MATURI VENKATA SUBBA RAO (MVSR) ENGINEERING COLLEGE
(Nadergul, Hyderabad)
Department of Computer Science and Engineering

Subject: OPERATING SYSTEMS LAB
RECORD MANUAL

Class: B.E- CSE (DS) - IV SEM
Acad. Yr: 2024-25

TABLE OF CONTENTS

3.3	Single function handling multiple threads using many to one thread model.	04-03-2025
	Part – IV (CPU scheduling algorithms)	
4.1	Simulation of FCFS CPU scheduling algorithm	11-03-2025
4.2	Simulation of SJF CPU scheduling algorithm	11-03-2025
4.3	Simulation of Round Robin (RR) CPU scheduling algorithm	18-03-2025
4.4	**Case Study:** Comparing CPU Scheduling Algorithms	25-03-2025
	Part – V (Inter Process Communication Techniques)	
5.1	Implementing Process Communication using Pipes	01-04-2025
5.2	Implementing Process Communication using Shared memory	01-04-2025
5.3	Implementing Process Communication using Message Queues	01-04-2025
	Part – VI (Process Synchronization)	
6.1	Implementing Dining-Philosophers Problem	15-04-2025
6.2	Implementing Producer-Consumer Problem	22-04-2025
6.3	Implementing Readers-Writers Problem	22-04-2025

PART-I SHELL SCRIPTS

SHELL

- It is a Command-Line Interpreter that executes commands to interact with the OS.
- It provides a basic Console User Interface for Unix/Linux based operating systems.
- A user uses a terminal emulator to give the commands.
- A shell is like a program that runs other programs.

PURPOSE OF SHELL: A shell is a basic level of interaction with the OS by a user. A shell is used for

- Reading/Printing text
- Storing & Accessing variables
- ALU computation
- Command substitutions
- Perform Control statements
- Creating & accessing sub routines
- File manipulation
- Process creation/execution

etc.

TYPES OF SHELL

a) Bourne shell (sh)
b) C Shell (csh)
c) Korn shell (ksh)
d) Bourne Again shell (bash) — It is the default shell.
e) Debian Almquist Shell (dash)
f) Restricted Bourne Again shell (rbash)
and etc.

SHELL SCRIPT

- A shell script is a computer program that contains a collection of Shell Commands to perform a task.
- A terminal emulator is used to run a shell script.
- A shell script can run either in Interactive mode or Script mode.

Writing, Executing a shell script

1) Using Interactive mode
    a) Open terminal and type shell command directly
2) Using script mode (.sh file)
    a) Create a file ***filename.sh*** extension and type shell commands
    b) Run using ***bash filename.sh*** in the terminal

1.1 Programs on conditional statements (if,if-else,if-elif-else)

AIM: To demonstrate various usages of IF conditions using Shell Scripts.

PROCESS:

The if-elif-fi statement is the one level advance form of control statement that allows Shell to make correct decisions out of several conditions.

Syntax

```
if [ Condition1 ]
then
        Statement(s)
elif [ Condition2 ]
then
        Statement(s)
elif [ Condition3 ]
then
        Statement(s)
....
else
        Statement(s)
fi
```

1.1.1 Write a shell script to check if the given number is even or odd.

PROGRAM:

```
#!/bin/bash
echo "Shell script to find even or odd"
read -p "Enter a number:" a
if [ $((a%2)) -eq 0 ]
then
echo $a "is an even number"
else
echo $a "is an odd number"
fi
```

OUTPUT:

1.1.2 Write a shell script to Generate two random numbers and guess whether their sum is correct or not.

PROGRAM:

```
#!/bin/bash
a=$(($RANDOM%100))
b=$(($RANDOM%100))
c=$((a+b))
echo -n $a + $b =?
read ans
if [ $c -eq $ans ]
then
        echo "Your Answer is correct"
else
        echo "Your answer is wrong..Correct answer=" $c
fi
```

OUTPUT:

1.1.3 Write a shell script to Check given string is a file or a directory

PROGRAM:

```
#!/bin/sh
read -p 'Enter a name:' n
if [ -f $n ]
then
        echo 'It is a file'
elif [ -d $n ]
then
        echo 'It is a directory'
else
        echo 'Invalid name entered'
fi
```

OUTPUT:

1.2 Programs to demonstrate batch cases

AIM: To demonstrate how cases can be handled in the shell using a simple arithmetic calculator.
PROCESS:
→ Unix/Linux shell supports case…esac statement which uses an expression to evaluate and several different statements to execute based on the value of the expression.
→ The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

Syntax:

case variable in
pattern-1)
Statement(s)
;;
pattern-2)
Statement(s)
;;
....
pattern-N)
Statement(s)
;;
**)*
Statement(s)
;;
esac

1.2.1 Program to perform arithmetic operations using case statement

PROGRAM:

```
#!/bin/bash
read -p "enter two numbers:" a b
read -p "Enter an operator:" op
case $op in
        '+') echo "Addition=" $((a+b));;
        '-') echo "Subtraction=" $((a-b));;
        '*') echo "Product=" $((a*b));;
        '/') echo "division=" $((a/b));;
        '%') echo "remainder=" $((a%b));;
        *) echo "Invalid"
Esac
```

OUTPUT:

1.3 Programs to demonstrate loops

→ Loops are a powerful programming tool that enable you to execute a set of commands repeatedly.

→ While loop would execute given commands until given condition remains true whereas until loop would execute until a given condition becomes true.

Syntax:

```
Variable Initialization
while [ Condition ]
do
        Statement(s)
        Variable Updation
done
```

→ For loop also works exactly like while loop but it is an enhanced version of while. In for loop, variable declaration,condition,variable updation are given on the same statement line.

→ Once a variable is initialized, condition is checked and if it satisfies,statements in the do block will be executed and the iteration variable is updated.

Syntax:

```
for((variable Initialization; condition ; VariableUpdation))
do
        Statement(s)
done
```

1.3.1 Write a shell script to generate fibonacci series using a for Loop.

PROGRAM:

```
read -p "enter N:" n
a=0 b=1
echo -n $a $b
for ((i=2;i<$n;i++))
do
        c=$((a+b))
        echo -n " "$c
        a=$b b=$c
done
Echo
```

OUTPUT:

1.3.2 Write a shell script to find the sum of N natural numbers using a while Loop.

PROGRAM:

```
#!/bin/bash
read -p "Enter N:" n
echo "Sum of First $n Numbers is:"
i=1 sum=0
while [ $i -le $n ]
do
        echo -n $i
        if [ $i -ne $n ]
        then
        echo -n "+"
        fi
        sum=$(($sum+$i))
        i=$(($i+1))
done
echo "="$sum
```

OUTPUT:

1.4 Programs to demonstrate Arrays

→ An array is a collection of elements that can be accessed through a single variable. In shell scripts, arrays can store similar or dissimilar values.

Array declaration & initialization :

arrayVariable = (value1 value2 value3 ... valueN)

Note: Each element of the array is separated by a space.

1.4.1 Write a shell script to find the sum of array elements.

PROGRAM:

```
read -p "Enter N:" n
echo "enter $n elements"
sum=0
for((i=0;i<$n;i++))
do
        read a[$i]
done
echo -n "Array is:"
for((i=0;i<$n;i++))
do
        echo -n " " $((a[i]))
        sum=$((sum+$((a[i]))))
done
echo
echo "sum=" $sum
```

OUTPUT:

Part – II (Processes & System calls)

2.1 Creating a new process & displaying process IDs

AIM: To create a process and to display the process ID.
PROCESS :

→ A process means any program under execution. A variable ***pid_t*** is used to refer to a process. Below function retrieves the process id.
pid_t getpid();

→ Each and every process will be associated with a parent process. If want to know its id,we use *pid_t getppid().*

Fork() system call :
→ This fork system call will create a child process which is a copy of the parent process itself. But in the parent process it (the fork system call)will return the id of the child (which is >0)and in the child process it will return 0,if it returns -1 the child can not be created. All variables, data structures will be inherited from parent to child.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
        pid_t pid;
        pid=fork();
        if(pid>0)
        {
                printf("Parent Process Created.Its Id=%d\n",getpid());
        }
        else if(pid==0)
        {
                printf("Child Process Created.Its Id=%d\n",getpid());
        }
        else
        printf("Error in process creation\n");
        return 0;
}
```

OUTPUT:

2.2 Demonstration of parent waiting for a child process to end.
AIM : To Write a program to make a parent process wait for child process
PROCESS:
→ By default, the parent process will start and complete first before the child process which is created using fork(). This leads to the child process to become an orphan process.
→ wait() : This function is used to make the parent process wait till the child process terminates.
→ sleep(): This function is used to stop a process for specified seconds.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<wait.h>
int main()
{
        pid_t pid;
        pid=fork();
        if(pid>0)
        {
        printf("Parent Process Started\n");
        wait(NULL);
        printf("Parent Process Completed\n");
        }
        else if(pid==0)
        {
        printf("Child Process Started\n");
        sleep(5);
        printf("Child Process Completed\n");
        }
        else
        printf("Error in process creation\n");
        return 0;
}
```

OUTPUT:

2.3. Demonstration of zombie process
AIM : Write a program to create a zombie process
PROCESS :
→ A zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state".

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<wait.h>
void main()
{
        pid_t p;
        printf("main starts\n");
        p=fork();
        if(p==0)
        {
                printf("Child executed\n");
        }
        else if(p>0)
        {
                printf("Parent process running\n");
                pause();
                printf("Parent process completed\n");
        }
        printf("main ends\n");
}
```

OUTPUT:

2.4. Demonstration of execl()
AIM : Write a program to create a child process different from parent process
PROCESS :
→ execl() is used to erase the current process and replace it with desired program.

Syntax: execl(char* path,char* program,char*arg1,char* arg2,… ..... NULL);

PROGRAM :

a) **p1.c**

```
#include<stdio.h>
#include<unistd.h>
void main()
{
        printf("p1 starts\n");
        execl("./p2","p2",NULL);
        printf("p1 ends\n");
}
```

b) **p2.c**

```
#include<stdio.h>
#include<unistd.h>
void main()
{
        printf("p2 starts\n");
        printf("p2 ends\n");
}
```

Note: Compile above files as

```
cc p1.c -o p1
cc p2.c -o p2
```

Run the above files as:

```
./p1
```

OUTPUT:

2.5 Copying contents from one file to other file using file system calls
AIM: To perform file copy using file system calls.
PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
void main()
{
        char src[100],dest[100];
        int fd1,fd2;
        char b;
        printf("Enter source file:");
        scanf("%s",src);
        printf("Enter destination file:");
        scanf("%s",dest);
        fd1=open(src,O_RDONLY,NULL);
        fd2=open(dest,O_WRONLY | O_CREAT,0666);
        while(read(fd1,&b,1))
        {
                write(fd2,&b,1);
        }
        printf("File copy completed\n");
}
```

OUTPUT:

Part – III (Threads)

→ A thread is a basic unit of execution under a process. It is a sequential flow of tasks carried under a process.
→ Used to implement multi-tasking concurrently.
→ Few things about threads:

a) Thread share code section, data section and file/IO sections of process
b) Each thread has their own thread id,program counter,stack set and register set
c) All threads share same memory area of process

We use POSIX Pthread API to implement threads in C.
We use phtread.h header file for creating and managing threads in C.

Basic Steps to create a Thread

a) Define a function with below syntax

```
void * functionName(void* args)
{
        //thread logic
}
```

b) Define main function int main(){}
c) Declare a thread variable in main
pthread_t tid;
d) Create and Map the thread using below function call
pthread_create(&tid,NULL,functionName,NULL);

Syntax to create a thread:
int pthread_create(pthread_t thread, pthread_attr_t* attr, fun-name, void* msg);*

Other thread methods are:

a) **pthread_exit() :** Used to terminate a thread
pthread_exit(int status);
b) **pthread_self() :** Used to fetch thread id
pthread_t T=pthread_self();
c) **pthread_equal() :** Used to compare two threads
pthread_equal(pthread_t T1,pthread_t T2);
d) **pthread_join() :** Used to wait a parent thread for child thread completion
pthread_join(pthread_t ChildThread, void args);*

3.1 Create and run a simple thread and display its thread id.

AIM: To create and run a thread and also print its id.

PROGRAM:

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
void* demo(void* args)
{
        printf("Thread executed...Pid=%d\n",getpid());
        printf("Thread ID=%ld\n",pthread_self());

}
void main()
{
        pthread_t tid;
        printf("Main starts...Pid=%d\n",getpid());
        pthread_create(&tid,NULL,demo,NULL);
        pthread_join(tid,NULL);
        printf("Main ends\n");
}
```

OUTPUT:

3.2 Sharing variables among threads and the main process (many to many) AIM : To illustrate sharing of variables between threads and main process

PROGRAM:

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
int a,b;
void* add(void* args)
{
        printf("Addition=%d\n",(a+b));

}
void* sub(void* args)
{
                printf("Subtraction=%d\n",(a-b));
}
void main()
{
        pthread_t tid1,tid2;
        printf("Main starts...Pid=%d\n",getpid());
        printf("Enter values for a,b:");
        scanf("%d%d",&a,&b);
        pthread_create(&tid1,NULL,add,NULL);
        pthread_create(&tid2,NULL,sub,NULL);
        pthread_join(tid1,NULL);
        pthread_join(tid2,NULL);
        printf("Main ends\n");
}
```

OUTPUT:

3.3 Single function handling multiple threads using many to one thread model.

AIM: To illustrate how a single function can handle more than one thread.

PROGRAM:

```
#include<stdio.h>
#include<pthread.h>
int a,b;
pthread_t tid[3];
void* demo(void* args)
{
        pthread_t temp;
        temp=pthread_self();
        if(pthread_equal(tid[0],temp))
        {
                printf("Thread-0 matched..sum=%d\n",a+b);
        }
        else if(pthread_equal(tid[1],temp))
        {
        printf("Thread-1 matched..subtract=%d\n",a-b);
        }
        else
        {
        printf("Thread-2 matched..product=%d\n",a*b);
        }
}
int main()
{
        int i;
        printf("Enter the values of a,b:");
        scanf("%d%d",&a,&b);
        for(i=0;i<3;i++)
                pthread_create(&tid[i],NULL,demo,NULL);
        for(i=0;i<3;i++)
                pthread_join(tid[i],NULL);
        return 0;
}
```

OUTPUT:

Part – IV (CPU scheduling algorithms)

4.1 Simulation of FCFS CPU scheduling algorithm
AIM: To implement FCFS CPU scheduling algorithm
PROGRAM:

OUTPUT:

4.2 Simulation of SJF CPU scheduling algorithm
AIM: To implement SJF CPU scheduling algorithm
PROGRAM:

OUTPUT:
4.3 Simulation of Round-Robin CPU scheduling algorithm
AIM: To implement Round-Robin CPU scheduling algorithm
PROGRAM:

OUTPUT:

4.4 CASE STUDY: Comparing CPU Scheduling Algorithms

Consider the processes below:

Process	Arrival time	Burst time	Priority
P1	4	2	5
P2	1	8	2
P3	3	1	3
P4	0	10	4
P5	2	4	1

Compare the following and draw Gantt Charts.Calculate average waiting time,average turnaround time for all scheduling algorithms. Assume a time quantum of 3ms for the RR algorithm.

Time	FCFS	SJF	RR	SRTF	Priority (Preemptive)
0	p1	p4	p1	p5	p3

1	…	…	…	…	…
2	…	…	…	…	…
3	…	…	…	…	…
4	…	…	…	…	…

Part-V : Inter Process Communication Techniques

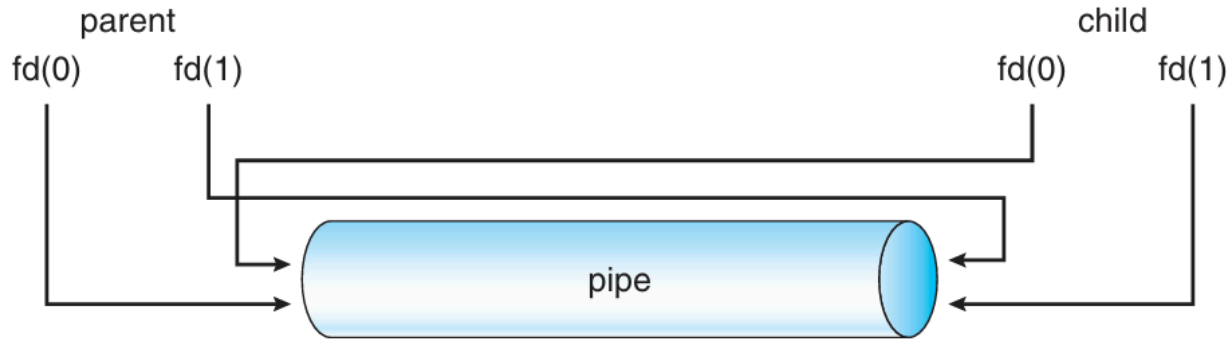
Two processes can communicate with each other by exchanging data between them. Techniques that we use are:

a) PIPES b) Shared Memory c) MessagePassing

A] Concept of PIPE

- A pipe is a conduit that allows two processes to communicate.
- A simple and basic IPC mechanism used in UNIX/LINUX based systems.
- A pipe is a unidirectional and half duplex communication.
- Pipe is treated as a special type of file such that it uses regular file system calls to perform read and write operations for process communication.
- A file descriptor array of size 2 is used to perform read,write operations.

Ex: fd[0] will be used for reading data from pipe and fd[1] is used to write data to pipe.



Steps to create a PIPE

a) Create a file descriptor array of size 2.
int fd[2];

b) Create a process id
pid_t pid;

c) Create a pipe using pipe(fd)
pipe(fd)

d) Fork a child process
pid=fork();

e) Write data inside a parent process
write(fd[1], "some data", int size);

f) Read and display data inside a child process
*read(fd[0],char *s, int size);*

5.1: Implementation of PIPES

AIM: To write a C program that implements the Concept of PIPES to share data by two processes.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<wait.h>
int main()
{
        pid_t pid;
        int pip[2];
        char s[20];
        pipe(pip);
        pid=fork();
        if(pid>0)
        {
                printf("Parent running....\n");
                write(pip[1],"Hello world",20);
                printf("Data written to pipe\n");
                wait(NULL);
                printf("Parent completed....\n");
        }
        else if(pid==0)
        {
                printf("child running...\n");
                read(pip[0],s,20);
                printf("Data from pipe=%s\n",s);
                printf("child completed...\n");
        }
        return 0;
}
```

OUTPUT:

B] Shared Memory

- It is an IPC mechanism in which two processes communicate with each other by reading and writing some data to a shared region.
- Process who initiates communication will create a shared region within its address space.
- Other processes will attach to the shared memory to establish a communication.
- Header files required: **sys/shm.h and sys/ipc.h**

System calls required for Shared Memory

a) **shmget()** — Create a shared memory segment and also obtains the identifier of existing one

Syntax:

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

Example:

int shmid= shmget(1234,20,IPC_CREAT|0666);

b) **shmat()** — Attach a process to shared memory

Syntax:

char shmat(int shmid,char* shmaddr, int flag);*

Example:

*char *s=shmat(shmid,NULL,0);*

c) **shmdt()** — Detach a process from shared memory

Syntax:

*int shmdt(const void *shmaddr);*

Example:

shmdt(s);

d) **shmctl()** — Deallocate/ remove shared memory

Syntax:

*int shmctl(int shmid, int operation, struct shmid_ds *buf);*

Example:

shmctl(shmid,0,NULL);

5.2: Implementation of Shared Memory

AIM: To write a C program that implements the Concept of Shared Memory to share data by two processes.

PROGRAM:

Save file as sharedwrite.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int main()
{
        char *s;
        int shmid=shmget(1234,20,IPC_CREAT|0666);
        s=shmat(shmid,NULL,0);
        printf("Enter a msg:");
        scanf("%s",s);
        return 0;
}
```

Save file as sharedread.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int main()
{
        char *s;
        int shmid=shmget(1234,20,IPC_CREAT|0666);
        s=shmat(shmid,NULL,0);
        printf("Msg received:%s\n",s);
        return 0;
}
```

PROCEDURE:

→Compile above files using below commands:

cc sharedwrite.c -o sharedwrite

cc sharedread.c -o sharedread

→Run the first file using below command

./sharedwrite

→Open a new terminal window and run the second file using below command

./sharedread

OUTPUT:

C] Message Passing

- Message queues are also one of the IPC techniques, in which we send messages from one process to another process in the form of blocks (structures in c).
- Just like in shared memory, first we have to get the msgqueue id using the key of the msgqueue and then we can either send a message to a process or receive a message from another process.
- A structure is defined as below for allocating message queue

```
stuct msgbuf
{
        int mtype;
        char mtext[20];
}m;
```

Header files required: **sys/msg.h and sys/ipc.h**

Following functions are used for implementing message queues

a) **msgget()** – creates a message queue segment and returns a non-negative id

Syntax:

int msgget(key_t key, int msgflg);

Example:

int msgid=msgget(1234,IPC_CREAT|0666);

b) **msgsnd()** – used to send a message to the message queue

Syntax:

*int msgsnd(int msgid, struct *msgbuf, size_t msgsize, int msgflag);*

Example:

msgsnd(msgid,&mbuff,20,0);

c) **msgrcv()** – used to receive a message from the message queue.

Syntax:

```
int msgrcv(
        int msgid, struct *msgbuf,
        size_t msgsize, int msgType,int msgflag
        );
```

Example:

msgrcv(msgid,&mbuf,20,1,0);

5.3: Implementation of Message Passing

AIM: To write a C program that implements the Concept of Message Passing to exchange data by two processes.

PROGRAM:

Save file as messagesend.c

```
#include<stdio.h>
#include<sys/msg.h>
#include<sys/ipc.h>
struct msgbuf
{
        long mtype;
        char mtext[20];
};
int main()
{
        struct msgbuf m;
        int msgid=msgget(124,IPC_CREAT|0666);
        m.mtype=1;
        printf("Enter a message:");
        scanf("%s",m.mtext);
        msgsnd(msgid,&m,20,0);
        printf("Message sent...\n");
        return 0;
}
```

Save file as messagereceive.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
struct msgbuf
{
        long mtype;
        char mtext[20];
};
int main()
{
        struct msgbuf m;
        int msgid=msgget(124,IPC_CREAT|0666);
        m.mtype=1;
        msgrcv(msgid,&m,20,1,0);
        printf("The message received=%s \t %d\n",m.mtext,msgid);
        return 0;
}
```

PROCEDURE:

→Compile above files using below commands:

```
cc messagesend.c -o messagesend
cc messagereceive.c -o messagereceive
```

→Run the first file using below command

```
./messagesend
```

→Open a new terminal window and run the second file using below command

```
./messagereceive
```

OUTPUT: