also known as “wildcards”, and the file name matching is also known as “globbing”. Filenames that contain
These special characters are called ambiguous file references because they do not refer to one specific file. The process the shell performs on these filenames is called pathname expansion or globbing.

$ **echo [^a-d]1{abc,Out,xx}***
reads this as:
$ **echo [^a-d]1abc* [^a-d]1Out* [^a-d]1xxx***

```
p1abc123.c p1abc123.o p1Out.txt p1OutExtra.txt [^a-d]1xxx*
#last command is printed because there’s no match
```

$ **ls [^a-d]1abc* [^a-d]1Out* [^a-d]1xxx***

```
ls: cannot access [^a-d]1xxx*: No such file or directory
```

Why is rm *.o potentially dangerous?
The star will match everything. If you did ‘* .o’ it matches all files in the directory, since adding a space is like: ‘rm *’ (delete everything)
Adding $ rm -i * .o The ‘-i’ gives additional protection; it asks whether or not you’re sure you want to delete xyz.

More Examples (From Book):

?:

```
$ ls
mem memo12 memo9 memomax newmemo5
memo memo5 memoa memos
$ ls memo?
memo5 memo9 memoa memos
```

*:

```
$ ls
amemo memalx memo.0612 memoalx.0620 memorandum sallymemo
mem memo memoa memoalx.keep memosally user.memo
$ echo memo *
memo memo.0612 memoa memoalx.0620 memoalx.keep memorandum memosally
$ echo *mo
amemo memo sallymemo user.memo
$ echo *alx *
memalx memoalx.0620 memoalx.keep
```

-a displays hidden filenames:

```
$ ls
aaa memo.0612 memo.sally report sally.0612 saturday thurs
$ ls -a
. aaa memo.0612 .profile sally.0612 thurs
.. .aaa memo.sally report saturday
$ echo *
aaa memo.0612 memo.sally report sally.0612 saturday thurs
$ echo .*
. .. .aaa .profile
```

[]:

These define a character class that includes all characters within the brackets (GNU calls this a character list).
[6-9] represents [6789], [a-z] represents all lowercase letters in the alphabet, [a-zA-Z] represents all letters, both upper and lowercase.

^/!:

When an exclamation point '!' or a caret '^' immediately follows the opening bracket '[' that starts a character-class definition, the character class matches any character not between the brackets.

```
$ ls
aa ab ac ad ba bb bc bd cc dd
$ ls *[^ab]
ac ad bc bd cc dd
$ ls [^b-d] *
aa ab ac ad
```

☆ The ls utility can't interpret ambiguous file references.

```
$ ls ?old  #is expanded to:
hold
$ ls \?old
ls: ?old: No such file or directory
```

Processes and Threads

A process is an instance of a program or shell script running in an OS. A program is just a static entity, just a file on some storage device, and just a collection of bytes.
A process is an instance of that program in execution.

- Each instance has its own address space (its own memory) and execution state (its own register set, etc).
- Each process has a process ID (PID) (integer)
- A process has at least one thread.
- A process can have many threads, allowing concurrent execution, but sharing of memory.

A thread is a flow of control within a process.

You can see the running processes by eecuting the 'ps' command.
It's not complex, just so many options/format specifiers.

```
$ ps -fu userId
UID PID PPID C STIME TTY TIME CMD
rslavin 10476 10387 0 18:05 ? 00:00:00 sshd: rslavin@pts/2
rslavin 10477 10476 0 18:05 pts/2 00:00:00 -tcsh
rslavin 10488 10477 0 18:06 pts/2 00:00:00 ps -fu rslavin
```

Prof says: Commit to memory a few invokations of 'ps' that'll keep you covered in 90%-ish circumstances.

```
☆ ps -ef #checks status of active processes + display tech info (PID)
☆ ps aux #snapshot of all running processes
☆ ps jf  #detailed list of processes w/ their parent processes

#good to pipe -ef with | more or | less because it's generally lengthy.
# see every process (not just a particular user)
$ ps -ef | more
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 May15 ? 00:00:01 /sbin/init
root 2 0 0 May15 ? 00:00:00 [kthreadd]
root 3 2 0 May15 ? 00:00:00 [ksoftirqd/0]
root 5 2 0 May15 ? 00:00:00 [kworker/0:0H]
root 7 2 0 May15 ? 00:00:19 [rcu_sched]
root 8 2 0 May15 ? 00:00:00 [rcu_bh]
root 9 2 0 May15 ? 00:00:00 [migration/0]
root 10 2 0 May15 ? 00:00:00 [watchdog/0]
root 11 2 0 May15 ? 00:00:00 [watchdog/1]
root 12 2 0 May15 ? 00:00:00 [migration/1]
root 13 2 0 May15 ? 00:00:00 [ksoftirqd/1]
root 14 2 0 May15 ? 00:00:00 [kworker/1:0]
root 15 2 0 May15 ? 00:00:00 [kworker/1:0H]
...
--More-
$ ps -ef | \grep --color -- "-bash" #how many instances of bash as a login shell
$ ps aux | wc -l #get a linecount
277 #277 processes on fox01
$ ps jf #gives a process status to see what's going on
PPID PID  PGID SID  TTY    TPGID STAT    UID  TIME COMMAND
7063 7064 7064 7064 pts/11 6252  Ss 7843 0:00 -bash #login shell bash
7064 6049 6049 7064 pts/11 6252  S  7843 0:00 \_bash #created this subshell
6049 6252 6252 7064 pts/11 6252  R+ 7843 0:00   \_ps jf
```

When you see the hyphen in the command name (-bash) you know that's the login shell.

Example:

```
$ ps jf
PPID PID  PGID SID  TTY    TPGID STAT UID  TIME COMMAND
7063 7064 7064 7064 pts/11 6252 Ss   7843 0:00 -bash #using login shell
6049 6252 6252 7064 pts/11 6252 R+   7843 0:00   \_ps jf

$ bash
$ bash
$ sleep 60 &
[1]4251
$ ps jf
 PPID   PID  PGID    SID TTY    TPGID  STAT  UID  TIME COMMAND
26323 26324 26324  26324 pts/12  4278  Ss   7843  0:00 -bash
26324  4014  4014  26324 pts/12  4278  S    7843  0:00 \_ bash
 4014  4033  4033  26324 pts/12  4278  S    7843  0:00    \_ bash
 4033  4251  4251  26324 pts/12  4278  S    7843  0:00       \_ sleep 60
 4033  4278  4278  26324 pts/12  4278  R+   7843  0:00       \_ ps jf
26212  26213 26313 26313 pts/11 26213  Ss+  7843  0:00 -bash
26076  26077 26077 26077 pts/2  26077  Ss+  7843  0:00 -bash
```

Both instances of _bash were created after typing bash and hitting enter, same with sleep 60 and ps jf.

- If 'ps' finds that the session doesn't have a controlling terminal, either 0 or -1 is printed depending on the platform.

Process Structure

- fork() system call -> parent process that forks (spawns) a child process, which in turn can spawn other processes.
  - (one process turns into two) initially the two spawns are identical except that one is identified as the parent, and the other the child. The OS routine/system call that creates a new process is named fork().
- init daemon -> a single process called a spontaneous process with PID number 1. this holds the same position in the process structure as the root directory does in the file structure: it's the ancestor of all processes the system and users work with.
- when entering the name of a program on the command line, the shell spawns a new process, creating a duplicate of the shell process (subshell). The new process then attempts to execute the program.
  - Like fork(), exec() is a system call. if the program is a binary executable, exec() succeeds, and the system overlays the newly created subshell with the executable program. If it fails, it's assumed to be a shell script and runs it in the script. Unlike a login shell (expects input from the command line), the subshell takes its input from a file (shell script).

Process Identification

- PID numbers -> Linux assigns a unique PID number at the inception of each process. As long as it exists, it keeps the same number.
- CMD/Command - process running the shell, PPID - PID number of the parent of the process.

- fork() and sleep() - When the shell is given a command, it usually forks a child to process to execute the command; while that is executing, the parent process sleeps (via sleep() sys call). When the child process finishes, it tells the parent of its success/failure via its exit status and then dies. The parent process then wakes up again and prompts for another command.

Background vs Foreground

We've executed processes in the foreground. Linux also allows execution of processes in the background, allowing the foreground to be used.

- Specifying '&' after a command line causes it to be executed in the background.
- To bring a bg job to the foreground, use the fg command.

Recall the 7 parsing steps; The 7th step was to optionally wait for the command line to finish before giving you the prompt back and giving the exit status. By default that's what it does.

```
$ jobs #shows what background jobs I have
$ sleep 10 & #quick sleep command to display a job
[1] 7936
#bash says: created a job, PID number associated with this job process is 7936
$ jobs
[1]+ Running     sleep 10 &
   + means last job running
   - means 2nd to last job
$ sleep 60 & sleep 61 & sleep 62 &
[1] 9620
[2] 9621
[3] 9622
$ jobs
[1]        Running sleep 60 & #neither + or - so unknown order
[2]-       Running sleep 61 & #second to last
[3]+       Running sleep 62 & #last one created
$ fg
sleep 62 #if you don't give it a job number, it brings back into the
          foreground whatever the last job was
$ fg 1
sleep 60 #gave first job in the foreground
```

- $ **fg %** (you're supposed to put a percent sign to separate job numbers since they're all integers technically)
- $ **kill %1** #kills job 1
- $ **kill -9 10685 10686** #kills particular process via process ID

Example:

```
$ sleep 123 & sleep 456 & sleep 789 &
[1] 12499
[2] 12500
[3] 12501
$ jobs
[1]  Running    sleep 123 &
[2]- Running    sleep 456 &
[3]+ Running    sleep 789 &
$ kill 12499 12500
[1]  Terminated  sleep 123
[2]- Terminated  sleep 456
$ jobs
[3]+ Running    sleep 789
$ kill %3
[3]+ Terminated sleep 789
```

By default when you use the kill command, it sends the SIGTERM signal which by default will terminate things; Equivalent to CTRL-C.

- `kill -l` -> shows list of signals

Can use other signals;

```
$ kill -6 %1
[1]+ Aborted (core dumped) sleep 123
```

Mess with later: C has its own library function for kill: `int kill(pid_t pid, int sig);`