EXPERIMENT 1

BASIC LINUX COMMANDS

AIM

To familiarise basic Linux commands.

BASIC LINUX COMMANDS

1. pwd :- To know the working directory pwd command can be used. It gives us the absolute path.
2. ls :- This command can be used to know what files are in the directory you are in.
    - Open last edited file by using ls-t : This command sorts the file by modification time, showing the last edited file first;head-1 picks up this first file. To open the last edited file in the current directory use the combination of ls and head.
    - ls-1 command can be used to display one file per time.
    - Display all information about files using ls-l.
    - ls-lh command can be used to display file size in human readable format.
    - ls-a command can be used to display the hidden files in a directory.
    - Display directory information using ls-lol.
3. cd :- cd command in linux known as change directory command. It is used to change current working directory.
   Different functionalities of cd command are:
    - cd 1 : To change directory to the root directory.
    - cd dir-1/dir-2/dir-3 : used to move inside a directory from a directory.
    - cd ~ : To change directory to home directory
    - cd.. : Used to move the parent directory to current directory.
4. mk dir :- This command allows the user to create directories. This can create multiple directories as well as the permissions for the directories.
   Syntax: mk dir[options..][directories..]
    - --version : It displays the version number. Some information regarding the licence.
    - --help : It displays the help related information and exits.
    - -v : It displays a message for every directory created.
      Syntax: mk dir –v [directories]
    - -p : A flag which enables the command to create parent directories as necessary.
      Syntax : mk dir –p[directories]

5. trmdir: This command is used toremove empty directories from the file system. It removes each and every specified in the command line only if these directories are empty options.

   - -help : Print the general syntax of the command along with various options that can be used.
   - rmdir-p : Each of the directory argumented is treated as path name of which all components will be removed.
   - rmdir-v1-verbase : Displays the verbose information for every directory being processed.
   - rmdir-vesion : Displays the version information and exit.

5. rm:- rm stands for remove. rm commands is used to remove objects such as files, directories, symbolic links

   Syntax: rm[option]..FILE

6. man:- Used to display the user manual of any command that we can run on the terminal. It provides a detailed view of the command.

   Syntax: $man[OPTION..]..[COMMAND NAME]

Options and examples

- No option: Displays the whole manual of the commands.
- Selection_num: Displays only the section number of the manual.
- -f option: Gives the section in which the given command is present.
- -a option: Displays all unavailable intro manual pages.
- -k option: Searches the given command as a regular expression in all manuals and returns the manual pages with section number.
- -w option: Returns the location in which manual page of a given command is present.

7. cp- cp stands for copy. The command is used to copy files. It creates an exact image of a file on a disk with different file name.

   Syntax: cp[option]source destination
   cp[option]source directory
   cp[option]source-1 source-2 source-n directory

Arguments passed can be

- Two file names: Copies the content of first file to another file. If the file 'n' exists it creates one and content is copied to it otherwise it is overwritten.
- One or more arguments: If the command has one or more arguments specifying file names and following those arguments, an argument specifying directory name this command copies each source file to destination directory with some name.
- Two directory names: This command copies al files of source directory to destination creating all files/directories needed.

8. mv- a mv command is used to move files and also to rename files.

   Syntax to move files: mv<filename><directory path>

Syntax to rename files: mv<oldfilename><newfilename>

9. cat- cat command is used to create a file, display content of the file and to copy the content of one file to another file.

   Syntax: cat[option]…[file]

   To create a file: cat<filename>

   To display: cat<filename>

10. echo- echo command is used to display line of text/string that are passes as an argument

    Syntax: echo[option][string]

11. chmod- chmod command is used to change the read, write and execute permission of files and directories.

    Syntax: chmod[reference][operation][modes]files

12. ping- Used to check connectivity between 2 nodes i.e.whether the server is connected. Full form of ping is packet internet groper

    Syntax: ping<destination>

13. grep- The grep filter searches a file for a particular pattern of characters and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for global search for regular expression and print out).

    Syntax: grep[options]pattern[files]

    Options Description

    - -c: Prints only a count of the line that match a pattern.
    - -h: Display the matched lines, but do not display the file names.
    - -i: Ignores case for matching.
    - -l: Display list of filenames only.
    - -n: Display the match lines and their line numbers.
    - -v: This prints out all the lines that do not match the pattern.
    - -e exp: specifies expression with this option can use multiple times.
    - -f file: Take pattern from file one per line.
    - -E: Treats the pattern as an extended regular expression.
    - -W:Match whole word.
    - -O: prints only the matched parts of a matching line with each part on a separate output line.
    - -An: prints searched line and n lines after the result.
    - -Bn: prints searched line and n lines before the result.
    - -Cn: prints searched lines and n lines after & before the result.

14. cut – cut command is used for cutting out the sections from each line of files and writing the result to standard output. It can be used to cut parts of a line by byte position, characters and field. It is necessary to specify option with command otherwise it gives errors.

- –b(byte):- To extract the specific bytes ,you need to follow =b option with the list of byte numbers separated by comma .Range of bytes can also be specified using hyphen.

  List without ranges: cut –b 1,2,3 state.txt
  List with ranges: cut –b 1,-3,5-7 state .txt
- -c(column):- To cut by character ,use the –c option. This selects the character given to the –c option. This can be a list of numbers separated by comma or a range of numbers separated by hyphen.

  Syntax: cut-c[(k)-(n)/(k);(n)/(n)].filename
  K denotes starting position of character n denotes ending position of character in each line if there are separated by '_'.
- -f (field):- -c option is used for fixed length lines .To extract the useful information you need to cut by fields rather than columns. List of field numbers specified must be seperated by comma. Ranges are not described here.

  Syntax: -d"delimeter" –f(field number)file.txt
- - complement:- As the name suggests it complement the output .This option can be used in the combination with other option either –f or with –c.
- -output-delimiter :- By default the output delimiter is same as input delimiter that we specify as input delimiter that we specify in the cut with –d option. To change the output delimiter we use option output –delimiter="delimiter".
- --version:- This option is used to display the version of cut which is currently running on your system.

RESULT

Basic Linux Command is familiarized successfully.

EXPERIMENT 2

BASIC SYSTEM CALLS IN LINUX

PID AND PPID

an executable code stored on a disk is called a program and a program loaded into memory and running is called a process. when a process starts it is given a unique number called process ID or PID that identifies that process to the system

In addition to a unique process ID each process is assigned a process process ID or PPID. It to tells which process started the PPID is the PID of the processes parent

FORK ( )

A new process is created by the fork() system call. A new process may be created with fork() without a new program being run – the new sub – process simply continue to execute exactly the same program that the first (parent) process was running. It is one of the most widely used system calls under process managament

EXEC ( )

A new program will start executing after a call to exec(). Running a new program does not require that a new process be created first. Any process may be created at anytime. The currently running program is immediatley terminated.

GetPid( )

getPid stands for get the process ID. The function shall return the process ID of the calling process. The getpid() function shall always be successfull and no return values is reserved to indicate error.

EXIT ( )

The exit() system call is used by a program to terminate it's execution. The operating system reclaims resources that were used by the process after the exit() system call

WAIT ( )

A call wait() blocks the calling process until one of its child process exits or a signal is recieved. After child process terminates parents continues its execution after wait system calls instruction

OPEN ( )

Its the system call to open a file. This system call just opens the file and inorder to execute read and write operation we need to call other system calls.

CLOSE ( )

This system calls closes an opened file

STAT ( )

To use the stat system call we need to include. Sys/stat.h header file. Stat() is used to get the status of a file. Its returns file attributes about an inode.

OPENDIR ( )

To use this system call one must include <dirent.h> header file. This function shall open a directory stream corresponding to the directory named by the dirname argument. Upon completion it returns a pointer to an object of type DIR. Its used with readdir() and closedir() to get the list of files names contained in the directory specified by the dirname

READDIR ( )

ReadDir() returns a pointer to a dirent structure describing the next entry in the stream associated with dir. A call to readdir() updates the st_atime (access time) field for the directory. If the successfull, readdir() returns a pointer to a dirent structure decribing the next directory entry in the directory stream. When it reaches the end of the datastream it returns NULL.

READ( )

This system call opens the file in writing mode. We cannot edit the files with this system calls. Multiple proces can execute the read() system call on the same file simultaneously

WRITE ( )

This system call opens the file in writing mode. We cannot edit the files with this system calls. Multiple processes can not execute the write() system call on the same file simultaneously

RESULT

Basic system calls in Linux successfully familiarised.

PROGRAM

```
#include<stdio.h>

#include<unistd.h>

int main()

{

    int arr[100],n,id;

    int sumEven=0,sumOdd=0;

    printf("Enter the limit: ");

    scanf("%d",&n);

    int check=fork();

    if(check==0)

    {

        id=getppid();

        for(int i=0;i<n;i++)

        {

            if(i%2==0)

            {

              sumEven+=i;

            }

        }

    }

    printf("The sum of even numbers: %d/n ",sumEven);

    printf("The child id: %d\n\n",id);


    if(check!=0)
```

EXPERIMENT 2(a)

<u>CREATING PROCESS USING FORK SYSTEM CALL</u>

AIM

Write a program to create two process , one process(parent process)must add number upto a limit n at the same time child process must add even numbers also print the process id

ALGORITHM

Step 1: Start.

Step 2: Declare the variables pid, pid1, pid2.

Step 3: Call fork() system call to create process.

Step 4: If pid==-1, exit.

Step 5: Ifpid!=-1 , get the process id using getpid().

Step 6: Print the process id.

Step 7: Stop

```
{
        int id = getpid();
        for(int i=0;i<n;i++)
        {
            if(i%2!=0)
            {
              sumOdd+=i;
            }
        }
         printf("The sum of odd numbers: %d/n ",sumOdd);
    printf("The parent id: %d\n\n",id);
    }
    return 0;
}
```

OUTPUT

```
The sum of odd number: 9

The parent id : 9138

The sum of Even number: 12

The child id : 9138
```

RESULT

Successfully printed odd and even numbers upto a limit n and also the sum of odd and even numbers individually along with corresponding process id

EXPERIMENT 3

INTRODUCTION TO SHELL PROGRAMMING

AIM

To familiarise shell programming and able to implement programs

VI EDITOR

vi editor is create and edit text, files documents and programs.
There are two modes: i) command mode ii) insert mode

COMMANDS IN Vi

$vi<filename> : This command is used to start the vi editor

ESC i command : Used to insert the text before the current cursor position

ESC I command : Used to insert at the beginning of the current line

ESC o command : Used to insert a blank line below the current line

ESC O command: Used to insert a Blank Line above and allow insertion of contents

ESC h command : Used to move to the left of the text

ESC l command: Used to move to the right of the cursor

ESC j command : Used to move down a single line.

ESC k command : Used to move up a single line

ESC 0 command : Bring the cursor to the beginning of the same current line

ESC $: Bring the cursor to the end of the current line

ESC x command : Delete a character to the right of current cursor

ESC X command: Delete a character to the left of the current cursor

ESC dd command: To delete the current line

ESC w command : To save given text present in the file

ESC $: Bring the cursor to the end of the current line

ESC x command : Delete a character to the right of current cursor

ESC X command: Delete a character to the left of the current cursor

ESC dd command: To delete the current line

ESC w command : To save given text present in the file

ESC q! Command : To the quit the given text without saving

ESC wq command: Quits the vi editor after saving in the mentioned file

RESULT

Successfully familiarised shell programming and basic commands used .

PROGRAM -1

```
# Program to find factorial of a number

echo -n "Enter number: "
read num

fact=1

while [ $num -ne 0 ]
do
        fact=$(( $fact * $num ))
        num=$(( $num - 1 ))
done

echo "Factorial is $fact"
```

OUTPUT

```
Enter the number: 5
Factorial is 120
```

PROGRAM -2

```
# Program to find the largest of N numbers


echo -n "Enter N numbers: "
read nums


largest=0


for num in $nums
do
        if [ $num -gt $largest ]
        then
                largest=$num
        fi
done


echo "Largest number is $largest"
```

EXPERIMENT 4

SHELL PROGRAMS

AIM

To write a programs in shell programming language and gain the necessary output

DESCRIPTION:

The activities of a shell are not restricted to command interpretation alone. The shell also has Rudimentary programming features. When a group of commands has to be executed regularly, they are stored in a file (with extension .sh). All such files are called shell scripts or shell programs. Shell programs run in interpretive mode. The original UNIX came with the Bourne shell (sh) and it is universal even today. Then came a plethora of shells offering new features. Two of them, C shell (csh) and Korn shell (ksh) has been well accepted by the UNIX fraternity. Linux offers Bash shell (bash) as a superior alternative to Bourne shell.

Preliminaries

1. Comments in shell script start with #. It can be placed anywhere in a line; the shell ignores contents to its right. Comments are recommended but not mandatory.

2. Shell variables are loosely typed i.e. not declared. Their type depends on the value assigned. Variables when used in an expression or output must be prefixed by $.

3. The read statement is shell's internal tool for making scripts interactive.

4. Output is displayed using echo statement. Any text should be within quotes. Escape sequence should be used with –e option.

5. Commands are always enclosed with ` ` (back quotes).

6. Expressions are computed using the expr command. Arithmetic operators are + - * / %. Meta characters * ( ) should be escaped with a \.

7. Multiple statements can be written in a single line separated by ;

8. The shell scripts are executed using the sh command (sh filename).

OUTPUT

```
Enter N numbers: 1 4 6 7
Largest number is 7
```

PROGRAM -3

```
# Program to check whether a number is odd or even

echo -n "Enter a number: "
read num

remainder=`expr $num % 2`

if [ $remainder -eq 0 ]
then
        echo "$num is even"
else
        echo "$num is odd"
fi
```

OUTPUT

```
Enter a number: 5
5 is odd
```

PROGRAM -4

```
# Program to find sum of three numbers

echo -n "First Number is: "
read num1

echo -n "Second Number is: "
read num2

echo -n "Third Number is: "
read num3

sum=`expr $num1 + $num2 + $num3`

echo "Sum is $sum"
```

OUTPUT

```
First Number is: 3
Second Number is: 5
Third Number is: 2
Sum is 10
```

ALGORITHM - 1

```
Step 1: Start
STEP 2: Read the value of n.
STEP 3: Calculate i = n-1.
STEP 4: If the value of i is greater than 1 then calculate n
= n \ i and i = i – 1
STEP 5: Print the factorial of the given number.
STEP 6: Stop
```

ALGORITHM -2

```
Step 1: Start
Step 2: Read the elements
step 3: initialise a variable largest  = 0
step 4: for each element check if it is less than largest
step 5: if step 4 is true set largest to the new element
and continue for all elements.
step 6: print the largest
step 7: stop
```

ALGORITHM - 3

```
Step 1: Start
Step 2: Read the number to check
step 3: find the remainder of the number with respect to 2 by
mod function and set it to remainder
step 4: if the remainder is 0 print even else odd
step 5: stop
```

ALGORITHM - 4

```
Step 1: Start
Step 2: Read the values of num1, num2 and num3
step 3: add the values of num1, num2, num3 and set it to sum.
Step 4: print the sum value
step 5: stop
```

ALGORITHM - 5

```
Step 1: Start
Step 2: Read the values of n1 and n2
Step 3: Interchange the values of a and b using another
variable t as follows: t = n1 n1 = n2 n2
```

PROGRAM -5

```
# Program to swap two variables

echo -n "Enter first number: "
read n1

echo -n "Enter second number: "
read n2

tmp=$n1
n1=$n2
n2=$tmp = t
Step 4: Print a and b Step 5: Stop
```

OUTPUT

```
Enter first number: 3
Enter second number: 4
First is now 4
Second is now 3
```

RESULT

Succesfully familiarised with shell programs and implemented basic programs and gained necessary outputs

EXPERIMENT 5

CPU SCHEDULING

CPU scheduling is a crucial aspect of operating systems that manages the allocation of the CPU's processing time among various tasks or processes. Here are some key points to note about CPU scheduling:

1. Purpose: The primary goal of CPU scheduling is to maximize CPU utilization and throughput while ensuring fair and efficient execution of processes.

2. Scheduling algorithms: Different scheduling algorithms exist, each with its own advantages and trade-offs. Some popular algorithms include First-Come, First-Served (FCFS), Shortest Job Next (SJN), Round Robin (RR), Priority Scheduling, and Multilevel Queue Scheduling.

3. Preemptive vs. Non-preemptive: Scheduling algorithms can be preemptive or non-preemptive. Preemptive scheduling allows a running process to be interrupted and moved out of the CPU if a higher-priority process arrives. Non-preemptive scheduling, on the other hand, allows a process to run until it voluntarily releases the CPU.

4. Scheduling criteria: Different scheduling algorithms use various criteria to make scheduling decisions. These criteria can include factors like process priority, burst time, deadline, response time, and fairness.

5. Multilevel queue scheduling: This approach involves dividing processes into different priority levels or queues. Each queue may have its own scheduling algorithm, and processes can move between queues based on predefined criteria, such as aging or priority changes.

6. Scheduling in multiprocessor systems: CPU scheduling becomes more complex in multiprocessor or multicore systems. Load balancing, affinity scheduling, and interprocessor communication are some of the additional considerations in such environments.

7. Real-time scheduling: Real-time systems have specific timing requirements. They often employ specialized scheduling algorithms to ensure that critical tasks meet their deadlines, such as Rate-Monotonic Scheduling (RMS) and Earliest Deadline First (EDF).

Overall, CPU scheduling plays a vital role in optimizing resource allocation and ensuring efficient utilization of the CPU's processing power in an operating system. The selection of an appropriate scheduling algorithm depends on the specific system requirements and characteristics of the workload.

PROGRAM

```
#include<stdio.h>

void findWaitingTime(int processes[],int n,int burstTime[],int 
waitingTime[],int arrivalTime[])
{
    int serviceTime[n];
    serviceTime[0]=0;
    waitingTime[0]=0;

    for (int i = 1; i < n; i++)
    {
        serviceTime[i]=serviceTime[i-1]+burstTime[i-1];
        waitingTime[i]=serviceTime[i]-arrivalTime[i];

        if (waitingTime[i]<0)
        {
            waitingTime[i]=0;
        }
    }
}

void findTurnAroundTime(int processes[],int n,int burstTime[],int 
waitingTime[],int turnAroundTime[])
{
    for(int i=0;i<n;i++)
        turnAroundTime[i]=burstTime[i]+waitingTime[i];
}


void findAvgTime(int processes[],int n,int burstTime[],int 
arrivalTime[])
{
    int waitingTime[n],turnAroundTime[n];


findWaitingTime(processes,n,burstTime,waitingTime,arrivalTime);

findTurnAroundTime(processes,n,burstTime,waitingTime,turnAroundTim
e);

    printf("\
n-----------------------------------------------------------------
------------------------");
    printf("\nProcesses  Burst Time    Arrival Time   Waiting Time   
Turn-Around Time  Completion Time");
    printf("\
n-----------------------------------------------------------------
------------------------");
    int total_waitingTime=0,total_turnAroundTime=0;
```

EXPERIMENT 5(A)

FIRST COME FIRST SERVE SCHEDULING

AIM

To write a program to calculate the average waiting time and turnaround time of a certain number of processes using FCFS scheduling.

ALGORITHM

Step 1: Initialize the ready queue to an empty state.

Step 2: Load the processes into the ready queue in the order they arrive. (i.e., based on the arrival time

Step 3: Set the current process as the next process in the ready queue.

Step 4: Execute the current process until it completes.

Step 5: Calculate the waiting time for the current process as the sum of the waiting times of all previous processes.

Step 6: Calculate the turnaround time for the current process as the sum of the burst time and waiting time.

Step 7: Calculate the average waiting time by summing up the waiting times of all processes and dividing by the total number of processes (n).

Step 8: Calculate the average turnaround time by summing up the turnaround times of all processes and dividing by the total number of processes (n).

```
for (int i = 0; i < n; i++)
    {
        total_waitingTime=total_waitingTime+waitingTime[i];

total_turnAroundTime=total_turnAroundTime+turnAroundTime[i];
        int compl_time=turnAroundTime[i]+arrivalTime[i];

        printf("\n%6d  %8d  %13d  %14d  %15d
%15d",i,burstTime[i],arrivalTime[i],waitingTime[i],turnAroundTime[
i],compl_time);
    }

    printf("\nAverage Waiting Time = %0.3f",
(float)total_waitingTime/(float)n);
    printf("\nAverage Turn Around Time = %0.3f",
(float)total_turnAroundTime/(float)n);
}

void main()
{
    int processes[20],n,burstTime[20],arrivalTime[20],i;

    printf("Enter the number of processes:");
    scanf("%d",&n);

    printf("\nEnter Burst Times(in ms) of Processes:");
    printf("\n\n");
    for (i = 0; i < n; i++)
    {
      printf("Process %d:",i+1);
        scanf("%d",&burstTime[i]);
    }

    printf("\nEnter Arrival Times(in ms) of Processes:");
    printf("\n\n");
    for (i = 0; i < n; i++)
    {
        printf("Process %d:",i+1);
        scanf("%d",&arrivalTime[i]);
    }

    findAvgTime(processes,n,burstTime,arrivalTime);
```

OUTPUT

```
Enter the number of processes : 3
Enter the burst time of processes :
Process 1: 24
Process 2: 3
Process 3: 3
Enter the arrival time of processes :
Process 1: 0
Process 2: 1
Process 3: 2
```

Processes	Burst time	Arrival time	Waiting time	Turnaround time	Completion time
0	24	0	0	24	24
1	3	1	23	26	27
2	3	2	25	28	30

Average waiting time = 16.000
Average turnaround time = 26.000

RESULT

Program executed successfully.

PROGRAM

```
#include<stdio.h>

struct process
{
    int pid; //process ID
    int at;  // Arrival Time
    int bt; //Burst Time
    int ct; //Completion Time
    int wt; //Waiting Time
    int tat; // Turn Around Time
    int flag;
};

int main()
{
    int n;

    printf("Enter total number of processes : ");
    scanf("%d",&n);
    struct process proc[n],temp;
    int i,j;

    for(i=0; i<n; i++)
    {
        proc[i].flag = 0;
    }
    printf("\nEnter Process-ID, Arrival-time and Burst-time for 
each process :\n");
    for(int j=0; j<n; j++)
    {
          printf("Process %d : ",j+1);
        scanf("%d %d %d",&proc[j].pid,&proc[j].at,&proc[j].bt);
    }

    // printf("before sorting according to arrival-time\n");
    // printf("process-id\tarrival-time\tburst-time\n");
    // for(i=0;i<n;i++)
    // {
    //     printf("%d\t\t%d\t\t%d\
n",proc[i].pid,proc[i].at,proc[i].bt);
    // }


    //sorting according to arrival time
    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)       {
```

EXPERIMENT 5(B)

SHORTEST JOB FIRST

AIM

To write a program to calculate the average waiting time and turnaround time of a certain number of processes using SJF scheduling.

ALGORITHM

Step 1: Initialize the ready queue to an empty state.

Step 2: Load all processes into the ready queue.

Step 3: Sort the processes in the ready queue based on their burst time in ascending order.

Step 4: Set the current process as the next process in the ready queue.

Step 5: Execute the current process until it completes.

Step 6: Calculate the waiting time for the current process as the sum of the waiting times of all previous processes.

Step 7: Calculate the turnaround time for the current process as the sum of the burst time and waiting time.

Step 8: Calculate the average waiting time by summing up the waiting times of all processes and dividing by the total number of processes (n).

Step 9: Calculate the average turnaround time by summing up the turnaround times of all processes and dividing by the total number of processes (n).

```
if(proc[i].at > proc[j].at)
            {
                temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }

    // printf("process-id\tarrival-time\tburst-time\n");
    // for(i=0;i<n;i++)
    // {
    //     printf("%d\t\t%d\t\t%d\
n",proc[i].pid,proc[i].at,proc[i].bt);
    // }
    // printf("after sorting according to arrival-time\n");

    int r=proc[0].at;
    int p=1;
    int k=0;
    int min=proc[0].bt;


    while(proc[p].at==r && p<=n)
    {
      if(proc[p].bt < min)
        {
            min = proc[p].bt;
            k=p;
        }
      p++;
    }

    proc[k].ct = proc[k].at+proc[k].bt;

    proc[k].flag=1;
    int l=k;
    int d=proc[k].ct;


    int counter=1;
    while(counter!=n)
    {
          int min1 = 1000;
          for(i=0;i<n;i++)
          {
               if(proc[i].flag==0 && proc[i].at<=d)
               {
                    if(proc[i].bt < min1)
```

```
min1 = proc[i].bt;
                         k=i;

                    }
               }
          }
          proc[k].ct = proc[l].ct+min1;
          proc[k].flag = 1;
          d = proc[k].ct;
          counter++;
          l=k;
    }
    int tat1=0,wt1=0; // Total waiting time and Total Turn Around
Time
    for(i=0;i<n;i++)
    {
        proc[i].tat=proc[i].ct-proc[i].at;
        tat1+=proc[i].tat;
    }

    for(i=0;i<n;i++)
    {
     proc[i].wt=proc[i].tat-proc[i].bt; // Waiting Time = Turn
Around Time - Burst Time
     wt1+=proc[i].wt;
    }

    //printf("process-id\tarrival-time\tburst-time\tcompletion-
time\twaiting-time\tturn-around-time\n");

     printf("\
n------------------------------------------------------------------
------------------------");
    printf("\nProcesses  Arrival Time    Burst Time   Completion
Time  Waiting Time  Turn-Around Time");
    printf("\
n------------------------------------------------------------------
------------------------");
    for(i=0;i<n;i++)
    {
     printf("\n%6d  %8d  %13d  %14d  %15d
%15d",proc[i].pid,proc[i].at,proc[i].bt,proc[i].ct,proc[i].wt,proc
[i].tat);
    }



    float tat2,wt2;
    tat2=(float)tat1/(float)n; //Average Turn Around Time
    wt2=(float)wt1/(float)n; //Average Waiting Time
    printf("\nAverage Turn-Around-Time = %.3f\n",tat2);
    printf("\nAverage Waiting-Time = %.3f\n",wt2);

    return 0;}
```

OUTPUT

```
Enter the number of processes : 3
Enter the process_id/arrival time/burst time of each processes :
Process 1: 0  0  3
Process 2: 1  1  3
Process 3: 2  2  24
------------------------------------------------------------------------------------------------
Processes      Burst time      Arrival time      Waiting time      Turnaround time      Completion time
------------------------------------------------------------------------------------------------
   0               3               0                 0                  3                    3
   1               3               1                 2                  5                    6
   2              24               2                 4                 28                   30

Average waiting time = 2.000
Average turnaround time = 12.000
```

RESULT

Program executed successfully.

PROGRAM

```
#include <stdio.h>
int i, j, n;
float tatAvg, wtAvg;
struct Process {
    int pId;
    int bt;
    int priority;
};
void read(struct Process p[]) {
    printf("\n");
    printf("Enter the Burst Time, Priority Of Each Process :\n");
    printf("\n");
    for (i = 0; i < n; ++i)
    {    p[i].pId = i + 1;
        printf("Process %d: ", p[i].pId);
        scanf("%d %d", &p[i].bt, &p[i].priority);
    } }
void display(struct Process p[], int wt[], int tat[]) {
    printf("\
n------------------------------------------------------------------
--");
    printf("\nProcessID | BurstTime | Priority | Waiting time |
Turn Around Time\n");

printf("------------------------------------------------------------------
-----------\n");
    for (i = 0; i < n; ++i) {
        printf("%5d\t\t%d\t%4d\t\t%d\t\t%d\n", p[i].pId, p[i].bt,
p[i].priority, wt[i], tat[i]);
    }
    printf("\nAverage Waiting Time: %.3f", wtAvg);
    printf("\naverage Turnaround Time: %.3f", tatAvg);
    printf("\n");
}

void sort(struct Process p[]) {
    struct Process temp;
    for (i = 0; i < n - 1; ++i) {
        for (j = 0; j < n - 1 - i; ++j) {
            if (p[j].priority > p[j + 1].priority) {
                temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }   }
    }}
void findWaitingTime(struct Process p[], int wt[]) {
    wt[0] = 0;
    int wtSum = 0;
    for (i = 1; i < n; ++i) {
        wt[i] = wt[i - 1] + p[i - 1].bt;
        wtSum += wt[i];
```

EXPERIMENT 5(C)

PRIORITY SCHEDULING

AIM

To write a program to calculate the average waiting time and turnaround time of a certain number of processes using Priority scheduling.

ALGORITHM

Step 1: Initialize the ready queue to an empty state.

Step 2: Load all processes into the ready queue.

Step 3: Sort the processes in the ready queue based on their priority level in ascending order. (A lower priority value indicates a higher priority process.)

Step 4: Set the current process as the next process in the ready queue with the highest priority.

Step 5: Execute the current process until it completes.

Step 6: Calculate the waiting time for the current process as the sum of the waiting times of all previous processes.

Step 7: Calculate the turnaround time for the current process as the sum of the burst time and waiting time.

Step 8: Calculate the average waiting time by summing up the waiting times of all processes and dividing by the total number of processes (n).

Step 9: Calculate the average turnaround time by summing up the turnaround times of all processes and dividing by the total number of processes (n).

```
}
    wtAvg = (float)wtSum / n;
}
void findTurnAroundtime(struct Process p[], int tat[], int wt[]) {
    int tatSum = 0;
    for (i = 0; i < n; ++i) {
        tat[i] = p[i].bt + wt[i];
        tatSum += tat[i];
    }    tatAvg = (float)tatSum / n;
}
int main()
{
    struct Process p[20];           /* Array object for the
structure */
    int wt[20], tat[20];
    printf("Enter Number of Processes:");
    scanf("%d", &n);
    read(p);
    sort(p);
    findWaitingTime(p, wt);
    findTurnAroundtime(p, tat, wt);
    display(p, wt, tat);

    return 0; }
```

OUTPUT

```
Enter the number of processes : 3
Enter the burst time and priority of each processes :
Process 1: 24  1
Process 2:  3  2
Process 3:  3  3
```

Processes	Burst time	Priority	Waiting time	Turnaround time
1	24	1	0	24
2	3	2	24	27
3	3	3	27	30

```
Average waiting time = 17.000
Average turnaround time = 27.000
```

RESULT

Program executed successfully.

PROGRAM

```
#include <stdio.h>
int i, n, quantum; /* give a time quantum */

void read(int b[]) {
    printf("\nEnter Burst Time of each process\n");
    for (i = 0; i < n; ++i) {
        printf("Process %d : ", i);
        scanf("%d", &b[i]);
    }}
void findWaitingtime(int b[], int wt[]) {
    int b_rem[20];
    for (i = 0; i < n; ++i) {
        b_rem[i] = b[i];                           /* Create a copy
of the burst time array */
    }
    int time = 0;                                  /* initialize time
as 0 */
    while (1)
    {      /* Traverse */
        int flag = 0;
        for (i = 0; i < n; ++i) {
            if (b_rem[i] > 0) {                    /* continue only
if burst time is greater than 0 */
                flag = 1;                          /* there is a
pending process */
                if (b_rem[i] > quantum) {
                    time += quantum;               /* shows how much
time a process has been processed */
                    b_rem[i] -= quantum;           /* Decrease the
burst_time of current process by quantum */
                }
                else {
                    time += b_rem[i];              /* shows how much
time a process has been processed */
                    wt[i] = time - b[i];           /* Waiting time is
current time minus time used by this process  */
                    b_rem[i] = 0;                  /* fully executed,
so remaining burst time=0 */
                }   }
        }        if (flag == 0) {                           /* there
is no pending process */
            break;
        }  }
}
void findTurnAroundtime(int tat[], int b[], int wt[]) {
    for (i = 0; i < n; ++i) {
        tat[i] = b[i] + wt[i];
    }
}
void display(int b[], int wt[], int tat[]) {
    int wtSum = 0, tatSum = 0;
```

EXPERIMENT 5(D)

ROUND ROBIN SCHEDULING

AIM

To write a program to calculate the average waiting time and turnaround time of a certain number of processes using RoundRobin scheduling.

ALGORITHM

Step 1: Initialize the ready queue to an empty state. Step 2: Load all processes into the ready queue. Step 3: Set the time quantum (a fixed time slice) for process execution.

While the ready queue is not empty:

For each process in the ready queue:

Step 1: Set the current process as the next process in the ready queue.

Step 2: Execute the current process for the time quantum or until it completes, whichever occurs first.

Step 3: If the current process has completed:

- Calculate the waiting time for the current process as the sum of the waiting times of all previous processes.

- Calculate the turnaround time for the current process as the sum of the burst time and waiting time.

- Remove the current process from the ready queue.

Step 4: If the current process has not completed:

- Reduce its remaining burst time by the time quantum.

- Move the current process to the end of the ready queue.

Step 5: Calculate the average waiting time by summing up the waiting times of all processes and dividing by the total number of processes (n).

Step 6: Calculate the average turnaround time by summing up the turnaround times of all processes and dividing by the total number of processes (n).

```
    printf("\n-------------------------------------------------\
n");
    printf("Process\tBurstTime WaitingTime  TurnAroundTime\n");


    printf("-------------------------------------------------\n");
    for (i = 0; i < n; ++i) {
        wtSum += wt[i];
        tatSum += tat[i];
        printf("%3d\t%6d\t\t%d\t%5d\n", i, b[i], wt[i], tat[i]);
    }
    printf("\nAverage waiting time: %.3f", (float)wtSum / n);
    printf("\naverage turnaround time: %.3f", (float)tatSum / n);
    printf("\n");
}void calcTime(int b[], int wt[], int tat[]) {
    findWaitingtime(b, wt);
    findTurnAroundtime(tat, b, wt);
    display(b, wt, tat);
}void main()
{
    int b[20], tat[20], wt[20];
    printf("Enter Number of Processes:");
    scanf("%d", &n);
    read(b);
    printf("\nEnter time quantum:");
    scanf("%d", &quantum);
    calcTime(b, wt, tat);
}
```

OUTPUT

```
Enter the number of processes : 4
Enter the burst time and arrival time of each processes :
Process 1: 4   0
Process 2:  7  1
Process 3:  5  2
Process 4:  6  3

------------------------------------------------------------------------------
        Processes      Burst time      Arrival time     Waiting time     Turnaround time
------------------------------------------------------------------------------
            1              4               0                9                 13
            2              7               1               14                 21
            3              5               2               11                 16
            4              6               3               12                 18


Average waiting time = 11.500
Average turnaround time = 17.000
```

RESULT

Program executed successfully.