Journal of Open Aviation Science (2023), **1**, 1–22
doi:10.74800/iucsp.1.2023

ARTICLE

Introduction to UNIX Commands and Shell Programming

Sumit Yadav,* Raju Kumar Yadav,* and Prashant Bhandari*

Pulchowk Campus, Lalitpur, Nepal

*Corresponding author: 076bct088.sumit@pcampus.edu.np; 076bct100.raju@pcampus.edu.np; 076bct049.prashant@pcampus.edu.np

(Received 1 March 2023; revised 1 March 2023; accepted 10 March 2023; first published online 20 March 2023)

(Editor: Sumit Yadav, Raju Kumar Yadav, Prashant Bhandari; open reviewed by: Bikal Adhikari, Lok Nath Regmi)

Abstract

This project highlights the importance of Unix commands, Vi editor commands, and Unix shell programming commands for programmers and system administrators working on Unix-based operating systems. The project covers various Unix commands for managing files, directories, and processes, along with Vi editor commands and Unix shell programming commands like variables, loops, conditional statements, and file handling.

Additionally, the project includes 22 programming questions that cover string manipulation, arithmetic operations, process management, file handling, and memory management schemes. These questions provide a useful exercise for enhancing programming skills and reinforcing the concepts learned in the project.

In conclusion, this project provides a comprehensive understanding of Unix commands, Vi editor commands, and Unix shell programming commands, and offers various programming questions to strengthen programming abilities.

Keywords: Unix, commands, Vi editor, shell programming, programmers, system administrators, operating systems, working directory, file management

1. Introduction

UNIX is an operating system that was originally developed in the 1960s and 70s by a group of computer scientists at Bell Labs. It has since become one of the most popular operating systems in the world, known for its stability, security, and powerful command-line interface. UNIX has been used to power everything from desktop computers to servers and even supercomputers.

One of the key features of UNIX is its command-line interface, which allows users to interact with the operating system through a series of commands typed into a terminal window. This interface provides a great deal of power and flexibility, allowing users to perform complex operations quickly and efficiently. Many software developers and system administrators prefer to use UNIX because of this flexibility and the ability to automate tasks using shell scripts.

UNIX is also known for its multi-user and multi-tasking capabilities. Multiple users can log into the same system at the same time and run their own programs and tasks, without interfering with one another. The system is also able to manage multiple tasks simultaneously, allowing users to run background tasks while still using the system for other purposes.

© Nepali Harvest 2023. This is an Open Access article, distributed under the terms of the Creative Commons Attribution 4.0 International (CC BY 4.0) licence (https://creativecommons.org/licenses/by/4.0/)

Over the years, many different versions of UNIX have been developed, including Linux, which is a popular open-source variant. UNIX has also been adapted for use in many different fields, including scientific research, finance, and government. Despite its age, UNIX remains a powerful and versatile operating system that is widely used and respected in the technology industry.

UNIX commands and shell programming are essential components of the UNIX operating system. UNIX provides a vast array of commands that can be used to perform a wide range of tasks, from managing files and directories to networking and system administration.

Some of the most commonly used UNIX commands include:

ls: Lists the files and directories in the current directory cd: Changes the current working directory mkdir: Creates a new directory touch: Creates a new file or updates the timestamp on an existing file cat: Concatenates files and prints the output to the terminal cp: Copies files or directories from one location to another mv: Moves or renames files and directories rm: Removes files or directories

In addition to these basic commands, UNIX also provides more advanced commands for tasks such as file compression, process management, and network administration.

UNIX shell programming involves using the shell scripting language to automate tasks and create custom tools. The shell is a command-line interface that provides access to the operating system's services and utilities. Shell scripts are programs written in the shell language and are executed by the shell.

Some of the key features of UNIX shell programming include:

Variables: Shell scripts can define and manipulate variables to store data and control program flow. Control structures: Shell scripts support control structures such as loops and conditional statements for more complex programming logic. Functions: Shell scripts can define functions to group code and create reusable code blocks. Input/output: Shell scripts can read input from files or the terminal and output data to files or the terminal. Error handling: Shell scripts can handle errors and exceptions to ensure the program continues running even if an error occurs.

Shell scripts can be used for a wide range of tasks, from automating routine system administration tasks to creating custom tools and utilities. They are a powerful tool for improving productivity and efficiency on UNIX systems.

2. AIMS

2.1 Basic UNIX commands

Some of the basic UNIX commands that we implement in os lab.

- Display date and time: The command `date` is used to display the current date and time in the terminal. It also allows you to set the system date and time if required.
- To display calendar of years and month in terminal:
  The command `cal` is used to display the calendar of the current month in the terminal. You can also specify the year using the option `-y` followed by the year number.
- Used to print text in linux:
  The command `echo` is used to print text in the terminal. It can also be used to print the values of variables and to create files.
- Used to display the argument in # symbol:
  The command `echo #` is used to display the argument in the terminal with a # symbol in front of it.
- Display the current working directory:
  The command `pwd` is used to display the current working directory in the terminal.

- Display the terminal names:
  The command `tty` is used to display the name of the terminal device in the terminal.
- Clear the screen:
  The command `clear` is used to clear the contents of the terminal screen. It does not delete the previous commands or output, but only hides them from view.

Command	Description
`date`	Displays the current date and time
`cal`	Displays a calendar for a specific month or year
`echo`	Prints text to the terminal
`#`	Comments out a line of `text`
`pwd`	Displays the current working directory
`tty`	Displays the terminal name
`clear`	Clears the terminal screen
`man`	Displays the manual page for a specific command
`help`	Displays help information for a specific command
`tput`	Modifies terminal settings
`ls`	Lists the files and directories in the current directory
`mkdir`	Creates a new directory
cd	Changes the current directory
cat	Displays the contents of a file
mv	Moves a file or directory
rm	Removes a file or directory
cat -n <filename>	Displays the contents of a file with line numbers
sort	Sorts the contents of a file
cp	Copies a file or directory
wc	Counts the number of lines, words, and characters in a file
lp	Sends a file to a printer
pg	Displays a file one page at a time
head	Displays the first few lines of a file
tail	Displays the last few lines of a file
ls -a	Lists all files and directories, including hidden files
more <filename>	Displays the contents of a file one page at a time
more	Continues displaying the contents of a file one page at a time
grep [optional] pattern <filename>	Searches for a pattern in a file
sort <filename>	Sorts the contents of a file
$merge <user name>	Merges multiple user accounts into one
$wall <message>	Sends a message to all users
$mail <user name>	Sends an email to a user
$reply <user name>	Replies to a user's email

2.2 Vi editor in UNIX

The vi editor is a popular text editor that is widely used in Unix-based operating systems. It has several modes that allow the user to perform different functions. The three primary modes of the vi editor are:

- Command mode:
  This is the default mode of vi editor. In this mode, the user can navigate through the text, delete, copy, paste and perform other commands. It is indicated by the absence of the word "INSERT" at the

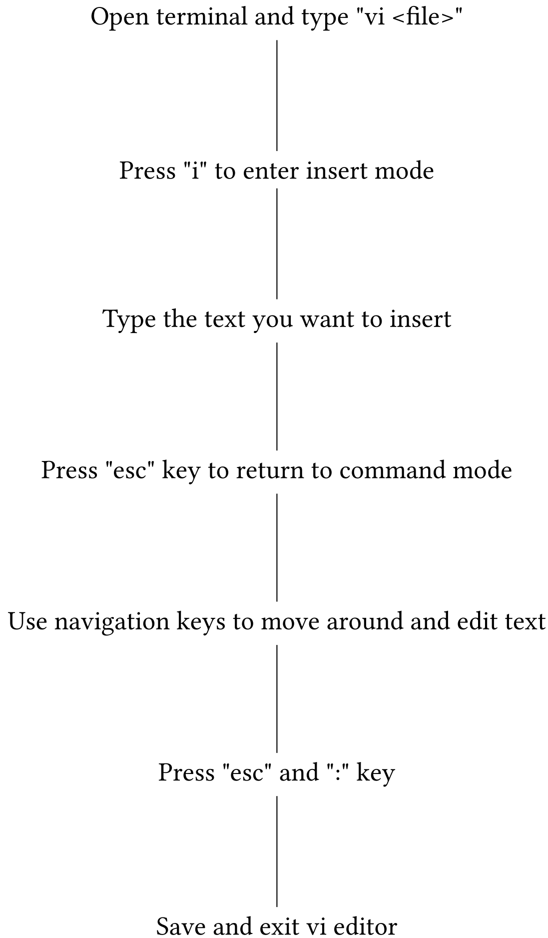
bottom left corner of the screen.

- Insert mode:
  In this mode, the user can insert text into the document. It is indicated by the word "INSERT" at the bottom left corner of the screen.
- Ex mode:
  This mode is used to execute commands that are not available in the command mode. It is entered by typing the colon (:) character while in the command mode.

Here are some of the most commonly used vi editor commands:

Command	**Description**
$	Moves the cursor to the end of the current line.
filename>	Opens the specified file in the vi editor.
vi	Starts the vi editor.
vi +n	Starts the vi editor and positions the cursor at line n.
vi -n	Starts the vi editor and positions the cursor n lines from the end of the file.
a	Enters insert mode and places the cursor after the current character.
A	Enters insert mode and places the cursor at the end of the current line.
i	Enters insert mode and places the cursor before the current character.
I	Enters insert mode and places the cursor at the beginning of the current line.
o	Enters insert mode and starts a new line below the current line.
O	Enters insert mode and starts a new line above the current line.
rx	Replaces the current character with the character x.
R	Enters overwrite mode and replaces characters as you type.
s	Deletes the current character and enters insert mode.
S	Deletes the current line and enters insert mode.
h	Moves the cursor one character to the left.
nh	Moves the cursor n characters to the left.
j	Moves the cursor one line down.
nj	Moves the cursor n lines down.
k	Moves the cursor one line up.
nk	Moves the cursor n lines up.
enter	Moves the cursor to the beginning of the next line.
+	Moves the cursor to the beginning of the next line.
^	Moves the cursor to the beginning of the current line.
b	Moves the cursor one word backward.
-	Moves the cursor to the beginning of the previous line.
0	Moves the cursor to the beginning of the current line.
$	Moves the cursor to the end of the current line.
e	Moves the cursor to the end of the current word.
w	Moves the cursor one word forward.
x	Deletes the current character.
dw	Deletes the current word.
db	Deletes the previous word.
d$	Deletes from the current cursor position to the end of the line.
q!	Quits the editor without saving changes.
q	Quits the editor.
wq	Saves changes and quits the editor.

To operate the vi editor in UNIX, you need to follow the following steps:



2.3 Shell Programming Commands

Shell programming commands are a set of commands and instructions that can be used to write scripts and programs in Unix or Linux shell environments. Here are explanations and examples of some commonly used shell programming commands:

2.3.1 Common Shells:

There are several shells available in Unix or Linux environments. The most commonly used shells are:

csh (C Shell): This shell is designed for interactive use and has C language-like syntax. It is mainly used for scripting and interactive use in scientific computing and data processing.

bsh (Bourne Shell): This is the oldest Unix shell and is widely used for scripting and system administration. It supports the basic features of shell programming, such as variables, loops, and conditional statements.

sh (Bourne-Again Shell): This is a more recent version of the Bourne shell and is the default shell for most Linux distributions. It has additional features, such as command-line editing, job control, and programmable completion. tcsh (TENEX C Shell): This is an extended version of the C shell that includes additional features such as file name completion, command history, and job control.

Example: To check which shell is currently being used, you can run the following command:

echo $SHELL

This will display the path to the current shell being used.

2.3.2 Shell Keywords:

Shell keywords are built-in commands in Unix or Linux shell environments. These commands are used to perform specific tasks such as input/output operations, conditional branching, and looping.

echo: This command is used to display messages or variables on the terminal.

read: This command is used to read input from the user or from a file.

if fi: This keyword is used to create conditional statements. The code inside the "if" block is executed only if the condition is true.

else: This keyword is used in conjunction with the "if" statement to provide an alternative code block to execute if the "if" condition is false.

case esac: This keyword is used to create a multiple choice menu. The code inside the "case" block is executed based on the user's choice.

for: This keyword is used to create loops that iterate over a set of values.

while: This keyword is used to create loops that execute until a condition becomes false.

do: This keyword is used in conjunction with the "for" and "while" keywords to define the code block that is executed during each iteration of the loop.

done: This keyword marks the end of a "for" or "while" loop.

until: This keyword is used to create loops that execute until a condition becomes true.

set: This keyword is used to set shell options or positional parameters.

unset: This keyword is used to unset or delete a shell variable or function.

readonly: This keyword is used to make a shell variable or function read-only.

shift: This keyword is used to shift positional parameters to the left.

export: This keyword is used to make a shell variable or function available to subprocesses.

break: This keyword is used to break out of a loop.

continue: This keyword is used to skip to the next iteration of a loop.

exit: This keyword is used to exit the shell or terminate a script.

return: This keyword is used to return a value from a function.

trap: This keyword is used to set up a signal handler for the shell.

wait: This keyword is used to wait for a subprocess to finish executing.

eval: This keyword is used to evaluate a string as a shell command.

exec: used to replace the current shell process with another command.

ulimit: used to set resource limits for the current shell.

umask: used to set the file mode creation mask.

2.3.3 General shell things:

Shbang line: The first line of a shell script that specifies the shell to be used to interpret the script.

Comments: Lines that begin with a hash symbol (#) and are ignored by the shell.

Wildcards: Special characters that are interpreted by the shell to match filenames or patterns.

List variables: Variables that contain a list of values separated by whitespace.

Global variables: Variables that are accessible from any part of a script or shell session.

Extracting values from variables: Using shell expansions to extract a portion of a variable's value.

Arithmetic operators: Shell built-in commands for performing arithmetic operations.

Arguments: The values passed to a shell script or function when it is invoked. Conditional statements: Statements that execute different commands based on the truth value of a condition.

Loops: Constructs that repeatedly execute a series of commands.

Arrays: Variables that can store multiple values indexed by integers.

File testing: Shell built-in commands for testing file attributes such as existence, type, and permissions.

2.4 Shell Programming Questions

2.4.1 Concatenation of two strings:

To concatenate two strings in shell programming, we can use the + operator. For example, to concatenate the string "hello" and "world", we can use the following command:

[language=bash] str1="hello"
str2="world"
concatenated=*str*1str2
echo *concatenated*

This will output "helloworld".

Comparison of two strings: To compare two strings in shell programming, we can use the = operator. For example, to check if the strings "hello" and "world" are equal, we can use the following command:

```
    a=10
b=20
c=30
if [ $a -gt $b ] && [ $a -gt $c ]; then
  echo "Maximum number is $a"
elif [ $b -gt $c ]; then
  echo "Maximum number is $b"
else
  echo "Maximum number is $c"
fi
```

Maximum of three numbers: To find the maximum of three numbers in shell programming, we can use the if statement. For example, to find the maximum of 10, 20 and 30, we can use the following command: verbatim

```
a=10
b=20
c=30
if [ $a -gt $b ] && [ $a -gt $c ]; then
  echo "Maximum number is $a"
elif [ $b -gt $c ]; then
  echo "Maximum number is $b"
else
  echo "Maximum number is $c"
fi
```

Fibonacci series: To generate a Fibonacci series in shell programming, we can use a loop. For example, to generate the first 10 numbers in the Fibonacci series, we can use the following command:

```
a=0
b=1
echo "Fibonacci series:"
for (( i=0; i<10; i++ )); do
  echo $a
  c=$((a + b))
  a=$b
  b=$c
done
```

Arithmetic operations using case: To perform arithmetic operations using case statements in shell programming, we can use the case statement. For example, to perform addition, subtraction, multiplication or division, we can use the following command:

```
echo "Enter two numbers:"
read a b
echo "Enter an operation (add, sub, mul, div):"
read op
case $op in
  add)
    echo "Result: $(($a + $b))"
    ;;
  sub)
    echo "Result: $(($a - $b))"
    ;;
  mul)
    echo "Result: $(($a * $b))"
    ;;
  div)
    echo "Result: $(($a / $b))"
    ;;
  *)
    echo "Invalid operation"
    ;;
```

```
esac
```

Process Creation:

To write a program to create a process in UNIX algorithm.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pid, status;
    pid = fork();
    if (pid < 0) {
        printf("Error: Failed to create process.\n");
    }
    else if (pid == 0) {
        printf("Child process: pid = %d\n", getpid());
        // execute command here
    }
    else {
        printf("Parent process: pid = %d\n", getpid());
        waitpid(pid, &status, 0);
    }
    return 0;
}
```

To write a program for executing a commands.

```
echo Program for executing UNIX command using shell programming
echo Welcome
ps
exec wc e
```

To create child with sleep commands.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid;
    pid = fork();

    if (pid < 0) {
        printf("Error: Failed to create process.\n");
    }
    else if (pid == 0) {
        printf("This is child process: pid = %d\n", getpid());
        sleep(2); // sleep for 2 seconds
    }
```

```
    else {
        printf("Parent process: pid = %d\n", getpid());
    }
    return 0;
}
```

To create child with sleep commands using getpid.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid;
    pid = fork();

    if (pid < 0) {
        printf("Error: Failed to create process.\n");
    }
    else if (pid == 0) {
        // Child process
        printf("child process\n");
        printf("child process id is %d\n", getpid());
        printf("its parent process id is %d\n", getppid());
        sleep(5); // sleep for 5 seconds
        printf("child process after sleep=5\n");
        printf("child process id is %d\n", getpid());
        printf("its parent process id is %d\n", getppid());
    }
    else {
        // Parent process
        printf("parent process\n");
        sleep(10); // sleep for 10 seconds
        printf("child process after sleep=10\n");
        printf("child id is %d\n", pid);
        printf("parent id is %d\n", getpid());
        printf("parent terminates\n");
    }

    return 0;
}
```

To create a program for signal handeling in UNIX.

```
echo program for performing KILL operations
ps
echo enter the pid
read pid
kill -9 $pid
echo
finished
```

To perform wait commands using c program.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pid, status;
    pid = fork();

    if (pid < 0) {
        printf("Error: Failed to create process.\n");
    }
    else if (pid == 0) {
        // Child process
        for (int i = 1; i <= 10; i++) {
            printf("Child process: i = %d\n", i);
        }
        _exit(0);
    }
    else {
        // Parent process
        wait(&status);
      printf("Parent process: Child process terminated with status %d\n", status);
    }
    return 0;
}
```

To write a C program to simulate the operations of "ls' commands in UNIX.

```
#include<stdio.h>
#include<sys/types.h>
#include<dirent.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
    DIR *dp;
    struct dirent *dirp;
    if(argc<2)
        {
        printf("\n You have provided only 1 argument\n");
        exit(0);
        }
    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);
    }
```

SHORTEST JOB FIRST

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

include <stdio.h>

typedef struct Process int pid; int ser; int wait; Process;

int main(void) int i, j, n, tot = 0, avwait, totwait = 0, tturn = 0, aturn; Process p[20], tmp;

// Get the number of processes and their service time printf("Enter the number of processes: "); scanf("

for (i = 0; i < n; i++) printf("Enter the service time for process scanf("

// Validate the input to ensure that the entered service time is not negative if (p[i].ser < 0) printf("Service time cannot be negative."); return 1;

p[i].pid = i + 1; p[i].wait = 0;

// Sort the processes based on their service time using bubble sort 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;

// Calculate the waiting and turnaround time for each process and the total waiting and turnaround time printf("PID");

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("

// Calculate the average waiting and turnaround time avwait = totwait / n; aturn = tturn / n;

// Display the results printf("TOTAL WAITING TIME: printf("AVERAGE WAITING TIME: printf("TOTAL TURNAROUND TIME: printf("AVERAGE TURNAROUND TIME:

return 0;

ROUND ROBIN

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

```
#include<stdio.h>

struct Process {
    char pname[5];
    int pburst, pburst1, wtime, endtime, arrivt, is_processed;
};

int n, tq;

void input(struct Process p[]);
```

```
void initialize(struct Process p[]);
void calculate(struct Process p[]);
void display_waittime(struct Process p[]);

int main() {
    struct Process p[5];
    input(p);
    initialize(p);
    calculate(p);
    display_waittime(p);
    return 0;
}

void input(struct Process p[]) {
    printf("Enter the total number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter process name: ");
        scanf("%s", p[i].pname);
        printf("Enter process burst time: ");
        scanf("%d", &p[i].pburst);
        printf("Enter process arrival time: ");
        scanf("%d", &p[i].arrivt);
    }
    printf("\nEnter the time quantum/Time Slice: ");
    scanf("%d", &tq);
}

void initialize(struct Process p[]) {
    for (int i = 0; i < n; i++) {
        p[i].pburst1 = p[i].pburst;
        p[i].wtime = 0;
        p[i].endtime = 0;
        p[i].is_processed = 0;
    }
}

void calculate(struct Process p[]) {
    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) && (p[i].arrivt <= count) && (p[i].is_processed == 0)) {
                p[i].wtime = count - p[i].arrivt;
                p[i].endtime = count;
                p[i].is_processed = 1;
                k++;
            }
            if ((p[i].pburst1 > tq) && (p[i].arrivt <= count)) {
```

```
                if (p[i].is_processed == 1) {
                    p[i].is_processed = 0;
                } else {
                    p[i].wtime = p[i].wtime + (count - p[i].endtime);
                }
                count = count + tq;
                p[i].pburst1 = p[i].pburst1 - tq;
                p[i].endtime = count;
                printf("%d %s| ", count, p[i].pname);
        } else if ((p[i].pburst1 > 0) && (p[i].pburst1 <= tq) && (p[i].arrivt <= count)) {
                if (p[i].is_processed == 1) {
                    p[i].is_processed = 0;
                } else {
                    p[i].wtime = p[i].wtime + (count - p[i].endtime);
                }
                count = count + p[i].pburst1;
                p[i].endtime = count;
                printf("%d %s| ", count, p[i].pname);
                p[i].pburst1 = 0;
                j++;
            } else if (j == n) {
                flag = 0;
            }
        }
    }
}

void display_waittime() {
    int i;
    float 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 += a[i].wtime;
        turn += a[i].endtime - a[i].arrivt;
    }

    printf("\n\n\tAverage waiting time=%.2f", tot / n);
    printf("\n\tAverage turnaround time=%.2f\n", turn / n);
}
```

PRIORITY SCHEDULING

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

```
#include<stdio.h>
#include<stdlib.h>

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("\n Enter the process no burst and arrivaltime");
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);
}
}
```

FIRST COME FIRST SERVE

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

```
#include<stdio.h>

void main()
{
    int i, j, n, t;
    int turn[20], burst[20], arrival[20], waiting[20], process[20];
    float avg_wait, avg_turn, total_wait = 0, total_turn = 0;

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

    printf("Enter the process number, burst time, and arrival time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d %d %d", &process[i], &burst[i], &arrival[i]);
    }

    // sort processes based on arrival time
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arrival[j] > arrival[j + 1]) {
                // swap arrival times
                t = arrival[j];
                arrival[j] = arrival[j + 1];
                arrival[j + 1] = t;

                // swap burst times
                t = burst[j];
                burst[j] = burst[j + 1];
                burst[j + 1] = t;

                // swap process numbers
                t = process[j];
                process[j] = process[j + 1];
                process[j + 1] = t;
            }
        }
    }

    // calculate waiting time and turn around time
    for (i = 0; i < n; i++) {
```

```
        if (i == 0) {
            waiting[i] = 0;
            turn[i] = burst[i];
        } else {
            turn[i] = turn[i - 1] + burst[i];
            waiting[i] = turn[i] - burst[i] - arrival[i];
        }

        total_wait += waiting[i];
        total_turn += turn[i];
    }

    avg_wait = total_wait / n;
    avg_turn = total_turn / n;

    printf("Process\tBurst\tArrival\tWaiting\tTurnaround\n");
    for (i = 0; i < n; i++) {
     printf("%d\t%d\t%d\t%d\t%d\n", process[i], burst[i], arrival[i], waiting[i], turn[i]);
    }

    printf("Average waiting time: %.2f\n", avg_wait);
    printf("Average turnaround time: %.2f\n", avg_turn);
}
```

PRIORITY SCHEDULING:

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

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int base[20], limit[20], n, i, segment_number, offset, physical_address;

    printf("Program for segmentation\n");

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

    printf("Enter the base address and limit register for each segment:\n");
    for (i = 0; i < n; i++) {
        printf("Segment %d:\n", i);
        scanf("%d%d", &base[i], &limit[i]);
    }

    printf("Enter the logical address (segment number and offset): ");
    scanf("%d%d", &segment_number, &offset);

    if (segment_number < 0 || segment_number >= n) {
```

```
        printf("Invalid segment number\n");
        exit(1);
    }

    if (offset < 0 || offset >= limit[segment_number]) {
        printf("Offset out of range\n");
        exit(1);
    }

    physical_address = base[segment_number] + offset;
   printf("\n\tSegmentNo.\tBaseAdd.\tPhysicalAdd.\n\t%d\t\t%d\t\t%d\n", segment_number, base[seg

    return 0;
}
```

Memory Management Scheme- Paging

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int base[20], limit[20], num_segments, logical_address, segment_number, offset;
    printf("Program for segmentation\n");

    // Input the number of segments and their base and limit registers
    printf("Enter the number of segments: ");
    scanf("%d", &num_segments);
    printf("Enter the base address and limit register for each segment:\n");
    for (int i = 0; i < num_segments; i++) {
        scanf("%d %d", &base[i], &limit[i]);
    }

    // Input the logical address
    printf("Enter the logical address: ");
    scanf("%d", &logical_address);

    // Find the segment number and offset from the logical address
    segment_number = -1;
    offset = -1;
    for (int i = 0; i < num_segments; i++) {
        if (logical_address >= base[i] && logical_address < (base[i] + limit[i])) {
            segment_number = i;
            offset = logical_address - base[i];
            break;
        }
    }

    // If the logical address is valid, compute the physical address and print it
```

```
    if (segment_number >= 0 && offset >= 0) {
        int physical_address = base[segment_number] + offset;
     printf("\n\tSegmentNo.\tBaseAdd.\tPhysicalAdd.\n\t%d\t\t%d\t\t%d\n", segment_number, base[s
        return 0;
    } else {
        printf("\nInvalid segment\n");
        return 1;
    }
}
```

Producer Consumer Problem using Semaphore

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

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_ITEMS 10
int buffer[MAX_ITEMS];
int empty, full = 0, mutex = 1; // Semaphores
int item, itemC, n;
int in = 0, out = 0;

int wait(int s) {
    return --s;
}

int signal(int s) {
    return ++s;
}

void producer() {
    mutex = wait(mutex);
    empty = wait(empty);
    full = signal(full);
    printf("Enter an item: ");
    scanf("%d", &item);
    buffer[in] = item;
    in = (in + 1) % n;
    mutex = signal(mutex);
}

void consumer() {
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    itemC = buffer[out];
    printf("Consumed item = %d \n", itemC);
    out = (out + 1) % n;
```

```
    mutex = signal(mutex);
}

void main() {
    printf("Enter the value of n: ");
    scanf("%d", &n);
    empty = n;

    int choice;
    printf("\nChoices: \n1. Producer \n2. Consumer \n3. Exit");

    while (1) {
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            if (mutex == 1 && empty != 0)
                producer();
            else
                printf("Buffer is full \n");
            break;
        case 2:
            if (mutex == 1 && full != 0)
                consumer();
            else
                printf("Buffer is empty \n");
            break;
        default:
            exit(0);
            break;
        }
    }
}
```

Memory Management Scheme - Segmentation

To write a C program to implement memory management using segmentation

```
#include <stdio.h>
#include <stdlib.h>

void main() {
    int base_address[20], limit[20], num_segments, memory_limit;
    int segment_number, displacement, physical_address;

    printf("Enter number of segments: ");
    scanf("%d", &num_segments);
```

```
    printf("Enter memory limit: ");
    scanf("%d", &memory_limit);

    printf("\nEnter base address and limit of each segment:\n");
    for (int i = 0; i < num_segments; i++) {
        printf("Segment %d: ", i);
        scanf("%d %d", &base_address[i], &limit[i]);
        if (base_address[i] + limit[i] > memory_limit) {
            printf("Invalid memory limit \n");
            exit(0);
        }
    }

    printf("\nEnter the segment number and displacement value: ");
    scanf("%d %d", &segment_number, &displacement);

    if (segment_number >= num_segments || displacement >= limit[segment_number]) {
        printf("Invalid segment number or displacement.\n");
        exit(0);
    }

    // Calculate the physical address
    physical_address = base_address[segment_number] + displacement;
    printf("\nSegment No.\tBase Address\tPhysical Address\n");
   printf("%d\t\t%d\t\t%d\n", segment_number, base_address[segment_number], physical_address);
}
```

3. Discussions

The programs listed above cover a wide range of topics related to UNIX commands and shell programming. The first program deals with basic UNIX commands such as displaying date and time, printing text in the terminal, displaying the current working directory, and clearing the screen. It also covers terminal commands such as man, help, ls, cd, cat, mv, rm, sort, cp, wc, pg, head, tail, and more.

The second program is focused on the vi editor in UNIX and provides an overview of the various modes such as commands and input mode. It also covers commands such as vi +n <filename>, vi -n <filename>, and various other commands that can be used in vi.

The third program deals with UNIX shell programming commands, covering various keywords such as 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, and umask. It also covers general shell concepts such as comments, wildcards, variables, arithmetic operators, conditional statements, loops, arrays, and file testing.

The questions posed at the end of the program cover various programming concepts such as string concatenation, comparison, arithmetic operations using case, process creation, executing commands, sleep commands, signal handling, wait commands, file reading and writing, and memory management schemes such as paging and segmentation.

Overall, these programs provide a good overview of various concepts related to UNIX commands

and shell programming and can be useful for anyone learning or working with UNIX systems. 852

4. Conclusion 853

In conclusion, the topics covered in this report include various commands and operations in Unix, Vi editor, and Unix shell programming. The first part of the report discussed the use of Unix commands for displaying date and time, calendar, printing text, and manipulating directories and files using various terminal commands. The second part of the report focused on vi editor commands, including input and commands modes, navigating files, editing text, and saving changes. 854 855 856 857 858

Finally, the report discussed Unix shell programming, including common shells and shell keywords, general shell concepts such as wildcards, variables, conditional statements, loops, and file testing. The report also provided sample questions covering string manipulation, arithmetic operations, process creation, file reading and writing, and memory management schemes such as paging and segmentation. 859 860 861 862 863

Overall, this report provides an introduction to the basics of Unix, Vi editor, and Unix shell programming, which are essential skills for any programmer or system administrator working in a Unix environment. 864 865 866