



ACAD-DI-86	Lab Manual	Academic Year: 2024-25
Rev : 01		Semester: II
Date: 02.04.2021		

Subject: Operating Systems Lab (BTCOL406)

List of Experiments

Exp. No.	Name of Experiments
01	Hands on Unix Commands.
02	Shell Script programming using the commands grep, awk, and sed.
03	Implementation of various CPU scheduling algorithms (FCFS, SJF).
04	Concurrent programming; use of threads and processes, system calls (fork and v-fork).
05	Study Pthreads and implement the following: write a program which shows a performance.
06	Implementation of Producer-Consumer problem.
07	Implementation of Bankers algorithm.
08	Implementation of various page replacement algorithms (FIFO, LRU).
09	Implementation of various memory allocation algorithms, (First fit, Best fit and Worst fit).
10	Scheduling algorithms (FCFS, SCAN).



Experiment No: 01

Aim: Hands on Unix Commands.

Objectives: To study and implement various UNIX Commands.

Theory:

UNIX:

It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969. Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS. By 1980, UNIX had been completely rewritten using C language.

LINUX:

It is similar to UNIX, which is created by Linus Torvalds. All UNIX commands work in Linux. Linux is open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

STRUCTURE OF A LINUX SYSTEM:

It consists of three parts.

- a) UNIX kernel
- b) Shells
- c) Tools and Applications

UNIX KERNEL:

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS. Decides when one program stops and another starts.

SHELL:

Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them.

CONTENT:

- 1) **gedit:** Use to open a text editor to create the C source code.

gedit

Ex. gedit

Enter the C source code below:

```
int main()
{
    int A, B, sum = 0;

    printf("Enter two numbers A and B: \n");

    scanf("%d%d", &A, &B);

    // Calculate the addition of A and B
    // using '+' operator
    sum = A + B;

    // Print the sum
    printf("Sum of A and B is: %d", sum);

    return 0;
}
```

Save the program and close the editor window.

2) gcc -o filename filename.c : Use to compile a C program.

Ex.

gcc -o sum sum.c

This command will invoke the GNU C compiler to compile the file sum.c and output (-o) the result to an executable called sum.

3) ./filename : Use to run a C program:

Ex. ./sum

This should result in the output

Enter two numbers A and B:
4 5

Sum of A and B is: 9

```
student@Student: ~  
student@Student:~$ gcc -o sum sum.c  
student@Student:~$ ./sum  
Enter two numbers A and B :  
4 5  
Sum of A and B is: 9student@Student:~$
```

- 4) **date:** used to check the date and time
Syntax: \$date [+format]

Ex. \$date

Output:

Wednesday 29 March 2023 02:02:06 PM IST

List of Format specifiers used with date command:

%D: Display date as mm/dd/yy.

%d: Display the day of the month (01 to 31).

%a: Displays the abbreviated name for weekdays (Sun to Sat).

%A: Displays full weekdays (Sunday to Saturday).

%h: Displays abbreviated month name (Jan to Dec).

%b: Displays abbreviated month name (Jan to Dec).

%B: Displays full month name(January to December).

%m: Displays the month of year (01 to 12).

%y: Displays last two digits of the year(00 to 99).

%Y: Display four-digit year.

%T: Display the time in 24 hour format as HH:MM:SS.

%H: Display the hour.

%M: Display the minute.

%S: Display the seconds.

```
student@Student: ~  
student@Student:~$ date  
Wednesday 07 February 2024 10:28:29 AM IST  
student@Student:~$ date %a  
date: invalid date '%a'  
student@Student:~$ date "+%D"  
02/07/24  
student@Student:~$ date "+%a"  
Wed  
student@Student:~$ date "+%B"  
February  
student@Student:~$ date "+%b"  
Feb  
student@Student:~$ date "+%h"  
Feb  
student@Student:~$ date "+%Y-%m-%d"  
2024-02-07  
student@Student:~$ date "+%A %B %d %T %y"  
Wednesday February 07 10:31:29 24  
student@Student:~$
```

- 5) **cal:** used to display the calendar.
Syntax:\$cal

```
student@Student: ~  
student@Student:~$ cal  
February 2024  
Su Mo Tu We Th Fr Sa  
1 2 3  
4 5 6 7 8 9 10  
11 12 13 14 15 16 17  
18 19 20 21 22 23 24  
25 26 27 28 29  
student@Student:~$ cal 2024  
2024  
January February March  
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  
1 2 3 4 5 6 1 2 3 1 2  
7 8 9 10 11 12 13 4 5 6 7 8 9 10 3 4 5 6 7 8 9  
14 15 16 17 18 19 20 11 12 13 14 15 16 17 10 11 12 13 14 15 16  
21 22 23 24 25 26 27 18 19 20 21 22 23 24 17 18 19 20 21 22 23  
28 29 30 31 25 26 27 28 29 24 25 26 27 28 29 30  
31  
April May June  
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  
1 2 3 4 5 6 1 2 3 4 1  
7 8 9 10 11 12 13 5 6 7 8 9 10 11 2 3 4 5 6 7 8  
14 15 16 17 18 19 20 12 13 14 15 16 17 18 9 10 11 12 13 14 15  
21 22 23 24 25 26 27 19 20 21 22 23 24 25 16 17 18 19 20 21 22  
28 29 30 26 27 28 29 30 31 23 24 25 26 27 28 29  
30  
July August September  
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  
1 2 3 4 5 6 1 2 3 1 2 3 4 5 6 7  
7 8 9 10 11 12 13 4 5 6 7 8 9 10 8 9 10 11 12 13 14  
14 15 16 17 18 19 20 11 12 13 14 15 16 17 15 16 17 18 19 20 21  
21 22 23 24 25 26 27 18 19 20 21 22 23 24 22 23 24 25 26 27 28  
28 29 30 31 25 26 27 28 29 30 31 29 30  
October November December  
Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa  
1 2 3 4 5 1 2 1 2 3 4 5 6 7  
6 7 8 9 10 11 12 3 4 5 6 7 8 9 8 9 10 11 12 13 14  
13 14 15 16 17 18 19 10 11 12 13 14 15 16 15 16 17 18 19 20 21  
20 21 22 23 24 25 26 17 18 19 20 21 22 23 22 23 24 25 26 27 28  
27 28 29 30 31 24 25 26 27 28 29 30 29 30 31
```

6) **echo** : used to print the message on the screen.

Syntax: \$echo "text"

```
student@Student:~$ echo "This is Ubuntu Os"  
This is Ubuntu Os  
student@Student:~$
```

7) **ls** : used to list the files.

Syntax: \$ls

```
student@Student: ~  
student@Student:~$ ls  
1jenny8 83.c 9 al1052 al25.c alvfork exe_eight exp_8.c file1 Fork.c Music result sjf  
1jenny8.c 8jenny1 91 al1052.c al264 alvfork.c exe_eight.c expr25.c file2 Hello Pictures resulti sjf.c  
2 8jenny1.c 91.c al105.c al264.c check exp_10 expr25.sh file3 Hello.c pp.c resulti sakshi  
2.c 8jenny2 9.c al1904 atpc check1 exp10 fcfs First Hello.txt pp.c prachi sakshi  
82 8jenny2.c al0 at1904.c atpc.c Desktop exp_10.c FCFS First Hello.txt pp.c prachi sakshi  
82.c 8jenny3 al0.c al25 atsjf Documents exp10.c fcfs.c fork hello.c producer.c shivam.c sun.c  
83 8jenny3.c al105 al251204.c alsjf.c Downloads exp_8 FCFS.c fork.c 'hello w' Public shweta Templates  
student@Student:~$
```

8) **who & whoami** –it displays data about all users who have logged into the system currently. The next command displays about current user only.

Syntax:\$who

student tty2 2023-03-29 13:54 (tty2)

\$whoami

Student

9) **uptime**: tells you how long the computer has been running since its last reboot or power-off.

Syntax:\$uptime

14:36:39 up 43 min, 1 user, load average: 0.71, 0.66, 0.63

10) **uname**: it displays the system information such as hardware platform, system name and processor, OS type.

Syntax:\$uname

Linux

11) **hostname** –displays and set system host name

Syn:\$ hostname

```
student@Student: ~  
student@Student:~$ who  
student tty2 2024-02-07 08:46 (tty2)  
student@Student:~$ whoami  
student  
student@Student:~$ hostname  
Student  
student@Student:~$ uname  
Linux  
student@Student:~$ uptime  
10:47:08 up 2:01, 1 user, load average: 0.32, 0.22, 0.25  
student@Student:~$
```

12) **bc** –stands for “best calculator”

Syntax:\$bc

10/2*3

15

Quit

```
student@Student: ~  
student@Student:~$ bc  
bc 1.07.1  
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundation, Inc.  
This is free software with ABSOLUTELY NO WARRANTY.  
For details type 'warranty'.  
7*5  
35  
5+5  
10  
quit  
student@Student:~$
```

13) **\$cat>filename** : This command is used to create a file.

Syntax: \$cat>filename

14) **\$cat filename** : This command is used to view a file

Syntax:\$cat filename

15) **\$cat>>filename** : this command is used to add text to an existing file:

Syntax:\$cat>>filename

```
student@Student: ~  
student@Student:~$ cat>fruits  
mango  
grapes  
banan  
watermelon  
chikko  
^C  
student@Student:~$ cat fruits  
mango  
grapes  
banan  
watermelon  
chikko  
student@Student:~$ cat>>fruits  
orange  
papaya  
^C  
student@Student:~$ cat fruits  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
student@Student:~$
```

16) **Concatenate:**

Syn:\$cat file1 file2>file3

```
student@Student: ~  
student@Student:~$ cat fruits  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
student@Student:~$ cat vegetables  
pees  
beans  
palak  
cabbage  
flower  
student@Student:~$ cat fruits vegetables>concat  
student@Student:~$ cat concat  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
pees  
beans  
palak  
cabbage  
flower  
student@Student:~$
```


17) **rm**—deletes a file from the file system

Syn:\$rm filename

```
student@Student: ~  
student@Student: $ ls  
1jenny8      83.c      9      ai1052    ai25.c    aivfork  Downloads  exp_8      FCFS.c    fork.c    hello.c    producer.c  shivamm.c  Templates  
1jenny8.c    8jenny1   91      ai1052.c  ai264     aivfork.c exe_eight  exp_8.c    file1     Fork.c    'hello w'  Public     shweta     vegetables  
2            8jenny1.c 91.c    ai105.c   ai264.c   check    exe_eight.c expr25.c   file2     fruits    Music      result     sjf        Videos  
2.c          8jenny2   9.c     ai1904    atpc      check1   exp_10     expr25.sh file3     Hello    Pictures   result1    sjf.c  
82           8jenny2.c ai0      ai1904.c  atpc.c    concat   exp10      fcf5       first     Hello.c   pp.c       sakshi     snap  
82.c         8jenny3   ai0.c   ai25      atsjf     Desktop  exp_10.c   FCFS      First     Hello.txt prachi     SE_AI     sum  
83           8jenny3.c ai105    ai251204.c atsjf.c   Documents exp10.c    fcf5.c    Fork      hello     princy     shivan.c  sum.c  
student@Student: $ rm hello1  
student@Student: $ ls  
1jenny8      83.c      9      ai1052    ai25.c    aivfork  Downloads  exp_8      FCFS.c    fork.c    'hello w'  Public     shweta     vegetables  
1jenny8.c    8jenny1   91      ai1052.c  ai264     aivfork.c exe_eight  exp_8.c    file1     Fork.c    Music      result     sjf        Videos  
2            8jenny1.c 91.c    ai105.c   ai264.c   check    exe_eight.c expr25.c   file2     fruits    Pictures   result1    sjf.c  
2.c          8jenny2   9.c     ai1904    atpc      check1   exp_10     expr25.sh file3     Hello    pp.c       sakshi     snap  
82           8jenny2.c ai0      ai1904.c  atpc.c    concat   exp10      fcf5       first     Hello.c   prachi     SE_AI     sum  
82.c         8jenny3   ai0.c   ai25      atsjf     Desktop  exp_10.c   FCFS      First     Hello.txt princy     shivan.c  sum.c  
83           8jenny3.c ai105    ai251204.c atsjf.c   Documents exp10.c    fcf5.c    Fork      hello     producer.c shivamm.c  Templates
```

18) **cp**—copies the files or directories

Syn:\$cp source file destination file

```
student@Student: ~  
student@Student: $ cat vegetables  
pees  
beans  
palak  
cabbage  
flower  
student@Student: $ cat fruits  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
student@Student: $ cat concat  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
pees  
beans  
palak  
cabbage  
flower  
student@Student: $ cp fruits concat  
student@Student: $ cat concat  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
student@Student: $
```

19) **mv**—to rename the file or directory

syn:\$mv old file new file

Eg:\$mv abc xyz

```
student@Student: ~  
student@Student:~$ ls  
1jenny8 83.c 9 ai1052 ai25.c aivfork Downloads exp_8 FCFS.c fork.c hello.c producer.c shivam.c Templates  
1jenny8.c 8jenny1 91 ai1052.c ai264 aivfork.c exe_eight exp_8.c file1 Fork.c 'hello w' Public shweta vegetables  
2 8jenny1.c 91.c ai105.c ai264.c check exe_eight.c expr25.c file2 fruits Music result sjf Videos  
2.c 8jenny2 9.c ai1904 atpc check1 exp_10 expr25.sh file3 Hello Pictures result1 sjf.c  
82 8jenny2.c ai0 ai1904.c atpc.c concat exp_10 fcfs first Hello.c pp.c sakshi snap  
82.c 8jenny3.c ai105 ai25 atsjf Desktop exp_10.c FCFS First Hello.txt prachi SE_AI sum  
83 8jenny3.c ai105 ai251204.c atsjf.c Documents exp10.c fcfs.c fork hello princy shivam.c sun.c  
student@Student:~$ mv hello hello1  
student@Student:~$ ls  
1jenny8 83.c 9 ai1052 ai25.c aivfork Downloads exp_8 FCFS.c fork.c hello.c producer.c shivam.c Templates  
1jenny8.c 8jenny1 91 ai1052.c ai264 aivfork.c exe_eight exp_8.c file1 Fork.c 'hello w' Public shweta vegetables  
2 8jenny1.c 91.c ai105.c ai264.c check exe_eight.c expr25.c file2 fruits Music result sjf Videos  
2.c 8jenny2 9.c ai1904 atpc check1 exp_10 expr25.sh file3 Hello Pictures result1 sjf.c  
82 8jenny2.c ai0 ai1904.c atpc.c concat exp_10 fcfs first Hello.c pp.c sakshi snap  
82.c 8jenny3.c ai0.c ai25 atsjf Desktop exp_10.c FCFS First Hello.txt prachi SE_AI sum  
83 8jenny3.c ai105 ai251204.c atsjf.c Documents exp10.c fcfs.c fork hello1 princy shivam.c sun.c  
student@Student:~$
```

20) head—displays 10 lines from the head(top) of a given file

Syn: \$head filename

Eg: \$head concat

To display the top two lines:

Syn: \$head -2 concat

21) tail—displays last 10 lines of the file

Syn: \$tail filename

Eg: \$tail -2 concat

```
student@Student: ~  
student@Student:~$ cat concat  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
student@Student:~$ head -4 concat  
mango  
grapes  
banan  
watermelon  
student@Student:~$ tail -3 concat  
chikko  
orange  
papaya  
student@Student:~$
```

22) wc—it counts the number of lines, words, character in a specified file(s)

\$ wc state.txt

5 7 58 state.txt



PCET-NMVPM's

Nutan College of Engineering and Research, Talegaon, Pune

DEPARTMENT OF CSE-ARTIFICIAL INTELLIGENCE



```
student@Student: ~  
student@Student:~$ cat concat  
mango  
grapes  
banan  
watermelon  
chikko  
orange  
papaya  
student@Student:~$ head -4 concat  
mango  
grapes  
banan  
watermelon  
student@Student:~$ tail -3 concat  
chikko  
orange  
papaya  
student@Student:~$ wc concat  
7 7 51 concat  
student@Student:~$
```



Experiment No: 02

Aim: Shell Script programming using the commands grep, awk, and sed.

Objectives: To study and implement various Shell Script programming UNIX Commands like grep, awk, sed.

Theory:

Grep, sed, and AWK are all standard Linux tools that are able to process text. Each of these tools can read text files line-by-line and use regular expressions to perform operations on specific parts of the file. Grep is used for finding text patterns in a file and is the simplest of the three. Sed can find and modify data, however, its syntax is a bit more complex than grep. AWK is a full-fledged programming language that can process text and perform comparison and arithmetic operations on the extracted text.

grep:

grep is a Linux utility used to find lines of text in files or input streams using regular expressions. It's name is short for Global Regular Expression Pattern

Beginning at the first line in the file, grep copies a line into a buffer, compares it against the search string, and if the comparison passes, prints the line to the screen. Grep will repeat this process until the Example data file.

Here is the basic syntax of the grep command:

```
grep [options] pattern [file...]
```

Ex. Consider a text file 'good'

goodness

goodlier

goodwills

goodwives

goodies

goodbyes

wellness

morning

sun

- 1) **“pattern”** : To print any line from a file that contains a specific pattern of characters.

\$grep “good” good

```
student@Student: ~  
student@Student:~$ cat good  
goodness  
goodlier  
goodwills  
goodwives  
goodies  
goodbyes  
wellness  
morning  
sun  
student@Student:~$ grep "good" good  
goodness  
goodlier  
goodwills  
goodwives  
goodies  
goodbyes  
student@Student:~$
```

- 2) **–n** : When grep prints results with many matches, it comes handy to see the line numbers.

\$grep –n “good” good

- 3) **–vn** : To print any line from a file that doesn’t contain a specific pattern of characters.

\$grep –vn “good” good

- 4) **–i** : ignores the case of the text string.

\$grep –i “good” good

- 5) **–c** : To print a count of matching lines to standard output.

\$grep –c “good” good

```
student@Student: ~  
student@Student:~$ cat good  
goodness  
goodlier  
goodwills  
goodwives  
goodies  
goodbyes  
wellness  
morning  
sun  
student@Student:~$ grep -vn "good" good  
7:wellness  
8:morning  
9:sun  
10:  
student@Student:~$ grep -n "good" good  
1:goodness  
2:goodlier  
3:goodwills  
4:goodwives  
5:goodies  
6:goodbyes  
student@Student:~$ grep -c "good" good  
6  
student@Student:~$ grep -i "good" good  
goodness  
goodlier  
goodwills  
goodwives  
goodies  
goodbyes  
student@Student:~$
```

awk:

Awk is a scripting language used for manipulating data and generating reports. The awk command programming language requires no compiling and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform the associated actions.

Awk is abbreviated from the names of the developers – Aho, Weinberger, and Kernighan.

awk 'pattern { action }' file

The awk command in Ubuntu (and other Unix-like operating systems) is a versatile text-processing tool used for pattern scanning and processing. It operates on a per-line basis, allowing users to specify patterns and actions to perform on lines that match those patterns. awk derives its name from the initials of its designers: Alfred Aho, Peter Weinberger, and Brian Kernighan.

Here is the basic syntax of the awk command:

awk 'pattern { action }' file

- **pattern:** This is a condition or a regular expression that is tested against each line of input. If the pattern matches, the associated action is executed.
- **action:** This is the action to be performed when the pattern matches. It can be any valid AWK script.

- **file:** This is the input file that awk will process. If not provided, awk reads from standard input.

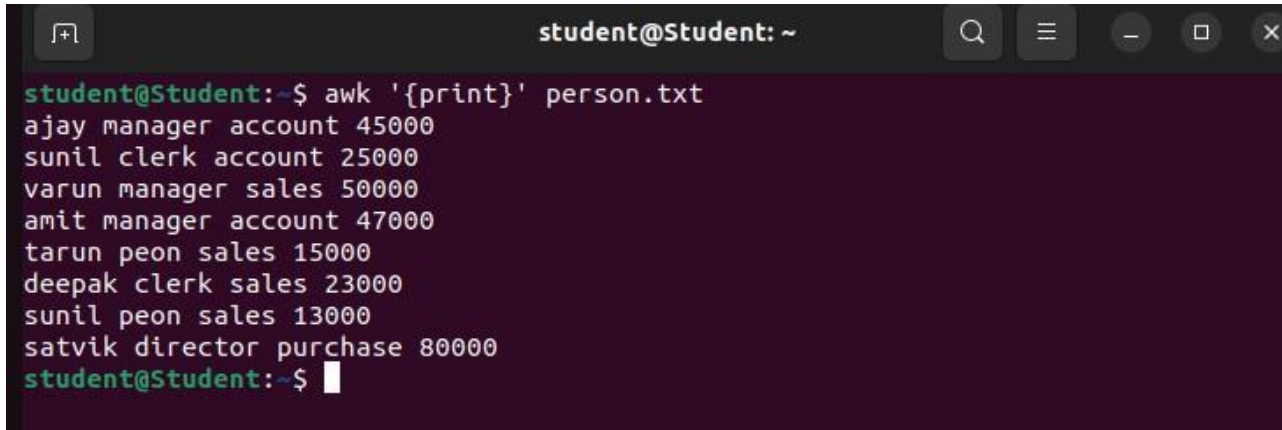
Consider the following text file as the input file for all cases below:

\$cat > person.txt

```
ajay manager account 45000
sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000
```

1. **Default behavior of Awk:** By default Awk prints every line of data from the specified file.

\$ awk '{print}' person.txt



```
student@Student: ~
student@Student:~$ awk '{print}' person.txt
ajay manager account 45000
sunil clerk account 25000
varun manager sales 50000
amit manager account 47000
tarun peon sales 15000
deepak clerk sales 23000
sunil peon sales 13000
satvik director purchase 80000
student@Student:~$
```

2. **Print the lines which match the given pattern.**

\$ awk '/account/' {print}' person.txt


```
student@Student: ~  
student@Student:~$ awk '{print}' person.txt  
ajay manager account 45000  
sunil clerk account 25000  
varun manager sales 50000  
amit manager account 47000  
tarun peon sales 15000  
deepak clerk sales 23000  
sunil peon sales 13000  
satvik director purchase 80000  
student@Student:~$ awk '/account/{print}' person.txt  
ajay manager account 45000  
sunil clerk account 25000  
amit manager account 47000  
student@Student:~$
```

3. Splitting a Line Into Fields : For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the \$n variables. If the line has 4 words, it will be stored in \$1, \$2, \$3 and \$4 respectively. Also, \$0 represents the whole line.

\$ awk '{print \$1,\$4}' person.txt

```
student@Student: ~  
student@Student:~$ awk '{print}' person.txt  
ajay manager account 45000  
sunil clerk account 25000  
varun manager sales 50000  
amit manager account 47000  
tarun peon sales 15000  
deepak clerk sales 23000  
sunil peon sales 13000  
satvik director purchase 80000  
student@Student:~$ awk '/account/{print}' person.txt  
ajay manager account 45000  
sunil clerk account 25000  
amit manager account 47000  
student@Student:~$ awk '{print $1,$4}' person.txt  
ajay 45000  
sunil 25000  
varun 50000  
amit 47000  
tarun 15000  
deepak 23000  
sunil 13000  
satvik 80000  
student@Student:~$
```


4. Use of NR built-in variables (Display Line Number)

```
$ awk '{print NR,$0}' person.txt
```

```
$ awk '{print NR,$1}' person.txt
```

```
student@Student: ~  
student@Student:~$ awk '{print NR,$0}' person.txt  
1 ajay manager account 45000  
2 sunil clerk account 25000  
3 varun manager sales 50000  
4 amit manager account 47000  
5 tarun peon sales 15000  
6 deepak clerk sales 23000  
7 sunil peon sales 13000  
8 satvik director purchase 80000  
student@Student:~$ awk '{print NR,$1}' person.txt  
1 ajay  
2 sunil  
3 varun  
4 amit  
5 tarun  
6 deepak  
7 sunil  
8 satvik  
student@Student:~$
```

5. Another use of NR built-in variables (Display Line From 3 to 6)

```
student@Student: ~  
student@Student:~$ awk 'NR==3, NR==6 {print NR,$0}' person.txt  
3 varun manager sales 50000  
4 amit manager account 47000  
5 tarun peon sales 15000  
6 deepak clerk sales 23000  
student@Student:~$ awk 'NR==3, NR==6 {print NR,$3}' person.txt  
3 sales  
4 account  
5 sales  
6 sales  
student@Student:~$
```

6 . Use of NF built-in variables (Display Last Field)

\$ awk '{print \$1,\$NF}' person.txt

```
student@Student: ~  
student@Student:~$ awk '{print $1,$NF}' person.txt  
ajay 45000  
sunil 25000  
varun 50000  
amit 47000  
tarun 15000  
deepak 23000  
sunil 13000  
satvik 80000  
student@Student:~$ awk '{print $3,$NF}' person.txt  
account 45000  
account 25000  
sales 50000  
account 47000  
sales 15000  
sales 23000  
sales 13000  
purchase 80000  
student@Student:~$
```

7. To find the length of the longest line present in the file:

\$ awk '{ if (length(\$0) > max) max = length(\$0) } END { print max }' person.txt

```
student@Student: ~  
student@Student:~$ awk '{ if (length($0) > max) max = length($0) } END { print m  
ax }' person.txt  
31
```

8. Print the number of lines in a file:

\$ awk 'END {print NR}' person.txt

```
student@Student: ~  
student@Student:~$ awk '{ if (length($0) > max) max = length($0) } END { print m  
ax }' person.txt  
31  
student@Student:~$ awk 'END { print NR }' person.txt  
8  
student@Student:~$
```

9. To print the squares of first numbers from 1 to n say 6:

\$ awk 'BEGIN { for(i=1;i<=6;i++) print "square of", i, "is",i*i; }'

```
student@Student: ~  
student@Student:~$ awk 'BEGIN { for(i=1;i<=6;i++) print "square of", i, "is",i*i  
; }'  
square of 1 is 1  
square of 2 is 4  
square of 3 is 9  
square of 4 is 16  
square of 5 is 25  
square of 6 is 36  
student@Student:~$
```

sed:

SED command in UNIX stands for stream editor and it can perform lots of functions on file like searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening them, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

Here is the basic syntax of the sed command:

sed options... [script] [inputfile...]

Consider the below text file as an input.

\$cat > geekfile.txt

unix is great os. unix is opensource. unix is free os.

learn operating system.



unix linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

1. Replacing or substituting string : Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word “unix” with “linux” in the file.

```
$sed 's/unix/linux/' unix
```

By default, the sed command replaces the first occurrence of the pattern in each line and it won't replace the second, third...occurrence in the line.

2. Replacing the nth occurrence of a pattern in a line : Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word “unix” with “linux” in a line.

```
$sed 's/unix/linux/2' unix
```

3. Replacing all the occurrence of the pattern in a line : The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.

```
$sed 's/unix/linux/g' unix
```

```
student@Student: ~  
student@Student:~$ cat>unix  
unix is great os. unix is opensource. unix is free os.  
learn operating system.  
unix linux which one you choose.  
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.  
^C  
student@Student:~$ sed 's/unix/linux/' unix  
linux is great os. unix is opensource. unix is free os.  
learn operating system.  
linux linux which one you choose.  
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.  
student@Student:~$ sed 's/unix/linux/2' unix  
unix is great os. linux is opensource. unix is free os.  
learn operating system.  
unix linux which one you choose.  
unix is easy to learn.linux is a multiuser os.Learn unix .unix is a powerful.  
student@Student:~$ sed 's/unix/linux/g' unix  
linux is great os. linux is opensource. linux is free os.  
learn operating system.  
linux linux which one you choose.  
linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.  
student@Student:~$
```

4. Parenthesize first character of each word : This sed example prints the first character of every word in parenthesis.

```
$ echo "Welcome to Unix Operating System" | sed 's/^(\\b[A-Z]\\)/^(\\1\\)/g'
```

```
student@Student: ~  
student@Student:~$ echo "Welcome to unix operating system" | sed 's/^(\\b[A-Z]\\)/^(\\1\\)/g'  
(W)elcome to unix operating system  
student@Student:~$ echo "Welcome to Unix Operating System" | sed 's/^(\\b[A-Z]\\)/^(\\1\\)/g'  
(W)elcome to (U)nix (O)perating (S)ystem  
student@Student:~$
```

5. Duplicating the replaced line with /p flag : The /p print flag prints the replaced line twice on the terminal. If a line does not have the search pattern and is not replaced, then the /p prints that line only once.

```
$sed 's/unix/linux/p' unix
```



```
student@Student: ~  
student@Student:~$ echo "Welcome to unix operating system" | sed 's/\\(\\b[A-Z]\\)/\\(\\1\\)/g'  
(W)elcome to unix operating system  
student@Student:~$ echo "Welcome to Unix Operating System" | sed 's/\\(\\b[A-Z]\\)/\\(\\1\\)/g'  
(W)elcome to (U)nix (O)perating (S)ystem  
student@Student:~$ sed 's/unix/linux/p' unix  
linux is great os. unix is opensource. unix is free os.  
linux is great os. unix is opensource. unix is free os.  
learn operating system.  
linux linux which one you choose.  
linux linux which one you choose.  
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.  
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.  
student@Student:~$
```

6. Deleting lines from a particular file : SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the file
Examples:

To Delete a particular line say n in this example

Syntax:

\$ sed 'nd' filename.txt

\$ sed '2d' filename.txt

```
student@Student: ~  
student@Student:~$ sed '2d' unix  
unix is great os. unix is opensource. unix is free os.  
unix linux which one you choose.  
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

Experiment No: 3

AIM: Implementation of FCFS and SJF CPU scheduling algorithms.

Objective: To study and implement CPU scheduling algorithms.

Theory:

FCFS Scheduling Algorithm:

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

Program:

1. Start.
2. Declare the array size.
3. Read the number of processes to be inserted
4. Read the arrival time of processes.
5. Read the burst time of processes.
6. Calculate the completion time of each process
7. Calculate the waiting time of each process: $wt[i] = tat[i] - bt[i]$
8. Calculate the turnaround time of each process: $tat[i] = ct[i] - at[i]$
9. Calculate the average waiting and average turnaround time
10. Display the values.
11. Stop

Source Code:

```
#include<stdio.h>

int main(){

    int bt[10]={0},at[10]={0},tat[10]={0},wt[10]={0},ct[10]={0}, p[10];

    int n,sum=0,i,j, temp=0;
```



```
float totalTAT=0,totalWT=0;

printf("Enter number of processes    ");

scanf("%d",&n);

printf("enter %d process:",n);

for(i=0;i<n;i++)
{
scanf("%d",&p[i]);

}   printf("Enter arrival time and burst time for each process\n\n");

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

printf("Arrival time of process[%d]    ",i+1);

scanf("%d",&at[i]);

printf("Burst time of process[%d]    ",i+1);

scanf("%d",&bt[i]);

printf("\n");

}

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

{

for(j=0; j<n; j++)

{

if(at[i]<at[j])
```



```
{  
  
    temp=at[i];  
  
    at[i]=at[j];  
  
    at[j]=temp;  
  
    temp=bt[i];  
  
    bt[i]=bt[j];  
  
    bt[j]=temp;  
  
    temp=p[j+1]; p[j+1]=p[j]; p[j]=temp;  
  
}  
  
}  
  
}          //calculate completion time of processes  
  
for(int j=0;j<n;j++)  
  
{  
  
    sum+=bt[j];  
  
    ct[j]+=sum;  
  
}  
  
//calculate turnaround time and waiting times  
  
for(int k=0;k<n;k++)  
  
{  
  
    tat[k]=ct[k]-at[k];  
  
    totalTAT+=tat[k];  
  
}
```

```
}  
  
for(int k=0;k<n;k++)  
  
{  
  
    wt[k]=tat[k]-bt[k];  
  
    totalWT+=wt[k];  
  
}  
    printf("Solution: \n\n");  
printf("P\t AT\t BT\t CT\t TAT\t WT\n\n");  
  
for(int i=0;i<n;i++)  
  
{  
  
    printf("P%d\t %d\t %d\t %d\t %d\t %d\n",p[i],at[i],bt[i],ct[i],tat[i],wt[i]);  
  
}  
  
printf("\n\nAverage Turnaround Time = %f\n",totalTAT/n);  
printf("Average WT = %f\n",totalWT/n);  
return 0;  
  
}
```

Output:

```
/tmp/TyUKQkcj6R.o
Enter number of processes  5
enter 5 process:1
2
3
4
5
Enter arrival time and burst time for each process

Arrival time of process[1]  0
Burst time of process[1]    2

Arrival time of process[2]  1
Burst time of process[2]    3

Arrival time of process[3]  3
Burst time of process[3]    4

Arrival time of process[4]  3
Burst time of process[4]    5
```

```
Arrival time of process[5] 4
Burst time of process[5] 5
```

Solution:

P	AT	BT	CT	TAT	WT
P3	0	2	2	2	0
P4	1	3	5	4	1
P2	3	5	10	7	2
P0	3	4	14	11	7
P1	4	5	19	15	10

Average Turnaround Time = 7.800000

Average WT = 4.000000

SJF Scheduling algorithm:

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

Program:

1. Start
2. Declare the array size
3. Read the number of processes to be inserted
4. Read the Burst times of processes
5. Sort the Burst times in ascending order and process with shortest burst time
6. Calculate the waiting time of each process: $wt[i] = wt[i-1] + bt[i-1]$
7. Calculate the turnaround time of each process: $tat[i] = tat[i-1] + bt[i]$
8. Calculate the average waiting time and average turnaround time.
9. Display the values
10. Stop

Source Code:

```
#include<stdio.h>
void main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg;

printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)

for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i]; wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\tPROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]); printf("\nAverage Waiting Time -- %f",
wtavg/n); printf("\nAverage Turnaround Time -- %f", tatavg/n);
}
```

Output:

```
Enter the number of processes -- 5
Enter Burst Time for Process 0 -- 7
Enter Burst Time for Process 1 -- 3
Enter Burst Time for Process 2 -- 2
Enter Burst Time for Process 3 -- 10
Enter Burst Time for Process 4 -- 8
```

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P2	2	0	2
P1	3	2	5
P0	7	5	12
P4	8	12	20
P3	10	20	30

```
Average Waiting Time -- 7.800000
Average Turnaround Time -- 13.800000
```

Experiment No: 4

AIM: Concurrent programming; use of threads and processes, systemcalls (fork and v-fork).

Objective: To study and implement use of threads, processes and fork v-forksystem call.

Theory:

Fork()

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a *new* process, which becomes the *child* process of the caller. After a new child process is created, *both* processes will execute the next instruction following the **fork()** system call. The child and parent processes are located in separate physical address spaces. As a result, the modification in the parent process doesn't appear in the child process. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the *process ID* of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Source Code:

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

    // Write C code here

    fork();

    printf("Process id=%d\n", getpid());

    return 0;

}
```

Output:

```
Process id=3530
Process id=3534
```

Vfork()

Similar to fork(), vfork() creates a new subprocess for the calling process. However, vfork() is specifically designed for subprocesses to execute exec() programs immediately.

vfork() creates a child process just like fork(), but it does not copy the address space of the parent process to the child process completely, because the child process will immediately call exec (or exit), so the address space will not be accessed. However, before a child process calls exec or exit, it runs in the space of the parent process. Another difference between vfork() and fork() is that vfork() ensures that the child process runs first, and that the parent process may not be scheduled until it calls exec or exit. (if the child process depends on further actions of the parent process before these two functions are called, a deadlock can result.)

Source Code:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
//main function begins
int main(){
    pid_t p= vfork(); //calling of fork system call
    if(p==-1)
        printf("Error occured while calling fork()");
    else if(p==0)
        printf("This is the child process with ID=%d\n", getpid());
    else
        printf("This is the parent processwith ID=%d\n", getpid());
    return 0;
}
```

Output:

```
This is the child process with ID=2975
This is the parent processwith ID=2974
```


Experiment No. 5

Aim: Study Pthreads and implement the following: write a program which shows a performance.

Objective: To study and implement use of pthreads to enhance the performance.

Theory:

POSIX Threads in OS :

The POSIX thread libraries are a C/C++ thread API based on standards. It enables the creation of a new concurrent process flow. It works well on multi-processor or multi-core systems, where the process flow may be scheduled to execute on another processor, increasing speed through parallel or distributed processing. Because the system does not create a new system, virtual memory space and environment for the process, threads need less overhead than “forking” or creating a new process. While multiprocessor systems are the most effective, benefits can also be obtained on uniprocessor systems that leverage delay in I/O and other system processes that may impede process execution.

To utilize the PThread interfaces, we must include the header pthread.h at the start of the Cscript.

```
#include <pthread.h>
```

PThreads is a highly concrete multithreading system that is the UNIX system's default standard. PThreads is an abbreviation for POSIX threads, and POSIX is an abbreviation for Portable Operating System Interface, which is a type of interface that the operating system must implement. PThreads in POSIX outline the threading APIs that the operating system must provide.

Source Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5 void
*PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread %ld!\n", tid);
    pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
```

```
pthread_t threads[NUM_THREADS];int rc;
long t;
for(t=0;t< NUM_THREADS;t++){
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);if (rc){
        printf("ERROR; return code from pthread_create() is %d\n", rc);exit(-1);
    }
}

/* Last thing that main() should do */
pthread_exit(NULL);
}
```

Compile & run(linux terminal):

```
gcc -o Pthread_Hello_world pthread_hello_world.c -lpthread
./Pthread_Hello_world
```

OUTPUT:

```
In main: creating thread 0 In
main: creating thread 1
Hello World! It's me, thread #0! In
main: creating thread 2
Hello World! It's me, thread #1! In
main: creating thread 3
Hello World! It's me, thread #2! In
main: creating thread 4
Hello World! It's me, thread #3!Hello
World! It's me, thread #4!
```



Experiment No. 6

Aim: Implementation of Producer-Consumer problem.

Objective: To understand process synchronization using Producer consumer problem.

Theory:

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size

// of buffer

int empty = 10, x = 0;

// Function to produce an item and

// add it to the buffer

void producer()

{

    // Decrease mutex value by 1
```



```
-----  
--mutex;  
  
// Increase the number of full  
  
// slots by 1  
  
++full;  
  
// Decrease the number of empty  
  
// slots by 1  
  
--empty;  
  
// Item produced  
  
x++;  
  
printf("\nProducer produces"  
      "item %d",  
      x);  
  
// Increase mutex value by 1  
  
++mutex;  
  
}  
  
// Function to consume an item and  
// remove it from buffer  
  
void consumer()  
{  
  
// Decrease mutex value by 1  
  
--mutex;  
  
// Decrease the number of full  
  
// slots by 1  
  
--full;
```



```
-----  
    // Increase the number of empty  
  
    // slots by 1  
  
    ++empty;  
  
    printf("\nConsumer consumes "  
        "item %d",  
        x);  
  
    x--;  
  
    // Increase mutex value by 1  
  
    ++mutex;  
  
}  
  
// Driver Code  
  
int main()  
{  
  
    int n, i;  
  
    printf("\n1. Press 1 for Producer"  
        "\n2. Press 2 for Consumer"  
        "\n3. Press 3 for Exit");  
  
  
    // Using '#pragma omp parallel for'  
  
    // can give wrong value due to  
  
    // synchronization issues.  
  
  
    for (i = 1; i > 0; i++) {  
        printf("\nEnter your choice:");
```



```
scanf("%d", &n);

// Switch Cases

switch (n) {

case 1:

    // If mutex is 1 and empty

    // is non-zero, then it is

    // possible to produce

    if ((mutex == 1)

        && (empty != 0)) {

        producer();

    }

    // Otherwise, print buffer

    // is full

    else {

        printf("Buffer is full!");

    }

    break;

case 2:

    // If mutex is 1 and full

    // is non-zero, then it is

    // possible to consume

    if ((mutex == 1)

        && (full != 0)) {

        consumer();
```



```
-----  
}  
  
    // Otherwise, print Buffer  
  
    // is empty  
  
    else {  
  
        printf("Buffer is empty!");  
  
    }  
  
    break;  
  
    // Exit Condition  
  
case 3:  
  
    exit(0);  
  
    break;  
  
}  
  
}  
  
}
```



Output:

```
1. Enter 1 for Producer
2. Enter 2 for Consumer
3. Enter 3 to Exit
Enter your choice: 1
Producer produces item number: 1

Enter your choice: 1
Producer produces item number: 2

Enter your choice: 2
Consumer consumes item number: 2.

Enter your choice: 2
Consumer consumes item number: 1.

Enter your choice: 3
```




Experiment No : 07

AIM: Implementation of Bankers algorithm

Objective: To study and implement bankers algorithms.

Theory:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that simulates resource allocation for predetermined maximum possible amounts of all resources before performing an "s-state" check to look for potential activities and determining whether allocation should be permitted to continue.

Utilizing the following data structures, the Banker's Algorithm is implemented:

Assume that there are n processes in the system and m different resource kinds.

Available:

- It is a 1-d array of size " m " that lists the total amount of resources of each type that are accessible.
- There are ' k ' instances of the resource type if $\text{Available}[j] = k$. R_j

Max:

- The maximum demand of each process in a system is specified by a 2-d array of size ' $n*m$ '.
- Process P_i may request a maximum of ' k ' instances of resource type R_j if

$\text{Max}[i, j] = k$.

Allocation:

- The number of resources of each type currently allocated to each process is specified by a 2-d array of size ' $n*m$ '.
- Process is indicated by $\text{Allocation}[i, j] = k$. P_i has been given ' k ' instances of the resource type at this time. R_j

Need:

- It is a 2-d array of size 'n*m' that shows how much more each process needs in terms of resources.
- $Need[i][j] = k$ denotes that process P_i need 'k' instances of the resource type at this time. R_j
- $Allocation[i, j] - Maximum[i, j] = Need[i, j]$

The resources that are now allotted to process P_i are identified by Allocation, while the extra resources that process P_i could yet need in order to do its work are identified by Need.

The safety algorithm **and the resource request algorithm make up the banker's algorithm.**

Algorithm:

Safety Algorithm

1) Let us assume Work and Finish be vectors of size 'm' and 'n'.

Initialize: $Work = Available$

$Finish[i] = false$; for $i=1, 2, \dots, n$

2) Find i such that

a) $Finish[i] = false$

b) $Need[i] \leq Work$

if no such i exists

goto step (4)

3) $Work = Work + Allocation[i]$

$Finish[i] = true$

goto step (2)

4) if Finish [i] = true for all i

the system is in a safe state

Resource-Request Algorithm

Let us consider a vector Request for Process P_i .

$Request_i[j] = k$ means process P_i requires k instances of Resource R_j .

1) If $Request[i] \leq Need[i]$

Goto step (2) ;

else

raise error condition as the process exceeds its maximum claim.

2) If $Request[i] \leq Available$

Goto step (3);

else wait as the resources are not available.

3) Pretend to have allocated the resources that are requested to process P_i by modifying the state as

$Available = Available - Request[i]$

$Allocation[i] = Allocation[i] + Request[i]$

$Need[i] = Need[i] - Request[i]$

Source code:

```
// Banker's Algorithm
```

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

```
int main()
```



NUTAN MAHARASHTRA VIDYA PRASARAK MANDAL'S
NUTAN COLLEGE OF ENGINEERING & RESEARCH (NCER)
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - ARTIFICIAL INTELLIGENCE



{

// P0, P1, P2, P3, P4 are the Process names here

int n, m, i, j, k;

n = 5; // Number of processes

m = 3; // Number of resources

int alloc[5][3] = {{0, 1, 0}, // P0 // Allocation Matrix

{2, 0, 0}, // P1

{3, 0, 2}, // P2

{2, 1, 1}, // P3

{0, 0, 2}}; // P4

int max[5][3] = {{7, 5, 3}, // P0 // MAX Matrix

{3, 2, 2}, // P1

{9, 0, 2}, // P2

{2, 2, 2}, // P3

{4, 3, 3}}; // P4

int avail[3] = {3, 3, 2}; // Available Resources

int f[n], ans[n], ind = 0;

for (k = 0; k < n; k++)

{

f[k] = 0;



```
}  
  
int need[n][m];  
  
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < m; j++)  
        need[i][j] = max[i][j] - alloc[i][j];  
}  
  
int y = 0;  
  
for (k = 0; k < 5; k++)  
{  
    for (i = 0; i < n; i++)  
    {  
        if (f[i] == 0)  
        {  
            int flag = 0;  
  
            for (j = 0; j < m; j++)  
            {  
                if (need[i][j] > avail[j])  
                {  
                    flag = 1;  
  
                    break;  
                }  
            }  
  
            if (flag == 0)
```



```
-----  
{  
    ans[ind++] = i;  
    for (y = 0; y < m; y++)  
        avail[y] += alloc[i][y];  
    f[i] = 1;  
}  
}  
}  
}  
int flag = 1;  
for (int i = 0; i < n; i++)  
{  
    if (f[i] == 0)  
    {  
        flag = 0;  
        printf("The following system is not safe");  
        break;  
    }  
}  
if (flag == 1)  
{  
    printf("Following is the SAFE Sequence\n");  
    for (i = 0; i < n - 1; i++)  
        printf(" P%d ->", ans[i]);
```



NUTAN MAHARASHTRA VIDYA PRASARAK MANDAL'S
NUTAN COLLEGE OF ENGINEERING & RESEARCH (NCER)
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - ARTIFICIAL INTELLIGENCE



```
-----  
    printf(" P%d", ans[n - 1]);  
  
}  
  
return (0);  
  
}
```

Output:

```
Following is the SAFE Sequence  
P1 -> P3 -> P4 -> P0 -> P2 \
```

Experiment No: 08

AIM: Implementation of various page replacement algorithms (FIFO,Optimal, LRU).

Objective: To study and implement various Page Replacement Algorithms.

Theory:

FIFO Page Replacement Algorithm:

FIFO which is also called First In First Out is one of the types of Replacement Algorithms. This algorithm is used in a situation where an Operating system replaces an existing page with the help of memory by bringing a new page from the secondary memory. FIFO is the simplest among all algorithms which are responsible for maintaining all the pages in a queue for an operating system and also keeping track of all the pages in a queue. The older pages are kept in the front and the newer ones are kept at the end of the queue. Pages that are in the front are removed first and the pages which are demanded are added.

Algorithm:

1. Start traversing the pages.
2. Now declare the size w.r.t length of the Page.
3. Check need of the replacement from the page to memory.
4. Similarly, Check the need of the replacement from the old page to new page in memory.
5. Now form the queue to hold all pages.
6. Insert Require page memory into the queue.
7. Check bad replacements and page faults.
8. Get no of processes to be inserted.
9. Show the values.
10. Stop

Source code:

```
#include<stdio.h>

int main()
{
    int incomingStream[] = {4, 1, 2, 4, 5};
```




```
int pageFaults = 0;

int frames = 3;

int m, n, s, pages;

pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");

int temp[frames];

for(m = 0; m < frames; m++)

{

    temp[m] = -1;

}

for(m = 0; m < pages; m++)

{

    s = 0;

    for(n = 0; n < frames; n++)

    {

        if(incomingStream[m] == temp[n])

        {

            s++;

            pageFaults--;

        }

    }

    pageFaults++;

    if((pageFaults <= frames) && (s == 0))

    {
```



```
-----
    temp[m] = incomingStream[m];

}

else if(s == 0)

{
    temp[(pageFaults - 1) % frames] = incomingStream[m];
}

    printf("\n");

printf("%d\t\t\t",incomingStream[m]);

for(n = 0; n < frames; n++)

{
    if(temp[n] != -1)

        printf(" %d\t\t\t", temp[n]);

    else

        printf(" - \t\t\t");

}

}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}
```

Output:

Incoming	Frame 1	Frame 2	Frame 3
4	4	-	-
1	4	1	-
2	4	1	2
4	4	1	2
5	5	1	2
Total Page Faults: 4			

LRU Page Replacement Algorithm:

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page is not found in the memory frames.

Algorithm:

Start the process

1. Declare the size
2. Get the number of pages to be inserted
3. Get the value
4. Declare counter and stack
5. Select the least recently used page by counter value
6. Stack them according to the selection.
7. Display the values
8. Stop the process

Source Code:

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

```
#include<limits.h>
```

```
int checkHit(int incomingPage, int queue[], int occupied){
```



```
for(int i = 0; i < occupied; i++){  
    if(incomingPage == queue[i])  
        return 1;  
}  
  
return 0;  
}  
  
void printFrame(int queue[], int occupied)  
{  
    for(int i = 0; i < occupied; i++)  
        printf("%d\t\t",queue[i]);  
}  
  
int main()  
{  
  
    // int incomingStream[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1};  
    // int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3, 6, 1, 2, 4, 3};  
    int incomingStream[] = {1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3};  
  
    int n = sizeof(incomingStream)/sizeof(incomingStream[0]);  
    int frames = 3;
```



```
int queue[n];

int distance[n];

int occupied = 0;

int pagefault = 0;

printf("Page\t Frame1 \t Frame2 \t Frame3\n");

for(int i = 0; i < n; i++)
{
    printf("%d: \t\t", incomingStream[i]);

    // what if currently in frame 7

    // next item that appears also 7

    // didnt write condition for HIT

    if(checkHit(incomingStream[i], queue, occupied)){
        printFrame(queue, occupied);
    }

    // filling when frame(s) is/are empty

    else if(occupied < frames){
        queue[occupied] = incomingStream[i];

        pagefault++;

        occupied++;

        printFrame(queue, occupied);
    }
}
```



```
}  
  
else{  
  
    int max = INT_MIN;  
  
    int index;  
  
    // get LRU distance for each item in frame  
    for (int j = 0; j < frames; j++)  
    {  
        distance[j] = 0;  
  
        // traverse in reverse direction to find  
        // at what distance frame item occurred last  
        for(int k = i - 1; k >= 0; k--)  
        {  
            ++distance[j];  
  
            if(queue[j] == incomingStream[k])  
                break;  
        }  
  
        // find frame item with max distance for LRU  
  
        // also notes the index of frame item in queue  
  
        // which appears furthest(max distance)  
        if(distance[j] > max){  
            max = distance[j];  
            index = j;  
        }  
    }  
}
```



```
-----
    queue[index] = incomingStream[i];

    printFrame(queue, occupied);

    pagefault++;
}

    printf("\n");
}

    printf("Page Fault: %d",pagefault);

    return 0;
}
```

Output:

Page	Frame1	Frame2	Frame3
1:	1		
2:	1	2	
3:	1	2	3
2:	1	2	3
1:	1	2	3
5:	1	2	5
2:	1	2	5
1:	1	2	5
6:	1	2	6
2:	1	2	6
5:	5	2	6
6:	5	2	6
3:	5	3	6
1:	1	3	6
3:	1	3	6
Page Fault: 8			



Experiment No: 09

AIM: Implementation of various memory allocation algorithms, (First fit, Best fit and Worst fit).

Objective: To study and implement various memory allocation algorithms.

Theory:

First Fit:

The First Fit memory allocation checks the empty memory blocks in a sequential manner. It means that the memory Block which found empty in the first attempt is checked for size. But if the size is not less than the required size then it is allocated.

Algorithm:

1. START.
2. At first get the no of processes and blocks.
3. Allocate the process by if(size of block \geq size of the process) then allocate the process else move to the next block.
4. Now Display the processes with blocks and allocate to respective process.
5. STOP.

Source code:

```
#include<stdio.h>

void main()
{
    int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

    for(i = 0; i < 10; i++)
    {
        flags[i] = 0;
        allocation[i] = -1;
    }

    printf("Enter no. of blocks: ");
```




```
scanf("%d", &bno);

printf("\nEnter size of each block: ");

for(i = 0; i < bno; i++)

    scanf("%d", &bsize[i]);

printf("\nEnter no. of processes: ");

scanf("%d", &pno);

printf("\nEnter size of each process: ");

for(i = 0; i < pno; i++)

    scanf("%d", &psize[i]);

for(i = 0; i < pno; i++)    //allocation as per first fit

    for(j = 0; j < bno; j++)

        if(flags[j] == 0 && bsize[j] >= psize[i])

            {

                allocation[j] = i;

                flags[j] = 1;

                break;

            }

//display allocation details

printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");

for(i = 0; i < bno; i++)

{

    printf("\n%d\t\t%d\t\t", i+1, bsize[i]);

    if(flags[i] == 1)

        printf("%d\t\t%d", allocation[i]+1, psize[allocation[i]]);
```

```
-----  
    else  
  
        printf("Not allocated");  
  
    }  
  
}
```

Output:

```
Enter no. of blocks: 6  
Enter size of each block: 200  
400  
600  
500  
300  
250  
Enter no. of processes: 4  
Enter size of each process: 357  
210  
468  
491  
Block no.   size   process no.   size  
1           200    Not allocated  
2           400     1           357  
3           600     2           210  
4           500     3           468  
5           300    Not allocated  
6           250    Not allocated
```

Worst Fit:

Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

Algorithm

1. Input memory blocks and processes with sizes.
2. Initialize all memory blocks as free.
3. Start by picking each process and find the maximum block size that can be assigned



to current process i.e., find $\max(\text{blockSize}[1], \text{blockSize}[2], \dots, \text{blockSize}[n])$
>

processSize[current], if found then assign it to the current process.

4. If not then leave that process and keep checking the further processes.

Source code:

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

```
void implimentWorstFit(int blockSize[], int blocks, int processSize[], int processes)
```

```
{
```

```
    // This will store the block id of the allocated block to a process
```

```
    int allocation[processes];
```

```
    int occupied[blocks];
```

```
    // initially assigning -1 to all allocation indexes
```

```
    // means nothing is allocated currently
```

```
    for(int i = 0; i < processes; i++){
```

```
        allocation[i] = -1;
```

```
    }
```

```
    for(int i = 0; i < blocks; i++){
```

```
        occupied[i] = 0;
```

```
    }
```

```
    // pick each process and find suitable blocks
```

```
    // according to its size and assign to it
```

```
    for (int i=0; i < processes; i++)
```

```
    {
```



```
-----
int indexPlaced = -1;

for(int j = 0; j < blocks; j++)
{
    // if not occupied and block size is large enough
    if(blockSize[j] >= processSize[i] && !occupied[j])
    {
        // place it at the first block fit to accomodate process
        if (indexPlaced == -1)
            indexPlaced = j;

        // if any future block is larger than the current block where
        // process is placed, change the block and thus indexPlaced
        else if (blockSize[indexPlaced] < blockSize[j])
            indexPlaced = j;
    }
}

// If we were successfully able to find block for the process
if (indexPlaced != -1)
{
    // allocate this block j to process p[i]
    allocation[i] = indexPlaced;

    // make the status of the block as occupied
    occupied[indexPlaced] = 1;
```



```
// Reduce available memory for the block

blockSize[indexPlaced] -= processSize[i];

}

}

printf("\nProcess No.\tProcess Size\tBlock no.\n");

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

{

    printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);

    if (allocation[i] != -1)

        printf("%d\n", allocation[i] + 1);

    else

        printf("Not Allocated\n");

}

}

// Driver code

int main()

{

    int blockSize[] = { 100, 50, 30, 120, 35 };

    int processSize[] = { 40, 10, 30, 60 };

    int blocks = sizeof(blockSize)/sizeof(blockSize[0]);

    int processes = sizeof(processSize)/sizeof(processSize[0]);
```



```
-----  
implimentWorstFit(blockSize, blocks, processSize, processes);  
  
return 0;  
  
}
```

Output:

Process No.	Process Size	Block no.
1	40	4
2	10	1
3	30	2
4	60	Not Allocated

Best Fit:

Best fit uses the best memory block based on the Process memory request. In best fit implementation the algorithm first selects the smallest block which can adequately fulfill the memory request by the respective process.

Algorithm:

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the above definition.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

Source code:

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

```
void implimentBestFit(int blockSize[], int blocks, int processSize[], int processes)
```



```
{  
  
    // This will store the block id of the allocated block to a process  
  
    int allocation[processes];  
  
  
    // initially assigning -1 to all allocation indexes  
  
    // means nothing is allocated currently  
  
    for(int i = 0; i < processes; i++){  
        allocation[i] = -1;  
    }  
  
  
    // pick each process and find suitable blocks  
  
    // according to its size and assign to it  
  
    for (int i=0; i<processes; i++)  
    {  
  
        int indexPlaced = -1;  
  
        for (int j=0; j<blocks; j++)  
        {  
            if (blockSize[j] >= processSize[i])  
            {  
                // place it at the first block fit to accommodate process  
  
                if (indexPlaced == -1)  
                    indexPlaced = j;  
  
                // if any future block is better than this
```



```
-----  
    // any future block with smaller size encountered  
  
    // that can accomodate the given process  
    else if (blockSize[j] < blockSize[indexPlaced])  
        indexPlaced = j;  
  
    }  
  
}  
  
// If we were successfully able to find block for the process  
if (indexPlaced != -1)  
{  
    // allocate this block j to process p[i]  
    allocation[i] = indexPlaced;  
  
    // Reduce available memory for the block  
    blockSize[indexPlaced] -= processSize[i];  
  
}  
  
}  
  
printf("\nProcess No.\tProcess Size\tBlock no.\n");  
for (int i = 0; i < processes; i++)  
{  
    printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);  
  
    if (allocation[i] != -1)  
        printf("%d\n", allocation[i] + 1);  
    else  
        printf("Not Allocated\n");  
}  
}
```




NUTAN MAHARASHTRA VIDYA PRASARAK MANDAL'S
NUTAN COLLEGE OF ENGINEERING & RESEARCH (NCER)
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - ARTIFICIAL INTELLIGENCE



// Driver code

```
int main()
{
    int blockSize[] = {50, 20, 100, 90};
    int processSize[] = {10, 30, 60, 30};
    int blocks = sizeof(blockSize)/sizeof(blockSize[0]);
    int processes = sizeof(processSize)/sizeof(processSize[0]);
    implimentBestFit(blockSize, blocks, processSize, processes);
    return 0 ;
}
```

Output:

Process No.	Process Size	Block no.
1	10	2
2	30	1
3	60	4
4	30	4

Experiment No : 10

AIM: Implementation of various Disk Scheduling Algorithms (FCFS,SCAN)

Objective: To study and implement various disk scheduling algorithms.

Theory:

FCFS:

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

Source code:

```
#include<math.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

    int i,n,req[50],mov=0,cp;

    printf("enter the current position\n");

    scanf("%d",&cp);

    printf("enter the number of requests\n");

    scanf("%d",&n);

    printf("enter the request order\n");
```

```
-----  
for(i=0;i<n;i++)  
{  
    scanf("%d",&req[i]);  
}  
mov=mov+abs(cp-req[0]); // abs is used to calculate the absolute value  
printf("%d -> %d",cp,req[0]);  
for(i=1;i<n;i++)  
{  
    mov=mov+abs(req[i]-req[i-1]);  
    printf(" -> %d",req[i]);  
}  
printf("\n");  
printf("total head movement = %d\n",mov);  
}
```

Output:

```
enter the current position  
45  
enter the number of requests  
3  
enter the request order  
12  
45  
45  
45 -> 12 -> 45 -> 45  
total head movement = 66
```

SCAN:

It is also called as Elevator Algorithm. In this algorithm, the disk arm moves into a

particular direction till the end, satisfying all the requests coming in its path, and then it turns back and moves in the reverse direction satisfying requests coming in its path. It works in the way an elevator works, elevator moves in a direction completely till the last floor of that direction and then turns back.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

Source code:

```
#include <stdio.h>

#include <math.h>

int main()
{
    int queue[20], n, head, i, j, k, seek = 0, max, diff, temp, queue1[20],
    queue2[20], temp1 = 0, temp2 = 0;

    float avg;

    printf("Enter the max range of disk\n");

    scanf("%d", &max);

    printf("Enter the initial head position\n");

    scanf("%d", &head);

    printf("Enter the size of queue request\n");
```

```
scanf("%d", &n);

printf("Enter the queue of disk positions to be read\n");

for (i = 1; i <= n; i++)
{
    scanf("%d", &temp);

    if (temp >= head)
    {
        queue1[temp1] = temp;
        temp1++;
    }
    else
    {
        queue2[temp2] = temp;
        temp2++;
    }
}

for (i = 0; i < temp1 - 1; i++)
{
    for (j = i + 1; j < temp1; j++)
    {
        if (queue1[i] > queue1[j])
        {
            temp = queue1[i];
            queue1[i] = queue1[j];
```

```
-----
    queue1[j] = temp;

    }

    }

    }

    for (i = 0; i < temp2 - 1; i++)
    {
        for (j = i + 1; j < temp2; j++)
        {
            if (queue2[i] < queue2[j])
            {
                temp = queue2[i];
                queue2[i] = queue2[j];
                queue2[j] = temp;
            }
        }
    }

    for (i = 1, j = 0; j < temp1; i++, j++)
        queue[i] = queue1[j];

    queue[i] = max;

    for (i = temp1 + 2, j = 0; j < temp2; i++, j++)
        queue[i] = queue2[j];

    queue[i] = 0;

    queue[0] = head;

    for (j = 0; j <= n + 1; j++)
    {
        diff = abs(queue[j + 1] - queue[j]);
```



```
-----
seek += diff;

printf("Disk head moves from %d to %d with seek %d\n", queue[j],
queue[j + 1], diff);
}

printf("Total seek time is %d\n", seek);

avg = seek / (float)n;

printf("Average seek time is %f\n", avg);

return 0;

}
```

Output:

```
Enter the max range of disk
199
Enter the initial head position
50
Enter the size of queue request
7
Enter the queue of disk positions to be read
82
23
65
96
45
45
25
```