



LABWORKBOOK

18CS2102 OPERATING SYSTEMS

II B.TECH CSE 2019-20 ODD SEMESTER
KLUNIVERSITY|OPERATINGSYSTEMS-18CS2102



LABORATORY WORKBOOK

STUDENT NAME	
REG. NO	
YEAR	
SEMESTER	
SECTION	
FACULTY	

A handwritten signature in black ink, appearing to read 'J. V. G. C.', is located above the printed name of the Head of Department.

HOD-CSE
K L UNIVERSITY

Organization of the STUDENT LAB WORKBOOK

The lab framework comprises a creative element but shifts the time-intensive aspects outside of the Four-Hour closed lab period(including skilling part). Within this structure, each lab includes **THREE** parts: Pre-lab, In-Lab and Skilling

a. Pre-Lab

The Prelab exercise is a homework assignment that links the lecture with the lab period - typically takes 2 hours to complete. The goal is to make students familiar with the basic topics which make in-lab easy for them. Students attending a two-hour closed lab are expected to make a good-faith effort to complete the Prelab exercise before coming to the lab. Their work need not be perfect, but their effort must be real.

b. In-Lab

The In-lab section takes place during the actual lab period. The First hour of the lab period can be used to resolve any problems the students might have experienced in completing the Prelab exercises. The intention is to give productive feedback so that students leave the lab with working Prelab software - a significant achievement on their part. During the second hour, students complete the In-lab exercise to strengthen the concepts learned in the Prelab.

c. Skilling

The last phase of each lab is a homework assignment that is done following the lab session. In Skilling, students analyze the efficiency or adequacy of given a system call. Each Skilling exercise will take 2 hours to complete.

2019-20 EVEN SEMESTER LAB CONTINUOUS EVALUATION

[illegible]

WEEK	TOPIC
1	BASIC COMMAND WITH EXAMPLES
2	FILE OPERATION
3	PROCESS API
4	PROCESS SCHEDULING & XV6
5	MEMORY API
6	MEMORY (PAGING , SEGMENTATION)
7	THREAD API
8	CONCURRENCY
9	DIRECTORY AND FILES
10	DISK SCHEDULING ALGORITHMS AND SOCKETS
11	XV6 RELATED
12	XV6 RELATED

CO-1

WEEK - 1 (BASIC COMMANDS WITH EXAMPLES)

Prerequisite:

- General idea of what an OS is (an interface between User and Hardware)
- How users communicate with the hardware? (using commands)
- What is a Shell?
- How commands can be executed (through Shell)?

Pre Lab Task:

1. What is the purpose of “whatis” command? Verify the functionality and learn the usage of different commands in UNIX operating system using “man” command.

Using “whatis” command, write down the output for the following commands

BASIC COMMANDS	FUNCTIONALITY
man	\$ whatis man man(1) - an interface to the on-line reference manuals man(7) - macros to format man pages
who	who(1) - show who is logged on

pwd	pwd(1) - print name of current/working directory
mkdir	mkdir(1) - make directories mkdir(2) - create a directory
cat	cat(1) - concatenate files and print on the standard output
touch	touch(1) - change file timestamps
nano	nano(1) - nano's another editor, an enhanced free pico clone
tar	tar(1) - an archiving utility
mv	mv(1) - move (rename) files
sort	sort(1) - sort lines of text files

ls	ls(1) - list directory contents
chmod	chmod(1) - change file mode bits chmod(2) - change permissions of a file
head	head(1) - output the first part of files HEAD(1p) - simple command line user agent
date	date(1) - print or set the system date and time
ping	ping(2) - send ICMP ECHO_REQUEST to network hosts
echo	echo(1) - display a line of text
cal	cal(1) - display a calendar and the date of Easter

grep	grep(1) - print lines matching a pattern
------	---

In Lab Task:

1.Read the problem description and write all necessary commands/code as elaborated below. Use space provided in the next page if required.

Problem Description

Stan lee wants to get started with terminal commands in Linux. Help him out to perform the following set of statements:

a .He wants to know his current directory that he is working with, in the system. After identifying the current directory, he desires to create a folder on his Desktop called Marvel.

Ans:

a) vaibhav@Gryffindor:~\$ pwd

/home/vaibhav

vaibhav@Gryffindor:~\$ cd Desktop/

vaibhav@Gryffindor:~/Desktop\$ mkdir marvel

b.Now, he wants to list out all the Avengers of the “Marvel” universe. He adds the following set of Avengers to Avengers.txt:

- I. Ironman
- Ii.Captain america
- iii.Thor
- iv.Hulk
- V.Black widow

Ans:

```
vaibhav@Gryffindor:~/Desktop$ cd marvel
```

```
vaibhav@Gryffindor:~/Desktop/marvel$ cat > avengers.txt
```

```
iron man
```

```
captain america
```

```
thor
```

```
hulk
```

```
black widow
```

```
^Z
```

```
[1]+  Stopped                  cat > avengers.txt
```

c. After adding the names displayed above check whether the names are inserted or not.

```
Ans: vaibhav@Gryffindor:~/Desktop/marvel$ cat avengers.txt
```

```
iron man
```

```
captain america
```

```
thor
```

```
hulk
```

```
black widow
```

d. Stan lee wants to relocate the file (Avengers.txt) from Marvel to Desktop, after relocating the file give all the permissions to user and group and give only read permission to others. Verify the permissions when done.

Ans:

```
vaibhav@Gryffindor:~/Desktop/marvel$ mv avengers.txt ~/Desktop/
```

```
vaibhav@Gryffindor:~/Desktop/marvel$ cd ..
```

```
vaibhav@Gryffindor:~/Desktop$ ls
```

```
a.out avengers.txt create.txt id.c marvel
```

```
vaibhav@Gryffindor:~/Desktop$ chmod 774 avengers.txt
```

```
vaibhav@Gryffindor:~/Desktop$ ls -l avengers.txt
```

```
-rwxrwxr-- 1 vaibhav Gryffindor 47 Jun 18 10:10 avengers.txt
```

e. Stan lee now wants to add more avengers to the Marvel , Add the following set of new avengers to Avengers.txt .

1. Black panther
2. Groot
3. Captain marvel
4. Spiderman

Ans: **vaibhav@Gryffindor:~/Desktop\$ nano avengers.txt**

iron man

captain america

thor

hulk

black widow

black panther

groot

captain marvel

spider man

**** (press ctrl+x and press y(yes) and press enter to come out from nano command editor)****

f. Sort the names of the file in lexicographical order and export the result to Sortedavengers.txt and display the content of it.

Ans: **vaibhav@Gryffindor:~/Desktop\$ sort avengers.txt > sortedavengers.txt**

vaibhav@Gryffindor:~/Desktop\$ cat sortedavengers.txt

black panther

black widow

captain america

captain marvel

groot

hulk

iron man

spider man

thor

g. Now, Stanlee sends the first avenger from Sortedavengers.txt to visit Wakanda(another directory inside marvel directory) as a part of mission to kill thanos.

Ans: **vaibhav@Gryffindor:~/Desktop\$ head -1 sortedavengers.txt > wakanda.txt**

vaibhav@Gryffindor:~/Desktop\$ cat wakanda.txt

Black panther

vaibhav@Gryffindor:~/Desktop\$ mv wakanda.txt ~/Desktop/marvel

vaibhav@Gryffindor:~/Desktop\$ cd marvel

vaibhav@Gryffindor:~/Desktop/marvel\$ ls

mission.log wakanda.txt

vaibhav@Gryffindor:~/Desktop/marvel\$ cat wakada.txt

Black panther

h. To know the status of the mission to kill Thanos , Stan lee pings the Avengers to the address 103.102.166.224 with 3 packets of data.

Ans:

```
vaibhav@Gryffindor:~/Desktop$ ping 103.102.166.224 -c 3
```

```
PING 103.102.166.224 (103.102.166.224) 56(84) bytes of data.
```

```
64 bytes from 103.102.166.224: icmp_seq=1 ttl=51 time=160 ms
```

```
64 bytes from 103.102.166.224: icmp_seq=2 ttl=51 time=163 ms
```

```
64 bytes from 103.102.166.224: icmp_seq=3 ttl=51 time=158 ms
```

```
--- 103.102.166.224 ping statistics ---
```

```
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
```

```
rtt min/avg/max/mdev = 158.269/160.649/163.479/2.175 ms
```

I.If pinging becomes successful then, the date is to be echoed to mission.log which is to be included in the Marvel directory which he created earlier.

,

Ans:

```
vaibhav@Gryffindor:~/Desktop$ date > mission.log
```

```
vaibhav@Gryffindor:~/Desktop$ mv mission.log ~/Desktop/marvel
```

```
vaibhav@Gryffindor:~/Desktop$ cd marvel/
```

```
vaibhav@Gryffindor:~/Desktop/marvel$ ls
```

```
mission.log
```

```
vaibhav@Gryffindor:~/Desktop/marvel$ cat mission.log
```

```
Tue Jun 18 10:15:45 IST 2019
```

2. Write commands/code in the space provided for each of the questions below:

Steps of the Program:

1. Open the Terminal.
2. Then, you can see the logged in user with the \$ symbol next to the username.
3. \$ means you are logged in as a regular user and the # means you are root user.

A. Count the number of users who logged in and display the result.

Ans. **praneeth@praneeth:~\$ who -q**

Praneeth

users=1

B. Count the number of files in the current directory.

Ans. **praneeth@praneeth:~\$ ls | wc -l**

23

C. Count the number of differences between two files Hint: cmp

Ans.- **praneeth@praneeth:~\$ cd Desktop**

praneeth@praneeth:~/Desktop\$ cmp avengers.txt sortedavengers.txt

Avengers.txt sortedavengers.txt differ: byte 1, line 1

D. Display the file /etc/passwd from the 45th line using the more command and start the vi editor on the current file and current line. Hint: Press 'h' or '?' Key While viewing the file with more command to get the help.

Ans. **praneeth@praneeth:~/Desktop\$ mkdir folder**

praneeth@praneeth:~/Desktop\$ cd folder

praneeth@praneeth:~/Desktop/folder\$ vim newfile

****/(NOW IT WILL OPEN A VI EDITOR IN THAT EDITOR GIVE SOME TEXT MORE THAN 50 LINES AND TO EXIT FROM VI EDITOR FOLLOW GIVEN STEPS**

- Press “ESC” button
- Now press “:”(colon button)
- Now type “wq” and press “enter”

praneeth@praneeth:~/Desktop/folder\$ more +45 newfile

**These
Are
Last five
Lines
From newfile**

E.Display your Birth Month and Year calendar.

Ans **praneeth@praneeth:~\$ cal 10/2000**

```

      October 2000
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 30 31
```

F. Which options are required for “who” command to display the column headers and more details list?

Ans:**praneeth@praneeth:~\$ who -H**

NAME	LINE	TIME	COMMENT
praneeth	:0	2019-06-18 10:04	(:0)

SKILL EXERCISE 1

BASIC COMMANDS	FUNCTIONALITY
uname	uname (1) - print system information uname (2) - get name and information about current kernel
mount	mount (2) - mount filesystem mount (8) - mount a filesystem
umount	umount (2) - unmount filesystem umount (8) - unmount file systems
more	more (1) - file perusal filter for crt viewing
less	less (1) - opposite of more
diff	diff (1) - compare files line by line

ln	ln (1) - make links between files
rm	rm (1) - remove files or directories
cp	cp (1) - copy files and directories
rmdir	rmdir (1) - remove empty directories rmdir (2) - delete a directory
gzip	gzip (1) - compress or expand files
find	find (1) - search for files in a directory hierarchy
history	history (3readline) - GNU History Library
telnet	telnet (1) - user interface to the TELNET protocol

nslookup	nslookup (1) - query Internet name servers interactively
df	df (1) - report file system disk space usage
du	du (1) - estimate file space usage
free	free (1) - Display amount of free and used memory in the system free (3) - allocate and free dynamic memory
top	top (1) - display Linux processes
ps	ps (1) - report a snapshot of the current processes.
kill	kill (1) - send a signal to a process kill (2) - send signal to a process

2. Write commands/code in the space provided for each of the questions below:

Steps of the Program:

Open the Terminal.

Then, you can see the logged in user with the \$ symbol next to the username.

\$ means you are logged in as a regular user and the # means you are root user.

A.How do you know your machine's name?

Ans.

```
a) praneeth@praneeth:~$ hostname  
praneeth
```

B.How do you spell check a file and display the mistakes? (Hint: ispell)

```
Ans.praneeth@praneeth:~$ cd Desktop  
praneeth@praneeth:~/Desktop$ ispell avengers.txt
```

```
iron man  
captain america
```

```
0: America  
1: American  
2: Americas  
3: am Erica  
4: am-Erica
```

C.Calculate the sum and difference of two numbers using expr.

Ans

```
vaibhav@Gryffindor:~$ expr 25 + 35  
60  
vaibhav@Gryffindor:~$ expr 25 - 35  
-10
```

D.Merge two sorted files (file1=UID'S, file2=GID'S)

Ans

```
praneeth@praneeth:~$ awk -F":" '{print $3}' /etc/passwd |sort -n | cat >file1
```

```
praneeth@praneeth:~$ cat file1
```

```
0
1
2
3
4
5
6
7
8
9
10
13
33
34
38
39
41
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
```

120
121
1000
1001
65534

```
praneeth@praneeth:~$ awk -F":" '{print $3}' /etc/group |sort -n | cat >file2  
praneeth@praneeth:~$ cat file2
```

0
1
2
3
4
5
6
7
8
9
10
12
13
15
20
21
22
24
25
26
27
29
30
33
34
37
38
39
40
41
42
43
44
45
46

50
60
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
1000
1001
65534

praneeth@praneeth:~\$ sort -mn file1 file2

0
0
1
1
2
2
3
3
4

4
5
5
6
6
7
7
8
8
9
9
10
10
12
13
13
15
20
21
22
24
25
26
27
29
30
33
33
34
34
37
38
38
39
39
40
41
41
42
43
44
45
46

50
60
100
100
101
101
102
102
103
103
104
104
105
105
106
106
107
107
108
108
109
109
110
110
111
111
112
112
113
113
114
114
115
115
116
116
117
117
118
118
119
119
120

120
121
121
122
123
124
125
126
1000
1000
1001
1001
65534
65534

E. Perform sort on the 2 and 4 fields (Create a new file name INPUT with Tab as a Delimiter and having repetition of data in the fields)

```
vaibhav@Gryffindor:~$ cat > input
yellow mango potato lavender
red apple tomato rose
pink kiwi carrot lotus
black chikoo radish marigold
green grape beans lilly
white papaya capsicum jasmine
^Z
[34]+ Stopped          cat > input
vaibhav@Gryffindor:~$ sort -k 2,4 input
red apple tomato rose
black chikoo radish marigold
green grape beans lilly
pink kiwi carrot lotus
yellow mango potato lavender
white papaya capsicum jasmine
```

F. Write a Grep command to search for a five-letter word, whose first letter is a 's' and last letter 'd'

```
vaibhav@Gryffindor:~$ cat > search
```

```
scold
```

```
stupid
```

```
ssfhd
```

```
stud
```

```
curd
```

```
^Z
```

```
[14]+ Stopped          cat > search
```

```
vaibhav@Gryffindor:~$ grep "s\w\w\wd" search
```

```
scold
```

G. Multiple jobs can be issued from the same command line using the operators: && and ||. Try combining the commands, cat nonexistent and echo hello using each of these operators. Reverse the order of the commands and try again. What are the rules about when the commands will be executed?

If we put a command on each side, the command on the right of a && will be run only if the command on the left is successful. The command on the right of a || will be run only if the command on the left fails.

The reason for this behavior is tied to the Boolean nature of the operators. If a command runs successfully, the result is evaluated as 'true'. If a command fails, the result is evaluated as 'false'.

```
vaibhav@Gryffindor:~$ cat && echo hello
```

```
i love basketball
```

```
i love basketball
```

```
^Z
```

```
[16]+ Stopped          cat
```

```
vaibhav@Gryffindor:~$ echo hello && cat
```

```
hello
```

```
i love basketball
```

```
i love basketball
```

```
^Z
```

```
[17]+ Stopped          cat
```

```
vaibhav@Gryffindor:~$ cat || echo hello
```

```
i love basketball
```

```
i love basketball
```

^Z

[18]+ Stopped cat

hello

vaibhav@Gryffindor:~\$ echo hello || cat

hello

WEEK-2 (FILE OPERATION)

Prerequisite:

- Creation of a new file (fopen with attributes as “a” or “a+” or “w” or “w++”)
- Opening an existing file (fopen) and Reading from a file (fscanf, fgets or fgetc)
- Writing to a file int fputc, fputs
- Moving to a specific location in a file (fseek, rewind) and Closing a file (fclose)
- Basic system calls on files.

Pre lab task:

1. Using the format “\$man 3 Library call name”, Learn the following Standard I/O Library functions and write the function prototypes.

Type \$man 3 fopen on Linux machine, to get function prototypes for all the library functions

a. The fopen function opens a specified file.

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

```
FILE *fopen(const char *restrict pathname, const char *restrict type);
```

b. The standard I/O function fopen returns a pointer to a FILE object.

Now write the functionality of file handling library functions in c given in the below:

SL NO	FUNCTIONS	FUNCTIONALITY/PROTOTYPE
1	fopen()	<p>a) vaibhav@Gryffindor:~\$ man 3 fopen</p> <p>FILE *fopen(const char *pathname, const char *mode);</p> <p>The fopen() function opens the file whose name is the string pointed to by pathname and associates a stream with it.</p> <p>Upon successful completion, returns a FILE pointer.</p> <p>Otherwise, NULL is returned and errno is set to indicate the error.</p>
2	fclose()	<p>int fclose(FILE *stream);</p> <p>The fclose() function flushes the stream pointed to by the given stream and closes the underlying file descriptor.</p>

		<p>Upon successful completion, 0 is returned. Otherwise, EOF is returned and errno is set to indicate the error. In either case, any further access (including another call to fclose()) to the stream results in undefined behavior.</p>
3	getc()	<p>int getc(FILE *stream);</p> <p>The getc() reads the next character from the given stream and returns it as an unsigned char cast to an int, or EOF on end of file or error.</p>
4	putc()	<p>int putc(int c, FILE *stream);</p> <p>The putc() writes the given character , cast to an unsigned char, to the given stream. Returns the character written as an unsigned char cast to an int or EOF on error.</p>
5	fscanf()	<p>int fscanf(FILE *stream, const char *format, ...);</p> <p>fscanf() reads input from the given stream</p>
6	fprintf()	<p>int fprintf(FILE *stream, const char *format, ...);</p> <p>fprintf() write output to the given output stream</p>
7	gets()	<p>char * gets (char * str); gets() reads a line from stdin into the buffer pointed to by str until either a terminating newline or EOF, which it replaces with a null byte (&#39;\0&#39;).</p>
8	puts()	<p>int puts(const char *s); puts() writes the string s and a trailing newline to stdout.</p>
9	fseek()	<p>int fseek(FILE *stream, long offset, int whence);</p> <p>The fseek() function sets the file position indicator for the stream pointed to by the given stream.</p>
10	ftell()	<p>long ftell(FILE *stream);</p>

		The ftell() function obtains the current value of the file position indicator for the stream pointed to by the given stream.
11	rewind()	void rewind(FILE *stream); The rewind() function sets the file position indicator for the stream pointed to by stream to the beginning of the file.

2. Using the format “\$man 2 system call name”, write the prototypes for the following system calls

a. open()

```
vaibhav@Gryffindor:~$ man 2 open
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

b. read()

```
vaibhav@Gryffindor:~$ man 2 read
ssize_t read(int fd, void *buf, size_t count);
```

c. write()

```
vaibhav@Gryffindor:~$ man 2 write
ssize_t write(int fd, const void *buf, size_t count);
```

d. lseek()

```
vaibhav@Gryffindor:~$ man 2 lseek
off_t lseek(int fd, off_t offset, int whence);
```

e. creat()

```
vaibhav@Gryffindor:~$ man 2 creat
int creat(const char *pathname, mode_t mode);
```

- f. `close()`
vaibhav@Gryffindor:~\$ man 2 close
int close(int fd);
- g. `unlink()`
vaibhav@Gryffindor:~\$ man 2 unlink
int unlink(const char *pathname);

In lab task

1. Write a C program that reads file.txt line by line and prints the first 10-digit number in the given file (digits should be continuous), If not found then print the first 10 characters excluding numbers.

```
#include <stdio.h>

int main()
{
    //Reading the file
    FILE *fp = fopen("file.txt" , "r");
    int count;
    int c=0;
    while(1)
    {
        count = 0;
        for(int i=0;i<10;i++){
            c = getc(fp);
            if( c == -1)
                break;
            if(c>='0' && c<='9')
                count++;
        }
        if(c == -1)
            break;
        if(count==10){
            char s[10];
            fseek(fp,-10,SEEK_CUR);
            fgets(s,11,fp);
            printf("%s",s);
            break;
        }
        fseek(fp,-9,SEEK_CUR);
    }
    if (count!=10)
    {
```

```

        rewind(fp);
        for(int i=0;i<10;)
        {
            c = getc(fp);
            if(c>='0' && c<='9')
                continue;
            printf("%c",c);
            i++;
        }
    }
    return 0;
}

```

2. Write a C program that saves 10 random numbers to a file, using own "rand.h" header file which contains your own random () function.

```

#include<stdlib.h>
int ownrand()
{
    return rand()%10;
}

```

(To save it, press esc and type :wq)

vaibhav@Gryffindor:~\$ vi random.c

```

#include<stdio.h>
#include<time.h>
#include"rand.h"
int main()
{
    srand(time(0));
    FILE *fptr;
    fptr = fopen("random.txt","w");
    int i;
    for(i=0;i<10;i++)
    {
        fprintf(fptr,"%d",ownrand());
        fprintf(fptr,"\n");
    }
    return 0;
}

```



```
}
```

```
vaibhav@Gryffindor:~$ gcc random.c
vaibhav@Gryffindor:~$ ./a.out
vaibhav@Gryffindor:~$ cat random.txt
8
3
7
0
2
8
4
2
4
3
```

3. Write a UNIX system program that creates a file with a hole in it.

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>

Char buf1[ ] = "abcdefghij";
Char buf2[ ] = "ABCDEFGHJIJ";

Int main(void)
{
    Int fd;
    if((fd=creat("file.hole",777))<0)
        perror("creat error");
    if(write(fd,buf1,10)!=10)
        perror("buf1 write error");
    if(lseek(fd,16384,SEEK_SET)==-1)
        perror("lseek error");
    if(write(fd,buf2,10)!=10)
        perror("buf2 write error");
    exit(0);
}
```

SKILL EXERCISE 2

1. Write commands/code in the space provided for each of the questions below:

a. Use the cut command on the output of a long directory listing in order to display only the file permissions. Then pipe this output to sort and unique to filter out any double lines. Then use the wc to count the different permission types in this directory.

Ans.

```
vaibhav@Gryffindor:~$ ls -l | grep ^[d-] | cut -c -10 | sort -u | wc -l  
6
```

b. Try ln -s /etc/passwd passwords and check with ls-l. Did you find anything extra?

Ans.

```
vaibhav@Gryffindor:~$ ls -l  
vaibhav@Gryffindor:~$ ls -s /etc/passwd  
4 /etc/passwd  
vaibhav@Gryffindor:~$ ls -l
```

After execution of command 'ln -s /etc/passwd' a symbolic link is created to a title of path name "/etc.passwd"

c. Create a new Directory LABTEMP and Copy the files from /var/log into it and display the files whose first alphabet is consonant that do not begin with uppercase letters that has an extension of exactly three characters.

Ans.

```
vaibhav@Gryffindor:~$ mkdir LABTEMP  
vaibhav@Gryffindor:~$ cp /var/log/* LABTEMP  
cp: -r not specified; omitting directory /var/log/apt;  
cp: cannot open /var/log/boot.log; for reading: Permission denied  
cp: cannot open /var/log/btmp; for reading: Permission denied  
cp: -r not specified; omitting directory /var/log/cups;  
cp: -r not specified; omitting directory /var/log/dist-upgrade;  
cp: -r not specified; omitting directory /var/log/gdm3;  
cp: -r not specified; omitting directory /var/log/hp;  
cp: -r not specified; omitting directory /var/log/installer;
```

```

cp: -r not specified; omitting directory &#39;/var/log/journal&#39;
cp: -r not specified; omitting directory &#39;/var/log/openvpn&#39;
cp: -r not specified; omitting directory &#39;/var/log/private&#39;
cp: -r not specified; omitting directory &#39;/var/log/speech-dispatcher&#39;
cp: cannot open &#39;/var/log/tallylog&#39; for reading: Permission denied
cp: -r not specified; omitting directory &#39;/var/log/unattended-upgrades&#39;
vaibhav@Gryffindor:~$ cd LABTEMP
vaibhav@Gryffindor:~/LABTEMP$ ls | grep
&quot;\&lt;[bcdfghjklmnpqrstvwxyz]&quot;;|grep &quot;\&gt;\.lw\w\w&quot;
alternatives.log
apport.log
apport.log.1
apport.log.2.gz
auth.log
auth.log.1
bootstrap.log
dpkg.log
fontconfig.log
gpu-manager.log
kern.log
kern.log.1

```

d.Find how many hours has the system been running?

Ans.

```

vaibhav@Gryffindor:~$ uptime -p
up 5 days, 21 hours, 28 minutes

```

e.What command can be used to display the current memory usage?

Ans.

The “free” command is used to display the current memory usage.

```

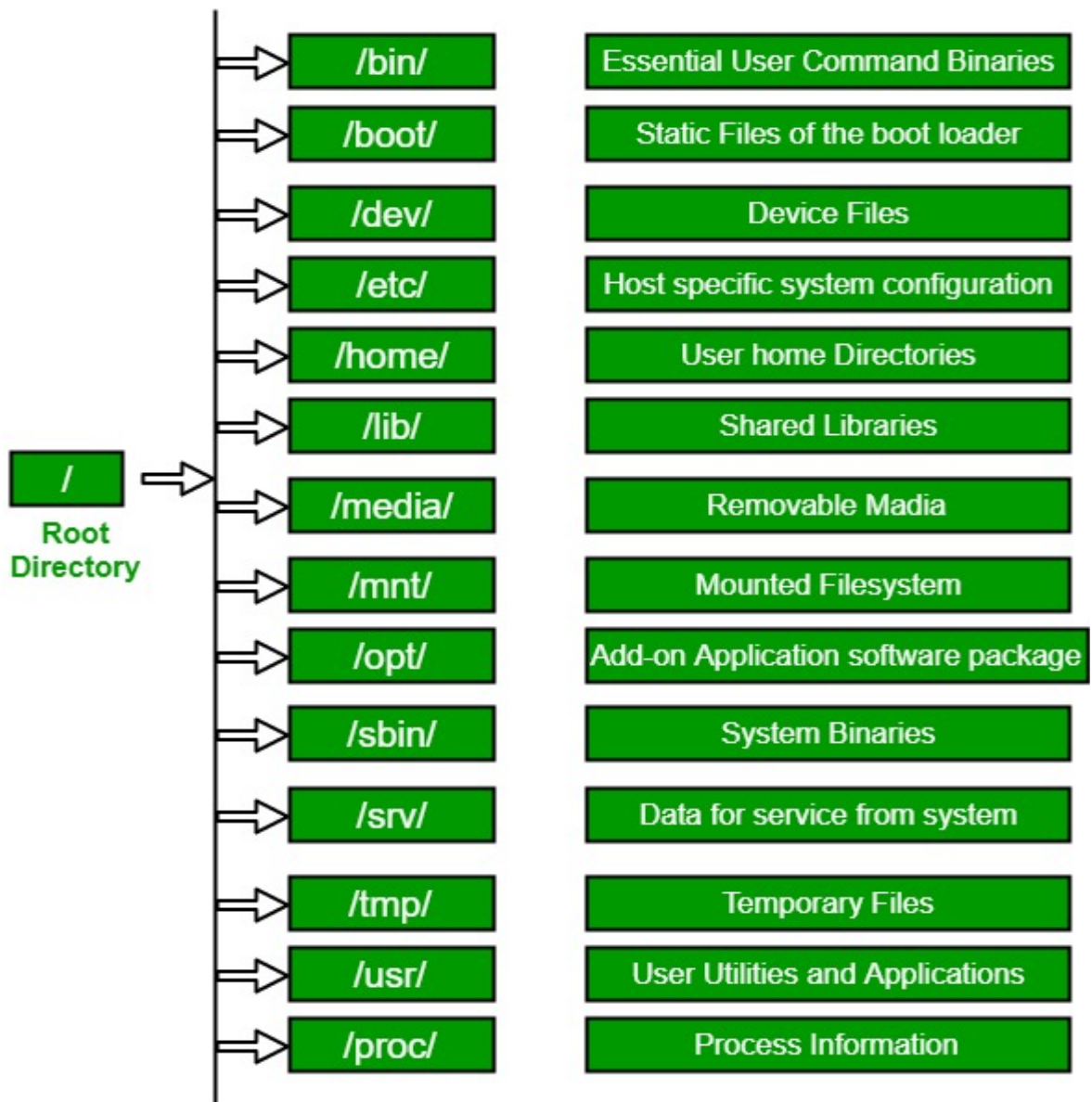
vaibhav@Gryffindor:~$ free

```

	total	used	free	shared	buff/cache	available
Mem:	12464612	9487368	2747892	17720	229352	2843512
Swap:	37748736	1352492	36396244			

f. Represent UNIX/LINUX File Hierarchy

Ans



g.Create a file using cat and find the number of lines, words and characters in it.

Ans.

```
vaibhav@Gryffindor:~$ cat > klu  
cse  
ece  
ecse  
eee
```

```
^Z
[23]+ Stopped          cat > klu
vaibhav@Gryffindor:~$ wc klu
4 4 17 klu
```

Perform the Following

h.How do you compare files? (Hint: cmp)

Ans.

cmp command in Linux/UNIX is used to compare the two files byte by byte and helps you to find out whether the two files are identical or not.

When cmp is used for comparison between two files, it reports the location of the first mismatch to the screen if difference is found and if no difference is found i.e the files compared are identical.

cmp displays no message and simply returns the prompt if the files compared are identical.

```
vaibhav@Gryffindor:~$ cat > klu
cse
ece
ecse
eee
^Z
[23]+ Stopped          cat > klu
vaibhav@Gryffindor:~$ cat > klef
bba
llb
b.com
mba
m.com
^Z
[24]+ Stopped          cat > klef
vaibhav@Gryffindor:~$ cmp klu klef
klu klef differ: byte 1, line 1
```

i. How do you display \ using echo command?

Ans. vaibhav@Gryffindor:~\$ echo vai\bhav
vai\bhav
vaibhav@Gryffindor:~\$ echo vai\\bhav
vai\bhav

j. How do you split a file into multiple files? (Hint: split)

Ans.

```
vaibhav@Gryffindor:~$ split -l 1 klu klef
vaibhav@Gryffindor:~$ ls
1.c  cat  file.hole  klef  myfile.c  output  vai.c
ERROR  compressed.tar.gz  file1  klefaa  myfile.txt  output.log  vai.txt
LABTEMP  cpyy.c  file2  klefab  nefile  rand.c  vaibhav
Marvel  crypt.txt  hello.txt  klefac  new  rand.h  xv6
a.out  decrypt.txt  hole.c  klefad  newfile  random.c  xv6-public
after  error  infile  klu  old  random.txt
before  ex.txt  kingdom  loo.txt  out  search
```

k. What happens when you enter a shell metacharacter * with the echo command.

Ans.

```
vaibhav@Gryffindor:~$ echo *
1.c ERROR LABTEMP Marvel a.out after before cat compressed.tar.gz cpyy.c
crypt.txt decrypt.txt error ex.txt file.hole file1 file2 hello.txt hole.c infile kingdom
klef klefaa klefab klefac klefad klu loo.txt myfile.c myfile.txt nefile new newfile old
out output output.log rand.c rand.h random.c random.txt search vai.c vai.txt
vaibhav xv6 xv6-public
vaibhav@Gryffindor:~$ echo *txt
crypt.txt decrypt.txt ex.txt hello.txt loo.txt myfile.txt random.txt vai.txt
vaibhav@Gryffindor:~$ echo *c
1.c cpyy.c hole.c klefac myfile.c rand.c random.c vai.c xv6-public
```

1.Perform the Following

a.Check \$ > newfile < infile wc and give result

Ans.

```
vaibhav@Gryffindor:~$ cat>infile
I am in infile
This is redirectioning
^Z
[26]+ Stopped          cat > infile
vaibhav@Gryffindor:~$ >newfile<infile wc
vaibhav@Gryffindor:~$ cat newfile
2 7 38
```

b.Redirect error message to file ERROR on cat command for non-existing file

Ans.

```
vaibhav@Gryffindor:~$ cat NEW 2>ERROR
vaibhav@Gryffindor:~$ cat ERROR
cat: NEW: No such file or directory
```

c.Redirect standard output and standard error streams for cat command with an example in one step.

Ans.vaibhav@Gryffindor:~\$ cat > out 2> error newFile NEW

```
vaibhav@Gryffindor:~$ cat out
Hello!!
This is my First program in operating System.
This is the first file I am creating using the command "cat"
vaibhav@Gryffindor:~$ cat error
cat: NEW: No such file or directory
```

m.Perform the following

a.Compress and uncompress a file ERROR

Ans.

```
vaibhav@Gryffindor:~$ cat > error
heyyya
^Z
[30]+ Stopped          cat > error
vaibhav@Gryffindor:~$ gzip error
vaibhav@Gryffindor:~$ cat error
cat: error: No such file or directory
vaibhav@Gryffindor:~$ ls
1.c  cat          file.hole  klef  myfile.c  output  vai.c
ERROR compressed.tar.gz file1  klefaa  myfile.txt output.log vai.txt
LABTEMP cpyy.c      file2  klefab  nefile  rand.c   vaibhav
Marvel  crypt.txt      hello.txt  klefac  new      rand.h   xv6
a.out  decrypt.txt    hole.c    klefad  newfile  random.c xv6-public
after  error.gz      infile  klu    old      random.txt
before ex.txt       kingdom  loo.txt out     search
vaibhav@Gryffindor:~$ gunzip error.zip
gzip: error.zip.gz: No such file or directory
vaibhav@Gryffindor:~$ gunzip error.gz
vaibhav@Gryffindor:~$ cat error
heyyya
```

b.Zip a group of files (ERROR, newfile, infile, passwd, group) and unzip them

Ans.

```
vaibhav@Gryffindor:~$ tar -czf compressed.tar.gz error klef klu newfile
new
vaibhav@Gryffindor:~$ tar -xzf compressed.tgz
tar (child): compressed.tgz: Cannot open: No such file or directory
tar (child): Error is not recoverable: exiting now
tar: Child returned status 2
tar: Error is not recoverable: exiting now
```

c.Gzip a group of files and gunzip them

Ans.

```
vaibhav@Gryffindor:~$ gzip error klu klef newfile
vaibhav@Gryffindor:~$ ls
1.c  cat          file.hole  klef.gz  myfile.c  output  vai.c
```



```

ERROR compressed.tar.gz file1 klefaa myfile.txt output.log vai.txt
LABTEMP cpyy.c file2 klefab nefile rand.c vaibhav
Marvel crypt.txt hello.txt klefac new rand.h xv6
a.out decrypt.txt hole.c klefad newfile.gz random.c xv6-public
after error.gz infile klu.gz old random.txt
before ex.txt kingdom loo.txt out search
vaibhav@Gryffindor:~$ gunzip error klu klef newfile
vaibhav@Gryffindor:~$ ls
1.c cat file.hole klef myfile.c output vai.c
ERROR compressed.tar.gz file1 klefaa myfile.txt output.log vai.txt
LABTEMP cpyy.c file2 klefab nefile rand.c vaibhav
Marvel crypt.txt hello.txt klefac new rand.h xv6
a.out decrypt.txt hole.c klefad newfile random.c xv6-public
after error infile klu old random.txt
before ex.txt kingdom loo.txt out search

```

n.Create a file called "hello.txt" in your home directory using the command cat -u > hello.txt. Ask your partner to change into your home directory and run tail -f hello.txt. Now type several lines into hello.txt. What appears on your partner's screen?

Ans.

```

vaibhav@Gryffindor:~$ ssh 192.168.2.56 -p 985 -l sambhav
sambhav@192.168.2.56's password :
Last login: Mon Jun 3 10:24:48 2019 from 10.52.11.225
sambhav@LAPTOP:~$ cd ..
sambhav@LAPTOP:/home$ ls
Lab vaibhav praneeth anil newfile stunt
tarun abhinav
sambhav@LAPTOP:/home$ cd vaibhav/
sambhav@LAPTOP vaibhav :~$ ls
1.c buzz doc hole.c klefad old search
ERROR cat dock infile klu out vai.c
LABTEMP compressed.tar.gz error input loo.txt output vai.txt
Marvel copyyyy ex.txt kingdom myfile.c output.log vaibhav
a.out cpyy.c file.hole klef myfile.txt rand.c xv6
after crypt.txt file1 klefaa nefile rand.h xv6-public
arroww darf file2 klefab new random.c
before decrypt.txt hello.txt klefac newfile random.txt
sambhav@LAPTOP vaibhav :~$ tail -f hello.txt
vaibhav@Gryffindor:~$ cat > hello.txt
Yo bro
Wassup

```

(whatever you wrote here will appear on your friend's screen)

o. Change the umask value and identify the difference with the earlier using touch, ls -l

Ans. vaibhav@Gryffindor:~\$ touch before
vaibhav@Gryffindor:~\$ umask u=rwx,g=,o=
vaibhav@Gryffindor:~\$ touch after
vaibhav@Gryffindor:~\$ ls -l
total 48
-rw----- 1 vaibhav vaibhav 0 Jun 20 12:46 after
-rw-rw-rw- 1 vaibhav vaibhav 0 Jun 20 12:45 before

p. Save the output of the who command in a file, display it, display lines count on the terminal.

Ans.
vaibhav@Gryffindor:~\$ who>hello.txt
vaibhav@Gryffindor:~\$ cat hello.txt
vaibhav@Gryffindor:~\$ wc -l hello.txt
0 hello.txt

q. Two or more commands can be combined, and the aggregate output can send to an output file. How to perform it. Write a sample command line.

Ans. vaibhav@Gryffindor:~\$ ls -l|wc -l>output.log
vaibhav@Gryffindor:~\$ cat output.log
48

r. Display all files in the current directory that begins with "a", "b" or "c" and are at least 5 characters long

Ans.
vaibhav@Gryffindor:~\$ echo [a-c]*????
a.out after before compressed.tar.gz cpyy.c crypt.txt

s. Display all files in the current directory that begin and with don't end with "x", "y" or "z"

Ans.
vaibhav@Gryffindor:~\$ echo [a-c]*[!x-z]
a.out after arroww before cat cpyy.c crypt.txt

t.Display all files in the current directory that begin with the letter D and have three more characters

Ans.

```
vaibhav@Gryffindor:~$ echo [d]*???  
darf decrypt.txt dock
```

u.How to redirect stdout of a command to a file, use the “>”

Ans.

```
vaibhav@Gryffindor:~$ echo "hey students" > week2.txt  
vaibhav@Gryffindor:~$ cat week2.txt  
hey students
```

WEEK - 3 (PROCESS API)

Prerequisite:

- Analyzing the concept of fork ()
- Use of wait system call for parent and child processes
- Retrieving the PID for parent and the child
- Concepts of dup() , dup2().
- Understanding various types of exec calls
- The init process

Pre lab task:

1. Write brief description and prototypes in the space given below for the following process sub system call

EX:- “\$ man system call name”

1. fork()

NAME

fork - create a child process

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);

2. getpid(), getppid() system calls

NAME

getpid, getppid - get process identification

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);

3. `exit()` system call

NAME

exit - cause normal process termination

SYNOPSIS

#include <stdlib.h>

void exit(int status);

4. `shmget()`

NAME

shmget - allocates a System V shared memory segment

SYNOPSIS

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

5. `wait()`

NAME

wait, waitpid, waitid - wait for process to change state

SYNOPSIS

#include <sys/types.h>

#include <sys/wait.h>

pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);

6. `sleep()`

NAME

sleep - delay for a specified amount of time

SYNOPSIS

sleep NUMBER[SUFFIX]...

sleep OPTION

(or)

```
#include<unistd.h>
unsigned int sleep(unsigned int seconds);
```

7. exec()

NAME

execl, execlp, execl, execv, execvp, execvpe - execute a file

SYNOPSIS

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

```
extern char **environ;
```

```
int execl(const char *path, const char *arg, ...
          /* (char *) NULL */);
int execlp(const char *file, const char *arg, ...
          /* (char *) NULL */);
int execl(const char *path, const char *arg, ...
          /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

8. waitpid()

NAME

wait, waitpid, waitid - wait for process to change state

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *wstatus);
```

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

9. `execlp()`, `execvp()`, `execv()`, `execl()` `execv()` system calls

NAME

`execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe` - execute a file

SYNOPSIS

`#include <unistd.h>`

`extern char **environ;`

**`int execl(const char *path, const char *arg, ...`
 `/* (char *) NULL */;`
**`int execlp(const char *file, const char *arg, ...`
 `/* (char *) NULL */;`
**`int execle(const char *path, const char *arg, ...`
 `/*, (char *) NULL, char * const envp[] */;`
`int execv(const char *path, char *const argv[]);`
`int execvp(const char *file, char *const argv[]);`
**`int execvpe(const char *file, char *const argv[],`
 `char *const envp[]);`********

10. `_exit()`

NAME

`_exit`, `_Exit` - terminate the calling process

SYNOPSIS

`#include <unistd.h>`

`void _exit(int status);`

`#include <stdlib.h>`

`void _Exit(int status);`

2. Write a C program to create a process in UNIX(using fork()) .

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
    int pid_t,pid,pid1,p,p1;

    pid = fork();

    if (pid ==-1)
    {
        printf("child process not created \n");
    }
    else if(pid==0)

    {
        printf("\n child process1 :\n");
        p=getppid();
        printf("parent process id of child1: %d\n",p);
        p1=getpid();
        printf("parent process id of child1: %d\n",p1);
    }
    else
    {
        pid1 = fork();
        if(pid==0)
        {
            printf("\nsecond child process :\n");
            p=getppid();
            printf("parent process id of second child is : %d\n",p);
            p1=getpid();
            printf("parent process id of second child is : %d\n",p1);
        }

        else
        {
            printf("this is parent process \n");
            p=getppid();
            printf("grand parent: %d \n",p);
            p1=getpid();
            printf("process id of parent: %d \n",p1);
```



```
}  
}  
return 0;  
}
```

3. Write a C program to print Process ID and Parent Process ID.

```
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
  
int main()  
{  
  
pid_t pid,ppid;  
  
gid_t gid;  
  
uid_t uid;  
  
pid = getpid();  
  
ppid = getppid();  
  
if(pid < 0)  
{  
  
perror("unable to get process id(PID) \n");  
}  
else  
{  
  
printf("PROCESS ID : %d \n",pid);  
  
}  
  
if(ppid < 0)  
  
{
```

```

    perror("unable to get parent process id(PPID) \n");

}

else
{

    printf("PARENT PROCESS ID IS : %d \n",ppid);

}

return 0;

}

```

In lab task:

1. The process obtains its own PID and its parent's PID using the getpid and getppid system calls. The program also prints the effective UID and GID, which normally are equal to the real UID and GID. Write a UNIX system program that prints PID, PPID, real and effective UIDs and GIDs.

```

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

int main()
{

    pid_t pid,ppid;

    gid_t gid;

    uid_t uid;

    pid = getpid();

    ppid = getppid();

    gid = getgid();

```

```
uid = getuid();

if(pid < 0)

{

    perror("unable to get process id(PID) \n");
}
else
{

    printf("PROCESS ID : %d \n",pid);

}

if(ppid < 0)

{

    perror("unable to get parent process id(PPID) \n");

}

else
{

    printf("PARENT PROCESS ID IS : %d \n",ppid);

}

if(gid < 0)

{

    perror("unable to get group id(GID) \n");

}

else

{
```

```

printf("GROUP ID IS : %d \n",gid);

}

if(uid < 0)

{

perror("unable to get user id(UID) \n");

}

else

{

printf("USER ID IS : %d \n",uid);

}

return 0;

}

```

2.T-series creates a text document(song.txt) that contains lyrics of a song. They want to know how many lines and words are present in the song.txt. They want to utilize Linux directions and system calls to accomplish their objective. Help T-series to finish their task by utilizing a fork system call. Print the number of lines in song.txt using the parent process and print the number of words in it using the child process.

```

#include <unistd.h>
#include <stdio.h>
#include<stdlib.h>
#include <sys/stat.h>
#include<wait.h>
#include <fcntl.h>

int main()
{
    char* pcomm[] = {"wc","-l","song.txt",NULL};
    char* ccomm[] = {"wc","-w","song.txt",NULL};
    switch(fork())

```

```

    {
    case -1:
    perror("fork");
    exit(1);
    case 0:
    //child
    if(execvp("wc",ccomm)<0)
    printf(" execvp error in child\n");
    exit(2);
    break;
    default:
    if(execvp("wc",pcomm)<0)
    printf(" execvp error in parent\n");
    exit(3);
    break;
    }
}

```

3. Write a program to execute a given command and show the usage of dup2 function in a way that instead of printing on monitor, redirecting the output to a file named "week3.txt"

```

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

#include<stdlib.h>

#include <sys/stat.h>
#include<wait.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
if(argc > 1 ){
    if(fork()==0){
        int fd =
open("week3.txt",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
        dup2(fd,1);
        execvp(argv[1], &argv[1]); //ls
    }
    perror("exec error");
}
}

```

```

    exit(2);
    }
else{
    int rv;
    wait(&rv);
    printf("Exit status: %d\n", WEXITSTATUS(rv));
    exit(0);
    }
}
else{
    perror("Error Provide arguments\n");
    }
}

```

SKILL EXERCISE 3

1. Write a program redirection.c that achieves the effect of redirection without using the < and > symbols. Child opens files and uses dup2 to reassign the descriptors and execute a command line.

Note: The first two arguments are input and output filenames. The command line to execute is specified as the remaining arguments.

1)

```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <wait.h>
#include <fcntl.h>
int main(int argc, char **argv)
{

```

```

if(argc> 1 ){
    if(fork()==0){
        int fd =
open("Lab3.txt",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
        dup2(fd,1);
        execvp(argv[1], &argv[1]); //ls
        perror("exec error");
        exit(2);
    }
    else{
        int rv;
        wait(&rv);
        printf("Exit status: %d\n", WEXITSTATUS(rv));
        exit(0);
    }
}
else{
    perror("Error Provide arguments\n");
}
}

```

2.open fire fox in ubuntu . write a program that sleeps for 5 seconds and after 5 seconds it should kill the fire fox browser which is opened before(means it should close the fire fox browser)

Ans

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
    sleep(5);

```

```

        printf("5 seconds completed.....");
        system("killall firefox");
        exit(0);
    return 0;
}

```

3. Write a program to implement FCFS Process Scheduling Algorithms

```

#include<stdio.h>

int main()
{
    int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
    printf("Enter total number of processes(maximum 20):");
    scanf("%d",&n);

    printf("\nEnter Process Burst Time\n");
    for(i=0;i<n;i++)
    {
        printf("P[%d]:",i+1);
        scanf("%d",&bt[i]);
    }

    wt[0]=0;
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }
    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }

    avwt/=i;

```



```

    avtat/=i;
    printf("\n\nAverage Waiting Time:%d",avwt);
    printf("\nAverage Turnaround Time:%d",avtat);

    return 0;
}

```

4. Write a program to implement SJF Process Scheduling Algorithms

```

#include<stdio.h>
void main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;          //contains process number
    }

    //sorting burst time in ascending order using selection sort
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }
}

```

```

    wt[0]=0;           //waiting time for first process will be zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=(float)total/n;   //average waiting time
    total=0;

    printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];   //calculate turnaround time
        total+=tat[i];
        printf("\np%d\t\t %d\t\t %d\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=(float)total/n;   //average turnaround time
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}

```

5. Write a program to implement RR Process Scheduling Algorithms

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

```

int main()
{

    int count,j,n,time,remain,flag=0,time_quantum;
    int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
    printf("Enter Total Process:\t ");
    scanf("%d",&n);
    remain=n;
    for(count=0;count<n;count++)
    {

```

```

    printf("Enter Arrival Time and Burst Time for Process Process Number %d
: ",count+1);
    scanf("%d",&at[count]);
    scanf("%d",&bt[count]);
    rt[count]=bt[count];
}
printf("Enter Time Quantum:\t");
scanf("%d",&time_quantum);
printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
for(time=0,count=0;remain!=0;)
{
    if(rt[count]<=time_quantum && rt[count]>0)
    {
        time+=rt[count];
        rt[count]=0;
        flag=1;
    }
    else if(rt[count]>0)
    {
        rt[count]-=time_quantum;
        time+=time_quantum;
    }
    if(rt[count]==0 && flag==1)
    {
        remain--;
        printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);
        wait_time+=time-at[count]-bt[count];
        turnaround_time+=time-at[count];
        flag=0;
    }
    if(count==n-1)
        count=0;
    else if(at[count+1]<=time)
        count++;
    else
        count=0;
}
printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);

return 0;
}

```

WEEK - 4(PROCESS SCHEDULING &XV6)

Pre-requisites:

- Knowledge on simple system calls of xv6
- Complete idea on spin() function
- Understanding concept of pipe
- Analyse the date system call in xv6
- Priority process scheduling algorithms

Pre- Lab:

1. write xv6 process system calls

1.fork() :-

```
Int sys_fork(void)
{
    return 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. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork()

2.exit() :-

```
Int sys_exit(void)
{
    exit();
    return 0;
}
```

The function _exit() terminates the calling process "immediately". Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, init, and the process's parent is sent a SIGCHLD signal. The value status is returned to the parent process as the process's exit status, and can be collected using one of the wait() family of calls.

3.wait() :-

```
int sys_wait(void)
{
    return wait();
}
```

The wait() system call suspends execution of the current process until one of its children terminates.

4.kill() :-

```
Int sys_kill(void)
{
    Int pid;
    If(argint(0,&pid)<0)
        Return -1;
    Return kill(pid);
}
```

The kill() system call can be used to send any signal to any process group or process.

5.getpid() :-

```
int sys_getpid(void)
{
    return myproc()->pid;
}
```

getpid() returns the process ID (PID) of the calling process. (This is often used by routines that generate unique temporary filenames.)

6.sbrk():-

```
Int sys_sbrk(void)
{
    Int addr;
    Int n;
    If(argint(0,&n)<0)
        return -1;
    return addr;
}
```

sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

7.sleep :-

```
Int sys_sleep(void)
{
    Int n;
    Uint ticks0;
    If(argint(0,&n)<0);
    return -1;
    acquire(&tickslock);
}
```

```

ticks0=ticks;
while(ticks-ticks0<n)
{
    release(&ticks,&tickslock);
    return -1;
}
Sleep(&ticks,&tickslock);
}release(&tickslock);
return 0;
}

```

8. sleep()

causes the calling thread to sleep either until the number of real-time seconds specified in seconds have elapsed or until a signal arrives which is not ignored.

9.uptime():-

```

Int sys_uptime(void)
{
    Uint xticks;
    acquire(&tickslock);
    xticks=ticks;
    release(&tickslock);
    return xticks;
}

```

2. Create a child process in xv6 using system calls

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program after this instruction
    fork();
    printf("Hello world!\n");
    return 0;
}

```

Output:-

Hello world!

Hello world!

3. Write a UNIX system program to create zombie (by letting parent sleep & child dies) and orphan process (by letting child sleep for 2 minutes Parent doesn't call wait and dies immediately)

Zombie

```
#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{    // fork() creates child process identical to parent

    int pid = fork();

    // if pid is greater than 0 then it is parent process

    // if pid is 0 then it is child process

    // if pid is -ve , it means fork() failed to create child process

    // Parent process

    if (pid > 0)

        sleep(20);

    // Child process

    else

    {

        exit(0); }

    return 0;
```

In-Lab:

1. Write a UNIX system program to simulate shell - - execute commands with options and background. (accepts user input as a command to be executed. uses the strtok library function for parsing command line)

ans:

```
#include<stdlib.h>

#include<stdio.h>
```

```
#include<string.h>

#include<unistd.h>

int main()

{

char str[100];

printf(">");

scanf("%[^\n]s",str);

char *token;

char *rest=str;

char *s[100];//for command to execvp

int i=0;

while((token=strtok_r(rest,"",&rest)))

{

s[i]=token;

i++;

}

s[i]=NULL;

int p=fork();

if(p==-1)

{

perror("error");

}

else if(p==0)

{

execvp(s[0],s);
```



```

}

else

{

if(strcmp(str,"exit")!=0)

{

wait();

char *re[]={"/a.out",NULL};

execvp(re[0],re);

}

else

{

printf("Bye...");

exit(1);

}

}

return 0;

}

```

2. Write a Unix System Program to establish a connection between two process in such a way that the output of one process is the input of the other using pipe() system call in xv6.

```

#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{

```

```

char inbuf[MSGSIZE];
int p[2], i;

if (pipe(p) < 0)
    exit(1);

/* continued */
/* write pipe */

write(p[1], msg1, MSGSIZE);
write(p[1], msg2, MSGSIZE);
write(p[1], msg3, MSGSIZE);

for (i = 0; i < 3; i++) {
    /* read pipe */
    read(p[0], inbuf, MSGSIZE);
    printf("%s\n", inbuf);
}
return 0;
}

```

3. Write a system program to demonstrate shared memory IPC -execute with and without command line arguments.

ANS:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */

```

```

if ((key = ftok("shmdemo.c", 'R')) == -1) {
    perror("ftok");
    exit(1);
}

/* connect to (and possibly create) the segment: */
if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
    perror("shmget");
    exit(1);
}

/* attach to the segment to get a pointer to it: */
printf("SHMID = %d\n", shmid);
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1)) {
    perror("shmat");    exit(1);
}

/* read or modify the segment, based on the command line: */
if (argc == 2) {
    printf("writing to segment: \"%s\"\n", argv[1]);
    strncpy(data, argv[1], SHM_SIZE);
} else
    printf("segment contains: \"%s\"\n", data);

/* detach from the segment: */
if (shmdt(data) == -1) {
    perror("shmdt");
    exit(1);
}

return 0;
}

```

SKILL EXERCISE 4 (xv 6):

1. Write a program to implement Priority Process Scheduling Algorithms

ANS:

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

```
int main()
```

```
{
```

```
    int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
```

```
    printf("Enter Total Number of Process:");
```

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

```

printf("\nEnter Burst Time and Priority\n");
for(i=0;i<n;i++)
{
    printf("\nP[%d]\n",i+1);
    printf("Burst Time:");
    scanf("%d",&bt[i]);
    printf("Priority:");
    scanf("%d",&pr[i]);
    p[i]=i+1;          //contains process number
}

```

```

//sorting burst time, priority and process number in ascending order using selection sort
for(i=0;i<n;i++)

```

```

{
    pos=i;
    for(j=i+1;j<n;j++)
    {
        if(pr[j]<pr[pos])
            pos=j;
    }

```

```

    temp=pr[i];
    pr[i]=pr[pos];
    pr[pos]=temp;

```

```

    temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp;

```

```

    temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
}

```

```

wt[0]=0;    //waiting time for first process is zero

```

```

//calculate waiting time

```

```

for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];
}

```

```

        total+=wt[i];
    }

    avg_wt=total/n;    //average waiting time
    total=0;

    printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i]; //calculate turnaround time
        total+=tat[i];
        printf("\nP[%d]\t\t %d\t\t %d\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n;    //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\n\nAverage Turnaround Time=%d\n",avg_tat);

    return 0; }

```

2. Write a procedure to update date system call to xv6 operating system.

```

#include "types.h"
#include "user.h"
#include "date.h"
Int main(int argc, char *argv[])
{
    struct rtcdate r;
    if (date(&r)) {
        printf(2, "date failed\n");
        exit();
    }
    // your code to print the time in any format you like...
    exit();
}

```

In order to make your new date program available to run from the xv6 shell, add `_date` to the `UPROGS` definition in `Makefile`.

Your strategy for making a date system call should be to clone all of the pieces of code that are specific to some existing system call like `uptime`. Grep for `uptime` in all the source files, using `grep -n uptime *.c`.

When you're done, typing `date` to an xv6 shell prompt should print the current UTC time.

Home Assignment :

1. Define Operating Systems? Explain in detail about different types of Operating Systems?

2. Explain the implementation of the following system calls: open(), read(), write(), close()?

3. What is a Process? What are different states of process?

Explain the implementation of the following system calls : fork(), wait(), and exec()?

4. Define Scheduler? Explain Various types of cooperative and non cooperative scheduling algorithms on the given problem.

Time quantum for RR is 3ms

Process	Arrival Time(ms)	Burst Time(ms)
P0	2	10
P1	4	6
P2	0	5
P3	5	12

5. List out the five rules of MLFQ?

REVIEW – PROBLEM SET I

1. Answer yes/no, and provide a brief explanation.

(a) Can two processes be concurrently executing the same program executable?

(b) Can two running processes share the complete process image in physical memory (not just parts of it)?

2. Consider a process executing on a CPU. Give an example scenario that can cause the process to undergo:

(a) A voluntary context switch.

(b) An involuntary context switch.

3. Consider a parent process P that has forked a child process C. Now, P terminates while C is still running. Answer yes/no, and provide a brief explanation.

(a) Will C immediately become a zombie?

(b) Will P immediately become a zombie, until reaped by its parent?

4. A process in user mode cannot execute certain privileged hardware instructions. [T/F]

ANS:

5. Which of the following C library functions do NOT directly correspond to (similarly named) system calls? That is, the implementations of which of these C library functions are NOT straight-forward invocations of the underlying system call?

- A. system, which executes a bash shell command.
- B. fork, which creates a new child process.
- C. exit, which terminates the current process.
- D. strlen, which returns the length of a string.

ANS:

6. Which of the following actions by a running process will always result in a context switch of the running process, even in a non-preemptive kernel design?

- A. Servicing a disk interrupt, that results in another blocked process being marked as ready/runnable.
- B. A blocking system call.
- C. The system call exit, to terminate the current process.
- D. Servicing a timer interrupt.

ANS:

7. Consider two machines A and B of different architectures, running two different operating systems OS-A and OS-B. Both operating systems are POSIX compliant. The source code of an application that is written to run on machine A must always be rewritten to run on machine B.

ANS:

8. Consider the scenario of the previous question. An application binary that has been compiled for machine A may have to be recompiled to execute correctly on machine B. [T/F]

ANS:

9. A process makes a system call to read a packet from the network device, and blocks. The scheduler then context-switches this process out. Is this an example of a voluntary context switch or an involuntary context switch?

ANS:

10. A context switch can occur only after processing a timer interrupt, but not after any other system call or interrupt. [T/F]

ANS:

11. A C program cannot directly invoke the OS system calls and must always use the C library for this purpose. [T/F]

ANS:

12. A process undergoes a context switch every time it enters kernel mode from user mode. [T/F]

ANS:

13. When a process makes a system call to transmit a TCP packet over the network, which of the following steps do NOT occur always?

- A. The process moves to kernel mode.
- B. The program counter of the CPU shifts to the kernel part of the address space.
- C. The process is context-switched out and a separate kernel process starts execution.
- D. The OS code that deals with handling TCP/IP packets is invoked.

ANS:

14. Consider a parent process P that has forked a child process C in the program below. `int a = 5; int fd = open(...) //opening a file int ret = fork(); if(ret > 0) { close(fd); a = 6;`

`...`

`}`

`else if(ret==0) { printf("a=%d\n", a); read(fd, something);`

`}`

After the new process is forked, suppose that the parent process is scheduled first, before the child process. Once the parent resumes after fork, it closes the file descriptor and changes the value of a variable as shown above. Assume that the child process is scheduled for the first time only after the parent completes these two changes.

(a) What is the value of the variable a as printed in the child process, when it is scheduled next? Explain.

(b) Will the attempt to read from the file descriptor succeed in the child? Explain.

15. Consider a parent process that has forked a child in the code snippet below.

```
int count = 0;
ret = fork(); if(ret == 0)
{
    printf("count in child=%d\n", count);
}
else
{
    count = 1;
}
```

The parent executes the statement "count = 1" before the child executes for the first time. Now, what is the value of count printed by the code above? Assume that the OS implements a regular fork (not a copy-on-write fork).

ANS:

16. Repeat the previous question for a copy-on-write fork implementation in the OS. Recall that with copy-on-write fork, the parent and child use the same memory image, and a copy is made only when one of them wishes to modify any memory location.

ANS:

17. Consider the wait family of system calls (wait, waitpid etc.) provided by Linux. A parent process uses some variant of the wait system call to wait for a child that it has forked. Which of the following statements is always true when the parent invokes the system call?

- A. The parent will always block.
- B. The parent will never block.
- C. The parent will always block if the child is still running.
- D. Whether the parent will block or not will depend on the system call variant and the options with which it is invoked.

ANS:

18. Consider a simple linux shell implementing the command 'sleep 100'. Which of the following is an accurate ordered list of system calls invoked by the shell from the time the user enters this command to the time the shell comes back and asks the user for the next input?

- A. wait-exec-fork
- B. exec-wait-fork
- C. fork-exec-wait
- D. wait-fork-exec

ANS:

19. Consider a process that has requested to read some data from the disk and blocks. Subsequently, the data from the disk arrives and the interrupt is serviced. However, the process doesn't start running immediately. What is the state of this process at this stage?

ANS:

20. Consider a process P1 that forks P2, P2 forks P3, and P3 forks P4. P1 and P2 continue to execute while P3 terminates. Now, when P4 terminates, which process must wait for and reap P4?

ANS:

21. Consider the following three processes that arrive in a system at the specified times, along with the duration of their CPU bursts. Process P1 arrives at time $t=0$, and has a CPU burst of 10 time units. P2 arrives at $t=2$, and has a CPU burst of 2 units. P3 arrives at $t=3$, and has a CPU burst of 3 units. Assume that the processes execute only once for the duration of their CPU burst, and terminate immediately. Calculate the time of completion of the three processes under each of the following scheduling policies. For each policy, you must state the completion time of all three processes, P1, P2, and P3. Assume there are no other processes in the scheduler's queue. For the preemptive policies, assume that a running process can be immediately preempted as soon as the new process arrives (if the policy should decide to preempt).

(a) First Come First Serve

(b) Shortest Job First (non-preemptive)

© Shortest Remaining Time First (preemptive)

(c) Round robin (preemptive) with a time slice of (atmost) 5 units per process

CO-2

WEEK - 5 (MEMORY API)

Prerequisite:

- General idea on heap memory
- malloc(), realloc(), calloc(), free() library functions
- Concept of altering program break
- Basic strategies of managing free space (first-fit , best-fit , worst-fit)
- Concepts of Splitting and coalescing in free space management

Pre-Lab Task:

1. Write the functionality, arguments required, and the possible return values in different cases of the following system calls

- brk()

ANS:

brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size

On success, brk() returns zero. On error, -1 is returned, and errno is set to ENOMEM.

- sbrk()

ANS:

sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

On success, sbrk() returns the previous program break. (If the break was increased, then this value is a

pointer to the start of the newly allocated memory). On error, (void *) -1 is returned, and errno is set to ENOMEM.

2.Consider the following program:

```
#include <stdio.h>
int main (int argc, char **argv) {
    int i;
    void *start, *finish;
    void *res[10];
    start = sbrk(0);
    for (i = 0; i < 10; i++) {
        res[i] = malloc(10);
    }
    finish = sbrk(0);
    /* TWO */
    return 0;
}
```

a) How many times does the system call **sbrk()** get called from within **malloc()**?

ANS:

1

b) On the i386 platform, what is the numeric value of start,finish?

ANS:

In Lab Task:

1. Develop a program to illustrate the effect of free() on the program break. This program allocates multiple blocks of memory and then frees some or all of them, depending on its (optional) command-line arguments. The first two command-line arguments specify the number and size of blocks to allocate. The third command-line argument specifies the loop step unit to be used when freeing memory blocks. If we specify 1 here (which is also the default if this argument is omitted), then the program frees every memory block; if 2, then every second allocated block; and so on. The fourth and fifth command-line arguments specify the range of blocks that we wish to free. If these arguments are omitted, then all allocated blocks (in steps given by the third command-line argument) are freed. Find the present address of the program break using sbrk() and expand the program break by the size 1000000 using brk().

ANS:

```
#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#define MAX_ALLOCS 1000000

int getInt (char* s)

{

    int sum=0,l=strlen(s);

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

        switch (s[i]){

            case '0':

                sum = sum*10;

                break;

            case '1':

                sum = sum*10 + 1;

                break;

            case '2':

                sum = sum*10 + 2;

                break;
```

```

        case '3':
            sum = sum*10 + 3;
            break;
        case '4':
            sum = sum*10 + 4;
            break;
        case '5':
            sum = sum*10 + 5;
            break;
        case '6':
            sum = sum*10 + 6;
            break;
        case '7':
            sum = sum*10 + 7;
            break;
        case '8':
            sum = sum*10 + 8;
            break;
        case '9':
            sum = sum*10 + 9;
            break;
    }

}

return sum;
}

int main(int argc, char *argv[])
{
    char *ptr[MAX_ALLOCS];

    int freeStep, freeMin, freeMax, blockSize, numAllocs, j;

```



```

printf("\n");

if (argc < 3 || strcmp(argv[1], "--help") == 0)
{
    printf("%s num-allocs block-size [step [min [max]]]\n", argv[0]);
    exit(1);
}

numAllocs = getInt(argv[1]);

if (numAllocs > MAX_ALLOCS){
    printf("num-allocs > %d\n", MAX_ALLOCS);
    exit(1);
}

blockSize = getInt(argv[2]);

freeStep = (argc > 3) ? getInt(argv[3]) : 1;
freeMin = (argc > 4) ? getInt(argv[4]) : 1;
freeMax = (argc > 5) ? getInt(argv[5]) : numAllocs;

if (freeMax > numAllocs){
    printf("free-max > num-allocs\n");
    exit(1);
}

printf("Initial program break:%10p\n", sbrk(0));

printf("Allocating %d*%d bytes\n", numAllocs, blockSize);

for (j = 0; j < numAllocs; j++) {
    ptr[j] = malloc(blockSize);

    if (ptr[j] == NULL){
        perror("malloc");
        exit(1);
    }
}

printf("Program break is now:%10p\n", sbrk(0));

```

```

printf("Freeing blocks from %d to %d in steps of %d\n",freeMin, freeMax, freeStep);

for (j = freeMin - 1; j < freeMax; j += freeStep)

free(ptr[j]);

printf("After free(), program break is: %10p\n", sbrk(0));

exit(0);

}

```

2. Problem Description:

Twin Sisters Riya and Siya collected money to help their friend Radha to buy books. Riya approached her elder sister Diya to help her calculate the total amount she collected. Riya said that she collected money from 'n' persons and gave Diya a list of values of amount she collected from each person. After a while Siya arrived and said that she also collected money from 'm' people and submitted information like Riya.

Now Diya has inputs as bellow

First value is 'n', followed by n values of amounts collected by Riya. Then the next value is 'm', followed by m values of amounts collected by Siya.

As you are explaining Diya, how to allocate memory dynamically. Take this situation and provide her with an example program, which stores the above n values provide by Riya in an array using malloc () and then m values provided by Siya in the same array and using realloc() calculate the total amount they have collected using malloc(), realloc(). Don't forget to free the memory allocated, by using free ().

ANS:

```

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

int main(){
    int n,m;
    scanf("%d",&n);
    int* arr = (int*) malloc(n*sizeof(int));
    for(int i=0;i<n;i++)
        scanf("%d",&arr[i]);
    scanf("%d",&m);
    arr = (int*) realloc(arr,sizeof(int)*(n+m));
    for(int i=n;i<n+m;i++)
        scanf("%d",&arr[i]);
    int total = 0;
    for(int i=0;i<n+m;i++)
        total+=arr[i];
    printf("Total = %d\n",total);
    free(arr);
}

```

OUTPUT:

5

12 45 67 45 89

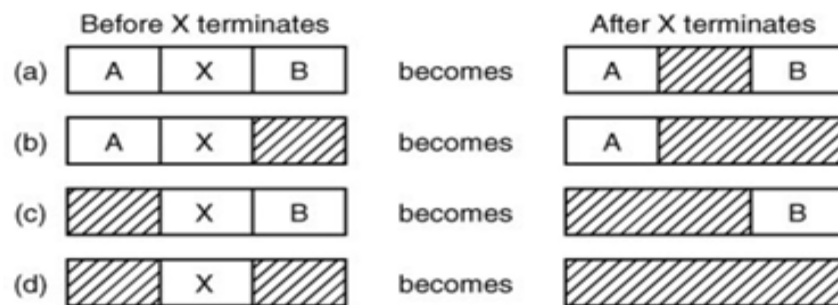
6

50 39 26 30 40 5

Total = 448

SKILL EXERCISE 5:

1. One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of enough size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.



ANS:

FIRST FIT

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
// Function to allocate memory to
// blocks as per First fit algorithm
void firstFit(int blockSize[], int m,
              int processSize[], int n)
```

```

{
    // Stores block id of the
    // block allocated to a process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
    // according to its size and assign to it
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                // allocate block j to p[i] process
                allocation[i] = j;

                // Reduce available memory in this block.
                blockSize[j] -= processSize[i];

                break;
            }
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t", i+1);
        printf("%d\t", processSize[i]);
        if (allocation[i] != -1)
            printf("%d", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

```

```

        firstFit(blockSize, m, processSize, n);

    return 0 ;
}

```

Best Fit

```

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

// Function to allocate memory to blocks as per Best fit
// algorithm
void bestFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i=0; i<n; i++)
    {
        // Find the best fit block for current process
        int bestIdx = -1;
        for (int j=0; j<m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (bestIdx == -1)
                    bestIdx = j;
                else if (blockSize[bestIdx] > blockSize[j])
                    bestIdx = j;
            }
        }

        // If we could find a block for current process
        if (bestIdx != -1)
        {
            // allocate block j to p[i] process
            allocation[i] = bestIdx;

            // Reduce available memory in this block.
            blockSize[bestIdx] -= processSize[i];
        }
    }
}

```

```

        }
    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t", i+1);
        printf("%d\t", processSize[i]);
        if (allocation[i] != -1)
            printf("%d", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);

    bestFit(blockSize, m, processSize, n);

    return 0 ;
}

```

Worst Fit

```

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

// Function to allocate memory to blocks as per worst fit
// algorithm
void worstFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));
}

```

```

// pick each process and find suitable blocks
// according to its size and assign to it
for (int i=0; i<n; i++)
{
    // Find the best fit block for current process
    int wstIdx = -1;
    for (int j=0; j<m; j++)
    {
        if (blockSize[j] >= processSize[i])
        {
            if (wstIdx == -1)
                wstIdx = j;
            else if (blockSize[wstIdx] < blockSize[j])
                wstIdx = j;
        }
    }

    // If we could find a block for current process
    if (wstIdx != -1)
    {
        // allocate block j to p[i] process
        allocation[i] = wstIdx;

        // Reduce available memory in this block.
        blockSize[wstIdx] -= processSize[i];
    }
}

printf("\nProcess No.\t\tProcess Size\t\tBlock no.\n");
printf("\n");
for (int i = 0; i < n; i++)
{
    printf("%d\t\t", i+1);
    printf("%d\t\t", processSize[i]);
    if (allocation[i] != -1)
        printf("%d", allocation[i] + 1);
    else
        printf("Not Allocated\n");
}
}

int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);

    worstFit(blockSize, m, processSize, n);
}

```

```
        return 0 ;
    }
```

2. Write a simple memory allocator: memalloc is a simple memory allocator. Which uses own malloc (), calloc(), realloc() and free() implemented using system calls.

ANS:

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

```
#include <string.h>
```

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

```
/* Only for the debug printf */
```

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

```
struct header_t {
    size_t size;
    unsigned is_free;
    struct header_t *next;
};
```

```
struct header_t *head = NULL, *tail = NULL;
```

```
pthread_mutex_t global_malloc_lock;
```

```
struct header_t *get_free_block(size_t size)
```

```
{
    struct header_t *curr = head;
    while(curr) {
        /* see if there's a free block that can accomodate requested size */
        if (curr->is_free && curr->size >= size)
            return curr;
    }
}
```



```

        curr = curr->next;
    }
    return NULL;
}

void free(void *block)
{
    struct header_t *header, *tmp;

    /* program break is the end of the process's data segment */
    void *programbreak;

    if (!block)
        return;

    pthread_mutex_lock(&global_malloc_lock);
    header = (struct header_t*)block - 1;

    /* sbrk(0) gives the current program break address */
    programbreak = sbrk(0);

    /*
     Check if the block to be freed is the last one in the
     linked list. If it is, then we could shrink the size of the
     heap and release memory to OS. Else, we will keep the block
     but mark it as free.
    */
    if ((char*)block + header->size == programbreak) {
        if (head == tail) {
            head = tail = NULL;
        } else {
            tmp = head;
            while (tmp) {

```

```

        if(tmp->next == tail) {
            tmp->next = NULL;
            tail = tmp;
        }
        tmp = tmp->next;
    }
}

/*
    sbrk() with a negative argument decrements the program break.
    So memory is released by the program to OS.
*/
sbrk(0 - header->size - sizeof(struct header_t));
/* Note: This lock does not really assure thread
    safety, because sbrk() itself is not really
    thread safe. Suppose there occurs a foreign sbrk(N)
    after we find the program break and before we decrement
    it, then we end up releasing the memory obtained by
    the foreign sbrk().
*/
pthread_mutex_unlock(&global_malloc_lock);
return;
}

header->is_free = 1;
pthread_mutex_unlock(&global_malloc_lock);
}

void *malloc(size_t size)
{
    size_t total_size;
    void *block;

```

```

struct header_t *header;

if (!size)

    return NULL;

pthread_mutex_lock(&global_malloc_lock);

header = get_free_block(size);

if (header) {

    /* Woah, found a free block to accomodate requested memory. */

    header->is_free = 0;

    pthread_mutex_unlock(&global_malloc_lock);

    return (void*)(header + 1);

}

/* We need to get memory to fit in the requested block and header from OS. */

total_size = sizeof(struct header_t) + size;

block = sbrk(total_size);

if (block == (void*) -1) {

    pthread_mutex_unlock(&global_malloc_lock);

    return NULL;

}

header = block;

header->size = size;

header->is_free = 0;

header->next = NULL;

if (!head)

    head = header;

if (tail)

    tail->next = header;

tail = header;

pthread_mutex_unlock(&global_malloc_lock);

return (void*)(header + 1);

}

```

```

void *calloc(size_t num, size_t nsize)
{
    size_t size;
    void *block;
    if (!num || !nsize)
        return NULL;
    size = num * nsize;
    /* check mul overflow */
    if (nsize != size / num)
        return NULL;
    block = malloc(size);
    if (!block)
        return NULL;
    memset(block, 0, size);
    return block;
}

void *realloc(void *block, size_t size)
{
    struct header_t *header;
    void *ret;
    if (!block || !size)
        return malloc(size);
    header = (struct header_t*)block - 1;
    if (header->size >= size)
        return block;
    ret = malloc(size);
    if (ret) {
        /* Relocate contents to the new bigger block */
        memcpy(ret, block, header->size);
        /* Free the old memory block */
    }
}

```

```
        free(block);
    }
    return ret;
}

/* A debug function to print the entire link list */
void print_mem_list()
{
    struct header_t *curr = head;
    printf("head = %p, tail = %p \n", (void*)head, (void*)tail);
    while(curr) {
        printf("addr = %p, size = %zu, is_free=%u, next=%p\n",
               (void*)curr, curr->size, curr->is_free, (void*)curr->next);
        curr = curr->next;
    }
}
```

WEEK - 6 (MEMORY(PAGING,SEGMENTATION))

Prerequisite:

- Basic idea on Segmentation.
- Accessing of memory with paging.
- TLB control flow algorithm.
- Multilevel page table control flow.
- Page fault control flow algorithm and page replacement.

Pre-Lab Task:

1. What is a segmentation. Describe segmentation in your own words by explaining the types of segmentation and draw the logical view of segmentation.

ANS:

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments.

Segmentation gives user's view of the process which paging does not give. Here the user's view is mapped to physical memory.

There are types of segmentation:

1. Virtual memory segmentation –

Each process is divided into a number of segments, not all of which are resident at any one point in time.

2. Simple segmentation –

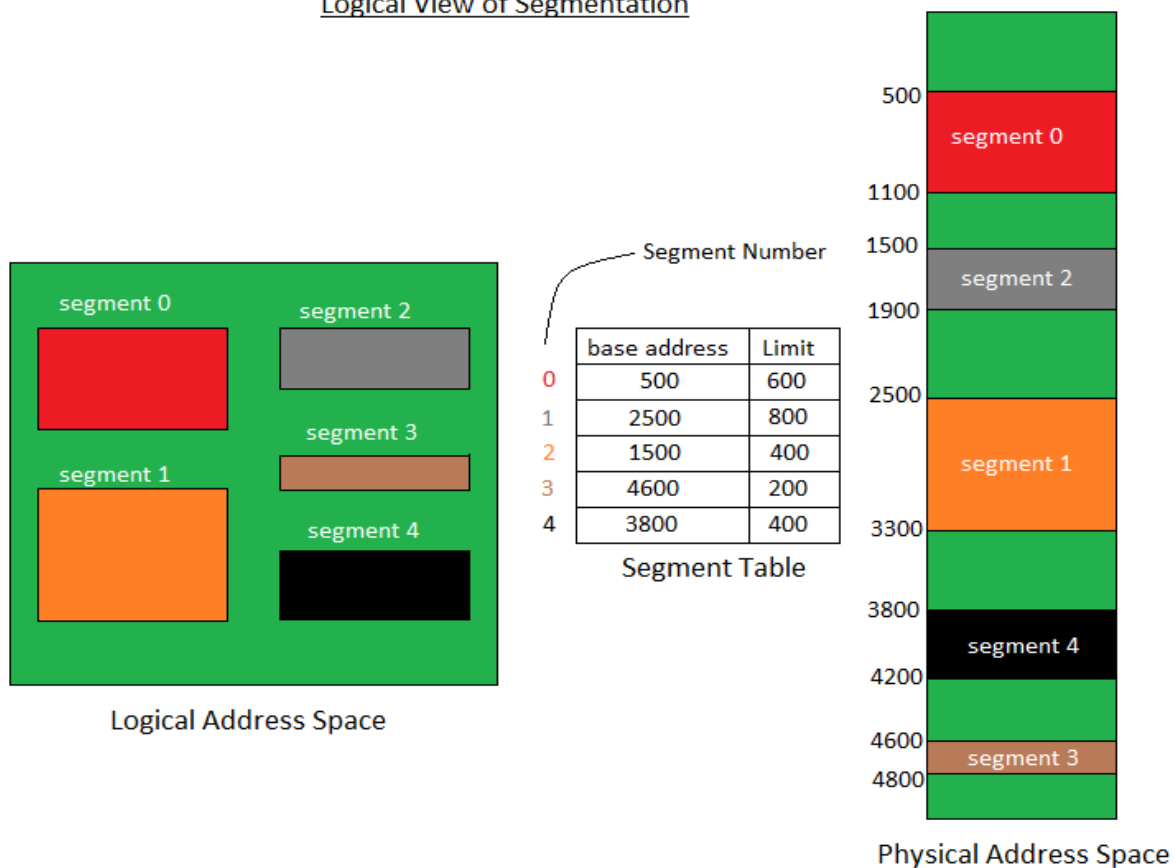
Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

Segment Table – It maps two-dimensional Logical address into one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Limit:** It specifies the length of the segment.

Logical View of Segmentation



2. What is the purpose of paging the page tables in OS? And explain types of page tables in brief.

ANS:

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the program executed by the accessing process, while physical addresses are used by the hardware, or more specifically, by the RAM subsystem. The page table is a key component of virtual address translation which is necessary to access data in memory.

TYPES OF PAGE TABLE:

1.Inverted page table

The *inverted page table* (IPT) is best thought of as an off-chip extension of the TLB which uses normal system RAM. Unlike a true page table, it is not necessarily able to hold all current mappings. The OS must be prepared to handle misses, just as it would with a MIPS-style software-filled TLB.

2.Multilevel Page Table

Multilevel paging is a paging scheme where there exists a hierarchy of page tables. The need for multilevel paging arises when-

- **The size of page table is greater than the frame size.**
- **As a result, the page table can not be stored in a single frame in main memory.**

In multilevel paging,

- **The page table having size greater than the frame size is divided into several parts.**
- **The size of each part is same as frame size except possibly the last part.**
- **The pages of page table are then stored in different frames of the main memory.**
- **To keep track of the frames storing the pages of the divided page table, another page table is maintained.**

3.Virtualized page table

It was mentioned that creating a page table structure that contained mappings for every virtual page in the virtual address space could end up being wasteful. But, we can get around the excessive space concerns by putting the page table in virtual memory, and letting the virtual memory system manage the memory for the page table.

4.Nested Page Table

Nested page tables can be implemented to increase the performance of hardware virtualization. By providing hardware support for page-table virtualization, the need to emulate is greatly reduced.

3. What is TLB. Explain it by writing an algorithm. Also mention the advantages and disadvantages of TLB.

ANS:

A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. It is a part of the chip's memory-management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache. A TLB may reside between the CPU and the CPU cache, between CPU cache and the main memory or between the different levels of the multi-level cache.

TLB ALGORITHM:

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else // TLB Miss
```

```

PTEAddr = PTBR + (VPN * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False) RaiseException(PROTECTION_FAULT)
else
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()

```

Advantages

The advantages of using TLB are-

- TLB reduces the effective access time.
- Only one memory access is required when TLB hit occurs.

Disadvantages

A major disadvantage of using TLB is-

- After some time of running the process, when TLB hits increases and process starts to run smoothly, a context switching occurs.
- The entire content of the TLB is flushed.
- Then, TLB is again updated with the currently running process.

4. With a linear page table, you need a single register to locate the page table, assuming that the hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?

ANS:

I believe the answer is still 1. Once the hardware has indexed into the first page-table it obtains the address of the next page-table (assuming the first is a valid entry). Requiring a register for each possible page table in a 32-bit virtual address space would be impractical.

5. Main memory and TLB are having accessing time of 220 ns and 120 ns respectively. The TLB hit ratio is about 98%. Calculate the effective time access.

ANS:

The percentage of times that a particular page number is found in the TLB is called the hit ratio. To find the effective access time, we weight each case by its probability:

$$\text{Effective access time} = 0.98 \times 120 + 0.02 \times 220$$

$$= 122 \text{ nanoseconds.}$$

In Lab Task:

1. Write a C program to simulate page replacement algorithms

- a) FIFO b) LRU

a) FIFO

ANS :

```
#include<stdio.h>

int main()
{
    int reference_string[10], page_faults = 0, m, n, s, pages, frames;
    printf("\nEnter Total Number of Pages:\t");
    scanf("%d", &pages);
    printf("\nEnter values of Reference String:\n");
    for(m = 0; m < pages; m++)
    {
        printf("Value No. [%d]:\t", m + 1);
        scanf("%d", &reference_string[m]);
    }
    printf("\nEnter Total Number of Frames:\t");
```

```

{
    scanf("%d", &frames);
}
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(reference_string[m] == temp[n])
        {
            s++;
            page_faults--;
        }
    }
    page_faults++;
    if((page_faults <= frames) && (s == 0))
    {
        temp[m] = reference_string[m];
    }
    else if(s == 0)

```

```

    {
        temp[(page_faults - 1) % frames] = reference_string[m];
    }
    printf("\n");
    for(n = 0; n < frames; n++)
    {
        printf("%d\t", temp[n]);
    }
}

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

return 0;
}

```

```

tushar@tusharsoni:~/Desktop$ gcc demo.c -lm
tushar@tusharsoni:~/Desktop$ ./a.out
Enter Total Number of Pages: 5
Enter Total Number of Frames: 4
Enter Values of Reference String
Enter Value No.[1]: 5
Enter Value No.[2]: 3
Enter Value No.[3]: 1
Enter Value No.[4]: 2
Enter Value No.[5]: 4

5      -1      -1      -1
5      3      -1      -1
5      3      1      -1
5      3      1      2
4      3      1      2
Page Hit: 0
tushar@tusharsoni:~/Desktop$

```

b) LRU

ANS:

```
#include<stdio.h>
main()
{
int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];
printf("Enter no of pages:");
scanf("%d",&n);
printf("Enter the reference string:");
for(i=0;i<n;i++)
    scanf("%d",&p[i]);
printf("Enter no of frames:");
scanf("%d",&f);
q[k]=p[k];
printf("\n\t%d\n",q[k]);
c++;
k++;

for(i=1;i<n;i++)
{
    c1=0;
    for(j=0;j<f;j++)
    {
        if(p[i]!=q[j])
            c1++;
    }
    if(c1==f)
    {
        c++;
        if(k<f)
        {
            q[k]=p[i];
            k++;
            for(j=0;j<k;j++)
                printf("\t%d",q[j]);
            printf("\n");
        }
        else
        {
            for(r=0;r<f;r++)
            {
                c2[r]=0;
                for(j=i-1;j<n;j--)
                {
```

```

        if(q[r]!=p[j])
            c2[r]++;
        else
            break;
    }
}
for(r=0;r<f;r++)
    b[r]=c2[r];
for(r=0;r<f;r++)
{
    for(j=r;j<f;j++)
    {
        if(b[r]<b[j])
        {
            t=b[r];
            b[r]=b[j];
            b[j]=t;
        }
    }
}
for(r=0;r<f;r++)
{
    if(c2[r]==b[0])
        q[r]=p[i];
    printf("\t%d",q[r]);
}
printf("\n");
}
}
printf("\nThe no of page faults is %d",c);
}

```

OUTPUT:

Enter no of pages:10

Enter the reference string:7 5 9 4 3 7 9 6 2 1

Enter no of frames:3

```

7
7   5
7   5   9
4   5   9
4   3   9
4   3   7
9   3   7
9   6   7
9   6   2

```

The no of page faults is 10

2. Designing a Virtual Memory Manager: write a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65,536$ bytes. Your program will read from a file containing logical addresses and, using a TLB as well as a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address.

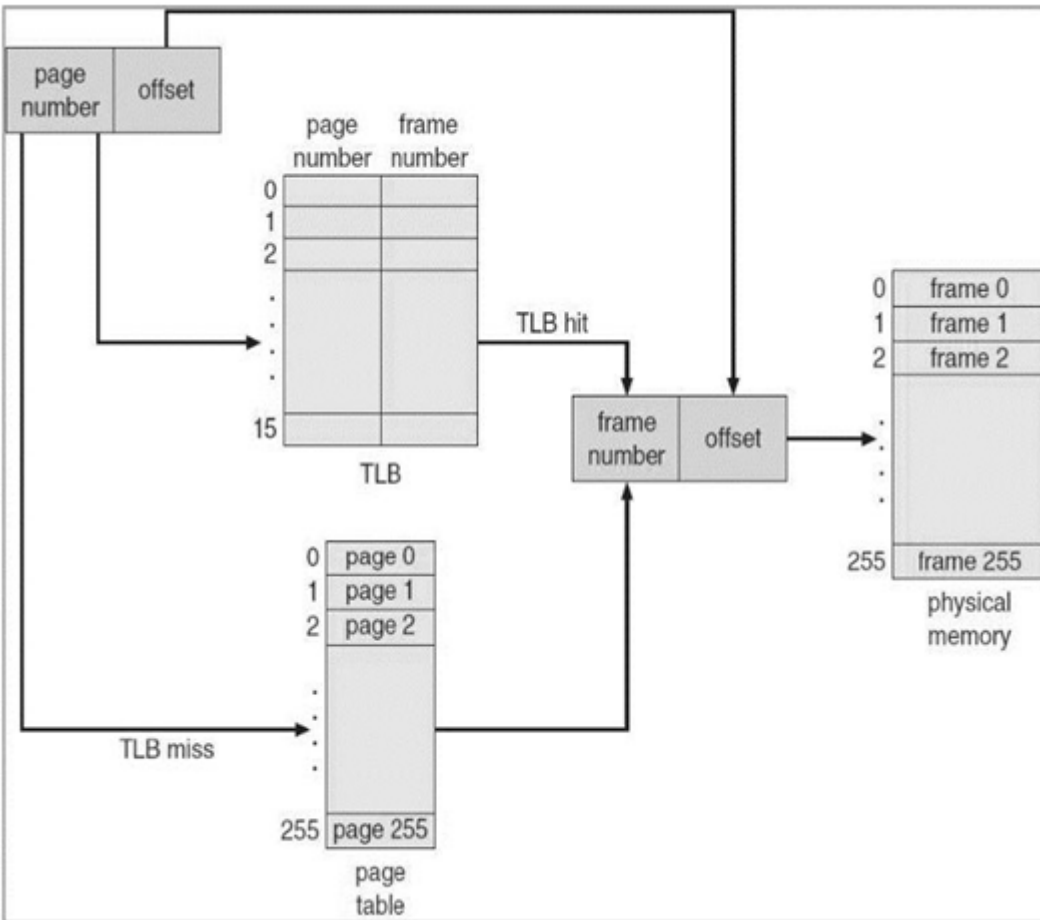
Specifics:

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into:

1. An 8-bit page number and
2. 8-bit page offset.

Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames * 256-byte frame size)



Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define SIZE 256
```

```
int main(int argc, char *argv[])
{
```

```
    int addressFile, backingStore;
    char ch, ct, input[1000], output;
    int logicalAddress, physicalAddress;
```

```
//    VARIABLE DECLARATIONS
//    file descriptors
```

```
int i=0, j=0;
```

```
//    LOGICAL MEMORY
```

```
int p;           //    page-number: used as an index into a page table
int d;           //    page-offset
```

```
//    PHYSICAL MEMORY
```

```
int f;           // frame-number: base address of each page in physical memory
char frames[SIZE*SIZE]; // physical memory where frame 0 is from frames[0] to
frames[255]
int frametable[SIZE]; //one free (-1) or allocated (0) flag entry for every frame 0 to
256
int start, current; //    var to handle page faults
int offset, pagefault=0;
int freeFrame=-1;
```

```
//    PAGE TABLE
```

```
int pagetable[SIZE]; //    page table where table[p] = f
for (j=0;j<SIZE;j++) //    flushing page table, frametable
{
    pagetable[j] = -1;
    frametable[j] = -1;
}
```

```
//    READING LOGICAL ADDRESS FROM FILE
```

```
addressFile = open(argv[1],O_RDONLY);
backingStore = open("BACKING_STORE.txt",O_RDONLY);

if(addressFile != -1)
{
while(read(addressFile, &ch, sizeof(char)) != 0) // read() returns the number of
bytes read and 0 for end of file (EOF)
{
    if(ch != '\n')
    {
        input[i] = ch;
        i++;
    }
    else
    {

```

```

logicalAddress = atoi(input);           //EXTRACT PAGE NUMBER AND OFFSET
p = (logicalAddress & 0x0000ff00UL) >> 8;
d = (logicalAddress & 0x000000ffUL);
printf("\nlogicalAddress: %d, p: %d, d: %d", logicalAddress,p,d);

```

//ADDRESS TRANSLATION THROUGH PAGE TABLE

//pagetable-hit, obtain frame number

```

if(pagetable[p] != -1){

```

```

    f = pagetable[p];

```

//locate free frame (-1) in physical memory

```

    physicalAddress = (f * SIZE) + d;

```

```

    printf("\nphysicalAddress: %d, f: %d", physicalAddress,f);

```

```

}

```

//pagetable-miss, page-fault

```

else

```

```

{

```

```

    pagefault++;

```

```

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

```

```

    {

```

```

        if(frameTable[j]==-1)

```

```

        {

```

```

            freeFrame = j;

```

```

            break;

```

```

        }

```

```

    }

```

//read page from backing-store into the available frame in the physical memory

```

    if(backingStore != -1)

```

```

    {

```

```

        offset=0;

```

```

        start = SIZE * p;

```

```

        current=lseek(backingStore, start, SEEK_SET);

```

```

        while((offset < SIZE)&&(current))

```

```

        {

```

```

            current = read(backingStore, &ct, sizeof(char));

```

```

            frames[freeFrame*offset] = ct;

```

```

            offset++;

```

```

        }

```

```

    }

```

```

        else
        {
            printf("Backing-Store Does not exist!");
            close(backingStore);
            close(addressFile);
            return 0;
        }
//update pagetable, frametable
    pagetable[p] = freeFrame;
    frametable[freeFrame] = 0;

    physicalAddress = (freeFrame * SIZE) + d;
    printf("\nphysicalAddress: %d, freeFrame: %d", physicalAddress,
freeFrame);
}
// READ CHAR STORED AT THE PHYSICAL ADDRESS
    output = frames[physicalAddress];
    printf("\nByte value stored at physicalAddress %d: %c\n",physicalAddress,
output);
    memset(input,0,sizeof(input));
    i=0;

}
}
printf("\nTotal Page Faults: %d",pagefault);
}
else
    printf("Addresses File Does not exist!");
    close(backingStore);
    close(addressFile);
    return 0;
}

```

SKILL EXERCISE 6

1. Write a C program to simulate page replacement algorithms
 - a) Optimal
 - b) LFU

ANS

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

```

#include<conio.h>

Int main()
{
    int i,j,k,l,m,n,p,c=0,s;
    int a[20],b[20],q,max;
    printf("enter no. of reference string: ");
    scanf("%d",&n);
    printf("enter size of frame: ");
    scanf("%d",&m);
    printf("enter the elements of ref. string: \n");
    for(i=0;i<n;i++)
scanf("%d",&a[i]);
    for(j=0;j<m;j++)
        b[j]=-1; //initialize all frame elements with -1
        for(i=0;i<n;i++)
        {

            for(k=0;k<m;k++)
                if(b[k]==a[i])
                    goto here;

            for(j=0;j<m;j++)
            {
                if(b[j]==-1) //check if element already present in frame,if true then no
page fault.

```

```

        {
            b[j]=a[i];
            c++;
            goto here;
        }
    }
if(j==m)
{
    l=i+1,max=0;
    for(j=0;j<m;j++)
    {
        for(s=l;s<n;s++)
        {
            if(a[s]==b[j])
            {
                if(s>max)
                {
                    max=s;
                    p=j;
                }
                break;
            }
        }
        if(s==n)
        {
            max=s;

```

```

        p=j;
    }
}
}

b[p]=a[i];

c++;

here: printf("\n\n");

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

printf(" %d",b[k]);

}

printf("\n No of page fault is:%d",c);

getch();

}

```

b))

ANS:

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

```

int main()
{
    int total_frames, total_pages, hit = 0;
    int pages[25], frame[10], arr[25], time[25];
    int m, n, page, flag, k, minimum_time, temp;
    printf("Enter Total Number of Pages:\t");
    scanf("%d", &total_pages);
    printf("Enter Total Number of Frames:\t");
    scanf("%d", &total_frames);
    for(m = 0; m < total_frames; m++)
    {
        frame[m] = -1;
    }
    for(m = 0; m < 25; m++)

```

```

{
    arr[m] = 0;
}
printf("Enter Values of Reference String\n");
for(m = 0; m < total_pages; m++)
{
    printf("Enter Value No.[%d]:\t", m + 1);
    scanf("%d", &pages[m]);
}
printf("\n");
for(m = 0; m < total_pages; m++)
{
    arr[pages[m]]++;
    time[pages[m]] = m;
    flag = 1;
    k = frame[0];
    for(n = 0; n < total_frames; n++)
    {
        if(frame[n] == -1 || frame[n] == pages[m])
        {
            if(frame[n] != -1)
            {
                hit++;
            }
            flag = 0;
            frame[n] = pages[m];
            break;
        }
        if(arr[k] > arr[frame[n]])
        {
            k = frame[n];
        }
    }
}
if(flag)
{
    minimum_time = 25;
    for(n = 0; n < total_frames; n++)
    {
        if(arr[frame[n]] == arr[k] && time[frame[n]] < minimum_time)

```



```

        {
            temp = n;
            minimum_time = time[frame[n]];
        }
    }
    arr[frame[temp]] = 0;
    frame[temp] = pages[m];
}
for(n = 0; n < total_frames; n++)
{
    printf("%d\t", frame[n]);
}
printf("\n");
}
printf("Page Hit:\t%d\n", hit);
return 0;
}

```

OUTPUT: LFU

```

tushar@tusharsoni:~/Desktop$ gcc demo.c -lm
tushar@tusharsoni:~/Desktop$ ./a.out
Enter Total Number of Pages: 5
Enter Total Number of Frames: 4
Enter Values of Reference String
Enter Value No.[1]: 5
Enter Value No.[2]: 3
Enter Value No.[3]: 1
Enter Value No.[4]: 2
Enter Value No.[5]: 4

5      -1      -1      -1
5      3      -1      -1
5      3      1      -1
5      3      1      2
4      3      1      2
Page Hit: 0
tushar@tusharsoni:~/Desktop$

```

2. Write an Algorithm to demonstrate Multi-level Page Table Control Flow??

ANS:

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
    else // TLB Miss
        // first, get page directory entry
        PDIndex = (VPN & PD_MASK) >> PD_SHIFT
        PDEAddr = PDBR + (PDIndex * sizeof(PDE))
        PDE = AccessMemory(PDEAddr)
        if (PDE.Valid == False)
            RaiseException(SEGMENTATION_FAULT)
        else
            // PDE is valid: now fetch PTE from page table
            PTIndex = (VPN & PT_MASK) >> PT_SHIFT
            PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
            PTE = AccessMemory(PTEAddr)
            if (PTE.Valid == False)
                RaiseException(SEGMENTATION_FAULT)
            else if (CanAccess(PTE.ProtectBits) == False)
                RaiseException(PROTECTION_FAULT)
            else
                TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
        RetryInstruction()
```

HOME ASSIGNMENT:-

1. Consider a reference string 0,2,1,6,4,0,1,0,3,1,2,1 and the size of frame is 4. Implement FIFO, LRU and OPTIMAL page replacement algorithms and find the no. of page faults and page fault rate. Which is the best algorithm?
2. Consider a reference string 1,2,3,2,1,5,2,1,6,2,5,6,3,1,3,6,1,2,4,3 and the size of frame is 3. Implement FIFO, LRU and OPTIMAL page replacement algorithms and find the no. of page faults and page fault rate. Which is the best algorithm?
3. Consider a reference string 1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6 and the size of frame is 3. Implement FIFO, LRU and OPTIMAL page replacement algorithms and find the
 - i) no. of page faults
 - ii) page fault rate
 - iii) page hit
- 4). Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4. Implement FIFO, LRU and OPTIMAL page replacement algorithms and find the
 - i) no. of page faults
 - ii) page fault rate
 - iii) page hit

REVIEW – PROBLEM SET II

1. Provide one advantage of using the slab allocator in Linux to allocate kernel objects, instead of simply allocating them from a dynamic memory heap.

2. Consider a program that memory maps a large file, and accesses bytes in the first page of the file. Now, a student runs this program on several machines running different versions of Linux, and finds that the actual physical memory consumed by the process (RSS or resident set size) varies from OS to OS. Provide one reason to explain this observation.

3. In a 32-bit architecture machine running Linux, for every physical memory address in RAM, there are at least 2 virtual addresses pointing to it. That is, every physical address is mapped at least twice into the virtual address space of some set of processes. [T/F]

4. Consider a system with N bytes of physical RAM, and M bytes of virtual address space per process. Pages and frames are K bytes in size. Every page table entry is P bytes in size, accounting for the extra flags required and such. Calculate the size of the page table of a process.

5. The memory addresses generated by the CPU when executing instructions of a process are called logical addresses. [T/F]

6. When a C++ executable is run on a Linux machine, the kernel code is part of the executable generated during the compilation process. [T/F]

7. When a C++ executable is run on a Linux machine, the kernel code is part of the virtual address space of the running process. [T/F]

8. Consider a process with 9 logical pages, out of which 3 pages are mapped to physical frames. The process accesses one of its 9 pages randomly. What is the probability that the access results in a TLB hit and a subsequent page fault?

9. Consider a system with paging-based memory management, whose architecture allows for a 4GB virtual address space for processes. The size of logical pages and physical frames is 4KB. The system has 8GB of physical RAM. The system allows a maximum of 1K (=1024) processes to run concurrently. Assuming the OS uses hierarchical paging, calculate the maximum memory space required to store the page tables of all processes in the system. Assume that each page table entry requires an additional 10 bits (beyond the frame number) to store various flags. Assume page table entries are rounded up to the nearest byte. Consider the memory required for both outer and inner page tables in your calculations.

10. Consider a simple system running a single process. The size of physical frames and logical pages is 16 bytes. The RAM can hold 3 physical frames. The virtual addresses of the process are 6 bits in size. The program generates the following 20 virtual address references as it runs on the CPU: 0, 1, 20, 2, 20, 21, 32, 31, 0, 60, 0, 0, 16, 1, 17, 18, 32, 31, 0, 61. (Note: the 6-bit addresses are shown in decimal here.) Assume that the physical frames in RAM are initially empty and do not map to any logical page.

(a) Translate the virtual addresses above to logical page numbers referenced by the process. That is, write down the reference string of 20 page numbers corresponding to the virtual address accesses above. Assume pages are numbered starting from 0, 1, ...

(b) Calculate the number of page faults generated by the accesses above, assuming a FIFO page replacement algorithm. You must also correctly point out which page accesses in the reference string shown by you in part (a) are responsible for the page faults.

(c) Repeat (b) above for the LRU page replacement algorithm.

(d) What would be the lowest number of page faults achievable in this example, assuming an optimal page replacement algorithm were to be used? Repeat (b) above for the optimal algorithm.

11. Consider a system with only virtual addresses, but no concept of virtual memory or demand paging. Define total memory access time as the time to access code/data from an address in physical memory, including the time to resolve the address (via the TLB or page tables) and the actual physical memory access itself. When a virtual address is resolved by the TLB, experiments on a machine have empirically observed the total memory access time to be (an approximately constant value of) t_h . Similarly, when the virtual address is not in the TLB, the total memory access time is observed to be t_m . If the average total memory access time of the system (averaged across all memory accesses, including TLB hits as well as misses) is observed to be t_x , calculate what fraction of memory addresses are resolved by the TLB. In other words, derive an expression for the TLB hit rate in terms of t_h , t_m , and t_x . You may assume $t_m > t_h$.

12. Consider a system with a 6 bit virtual address space, and 16 byte pages/frames. The mapping from virtual page numbers to physical frame numbers of a process is (0,8), (1,3), (2,11), and (3,1). Translate the following virtual addresses to physical addresses. Note that all addresses are in decimal. You may write your answer in decimal or binary.

(a) 20

(b) 40

13. Consider a system with several running processes. The system is running a modern OS that uses virtual addresses and demand paging. It has been empirically observed that the memory access times in the system under various conditions are: t_1 when the logical memory address is found in TLB cache, t_2 when the address is not in TLB but does not cause a page fault, and t_3 when the address results in a page fault. This memory access time includes all overheads like page fault servicing and logical-to-physical address translation. It has been observed that, on an average, 10% of the logical address accesses result in a page fault. Further, of the remaining virtual address accesses, two-thirds of them can be translated using the TLB cache, while one-third require walking the page tables. Using the information provided above, calculate the average expected memory access time in the system in terms of t_1 , t_2 , and t_3 .

14. Consider a system where each process has a virtual address space of 2^v bytes. The physical address space of the system is 2^p bytes, and the page size is 2^k bytes. The size of each page table entry is 2^e bytes. The system uses hierarchical paging with l levels of page tables, where the page table entries in the last level point to the actual physical pages of the process. Assume $l \geq 2$. Let v_0 denote the number of (most significant) bits of the virtual address that are used as an index into the outermost page table during address translation.

(a) What is the number of logical pages of a process?

(b) What is the number of physical frames in the system?

(c) What is the number of PTEs that can be stored in a page?

(d) How many pages are required to store the innermost PTEs?

(e) Derive an expression for l in terms of v , p , k , and e .

(f) Derive an expression for v_0 in terms of l , v , p , k , and e .

15. Consider an operating system that uses 48-bit virtual addresses and 16KB pages. The system uses a hierarchical page table design to store all the page table entries of a process, and each page table entry is 4 bytes in size. What is the total number of pages that are required to store the page table entries of a process, across all levels of the hierarchical page table?

CO 3

WEEK - 7 (THREAD API)

Prerequisite:

- Basic functionality of pthreads.
- Basic functionality of semaphores.
- Complete idea of thread creation and thread termination.
- Readers and writers problems
- Producer and consumer problems
- Bankers algorithm
- Concepts of Synchronization
- Basic concepts of mutex and semaphores.

Pre-lab task:

1.A) Analyze the purpose of thread and write description for the following pthread library functions

1. **pthread_create()** :- create a new thread

Description: The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

2. **pthread_exit()**:-terminate calling thread

Description: The pthread_exit() function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread_join.

3. **pthread_join()**:-join with a terminated thread

Description: The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

4. **pthread_self()**:-obtain ID of the calling thread

Description: The `pthread_self()` function returns the ID of the calling thread. This is the same value that is returned in `*thread` in the `pthread_create(3)` call that created this thread.

5. **pthread_cancel()**:-send a cancellation request to a thread

Description: The `pthread_cancel()` function sends a cancellation request to the thread `thread`. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type.

6. **pthread_detach()**:-detach a thread

Description: The `pthread_detach()` function marks the thread identified by `thread` as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

7. **pthread_equal()**:-compare thread IDs

Description: The `pthread_equal()` function compares two thread identifiers.

8. **pthread_mutex_init()**:-initialise or destroy a mutex

Description: The `pthread_mutex_init()` function initialises the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

9. **pthread_mutex_destroy()**:-destroy and initialize a mutex

Description: The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. An implementation may cause `pthread_mutex_destroy()` to set the object referenced by `mutex` to an invalid value. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

10. **pthread_mutex_lock()**:-Lock a mutex

Description: The *pthread_mutex_lock()* function locks the mutex object referenced by *mutex*. If the mutex is already locked, then the calling thread blocks until it has acquired the mutex. When the function returns, the mutex object is locked and owned by the calling thread.

11. **pthread_mutex_trylock()**:-Attempt to lock a mutex

Description: The *pthread_mutex_trylock()* function attempts to lock the mutex *mutex*, but doesn't block the calling thread if the mutex is already locked.

12. **pthread_mutex_unlock()**:-Unlock a mutex

Description: The *pthread_mutex_unlock()* function unlocks the mutex *mutex*. The mutex should be owned by the calling thread. If there are threads blocked on the mutex then the highest priority waiting thread is unblocked and becomes the next owner of the mutex.

1.B)Write description for the following calls and match them with their functions

Sl no	call	functions
1	sem_post()	() (a)wait on a named or unnamed semaphore
2	sem_wait()	() (b)remove a named semaphore(REAL TIME)
3	sem_open()	() (c) get the value of a semaphore(REAL TIME)
4	sem_getvalue()	() (d)initialise and open a named semaphore.
5	sem_unlink()	() (e) unlock a semaphore (REALTIME).

1. **sem_post()**- unlock a semaphore (REALTIME).

Description:- *sem_post()* increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a *sem_wait(3)* call will be woken up and proceed to lock the semaphore.

2. **sem_wait()**- wait on a named or unnamed semaphore.

Description:- *sem_wait()* decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

3. **sem_open()**- initialize and open a named semaphore (REALTIME)

Description:- sem_open() creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by name.

4. **sem_getvalue()**- get the value of a semaphore (**REALTIME**)

Description:- sem_getvalue() places the current value of the semaphore pointed to sem into the integer pointed to by sval.

5. **sem_unlink()**- remove a named semaphore (**REALTIME**)

Description:- sem_unlink() removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

2. Write a program to create 5 pthreads and display Hello world. Main thread should wait until new threads are terminated.

Ans:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <pthread.h>

void* func_one(void* ptr)

{

int *i=(int *)ptr;

printf("Hello World !!!! by Thread ID : %d, Thread Number : %d\n",pthread_self(),*i+1);

}

int main()

{

pthread_t thread_one[5];

int i=0;

int a[5];
```

```

// creating thread one

for(i=0;i<5;i++)
{
    a[i]=i;

    pthread_create(&thread_one[i], NULL, func_one, (void *)&a[i]);
}


for(i=0;i<5;i++)

pthread_join(thread_one[i], NULL);

// wait for thread two

printf("done\n");
}

```

3. Analyze the following program and write the output ?

```

#include<stdio.h>

#include<pthread.h>

int main()
{
    printf("main start \n");

    printf("https://www.sanfoundry.com/operating-system-mcqs-critical-section-
problem/www.Sanfoundry\n");

    printf("calling exit \n ");

    pthread_exit(" ");

    printf("Linux");

    return 0;
}

```

```
}
```

Output:

main start

<https://www.sanfoundry.com/operating-system-mcqs-critical-section-problem/www.Sanfoundry>

calling exit

4. What is the output of this program

```
#include<stdio.h>

#include<pthread.h>

void *fun_t(void *arg);

void *fun_t(void *arg)
{
    pthread_exit("Bye");
}

int main()
{
    pthread_t pt;
    void *res_t;
    int ret;
    ret = pthread_join(pt,&res_t);
    printf("%d\n",ret);
    return 0;
}
```

Output:

```
[root@localhost sanfoundry]#gcc-osansan.c-lpthread
[root@localhost sanfoundry]#./san
3
[root@localhost sanfoundry]#
```

In-lab task :

1.Illustrate how semaphores are used for thread synchronization, print the counter variable upon each increment which is in the critical section.(how Two threads update a global shared variable with synchronization)

Ans:

//Semaphores:

```
#include<stdio.h>
#include<string.h>
#include<semaphore.h>
#include<stdlib.h>
#include<unistd.h>

sem_t m;
pthread_t tid[2];
unsigned long int counter=0;
void* trythis(void *arg)
{
    sem_wait(&m);
    int *i=(int *)arg;
    printf("\n Thread Execution started with a counter value : %ld\n",counter);
    printf("Thread Execution started by thread %d\n",*i);
    for(int j=0; j<10;j++){
        counter =counter +1;
        printf("counter : %d\n",counter);
    }
    printf("\n Thread Execution completed with counter value : %ld \n", counter);
    sem_post(&m);
    return NULL;
}

int main(void)
{

```



```

int i = 0;

int error;

sem_init(&m,0,1);

int a[2]={1,2};

while(i < 2)
{
    error = pthread_create(&(tid[i]), NULL, &trythis, (void *)&a[i]);

    if (error != 0)

        printf("\nThread can't be created :[%s]", strerror(error));

    i++;
}

pthread_join(tid[1], NULL);

pthread_join(tid[0], NULL);

sem_destroy(&m);

return 0;
}

```

2. Implement parallel global sum (sum of n natural numbers) using pthreads.

ANS:

```

#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
#include<stdlib.h>
void *globalsum(void*);
int totalsum=0;
pthread_mutex_t mvar=PTHREAD_MUTEX_INITIALIZER;
int main()
{
    int inumber,icount;
    pthread_t tid;
    printf("Enter up to how many numbers you want to sum:");
    scanf("%d",&inumber);
    pthread_create(&tid,NULL,*globalsum,(void*)&inumber);
    for(icount=1;icount<=inumber;icount+=2)
    {
        pthread_mutex_lock(&mvar);

```

```

    totalsum+=icount;
    pthread_mutex_unlock(&mvar);
}
pthread_join(tid,NULL);
printf("TotalSum:%d\n",totalsum);
return 0;
}
void *globalsum(void *arg)
{
    int *inumber,icount;
    inumber=(int*)arg;
    for(icount=2;icount<=*inumber;icount+=2)
    {
        pthread_mutex_lock(&mvar);
        totalsum+=icount;
        pthread_mutex_unlock(&mvar);
    }
    pthread_exit(NULL);
}

```

3. write a c program to implement readers writers problem using semaphores,mutex,threads,system programming.

Ans:

```

#include<stdio.h>

#include<pthread.h>

#include<semaphore.h>

sem_t mutex;

sem_t db;

int readercount=0;

pthread_t reader1,reader2,writer1,writer2;

void *reader(void *);

void *writer(void *);

main()

```

```

{

sem_init(&mutex,0,1);

sem_init(&db,0,1);

while(1)

{

pthread_create(&reader1,NULL,reader,"1");

pthread_create(&reader2,NULL,reader,"2");

pthread_create(&writer1,NULL,writer,"1");

pthread_create(&writer2,NULL,writer,"2");

}

}

void *reader(void *p)

{

printf("previous value %dn",mutex);

sem_wait(&mutex);

printf("Mutex acquired by reader %dn",mutex);

readercount++;

if(readercount==1) sem_wait(&db);

sem_post(&mutex);

printf("Mutex returned by reader %dn",mutex);

printf("Reader %s is Readingn",p);

//sleep(3);

sem_wait(&mutex);

printf("Reader %s Completed Readingn",p);

```

```

readercount--;

if(readercount==0) sem_post(&db);

sem_post(&mutex);

}

void *writer(void *p)

{

printf("Writer is Waiting n");

sem_wait(&db);

printf("Writer %s is writingn ",p);

sem_post(&db);

//sleep(2);

}

```

4. Write A C Program To Implement Banker's Algorithm for avoiding Deadlocks in Operating Systems.

ANS:

```

#include <stdio.h>
int main()
{
    int count = 0, m, n, process, temp, resource;
    int allocation_table[5] = {0, 0, 0, 0, 0};
    int available[5], current[5][5], maximum_claim[5][5];
    int maximum_resources[5], running[5], safe_state = 0;
    printf("\nEnter The Total Number Of Processes:\t");
    scanf("%d", &process);
    for(m = 0; m < process; m++)
    {
        running[m] = 1;
        count++;
    }
    printf("\nEnter The Total Number Of Resources To Allocate:\t");
    scanf("%d", &resource);
    printf("\nEnter The Claim Vector:\t");
    for(m = 0; m < resource; m++)

```

```

{
    scanf("%d", &maximum_resources[m]);
}
printf("\nEnter Allocated Resource Table:\n");
for(m = 0; m < process; m++)
{
    for(n = 0; n < resource; n++)
    {
        scanf("%d", &current[m][n]);
    }
}
printf("\nEnter The Maximum Claim Table:\n");
for(m = 0; m < process; m++)
{
    for(n = 0; n < resource; n++)
    {
        scanf("%d", &maximum_claim[m][n]);
    }
}
printf("\nThe Claim Vector \n");
for(m = 0; m < resource; m++)
{
    printf("\t%d ", maximum_resources[m]);
}
printf("\n The Allocated Resource Table\n");
for(m = 0; m < process; m++)
{
    for(n = 0; n < resource; n++)
    {
        printf("\t%d", current[m][n]);
    }
    printf("\n");
}
printf("\nThe Maximum Claim Table \n");
for(m = 0; m < process; m++)
{
    for(n = 0; n < resource; n++)
    {
        printf("\t%d", maximum_claim[m][n]);
    }
    printf("\n");
}
for(m = 0; m < process; m++)
{
    for(n = 0; n < resource; n++)
    {
        allocation_table[n] = allocation_table[n] + current[m][n];
    }
}

```

```

}
printf("\nAllocated Resources \n");
for(m = 0; m < resource; m++)
{
    printf("\t%d", allocation_table[m]);
}
for(m = 0; m < resource; m++)
{
    available[m] = maximum_resources[m] - allocation_table[m];
}
printf("\nAvailable Resources:");
for(m = 0; m < resource; m++)
{
    printf("\t%d", available[m]);
}
printf("\n");
while(count != 0)
{
    safe_state = 0;
    for(m = 0; m < process; m++)
    {
        if(running[m])
        {
            temp = 1;
            for(n = 0; n < resource; n++)
            {
                if(maximum_claim[m][n] - current[m][n] > available[n])
                {
                    temp = 0;
                    break;
                }
            }
        }
        if(temp)
        {
            printf("\nProcess %d Is In Execution \n", m + 1);
            running[m] = 0;
            count--;
            safe_state = 1;
            for(n = 0; n < resource; n++)
            {
                available[n] = available[n] + current[m][n];
            }
            break;
        }
    }
}
if(!safe_state)
{

```

```

        printf("\nThe Processes Are In An Unsafe State \n");
        break;
    }
    else
    {
        printf("\nThe Process Is In A Safe State \n");
        printf("\nAvailable Vector\n");
        for(m = 0; m < resource; m++)
        {
            printf("\t%d", available[m]);
        }
        printf("\n");
    }
}
return 0;
}

```

SKILL EXERCISE 7:

1. Write UNIX system program to implement producer-consumer problem in which multiple producer threads fill an array that is processed by one consumer thread.

Version 1: the consumer started only after the producers were finished, and we were able to solve this synchronization.

Ans:

```

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

#define BUF_SIZE 3    /* Size of shared buffer */

int buffer[BUF_SIZE]; /* shared buffer */
int add = 0;          /* place to add next element */
int rem = 0;          /* place to remove next element */
int num = 0;          /* number elements in buffer */

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; /* mutex lock for
buffer */
pthread_cond_t c_cons = PTHREAD_COND_INITIALIZER; /* consumer waits on
this cond var */
pthread_cond_t c_prod = PTHREAD_COND_INITIALIZER; /* producer waits on
this cond var */

void *producer (void *param);
void *consumer (void *param);

```

```

int main(int argc, char *argv[]) {

    pthread_t tid1, tid2; /* thread identifiers */
    int i;

    /* create the threads; may be any number, in general */
    if(pthread_create(&tid1, NULL, producer, NULL) != 0) {
        fprintf(stderr, "Unable to create producer thread\n");
        exit(1);
    }

    if(pthread_create(&tid2, NULL, consumer, NULL) != 0) {
        fprintf(stderr, "Unable to create consumer thread\n");
        exit(1);
    }

    /* wait for created thread to exit */
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Parent quitting\n");

    return 0;
}

/* Produce value(s) */
void *producer(void *param) {

    int i;
    for (i=1; i<=10; i++) {

        /* Insert into buffer */
        pthread_mutex_lock (&m);
        if (num > BUF_SIZE) {
            exit(1); /* overflow */
        }

        while (num == BUF_SIZE) { /* block if buffer is full */
            pthread_cond_wait (&c_prod, &m);
        }

        /* if executing here, buffer not full so add element */
        buffer[add] = i;
        add = (add+1) % BUF_SIZE;
        num++;
        pthread_mutex_unlock (&m);
    }
}

```



```

        pthread_cond_signal (&c_cons);
        printf ("producer: inserted %d\n", i);
    }

    printf("producer quitting\n");
    return 0;
}

/* Consume value(s); Note the consumer never terminates */
void *consumer(void *param) {

    int i,j;

    for(j=1;j<=10;j++) {

        pthread_mutex_lock (&m);
        if (num < 0) {
            exit(1);
        } /* underflow */

        while (num == 0) { /* block if buffer empty */
            pthread_cond_wait (&c_cons, &m);
        }

        /* if executing here, buffer not empty so remove element */
        i = buffer[rem];
        rem = (rem+1) % BUF_SIZE;
        num--;
        pthread_mutex_unlock (&m);

        pthread_cond_signal (&c_prod);
        printf ("Consume value %d\n", i);
    }
    printf ("consumer quitting\n");
    return 0;
}

```

2. Implement dining philosopher problem using semaphores and threads.

Suppose there are N philosophers meeting around a table, eating spaghetti and talking about philosophy. Now let us discuss the problem. There are only N forks available such that only one fork between each philosopher. Since there are only 5 philosophers and each one requires 2 forks to eat, we need to formulate an algorithm which ensures that the utmost number of philosophers can eat spaghetti at once.

ANS:

```
# include<stdio.h>

# include<pthread.h>

# include<stdlib.h>

# include<unistd.h>

# include<ctype.h>

# include<sys/types.h>

# include<sys/wait.h>

# include<semaphore.h>

# include<sys/sem.h>

sem_t chopstick[100];

int n;

void *thread_func(int no)

{

    int i,iteration=5;

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

    {

        sem_wait(&chopstick[no]);

        sem_wait(&chopstick[(no+1)%n]);

        printf("\nPhilosopher %d -> Begin eating",no);

        sleep(1);

        printf("\nPhilosopher %d -> Finish eating\n",no);

        sem_post(&chopstick[no]);
```

```

sem_post(&chopstick[(no+1)%n]);

}

pthread_exit(NULL);

}

int main()

{

int i,res;

pthread_t a_thread[100];

printf("\nEnter the number of philosopher :");

scanf("%d",&n);

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

{

res=sem_init(&chopstick[i],0,0);

if(res==-1)

{

perror("semaphore initialization failed");

exit(1);

}

}

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

{

res=pthread_create(&a_thread[i],NULL,thread_func,(int*) i);

```

```

if(res!=0)

{

perror("semaphore creation failed");

exit(1);

}

sem_post(&chopstick[i]);

}

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

{

res=pthread_join(a_thread[i],NULL);

if(res!=0)

{

perror("semaphore join failed");

exit(1);

}

}

printf("\n \n thread join succesfull\n");

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

{

res=sem_destroy(&chopstick[i]);

if(res==-1)

{

```

```

perror("semaphore destruction failed");

exit(1);

}

}

exit(0);

}

```

WEEK - 8 (CONCURRENCY)

Prerequisite:

- Complete Idea on deadlock system and its process
- Basic idea on concurrent data structures Linked lists and queues
- Basic idea on concurrent hash tables
- Producer and consumer problems
- Conditional variables
- Signals in UNIX

Pre-lab task:

1. Write description for the following system calls

1. `pthread_cond_init (condition,attr):-`

The function *pthread_cond_init()* initialises the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialisation, the state of the condition variable becomes initialised.

2. `pthread_cond_destroy (condition):-`

The *pthread_cond_destroy()* function shall destroy the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause *pthread_cond_destroy()* to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be reinitialized using *pthread_cond_init()*; the results of otherwise referencing the object after it has been destroyed are undefined.

3. pthread_condattr_init (attr):-

The function *pthread_condattr_init()* initialises a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation.

4. pthread_condattr_destroy (attr):-

The *pthread_condattr_destroy()* function shall destroy a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause *pthread_condattr_destroy()* to set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be reinitialized using *pthread_condattr_init()*; the results of otherwise referencing the object after it has been destroyed are undefined.

2 .Write a UNIX system program to implement concurrent Linked List

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

typedef struct __node_t {
int key;
struct __node_t *next;
} node_t;
// basic list structure (one used per list)
typedef struct __list_t {
node_t *head;
pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
L->head = NULL;
pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
pthread_mutex_lock(&L->lock);
node_t *new = malloc(sizeof(node_t));
if (new == NULL) {
perror("malloc");
pthread_mutex_unlock(&L->lock);
return -1; // fail
}
new->key = key;
```

```

new->next = L->head;
L->head = new;
pthread_mutex_unlock(&L->lock);
return 0; // success
}

```

```

int List_Lookup(list_t *L, int key) {
pthread_mutex_lock(&L->lock);
node_t *curr = L->head;
while (curr) {
if (curr->key == key) {
pthread_mutex_unlock(&L->lock);
return 0; // success
}
curr = curr->next;
}
pthread_mutex_unlock(&L->lock);
return -1; // failure
}

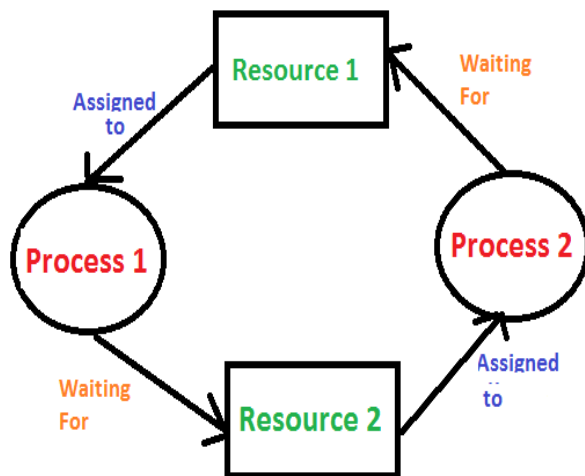
```

```

int main(){
    list_t *L=(struct __list_t*)malloc(sizeof(struct __list_t));
    List_Init(L);
    List_Insert(L,10);
    List_Insert(L,20);
    List_Insert(L,30);
    List_Insert(L,40);
    printf("%d",List_Lookup(L,20);
}

```

3.Explain briefly about deadlock and mention what are the methods to handle deadlocks.



Sol:

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)

Hold and Wait: A process is holding at least one resource and waiting for resources.

No Preemption: A resource cannot be taken from a process unless the process releases the resource.

Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock

There are three ways to handle deadlock

1) **Deadlock prevention or avoidance:** The idea is to not let the system into deadlock state. One can zoom into each category individually, Prevention is done by negating one of the above mentioned necessary conditions for deadlock.

Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) **Deadlock detection and recovery:** Let deadlock occur, then do preemption to handle it once occurred.

3) **Ignore the problem all together:** If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

4 . Write UNIX system program that handles SIGINT and SIGTSTP generated from terminal and requires two [Ctrl-c]'s to terminate.

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

/* Signal Handler for SIGINT */
int c;
void sigintHandler(int sig_num)
{
    /* Reset handler to catch SIGINT next time.
       Refer http://en.cppreference.com/w/c/program/signal */
    if(sig_num==20){
        printf("Ctrl+Z will not work please try Ctrl+C twice!!\n");
    }
    if(c==0&&sig_num==2){
        printf("\n You need to give that once more !!!!! (Ctrl+C) \n");
        sleep(1);
        c=1;
    }
    else if(c==1&&sig_num==2)
        exit(1);
}

int main ()
{

    signal(SIGINT, sigintHandler);
    signal(SIGTSTP,sigintHandler);

    /* Infinite loop */
    while(1)
    {
        printf("Signal Handling \n");
        sleep(1);
    }
    return 0;
}
```

In lab task:

1) Write a Unix System program to make parent Waiting for Child using a condition variable.

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

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

```
int done = 0;
```

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```
void thr_exit() {
```

```
pthread_mutex_lock(&m);
```

```
done = 1;
```

```
pthread_cond_signal(&c);
```

```
pthread_mutex_unlock(&m);
```

```
}
```

```
void *child(void *arg) {
```

```
printf("child\n");
```

```
thr_exit();
```

```
return NULL;
```

```
}
```

```
void thr_join() {
```

```
pthread_mutex_lock(&m);
```

```
while (done == 0)
```

```
pthread_cond_wait(&c, &m);
```

```

pthread_mutex_unlock(&m);

}

int main(int argc, char *argv[]) {

printf("parent: begin\n");

pthread_t p;

pthread_create(&p, NULL, child, NULL);

thr_join();

printf("parent: end\n");

return 0;

}

```

- 2) Write a UNIX system program to Solve Producer Consumer problem using POSIX semaphores. Three conditions must be maintained by the code when the shared buffer is considered as a circular buffer:
- a) The consumer cannot try to remove an item from the buffer when the buffer is empty.
 - b) The producer cannot try to place an item into the buffer when the buffer is full.
 - c) Shared variables may describe the current state of the buffer (indexes, counts, linked list pointers, etc.), so all buffer manipulations by the producer and consumer must be protected to avoid any race conditions.

```

#include <stdio.h>
#include <stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
int n;

```

OS LAB EXPERIMENTS CODES

```

void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.Producer\n2.Consumer\n3.Exit");

```

```

while(1)
{
printf(&quot;\nEnter your choice:&quot;);
scanf(&quot;%d&quot;,&amp;n);
switch(n)
{
case 1: if((mutex==1)&amp;&amp;(empty!=0))
producer();
else
printf(&quot;Buffer is full!!&quot;);
break;
case 2: if((mutex==1)&amp;&amp;(full!=0))
consumer();
else
printf(&quot;Buffer is empty!!&quot;);
break;
case 3:
exit(0);
break;
}
}
return 0;
}
int wait(int s)
{
return (--s);
}
int signal(int s)
{
return(++s);
}
void producer()
{
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf(&quot;\nProducer produces the item %d&quot;,x);
mutex=signal(mutex);
}
void consumer()
{
mutex=wait(mutex);

```

```
full=wait(full);  
empty=signal(empty);
```

OS LAB EXPERIMENTS CODES

```
printf("&quot;\nConsumer consumes item %d&quot;,x);  
x--;  
mutex=signal(mutex);  
}
```

3)Write a program to implement concurrent queues.

```
#include<stdio.h>  
#include<stdlib.h>  
#include<pthread.h>  
typedef struct __node_t {  
    int value;  
    struct __node_t *next;  
} node_t;  
  
typedef struct __queue_t {  
    node_t *head;  
    node_t *tail;  
    pthread_mutex_t headLock;  
    pthread_mutex_t tailLock;  
} queue_t;  
  
void Queue_Init(queue_t *q) {  
    node_t *tmp = malloc(sizeof(node_t));  
    tmp->next = NULL;  
    q->head = q->tail = tmp;  
    pthread_mutex_init(&q->headLock, NULL);  
    pthread_mutex_init(&q->tailLock, NULL);  
}  
  
void Queue_Enqueue(queue_t *q, int value) {  
    node_t *tmp = malloc(sizeof(node_t));  
    tmp->value = value;
```

```

tmp->next = NULL;
pthread_mutex_lock(&q->tailLock);
q->tail->next = tmp;
q->tail = tmp;
pthread_mutex_unlock(&q->tailLock);
}

```

```

int Queue_Dequeue(queue_t *q, int value) {
pthread_mutex_lock(&q->headLock);
node_t *tmp = q->head;
node_t *newHead = tmp->next;
if (newHead == NULL) {
pthread_mutex_unlock(&q->headLock);
return -1; // queue was empty
}
*value = newHead->value;
q->head = newHead;
pthread_mutex_unlock(&q->headLock);
free(tmp);
return 0;
}

```

```

int main(){
    queue_t *q=(struct __queue_t*)malloc(sizeof(struct __queue_t));
    Queue_Init(q);
    Queue_Enqueue(q,10);
    Queue_Enqueue(q,20);
    Queue_Enqueue(q,30);
    Queue_Enqueue(q,40);
    queue_t *p=q;
    for(int i=0;i<4;i++){
        printf("%d\n",p->head->value);
    }
}

```

SKILL EXERCISE 8:

1. Write UNIX system program to implement producer-consumer problem in which multiple producer threads fill an array that is processed by one consumer thread.

Version 3: The consumer started only after the producers were finished and this required a conditional variable and its mutex.

```
#include <stdio.h>
#include <pthread.h>
#define BufferSize 10
void *Producer();
void *Consumer();
int BufferIndex=0;
char *BUFFER;
pthread_cond_t Buffer_Not_Full=PTHREAD_COND_INITIALIZER;
pthread_cond_t Buffer_Not_Empty=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mVar=PTHREAD_MUTEX_INITIALIZER;
int main()
{
    pthread_t ptid,ctid;
    BUFFER=(char *) malloc(sizeof(char) * BufferSize);
    pthread_create(&ptid,NULL,Producer,NULL);
    pthread_create(&ctid,NULL,Consumer,NULL);
    pthread_join(ptid,NULL);
    pthread_join(ctid,NULL);
    return 0;
}
void *Producer()
{
    for(;;)
    {
        pthread_mutex_lock(&mVar);
        if(BufferIndex==BufferSize)
        {
            pthread_cond_wait(&Buffer_Not_Full,&mVar);
        }
        BUFFER[BufferIndex++]=&#39;@&#39;;
        printf(&quot;Produce : %d \n&quot;,&BufferIndex);
        pthread_mutex_unlock(&mVar);
        pthread_cond_signal(&Buffer_Not_Empty);
    }
}
void *Consumer()
```

```
{
for(;;)
```

OS LAB EXPERIMENTS CODES

```
{
pthread_mutex_lock(&mVar);
if(BufferIndex==1)
{
pthread_cond_wait(&Buffer_Not_Empty,&mVar);
}
printf("Consume : %d \n",BufferIndex--);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Not_Full);
}
}
```

2) Your solution using semaphores should demonstrate three different types of semaphores:

a) A binary semaphore named mutex protects the critical regions: inserting a data item into the buffer (for the producer) and removing a data item from the buffer (for the consumer). A binary semaphore that is used as a mutex is initialized to 1. (Obviously we could use a real mutex for this, instead of a binary semaphore.)

b) A counting semaphore named nempty counts the number of empty slots in the buffer. This semaphore is initialized to the number of slots in the buffer (NBUFF).

c) A counting semaphore named nstored counts the number of filled slots in the buffer. This semaphore is initialized to 0, since the buffer is initially empty

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>
#define NBUFF 10
#define SEM_MUTEX "mutex"
/* these are args to px_ipc_name() */
```



```

#define SEM_EMPTY "empty"
#define SEM_NSTORED "nstored"
int nitems; /* read-only by producer and consumer */
struct {
    /* data shared by producer and consumer */
    int buff[NBUFF];
    sem_t *mutex;
    sem_t *empty;
    sem_t *nstored;
} shared; void *produce(void *), *consume(void *);
int main(int argc, char **argv)
{
    pthread_t
    tid_produce, tid_consume;
    if (argc != 2)
    {
        printf("usage: prodcons1 <#items>");
        exit(1);
    }
    nitems = atoi(argv[1]);
    Synchronization - II
    /* 4create three semaphores */
    shared.mutex = sem_open(SEM_MUTEX, O_CREAT | O_EXCL, 0644, 1);
    shared.empty = sem_open(SEM_EMPTY, O_CREAT | O_EXCL, 0644, NBUFF);
    shared.nstored = sem_open(SEM_NSTORED, O_CREAT | O_EXCL, 0644, 0);
    /* 4create one producer thread and one consumer thread */
    pthread_setconcurrency(2);
    pthread_create(&tid_produce, NULL, produce, NULL);
    pthread_create(&tid_consume, NULL, consume, NULL);
    /* 4wait for the two threads */
    pthread_join(tid_produce, NULL); pthread_join(tid_consume, NULL);
    /* 4remove the semaphores */
    sem_unlink(SEM_MUTEX);
    sem_unlink(SEM_EMPTY);
    sem_unlink(SEM_NSTORED);
    exit(0);
}
/* end main */
/* include prodcons */
void *produce(void *arg)
{
    int i;
    for (i = 0; i < nitems; i++) {

```

```

sem_wait(shared.nempty);
/* wait for at least 1 empty slot */
sem_wait(shared.mutex);
shared.buff[i % NBUFF] = i;
/* store i into circular buffer */
sem_post(shared.mutex);
sem_post(shared.nstored);
/* 1 more stored item */
}
return(NULL);
}

void *consume(void *arg)
{
int i;
for (i = 0; i < nitems; i++) {
sem_wait(shared.nstored); /* wait for at least 1 stored item */
sem_wait(shared.mutex);
if (shared.buff[i % NBUFF] == i)
printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
sem_post(shared.mutex);
sem_post(shared.nempty);
/* 1 more empty slot */
}
return(NULL);
}

```

HOME ASSIGNMENT:

- 1) Write a program to implement concurrent queues.
- 2) Illustrate the purpose of and types of locks?
- 3) What is the difference between mutexes and semaphores?
- 4) what are the principles and problems in concurrency ?

REVIEW - PROBLEM SET III

1. Answer yes/no, and provide a brief explanation.

(a) Is it necessary for threads in a process to have separate stacks?

(b) Is it necessary for threads in a process to have separate copies of the program executable?

1. Can one have concurrent execution of threads/processes without having parallelism? If yes, describe how. If not, explain why not.

3. Consider a multithreaded web server running on a machine with N parallel CPU cores. The server has M worker threads. Every incoming request is put in a request queue, and served by one of the free worker threads. The server is fully saturated and has a certain throughput at saturation. Under which circumstances will increasing M lead to an increase in the saturation throughput of the server?

4. Consider a process that uses a user level threading library to spawn 10 user level threads. The library maps these 10 threads on to 2 kernel threads. The process is executing on a 8-core system. What is the maximum number of threads of a process that can be executing in parallel?

5. Consider a user level threading library that multiplexes $N > 1$ user level threads over $M \geq 1$ kernel threads. The library manages the concurrent scheduling of the multiple user threads that map to the same kernel thread internally, and the programmer using the library has no visibility or control on this scheduling or on the mapping between user threads and kernel threads. The N user level threads all access and update a shared data structure. When (or, under what conditions) should the user level threads use mutexes to guarantee the consistency of the shared data structure?

A. Only if $M > 1$.

B. Only if $N \geq M$.

C. Only if the M kernel threads can run in parallel on a multi-core machine.

D. User level threads should always use mutexes to protect shared data.

6. Creating user level threads in a Linux application via any threading library always leads to the creation of corresponding kernel-level threads. [T/F]

7. Consider a Linux application with two threads T_1 and T_2 that both share and access a common variable x . Thread T_1 uses a pthread mutex lock to protect its access to x . Now, if thread T_2 tries to write to x without locking, then the Linux kernel generates a trap. [T/F]

8. In a single processor system, the kernel can simply disable interrupts to safely access kernel data structures, and does not need to use any spin locks. [T/F]

9. In the pthread condition variable API, a process calling wait on the condition variable must do so with a mutex held. State one problem that would occur if the API were to allow calls to wait without requiring a mutex to be held.

10. Consider N threads in a process that share a global variable in the program. If one thread makes a change to the variable, is this change visible to other threads? (Yes/No)

11. Consider N threads in a process. If one thread passes certain arguments to a function in the program, are these arguments visible to the other threads? (Yes/No)

12. Consider a user program thread that has locked a pthread mutex lock (that blocks when waiting for a lock to be released) in user space. In modern operating systems, can this thread be context switched out or interrupted while holding the lock? (Yes/No)

13. Repeat the previous question when the thread holds a pthread spinlock in userspace.

14. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process be interrupted by external hardware before it releases the spinlock? (Yes/No)

15. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process initiate a disk read before it releases the spinlock? (Yes/No)

16. When a user space process executes the wakeup/signal system call on a pthread condition variable, does it always lead to an immediate context switch of the process that calls signal (immediately after the signal instruction)? (Yes/No)

17. Several processes wish to read and write data shared between them. Some processes only want to read, while others want to update the shared data. Multiple readers may concurrently access the data. However, a writer must not access the data concurrently with anyone else, either a reader or a writer. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use condition variables and mutexes only in your solution.

18. Consider the readers and writers problem discussed above. Recall that multiple readers can be allowed to read concurrently, while only one writer at a time can access the critical section. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use only semaphores, and no other synchronization mechanism, in your solution. Further, you must avoid using more semaphores than is necessary. Clearly list all the variables (semaphores, and any other flags/counters you may need) and their initial values at the start of your solution. Use the notation down(x) and up(x) to invoke atomic down and up operations on a semaphore x that are available via the OS API. Use sensible names for your variables.

19. Consider the readers and writers problem as discussed above. We wish to implement synchronization between readers and writers, while giving preference to writers, where no waiting writer should be kept waiting for longer than necessary. For example, suppose reader process R1 is actively reading. And a writer process W1 and reader process R2 arrive while R1 is reading. While it might be fine to allow R2 in, this could prolong the waiting time of W1 beyond the absolute minimum of waiting until R1 finishes. Therefore, if we want writer preference, R2 should not be allowed before W1. Your goal is to write down pseudocode for read lock, read unlock, write lock, and write unlock functions that the processes should call, in order to realize read/write locks with writer preference. You must use only simple locks/mutexes and conditional variables in your solution. Please pick sensible names for your variables so that your solution is readable.

20. Write a solution to the readers-writers problem with preference to writers discussed above, but using only semaphores

CO 4

WEEK-9 (DIRECTORIES AND FILES)

Prerequisite:

- Knowledge on functionalities of different Commands like:
- `dir`
- `ls`
- `ln`
- `chmod`
- `chown`
- `rm`
- `stat`
- should have a complete idea of `umask` linux command.
- knowledge on `chmod()` and `chown()` functions

Pre-Lab Task:

1. Write the function prototypes for the following calls

1) `opendir()` :-

```
#include <dirent.h>
```

```
DIR *opendir(const char *dirname);
```

The `opendir()` function shall open a directory stream corresponding to the directory named by the `dirname` argument. The directory stream is positioned at the first entry. If the type `DIR` is implemented using a file descriptor, applications shall only be able to open up to a total of `{OPEN_MAX}` files and directories.

2) `readdir()`:-

```
#include <dirent.h>
```

```
struct dirent *readdir(DIR *dirp);
```

The type `DIR`, which is defined in the [dirent.h](#) header, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of `readdir()`

3) `closedir()`:-

```
#include <dirent.h>
```

```
int closedir(DIR *dirp);
```

The **`closedir()`** function shall close the directory stream referred to by the argument *dirp*. Upon return, the value of *dirp* may no longer point to an accessible object of the type DIR. If a file descriptor is used to implement type DIR, that file descriptor shall be closed.

4) `stat()`:-

```
#include <sys/stat.h>
```

```
int stat(const char *restrict path, struct stat *restrict buf);
```

The **`stat()`** function shall obtain information about the named file and write it to the area pointed to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or execute permission of the named file is not required. An implementation that provides additional or alternate file access control mechanisms may, under implementation-defined conditions, cause **`stat()`** to fail. In particular, the system may deny the existence of the file specified by *path*

5) `lstat()`:-

```
#include <sys/stat.h>
```

```
int lstat(const char *restrict path, struct stat *restrict buf);
```

The **`lstat()`** function shall be equivalent to **`stat()`**, except when *path* refers to a symbolic link. In that case **`lstat()`** shall return information about the link, while **`stat()`** shall return information about the file the link references.

6) `chown()`:-

```
#include <sys/types.h>
```

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

```
int chown(const char *pathname, uid_t owner, gid_t group);
```

The **`chown()`** function sets the owner ID and group ID of the file that *pathname* specifies. For this call to succeed, the effective user ID of the process must match the owner of the file, or the process must have appropriate privileges.

7) `chmod()`:-

```
#include <sys/stat.h>
```

```
int chmod(const char *pathname, mode_t mode);
```

`chmod()` changes the mode of the file specified whose pathname is given in *pathname*, which is dereferenced if it is a symbolic link.

8) fchown():-

```
#include <sys/types.h>
```

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

```
int fchown(int filde, uid_t owner, gid_t group);
```

The fchown() function has the same effect as chown() except that the file whose ownership is to be changed is specified by file descriptor rather than path name.

9) fchmod():-

```
#include <sys/stat.h>
```

```
int fchmod(int filde, mode_t mode);
```

The fchmod() function shall be equivalent to [chmod\(\)](#) except that the file whose permissions are changed is specified by the file descriptor *filde*.

10) fcntl():-

```
#include <fcntl.h>
```

The <fcntl.h> header shall define the requests and arguments for use by the functions [fcntl\(\)](#) and [open\(\)](#).

11) ftruncate():-

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

```
int ftruncate(int filde, off_t length);
```

```
int truncate(const char *path, off_t length);
```

The ftruncate() function causes the regular file referenced by *filde* to have a size of *length* bytes

12) ioctl() :-

```
#include <sys/ioctl.h>
```

```
int ioctl(int fd, unsigned long request, ...);
```

The ioctl() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests

13) link():-

the link() method used to create an HTML snippet for a hypertext link. The returned string can then be added to the document via [document.write\(\)](#) or [element.innerHTML](#).

Links created with the link() method become elements in the links array of the documentobject. See [document.links](#)

14) mknod():-

```
#include <sys/stat.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

The `mknod()` function creates a new special file named by *pathname*. *mode* specifies the type of the file to be created and the file's access mode. With the PTC MKS Toolkit UNIX APIs, `mknod()` only supports creating FIFO special files, indicated by ORing `S_IFIFO` into the *mode* parameter. The *dev* parameter is ignored. Refer to [mkfifo\(\)](#) for details on creating FIFO special files.

15) sync():-

```
#include <unistd.h>
void sync(void);
int syncfs(int fd);
```

`sync()` causes all pending modifications to filesystem metadata and cached file data to be written to the underlying filesystems.

16) truncate():-

```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
```

The `truncate()` is a function cause the regular file named by *path* or referenced by *fd* to be truncated to a size of precisely *length* bytes.

17) mount():-

```
#include <sys/mount.h>
int mount(const char *source, const char *target,
          const char *filesystemtype, unsigned long mountflags, const void *data);
```

`mount()` attaches the filesystem specified by *source* (which is often pathname referring to a device, but can also be the pathname of a directory or file, or a dummy string) to the location (a directory or file) specified by the pathname in *target*.

18) getcwd():-

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

The **getcwd()** function copies an absolute pathname of the current working directory to the array pointed to by *buf*, which is of length *size*.

19) chdir():-

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

```
int chdir(const char *path);
```

The **chdir()** function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with '/'

2. Write a program to print list of files and folders in column format (dir.c) and demonstrate

Directory navigation with chdir and getcwd ?

```
#include<unistd.h>
#include<stdio.h>
#include<dirent.h>
int main(){
    char s[100];

    struct dirent *de; // Pointer for directory entry

    // opendir() returns a pointer of DIR type.
    DIR *dr = opendir(".");

    if (dr == NULL) // opendir returns NULL if couldn't open directory
    {
        printf("Could not open current directory" );
        return 0;
    }

    // Refer http://pubs.opengroup.org/onlinepubs/7990989775/xsh/readdir.html
    // for readdir()
    while ((de = readdir(dr)) != NULL)
        printf("%s\n", de->d_name);

    closedir(dr);

    printf("%s\n", getcwd(s, 100));
```

```

// using the command
chdir("new12");

// printing current working directory
printf("%s\n", getcwd(s, 100));

// after chdir is executed
return 0;
}

```

In Lab Task:

Steps of the Program:

- Open the Terminal.
- Then, you can see the logged in user with the \$ symbol next to the username.
- \$ means you are logged in as a regular user and the # means you are root user.

Problem Description:

1. Write a system program that demonstrates chmod(), chown() System calls for changing file permissions.

```

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int main(){
    int p=fork();
    if(p==0){
        char f[]="hello2";
        FILE *str;
        str=fopen(f,"w");
        fclose(str);
        struct stat s;
        stat(f,&s);

        printf("%08x\n",s.st_mode);
        if(chmod(f,S_IRWXU|S_IRWXG)==0){
            stat(f,&s);

```

```

        printf("%08x\n",s.st_mode);
        unlink(f);
    }
    else
        exit(1);
}
}

```

2.umask returns the previous value of the mask. Unlike the shell's umask statement, however,the umask system call can't display the current value of the mask without changing it. To use a workaround, store the current mask by changing it to some arbitrary value, and then display it before restoring it. Write a Program that Changes umask twice and checks effect on permissions.

```

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int main(){
    int p=fork();
    if(p==0){
        char f[]="hello2";
        FILE *str;
        str=fopen(f,"w");
        fclose(str);
        struct stat s;
        stat(f,&s);

        printf("%08x\n",s.st_mode);
        if(chmod(f,S_IRWXU|S_IRWXG)==0){
            stat(f,&s);
            printf("%08x\n",s.st_mode);
            unlink(f);
        }
        else
            exit(1);
    }
}

```

SKILL EXERCISE 9:

1. Write a system program that lists only directories - Uses S_IFMT and S_ISDIR macros.

```
#include <sys/types.h>
#include
<sys/stat.h>
#include
<stdio.h>
#include
<dirent.h>
int main(int argc, char
*argv[]) { DIR *dir;
struct dirent *dirent; /* Returned by readdir() */
struct stat statbuf; /* Address of statbuf used by lstat() */
UNIX Systems Programming Exercises-3
mode_t file_type, file_perm;
if ((dir = opendir(argv[1])) ==
NULL) quit("Couldn't open
directory", 1);
if ((chdir(argv[1]) == -1)) /* Change to the directory before */
quit("chdir", 2); /* you starting reading its entries */
while ((dirent = readdir(dir)) != NULL) { /* Read each entry in directory*/ if
(lstat(dirent->d_name, &statbuf) < 0) { /* dname must be in */ perror("lstat"); /*
current directory */
continue;
}
if (S_ISDIR(statbuf.st_mode)) { /* If file is a directory */
file_type = statbuf.st_mode & S_IFMT;

file_perm = statbuf.st_mode & ~S_IFMT;
printf("%o %4o %s\n", file_type, file_perm, dirent->d_name);
}
}
exit(0);
}
unlink
int unlink (const char* fileName)
    unlink () removes the hard link from the name fileName to its file. If fileName is the
    last link to the file,
    the file's resources are
    deallocated. stat
    int stat (const char* name, struct stat*
    buf) struct stat {
    mode_t st_mode; /* file type & mode
    (permissions) */ ino_t st_ino; /* i-node number
    (serial number) */ dev_t st_dev; /* device number
    (file system) */
```



```

dev_t st_rdev; /* device number for special files
*/ nlink_t st_nlink; /* number of links */
uid_t st_uid; /* user ID of
owner */ gid_t st_gid; /* group
ID of owner */
off_t st_size; /* size in bytes, for regular files */
time_t st_atime; /* time of last access */
time_t st_mtime; /* time of last modification */
time_t st_ctime; /* time of last file status change */
blksize_t st_blksize; /* best I/O block size */
blkcnt_t st_blocks; /* number of disk blocks allocated */
};

```

Link for

reference: <https://drive.google.com/open?id=1JO0oK7nqHhXss4IzeC-NfNbNOzzK1LxT>

2. Write a UNIX system program to demonstrate file locking. Implementation of Sequence Number increment problem with locking using fcntl()

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    /* l_type    l_whence  l_start  l_len  l_pid    */
    struct flock fl = { F_WRLCK, SEEK_SET, 0,      0,      0 };
    int fd;

    fl.l_pid = getpid();

    if (argc > 1)
        fl.l_type = F_RDLCK;

    if ((fd = open("lockdemo.c", O_RDWR)) == -1) {
        perror("open");
        exit(1);
    }

    printf("Press <RETURN> to try to get lock: ");
    getchar();
    printf("Trying to get lock...");
}

```

```

if (fcntl(fd, F_SETLK, &fl) == -1) {
    perror("fcntl");
    exit(1);
}

printf("got lock\n");
printf("Press <RETURN> to release lock: ");
getchar();

fl.l_type = F_UNLCK; /* set to unlock same region */

if (fcntl(fd, F_SETLK, &fl) == -1) {
    perror("fcntl");
    exit(1);
}

printf("Unlocked.\n");

close(fd);
}

```

Reference link:-

<https://www.ict.griffith.edu.au/teaching/2501ICT/archive/guide/ipc/flock.html>


WEEK - 10(DISK SCHEDULING ALGORITHMS AND SOCKETS)

Prerequisite:

- Complete idea on very simple file systems.
- Brief idea on disk scheduling algorithms
- Should have the basic idea on Linux inodes {included newly}

Pre-Lab Task:

1. Write the contents of ext2 inode, inode Table

In the EXT2 file system, the inode is the basic building block; every file and directory in the file system is described by one and only one inode. The EXT2 inodes for each Block Group are kept in the inode table together with a bitmap that allows the system to keep track of allocated and unallocated inodes. Figure  shows the format of an EXT2 inode, amongst other information, it contains the following fields:

mode

This holds two pieces of information; what does this inode describe and the permissions that users have to it. For EXT2, an inode can describe one of file, directory, symbolic link, block device, character device or FIFO.

Owner Information

The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size

The size of the file in bytes,

Timestamps

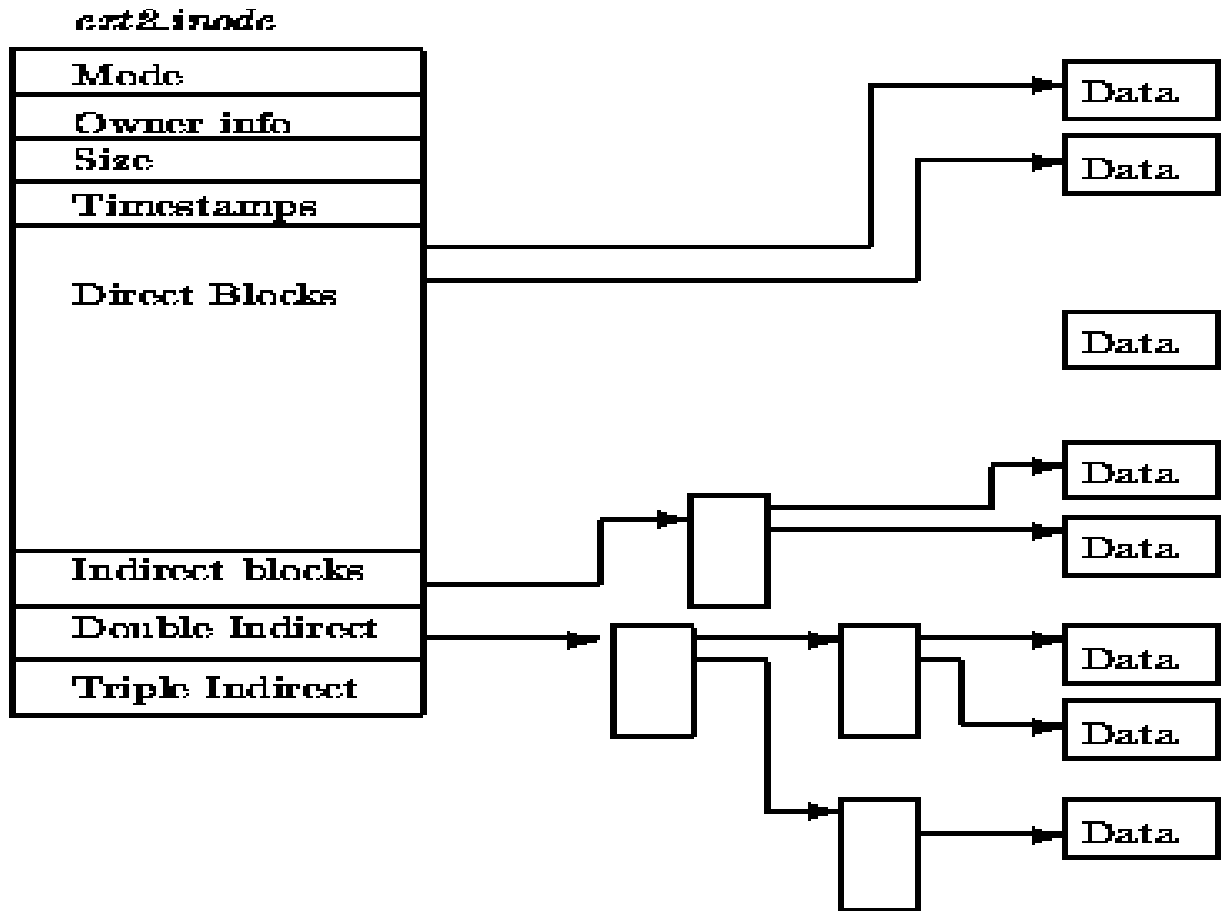
The time that the inode was created and the last time that it was modified,

Datablocks

Pointers to the blocks that contain the data that this inode is describing. The first twelve are pointers to the physical blocks containing the data described by this inode and the last three pointers contain more and more levels of indirection. For example, the double indirect blocks pointer points at a block of pointers to blocks of pointers to data blocks. This means that files less than or equal to twelve data blocks in length are more quickly accessed than larger files.

You should note that EXT2 inodes can describe special device files. These are not real files but handles that programs can use to access devices. All of the device files in /dev are there to allow programs to access Linux's devices. For example the *mount* program takes the device file that it

wishes to mount as an argument.



2. Write function prototypes for the following system calls

1. `accept()`:-

```
#include <sys/socket.h>
```

```
int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);
```

2. `bind()`:-

```
#include <sys/socket.h>
```

```
int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);
```

3. `connect()`:-

```
#include <sys/socket.h>
```

```
int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);
```

4. `fcntl()`:-

```
#include <sys/socket.h>
```

```
int connect(int socket, const struct sockaddr *address,  
            socklen_t address_len);
```

5. `getpeername()`:-

```
#include <sys/socket.h>
```

```
int getpeername(int socket, struct sockaddr *restrict address,  
                socklen_t *restrict address_len);
```

6. `getsockname()` :-

```
#include <sys/socket.h>
```

```
int getsockname(int socket, struct sockaddr *restrict address,  
                socklen_t *restrict address_len);
```

7. `getsockopt()`:-

```
#include <sys/socket.h>
```

```
int getsockopt(int socket, int level, int option_name,  
               void *restrict option_value, socklen_t *restrict option_len);
```

8. `ioctl()`:-

```
#include <sys/ioctl.h>
```

```
int ioctl(int fd, unsigned long request, ...);
```

9. `listen()`:-

```
#include <sys/socket.h>
```

```
int listen(int socket, int backlog);
```

10. `read()`:-

```
#include <sys/socket.h>
```

```
int listen(int socket, int backlog);
```

11. `recv()`:-

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags)
```

```
;
```

12. select ():-

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

13. send ():-

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

14. socketpair ():-

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol,
               int socket_vector[2]);
```

15. write ():-

```
#include <unistd.h>
#include <sys/types.h>
int write(int fd, char *Buff, int NumBytes);
```

3) A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the filesystem.

Now write a code for very simple file system.

In Lab Task:

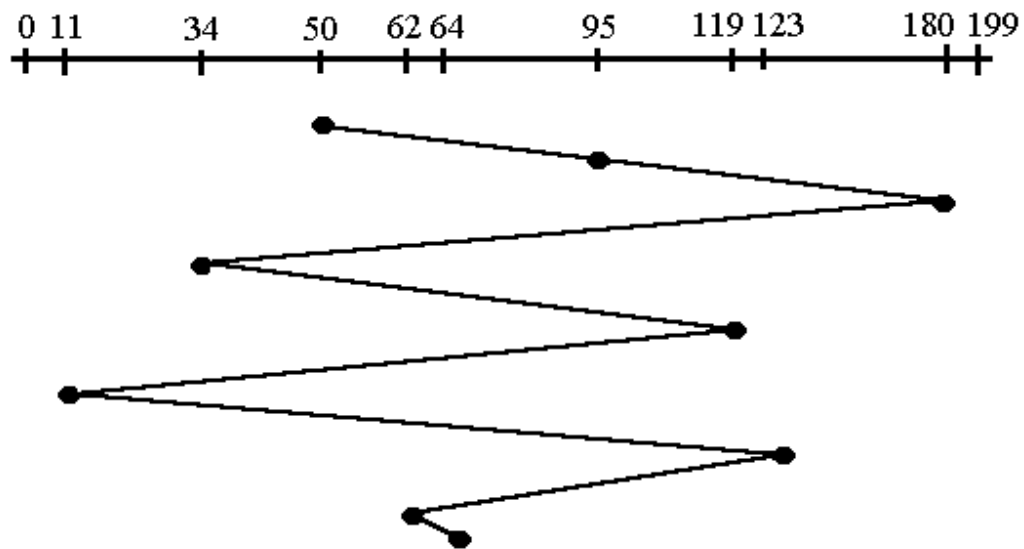
Steps of the Program:

- Open the Terminal.
- Then, you can see the logged in user with the \$ symbol next to the username.
- \$ means you are logged in as a regular user and the # means you are root user.

1.Demonstrate I/O Disk Scheduling Algorithms using Queue Data Structure using the algorithms

- FIFO or FCFS

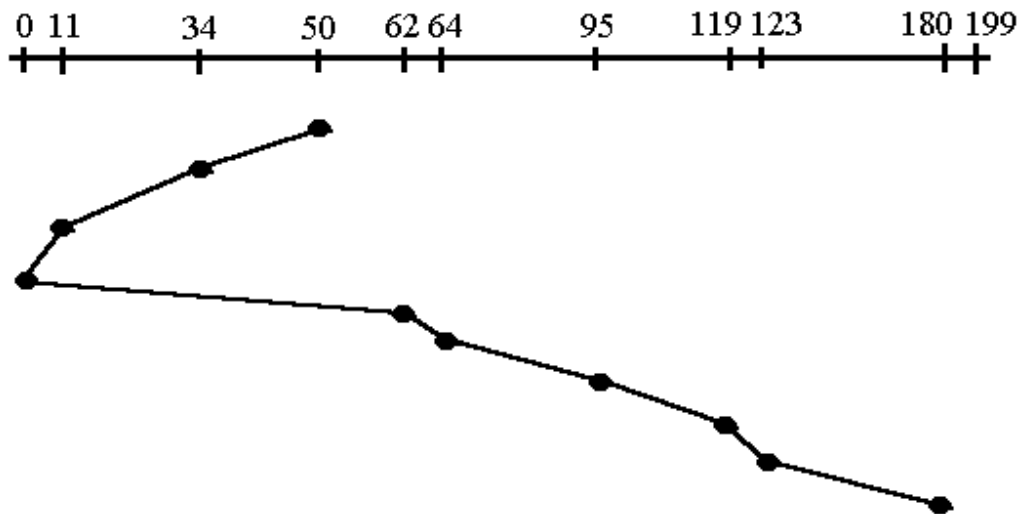
All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.



- SSTF

In this case request is serviced according to next shortest distance. Starting at 50, the next shortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16 tracks away from 34. The process would continue until all the process are taken care of. For example the next case would be to move from 62 to 64 instead of 34 since there are only 2 tracks between them and not 18 if it were to go the other way. Although this seems to be a better service being that it moved a total of 236 tracks, this is not an optimal one. There is a great chance that starvation would take place. The reason for this is if there were a lot of requests close to each other the other

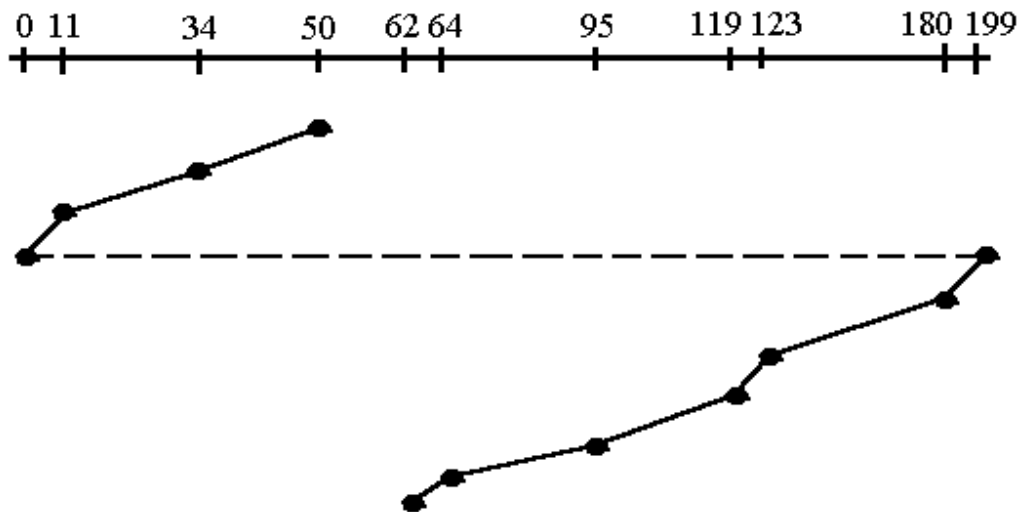
requests will never be handled since the distance will always be greater.



- SCAN

This approach works like an elevator does. It scans down towards the nearest end and then when it hits the bottom it scans up servicing the requests that it didn't get going down. If a request comes in after it has been scanned it will not be serviced until the process comes back down or moves back up. This process moved a total of 230 tracks. Once again this is more optimal than the previous algorithm, but it is

not the best.



2. Write a program to print the attributes of a file.

1. what time was this file created?
2. time of last modification
3. time of last access

```
#include <stdio.h>
#include <sys/stat.h>
#include <time.h>
int main()
{
    struct stat filestat;
```

```

stat("gettysburg.txt",&filestat);
/* newline included in ctime() output */
printf(" File last accessed time %s",ctime(&filestat.st_atime));
printf(" File last modified time %s",ctime(&filestat.st_mtime));
printf(" File created time %s",ctime(&filestat.st_birthtime));
return(0);
}

```

SKILL EXERCISE 10:

1. Write a system program to demonstrate link() – to create a hard link between “File1.txt” and “File2.txt”. Add some text to “File1.txt” and print the content in both files. Demonstrate truncate() - modifies the size of the file to “N” bytes system calls

```

#include <stdio.h>
#include <direct.h>
int main ()
{
    DIR *d;
    struct direct *dir;
    L = opendir(".");
    if(L)
    {
        while((dir = readdir(d))!=NULL)
        {
            printf("T.s\n",dir->d_name);
        }
        closedir(d);
    }
    return 0;
}

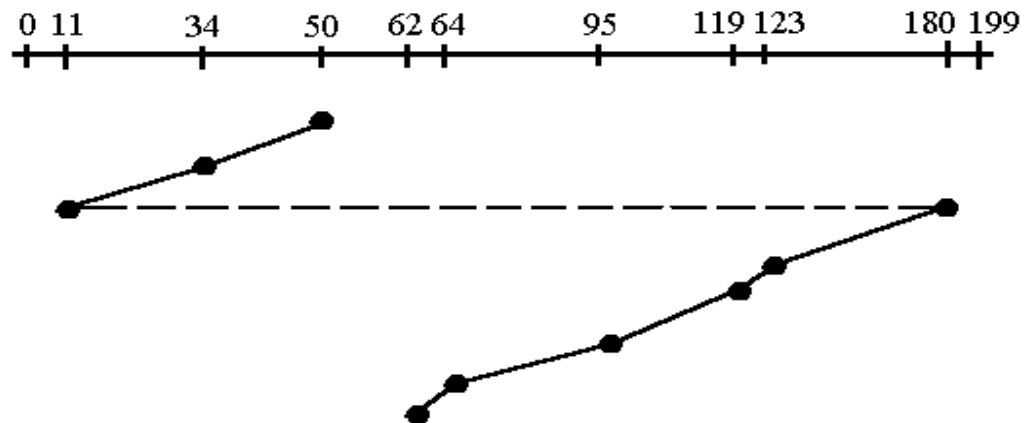
```

2. Demonstrate I/O Disk Scheduling Algorithms using Queue Data Structure using the algorithms

- CSCAN
- FSCAN
- LOOK
- CLOOK

cscan

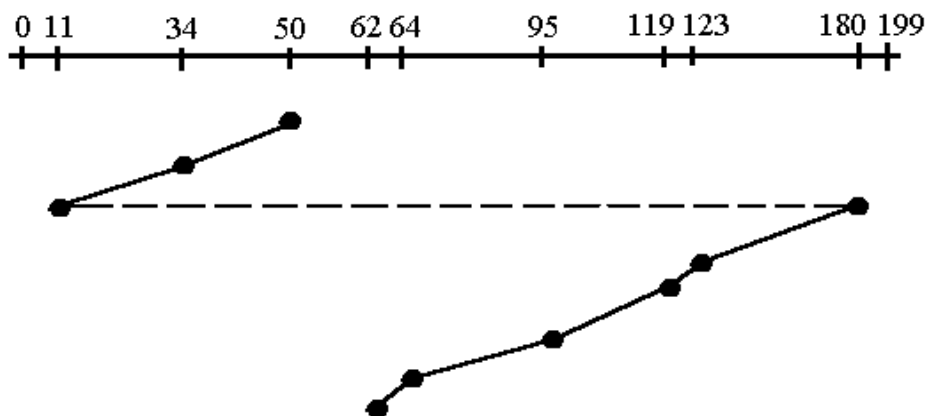
Circular scanning works just like the elevator to some extent. It begins its scan toward the nearest end and works its way all the way to the end of the system. Once it hits the bottom or top it jumps to the other end and moves in the same direction. Keep in mind that the huge jump doesn't count as a head movement. The total head movement for this algorithm is only 187 track, but still this isn't the most sufficient.



Clook

This is just an enhanced version of C-SCAN. In this the scanning doesn't go past the last request in the direction that it is moving. It too jumps to the other end but not all the way to the end. Just to the furthest request. C-SCAN had a total movement of 187 but this scan (C-LOOK) reduced it down to 157 tracks.

From this you were able to see a scan change from 644 total head movements to just 157. You should now have an understanding as to why your operating system truly relies on the type of algorithm it needs when it is dealing with multiple processes.



look

- The disk arm starts at the first I/O request on the disk, and moves toward the last I/O request on the other end, servicing requests until it gets to the other extreme I/O request on the disk, where the head movement is reversed and servicing continues.

- It moves in both directions until both last I/O requests; more inclined to serve the middle cylinder requests.

WEEK-11(XV6 RELATED)

Q) Add wait2() system call to xv6 kernel to calculate run & wait time of a command. Use time followed by command.

Here we try to create a basic system-call. The first step is to extend the current proc structure and add new fields ctime, etime and rtime for creation time, end-time and total time respectively of a process. When a new process gets created the kernel code should update the process creation time. The run-time should get updated after every clock tick for the process. To extract this information from the kernel, add a new system call which extends wait. The new call will be

```
int waitx(int *wtime, int *rtime)
```

The two arguments are pointers to integers to which waitx will assign the total number of clock ticks during which process was waiting and total number of clock ticks when the process was running. The return values for waitx should be same as that of wait system-call. Created a test program which utilises the waitx system-call by creating a 'time' like command for the same.

1. ADDING SYSTEM CALL:

- waitx() system call was added modifying user.h , usys.S , syscall.h , syscall.c , sysproc.c, defs.h .
- proc.c contains the actual waitx() system call. Copied the code of wait() and modified it as follows-Searched for a zombie child of parent in the proc table. -When the child was found, following pointers were updated : *wtime= p->etime - p->ctime1 - p->rtime - p->iotime; *rtime=p->rtime; -sysproc.c is just used to call waitx() which is present in proc.c . The sys_waitx() function in sysproc.c although does one job - it passes the parameters (rtime,wtime) to the waitx() of proc.c , just like all other system calls do.

2. MODIFICATIONS DONE TO proc structure

- ctime1 (records CREATION TIME) , etime (records END TIME) , rtime (calculates RUN TIME) & iotime (calculates IO TIME) fields added to proc structure of proc.h file

3. HOW ctime,etime & rtime ARE CALCULATED :

- ctime is recorded in allocproc() function of proc.c.(When process is born)
- etime is recorded in exit() function (i.e when child exists, ticks are recorded) of proc.c.
- rtime is updated in trap() function of trap.c .(IF STATE IS RUNNING , THEN UPDATE rtime)
- iotime is updated in trap() function of trap.c.(IF STATE IS SLEEPING , THEN UPDATE wtime)

4. THE time COMMAND -time inputs a command and exec it normally.

- The only thing different is that it uses 'waitx()' instead of normal wait(). -at the end it displays rtime and wtime. -status returned is also displayed. -to get status (IN QUESTION IT WAS ASKED TO RETURN SAME STATUS AS wait()) wait() is called from within waitx().

(Install QEMU Emulator, then run the Xv6 code)

Ans:-

<https://github.com/prakhar987/xv6-system-calls>

SKILL EXERCISE 11:

1. Write a system program to demonstrate UNIX Domain sockets - implementing echo client server

Client-program:-

```
// Write CPP code here
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int sockfd)
{
    char buff[MAX];
    int n;
    for (;;) {
        bzero(buff, sizeof(buff));
        printf("Enter the string : ");
        n = 0;
        while ((buff[n++] = getchar()) != '\n')
            ;
        write(sockfd, buff, sizeof(buff));
        bzero(buff, sizeof(buff));
        read(sockfd, buff, sizeof(buff));
        printf("From Server : %s", buff);
        if ((strcmp(buff, "exit", 4)) == 0) {
            printf("Client Exit...\n");
            break;
        }
    }
}

int main()
{
    int sockfd, connfd;
    struct sockaddr_in servaddr, cli;

    // socket create and varification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    }
```



```

else
    printf("Socket successfully created..\n");
    bzero(&servaddr, sizeof(servaddr));

    // assign IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);

    // connect the client socket to server socket
    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("connection with the server failed...\n");
        exit(0);
    }
    else
        printf("connected to the server..\n");

    // function for chat
    func(sockfd);

    // close the socket
    close(sockfd);
}

```

SERVER PROGRAM

```

#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr

// Function designed for chat between client and server.

```

```

void func(int sockfd)
{
    char buff[MAX];
    int n;
    // infinite loop for chat
    for (;;) {
        bzero(buff, MAX);

        // read the message from client and copy it in buffer
        read(sockfd, buff, sizeof(buff));
        // print buffer which contains the client contents
        printf("From client: %s\t To client : ", buff);
        bzero(buff, MAX);
        n = 0;
        // copy server message in the buffer
        while ((buff[n++] = getchar()) != '\n')
            ;

        // and send that buffer to client
        write(sockfd, buff, sizeof(buff));

        // if msg contains "Exit" then server exit and chat ended.
        if (strncmp("exit", buff, 4) == 0) {
            printf("Server Exit...\n");
            break;
        }
    }
}

// Driver function
int main()
{
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;

    // socket create and verification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully created..\n");

```

```

bzero(&servaddr, sizeof(servaddr));

// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);

// Binding newly created socket to given IP and verification
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
    printf("socket bind failed...\n");
    exit(0);
}
else
    printf("Socket successfully binded..\n");

// Now server is ready to listen and verification
if ((listen(sockfd, 5)) != 0) {
    printf("Listen failed...\n");
    exit(0);
}
else
    printf("Server listening..\n");
len = sizeof(cli);

// Accept the data packet from client and verification
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
    printf("server acccept failed...\n");
    exit(0);
}
else
    printf("server acccept the client...\n");

// Function for chatting between client and server
func(connfd);

// After chatting close the socket
close(sockfd);
}

```

WEEK - 12 (XV6 RELATED)

Pre-Lab Task:

Complete the Review – Problem Set VI prior to working on this lab.

In Lab Task:

Add a System Call in XV6 operating system which will return number of System call a process does

Ans:-

https://github.com/susobhang70/xv6_new_sys_call/tree/master/Code

SKILL EXERCISE 12:

1) Does the kernel stack of the process calling exec get reallocated during the execution of the exec system call? Answer yes/no and justify.

2) Do the pages holding the memory image of the process calling exec get reallocated during the execution of the exec system call? Answer yes/no and justify.

3) Do the pages that hold the page table entries of a process calling exec get reallocated during the execution of the exec system call? Answer yes/no and justify

4) The function walkpgdir is invoked on every memory access by the CPU, to translate virtual addresses to physical addresses. Answer True/False and justify.

5) Consider two processes in xv6 that both wish to read a particular disk block, i.e., either process does not intend to modify the data in the block. The first process obtains a pointer to the struct buf using the function "bread", but never causes the buffer to become dirty. Now, if the second process calls "bread" on the same block before the first process calls "brelse", will this second call to "bread" return immediately, or would it block? Briefly describe what xv6 does in this case and justify the design choice.

2. The hard disk is composed of a disk on which information is written and a cursor.

The disk is composed of multiple circular lists, each with double the capacity of the previous.

For example, if the first circular list has 16 positions, the next one has 32 and so on. The cursor can move from a circular list to another only from its starting index, 0. Execution will be done using 'a time division'. On each 'time division', the cursor can either execute a command, move to another address or 'do nothing'. There are multiple COMMANDS that can be given to the disk:

READ DATA - ::r [line] [index]

WRITE DATA - ::w [line] [index] [data]

READ DAMAGE - ::d [line] [index]

END - ::e

MULTIPLE READ DATA - ::mr [line] [index] [numberOfReads]

MULTIPLE WRITE DATA - ::mw [line] [index] [data] [data] ... [data]

Each command applies damage to the disk, as referenced:

READ DATA - 5 Damage

WRITE DATA - 30 Damage

READ DAMAGE - 2 Damage

'time division' - 1 Damage (each time division spent on an address adds 1 damage to it)
Develop a Hard Disk simulator using lists, stacks and queues.

HOME ASSIGNMENT :

1) Suppose a disk has 201 cylinders, numbered from 0 to 200. At some time the disk arm is at cylinder 100, and there is a queue of disk access requests for cylinders 30, 85, 90, 100, 105, 110, 135 and 145. If Shortest-Seek Time First (SSTF) is being used for scheduling the disk access, the request for cylinder 90 is serviced after servicing _____ number of requests.

- (A) 1
- (B) 2
- (C) 3
- (D) 4

2) Consider an operating system capable of loading and executing a single sequential user process at a time. The disk head scheduling algorithm used is First Come First Served (FCFS). If FCFS is replaced by Shortest Seek Time First (SSTF), claimed by the vendor to give 50% better benchmark results, what is the expected improvement in the I/O performance of user programs?

- (A) 50%
- (B) 40%
- (C) 25%
- (D) 0%

3) Suppose the following disk request sequence (track numbers) for a disk with 100 tracks is given: 45, 20, 90, 10, 50, 60, 80, 25, 70. Assume that the initial position of the R/W head is on track 50. The additional distance that will be traversed by the R/W head when the Shortest Seek Time First (SSTF) algorithm is used compared to the SCAN (Elevator) algorithm (assuming that SCAN algorithm moves towards 100 when it starts execution) is _____ tracks

- (A) 8
- (B) 9
- (C) 10
- (D) 11

4) Consider a typical disk that rotates at 15000 rotations per minute (RPM) and has a transfer rate of 50×10^6 bytes/sec. If the average seek time of the disk is twice the average rotational delay and the controller's transfer time is 10 times the disk transfer time, the average time (in milliseconds) to read or write a 512 byte sector of the disk is _____

5) Given Work Queue: 23, 89, 132, 42, 187. There are 200 cylinders numbered from 0 - 199 and the disk head starts at number 100

Discuss all 6 disk scheduling algorithms.

- Work Queue: 23, 89, 132, 42, 187
- there are 200 cylinders numbered from 0 - 199
- the diskhead starts at number 100

6) Compute the seek, rotation, and transfer times for the following sets of re-requests: -a 0, -a 6, -a 30, -a 7,30,8, and finally -a 10,11,12,13.

7) Do the same requests above, but change the seek rate to different values: -S2, -S 4, -S 8, -S 10, -S 40, -S 0.1. How do the times change?

8) Do the same requests above, but change the rotation rate: -R 0.1, -R 0.5, -R 0.01. How do the times change?

9) FIFO is not always best, e.g., with the request stream -a 7,30,8, what order should the requests be processed in? Run the shortest seek-time first (SSTF) scheduler (-p SSTF) on this workload; how long should it take (seek, rotation, transfer) for each request to be served?

10) Now use the shortest access-time first (SATF) scheduler (-p SATF). Does it make any difference for -a 7,30,8 workload? Find a set of requests where SATF outperforms SSTF; more generally, when is SATF better than SSTF?

11) Here is a request stream to try: -a 10,11,12,13. What goes poorly when it runs? Try adding track skew to address this problem (-o skew). Given the default seek rate, what should the skew be to maximize performance? What about for different seek rates (e.g., -S 2, -S 4)? In general, could you write a formula to figure out the skew?

12) Specify a disk with different density per zone, e.g., -z 10,20,30, which specifies the angular difference between blocks on the outer, middle, and inner tracks. Run some random requests (e.g., -a -1 -A 5,-1,0, which specifies that random requests should be used via the -a -1 flag and that five requests ranging from 0 to the max be generated), and compute the seek, rotation, and transfer times. Use different random seeds. What is the bandwidth (in sectors per unit time) on the outer, middle, and inner tracks?

13) A scheduling window determines how many requests the disk can examine at once. Generate random workloads (e.g., -A 1000,-1,0, with different seeds) and see how long the SATF scheduler takes when the scheduling window is changed from 1 up to the number of requests. How big of a window is needed to maximize performance? Hint: use the -c flag and don't turn on graphics (-G) to run these quickly. When the scheduling window is set to 1, does it matter which policy you are using?

14) Create a series of requests to starve a particular request, assuming an SATF policy. Given that sequence, how does it perform if you use a boundedSATF (BSATF) scheduling approach? In this approach, you specify the scheduling window (e.g., -w 4); the scheduler only moves onto the next window of requests when *all* requests in the current window have been serviced. Does this solve starvation? How does it perform, as compared to SATF? In general, how should a disk make this trade-off between performance and starvation avoidance

REVIEW - PROBLEM SET IV:

1. Provide one reason why a DMA-enabled device driver usually gives better performance over a non-DMA interrupt-driven device driver.

2. Which of the following statements is/are true regarding memory-mapped I/O?
 - A. The CPU accesses the device memory much like it accesses main memory.
 - B. The CPU uses separate architecture-specific instructions to access memory in the device.
 - C. Memory-mapped I/O cannot be used with a polling-based device driver.
 - D. Memory-mapped I/O can be used only with an interrupt-driven device driver.

3. Consider a file D1/F1 that is hard linked from another parent directory D2. Then the directory entry of this file (including the filename and inode number) in directory D1 must be exactly identical to the directory entry in directory D2. [T/F]

4. It is possible for a system that uses a disk buffer cache with FIFO as the buffer replacement policy to suffer from the Belady's anomaly. [T/F]

5. Reading files via memory mapping them avoids an extra copy of file data from kernel space buffers to user space buffers. [T/F]

6. A soft link can create a link between files across different file systems, whereas a hard link can only create links between a directory and a file within the same file system. [T/F]

7. Consider the process of opening a new file that does not exist (obviously, creating it during opening), via the "open" system call. Describe changes to all the in-memory and disk-based file system structures (e.g., file tables, inodes, and directories) that occur as part of this system call implementation. Write clearly, listing the structure that is changed, and the change made to it.

8. Repeat the above question for the implementation of the "link" system call, when linking to an existing file (not open from any process) in a directory from another new parent directory.

9. Repeat the above question for the implementation of the "dup" system call on a file descriptor.

10. Consider a file system with 512-byte blocks. Assume an inode of a file holds pointers to N direct data blocks, and a pointer to a single indirect block. Further, assume that the single indirect block can hold pointers to M other data blocks. What is the maximum file size that can be supported by such an inode design?

11. Consider a FAT file system where disk is divided into M byte blocks, and every FAT entry can store an N bit block number. What is the maximum size of a disk partition that can be managed by such a FAT design?

12. Consider a secondary storage system of size 2 TB, with 512-byte sized blocks. Assume that the filesystem uses a multilevel inode datastructure to track data blocks of a file. The inode has 64 bytes of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?

REVIEW – PROBLEM SET V:

1. Consider a socket program that has exactly one socket open for communication. The logic of the application is as follows: it calls read once on the socket by providing a buffer of size 1KB, prints out whatever has been read (if any), and repeats the read again in an infinite loop. It is known that the other remote end point of the socket rarely sends data. The designer of the application has the option of using a blocking or non-blocking socket. Which of two choices will lead to a lower usage of CPU cycles by the application, and why?

2. Consider a system running a web server application on a standard Linux system. The system has N processor cores. The web server has N worker threads, each pinned to execute on a core. All threads simultaneously run accept on the same server socket and accept new TCP connections in parallel. Each server thread manages several clients that it has accepted, and communicates with the remote clients via read and write system calls to service their requests. Answer the following design questions.

(a) Suggest one I/O mechanism by which a single worker thread on one core can handle potentially blocking I/O operations (such as network reads) across multiple client sockets simultaneously. (Note that you are not allowed to spawn new threads.)

(b) When the number of incoming connections is large, it is found that the threads spend a long time waiting in the execution of the accept system call. Suggest one explanation for why this may be happening, and suggest one kernel modification to address this issue.

(c) When a packet p of a TCP connection arrives at the network card, it is possible that the interrupt handling and TCP/IP processing for p runs on one core $C1$ and the final application layer processing is done by the server worker thread running on a different core $C2$. State one disadvantage of handling the same packet on two different cores. And state one mechanism by which this problem can be mitigated.

3. Consider a high performance networking application running on a high end multicore server. It is found that, under high incoming packet load, the system spends a large fraction of its time handling interrupts and context switches, leading to very little productive work at the application layer. Suggest one mechanism by which this problem can be mitigated. (For this question and the next, you are required to provide a description of the mechanism, not just its name.)

4. The current Linux network stack copies packet buffers several times, from the device to kernel space to user space. Suggest one mechanism by which the overhead of packet copies can be minimized, in order to build a high performance network stack.

5. Consider a web server that uses non-blocking event-driven I/O for network communication, but uses blocking I/O to access the disk. The web server wishes to run as multiple processes, so that the server can be available even if some subset of the processes block on disk I/O. Further, the web server wishes to receive web requests only on port 80, and not at different ports in the different processes. Suggest one

mechanism by which the multiple server processes can handle requests arriving on a single port on the system.

6. Below are several problems with the kernel network stack that arise in multicore systems desiring high-performance network I/O. For each problem below, describe one technique studied in class that attempts to solve the stated problem. You are required to provide a 1–2 sentence description of the mechanism and how it fixes the stated problem, and not just its name.

(a) Buffers to store packets are dynamically allocated and deallocated in the kernel, leading to dynamic memory allocation overheads.

(b) The payload of a packet is copied multiple times, once from the NIC to kernel memory, and once again from kernel memory to user space memory.

(c) Multiple threads of an application running on different cores all contend for a lock to accept connections on the shared listen socket.

(d) Opening a new socket requires a lock on the per-process file descriptor table and other unnecessary file-system related locking overheads.

7. Consider a multicore system running a TCP-based multi-threaded key-value store application. The incoming traffic to the system consists of new TCP connection requests, and get/put re-quests over the established TCP connections. In order to distribute the interrupt load across all cores, the NIC partitions incoming packets into multiple hardware queues using the hash of the 4-tuple (source and destination IP address and port) of the packet. The interrupts from each hardware queue are delivered to a separate core. The interrupts are processed via the regular Linux network stack on the various cores thereafter. The key-value store application consists of multiple threads, all of which access a shared hashmap data structure containing the key-value pairs.

(a) Are the interrupts generated for all packets of a certain TCP flow guaranteed to be delivered to the same core by the NIC? Answer yes/no and justify.

(b) Are all get requests for a certain key guaranteed to be handled by the same core at the application layer? Answer yes/no and justify