Introduction to Linux / Unix Shell

What is shell ?

– command interpreter in Linux / Unix like systems
– provides text mode user interface
– reads what user has typed, interprets as a command and executes
– one of the reasons why Unix is so famous

What are the commonly used shells?

– Bourne shell
– (Bourne-Again shell) bash – the shell used in this course
– C shell
– Korn shell

What is shell script or a shell program ?

– list of commands which will be interpreted and executed by shell
– usually saved in a file with .sh extension for frequent execution rather typing them on prompt every time

When to choose shell script ?

– performance is immaterial – but you can write efficient scripts
– development time is too less

Shell as a Programming language : basic constructs

- supports a single data type - a string of characters (not the same as string in C++)
- variables : supports both predefined and user defined
- restricted set of operators in order of reduced priority
  | (pipe)
  && (AND)
  | | (OR)
  ; (sequencing)
  &(background process)
- Operators for
  i/o redirection :
  < (input)
  > (ouput or 1>)
  2> (error)
  >> (append output)
  2>&1 (error is same as output)
  1>&2 (output is same as error)
- structured control flow constructs
  - sequencing
  - selection : if case
  - iteration : for while until
- predefined unix commands or user defined executables

Shell as an interface to the Filesystem

- Operations on files and directories
- Information about the file system

Using shell to interface with process system

- run one or more processes
- get status of running processes
- get process attributes

Shell operators – syntax and semantics

Exit status : Every command executed by the shell, on its termination, returns a value to the shell. This returned value is called the **exit status** of the command executed. The return value 0 is interpreted as true for successful execution and a non-zero value (usually >=1) as interpreted as false.

The parentheses [ ] enclose terms that are optional; p1, p2, p3 are unix commands (executable files)

Operator	**Syntax**	**Exit status**	**Semantics**
Sequence (;)	p1 ; [p2]	Exit status of p2 if present, else p1	After termination of p1, shell starts execution of p2
Background process (&)	p1 & [p2]	Exit status of p1& is 0	Shell initiates execution of p1 and returns immediately to execute the next command. Command p1 is said to run in background.
AND (&&)	p1 && p2	exit_status (p1) ? exit_status(p2) : exit_status (p1)	If exit status of p1 is true then shell runs the command p2 else p2 is not executed
OR (| |)	p1 | | p2	exit_status (p1) ? exit_status(p1) : exit_status (p2)	If exit status of p1 is false then shell runs the command p2 else p2 is not executed
Pipe (|)	p1 | p2	Exit status of last command	Shell initiates both the commands p1 and p2 for execution. The o/p produced by p1 on stdout is redirected to be the input (stdin) for p1

The precedence of the operators in order from highest to lowest are given below; also the binary operators are left associative.

pipe (|)
&& | |
; &

Variables of the Shell : Shell has several pre-defined variables, a few of which we have encountered already, such as HOME, PWD, PATH, etc. There are many other variables that specify the environment of the shell.

To observe all the variables of the shell and the values assigned to them, the set command may be used.

$ set > shell_variables

A small subset of shall variables along with the values assigned on this computer is listed below.

```
BASH=/bin/bash                                  sr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/game
HISTFILESIZE=2000                               s:/usr/local/games:~/nsl/Desktop/cs101_2013/201
HISTSIZE=1000                                   3/simplecpp:/home/prof-biswas/nsl_backup/nsl_b
HOME=/home/prof-biswas                          ackup_oct29/Desktop/cs101_2013/2013/simplec
HOSTNAME=prof-Biswas                            pp'
HOSTTYPE=x86_64                                 PWD=/home/prof-biswas/nsl_backup/nsl_backup
IFS=$' \t\n'                                    _oct29/Desktop/cs101_2013/2013/shell/new_pro
LANG=en_IN                                      gs
LANGUAGE=en_IN:en                               SHELL=/bin/bash
LOGNAME=prof-biswas                             USER=prof-biswas
OSTYPE=linux-gnu                                .....
PATH='/usr/lib/lightdm/lightdm:/usr/local/sbin:/u
```

The other $-variables of the shell have been discussed earlier.

Useful Constructs of shell language

1. if-then or if-then-else construct (similar to C++, the **else**-part is optional and **elif** is permitted)

```
if  command1
then
   command(s)  # executed when exit status of command1 is true
else
   command(s)  # executed when exit status of command1 is false
fi
```

Note that command(s) denotes one or more commands.

2. **for construct**

Syntax of use :

```
for var  in  List
do
  command(s)   # body of the loop
done
```

Semantics : List is a list of words (may be formed by the shell or as the result of another command), and var is a user defined shell variable. The command(s) in the body of the loop are executed as many times as there are words in the List. The n-th execution of the loop sets the value of var to the n-th word in the list ($var=...) and then executes the body of the loop enclosed between **do** and **done**. The common forms of List can be

$*	expands to List = {$1, $2, ...} where the size of List is $#
*****	expands to List = { all files in the current directory}
{m .. n}	expands to List = {m, m+1, m+2, ... n}
`command`	expands to the List being formed by the executing the command using its

output

Another syntax : C like structure

```
for ( (var=1; var<20; var++) )
do
  command(s)   # body of the loop
done
```

The usual semantics of C/C++, note the use of extra parentheses however.

3. **while -do construct**

Syntax of use :

```
while command1
do
  command(s)   # body of the loop
done
```

Semantics : The command1 after the unitl keyword is executed first. If the exit status is false the body of the loop enclosed in **do** ... **done** are executed. After every iteration, the command1 is executed and so long as the its exit status is false the body is also executed. The loop is exited when the exit status of command1 is true.

4. **until – do construct**

Syntax of use :

```
until command1
do
  command(s)  # body of the loop
done
```

Semantics : The command1 after the unitl keyword is executed first. If the exit status is false the body of the loop enclosed in **do** ... **done** are executed. After every iteration, the command1 is executed and so long as the its exit status is false the body is also executed. The loop is exited when the exit status of command1 is true.

5. case construct

Syntax of use :

```
case  word  in
  pattern)  commands ;;
  pattern)  commands ;;
  ......
  pattern) commands ;;
esac
```

Semantics : This construct compares word with the patterns, in the order from top to bottom, and executes the commands with the first pattern (and only the first pattern) that matches. The ;; is

equivalent to a break and the next command after esac is executed thereafter. The last pattern is expected to be written carefully so that it becomes the default, if all previous patterns fail to match. In the event that none of the patterns match, no commands in the body of case are executed and control goes to the command following esac.
The patterns may be of the following forms :

1)	denotes the string 1
[mn])	denotes either of the strings m or n
m | n)	denotes either of the strings m or n
[m-n])	denotes the set of strings {m, m+1, ..., n}
[mn][pq]	denotes the set of strings {mp, mq, np, nq}

Examples : Read and execute the shell scripts :

1. **max3num.sh** : find the maximum among 3 numbers : use of read, test command and nested if-then-else construct; the script **maxstr3.sh** is a similar attempt to find the string that has the maximum length

```
# program for finding max among 3 integers
echo -n "enter 3 values : "
read var1 var2 var3
echo "$var1 $var2 $var3"
#echo “sleeping for 2secs”
sleep 2
if test $var1 -ge $var2 -a $var1 -ge $var3
then
echo "$var1 is maximum"
elif test $var2 -ge $var1 -a $var2 -ge $var3
then echo "$var2 is maximum"
else
echo "$var3 is maximum"
fi
exit 0
# end of script
```

2. **case.sh** : shows usage of case construct with variety of pattern specifications.

```
# switch case in bash
echo "SWITCH CASE in shell"
x=$1 # supply the value through argument 1
case $x in
# compared with pattern instead of integers
1) echo "one" ;; # equivalent to break
2) echo "two" ;;
[35]) echo "three or five" ;;
[78])  echo " seven or eight " ;;
[2-4][59]) echo "1st digit is from [2-4], 2nd from [59]" ;;
*)
# default case
echo "default pattern"  ;;
esac
```

3. **shell_constructs.sh** : illustrates the usage of the loop constructs of shell in different applications.

Run the following script in some directory and explain the output produced.

```
# for on a list created by a command or expanded by shell
        # `seq 1 9` expands to 1 2 3 4 5 6 7 8 9
        # then for works on that list
        echo "FOR LOOP a command generated list"
        for x in `seq 1 9`
        do
          echo -n " $x "
        done
        # list is created by shell expansion syntax
        echo; echo "FOR LOOP on a shell generated list"
        for x in {1..9}
        do
           echo -n " $x "
        done
        echo; echo "C LIKE FOR LOOP"
        # using C like structue with use of break
        for ((x=1; x<20; x++))
        do
          echo -n " $x "
          if test $x -eq 9
          then
            break
          fi
        done
        echo ; echo "FOR LOOP on a list of files in current directory"
        # loop over list files
        for f in `ls`
        do
         echo -n " $f  "
        done
        echo ; echo "FOR LOOP on a list of files using shell operator *"
        # loop over list files
        for f in *
        do
         echo -n " $f  "
        done
        echo
        echo " WHILE LOOP FOR SEQUENCE "
        x=1
        while test $x -le  9 #
        do
          echo -n " $x "
          x=`expr $x + 1` # get the value after executing a command
        done
        echo
```

```
echo " UNTIL LOOP FOR SEQUENCE "
x=1
until test $x -gt  9 #
do
  echo -n " $x "
  x=`expr $x + 1` # ' get the value after executing a command
done

echo ; echo "WHILE LOOP on a list of files"
ls | while read f
do
  echo -n " $f "
done
echo ; echo "WHILE LOOP with i/o redirection"
# using input redirection
ls > /tmp/ls.txt
while read f
do
 echo -n " $f "
done < /tmp/ls.txt
# output redirection from a loop
while read f
do
  echo -n " $f \ "
done < /tmp/ls.txt > /dev/null
echo
# until do loop of the shell


# switch case in bash
echo "SWITCH CASE in shell"
x=$1 # supply the value through argument 1
case $x in
# compared with pattern instead of integers
1) echo "one" ;; # equivalent to break
2) echo "two" ;;
[3-5]) echo "three or five" ;;
6|7) echo " six or seven " ;;
[67][89]) echo "1st digit is from [67], 2nd from [89]" ;;
*)
# default case
echo "default pattern"
;;
esac
rm /tmp/ls.txt
exit 0
```

Few Commonly used shell commands and utilities :

test expr cut sed sort grep

test command : This command is a very useful shell program and is often used in the conditional part of the if-statement constructs. The test command is used to check file types and compare values.

Syntax of use : **test** EXPR
There are other forms of test but ignored for our purpose.

If EXPR is omitted, **test** returns false. If EXPR is a single argument, test returns false if the argument is null and true otherwise.

Exit status:

0 if the expression is true,
1 if the expression is false,
2 if an error occurred.

test has **file status** checks, **string operators**, and **numeric comparison operators**.

Though there are several options in the various forms of use of test, we restrict to the following for our course.

File type tests : These options test for particular types of files. (Everything's a file, but not all files are the same)

EXPR (shell)	Semantics
-d FILE	True if FILE exists and is a directory
-f FILE	True if FILE exists and is a regular file
-L FILE	True if FILE exists and is a symbolic link

Access permission tests : These options test for particular access permissions of FILE, which can be of any type – regular file, directory or the other file types that are skipped here.

EXPR (shell)	Semantics
-r FILE	True if FILE exists and read permission is granted
-w FILE	True if FILE exists and write permission is granted
-x FILE	True if FILE exists and execute permission is granted (or search permission if it is a directory)

File characteristic tests : These options test other file characteristics.

EXPR (shell)	Semantics

-e FILE	True if FILE exists
-s FILE	True if FILE exists and size > zero
FILE1 -nt FILE2	True if FILE1 is newer (as per modification date) than FILE2 or if FILE1 exists and FILE2 does not
FILE1 -ot FILE2	True if FILE1 is older (as per modification date) than FILE2 or if FILE2 exists and FILE1 does not
FILE1 -ef FILE2	True if FILE1 and FILE2 have the same device and inode numbers, i.e., if they are hard links to each other

String tests : These options test string characteristics.

EXPR (shell)	Semantics
-z STRING	length (STRING) is equal to zero
-n STRING	length (STRING) is not equal to zero
STRING1 = STRING2	True if the strings are equal
STRING1 == STRING2	True if the strings are equal (synonym for =)
STRING1 != STRING2	True if the strings are not equal

Numeric tests : Numeric relational operators. The arguments must be entirely numeric (possibly negative), or the special expression **-l STRING**, which evaluates to the length of STRING.

EXPR (shell)	Semantics EQUIVALENT C++ EXPRESSION
ARG1 -eq ARG2	ARG1 == ARG2
ARG1 -ne ARG2	ARG1 != ARG2
ARG1 -lt ARG2	ARG1 < ARG2
ARG1 -le ARG2	ARG1 ≤ ARG2
ARG1 -gt ARG2	ARG1 > ARG2
ARG1 -ge ARG2	ARG1 ≥ ARG2

Connectives for test : The usual logical connectives as given below.

EXPR (shell)	Semantics
! EXPR	True if EXPR is false
EXPR1 -a EXPR2	True if both EXPR1 and EXPR2 are true
EXPR1 -o EXPR2	True if either EXPR1 or EXPR2 is true

1. Installing new software on Linux / ubuntu

- **apt-get** is the program that does the job seamlessly
- The program **apt-get** reads a file, **apt.conf**, which should have information in a certain format. This is done by editing the apt.conf file:

  $ sudo gedit /etc/apt/apt.conf

In this file, add the following line

  Acquire::http::Proxy"http://username:password@proxyhost:port/";

Now install new s/w using the following

  $ sudo apt-get update
  $ sudo apt-get install dot
  $ sudo apt-get install ddd

2. login shell

When we **login** (enter username and password) either by sitting in front of the machine, or remotely through ssh, the shell script **.bash_profile** for bash (equivalent for other shells) is executed to configure our shell before the initial command prompt appears. This is the login shell.

The login shell for every user is specified in the passwd file, see the file passwd stored in the directory /etc

  $ cat /etc/passwd

A few lines of this file from the instructor's laptop are extracted below.

root:x:0:0:root:/root:/bin/bash
sync:x:4:65534:sync:/bin:/bin/sync
prof-biswas:x:1000:1000:prof biswas,,,:/home/prof-biswas:/bin/bash

The seven fields of an entry of passwd file are summarized in the following table.

S. N.	**Fieldname**	**1st Line**	**2nd Line**	**3rd Line**
1	**Username**	root	sync	prof-biswas
2	**Password**	x	x	x
3	**User ID**	0	4	1000
4	**Group ID**	0	65534	1000
5	**User ID Info**	root	sync	prof biswas,,,
6	**Home directory**	/root	/bin	/home/prof-biswas
7	**Command/shell**	/bin/bash	/bin/sync	/bin/bash

Fields in /etc/passwd

The /etc/passwd contains one entry per line for each user (or user account) of the system, fields are

separated by colon (:).

1. **Username**: The name assigned to an user for the purpose of logging. It is usually between 1 and 32 characters in length.
2. **Password**: An **x** character indicates that encrypted password is stored in /etc/shadow file.
3. **User ID (UID)**: Each user is assigned a user ID (UID). UID 0 (zero) is reserved for root and UIDs 1-99 are reserved for other predefined accounts, also UID 100-999 are reserved by system for administrative and system accounts/groups.
4. **Group ID (GID)**: The primary group ID (stored in /etc/group file)
5. **User ID Info**: This field enables adding extra information about an user - such as full name, phone number etc. This field is used by **finger** command.
6. **Home directory**: The absolute path to the directory the user will be in after a successful log in. If this directory does not exists then users directory becomes /
7. **Command/shell**: The absolute path of a command or shell (/bin/bash). Typically, this is a shell but it does not have to be as in the 2nd line of passwd file.

Our encrpted password is not stored in /etc/passwd file. It is stored in /etc/shadow file.

Almost, all modern Linux / UNIX line operating systems use some sort of the shadow password suite, where /etc/passwd has asterisks (*) instead of encrypted passwords, and the encrypted passwords are in /etc/shadow which is readable by the superuser only.

3. read command- Read a line from standard input

Syntax: read [options] var

Description: "Reads" the value of a variable from stdin, that is, interactively fetches input from the keyboard. The -a option lets read get array variables (arrays are not part of the course)

Option Meaning

-d DELIM The first character of DELIM is used to terminate the input line, rather than newline.
-e readline is used to obtain the line.
-n NCHARS read returns after reading NCHARS characters rather than waiting for a complete line of input.
-p PROMPT Display PROMPT, without a trailing newline, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.
-r If this option is given, backslash does not act as an escape character. The backslash is considered to be part of the line. In particular, a backslash-newline pair may not be used as a line continuation.
-s Silent mode. If input is coming from a terminal, characters are not echoed.
-t TIMEOUT Cause read to time out and return failure if a complete line of input is not read within TIMEOUT seconds. This option has no effect if read is not reading input from the terminal or from a pipe.

Examples:

Read from the /etc/passwd file and store the distinct words in user defined variables. Run on your system and check the output.

```
IFS=:
while read -r f1 f2 f3 f4 f5 f6 f7
do
 echo "User with login $f1 uses $f7 shell and home directory in $f6 "
done < /etc/passwd
```

Read input from user and prompt the user about the expected values. We use -p option for this purpose. The echo command is not needed to display information back to user, as one read -p option takes care of the same.

read -p "Please enter a number, between 1 and 10 : " VAL1

Two interesting options are -t and -s. The -t option followed by a number of seconds provides an automatic timeout for the read command. This means that the read command will give up after the specified number of seconds if no response has been received from the user. This option could be used in the case of a script that must continue (perhaps resorting to a default response) even if the user does not answer the prompts. Here is the -t option in action.

The -s option causes the user's typing not to be displayed. This is useful when you are asking the user to type in a password or other security related information.

4. Non-login shell

If we've already logged into your machine and open a new terminal window (xterm), then .bashrc is executed before the window command prompt. .bashrc is also run when you start a new bash instance by typing /bin/bash in a terminal.

Why two different files, .bashrc and .bash_profile ?

If we'd like to print some lengthy diagnostic information about our machine each time we login (load average, memory usage, current users, etc). We only want to see it on login, so only want to place this in our .bash_profile. If we put it in our .bashrc, we'd see it every time a new terminal window is opened.

Most of the time we don't want to maintain two separate config files for login and non-login shells — when we set a PATH, we want it to apply to both. We can fix this by sourcing .bashrc from our .bash_profile file, then putting PATH and common settings in .bashrc.

5. The command cut :

Select a Specific Field from a File

If we wish to extract a whole field from a line, we use the cut command. The option -f specifies which field we want to extract, and the option -d specifies what is the field delimiter that is used in the input file.

The following example displays only first field of each lines from /etc/passwd file using the field delimiter : (colon). In this case, the 1st field is the username. The file

```
$ cut -d':' -f1 /etc/passwd
root
daemon
bin
sys
```

```
sync
games
bala
```

Select Multiple Fields from a File

We can also extract more than one fields from a file or stdout. The command given below displays username and home directory of users who has the login shell as “/bin/bash”.

```
$ grep "/bin/bash" /etc/passwd | cut -d':' -f1,6
root:/root
xyz:/home/xyz
```

To display the range of fields specify start field and end field as shown below. In this example, we are selecting field 1 through 4, 6 and 7

```
$ grep "/bin/bash" /etc/passwd | cut -d':' -f1-4,6,7
root:x:0:0:/root:/bin/bash
bala:x:1000:1000:/home/bala:/bin/bash
```

Select Fields Only When a Line Contains the Delimiter

In /etc/passwd example, if we pass a different delimiter other than : (colon), cut will just display the whole line.

In the following example, we’ve specified the delimiter as | (pipe), and cut command simply displays the whole line, even when it doesn’t find any line that has | (pipe) as delimiter.

```
$ grep "/bin/bash" /etc/passwd | cut -d'|'  -f1
root:x:0:0:root:/root:/bin/bash
bala:x:1000:1000:bala,,,:/home/bala:/bin/bash
```

But, it is possible to filter and display only the lines that contains the specified delimiter using **-s option.**

The following example doesn’t display any output, as the cut command didn’t find any lines that has | (pipe) as delimiter in the /etc/passwd file.

```
$ grep "/bin/bash" /etc/passwd | cut -d'|' -s -f1
```

Select All Fields Except the Specified Fields

In order to complement the selection field list use option –complement.

The following example displays all the fields from /etc/passwd file except field 7

```
$ grep "/bin/bash" /etc/passwd | cut -d':' --complement -s -f7
root:x:0:0:root:/root
bala:x:1000:1000:bala,,,:/home/bala
```

Change Output Delimiter for Display

By default the output delimiter is same as input delimiter that we specify in the cut -d option.

To change the output delimiter use the option –output-delimiter as shown below. In this example, the input delimiter is : (colon), but the output delimiter is # (hash).

```
$ grep "/bin/bash" /etc/passwd | cut -d':'  -s -f1,6,7 --output-delimiter='#'
root#/root#/bin/bash
bala#/home/bala#/bin/bash
```

Change Output Delimiter to Newline

In this example, each and every field of the cut command output is displayed in a separate line. We still used –output-delimiter, but the value is $'\n' which indicates that we should add a newline as the output delimiter.

```
$ grep bala /etc/passwd | cut -d':' -f1,6,7 --output-delimiter=$'\n'
bala
/home/bala
/bin/bash
```

6. The sort command :

Command : sort <options> file
The sort command sorts its input in ascending order
options : -n for numeric sort -kc for column number c
-r for reverse sort (descending order)
-k24 (muliple column - first on col2 then on col 4)
-u unique, i.e., sorted output without duplicates
-b : Ignores leading spaces in each line
-d : Uses dictionary sort order. Conisders only spaces and alphanumeric characters in sorting
-f : Uses case insensitive sorting.
-M : Sorts based on months. Considers only first 3 letters as month. Eg: JAN, FEB
-n : Uses numeric sorting
-R : Sorts the input file randomly.
-r : Reverse order sorting
-k : Sorts file based on the data in the specified field positions.
-u : Suppresses duplicate lines
-t : input field separator

Sorting based on numeric value of String using UNIX sort command:
Many times instead of alphabetic sorting we need numeric sorting. Just like in below example of Unix sort command if we want to sort based upon numeric value of PID we can use sort -n along with sort -k(column). Since here PID is second coloumn sort -nk2 will work for us. This is also another great example of UNIX sort by column, which allows you do sort the data based on any column in UNIX.

```
$  ps -ef | sort -nk2
UID              PID      PPID TTY     STIME COMMAND
.....            ......   ......       ......    .....
```

Reverse sort by using UNIX sort command

Some time we need to sort in reverse order e.g. descending order. sort -r option allow us to perform reverse sorting in unix.

```
$ ps -ef | sort -rnk2
.....            ......   ......       ......    .....

 UID     PID    PPID TTY     STIME COMMAND
```

unix sort by column : Sorting based on any column in the input.
sort command in unix mostly used in combination of other unix commands like find, grep, ls or ps and most of these command produce output in tabular format and we want to sort based on any column. unix sort command allow us to do this by using sort -k option. Let's see an example or unix sort command to sort the output on any column we will use ps command output for this example and we will sort this output on column 2 (PID) and later on column 3 (PPID). Execute the following shell commands and check the display.

```
$ ps -ef | sort -nk2

$ ps -ef | sort -nk3
  UID     PID    PPID TTY     STIME COMMAND
```

We can also sort based upon multiple column using sort command as sort -nk23 will sort the output first on second column and then on 3rd column.
Sorting output on alphabetical order by sort command

In this example of UNIX sort command we will see how to sort output of any command on alphabetical order. Sort command in UNIX sorts the output in alphabetic order if you don't provide any options as shown in below example.

unsorted output

```
$ cat names
stocks trading
futures trading
options trading
forex trading
electronic trading
```

sorted output

```
 cat names | sort
electronic trading
forex trading
futures trading
options trading
stocks trading
```

How to remove duplicates from sorted output in UNIX

As you have seen in above example of sort command in UNIX we have duplicates "stock trading" is coming two times. We can produce sorted output without duplicates in two ways in UNIX either by passing output of sort command to "uniq" command or by using sort -u option. Let's see an example of sorting with unique elements using UNIX sort command:

Sorted output with duplicates
unix-sort-examples@unix-tutorial:~/test cat names | sort
electronic trading
forex trading
stocks trading
stocks trading

Sorted output without duplicates
unix-sort-examples@unix-tutorial:~/test cat names | sort | uniq
electronic trading
forex trading
stocks trading

unix-sort-examples@unix-tutorial:~/test cat names | sort -u
electronic trading
forex trading
stocks trading

More Examples :

i) Let the contents of the current directory be as shown below and
LC_COLLATE=C and LANG=C

$ ls -l

-rw-r--r-- 1 prof-biswas prof-biswas 780 Nov 6 04:48 bachchan_poem_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 1688 Oct 29 22:53 casabianca_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 844 Oct 29 22:53 daffodils_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 563 Oct 29 22:53 memoriam_formatted.txt
-rw-rw-r-- 1 prof-biswas prof-biswas 773 Nov 6 22:15 out
-rw-rw-r-- 1 prof-biswas prof-biswas 714 Nov 6 22:15 outsort
-rw-r--r-- 1 prof-biswas prof-biswas 626 Oct 29 22:53 patriotism_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 564 Oct 29 22:53 sarojini_poem_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 3683 Oct 29 22:53 tagore_poem_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 1482 Oct 29 22:53 tennyson_formatted.txt

$ ls -l | sort

The output displayed on the screen

-rw-r--r-- 1 prof-biswas prof-biswas 563 Oct 29 22:53 memoriam_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 564 Oct 29 22:53 sarojini_poem_formatted.txt

```
-rw-r--r-- 1 prof-biswas prof-biswas  626 Oct 29 22:53 patriotism_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  780 Nov  6 04:48 bachchan_poem_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  844 Oct 29 22:53 daffodils_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 1482 Oct 29 22:53 tennyson_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 1688 Oct 29 22:53 casabianca_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 3683 Oct 29 22:53 tagore_poem_formatted.txt
-rw-rw-r-- 1 prof-biswas prof-biswas  714 Nov  6 22:15 outsort
-rw-rw-r-- 1 prof-biswas prof-biswas  773 Nov  6 22:15 out
```

The first 8 lines are identical upto 4 fields. The value in the fifth field decides the ordering between these 8 lines but not by its numeric value and because of the lexicographic ordering (ascii values) of this field as a string. The ordering of the lines 9 and 10 can be explained along similar lines.

ii) $ ls -l | sort -r

The output is sorted in descending order as is to be expected and can be seen in the display.

iii) The 5th field, which gives the size in bytes, is numerical. To sort all the lines in ascending order based on column (or field) 5, we use the option
-nk5 (n for numeric sort and k5 for column 5)

```
$ ls -l | sort -nk5
```

The output displayed is consistent with the ordering of lines based on the numeric values of column 5.

```
-rw-r--r-- 1 prof-biswas prof-biswas  563 Oct 29 22:53 memoriam_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  564 Oct 29 22:53 sarojini_poem_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  626 Oct 29 22:53 patriotism_formatted.txt
-rw-rw-r-- 1 prof-biswas prof-biswas  714 Nov  6 22:15 outsort
-rw-rw-r-- 1 prof-biswas prof-biswas  773 Nov  6 22:15 out
-rw-r--r-- 1 prof-biswas prof-biswas  780 Nov  6 04:48 bachchan_poem_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  844 Oct 29 22:53 daffodils_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 1482 Oct 29 22:53 tennyson_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 1688 Oct 29 22:53 casabianca_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 3683 Oct 29 22:53 tagore_poem_formatted.txt
```

iv) If we wish to sort the lines in the the order of columnn 6 first (month) and then by the value in column 7 (date) and then by column 8(time), we write

```
$ ls -l | sort -k6 -nk7,8
```

and the display on the screen is given below.

```
-rw-r--r-- 1 prof-biswas prof-biswas  780 Nov  6 04:48 bachchan_poem_formatted.txt
-rw-rw-r-- 1 prof-biswas prof-biswas  714 Nov  6 22:15 outsort
-rw-rw-r-- 1 prof-biswas prof-biswas  773 Nov  6 22:15 out
-rw-r--r-- 1 prof-biswas prof-biswas  563 Oct 29 22:53 memoriam_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  564 Oct 29 22:53 sarojini_poem_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  626 Oct 29 22:53 patriotism_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas  844 Oct 29 22:53 daffodils_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 1482 Oct 29 22:53 tennyson_formatted.txt
```

-rw-r--r-- 1 prof-biswas prof-biswas 1688 Oct 29 22:53 casabianca_formatted.txt
-rw-r--r-- 1 prof-biswas prof-biswas 3683 Oct 29 22:53 tagore_poem_formatted.txt

Examine the display and verify that the output is correct with respect to the specs.

v) Collating sequence and character set

Behaviour of shell scripts on character data are often not as expected. This is because a few environment variables (that are predefined) may not have been assigned proper values. The are basically two variables that are central to this issue, i) setting character collation order, variable LC_COLLATE, and ii) setting the acceptable input character set, variable LANG.

You may find the current values through the echo command

```
$ echo -n '$LC_COLLATE = ' $LC_COLLATE ; echo  '$LANG = ' $LANG
```

The output produced on the screen is

```
LC_COLLATE = $LANG =  en_IN
```

which states that LC_COLLATE in not assigned and $LANG is set to en_IN

With many modern Linux locale settings, such as en_US, en_CA, en_IN, ... the character set is not laid out in strict numeric order; the collating order places upper and lower case together. The listing in ascending order for the alphabets are :

```
a A b B c C .... x X y Y z Z
```

Example : create a file with the following characters, one per line, as shown below.

```
// contents of file alpha
a
A
b
B
c
C
y
Y
z
Z
```

Now sort the file using the sort command

```
$ cat alpha | sort
```

The output shown on the terminal is

```
a
A
b
B
c
C
y
Y
```

z
Z

Now run the following commands in sequence.

$LC_COLLATE=C; LANG=C; sort alpha

The output displayed on the screen is the sorting in the expected ascii order.

A
B
C
Y
Z
a
b
c
y
z

End of Document

Supratim Biswas
CS 101 Supplementary Notes