Unix and Linux compact tutorial

Alex. Gerbessiotis
CS Department
NJIT

November 24, 2023

revised March 22, 2024

1. Introduction

The implicit assumption is that you are familiar with the use and requirements of a VPN connection and the use of an open ssh client (generically referred to as ssh). These might be needed if you plan to connect to a Unix or Linux machine from outside of NJIT (VPN and then ssh) or from within NJIT (ssh) but from a non *nix machine (e.g. a windows computer).

A secure shell client for Windows is MobaXterm; NJIT's software repository allows you to download a copy of it. Windows also provides a command-line ssh client as well (it is possible that some versions of Windows do not have it). OSX (Mac) also has a command line ssh client available through a Terminal application. And naturally *nix has both a client and a server SSH; the client allows users to connect to other machines.

If you need more information on how to use VPN or the MobaXterm client read document AVG-24-01r2 "Connecting to Unix/Linux at NJIT".

An NJIT URL (Uniform resourse locator) with info on *nix commands is shown below.

https://ist.njit.edu/common-UNIX-commands

For the remainder it is assumed you have opened a login session to a Unix/Linux system either through a direct login or through ssh (secure shell).

2. Unix and Linux

Linux is based on the Unix operating system (OS) and environment. The Unix operating system consists of a kernel (the part of the OS that is in main memory all of the time along with the data structures required for its proper operation) plus a variety of services that the operating system provides.

Unix was introduced in the late 60s/early 70s in the then AT&T Bell Labs by K. Thompson and D. Ritchie and was originally developed for a DEC PDP-11 minicomputer. It was originally written in assembly. In 1973 it was rewritten in the programming language known as C. The OS initial interaction with a user is done with some teletype terminals (also known as Video Display Units) to which a keyboard is attached. Thus interaction with a user was done through a device accepting and displaying 80 characters per line of fixed width ASCII output (and a total of 24 such lines). Think of this Courier font being the only option for terminal input (through the keyboard) and output (through the small 8-10inch screen of the terminal).

The UNIX operating system supported timeshared multitasking of user processes. A program in execution is known as a process and the OS's kernel is actively managing processes in main memory as opposed to a program that resides passively in secondary memory, i.e. a hard disk drive for which the OS knows nothing about its contents.

Timesharing means that multiple user’s processes have shared access of the CPU (processor) for a limited amount of time in a round-robin fashion. Thus over a period of roughly eight seconds, five users have access to the CPU for roughly 1 second per user at a time in a round-robin fashion (1-2-3-4-5-1-2-3-etc). The period of 1second is known as the quantum or timeslice. The operating system's kernel is responsible for efficiently switching from one user to another user's process(es). If a user process has no activity during its timeslice/quantum in timeshared multitasking the OS would switch the usage of the CPU to another process of the same user or some other user. And if one process had to stop using the CPU to do an I/O operation (e.g. printing) then another user's process or another process of the same user could take over the CPU since multitasking is also supported. Whereas the objective of timesharing is to minimize CPU response time for processes, and the objective of multitasking is just to increase CPU

utilization (fraction of CPU time used to execute user processes), the objective of timeshared multitasking is to both minimize CPU response time and maximize CPU utilization.

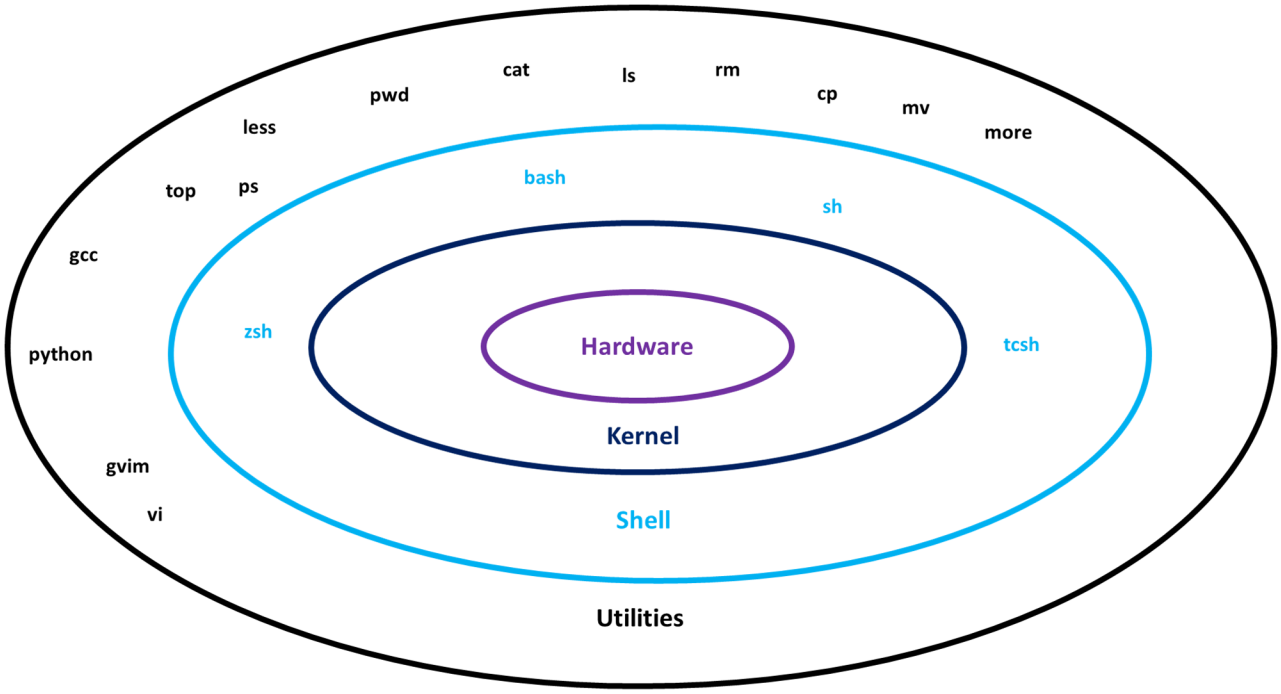
In a Unix system, a user is logged on to the system by providing a set of credentials (login name and an associated password), a process already familiar to you from the earlier sections of this document. At NJIT we call the credentials MyUCID credentials consisting of a MyUCID (login name) and a MyUCID password. (In our prior examples we used for a MyUCID login name the name myUCID.)

Immediately after the login has been completed successfully the Unix environment would become available to the logged on user and a program (process) would start executing in the user's environment after the user's login: the Unix shell process.

The UNIX shell process would allow the user to interact with the operating system and start, stop, suspend and resume the execution of services provided by the OS or create and manage user created processes. These services are programs and when run, those programs become processes. The definition of a process is 'a program in execution'. All interaction is done through the terminal and its associated keyboard that was used by the user to gain access to the system: the user types in commands to the shell and after the shell interprets these commands, it invokes services of the OS to execute/realize those user commands as needed and as privileged. The OS might decline to execute some of these commands for safety or security reasons based on the credentials (privileges) of the user.

To keep interaction short UNIX commands to the shell are short and sometimes intuitive. For example command **ls** is short for list (the contents of a directory), **cat** for catenate (list contents), **mkdir** for making/creating a directory, **ps** for process status, **cd** for changing directory, **pwd** for printing the name of the (current) working directoy, **mv** for move, and **cp** for copy.

Moreover, the UNIX shell provides command line editing, and history of interactions with the shell thus allowing for editing a previously lengthy command instead of retyping it before re-execution or repeating a frequently executed command by easily recalling it from a history of prior interactions. This is faciliated by the arrow keys. Autocompletion of commands is possible by using the TAB key.



Every command of the shell such as ls, ps, mv, cp, pwd, cat, cd, is an executable program residing in secondary memory (disk). It was originally written in C and compiled and assembled subsequently into the executable file named ls, ps, etc. Thus typing a command such as

```
%  ls
```

would cause the shell (an OS process) to load the program named ls from secondary memory into main memory, thus turning it into a process and then the process created executes the code of (the program) ls.

A reminder: the % is the shell prompt. You do not type it. It is output by the shell to remind you that 'I, the shell, have your full attention: please type in your request'. Moreover when you do so and type your request (ls in this case) do not forget to tell the shell that you are done when you are done typing your command. You do so by pressing the ENTER key of the keyboard at the end.

At that moment the shell interprets your input (in the example above it is an ls), the text between the prompt and the ENTER,

and executes it as needed. Every execution of a process in linux (in this case ls) by default creates and interacts with three files associated and connected with (two or) three devices:

(a) standard input also known to the user as the file with fd (file descriptor) 0,

(b) standard output with fd equal to 1, and

(c) standard error with fd equal to 2.

Unless the shell is instructed otherwise by you, standard input is associated with your terminal's keyboard, and standard output and standard error are both associated with the terminal's screen. Terminal in modern computing refers to your laptop or desktop!

When a user logs on the system the location of the user in the filesystem is the user's home directory. The user can identify this location in two different ways either by typing

```
% pwd
```

(The other way is to type `echo ~`.)

Moreover in Linux, multiple commands can be executed one after the other in the command line. It is therefore possible to

```
% ls ; date ; echo "Hi"
```

Thus in the above example after ls is executed, the current date and time is printed, and afterwards a message gets printed on the standard output, the terminal/screen used by the user.

Commands can have options. An option is preceded by a dash. An option is usually a single (english language) letter. The option in the invocation of the shell below is an ell (not a numeric one) and the request to ls is for a long output or listing

```
% ls -l
```

In addition to options a command can have one or more argument list usually after the option(s) and separated by at least one space character.

```
% ls -l  filename1 dir2 filename3
```

In the example above we are requesting a long listing of a variety of names that map to filenames or directories. So we have provided three arguments to ls above. Note that in a Unix or Linux filesystem everything is a file. Every file has a type.

A directory is a file of type directory; an (ordinary) file is a file of type (regular) file. Other types include a character device file type, a block device file type, named pipe, socket, and symbolic link.

A little secret that is about to be revealed. The request (command)

```
% ls -l
```

has an implicit argument. The request is equivalent to

```
% ls -l .
```

The indicated period is the argument and it is an alias to the "current working directory". This is the directory of the filesystem the current user (activity) is residing at the moment of invocation. Thus all the files of the current working directory would be listed. Note that in Unix, every directory has a parent directory. The alias to it is .. (two periods next to each other).

```
% ls -l ..
```

If you are interested in printing the current working directory, as noted earlier the command to do so is pwd.

Moreover in Linux, multiple commands can interact with each other with a mechanism known as an unnamed pipe. A pipe is a FIFO (First In First Out) queue that accepts input from the output of one command and generates output that will become the input of another command. Thus

```
% ls | egrep filename
```

consists of the command ls that outputs the contents of the current directory (this description makes sense after you read the next section if you are not familiar with any operating system's structure) and directs this output not on standard output (screen) but to the unnamed pipe indicated by the pipe | symbol. The unnamed pipe indicated receives as input the transmitted by ls output and then it generates its own output that is to become the input of the command egrep. The command egrep filters its input by discarding all lines that do not

contain the string/word filename and thus preserving to the output that it will generate the lines that contain the string filename. The combined execution thus prints the output lines of ls that contain the string filename.

Pipes can allow multiple cascade communication such as the following one.

```
% ls -l |  egrep filename | sort  | less
```

Let us conclude this section on a different note. Linux is the Linux kernel (written in the programming language known as C though it includes a bit of assembly as well) plus all the other services provided by the operating system (such as the GNU's utilities). The combination of kernel + OS services is known as the Linux operating system. Android consists of a Linux kernel plus all the services made available by Android. The combination of the Linux kernel and the OS services is known as the Android operating system. Kernel is the part of the OS that is expected to be in main memory ALL the time. A PASCAL or FORTRAN compiler is not part of the kernel!

3. Unix filesystem hierarchy

The Unix file system structure is hierarchical. This extends to Linux. The term file system has not been defined yet and it is thus being used generically at this point. In fact there are more than one types of a filesystem in Unix (and Linux) yet the discussion is generic and applies to all of them. The same interface is being used even if internally the systems are different. Moreover, on the osl machines at NJIT, to this hierarchical structure an external file system is further attached (mounted) that is known as AFS (for Andrew File System). It is a distributed file system with certain advanced features that we will not describe here. In summary AFS allows you to access your files independently on whether you are logged on to osl10.njit.edu or osl20.njit.edu or some other machine at NJIT that has access to AFS (including Windows or OSX machines but also Linux or Unix machines).

A Unix filesystem (or its structure) resembles a rooted acyclic directed graph (some people might view it as a tree) whose nodes are files: a filesystem of files! Since at this point we get on into discrete math territory, we won't pursue further those terms.

If there is a term overload let us start from scratch defining them in sequence.

A hard disk drive (HDD) can be split into logical subdivisions that are known as partitions. We won't disk the hardware subdivisions of a hard disk drive: sectors, track, clusters, cylinders or other logical formations such as volumes.

A partition can be assigned a format. The format of a partition of a hard disk drive is known as a filesystem.

A filesystem (on a partition) can be created, mounted (to the OS and thus activated and its files and structure can be revealed and viewed through the OS) or unmounted. One can not destroy (delete) a filesystem directly: creating a new one on top of an older one overwrites the older one. The filesystem describes how a partition is organized logically into files and also describes the areas of the partition that stores information on those files and their data (metadata).

A filesystem contains files of different types.

A file is a collection of data on external memory (also known as secondary memory, and coloquially referred to as a disk drive). Several time by abuse of notation or slip of the tongue a disk drive becomes a hard disk drive (HDD) even if it is a solid state drive (SSD). The generic term drive include FDD (floppy disk drives) or CD-ROM drive or DVD drives! To cut a long story short a file is a collection (organized form) of data on a disk drive's formated partition (i.e. a filesystem).

Unix and Linux currently support several common file types. The most common ones are listed below. Associated with the file type's name we use a single character to represent and describe the file's type. Types of files in Unix and Linux are as follows.

- :(regular) file,

d :directory,

l :symbolic link, also known as soft-link,

p :named pipe,

b :block device file,

c :character device file, and

s :socket (used in networked communication).

In Unix every file is identified by a numeric identifier (value) known as the file's inode (number). Inode stands for index node. The inode (or inode number) is an index on a table that is also known as the inode table. Thus the index to the table is the inode (aka inode number).

An inode value 10 indicates that the information for a specific file (the one with inode value or inode number 10) is available at index 10 of a table that is known as the inode table. The inode table was created when a filesystem was created on a partition of a HDD through the formatting process. Index 10 of that table contains information about the file identified with inode value 10 such as the size of the file in bytes, its file type, and other useful information including the locations on the hard disk drive that contain the data (contents) of the file.

Users do not like numbers (inodes) to reference files. An inode number such as 315156789 is difficult to memorize. Users prefer names. The term filename would then be established.

A filename is a mnemonic name that is associated with a given inode number. The association is effected inside a directory and the pair (filename,inode number) is recorded in the data area of the directory, i.e. its data contents. Be reminded that a directory is a file of type directory. In a given directory there can be only one pair with a given filename and given inode number (filename, inode number). Different directories can however contain the same pair: say dir1 and dir2 both contain(filename, inode number). Furthermore, it is possible that in the same directory we have another association of the the same inode number but with a different file name such as (filename2, inode number). In other words the same file (inodenumber) has two distinct names (filename and filename2) for a total of three aliases (filename appears in dir1 and dir2). We call these aliases links or hard links. The latter term hard link is to distinguish from the filetype symbolic link (also known as soft link)!

We furthermore prefer to say that a directory contains a filename rather than it contains a filename and inode number association. And sometimes two different filenames in the such directory such as filename and filename2 map to the same file of the same HDD.

In Unix and Linux we create a regular file by typing

```
% touch myfile
```

Several things happen with this 'command'

(a) A file in the HDD is created by assigning a currently available inode number say N to the file that is to be established and the space in the inode table index N is initialized appropriately, for example setting the type of the new file to – (regular file), and

(b) an entry is made in the directory into which the command touch was typed establishing the relationship (myfile,N).

Subsequently the file creation process of myfile is continued and gets completed.(Note that the size of the created file would be zero bytes.)

Implicit in all this discussion is the fact that we, the user, know where we are in the hierarchical structure of the filesystem with which we interact. Thus "in the directory into which the command was typed" needs some explanation.

When a user logs on the system the location of the user in the filesystem is the user's home directory. The user can identify this location in two diffrent ways either by typing

```
% pwd
```

for print working directory (which immediately after login is the user's home directory) or by typing

```
% echo ~
```

where tilde ~ is an alias for the user's home directory location. The command echo prints the value of the cryptic tilde. Tilde is an alias to 'current user's home directory'. If a user is lost in the filesystem hierarchy a user can do a

```
% cd ~
```

and this moves him to the user's home directory, the starting location immediately after login. The command cd ~ literally means change (the current) directory to become the home directory (of mine).

The hierarchical top of a Unix or Linux filesystem is **the root of the filesystem**. It is depicted by a slash symbol / and it is a directory, i.e. a file of type directory. It can contain files of any type including directories. The latter can be referred to as subdirectories since they are subordinate to the root directory /. A parent-child hierarchical relationship can then be established. The root is the parent of its subdirectories, and the subdirectories are the root's children.

Every file in the filesystem is associated with **an absolute path** that describes its location in the hierarchy relative to the root /, the common ancestor of all files (and of all types of files) in existence in the filesystem.

Thus the file with name filename might be associated with the absolute path

/afs/cad.njit.edu/u/u/s/user5/filename

This is to be read as follows.

(a) start with the root / directory and read its data, and locate in the data of directory / a filename and inode number association for a file named afs,

(b) then use the inode number of filename afs, to retrieve information about the file, confirm it is a directory (type) and access its data by reading its data contents and locating in it a filename and inode number association for a file named cad.njit.edu,
(c) then use the inode number of filename cad.njit.edu, to retrieve information about the file, confirm it is a directory (type) and access its data by reading its data contents and locating in it a filename and inode number association for a file named u,
(d) do the same for u and its file of type directory also named u, and
(e) do the same for u and its file of type directory s, and
(f) do the same for s and its file of type directory user5, and
(g) in directory user5 find the file named filename through the association (filename,inode number). This is the file in question. The inode of filename say 315156789 allows us to determine the type of file with inode 315156789 by going to the inode table and retrieving information about index 315156789.

The absolute path also provides us with some other hierarchical information. For example a 'child' of user5 is filename, or the parent of filename is user5. The parent of user5 is s. The parent of a file is always a directory that contains the file. By the way, the parent of the root / is the root itself, a directory. The root / is the only element of the hierarchy that is the parent of itself! A file with no children is a non-directory file (a file of a type other than directory) or an empty directory (without files i.e. children).

In the absolute path we observe two usages of the slash symbol.

The slash symbol is being used to denote the root (directory) of the file system. Subsequently the slash symbol is being used to separate directories (and arbitrary files) in the absolute path of a file. The absence of a slash symbol at the end of the absolute path for filename also indicates that filename is a file of type OTHER than directory. If it was a directory a slash would have been the last character of the path. (But different programs/commands use this inconsistently.)

A relative path can also describe a file such as the one known as filename. First by using the command cd (change directory) we move ourselves (the logged on user) to a specific (directory)

location in the hierarchical structure of the filesystem. For example,

```
% cd /afs/cad.njit.edu/u/u/s/user5/
```

We can the confirm the current location with the command pwd (print working directory)

```
% pwd
```

```
/afs/cad.njit.edu/u/u/s/user5
```

and then file inquiries are relative to this directory

```
% ls  filename
```

is then equivalent to an

```
% ls /afs/cad.njit.edu/u/u/s/user5/filename
```

in other words we are looking for information on filename in directory user5.

The current directory is denoted by (or aliased to) a dot. Thus

```
% ls .
```

and

```
% ls /afs/cad.njit.edu/u/u/s/user5/
```

are equivalent. The parent of the current directory is denoted by (or aliased to) two dots (no space in between) Thus

```
% ls ..
```

is equivalent to

```
% ls /afs/cad.njit.edu/u/u/s/
```

And of course

```
% cd .
```

has no effect as we request that we move to the current directory even if we are in it already. Note that relative paths are allowed when we use cd. Thus

```
% cd  ..
```

moves to the parent of the current directory thus

```
% pwd
```

prints

/afs/cad.njit.edu/u/u/s

and then a relative

% cd user5

moves us back to the original location.

If your myUCID is user5, the OS the moment you login into a remote host and the shell starts running, it automatically does (by itself) a

% cd /afs/cad.njit.edu/u/u/s/user5/

for you. The indicated path (directory) is your home directory. The tilde symbol ~ is aliased to your home directory: it saves time typing it!

But beware of the following nuance.

Your home directory is not user5.

Your home directory is user5 of directory s of directory u of directory u of directory cad.njit.edu of directory afs of the root file system!

This is because it is possible that there are multiple user5 directories elsewhere in the file system hierarchy for example

/usr/local/user5/

/user5/

/bin/user5/

/user/local/bin/user5

/afs/cad.njit.edu/u/u/s/user5/user5/my.txt

In the last line above you might have observed that the home directory contains a (sub) directory user5 that contains a file named my.txt! (And the ! is an exclamation mark not part of the file name!)

In order to find information about the file with name filename we can type

```
% ls -l filename
```

and the output might look like as follows.

```
-rw-r--rw- 1 user5 group 1178078 Apr 26 12:14 filename
```

If we type the following the inode number of filename is also output.

```
% ls -li filename
```

```
315156789 -rw-r----- 1 user5 group 1178078 Apr 26 12:14 filename
```

If you want information about the inode number 315156789 stored in the inode table with index 315156789, this can be obtained through filename as follows.

```
% stat filename
```

The -l or -li is an option: the dash - alerts the operating system's shell that an option would be presented, and the ell indicates the long option (details about filanme). The left most character of the output of ls -l is the dash (left of the rw) that indicates that the file type is a (regular) file. When -li is typed two options follow the dash symbol one after the other with out space in between: the l option and the i option indicating a request to obtain the inode number associated with filename. The order does not ordinarily matter: we might have typed -il.

The nine character rw-r----- are the three triads that describe permissions for three entities associated with filename: the user owner of the file, the group of the user owner of the file, and everybody else. We refer to these entities generically as u,g, and o respectively. In this example u is user5, g is the group named group, and o everybody else i.e. neither user5 nor anyone in the group named group. The permissions assigned by a triad to an entity are read (r), write (w), and execute(x) in the presence of the corresponding letter and are positionally dependent (r on the left of w on the left of x). The dash indicates the absence of the corresponding privilege/perfmission for the corresponding entity for filename. Thus user5 has r and w privileges it can read and edit (write) the file named filename. The group (users other than user5 of the group group) has only r privileges, and everybody else has none.

Observe another output below.

```
% ls -ld 2021linux
```

drwxrwxrwx 3 user5 afs 2048 Oct 14 2021 2021linux

It is clear that 2021linux is a file of type directory. Note that the term folder is being used in Windows as an alternative to the term directory. This is not the case in Unix. Do not use the term folder in Unix (including Linux). There is no file of type folder. In fact in Windows a folder can be a non-directory structure. The meaning of privilege x for a directory is different from that a file. All (user5, afs, and everybody else) are allowed to cd into directory 2021linux because of the x privilege. All can delete the directory or write into it i.e. create files (of any type) or delete files. Thus the dangerous

% rm –rf 2021linux

if executed would delete everything in 2021linux including directory 2021linux and all of its file recursively and completely. The OS won't ask you to confirm your recklessness. You explicitly specified f as an option in –rf to indicate "don't ask". Moreover the r of –rf is 'recursively'!

Directory user5 contains in the data of file/directory user5 a line

filename 315156789.

Thus the only way to find the alias to 315156789 is by looking inside the directory that contains the relationship between the alias (filename) and the actual name (inode) of the file (315156789). A file (such as 315156789) can have multiple aliases known as hard links. Thus it is possible inside user5 to have another entry

myfile   315156789

In fact somewhere else (in some other directory) it is possible to have the same. However we do know that this is not the case: this is because in the output of ls –l filename on the left side of user5 we see a 1. The 1 indicates one alias exists for 315156789 and that is filename. The operating system keeps track of all the associations with file 315156789.

If you do however a

```
%  ln  filename  myfile
```

and then do a

```
% ls -l filename
```

or an

```
% ls -l myfile
```

or

```
% ls -li filename
```

or

```
% ls -li myfile
```

things would become interesting.

Below we use the symbol sharp #. The # indicates a comment for the shell and thus the remainder of the line is ignored by the shell when it tries to interpret and execute the line.

The commands (in fact executable files) we introduced that manipulate files of a filesystem or traverse the hierarchy of a filesystem are as follows.

```
% pwd          #print current working directory
% cd path      # change the current working directory to path
               # path name path can be an absolute path name
               # starting with the root / of the filesystem or
               # a relative path name and starts with a file of
               # current directory
% ls path      # list contents of directory path
% ls file      # list contents of directory file is
               # directory or confirms file exists otherwise
% ls           # list contents of current working directory
% ls .         # equivalent to  ls ; same as above
% ls -l        # long list ; equivalent to ls -l .
% ls -la       # detailed long list including . and ..
% ls -lai      # the inode (numeric name/ID of the file) is also
               # listed on the extreme left side.
% rm file      # delete a file
```

or

```
% rm pathTofile
```

but

```
% rmdir dir          # remove an empty directory dir
% rmdir path2dir     # remove an empty directory described in
                     # the path2dir
% rm -rf file        # A pretty dangerous command... Avoid it.
% rm -rf directory   # A pretty dangerous command... Avoid it.
% mkdir directory    # Create a directory
% mv old new         # rename filename old into new
```

4. Connecting to a remote host

Connecting to a remote host (host.njit.edu). The remote host has a user with login name myUCID whose credentials are known to you (e.g. you identify yourslelf as myUCID on host.njit.edu). Open a window on the client (eg through MobaXterm, see Figure 2) and type in the command (as explained earlier).

```
% ssh myUCID@host.njit.edu
```

Transferring (copying) a file from the local (client) host to the remote host. File local.tar is in the current directory of the client. It will be copied to the remot host in the directory cs332 residing inside the home directory of the remote host. Note that we do not write the home directory (absolute path) of user myUCID in the remote host but we use the tilde alias instead followed by directory cs332 that already exists in the home directory. In another MobaXterm window on the client (see Figure 2) type instead

```
% scp local.tar myUCID@host.njit.edu:~/cs332
```

The command scp stands for secure copy. The first argument is the local file name. The second argument describes first the details of the remote machine: first the login name of a the user in question in the remote machine then the at sign (@) followed by the host name of the remote sign followed by a colon(:) and after that the absolute path to the directory into which the file will get copied. If a regular file is specified the remote file will be overwritten by the client file (local.tar). Authentication will be requested by the remote host in every invocation of scp. The command scp stands for secure copy. When the command completes the transfer of the file, the prompt reappears on the client host machine and the host machine is read to receive more input. Note that MobaXterm allows for the same operation (upload) to be performed through a graphical interface. This is described in document **AVG-24-01r2.**

5. Shell interaction

The moment we login to the remote host a shell starts running. The default shell used in this discussion is the bash shell. The shell expects input from the remote use interacting through the client machine's keyboard and screen (terminal display). The remote host shell's declares its existence by a prompt that is user (remote user myUCID) defined. The remote user myUCID has chosen the prompt to be the percent sign (%).

```
%
```

Every time a user types a command, the user must end the command by pressing the ENTER key. Then and only then will the typed text transmitted to the shell of the remote host and interpreted by it and then acted upon its interpretation. The remote host will create a process to realize the command requrest typed in by the user of the client machine.

Options and arguments.

In the example(s) below the command is ls (list).

The argument is a filename. It might refer to a file of any type including a directory or a (regular) file. It might be an alias such as dot (.) or dot dot (..) or ~ for the current directory, parent directory or home directory respectively. The option specified is i to indicate that the inode (number) associated with the filename is to be output. Naturally a - to indicate an option preceded the option itself (no space between the - and the option). An alternative is to specify the long name of the option. For option i the long name is inode. In the latter case two dashes are to precede the long name of the option. The two invocations below are equivalent

```
% ls -i filename
   315204012 filename
```

```
% ls --inode filename
   315204012 filename
```

Some additional commands include the following.

```
man  cmd          # print manual page of  cmd
info cmd          # more elaborate version of man
man  man          # man on cmd man
help              # inside bash a help re bash
who               # users in system
whoami            # login name of user
hostname          # hostname of computer
date              # current date and time information
echo              # print on standard output (terminal screen)
echo ~            # print home directory of user
echo $EDITOR      # default editor (env var $EDITOR)
printenv          # all environment(shell's) variables+values
echo $(cmd)       # print output of cmd
echo $$           # print process ID of current process
                  # which is by the way the bash shell
echo $?           # print return value of last executed cmd
echo $0           # print name of shell (bash)
echo $-           # detech interactive shell
bash              # run an instance of bash within bash!
exit              # exit bash
exit  n           # exit bash with return value integer n
cal               # calendar
var = "value"     # variable var defined and assigned a value
echo $var         # print value of variable var
export var        # make it visible to other
                  # programs/processes (use reference $var)
 #ls              # is the comment character ; rest is ignored
```

```
% whoami
myucid
```

```
% hostname
host.njit.edu
```

```
% date
Tue Nov 14 12:00:08 EST 2023
```

```
% echo “hello world”
hello world
```

```
% echo ~
/afs/cad/u/m/y/myucid
```

```
% echo $EDITOR
/bin/vi
```

```
% echo $(date)
Tue Nov 14 12:12:47 EST 2023
```

```
% var=”this is a value”
% echo $var
this is a value
% export var
```

```
% cal
    November 2023
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

```
% echo $$
11903
% bash
% echo $$
12789
% exit 123
exit
% echo $?
123
% echo $$
11903
```

If in doubt

```
% man command
```

provides a manual page for a given command.

COMMAND line history

```
history                 # history of user commands ; each prior
                        # command is numbered
                        # (higher numbered  more recent)
!50                     # execute command number 50
!!                      # redo last command
UP arrow key            # move UP in history
DOWN arrow key          # move down in history (if possible)
<Backspace>             # delete a character from command
CTRL-A                  # move to the start of command in line
CTRL-E                  # move to the end of command in line
CTRL-B CTRL-F           # move to previous , to next word resp.
CTRL-K                  # delete/kill text from cursor position to
                        # the end of line
CTRL-U                  # cuts to start of line
CTRL-Y                  # yank/paste killed text
CTRL-L                  # clear (terminal) screen
CTRL-P                  # list / browse previous commands
TAB                     # autocomplete command or file name
!cmd                    # recall and execute more recent command
                        # starting with text cmd
```

COMMAND OPTIONS

[applicable to man/less/more]

```
Space               # move forward by one page
f                   # same as Space
b                   # move backwards by one page
<                   # go to first line
>                   # go to last line
/                   # search forward
                    #    /string   searches for string
?                   # search backward
n                   # repeats /
N                   # repeats ?
h                   # help
q                   # quit help
q                   # quit man (outside of help) or less or more
```

6. Filesystem commands

A Unix or Linux filesystem (generic reference) is a rooted acyclic directed graph that sometimes it is erroneously depicted as a tree. The usual view is with the root of the filesystem at the top (rather than at the bottom of a biological tree).

The filesystem contains files of different types.

A type of a file is a regular file. We may omit regular from the file type description.

A type of a file is a directory. A directory is a file that contains other files of any type including of course directories. The latter ones are sometimes called subdirectories i.e. a directory that is subordinate to the directory into which it was created.

A folder sometimes is used as an alternative to the word directory in operating systems other than Unix or Linux. This is to be avoided in Unix and Linux. In Windows folder refers to a directory but also to a collection of information that is not mapped to a file of type directory.

There is a parent child relationship between directories.

If inside directory dir1 a directory dir2 is created, dir1 is the parent and dir2 is the child of dir1. Directory dir2 is subordinate to dir1 i.e. it is a subdirectory of dir1.

The root of a filesystem is a directory. We refer to it as the root directory as well or the root of the given file system. The root directory is named / (slash or forward slash). The parent of the root (directory) is the root (directory). It is the only directory whose parent is itself!

In the root structure of a Unix or Linux filesystem a node that has children is a directory and sometimes it is called an internal node.

A node with no children can be an empty directory or a file of type other than directory. It is sometimes called an external node or a leaf.

The directory view below was created using a command available in a posting of stackoverflow. The command is

```
find example | sed -e "s/[^-][^\/]*\// |/g" -e "s/|\([^ ]\)/|-\1/"
```

```
example
 |-dir1
 | |-file1
 | |-dir3
 | |-dir2
 |-dir2
 |-file1
```

Directory example has files dir1, dir2, file1

dir1 is an internal node

dir2 is an external node (leaf);

it is an empty directory

file1 is an external node (leaf); it is a regular file

Directory dir1 of examples has files dir2, dir3, file1.

file1 is an external node and a regular file

dir2 is an external node and an empty directory

dir3 is an external node and an empty directory

The name of a file should never be used to determine if two file are the same or not. The inode number should be used instead. For the sake of an example, example/dir2 has inode number 315165133 and example/dir1/dir2 has inode number 315165139. Those two files are distinct from each other; they are empty directories.

Likewise example/file1 has inode number 315211770 and example/dir1/file1 has inode number 315211796. They are different, yet they are empty files.

```
% ls -lRi example/
example/:
total 4
315165127 drwx------ 4 alexg users 2048 Nov 14 13:20
dir1
315165133 drwx------ 2 alexg users 2048 Nov 14 13:19
dir2
315211770 -rw------- 1 alexg users    0 Nov 14 13:19
file1

example/dir1:
total 4
315165139 drwx------ 2 alexg users 2048 Nov 14 13:20
dir2
315165135 drwx------ 2 alexg users 2048 Nov 14 13:19
dir3
315211796 -rw------- 1 alexg users    0 Nov 14 13:19
file1

example/dir1/dir2:
total 0

example/dir1/dir3:
total 0

example/dir2:
total 0total 0
%
```

Directory example contains files dir1, dir2 and file1.

Files dir2 and file1 are external nodes (leaves)

FILESYSTEM NAVIGATION

Absolute path names An absolute path name identifies a file starting with the root of the filesystem.

Relative path names A relative path name identifies a file relative to the current working directory.

Special characters.

~	: Home directory
.	: (dot) is current directory
..	: (dot)(dot) is parent directory
*	: (star) matches any filename of any file
?	: matches any one character
abcTAB	: abcTAB will attempt to match to a file whose filename starts with abcd; note that TAB is the Tab character not T A B.

Example. For the directory example used earlier the absolute path name of dir2 within dir1 within example is

/afs/cad/u/m/y/myucid/example/dir1/dir2

We then move (cd ~/example) to directory example whose absolute path name is

/afs/cad/u/m/y/myucid/example

We can reach the same location (of dir2 in dir1 in example) afterwards with a cd command that described a relative path (relative to the current working directory that currently is /afs/cad/u/m/y/myucid/example) Thus a

cd dir1/dir2

brings us back to our original location and thus a pwd

with output

/afs/cad/u/m/y/myucid/example/dir1/dir2

confirms that!

Moreover a cd ../.. brings us back two levels up!

```
% pwd
/afs/cad/u/m/y/myucid/example/dir1/dir2
% cd ~/example
% pwd
/afs/cad/u/m/y/myucid/example
% cd dir1/dir2
% pwd
/afs/cad/u/m/y/myucid/example/dir1/dir2
% cd ../..
% pwd
/afs/cad/u/m/y/myucid/example
```

A file system navigation command summary is available below. Several of the commands can accept more than one argument that is a file. Thus ls file1 can become ls file1 file2 file3 file4.

```
% pwd                  # print working directory
% man pwd              # manual page for pwd
% man man              # manual page for man
% cd  path             # change to directory path
                       # path is an absolute path name
                       #  or a relative path name
% cd  /                # go to the root of the file system
% cd                   # go to the home directory of user.
% cd ~                 # same as cd
% cd  .                # go the current directory
                       # no effect already there
% cd ..                # go one directory up
                       # to parent directory
% cd ../..             # go two levels up (grandaparent)
```

FILE VIEW

```
% ls .                  # list contents of current dir.
% ls                    # same effect as ls
% ls path               # list contents if path is
                        # directory or confirm file exists
                        # otherwise
% ls directory          # list directory's files
% ls file               # confirm file exists
% ls ~                  # list files at home directory
% ls .                  # same as ls
% ls ..                 # ls of parent directory
% ls -l                 # detailed  long listing
% ls -d                 # list directories not their
                        # contents
% ls -a                 # show hidden files as well
                        # hidden files start with .
% ls -F                 # put a / at end of directory name
% ls -li                # inode information included
% ls -lia               # hidden files included
% ls -liaR              # also recursively follow
% ls file*              # list files that start with file
% ls *file              # list files that end   with file
% ls file{1,2}          # list file1 and/or file2
% ls file[0-9].txt      # list files file0.txt,…,file9.txt
% ls -d dir?            # list directories that start with
                        # dir and have one additional char
                        # in the filename
% ls -d *1              # list files ending with 1
% ls -lS                # sort by file size
% ls -lt                # sort by modication time/date
```

FILE INFO

```
% stat  filename       # info about file
% ls -l filename       # info about file
% file file1           # type of file1
```

FILE MANAGEMENT

CREATE FILES or DIRECTORIES

```
% touch file1              # creates regular file
                           # with 0 size of name file1
                           # if file1 exists updates
                           # modification time
% mkdir dir1               # creates an empty directory
                           # named dir1 inside current
                           # directory
```

COPY or MOVE FILES or DIRECTORIES

```
% cp file1 file2           # makes a copy of file1;
                           # new file named file2
% cp -r dir1 dir2          # copies a full directory

% mv dir1 dir2             # rename dir1 into dir2
% mv file1 file2           # rename file1 into file2
% mv file1 dir2            # mv file1 into dir2
```

DELETE FILES or DIRECTORIES

```
% rm filename              # delete filename in directory
                           # name disappears from current
                           # directory that contains
                           # the command does NOT ERASE
                           # the contents of the file
% shred filename           # contents erased if on HDD;
                           # file not deleted,
% shred -u filename        # contents erased if on HDD;
                           # and file deleted.
% rmdir dir1               # deletes(removes) empty
                           # directory dir1
                           # if not empty delete its
                           # files first or see below!
```

DANGEROUS COMMANDS!!

```
% rm -f file1              # silent removal of file1 !!!
% rm -rf dir1              # silent removal of all files
                           # in directory dir1 including
                           # files in subdirectories of
                           # it
```

(HARD) LINK to a file

Linux and Unix Definition of file equlity.

Two files are identical if and only ifthey have the same inode#. This means the same location of the filesystem stores information about the two 'files' and this is the inode table at offset the inode#. Moreover the contents are also the same.

What makes a file a file is the inode# not the filename. The filename is an alias to the inode#; humans cannot memorize long integers they prefer (English) words that are shorter.

Hard link creation syntax

```
% ln existingfile newhardlink
```

Semantics

Precondition: An existing file with alias existingfile that maps to a file of the filesystem with inode# (inode number) N.

There are no other aliases to inode# N other than existingfile. This can be confirmed by examining the inodetable entry for N and confirming that the (hard) link count is a 1.

Postcondition: A new alias newhardlink is created for the file with inode# N. Thus in the same directory we have two entries

(existingfile, N)

and

(newhardlink, N)

both pointing to the same file with inode# N. Thus there are two files with different file names that are identical in the sense that their inode# are the same. They both point to the same location of the inode table and by way of the latter to the same sectors of the disk that store the date (contents) of the file itself (and not the files themselves). The inode table stores an inode (hard link) count. It used to be 1 but after the command ln successfully completed is now a 2. Command stat can confirm it.

Sometimes one can call the inode# namefile to distinguish it from the two filenames that map to it! What makes a file unique is its inode number. A file with inode number 315211796 can exist under multiple filename. See an example below.

(Step 1) We create an empty file named a.

(Step 2) We add some content to a

(Step 3) The current directory into which a created has absolute path

/afs/cad/u/m/y/myucid/example

(Step 4) Creating a added into the contents of directory example a pair

(a, 315211796)

This establishes the name a as an alias to

the file with inode number 315211796.

The hard disk drive only know file 315211796

and so does the filesystem

(Step 5) We generate additional pairs i.e. aliases

(b, 315211796)

(c, 315211796)

(d, 315211796)

There is no difference in doing ln a c vs ln b d.

Both aliases c and d are to the same inode number as

a and b map to inode number 315211796.

(Step 6) We verify that all a,b,c, are the same with a

```
% ls -i
```

or

```
% ls -I a b c d
```

and the hardlink count of 315211796 is 4 to match the number of aliases (a,b,c,d) as it can be revealed either by executing a stat command (eg stat a) or an ls -li command as shown below for both cases.

HARD LINK EXAMPLE SETUP

```
% touch a                    # create file a of size 0
% echo "Hello world" >>a     # add some content into a
% gvim a                     # edit a to add content as
% vi a                       # well (vi vs gvim)
                             # use vi if you know how!
% ls -i a                    # inode# of a is 315211796
315211796 a
% ln a b                     # create a hard link b of a
% ln a c                     # create a hard link c of a
% ln b d                     # create a hard link d of b
% ls -i
315211796 a  315211796 b  315211796 c  315211796 d
% cat a
Hello world
% cat b
Hello world
% stat a
  File: 'a'
  Size: 12              Blocks: 2          IO Block: 4096   regular
file
Device: 28h/40d Inode: 315211796   Links: 4
Access: (0600/-rw-------)  Uid: (32252/   myucid)   Gid: (  100/
users)
Access: 2023-11-14 15:26:35.000000000 -0500
Modify: 2023-11-14 15:26:35.000000001 -0500
Change: 2023-11-14 15:26:35.000000000 -0500
 Birth: -
% ls -li
total 4
315211796 -rw------- 4 myucid users 12 Nov 14 15:26 a
315211796 -rw------- 4 myucid users 12 Nov 14 15:26 b
315211796 -rw------- 4 myucid users 12 Nov 14 15:26 c
315211796 -rw------- 4 myucid users 12 Nov 14 15:26 d
```

We then start a systematic delete of some of the aliases also called colloquially 'files'.

```
% rm a                          # alias a is gone
% ls -li
total 3
315211796 -rw------- 3 myucid users 12 Nov 14 15:26 b
315211796 -rw------- 3 myucid users 12 Nov 14 15:26 c
315211796 -rw------- 3 myucid users 12 Nov 14 15:26 d
% cat b
Hello world
% cat c
Hello world
% rm c
% rm d
% ls -li
total 1
315211796 -rw------- 3 myucid users 12 Nov 14 15:26 b
% cat b
Hello world
% cat a
cat: a: No such file or directory
```

Deleting a file using an alias such as a, c, and d has no effect on the inode# associated with a, c and d while the hard link count is greater than 0.

After the rm a, the hard link count of inode# 315211796 went down from 4 into a 3. After them rm b it went further down into a 2. And after the third rm involving d it went down into 1.

The OS DOES NOT USE the filename to access the file itself. It uses the inode# associated with the filename. Even if we cannot use the name (alias) a,c,d to access it we can still use the remaining name (alias) b to do so and we did so.

If we remove a file with hard link count 1, first the filename and inode# association (eg (b, 315211796)) gets wiped out from the directory in which it was established, then the hard link count is decremented by one to reach a 0 and when this link count of 0 is detected then and only then the inode number is freed from the inode table (become available for reassignment) and the file is thus 'removed' from the table. The file is not deleted in the sense that the contents are still etched on a sector of the hard disk drive. The rm command does not touch or access the data of the file, but only the METADATA that are

available in the inode table entry of offset (or index or number) 315211796.

Conclusion: rm removes only METADA of its file argument(s) and DOES NOT ERASE the DATA of the respective file arguments(s).

SYMBOLIC LINKS to a file

Symbolic link (or equivalently soft link) creation syntax

% ln -s existingfile newsoftlink

Semantics

Precondition: An existing file with alias existingfile that maps to a file of the filesystem with inode# (inode number) N.

There are no aliases to inode# N other than existingfile. This can be confirmed by examining the inodetable entry for N and confirming that the (hard) link count is a 1.

Postcondition: A new file is created and thus a new entry in the inode table is to be used up. Let the new file has inode# M. The type of the file would be flagged as a symbolic link (and not a regular file). Its hard link count would be 1.

In the current directory a pair

(newsoftlink, M)

would be created.

The data contents of the file with inode# M would be

newsoftlink -> existingfile

If we attempt to perform a

cat newsoftlink

the OS realizes that the newsoftlink is a symbolic link,

it find from the contents newsoftlink -> existingfile

of newsoftlink that it needs to read file existingfile and provided existing file still exists its contents would be displayed through the cata command.

We provide a comprehensive example below

(Step 1) Using the leftover file b of the hard link example we rename b into a (mv b a) and thus our original set up exists.

(Step 2) We issue a

ln -s a newsoftlink

(Step 3) and then we perform a

ls -li

to realize that a and newsoftlink have different inode numbers.

(Step 4) A

```
cat a
```

or

```
cat newsoftlink
```

displays the same contents.

(Step 6) We then issue a

```
rm a
```

The contents are destroyed and the

```
newsoftlink -> a cannot be resolved
```

as file a with inode# 315211796 is gone (and the inode number 315211796 has been released to the filesystem for future usage/allocation.

SYMBOLIC LINK EXAMPLE SETUP

```
% mv b a                    # leftover from hard link
                            # example generates initial
                            # base case as in other
                            # example
% ls -i a                   # inode# of a is 315211796
315211796 a
% cat a
Hello world
% ln -s a newsoftlink
% cat b
Hello world
% ls -li
total 2
315211796 -rw------- 1 myucid users 12 Nov 14 15:26 a
315211770 lrwxr-xr-x 1 myucid users  1 Nov 14 16:27 newsoftlink ->
a
% rm a
rm: remove regular file ‘a’? y
% ls -l
total 1
lrwxr-xr-x 1 myucid users 1 Nov 14 16:27 newsoftlink -> a
% ls -li
total 1
315211770 lrwxr-xr-x 1 myucid users 1 Nov 14 16:27 newsoftlink -> a
% cat newsoftlink
cat: newsoftlink: No such file or directory
% stat newsoftlink
  File: ‘newsoftlink’ -> ‘a’
  Size: 1               Blocks: 2          IO Block: 4096
symbolic link
Device: 28h/40d Inode: 315211770   Links: 1
Access: (0755/lrwxr-xr-x)  Uid: (32252/   myucid)   Gid: (  100/
users)
Access: 2023-11-14 16:27:52.000000000 -0500
Modify: 2023-11-14 16:27:52.000000001 -0500
Change: 2023-11-14 16:27:52.000000000 -0500
 Birth: -
```

We then start a systematic delete of some of the aliases also called colloquially ‘files’.

7. Text processing commands

LISTING of CONTENTS of FILES

```
% cat file1             # catenate (display) contents of
                        # file1
% tac file1             # end to start display (reversal)
% cat file1 file2       # multiple file content display
                        # join two files vertically
% cat file1 file2 >> file3  # add file1 and file 2
                        # contents at end of file3
                        # if file3 does not exist it is
                        # created (contains file1, file2)
% more file1            # controlled display of contents
% less file1            # alternative controlled display
                        # of contents
% head -n N file1       # display first N lines of file1
% tail -n N file1       # display last  N lines of file2
% od -c file1           # display contents in detail
% od -h file1           # in hexadecimal
% od -o file1           # in octal
```

COMMAND OPTIONS for less or more

```
Space                   # move forward by one page
f                       # same as Space
b                       # move backwards by one page
<                       # go to first line
>                       # go to last line
/                       # search forward
                        #    /string   searches for string
?                       # search backward
n                       # repeats /
N                       # repeats ?
h                       # help
q                       # quit help
q                       # quit man (outside of help) or less or more
```

MANIPULATION of FILES

```
% diff file1 file2    # compare two files
% egrep string file1 # find lines of file1 that contain
                      # string and display them
% sort file1          # sort the lines of file1
% sort -r file1       # reverse sort the lines of file1
% sort -nr file1      # treat lines as numbers
% sort -n  file1
% tr l q <file1 >file2 # change l in file1
                      # into q and store result in file2
% wc file1            # word count of file file1
% split file1 -l 3    # split file file into multiple
                      # files 3 lines per file
% md5sum file1        # MD5 fingerprint (crypto) of
                      # file1
```

IDIOSYNCRATIC BEHAVIOR of BASH

```
% date                # explained earlier

% echo \$date         # $ is a special character!
% echo "`date`"      # similar to date
% echo $(date)        # think! also echo $(ls) ?
% sort -nr file1      # treat lines as numbers
% sort -n  file1
% tr l q <file1 >file2 # change l in file1
                      # into q and store result in file2
```

SEARCHING FILES

```
% egrep string file1 # searching the contents of a file

% find ~ -name f      # search home directory for f
% find ~ -name f1     # search home directory for
directory f1!
% find ~ -name f -print # also print location found
% find ~ -name `*.txt'  # for all .txt file
```

8. FILE PERMISSIONS

In Unix and by extension to Linux access permissions are assigned to every file of the filesystem.

This corresponds to traditional file access permisions. Moden version of Unix and Linux support POSIX permissions. In addition, NJIT's AFS supports AFS permissions as well. Whereas traditional file permissions are used for non-directory file types, AFS permissions are used for files of type directory (i.e directories).

In this section we discuss only traditional file access permissions. We might also call them access privileges.

How to find out the (current) access permissions of a file? One way to obtain them is to use the command ls -l followed by the name of the file in question. An alternative is to use the command stat followed by the file name (see earlier sections).

OBTAININING PERMISIONS

```
% ls -l  filename     # obtain file permisions
% stat   filename     # alternative approach
```

Example

```
% ls -l fname
-rwxr-xr-- 1 myucid users 0 Nov 16 10:05 fname
```

In this example we obtain the access privileges of file fname. Note that the file is of type regular file and this is indicated by the -, the leftmost character of the output.

In the output of ls -l the privileges start from the second and extend to the 10th character of the output. They are rwxr-xr-- and they describe access privileges for three entities. We rewrite them below by adding some space to distinguish the three entity separate access privileges.

```
  entity u   entity g   entity o    #privileges
  rwx        r-x        r--
   u triad    g triad    o triad

u:  user owner of file                # myucid of numericID 32252
g:  group of user owner of file       # users of numericID 100
o:  others (neither entity u nor users of entity g)

Positional importance
r is always the leftmost of three privileges in a triad
absence of r privilege is indicate by a dash in leftmost position

w is always th  middlemost of three privileges in a triad
absence of w privilege is indicated by a dash in middlemost position

x is always th  rightmost of three privileges in a triad
absence of x privilege is indicated by a dash in rightmost position
```

The left-most group (left triad i.e. triplet of characters) describes the access privileges assigned to entity **u**. Entity u

is the user owner of the file. In this example the user owner of the file is the user with userID myucid.

The middle group (middle triad) describes the access privileges assigned to entity g. Entity g is the group of the user owner of the file. A group is a collection of userID. In this example the group of the user owner of the file is the group with groupID users.

Finally the right-most group (right triad) describes the access privileges assigned to entity o. Entity o is the Other users. Other means other than the user-owner of the file and the collection of users that form the group of the user owner of the file. Thus in this example entity o describes all the users other than myucid and the collection of user belonging to group users.

Both the userID and the groupID are identified in the output of ls -l, but also the output of the stat command and of course through the command id. Both the numeric ID and the symbolic ID are shown for some of thos commands. Thus the user owner has symbolic ID the noted myucid; its numeric id is 32252.

```
% ls -l fname
-rwxr-xr-- 1 myucid users 0 Nov 16 10:05 fname
% stat fname
  File: ‘fname’
  Size: 0               Blocks: 0          IO Block:
4096   regular empty file
Device: 27h/39d Inode: 315211770   Links: 1
Access: (0754/-rwxr-xr--)  Uid: (32252/   myucid)
Gid: (  100/   users)
Access: 2023-11-16 10:05:32.000000000 -0500
Modify: 2023-11-16 10:05:32.000000001 -0500
Change: 2023-11-16 10:05:32.000000000 -0500
 Birth: -
% id
uid=32252(myucid) gid=100(users)
groups=100(users),1096151802
```

(a) Permissions of user owner (entity u)

The permissions of the user owner of the file are described positionally through the triad rwx. User owner is the user with userID mycuid (numeric value 32252).

For a file of type (regular) file or in general a non-directory file the triad rwx has the following meaning.

rwx : r stands for read privileges are allowed,

w stands for write privileges are allowed, and

x stands for execute privileges are allowed.

For a directory, and fname is a regular file and not a directory, the read, write and execute privileges are interpreted as follows.

read (r) means one is able to read the directory and list its files.

write (w) means one is able to create or remove files from the directory.

execute (x) means that one is able to search the directory and its (subordinate) directories inside it.

(b) Permissions of group owner (entity g)

The permissions of the group owner of the file are described positional through the triad r-x. Group owner is the group with groupID users (numeric value 100).

The **r-x** indicates again that entity g (group owner) has read and execute access privileges on fname. The absence of write privileges is indicated by the w being replaced with a dash(-).

(c) Permission of other users (entity o)

The permissions of entity o are described positional through the triad r--.

The r-- indicates again that entity o has read but neither write nor execute access privileges on fname. The absence of

write and execute privileges is indicated by both the w and x being replaced with a dash(-).

(d) Numerical view of triad privileges

The character view of the access privileges of a triad (rwx, r-x and r-- of our examples) can be converted into an octal digit (0-7 range) in two possible ways

[i] Binary number view

The presence of a r,w,x is mapped onto a 1; the presence of a dash(-) is mapper onto a 0. Then the three-bit binary number is converted into denary (base-10, radix-10) or octal (radix-8).

Example: rwx becomes 111 that is 7 in denary or octal.

r-x becomes 101 that is 5 in denary or octal

r—- becomes 100 that is 4 in denary or octal.

[ii] Denary number view

The presence of a r is mapped onto a 4; a w is mapped onto a 2 and an x is mapped onto a 1. A dash(-) is mapped onto a zero. The three numeric values generated for a triad are then added up to generate a digit from 0 through 7.

Example rwx maps onto 4,2,1 and the sum 4+2+1 gives a 7.

r-x maps onto 4,0,1 and the sum 4+0+1 gives a 5.

r-- maps onto 4,0,0 and ths sum 4+0+0 gives a 4.

Changing permissions

```
% chmod ent=lperm  filename    # set permissions
% chmod ent+lperm  filename    # add   permisions
% chmod ent-lperm  filename    # remove permissions
                               # to/from file filename
% chmod nperm      filename    # set a new set of
                               # permissions to filename

ent is an entity such as  u, g, o or
    entity a to indicate all and equivalent to ugo, or
    a combination of u,g,o such  as uo, go, ug, etc.
lperm describes letter based permissions
      the new set of permissions (= precedes lperm),
      the added permissions(+ precedes lperm), or
      the to be removed permisions(- precedes lperm)
      lperm can be r,w,x or combination of eg rw,
      rx,etc
nperm are the new permissions for the three entities
      one denary(or octal) digit per entity/triad.
      For example,765 indicates a 7 for entity u (rwx),
      a 6 for entity g (rw-),  and
      a 5 for entity o (r-x)
```

Examples.

```
% ls -l fname
-rwxr-xr-- 1 myucid users 0 Nov 16 10:05 fname
% chmod 500 fname
% ls -l
-r-x------ 1 myucid users 0 Nov 16 10:05 fname
% chmod 755 fname  ; ls -l
-rwxr-xr-x 1 myucid users 0 Nov 16 10:05 fname
% chmod g+w fname  ; ls -l
-rwxrwxr-x 1 myucid users 0 Nov 16 10:05 fname
% chmod a-x fname  ; ls -l
-rw-rw-r-- 1 myucid users 0 Nov 16 10:05 fname
```

AFS Permissions

We only provide some elementary information about AFS permissions. The AFS access privileges are known as ACL (access control list).

ACLs apply to directories and not to files. It ignores standard Unix (Linux) permissions. A directory created inherits by default the ACLs of its parent directory. (If the parent directory's ACLS are subsequently changed the new changes do not progagate to subdirectories.)

One can access a subdirectory if ACL privilege l (lookup) is available to all parent directories.

ACL permission are normal or negative. A normal permission grants a privilege; a negative removes a privilege.

Thus if an r permission is both granted and negated, the system first includes r when it examine the normal permissions, but when later on examine the negative permission r is removed from the available permission list.

ACL	**Permissions**
DIRECTORY PERMISSIONS	
l (lookup)	listing of directory contents allowed
i (insert)	creating new file in directory allowed (copying of files in directory allowed).
d (delete)	removal of files from directory allowed
a (admin)	allows one to change a directory's ACL (other than the owner of the directory who can do so by default or member of system:administrators)
FILE PERMISSIONS	
r (read)	read contents of files in directory and issue ls -l
w (write)	modify contents of files in direcoty and allowed to use chmod on those files
k (lock)	allows program to lock files in directoy
UNDEFINED MEANING PERMISSIONS :A B C D E F G H	
SHORTCUTS	**Meaning**
all	rlidwka
read	rl
write	rlidwk
none	remove all assigned privileges

Command	Explanation
fs listacl dir1	list permission of directory dir1
fs la dir1	shortcut for listacl is la

Assistance

```
fs help listacl
```

Example

```
%  fs la example
Access list for example is
Normal rights:
  arcsnewstaff rlidwk
  homer l
  system:administrators rlidwka
  myucid rlidwka
  http l
```

SPECIAL USER (groups)

system:anyuser	Must never have write privileges
system:authuser	Some systems differentiate by providing defining such a group (eg NJIT users)
system:administrators	

Command	Explanation
fs setacl -dir dirname -acl aclentries	Add to an ACL
fs sa -dir dirname -acl aclentries	
fs sa -dir dirname -acl aclentries -negative	
fs sa dirname user permisions	aclentries is a user permissions pair
	Remove from an ACL
fs sa dirname aclentries	Must use order as above
fs sa dirname user none	Negate privileges for user user
fsr sa -dir dname -acl aclentries	Recurive set on all subdirectories of dname
fs sa -dir dname system:anyuser none	
fs sa -dir dname myucid all -clear	Remove all other users except mycid

Assistance

```
fs help setacl
```

Example

```
%  fs sa -dir . -acl homer w   or fs sa . homer w
Access list for example is
Normal rights:
  arcsnewstaff rlidwk
  homer w
  system:administrators rlidwka
  myucid rlidwka
  http l
% fs sa . homer rw
Access list for example is
Normal rights:
  arcsnewstaff rlidwk
  homer rw
  system:administrators rlidwka
  myucid rlidwka
  http l
```

Additional example for fs setacl

```
% fs sa . homer none  ; fs la
Access list for example is
Normal ri%ghts:
  arcsnewstaff rlidwk
  system:administrators rlidwka
  myucid rlidwka
  http l
% fs sa -dir . -acl arcsnewstaff dwk -negative  ;fs la
Access list for example is
Normal ri%ghts:
  arcsnewstaff rlidwk
  system:administrators rlidwka
  myucid rlidwka
  http l
Negative rights:
  arcsnewstaff dwk
% fa sa . mycid all -clear  ; fs la
Access list for . is
Normal rights:
  myucid rlidwka
```

Command	Explanation
fs copyacl -fromdir d1 -todir d2	copy acl
fs ca -fromdir d1 -todir d2	alternative
fs ca -clear -fromdir d1 -todir d2	clears d2 before copying

Assistance

fs help copyacl

Caution

Keeps d2 privileges that are not in conflict with d1's

Command	Explanation
fs listquota	List quota
fs lq	**A**lternative

Assistance

fs help lq
fs help

9. System status

Command	Explanation
hostname	**Name of host**
who	**List of users in host**
whoami	Logged on user information
id	Detailed id of logged on user
date	Current date and time
cal	Calendar of current month
cal mo ye	Calendar of month mo (1-12) and year (yyyy)
cal year	Calendar of year (yyyy)
which command	absolute path to exe file of command
locate string	Find all dirs containing string
lsof	List of Open Files
uname -a	System limits

Command Redirection	Explanation
ls	Output on standard output
ls > x1	Output on file x1 ; if x1 exists it is overwritten or error file exists
ls >>x1	Output appended on file x1

BASH shell

Bash file	Purpose
~/.bash_profile	Configuration file ; it reads ~/.bashrc
~/.bashrc	No login bash terminal configuration file
/etc/profile	System wide config

1. When bash is invoked as interactive login shell or as a non-interactive login shell (--login option) it first reads and executes /etc/profile
2. Then it checks in sequence the following files. The first one that exists/is readable is read and executed.
   ~/.bash_profile
   ~/.bash_login
   ~/.profile

   If shell is started with –noprofile then step 2 is skipped.

3. When an interactive shell is started that is not a login shell then
   ~/.bashrc
   is read and commands in it executed.
4. Several time a ~/.bash_profile that exists (step 2) enforces the read and execution of the commands in .bashrc with the following command in ~/.bash_profile
   if [-f ~/.bashrc]; then . ~/.bashrc; fi
5. Moreover a ~/.bashrc file might have a line
   if [-f /etc/bashrc]; then . /etc/bashrc; fi

Most frequent setup interactive login

1. /etc/profile
2. ~/bash_profile that call ~/.bashrc
3. with ~/.bashrc calling /etc/bashrc as well

An interactive non-login shell invocation calls

4. ~/.bashrc (which also calls /etc/bashrc)

On exit

5. ~/bash_logout

OSX

OSX: No interactive non-login; ~/.bash_profile the latter might call ~/.bashrc [only then is the latter being used]

NJIT setup: interactive login

1. /etc/profile used first [STEP-1]
2. File /etc/.bash_profile does not exist
3. /afs/cad/linux/local/etc/std-bash_profile
   sources ~/.bashrc [STEP-2a]
4. /afs/cad/linux/local/etc/std-bashrc [STEP-3a]
   sources /etc/bashrc
5. ~/.bash_profile [STEP-2]
   sources /afs/cad … /std-bash_profile [STEP-2a]
   sources ~/.bashrc [STEP-2a repeat]
6. ~/.bashrc [STEP-3]
   sources /afs/cad … /std-bashrc [STEP-3a]
   source ~/…myaliases …
7. ~/.profile is a Bourne Shell relic and does not exist
8. On exit ~/.bash_logout

NJIT setup: interactive non-login

1. ~/.bashrc [STEP-1]
2. with ~/.bashrc sourcing …/std-bashrc
3. with …/std-bashrc sourcing /etc/bashrc
   [another alternative : /etc/bash.bashrc]
4. On exist ~/.bash_logout

NJIT invocation : sh (not bash)

1. /etc/profile and then ~/.profile

NJIT : bash invoked by ssh

1. ~/.bashrc and see interactive non-login

```
% echo $-                #shows shell
```

TCSH SHELL : interactive login

1. Either ~/.cshrc or ~/.login
   [/etc/csh.cshrc or /etc/csh.login are executed]

2. OR ~/.tcshrc
3. At logout ~/.logout [/etc/csh.logout]

TCSH SHELL : non-interactive login

1. ~/.cshrc
2. At logout ~/.logout

PID below stands for ProcessID

PPID below stands for Parent ProcessID

UID below stands for UID (eg myucid)

PROCESS STATUS	Meaning
w	what is going on in system (Load info shown)
finger	
ps	Current login (terminal) process status (ps)
ps ef	More informative compared to ps
ps -f	PID and PPID displayed
ps aux	All processes in system (BSD syntax)
ps -aux	Same
ps -ef |egrep myucid	Find processes of user myucid
ps -ef |egrep bash	Find all bash instances + commands with word bash
kill -9 PID	Kill(Terminate) a process with ID PID (eg kill -9 12345) 9 is signal SIGKILL
man 7 signal	List of all signals including SIGKILL
top	List of all processes (scree nview)
cat /proc/cpuinfo	CPU info
cat /proc/cmdline	Kernel command line invocationinfo
cat /proc/partitions	Partition info
ls /proc/pid	Information about process with PID pid

10. Process management

A process is a program in execution. An executable program, the result usually of the compilation process of a source code file written in a high-level compiled programming language, resides on secondary memory. Let for the sake of an example am executable program be name prx.

A user intents to start running prx. The user needs to locate prx in the filesystem hierarchy. On a visual (graphical system) it suffices to click on the icon that is associate with the executable file named prx. In a command line mode, the users needs to describe and write the name of the executable file. This way the user preceded the name prx of the file with a ./ to indicate that the location of the file is in the current directory (the dot preceding the slash). If the ./ is omitted the OS would search for prx in a predefined collection of directories. This collection does not include the current directory (.) unless the user has explicitly done so in advance.

As soon as typing ./prx followed by an ENTER is completed control goes to the OS (kernel) and its long term scheduler (sometimes known as the JOB scheduler). More often than not the long term scheduler would OK the execution request, and then the loader would load the executable prx in main memory as needed (and instructed) by the OS kernel and kernel data structures would be updated accordingly. Three files would be opened by default in UNIX (and Linux): standard input, standard output and standard error mapped to the keyabord, terminal/screen and terminal/screen of the user's device used to connect to the host. At that point execution of prx would eventually start. In rare cases the system is fully loaded and ./prx 's invocation would trigger no reaction as if the system is frozen. The user can wait or kill the invocation with a CTRL-C. In the setup as described the parent process of the process associated with prx is the shell that is going to create the process prx. The (bash) shell presence is denoted by the prompt %. So there is an intermediary between the program prx and the kernel and this is the shell process.

PROCESS STATUS	Meaning
% ./prx	Run process (program) prx
% CTRL-Z	while prx is running CTRL-Z suspends execution of prx
% jobs	A list of processes in the background is shown
% jobs	A jobs show the process STOPPED
% bg	The suspended process (prx) is reinstated and resume execution in the background
% fg	A suspended process resume execution in the foregound
% bg %n	n a numeric value associated with process
% fg %n	pick a given among multiply available processes
% kill %n	Terminates / kill process in the background
% fg %m	Bring process in the foreground
% CTRL-C	and then terminate it : a two step equivalent to a kill %m

11. Tar, zip and gzip usa

Tar/Untar, Zip, gzip	Meaning
tar cvf x.tar a.txt b.txt	Pack a.txt,b.txt into x.tar
tar xvf x.tar	Unpack x.tar into its contents x: extract, v: verbose f: tar file follows (x.tar) c: create
zip f.zip a.txt b.txt	Pack a.txt,b.txt into f.zip
unzip f.zip	Unpack f.zip
gzip x.tar	Compress x.tar
gzip -d x.tar.gz	Uncompress x.tar.gz

12. Compilation processes

Program module management	**Meaning**
module avail gcc	List all gcc versions available
module load gcc/9.1.0	Load version 9.1.0 of gcc
module list	List currently loaded module files

GCC compiler invocation	**Meaning**
gcc -v	List gcc version
gcc pr1.c	Compile + Assemble +Link into a.out Compile : from C to Assembly Assemble: from Assembly to Object code Link: from Object code link with other libraries to eventually an exec code
gcc pr1.o -o pr1	Override default: use pr1 instead
gcc pr1.c -lm -o pr1	Link also with math library (eg your code has sin, cos, exp etc) -l : link with library m of -lm is libm.a (static) or libm.so (dynamically linked)
gcc pr1.c -S	Compile only (assembly output generated is file pr1.s)
gcc pr1.s -o pr1s	Assemble + Link ; output is pr1s
gcc pr1.s -c	Compile + Assemble : object code pr1.o
gcc pr1.o -o pr1so	Link + generate executable pr1so
objdump -D pr1s	

Program Execution Management	Meaning
./pr1	Run program from current dir.
echo $?	For C/C++/Java program on completion print return value of main function returned to shell program
export PATH=.: $PATH	Add current directory to the PATH (of directories) the shell will be exploring to find an executable file such as pr1; then pr1 can be used rather than ./pr1
pr1	Might work then
export PATH=.:$PATH	Add it to ~/.profile and activate it in current session
source ~/.profile	If it does not work
echo "PATH=.:$PATH" >> ~/.bash_profile	
echo "PATH=.:$PATH" >> ~/.bashrc	

Task1: Generate named executable file from C source code file

Input: source code file exe.c (resides in SM)

Output: executable file myexe (resides in SM)

```
% gcc exe.c -o myexe
```

Task2: Generate default-named executable file from C source code file

Input: source code file exe.c (resides in SM)

Output: By default in file a.out (resides in SM)

```
% gcc exe.c
```

Task3: Generate default-named object file from C source code file

Input: source code file exe.c (resides in SM)

Output: object code file exe.o (resides in SM)

```
% gcc -c exe.c
```

Task4: Generate named executable file from object code file

Input: object code file exe.o (resides in SM)

Output: executable file oexe (resides in SM)

```
% gcc exe.o -o oexe
```

Note that myexe, a.out and oexe are the same structured executable file but have different filenames. You may verify that by doing a

```
% md5sum myexe oexe a.out
```

When i did so i got an output

```
c004edff4c972e24106370a2acbebbb0  myexe
c004edff4c972e24106370a2acbebbb0  oexe
c004edff4c972e24106370a2acbebbb0  a.out
```

The long string of hexadecimal characters is the fingerprint of the corresponding file name listed to its right. The three files have identical fingerprints. It is highly unlikely to be different.

If you are curious in finding out what is going on, replace gcc in the previous commands with gcc -v

For example

```
% gcc exe.c -o myexe
```

would become

```
% gcc -v exe.c -o myexe
```

You execute an (executable) file by invoking the file's namefile from the directory it is located. The OS's loader will load the progam into main memory(MM) and thus turn the program into a program in execution, i.e. a process.

For the bash commands (eg date,ls, rm, cp) everytime you issue a command you in fact request bash to run the corresponding identically named executable file (i.e. date, ls, rm, cp). You do not need to go the directory these exe files (short for executable files) are located. Bash know they are in /bin. Thus

```
% date
```

or

```
% /bin/date
```

both run the same exe file i.e. date. You do not need to do

```
% cd /bin
% ./date
```

Task 5: Execute i.e. run an executable file

Input: Executable file myexe

Effect: myexe is being run

```
% ./myexe
```

A program resides in secondary memory (SM). A program in execution (process) resides in Main Memory (MM) all of it or part of it. We need to move the program from SM into MM. The 'loader' moves myexe from SM into MM using the loader invocation ./myexe. The ./ means "in the current directory find a file named as indicated afterwards". Thus ./myexe means "in the current directory find a file named myexe and load/run

it". The actor that will find the file and run it is an operating system (kernel) program known as the UNIX/LINUX loader. Subsequently the execution of myexe starts.

Task 6: Entry point of C or C++ or Java programs.

ENTRY POINT of a C/C++/Java program. The entry point of myexe is indicated in exe.c by a function of a special name: main.

RETURN VALUE OF MAIN (C or C++).When the main function completes its execution it returns a value: 0 means regular termination, no errors detected or reported. Another value might indicate something else: an error and its error code is then returned. The prototype for main should be in C or C+

```
int  main(...
```

and never

```
void main(...
```

CAPTURING the RETURN VALUE of main in UNIX or LINUX. In the shell (command prompt, terminal) after the invocation ./myexe, if you type the following line, you can capture the return value of main with the following bash shell command.

```
% echo $?
```

Note that % is the bash prompt. You do not type it. And of course you press ENTER at the end to submit the request echo $? to the bash shell. If the value returned by a main function is more than 255 then only the (unsigned) integer value identified by the rightmost 8 bit will be returned!

Task 7: Generate assembly file from C source code file

Input: source code file exe.c [in SM]

Output: assempy file file exe.s [in SM]

```
% gcc -S exe.c
```

You may view the assembly file by typing one of the following

```
% cat  exe.s
% more exe.s
```

Task 8: Generate from assembly file an object code file

```
Input:  assembly      file exe.c              [in SM]
Output: object code   file asexe.o or gccexe.o  [in SM]
```

```
% as  exe.s  -o asexe.o
```

or by using the gcc compile infrastructure

```
% gcc -c exe.s  -o gccexe.o
```

```
void main(...
```

CAPTURING the RETURN VALUE of main in UNIX or LINUX. In the shell (command prompt, terminal) after the invocation ./myexe, if you type the following line, you can capture the return value of main with the following bash shell command.

```
% echo $?
```

Note that % is the bash prompt. You do not type it. And of course you press ENTER at the end to submit the request echo $? to the bash shell. If the value returned by a main function is more than 255 then only the (unsigned) integer value identified by the rightmost 8 bit will be returned!

Task 7: Generate assembly file from C source code file

```
Input:  source  code  file exe.c  [in SM]
Output: assempy file  file exe.s  [in SM]
```

```
% gcc -S exe.c
```

You may view the assembly file by typing one of the following

```
% cat  exe.s
```

```
% more exe.s
```

Task 8: Generate from assembly file an object code file

```
Input:  assembly      file exe.c              [in SM]
Output: object code   file asexe.o or gccexe.o  [in SM]
```

```
% as  exe.s  -o asexe.o
```

or by using the gcc compile infrastructure

```
% gcc -c exe.s  -o gccexe.o
```

You may observe that all the files exe.o asexe.o gccexe.o have the same content by doing a

```
%  md5sum asexe.o exe.o gccexe.o
```

(When i did so i got the following output

```
02662f47e95ce03fd2a7faff2c6a6b35  asexe.o
02662f47e95ce03fd2a7faff2c6a6b35  exe.o
02662f47e95ce03fd2a7faff2c6a6b35  gccexe.o
```

Task 9: Generate from object code file an executable file

```
Input:  object code   file asexe.o or gccexe.o  [in SM]
Output: executable    file exeas   or exegcc  [in SM]
```

```
%  gcc gccexe.o -o  exegcc
%  gcc asexe.o  -o  exeas
```

The object code contains machine code for the user defined functions. System functions maps to operating system supplied code that resides elsewhere (not in your current directory). The next step links you object code file (eg asexe.o) to the

system code files (eg the ones realizing function printf) and the combination become the executable file (eg exeas).

Task 10: Object file in readable form

Input: object code file exe.o [in SM]

Output: object dump file dexe.txt [in SM]

```
% objdump -D exe.o >dexe.txt
```

Task 11: Executable file in readable form

Input: executable file myexe [in SM]

Output: exe dump file dmyexe.txt [in SM]

```
% objdump -D myexe >dmyexe.txt
```

You may observe the difference in size between the two dump files.If the >dexe.txt or >dmyexe.txt part of the command is missing the output is displayed on screen (stdout).

Objdump	Meaning
objdump -f pr1	File header info
objdump -h pr1	Section header info
objdump -d pr1	Assembler content of exec section of pr1
objdump -D pr1	Assembler content of all sections
objdump -s pr1	All content printed
objdump --help	Help : -h is section header info option