

(Implement the following on LINUX or other Unix like platform. Use C for high level language implementation)

1. Write programs using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir
2. Write programs using the I/O system calls of UNIX operating system (open, read, write, etc)
3. Write C programs to simulate UNIX commands like ls, grep, etc.
4. Given the list of processes, their CPU burst times and arrival times, display/print the Gantt chart for FCFS and SJF. For each of the scheduling policies, compute and print the average waiting time and average turnaround time. (2 sessions)
5. Given the list of processes, their CPU burst times and arrival times, display/print the Gantt chart for Priority and Round robin. For each of the scheduling policies, compute and print the average waiting time and average turnaround time. (2 sessions)
6. Developing Application using Inter Process communication (using shared memory, pipes or message queues)
7. Implement the Producer – Consumer problem using semaphores (using UNIX system calls).
8. Implement some memory management schemes – I
9. Implement some memory management schemes – II
10. Implement any file allocation technique (Linked, Indexed or Contiguous)

**Example for exercises 8 & 9 :**

Free space is maintained as a linked list of nodes with each node having the starting byte address and the ending byte address of a free block. Each memory request consists of the process-id and the amount of storage space required in bytes. Allocated memory space is again maintained as a linked list of nodes with each node having the process-id, starting byte address and the ending byte address of the allocated space. When a process finishes (taken as input) the appropriate node from the allocated list should be deleted and this free disk space should be added to the free space list. [Care should be taken to merge contiguous free blocks into one single block. This results in deleting more than one node from

# Operating System Lab Manual CS 2254

the free space list and changing the start and end address in the appropriate node]. For allocation use first fit, worst fit and best fit

## INDEX

E.No	Date	Contents	Page No.
1		Shell Programming: a) Unix Commands b) Vi Commands c) Unix Shell programming commands a) Concatenation of two strings b) Comparison of two strings c) Maximum of three numbers d) Fibonacci series e) Arithmetic operation using case	
2		System Calls a) Process Creation b) Executing a command c) Sleep command d) Sleep command using getpid e) Signal handling using kill k) Wait command	
3		I/O System Calls a) Reading from a file b) Writing into a file c) File Creation	
4		a) Implementation of ls command b) Implementation of grep command	
5		Given the list of processes, their CPU burst times and arrival times, display/print the Gantt chart for FCFS and SJF. Print avg. waiting time and turnaround time.	
6		Given the list of processes, their CPU burst times and arrival times, display/print the Gantt chart for Priority and Round robin. Print avg. waiting time and turnaround time.	
7		Developing Application using Inter Process communication (using shared memory, pipes or message queues)	
8		Producer-Consumer Problem using Semaphore	
9		Memory management Scheme-I a) Paging Concept	
10		Memory management Scheme-II a) Segmentation Concept	

11		Implement any file allocation technique (Linked, Indexed or Contiguous)	
----	--	---	--



**Ex.No : 1.a**

## **UNIX COMMANDS**

**Date:**

**AIM :**

To study and excute the commands in unix.

**COMMAND :**

### **1.Date Command :**

This command is used to display the current data and time.

**Syntax :**

\$date

\$date +%ch

**Options : -**

a = Abbrevated weekday.

A = Full weekday.

b = Abbrevated month.

B = Full month.

c = Current day and time.

C = Display the century as a decimal number.

d = Day of the month.

D = Day in 'mm/dd/yy' format

h = Abbrevated month day.

H = Display the hour.

L = Day of the year.

m = Month of the year.

M = Minute.

P = Display AM or PM

S = Seconds

T = HH:MM:SS format

u = Week of the year.

y = Display the year in 2 digit.

Y = Display the full year.

Z = Time zone .

To change the format :

**Syntax :**

\$date '+%H-%M-%S'

### **2.Calender Command :**

This command is used to display the calendar of the year or the particular month of calendar year.

**Syntax :**

a.\$cal <year>

b.\$cal <month> <year>

Here the first syntax gives the entire calendar for given year & the second Syntax gives the calendar of reserved month of that year.

### **3.Echo Command :**

This command is used to print the arguments on the screen .

**Syntax :**                      \$echo <text>

#### **Multi line echo command :**

To have the output in the same line , the following commands can be used.

**Syntax :**                      \$echo <text>\>text

To have the output in different line, the following command can be used.

**Syntax :**                      \$echo "text  
>line2  
>line3"

### **4.Banner Command :**

It is used to display the arguments in '#' symbol .

**Syntax :**                      \$banner <arguments>

### **5.'who' Command :**

It is used to display who are the users connected to our computer currently.

**Syntax :**                      \$who - option's

#### **Options : -**

H-Display the output with headers.

b-Display the last booting date or time or when the system was lastely rebooted.

### **6.'who am i' Command :**

Display the details of the current working directory.

**Syntax :**                      \$who am i

### **7.'tty' Command :**

It will display the terminal name.

**Syntax :**                      \$tty

### **8.'Binary' Calculator Command :**

It will change the '\$' mode and in the new mode, arithmetic operations such as +,-,\*,/,%,n,sqrt(),length(),=, etc can be performed . This command is used to go to the binary calculus mode.

#### **Syntax :**

\$bc                                      operations  
^d  
\$

1 base -inputbase

0 base - outputbase are used for base conversions.

Base :

Decimal = 1    Binary    = 2    Octal    = 8                      Hexa    = 16

### **9.'CLEAR' Command :**

It is used to clear the screen.

**Syntax :**                      \$clear

### **10.'MAN' Command :**

It help us to know about the particular command and its options & working. It is like 'help' command in windows .

**Syntax :** \$man <command name>

### **11.MANIPULATION Command :**

It is used to manipulate the screen.

**Syntax :** \$tput <argument>

**Arguments :**

- 1.Clear – to clear the screen.
- 2.Longname – Display the complete name of the terminal.
- 3.SMSO – background become white and foreground become black color.
- 4.rmso – background become black and foreground becomes white color.
- 5.Cop R C – Move to the cursor position to the specified location.
- 6.Cols – Display the number of columns in our terminals.

### **12.LIST Command :**

It is used to list all the contents in the current working directory.

**Syntax :** \$ ls – options <arguments>

If the command does not contain any argument means it is working in the Current directory.

**Options :**

- a- used to list all the files including the hidden files.
- c- list all the files columnwise.
- d- list all the directories.
- m- list the files separated by commas.
- p- list files include '/' to all the directories.
- r- list the files in reverse alphabetical order.
- f- list the files based on the list modification date.
- x-list in column wise sorted order.

### **DIRECTORY RELATED COMMANDS :**

#### **1.Present Working Directory Command :**

To print the complete path of the current working directory.

**Syntax :** \$pwd

#### **2.MKDIR Command :**

To create or make a new directory in a current directory .

**Syntax :** \$mkdir <directory name>

#### **3.CD Command :**

To change or move the directory to the mentioned directory .

**Syntax :** \$cd <directory name>

#### **4.RMDIR Command :**

To remove a directory in the current directory & not the current directory itself.

**Syntax :** \$rmdir <directory name>

## **FILE RELATED COMMANDS :**

### **1.CREATE A FILE :**

To create a new file in the current directory we use CAT command.

#### **Syntax :**

\$cat > <filename.

The > symbol is redirectory we use cat command.

### **2.DISPLAY A FILE :**

To display the content of file mentioned we use CAT command without '>' operator.

#### **Syntax :**

\$cat <filename.

Options -s = to neglect the warning /error message.

### **3.COPYING CONTENTS :**

To copy the content of one file with another. If file doesnot exist, a new file is created and if the file exists with some data then it is overwritten.

#### **Syntax :**

\$ cat <filename source> >> <destination filename>

\$ cat <source filename> >> <destination filename> it is avoid  
overwriting.

#### **Options : -**

-n content of file with numbers included with blank lines.

#### **Syntax :**

\$cat -n <filename>

### **4.SORTING A FILE :**

To sort the contents in alphabetical order in reverse order.

#### **Syntax :**

\$sort <filename >

#### **Option :**

\$ sort -r <filename>

### **5.COPYING CONTENTS FROM ONE FILE TO ANOTHER :**

To copy the contents from source to destination file . so that both contents are same.

#### **Syntax :**

\$cp <source filename> <destination filename>

\$cp <source filename path > <destination filename path>

### **6.MOVE Command :**

To completely move the contents from source file to destination file and to remove the source file.

#### **Syntax :**

\$ mv <source filename> <destination filename>

### **7.REMOVE Command :**

To permanently remove the file we use this command .

#### **Syntax :**

\$rm <filename>

## **8.WORD Command :**

To list the content count of no of lines , words, characters .

### **Syntax :**

\$wc<filename>

### **Options :**

-c – to display no of characters.

-l – to display only the lines.

-w – to display the no of words.

## **9.LINE PRINTER :**

To print the line through the printer, we use lp command.

### **Syntax :**

\$lp <filename>

## **10.PAGE Command :**

This command is used to display the contents of the file page wise & next page can be viewed by pressing the enter key.

### **Syntax :**

\$pg <filename>

## **11. FILTERS AND PIPES**

**HEAD :** It is used to display the top ten lines of file.

**Syntax:** \$head<filename>

**TAIL :** This command is used to display the last ten lines of file.

**Syntax:** \$tail<filename>

**PAGE :** This command shows the page by page a screenfull of information is displayed after which the page command displays a prompt and passes for the user to strike the enter key to continue scrolling.

**Syntax:** \$ls -a\p

**MORE :** It also displays the file page by page .To continue scrolling with more command , press the space bar key.

**Syntax:** \$more<filename>

**GREP :** This command is used to search and print the specified patterns from the file.

**Syntax:** \$grep [option] pattern <filename>

**SORT :** This command is used to sort the datas in some order.

**Syntax:** \$sort<filename>



**PIPE :** It is a mechanism by which the output of one command can be channeled into the input of another command.

**Syntax:** \$who | wc-l

**TR :** The tr filter is used to translate one set of characters from the standard inputs to another.

**Syntax:** \$tr "[a-z]" "[A-Z]"

### **COMMUNICATION THROUGH UNIX COMMANDS**

#### **MESG**

Description: The message command is used to give permission to other users to send message to your terminal.

Syntax: \$mesg y

#### 2.Command: **WRITE**

Description: This command is used to communicate with other users, who are logged in at the same time.

Syntax: \$write <user name>

#### 3.Command: **WALL**

Description: This command sends message to all users those who are logged in using the unix server.

Syntax: \$wall <message>

#### 4.Command: **MAIL**

Description: It refers to textual information, that can be transferred from one user to another

Syntax: \$mail <user name>

#### 5.Command: **REPLY**

Description: It is used to send reply to specified user.

Syntax: \$reply<user name>

### **RESULT:**

**EX.NO :1.b**

**vi EDITOR COMMANDS**

**DATE :**

**AIM :**

To study the various commands operated in vi editor in UNIX.

**DESCRIPTION :**

The Vi editor is a visual editor used to create and edit text, files, documents and programs. It displays the content of files on the screen and allows a user to add, delete or change part of text . There are three modes available in the Vi editor , they are

- 1.Command mode
- 2.Input (or) insert mode.

**Starting Vi :**

The Vi editor is invoked by giving the following commands in UNIX prompt.

**Syntax :**      \$vi <filename> (or)  
                     \$vi

This command would open a display screen with 25 lines and with tilt (~) symbol at the start of each line. The first syntax would save the file in the filename mentioned and for the next the filename must be mentioned at the end.

**Options :**

- 1.vi +n <filename> - this would point at the nth line (cursor pos).
- 2.vi -n <filename> - This command is to make the file to read only to change from one mode to another press escape key.

**INSERTING AND REPLACING COMMANDS :**

To move editor from command node to edit mode, you have to press the <ESC> key. For inserting and replacing the following commands are used.

**1.ESC a Command :**

This command is used to move the edit mode and start to append after the current character.

**Syntax :**      <ESC> a

**2.ESC A COMMAND :**

This command is also used to append the file , but this command append at the end of current line.

**Syntax :**      <ESC> A

### **3.ESC i Command :**

This command is used to insert the text before the current cursor position.

**Syntax :** <ESC> i

### **4.ESC I Command :**

This command is used to insert at the beginning of the current line.

**Syntax :** <ESC> I

### **5.ESC o Command :**

This command is insert a blank line below the current line & allow insertion of contents.

**Syntax :** <ESC> o

### **6.ESC O Command :**

This command is used to insert a blank line above & allow insertion of contents.

**Syntax :** <ESC> O

### **7.ESC r Command :**

This command is to replace the particular character with the given characters.

**Syntax :** <ESC> rx Where x is the new character.

### **8.ESC R Command :**

This command is used to replace the particular text with a given text.

**Syntax :** <ESC> R text

### **9.<ESC> s Command :**

This command replaces a single character with a group of character .

**Syntax :** <ESC> s

### **10.<ESC> S Command :**

This command is used to replace a current line with group of characters.

**Syntax :** <ESC> S

## **CURSOR MOVEMENT IN vi :**

### **1.<ESC> h :**

This command is used to move to the previous character typed. It is used to move to left of the text . It can also used to move character by character (or) a number of characters.

**Syntax :** <ESC> h – to move one character to left.  
<ESC> nh – to move 'n' character to left.

### **2.<ESC> l :**

This command is used to move to the right of the cursor (ie) to the next character. It can also be used to move the cursor for a number of character.

**Syntax :** <ESC> l – single character to right.  
<ESC> nl - 'n' characters to right.

### **3.<ESC> j :**

This command is used to move down a single line or a number of lines.

#### **Syntax :**

<ESC> j – single down movement.  
<ESC> nj – ‘n’ times down movement.

### **4.<ESC> k :**

This command is used to move up a single line or a number of lines.

#### **Syntax :**

<ESC> k – single line above.  
<ESC> nk – ‘n’ lines above.

### **5.ENTER (OR) N ENTER :**

This command will move the cursor to the starting of next lines or a group of lines mentioned.

#### **Syntax :**

<ESC> enter                      <ESC> n enter.

### **6.<ESC> + Command :**

This command is used to move to the beginning of the next line.

#### **Syntax :**

<ESC> +                      <ESC> n+

### **7.<ESC> - Command :**

This command is used to move to the beginning of the previous line.

#### **Syntax :**

<ESC> -                      <ESC> n-

### **8.<ESC> 0 :**

This command will bring the cursor to the beginning of the same current line.

#### **Syntax :**

<ESC> 0

### **9.<ESC> \$ :**

This command will bring the cursor to the end of the current line.

#### **Syntax :**

<ESC> \$

### **10.<ESC> ^ :**

This command is used to move to first character of first lines.

#### **Syntax :**

<ESC> ^

### **11.<ESC> b Command :**

This command is used to move back to the previous word (or) a number of words.

#### **Syntax :**

<ESC> b                      <ESC>nb

### **12.<ESC> e Command :**

This command is used to move towards and replace the cursor at last character of the word (or) no of words.

#### **Syntax :**

<ESC> e                      <ESC>ne

### **13.<ESC> w Command :**

This command is used to move forward by a single word or a group of words.

#### **Syntax :**

<ESC> w                      <ESC> nw

### **DELETING THE TEXT FROM vi :**

#### **1.<ESC> x Command :**

To delete a character to right of current cursor positions , this command is used.

#### **Syntax :**

<ESC> x                      <ESC> nx

#### **2.<ESC> X Command :**

To delete a character to left of current cursor positions , this command is used.

#### **Syntax :**

<ESC> X                      <ESC> nX

#### **3.<ESC> dw Command :**

This command is to delete a single word or number of words to right of current cursor position.

#### **Syntax :**

<ESC> dw                      <ESC> ndw

#### **4.db Command :**

This command is to delete a single word to the left of the current cursor position.

#### **Syntax :**

<ESC> db                      <ESC> ndb

#### **5.<ESC> dd Command :**

This command is used to delete the current line (or) a number of line below the current line.

#### **Syntax :**

<ESC> dd                      <ESC> ndd

#### **6.<ESC> d\$ Command :**

This command is used to delete the text from current cursor position to last character of current line.

**Syntax :**              <ESC> d\$

### **SAVING AND QUITTING FROM vi :-**

#### **1.<ESC> w Command :**

To save the given text present in the file.

**Syntax :**              <ESC> : w

### **2.<ESC> q! Command :**

To quit the given text without saving.

**Syntax :**       <ESC> :q!

### **3.<ESC> wq Command :**

This command quits the vi editor after saving the text in the mentioned file.

**Syntax :**       <ESC> :wq

### **4.<ESC> x Command :**

This command is same as 'wq' command it saves and quit.

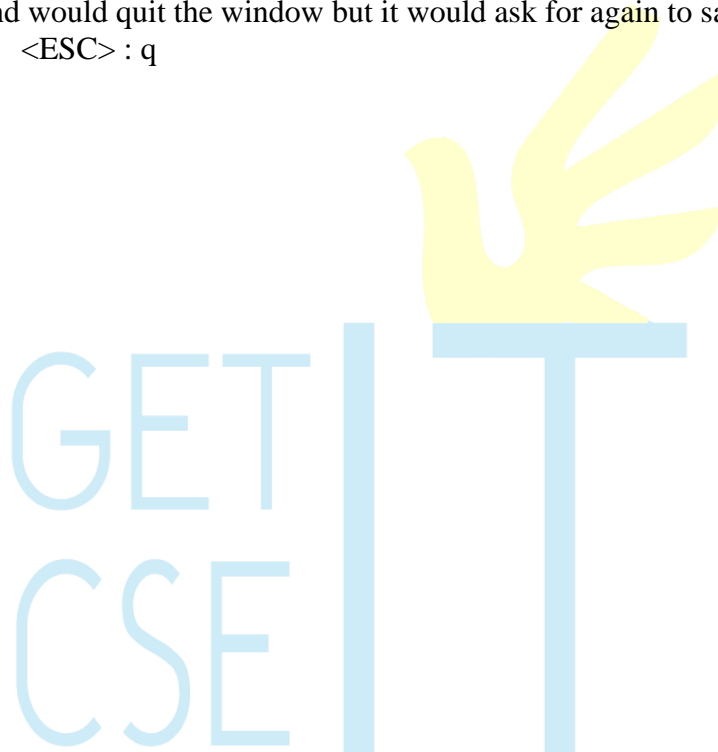
**Syntax :**       <ESC> :x

### **5.<ESC> q Command :**

This command would quit the window but it would ask for again to save the file.

**Syntax :**       <ESC> :q

### **RESULT:**



**EX.NO :1.c**

## **UNIX SHELL PROGRAMMING COMMANDS.**

**DATE :**

**AIM :**

To study about the Unix Shell Programming Commands.

### **INTRODUCTION :**

Shell programming is a group of commands grouped together under single filename. After logging onto the system a prompt for input appears which is generated by a Command String Interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. The shell can be used either interactively - enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled.

#### **Common Shells.**

**C-Shell - csh :** The default on teaching systems Good for interactive systems Inferior programmable features

**Bourne Shell - bsh or sh - also restricted shell - bsh :** Sophisticated pattern matching and file name substitution

**Korn Shell :** Backwards compatible with Bourne Shell Regular expression substitution emacs editing mode

**Thomas C-Shell - tcsh :** Based on C-Shell Additional ability to use emacs to edit the command line Word completion & spelling correction Identifying your shell.

### **01. SHELL KEYWORDS :**

echo, read, if fi, else, case, esac, for , while , do , done, until , set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

### **02. General things SHELL**

**The shbang line** The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.

#### **EXAMPLE**

#!/bin/sh

**Comments** Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.

#### **EXAMPLE**

# this text is not

# interpreted by the shell

**Wildcards** There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the \*, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

#### **EXAMPLE**

Filename expansion:  
rm \*; ls ??; cat file[1-3];  
Quotes protect metacharacter:  
echo "How are you?"

### **03. SHELL VARIABLES :**

Shell variables change during the execution of the program .The C Shell offers a command "Set" to assign a value to a variable.

For example:

```
% set myname= Fred
% set myname = "Fred Bloggs"
% set age=20
```

A \$ sign operator is used to recall the variable values.

For example:

```
% echo $myname will display Fred Bloggs on the screen
```

A @ sign can be used to assign the integer constant values.

For example:

```
% @myage=20
% @age1=10
% @age2=20
% @age=$age1+$age2
% echo $age
```

#### **List variables**

```
% set programming_languages= (C LISP)
% echo $programming_languages
```

#### **C LISP**

```
% set files=*. *
% set colors=(red blue green)
% echo $colors[2]
blue
% set colors=($colors yellow)/add to list
```

**Local variables** Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

#### **EXAMPLE**

```
variable_name=value
name="John Doe"
x=5
```

**Global variables** Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.

#### **EXAMPLE**

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```



**Extracting values from variables** To extract the value from variables, a dollar sign is used.

**EXAMPLE**

```
echo $variable_name
echo $name
echo $PATH
```

**Rules : -**

1. A variable name is any combination of alphabets, digits and an underscore ('-');
2. No commas or blanks are allowed within a variable name.
3. The first character of a variable name must either be an alphabet or an underscore.
4. Variables names should be of any reasonable length.
5. Variables name are case sensitive . That is , Name, NAME, name, NAmE, are all different variables.

**04. EXPRESSION Command :**

To perform all arithmetic operations .

**Syntax :**

Var = 'expr\$value1' + \$ value2'

**Arithmetic**

The Bourne shell does not support arithmetic. UNIX/Linux commands must be used to perform calculations.

**EXAMPLE**

```
n=`expr 5 + 5`
echo $n
```

**Operators**

The Bourne shell uses the built-in test command operators to test numbers and strings.

**EXAMPLE**

Equality:

```
=      string
!=     string
-eq    number
-ne    number
```

Logical:

```
-a     and
-o     or
!      not
```

Logical:

```
AND &&
OR  ||
```

Relational:

```
-gt    greater than
-ge    greater than, equal to
```

-lt    *less than*  
-le    *less than, equal to*

Arithmetic :  
+, -, \\*, /, %

**Arguments (positional parameters)** Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script.

**EXAMPLE**

At the command line:

\$ scriptname arg1 arg2 arg3 ...

In a script:

echo \$1 \$2 \$3    *Positional parameters*

echo \$\*            *All the positional parameters*

echo \$#            *The number of positional parameters*

## **05.READ Statement :**

To get the input from the user.

**Syntax :**

read x y  
(no need of commas between variables)

## **06. ECHO Statement :**

Similar to the output statement. To print output to the screen, the echo command is used. Wildcards must be escaped with either a backslash or matching quotes.

**Syntax :**

Echo "String" (or) echo \$ b(for variable).

**EXAMPLE**

echo "What is your name?"

**Reading user input** The read command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The read command can accept multiple variable names. Each variable will be assigned a word.

**EXAMPLE**

echo "What is your name?"  
read name  
read name1 name2 ...

## **6. CONDITIONAL STATEMENTS :**

The if construct is followed by a command. If an expression is to be tested, it is enclosed in square brackets. The then keyword is placed after the closing parenthesis. An if must end with a fi.

**Syntax :**

1.if

This is used to check a condition and if it satisfies the condition if then does the next action , if not it goes to the else part.

## 2.if...else

### Syntax :

```
If cp $ source $ target
Then
    Echo File copied successfully
Else
    Echo Failed to copy the file.
```

## 3.nested if

here sequence of condition are checked and the corresponding performed accordingly.

### Syntax :

```
if condition
then
    command
    if condition
    then
        command
    else
        command
fi
fi
```

## 4.case ..... esac

This construct helps in execution of the shell script based on Choice.

### EXAMPLE

#### The if construct is:

```
if command
then
    block of statements
fi
```

```
-----
if [ expression ]
then
    block of statements
fi
```

#### The if/else/else if construct is:

```
if command
then
    block of statements
elif command
then
    block of statements
elif command
then
```

#### The case command construct is:

```
case variable_name in
    pattern1)
        statements
        ;;
    pattern2)
        statements
        ;;
    pattern3)
        ;;
    *) default value
        ;;
esac
case "$color" in
    blue)
        echo $color is blue
        ;;
    green)
        echo $color is green
        ;;
```

```
    block of statements
else
    block of statements
fi
-----
```

```
if [ expression ]
then
    block of statements
elif [ expression ]
then
    block of statements
elif [ expression ]
then
    block of statements
else
    block of statements
fi
-----
```

```
red|orange)
    echo $color is red or orange
;;
*) echo "Not a color" # default
esac
```

## **The if/else construct is:**

```
if [ expression ]
then
    block of statements
else
    block of statements
fi
-----
```

## **07. LOOPS**

There are three types of loops: while, until and for. The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed.

The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false.

The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed by a variable name, the in keyword, and a list of words then a block of statements, and terminates with the done keyword.

The loop control commands are break and continue.

### **EXAMPLE**

```
while command
do
```

```
    block of statements
```

```
done
-----
```

```
while [ expression ]
do
```

```
    block of statements
```

```
done
```

```
until command
```

```
do
```

```
    block of statements
```

```
done
-----
```

```
until [ expression ]
```

```
for variable in word1 word2 word3 ...
```

```
do
```

```
    block of statements
```

```
done
```

```
do
    block of statements
done
-----
until control command
do
    commands
done
```

## **08. Break Statement :**

This command is used to jump out of the loop instantly, without waiting to get the control command.

## **09. ARRAYS**

**(positional parameters)** The Bourne shell does support an array, but a word list can be created by using positional parameters. A list of words follows the built-in set command, and the words are accessed by position. Up to nine positions are allowed. The built-in shift command shifts off the first word on the left-hand side of the list. The individual words are accessed by position values starting at 1.

### **EXAMPLE**

```
set word1 word2 word3
echo $1 $2 $3      Displays word1, word2, and word3
set apples peaches plums
shift              Shifts off apples
echo $1            Displays first element of the list
echo $2            Displays second element of the list
echo $*            Displays all elements of the list
```

**Command substitution** To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, backquotes are used.

### **EXAMPLE**

```
variable_name=`command`
echo $variable_name
now=`date`
echo $now
echo "Today is `date`"
```

## **10. FILE TESTING**

The Bourne shell uses the test command to evaluate conditional expressions and has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth.

### **EXAMPLE**

```
-d File is a directory
-f File exists and is not a directory
-r Current user can read the file
-s File is of nonzero size
-w Current user can write to the file
```

*-x Current user can execute the file*

```
#!/bin/sh
1 if [ -f file ]
  then
    echo file exists
  fi

2 if [ -d file ]
  then
    echo file is a directory
  fi

3 if [ -s file ]
  then
    echo file is not of zero length
  fi

4 if [ -r file -a -w file ]
  then
    echo file is readable and writable
  fi
```

### **11. EXECUTION OF SHELL SCRIPT :**

- 1.By using change mode command
  - 2.\$ chmod u + x sum.sh
  - 3.\$ sum.sh
- or
- \$ sh sum.sh

### **RESULT:**

## SHELL PROGRAMMING

**Ex.No :2a**

**CONCATENATION OF TWO STRINGS**

**Date:**

**Aim:**

To write a shell program to concatenate two strings.

**Algorithm:**

Step1: Enter into the vi editor and go to the insert mode for entering the code

Step2: Read the first string.

Step3: Read the second string

Step4: Concatenate the two strings

Step5: Enter into the escape mode for the execution of the result and verify the output

**Program:**

```
echo "enter the first string"
read str1
echo "enter the second string"
read str2
echo "the concatenated string is" $str1$str2
```

**Sample I/P:**

Enter first string: Hello  
Enter first string: World

**Sample O/P:**

The concatenated string is HelloWorld

**Result:**

Thus the shell program to concatenate two strings is executed and output is verified successfully.

**Ex.No. :2b**

## **COMPARISON OF TWO STRINGS**

**Date:**

**Aim:**

To write a shell program to compare the two strings.

**Algorithm:**

Step1: Enter into the vi editor and go to the insert mode for entering the code

Step2: Read the first string.

Step3: Read the second string

Step4: Compare the two strings using the if loop

Step5: If the condition satisfies then print that two strings are equal else print two strings are not equal.

Step6: Enter into the escape mode for the execution of the result and verify the output

**Program:**

```
echo "enter the first string"
read str1
echo "enter the second string"
read str2
if [ $str1 = $str2 ]
then
echo "strings are equal"
else
echo "strings are unequal"
fi
```

**Sample I/P:1**

Enter first string: hai

Enter second string: hai

**Sample O/P:1**

The two strings are equal

**Sample I/P:2**

Enter first string: hai

Enter second string: cse

**Sample O/P:2**

The two strings are not equal

**Result:**

Thus the shell program to compare the two strings is executed and output is verified successfully.



**Ex.No:2c**

## **MAXIMUM OF THREE NUMBERS**

**Date:**

**Aim:**

To write a shell program to find greatest of three numbers.

**Algorithm:**

- Step1: Declare the three variables.
- Step2: Check if A is greater than B and C.
- Step3: If so print A is greater.
- Step4: Else check if B is greater than C.
- Step5: If so print B is greater.
- Step6: Else print C is greater.

**Program:**

```
echo "enter A"
read a
echo "enter B"
read b
echo "enter C"
read c
if [ $a -gt $b -a $a -gt $c ]
then
echo "A is greater"
elif [ $b -gt $a -a $b -gt $c ]
then
echo "B is greater"
else
echo "C is greater"
fi
```

**Sample I/P:**

```
Enter A:23
Enter B:45
Enter C:67
```

**Sample O/P:**

C is greater

**Result:**

Thus the shell program to find the maximum of three numbers is executed and output is verified successfully.

**Ex.No:2d**

## **FIBONACCI SERIES**

**Date:**

**Aim:**

To write a shell program to generate fibonacci series.

**Algorithm :**

- Step 1 : Initialise a to 0 and b to 1.
- Step 2 : Print the values of 'a' and 'b'.
- Step 3 : Add the values of 'a' and 'b'. Store the added value in variable 'c'.
- Step 4 : Print the value of 'c'.
- Step 5 : Initialise 'a' to 'b' and 'b' to 'c'.
- Step 6 : Repeat the steps 3,4,5 till the value of 'a' is less than 10.

**Program :**

```
echo enter the number
read n
a=-1
b=1
i=0
while [ $i -le $n ]
do
t=`expr $a + $b`
echo $t
a=$b
b=$t
i=`expr $i + 1`
done
```

**Sample I/P :**

Enter the no: 5

**Sample O/P:**

0  
1  
1  
2  
3  
5

**Result :**

Thus the shell program to find the fibonacci series is executed and output is verified successfully.

**Ex.No:2e**

## **ARITHMETIC OPERATIONS USING CASE**

**Date:**

**Aim:**

To write a shell program to perform the arithmetic operations using case

**Algorithm :**

- Step 1 : Read the input variables and assign the value
- Step 2 : Print the various arithmetic operations which we are going to perform
- Step 3 : Using the case operator assign the various functions for the arithmetic operators.
- Step 4 : Check the values for all the corresponding operations.
- Step 5 : Print the result and stop the execution.

**Program :**

```
echo 1.Addition
echo 2.Subraction
echo 3.Multiplication
echo 4.Division
echo enter your choice
read a
echo enter the value of b
read b
echo enter the value of c
read c
echo b is $b c is $c
case $a in
1)d=`expr $b + $c`
echo the sum is $d
;;
2)d=`expr $b - $c`
echo the difference is $d
;;
3)d=`expr $b \* $c`
echo the product is $d
;;
4)d=`expr $b / $c`
echo the quotient is $d
;;
esac
```

**Sample I/P :**

- 1.Addition
- 2.Subraction
- 3.Multiplication

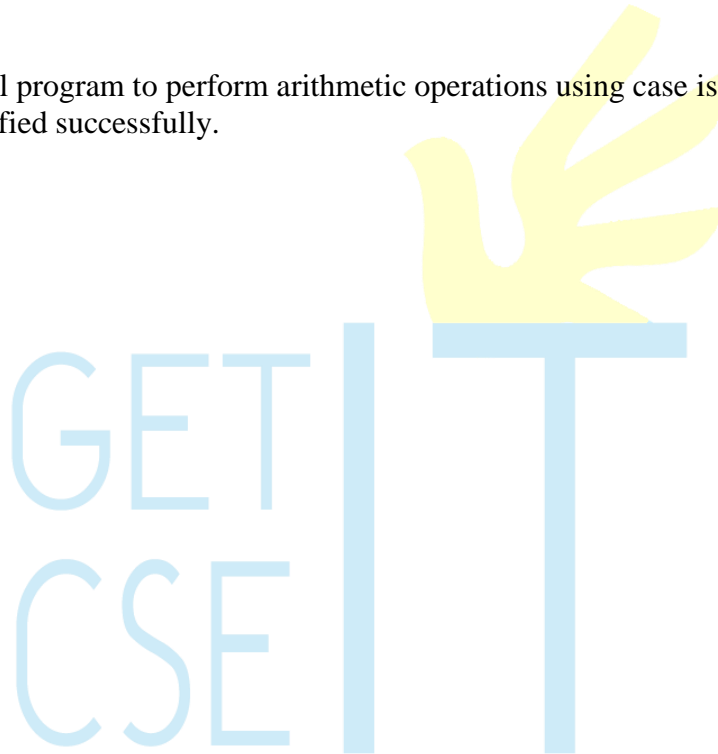
Division  
Enter your choice:1  
Enter the value of b:3  
Enter the value of c:4  
b is 3 c is 4  
the sum is 7

**Sample O/P:**

b is 3 c is 4  
the sum is 7

**Result :**

Thus the shell program to perform arithmetic operations using case is executed and output is verified successfully.



## **SYSTEM CALLS** **PROCESS CREATION**

**Ex.No:3a**

**Date:**

**AIM:**

To write a program to create a process in UNIX.

**ALGORITHM:**

STEP 1: Start the program.

STEP 2: Declare pid as integer.

STEP 3: Create the process using Fork command.

STEP 4: Check pid is less than 0 then print error else if pid is equal to 0 then execute command else parent process wait for child process.

STEP 5: Stop the program.

**PROGRAM:**

```
void main()
{
    int id;
    id=fork();
    if(id<0)
    {
        Printf("cannot create the file");
        Exit(-1);
    }
    if(id==0)
    {
        Printf("child process");
        Exit(0);
    }
    else
    {
        Printf("parent process");
    }
}
```

**SAMPLE OUTPUT:**

```
$cc pc.c
$a.out
parent process$ child process
$ps
  PID  CLS  PRI  TTY      TIME  CMD
 5913   TS   70  pts022  0:00  ksh
 6229   TS   59  pts022  0:00  ps
```

**RESULT:** Thus the program was executed and verified successfully

**Ex.No:3b**

## **EXECUTING A COMMAND**

**Date:**

**AIM:**

To write a program for executing a command.

**ALGORITHM:**

STEP 1: Start the program.

STEP 2: Execute the command in the shell program using exec ls.

STEP 3: Stop the execution.

**PROGRAM:**

echo Program for executing UNIX command using shell programming

echo Welcome

ps

exec wc e

**SAMPLE OUTPUT:**

\$ sh exec.sh

program for executing UNIX command using shell programming

Welcome

PID	CLS	PRI	TTY	TIME	COMD
-----	-----	-----	-----	------	------

958	TS	70	pts001	0:00	ksh
-----	----	----	--------	------	-----

971	TS	70	pts001	0:00	sh
-----	----	----	--------	------	----

972	TS	59	pts001	0:00	ps
-----	----	----	--------	------	----

3 41 81 e

**RESULT:**

Thus the program was executed and verified successfully.

**Ex.No:3c**

## **SLEEP COMMAND**

**Date:**

**AIM:**

To create child with sleep command.

**ALGORITHM:**

STEP 1: Start the program.

STEP 2: Create process using fork and assign into a variable.

STEP 3: If the value of variable is  $< 0$  print not create and  $> 0$  process create and else print child create.

STEP 4: Create child with sleep of 2.

STEP 5: Stop the program.

**PROGRAM:**

```
void main()
{
    int id=fork();
    if(id==-1)
    {
        printf("cannot create the file");
        exit(1);
    }
    else if(id==0)
    {
        sleep(2);
        printf("this is child process");
    }
    else
    {
        printf("parent process");
        exit(1);
    }
}
```

### **OUTPUT:**

```
$ cc sleep.c
```

```
$ a.out
```

```
parent process$ this is child process
```

### **RESULT:**

Thus the program was executed and verified successfully.





**Ex.No:3d**

## **SLEEP COMMAND USING GETPID**

**Date:**

**AIM:**

To create child with sleep command using getpid.

**ALGORITHM:**

STEP 1: Start the execution and create a process using fork() command.

STEP 2: Make the parent process to sleep for 10 seconds.

STEP 3: In the child process print its pid and its corresponding pid.

STEP 4: Make the child process to sleep for 5 seconds.

STEP 5: Again print its pid and its parent pid.

STEP 6: After making the sleep for the parent process for 10 seconds print its pid.

STEP 7: Stop the execution.

**PROGRAM:**

```
void main()
{
    int pid;
    pid=fork();
    if (pid==0)
    {
        printf("\n Child Process\n");
        printf("\n Child Process id is %d ",getpid());
        printf("\n Its parent process id is %d",getppid());
        sleep(5);

        printf("\n Child process after sleep=5\n");
        printf("\n Child Process id is %d ",getpid());
        printf("\n Its parent process id is %d",getppid());
    }
    else
    {
        printf("\nParent process");
    }
}
```

```
        sleep(10);
        printf("\n Child Process id is %d ",getpid());
        printf("\n Its parent process id is %d",getppid());
        printf("\nParent terminates\n");
    }
}
```

### **OUTPUT:**

```
$ cc sleepid.c
$ a.out
parent process
child process
child process id is 12691
its parent process id is 12690
child process after sleep=5
child process id is 12691
its parent process id is 12690
child process after sleep=10
child id is 12690
parent id is 11383
parent terminates
$
```

### **RESULT:**

Thus the program was executed and verified successfully.

**Ex.No:3e**

## **SIGNAL HANDLING**

**Date:**

**AIM:**

To write a program for signal handling in UNIX.

**ALGORITHM:**

STEP 1: start the program

STEP 2: Read the value of pid.

STEP 3: Kill the command surely using kill-9 pid.

STEP 4: Stop the program.

**PROGRAM:**

echo program for performing KILL operations

ps

echo enter the pid

read pid

kill-9 \$pid

echo finished

**OUTPUT:**

\$sh kill.sh

program for performing KILL operations

PID	CLS	PRI	TTY	TIME	COMD
858	TS	70	pts001	0:00	ksh
858	TS	70	pts001	0:00	sh
858	TS	59	pts001	0:00	ps

enter the pid

872

killed

\$sh kill.sh

program for performing KILL operations

PID	CLS	PRI	TTY	TIME	COMD
858	TS	70	pts001	0:00	ksh

```
858 TS 70 pts001 0:00 sh
858 TS 59 pts001 0:00 ps
```

enter the pid

876

UX:kill(kill.sh):ERROR: no such proccess

\$sh kill.sh

program for performing KILL operations

```
PID CLS PRI TTY      TIME COMD
858 TS 70 pts001 0:00 ksh
858 TS 70 pts001 0:00 sh
858 TS 59 pts001 0:00 ps
```

enter the pid

858

finished

### **RESULT:**

Thus the program was executed and verified successfully.

**Ex.No:3f**

## **WAIT COMMAND**

**Date:**

**AIM:**

To perform wait command using c program.

**ALGORITHM:**

STEP 1:Start the execution

STEP 2:Create process using fork and assign it to a variable

STEP 3:Check for the condition pid is equal to 0

STEP 4:If it is true print the value of i and terminate the child process

STEP 5:If it is not a parent process has to wait until the child terminate

STEP 6:Stop the execution

**PROGRAM:**

```
int i=10;
void main()
{
    int pid=fork();
    if(pid==0)
    {
        printf("initial value of i %d \n ",i);
        i+=10;
        printf("value of i %d \n ",i);
        printf("child terminated \n");
    }
    else
    {
        wait(0);
        printf("value of i in parent process %d",i);
    }
}
```

**OUTPUT:**

\$cc wait.c

\$a.out

initial value of i 10

value of i 20

child terminated

value of i in parent process 10\$

### **RESULT:**

Thus the program was executed and verified successfully.



**ExNo:4a**

## **READING FROM A FILE**

**Date:**

### **AIM:**

To create the file, read data from the file, update the file.

### **ALGORITHM:**

1. Get the data from the user.
2. Open a file.
3. Read from the file.
4. Close the file.

### **PROGRAM:**

```
#include<stdio.h>
int main()
{
    char str[100];
    FILE *fp;
    fp=fopen("file1.dat","r");
    while(!feof(fp))
    {
        fscanf(fp,"%s",str);
        printf(" %s ",str);
    }
    fclose(fp);
}
```

### **OUTPUT:**

```
$ vi read1.c
$ gcc read1.c
$ ./a.out
hai this is a program to read the content of the file.
```

### **RESULT:**

Thus C program to write data into a file was executed successfully

**ExNo:4b**

## **WRITING INTO A FILE**

**Date**

**AIM:**

To write a C program to write the data into a file.

**ALGORITHM:**

- Step1. Get the data from the user.
- Step2. Open a file.
- Step3. Write the data from the file.
- Step4. Get the data and update the file.

**PROGRAM:**

```
#include<stdio.h>
int main()
{
    char str[100];
    FILE *fp;
    printf("Enter the string");
    gets(str);
    fp=fopen("file1.dat","w+");
    while(!feof(fp))
    {
        fscanf(fp,"%s",str);
    }
    fprintf(fp,"%s",str);
}
```

**OUTPUT:**

```
$ gcc write.c
$ ./a.out
Enter the string: os lab
$ vi file1.dat
```

os lab

**RESULT:**

Thus C program to write data into a file was executed successfully.



**ExNo:4c**

## **FILE CREATION**

**Date:**

**AIM:**

To write a C program to create a file.

**ALGORITHM:**

Step1:Start the program.

Step2:Create the file using create function and assign a variable to it.

Step3:If the value of the variable is less then print file cannot be created ,otherwise print file is created.

Step4:Stop the program.

**PROGRAM:**

```
void main()
{
    int id;
    if(id=creat("z.txt",0)==-1)
    {
        printf("cannot create the file");
        exit(1);
    }
    else
    {
        printf("file is created");
        exit(1);
    }
}
```

**OUTPUT:**

\$ cc fc.c

\$ a.out

file is created \$

**RESULT:**

Thus the C program to create a file was executed successfully.

**Ex. No:5a**

## **IMPLEMENTATION OF ls COMMAND**

**Date:**

### **AIM:**

To write a C program to simulate the operation of “ls” command in Unix.

### **ALGORITHM:**

1. Check if the number of command line arguments is less than 2. If yes, Print error and exit.
2. Check if the second argument (i.e. directory to be listed) is valid or not. If not then exit.
3. Print the content of the directory till it becomes NULL.
4. Close the directory entry file.

### **PROGRAM:**

```
#include<stdio.h>
#include<sys/types.h>
#include<dirent.h>
main(int argc, char *argv[])
{
    DIR *dp;
    struct dirent *dirp;
    if(argc<2)
    {
        printf("\n You have provided only 1 argument\n");
        exit(1);
    }
    if((dp=opendir(argv[1]))==NULL)
    {
        printf("\nCannot open %s file!\n",argv[1]);
        exit(1);
    }
    while((dirp=readdir(dp))!=NULL)
        printf("%s\n",dirp->d_name);
    closedir(dp);
}
```

### **OUTPUT:**

```
[root@lab1cab01 orbit]# gcc lsdemo.c
[root@lab1cab01 orbit]# ./a.out
```

```
You have provided only 1 argument
[root@lab1cab01 orbit]# ./a.out /xyz/foo
```

```
Cannot open /xyz/foo file!  
[root@lab1cab01 orbit]# ./a.out /root  
[root@lab1cab01 orbit]# ./a.out /tmp/orbit  
.  
..  
sem.c  
first.c  
best.c  
a.out  
lsdemo.c  
roundrobin.c  
bestop.doc  
firstop.doc  
roundrobin.doc  
[root@lab1cab01 orbit]#
```

**RESULT:**

Thus the program for implementing the ls command was executed and the output was verified successfully.

**ExNo:6a**

**FIRST COME FIRST SERVE**

**Date :**

**AIM:**

To write a C program to implement the CPU scheduling algorithm for FIRST COME FIRST SERVE.

**PROBLEM DESCRIPTION:**

Cpu scheduler will decide which process should be given the CPU for its execution. For this it uses different algorithms to choose among the processes. One among those algorithms is FCFS algorithm.

In this algorithm the process which arrives first is given the CPU after finishing its request. Only then it will allow the CPU to execute other processes.

**ALGORITHM:**

Step1: Create the number of processes.

Step2: Get the ID and Service time for each process.

Step3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.

Step4: Calculate the Total time and Processing time for the remaining processes.

Step5: Waiting time of one process is the Total time of the previous process.

Step6: Total time of process is calculated by adding Waiting time and Service time.

Step7: Total waiting time is calculated by adding the waiting time for each process.

Step8: Total turn around time is calculated by adding all total times of each process.

Step9: Calculate Average waiting time by dividing the total waiting time by total number of processes.

Step10: Calculate Average turn around time by dividing the total time by the number of processes.

Step11: Display the result.

**PROGRAM**

```
#include<stdio.h>
struct process
{
```

```
int id,wait,ser,tottime;
}p[20];

main()
{
    int i,n,j,totalwait=0,totalser=0,avturn,avwait;
    printf("enter number of process");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        printf("enter process_id");
        scanf("%d",&p[i].id);
        printf("enter process service time");
        scanf("%d",&p[i].ser);
    }
    p[1].wait=0;
    p[1].tottime=p[1].ser;

    for(i=2;i<=n;i++)
    {
        for(j=1;j<i;j++)
        {
            p[i].wait=p[i].wait+p[j].ser;
        }
        totalwait=totalwait+p[i].wait;
        p[i].tottime=p[i].wait+p[i].ser;
        totalser=totalser+p[i].tottime;
    }
    avturn=totalser/n;
    avwait=totalwait/n;
    printf("Id\tservice\twait\ttotal");
    for(i=1;i<=n;i++)
    {
        printf("\n%d\t%d\t%d\t%d\n",p[i].id,p[i].ser,p[i].wait,p[i].tottime);
    }
    printf("average waiting time %d\n",avwait);
    printf("average turnaround time %d\n",avturn);
}
```

### **OUTPUT**

```
[admin@lab1cab5 admin]$ gcc fcfs.c
[admin@lab1cab5 admin]$ ./a.out
enter number of process4
enter process_id901
enter process service time4
```

```
enter process_id902
enter process service time3
enter process_id903
enter process service time5
enter process_id904
enter process service time2
Id  service wait  total
901  4    0    4

902  3    4    7

903  5    7   12

904  2   12   14
average waiting time 5
average turnaround time 8
```

### **RESULT:**

Thus the C program to implement CPU scheduling algorithm for first come first serve was executed successfully.

**ExNo:6b**

**SHORTEST JOB FIRST**

**Date :**

**AIM:**

To write a C program to implement the CPU scheduling algorithm for Shortest job first.

**PROBLEM DESCRIPTION:**

Cpu scheduler will decide which process should be given the CPU for its execution. For this it use different algorithm to choose among the process. one among that algorithm is sjf algorithm.

In this algorithm the process which has less service time given the cpu after finishing its request only it will allow cpu to execute next other process.

**ALGORITHM:**

Step1: Get the number of process.

Step2: Get the id and service time for each process.

Step3: Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.

Step4: Calculate the total time and waiting time of remaining process.

Step5: Waiting time of one process is the total time of the previous process.

Step6: Total time of process is calculated by adding the waiting time and service time of each process.

Step7: Total waiting time calculated by adding the waiting time of each process.

Step8: Total turn around time calculated by adding all total time of each process.

Step9: calculate average waiting time by dividing the total waiting time by total number of process.

Step10: Calculate average turn around time by dividing the total waiting time by total number of process.

Step11: Display the result.

**PROGRAM:**

```
#include<stdio.h>
struct ff
{
    int pid,ser,wait;
}p[20];
```

```
struct ff tmp;
main()
{
    int i,n,j,tot=0,avwait,totwait=0,tturn=0,aturn;
    printf("enter the number of process");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("enter process id");
        scanf("%d",&p[i]);
        printf("enter service time");
        scanf("%d",&p[i].ser);
        p[i].wait=0;
    }
    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(p[i].ser>p[j].ser)
            {
                tmp=p[i];
                p[i]=p[j];
                p[j]=tmp;
            }
        }
    }
    printf("PID\tSER\tWAIT\tTOT\n");
    for(i=0;i<n;i++)
    {
        tot=tot+p[i].ser;
        tturn=tturn+tot;
        p[i+1].wait=tot;
        totwait=totwait+p[i].wait;
        printf("%d\t%d\t%d\t%d\n",p[i].pid,p[i].ser,p[i].wait,tot);
    }
    avwait=totwait/n;
    aturn=tturn/n;
    printf("TOTAL WAITING TIME :%d\n",totwait);
    printf("AVERAGE WAITING TIME : %d\n",avwait);
    printf("TOTAL TURNAROUND TIME :%d\n",tturn);
    printf("AVERAGE TURNAROUND TIME:%d\n",aturn);
}
```

### OUTPUT :

```
[root@lab1cab5 home]# ./a.out
enter the number of process4
```



```
enter process id701
enter service time6
enter process id702
enter service time4
enter process id703
enter service time8
enter process id704
enter service time1
PID  SER  WAIT TOT
704  1    0    1
702  4    1    5
701  6    5   11
703  8   11   19
TOTAL WAITING TIME :17
AVERAGE WAITING TIME : 4
TOTAL TURNAROUND TIME :36
AVERAGE TURNAROUND TIME:9
```

**Result:**

Thus the C program to implement the CPU scheduling algorithm for shortest job first was executed successfully.

**Ex.No :7a**

## **ROUND ROBIN**

**Date:**

**AIM :**

To write a C program to simulate the CPU scheduling algorithm for round robin

### **PROBLEM DESCRIPTION:**

CPU scheduler will decide which process should be given the CPU for its execution .For this it use different algorithm to choose among the process .one among that algorithm is Round robin algorithm.

In this algorithm we are assigning some time slice .The process is allocated according to the time slice ,if the process service time is less than the time slice then process itself will release the CPU voluntarily .The scheduler will then proceed to the next process in the ready queue .If the CPU burst of the currently running process is longer than time quantum ,the timer will go off and will cause an interrupt to the operating system .A context switch will be executed and the process will be put at the tail of the ready queue.

### **ALGORITHM:**

Step 1: Initialize all the structure elements

Step 2: Receive inputs from the user to fill process id,burst time and arrival time.

Step 3: Calculate the waiting time for all the process id.

i) The waiting time for first instance of a process is calculated as:

$a[i].waittime = count + a[i].arrivt$

ii) The waiting time for the rest of the instances of the process is calculated as:

a) If the time quantum is greater than the remaining burst time then waiting time is calculated as:

$a[i].waittime = count + tq$

b) Else if the time quantum is greater than the remaining burst time then waiting time is calculated as:

$a[i].waittime = count - remaining\ burst\ time$

Step 4: Calculate the average waiting time and average turnaround time

Step 5: Print the results of the step 4.

### **PROGRAM**

```
/*
```

This program has assumed that the inputs are given in ordered fashion according to their arrival times.

```
*/
```

```
struct roundRobin
```

```
{
```

```
int pburst,pburst1,wtime,endtime,arrivt,boolean,flagcntl;  
char pname[5];  
}a[5];
```

```
int n,tq;  
void input();  
void initialize();  
void calculate();  
void display_waittime();
```

```
int main()  
{  
    input();  
    initialize();  
    calculate();  
    display_waittime();  
    //getch();  
    //return 0;  
}
```

```
void input()  
{ int i;  
    printf("Enter Total no. of processes\n");  
    scanf("%d",&n);  
    for(i=0;i<n;i++)  
    {  
        printf("Enter process name:");  
        scanf("%s",&a[i].pname);  
        printf("Enter process burst time:");  
        scanf("%d",&a[i].pburst);  
        printf("Enter process arrival time:");  
        scanf("%d",&a[i].arrivt);  
    }  
    printf("\nEnter the time quantum/Time Slice:");  
    scanf("%d",&tq);  
}
```

```
void initialize()  
{ int i;  
  
    for(i=0;i<n;i++)  
    {  
        a[i].pburst1=a[i].pburst;  
        a[i].wtime=0;  
        a[i].endtime=0;  
        a[i].boolean=0;
```

```
a[i].flagcntl=0;
}
}

void calculate()
{ int i,j=0,k=0,flag=1,count=0;
printf("\n---GANTT CHART---\n");
printf("0 | ");
while(flag)
{
for(i=0;i<n;i++)
{
if((k<n)&&(a[i].arrivt<=count)&&(a[i].flagcntl==0)) //calculating waiting time for first
time
{
a[i].wtime=count-a[i].arrivt;
a[i].endtime=count;
a[i].boolean=1;
a[i].flagcntl=1;
k++;
}
if((a[i].pburst1>tq)&&(a[i].arrivt<=count))
{
if(a[i].boolean==1)
a[i].boolean=0;
else
a[i].wtime=a[i].wtime+(count-a[i].endtime);
count=count+tq;
a[i].pburst1=a[i].pburst1-tq;
a[i].endtime=count;
printf("%d %s| ",count,a[i].pname);
}
else if((a[i].pburst1>0) && (a[i].pburst1<=tq) && (a[i].arrivt<=count))
{
if(a[i].boolean==1)
a[i].boolean=0;
else
a[i].wtime=a[i].wtime+(count-a[i].endtime);
count=count+a[i].pburst1;
a[i].endtime=count;
printf("%d %s| ",count,a[i].pname);
a[i].pburst1=0;
j++;
}
else if(j==n) flag=0;
}
}
}
```

```
//end of while loop
}

void display_waittime()
{ int i,tot=0,turn=0;

for(i=0;i<n;i++)
{
printf("\n\nWaiting time for Process %s is %d",a[i].pname,a[i].wtime);
tot=tot+a[i].wtime;
turn=turn+a[i].endtime-a[i].arrivt;
}
printf("\n\n\tAverage waiting time=%f", (float)tot/(float)n);
printf("\n\n\tAverage turnaround time=%f\n", (float)turn/(float)n);
}
```

### **OUTPUT:**

[root@lab1cab7 orbit]# gcc roundrobin.c

[root@lab1cab7 orbit]# ./a.out

Enter Total no. of processes

3

Enter process name:A

Enter process burst time:4

Enter process arrival time:0

Enter process name:B

Enter process burst time:3

Enter process arrival time:3

Enter process name:C

Enter process burst time:2

Enter process arrival time:3

Enter the time quantum/Time Slice:2

---GANTT CHART---

0 | 2 A| 4 A| 6 B| 8 C| 9 B|

Waiting time for Process A is 0

Waiting time for Process B is 3

Waiting time for Process C is 3

Average waiting time=2.000000

Average turnaround time=5.000000

[root@lab1cab7 orbit]#

**RESULT:**

Thus the C program to simulate CPU scheduling algorithm for round robin was executed successfully.



**ExNo:7b**

## **PRIORITY SCHEDULING**

**Date:**

**AIM:**

To write a C program to implement CPU scheduling algorithm for priority scheduling.

### **PROBLEM DESCRIPTION:**

Cpu scheduler will decide which process should be given the CPU for its execution. For this it uses different algorithms to choose among the processes. One among those algorithms is the FCFS algorithm.

In this algorithm, the process which arrives first is given the CPU after finishing its request. Only then it will allow the CPU to execute other processes.

### **ALGORITHM:**

- Step1: Get the number of processes, burst time, and priority.
- Step2: Using for loop  $i=0$  to  $n-1$  do step 1 to 6.
- Step3: If  $i=0$ , wait time  $=0$ ,  $T[0]=b[0]$ ;
- Step4:  $T[i]=T[i-1]+b[i]$  and  $wt[i]=T[i]-b[i]$ .
- Step5: Total waiting time is calculated by adding the waiting time for each process.
- Step6: Total turn around time is calculated by adding all total times of each process.
- Step7: Calculate Average waiting time by dividing the total waiting time by the total number of processes.
- Step8: Calculate Average turn around time by dividing the total time by the number of processes.
- Step9: Display the result.

### **PROGRAM:**

```
void main()
{
    int i,j,n,t,turn[20],burst[20],p[20],wt[20],c[20];
    float await,aturn,twait=0,tturn=0;
    printf("\nEnter the value of n:");
    scanf("%d",&n);
    printf("\nEnter the process no burst and arrival time");
    for(i=0;i<n;i++)
    {
        scanf("%d",&c[i]);
        scanf("%d",&burst[i]);
    }
}
```

```
        scanf("%d",&p[i]);
    }
    for(i=0;i<n;i++)
        for(j=i+1;j<n;j++)
        {
            if(p[i]>p[j])
            {
                t=p[i];
                p[i]=p[j];
                p[j]=t;
                t=burst[i];
                burst[i]=burst[j];
                burst[j]=t;
                t=c[i];
                c[i]=c[j];
                c[j]=t;
            }
        }
    for(i=0;i<n;i++)
    {
        if(i==0)
        {
            wt[i]=0;
            turn[i]=burst[i];
        }
        else
        {
            turn[i]=turn[i-1]+burst[i];
            wt[i]=turn[i]-burst[i];
            twait=twait+wt[i];
            tturn=tturn+turn[i];
        }
    }
    await=twait/n;
    aturn=tturn/n;
    printf("pno\tbtime\tatime\twtime\tttime");
    for(i=0;i<n;i++)
    {
        printf("\n%d\t%d\t%d\t%d\t%d\n",c[i],burst[i],p[i],wt[i],turn[i]);
    }
    printf("\n The average waiting time is:%f",await);
    printf("\n The average turn around time is:%f",aturn);
}
```



### **OUTPUT:**

\$\$ cc ps.c

\$a.out

Enter the process burst no and priority

1      15      2

2      5      1

3      10      3

pno	btime	priority	wtime	ttime
2	5	1	0	5
1	15	2	5	20
3	10	3	20	30

The average waiting time is :8.333333

The average turn around time is:18.333334

### **RESULT:**

Thus C program to implement CPU scheduling algorithm for priority scheduling was executed successfully.

## **ExNo:8**      **PRODUCER CONSUMER PROBLEM USING SEMAPHORE**

**Date :**

**AIM:**

To write a C program to implement the Producer & consumer Problem (Semaphore)

### **ALGORITHM:**

Step 1: The Semaphore mutex, full & empty are initialized.

Step 2: In the case of producer process

- i) Produce an item in to temporary variable.
- ii) If there is empty space in the buffer check the mutex value for enter into the critical section.
- iii) If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

Step 3: In the case of consumer process

- i) It should wait if the buffer is empty
- ii) If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer
- iii) Signal the mutex value and reduce the empty value by 1.
- iv) Consume the item.

Step 4: Print the result

### **PROGRAM :**

```
#define BUFFERSIZE 10
int mutex,n,empty,full=0,item,item1;
int buffer[20];
int in=0,out=0,mutex=1;
void wait(int s)
{
    while(s<0)
    {
        printf("\nCannot add an item\n");
        exit(0);
    }
    s--;
}
void signal(int s)
{
    s++;
}
void producer()
{
    do
    {
        wait (empty);
        wait(mutex);
        printf("\nEnter an item:");
```

```
        scanf("%d",&item);
        buffer[in]=item;
        in=in+1;
        signal(mutex);
        signal(full);
    }
    while(in<n);
}
void consumer()
{
    do
    {
        wait(full);
        wait(mutex);
        item1=buffer[out];
        printf("\nConsumed item =%d",item1);
        out=out+1;
        signal(mutex);
        signal(empty);
    }
    while(out<n);
}
void main()
{
    printf("Enter the value of n:");
    scanf("%d",&n);
    empty=n;
    while(in<n)
    producer();
    while(in!=out)
    consumer();
}
```

### **OUTPUT:**

```
$ cc prco.c
$ a.out
Enter the value of n :3
Enter the item:2
Enter the item:5
Enter the item:9
consumed item=2
consumed item=5
consumed item=9
$
```

### **RESULT:**

Thus the program for solving producer and consumer problem using semaphore was executed successfully.

**Ex.No:9**

## **MEMORY MANAGEMENT SCHEME- PAGING**

**Date:**

### **AIM:**

To write a C program to implement memory management using paging technique.

### **ALGORITHM:**

- Step1 : Start the program.
- Step2 : Read the base address, page size, number of pages and memory unit.
- Step3 : If the memory limit is less than the base address display the memory limit is less than limit.
- Step4 : Create the page table with the number of pages and page address.
- Step5 : Read the page number and displacement value.
- Step6 : If the page number and displacement value is valid, add the displacement value with the address corresponding to the page number and display the result.
- Step7 : Display the page is not found or displacement should be less than page size.
- Step8 : Stop the program.

### **PROGRAM:**

```
#include<stdio.h>
#include<unistd.h>
void main()
{
    int b[20],n,i,pa,p,a,d;
    printf("\nProgram for paging");
    scanf("%d",&n);
    printf("\nEnter the base address:");
    for(i=0;i<n;i++)
    {
        scanf("%d",&b[i]);
    }
    printf("\nEnter the logical address:");
    scanf("%d",&p);
    for(i=0;i<n;i++)
    {
        if(i==p)
        {
            pa=b[i]+d;
            a=b[i];
            printf("\n\tPageNo.\t BaseAdd. PhysicalAdd. \n\t %d \t %d \t %d \t",p,a,pa);
        }
    }
}
```

```
    }  
    printf("\nInvalid page");  
}
```

### **Sample Input 1:**

Program for paging  
Enter the number of pages:2  
Enter the base address:  
100  
150  
Enter the Logical address:50  
Enter the page number:1

### **Sample Output 1:**

PageNo.	BaseAdd.	PhysicalAdd.
1	150	200

### **Sample Input 2:**

Program for paging  
Enter the number of pages:1  
Enter the base address:  
100  
Enter the Logical address:2  
Enter the page number:2

### **Sample Output 2:**

Invalid page.

### **RESULT:**

Thus the program for implementing the paging concept was executed and the output was verified successfully.

## **Ex.No:10      MEMORY MANAGEMENT SCHEME-SEGMENTATION**

**Date:**

### **AIM:**

To write a C program to implement memory management using segmentation

### **ALGORITHM:**

- Step1 : Start the program.
- Step2 : Read the base address, number of segments, size of each segment, memory limit.
- Step3 : If memory address is less than the base address display “invalid memory limit”.
- Step4 : Create the segment table with the segment number and segment address and display it.
- Step5 : Read the segment number and displacement.
- Step6 : If the segment number and displacement is valid compute the real address and display the same.
- Step7 : Stop the program.

### **PROGRAM:**

```
#include<stdio.h>
#include<unistd.h>
void main()
{
    int b[20],l[20],n,i,pa,s,a,d;
    printf("\nProgram for segmentation");
    printf("\nEnter the number of segments:");
    scanf("%d",&n);
    printf("\nEnter the base address and limit register:");
    for(i=0;i<n;i++)
    {
        scanf("%d",&b[i]);
        scanf("%d",&l[i])
    }
    printf("\nEnter the logical address:");
    scanf("%d",&d);
    for(i=0;i<n;i++)
    {
        if(i==s)
        {
            if(d<l[i])
            {
                pa=b[i]+d;
                a=b[i];
            }
        }
    }
}
```

```
        printf("(\\n\\tPageNo.\\t BaseAdd. PhysicalAdd. \\n\\t %d \\t %d \\t %d \\t",s,a,pa);
        exit(0);

    }
    else
    {
        printf("\\nPage size exceeds");
        exit(0);
    }
}
printf("\\nInvalid segment");
}
```

### **Sample Input 1:**

Program for segmentation

Enter the number of segments:3

Enter the base address and limit register:

100 50

150 20

130 34

Enter the Logical address:25

Enter the segment number:1

### **Sample Output 1:**

PageNo. BaseAdd. PhysicalAdd.

2        130        155

### **Sample Input 2:**

Program for segmentation

Enter the number of segments:2

Enter the Logical address and limit register:

100 50

150 20

Enter the logical address:25

Enter the segment number:1

### **Sample Output 2:**

page size exceeds

### **RESULT:**

Thus the program for implementing the segmentation concept was executed and the output was verified successfully.