PROGRAM 4

4. Develop a C program which demonstrates interprocess communication between a reader process and writer process. Use mkfifo, open, read,write and close API's in your program.

DESCRIPTION:

The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes. The kernel presents and accepts data from the channel as a process reads and writes that channel. To a process then, all input and output operations are synchronous and unbuffered.

SYSTEM CALLS USED:

System calls are functions that a programmer can call to perform the services of the operating system.

Open (): Open () system call to open a file. open () returns a file descriptor, an integer specifying the position of this open n file in the table of open files for the current process.

The return value is the descriptor of the file. Returns -1 if the file could not be opened. The first parameter is path name of the file to be opened and the second parameter is the opening mode

Close (): Close () system call to close a file.

It is always good practices to close files when not needed as open files do consume resources and all normal systems impose a limit on the number of files that a process can hold open.

Read (): The read () system call is used to read data from a file or other object identified by a file descriptor.

fd is the descriptor, buf is the base address of the memory area into which the data is read and MAX_BUF is the maximum amount of data to read. The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read. An attempt to read beyond the end of a file results in a return value of zero.

Write (): The write () system call is used to write data to a file or other object identified by a file descriptor.

PROGRAM:

/*reader process*/

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
 #define MAX_BUF 1024
 int main()
{
        int fd;
        /* A temp FIFO file is not created in reader */
        char *myfifo = "/tmp/myfifo.txt";
        char buf[MAX_BUF];
        /* open, read, and display the message from the FIFO */
        fd = open(myfifo, O_RDONLY);
        read(fd, buf, MAX_BUF);
        printf("Writer: %s\n", buf);
        close(fd);
        return 0;
}
```

/*Writer Process*/

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        int fd;
        char buf[1024];
        /* create the FIFO (named pipe) */
        char * myfifo = "/tmp/myfifo.txt";
        mkfifo(myfifo, 0666);
        printf("Run Reader process to read the FIFO File\n");
        fd = open(myfifo, O_WRONLY);
        write(fd,"Hi", sizeof("Hi"));
        /* write "Hi" to the FIFO */
        close(fd);
        unlink(myfifo); /* remove the FIFO */
        return 0;
}
```

OUTPUT

```
manju@Manjushree:~/os_lab$ cc ip_write.c
manju@Manjushree:~/os_lab$ ./a.out
Run Reader process to read the FIFO File
manju@Manjushree:~/os_lab$ cc ip_read.c
manju@Manjushree:~/os_lab$ ./a.out
Writer: Hi
```

PROGRAM 5

5. Develop a C program to simulate Banker's algorithm for deadlock avoidance.

DESCRIPTION:

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type. When the user requests a set of resources, the system must determine whether the allocation of each resource will leave the system in safe state.

PROGRAM

```
#include<stdio.h>
struct process
{
        int all[6],max[6],need[6],finished,request[6];
}p[10];

int avail[6],sseq[10],ss=0,check1=0,check2=0,n,pid,nor,nori,work[6];
int main()
{
        int safeseq(void);
        int ch,k,i=0,j=0,pid,ch1;
        int violationcheck=0,waitcheck=0;
        do
        {
        printf("\n1.Input\n2.New Request\n3.Safe State or Not\n4.Print\n5.Exit\nEnter your
choice:");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1: printf("\nEnter the number of processes:");
```

```
                scanf("%d",&n);
                printf("\nEnter the number of resources:");
                scanf("%d",&nor);
                printf("\nEnter the available resources:");
                for(j=0;j<n;j++)
                {
                        for(k=0;k<nor;k++)
                        {
                                if(j==0)
                        {
                                printf("\nFor Resource Type %d:",k);
                                scanf("%d",&avail[k]);
                        }
                        p[j].max[k]=0;
                        p[j].all[k]=0;
                        p[j].need[k]=0;
                        p[j].finished=0;
                        p[j].request[k]=0;
                        }
                }
                for(i=0;i<n;i++)
                {
                        printf("\nEnter Max and Allocated Resources for P %d :",i);
                        for(j=0;j<nor;j++)
                        {
                        printf("\nEnter the Max of Resources %d:",j);
                        scanf("%d",&p[i].max[j]);
                        printf("\nAllocation of Resources %d:",j);
                        scanf("%d",&p[i].all[j]);
                        if(p[i].all[j]>p[i].max[j])
                        {
                                printf("\nAllocation should be less than or equal to
Max\n");
                                j--;
                        }
                        else
                                p[i].need[j]=p[i].max[j]-p[i].all[j];
                        }
                }
        break;

        case 2:
                violationcheck=0;
                waitcheck=0;
                printf("\nRequesting Process ID:\n");
                scanf("%d",&pid);
                for(j=0;j<nor;j++)
                {
                        printf("\nNumber of Request for Resource %d:",j);
                        scanf("%d",&p[pid].request[j]);
```

```
                if(p[pid].request[j]>p[pid].need[j])
                violationcheck=1;
                if(p[pid].request[j]>avail[j])
                waitcheck=1;
        }
        if(violationcheck==1)
                printf("\nThe Process Exceeds its Max needs: Terminated\n");
        else if(waitcheck==1)
                printf("\nLack of Resources: Process State - Wait\n");
        else
        {
                for(j=0;j<nor;j++)
                {
                        avail[j]=avail[j]-p[pid].request[j];
                        p[pid].all[j]=p[pid].all[j]+p[pid].request[j];
                        p[pid].need[j]=p[pid].need[j]-p[pid].request[j];
                }
                ch1=safeseq();
                if(ch1==0)
                {
                for(j=0;j<nor;j++)
                {
                        avail[j]=avail[j]+p[pid].request[j];
                        p[pid].all[j]=p[pid].all[j]-p[pid].request[j];
                        p[pid].need[j]=p[pid].need[j]+p[pid].request[j];
                }
                }
                else if(ch1==1)
                printf("\nRequest committed.\n");
        }
break;

case 3:
        if(safeseq()==1)
                printf("\nThe System is in Safe State\n");
        else
                printf("\nThe System is not in Safe State\n");
break;
case 4:
        printf("\nNumber of Process:%d\n",n);
        printf("\nNumber of Resources:%d\n",nor);
        printf("\nPid\tMax\tAllocated\tNeed\n");
        for(i=0;i<n;i++)
        {
                printf(" P%d :",i);
                for(j=0;j<nor;j++)
                printf(" %d ",p[i].max[j]);
                printf("\t");
                for(j=0;j<nor;j++)
                        printf(" %d ",p[i].all[j]);
```

```
                                printf("\t");
                                for(j=0;j<nor;j++)
                                printf(" %d ",p[i].need[j]);
                                printf("\n");
                        }
                        printf("\nAvailable:\n");
                        for(i=0;i<nor;i++)
                                printf(" %d ",avail[i]);
                break;

                case 5:  break;
        }
        }while(ch!=5);
        return 0;
}

int safeseq()
{
        int tj,tk,i,j,k;
        ss=0;
        for(j=0;j<nor;j++)
        work[j]=avail[j];
        for(j=0;j<n;j++)
        p[j].finished=0;
        for(tk=0;tk<nor;tk++)
        {
        for(j=0;j<n;j++)
        {
                if(p[j].finished==0)
                {
                        check1=0;
                        for(k=0;k<nor;k++)
                        if(p[j].need[k]<=work[k])
                        check1++;
                        if(check1==nor)
                        {
                                for(k=0;k<nor;k++)
                                {
                                work[k]=work[k]+p[j].all[k];
                                p[j].finished=1;
                                }
                                sseq[ss]=j;
                                ss++;
                        }
                }
        }
        }
        check2=0;
        for(i=0;i<n;i++)
                if(p[i].finished==1)
```

```
                check2++;
        printf("\n");
        if(check2>=n)
        {
        for(tj=0;tj<n;tj++)
                printf("p%d",sseq[tj]);
        return 1;
        }
        else
                printf("\nThe System is not in Safe State\n");
        return 0;
}
```

O/p:
1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:1

Enter the number of processes:5

Enter the number of resources:3

Enter the available resources:
For Resource Type 0:3

For Resource Type 1:3

For Resource Type 2:2

Enter Max and Allocated Resources for P 0 :
Enter the Max of Resources 0:7

Allocation of Resources 0:0

Enter the Max of Resources 1:5

Allocation of Resources 1:1

Enter the Max of Resources 2:3

Allocation of Resources 2:0

Enter Max and Allocated Resources for P 1 :
Enter the Max of Resources 0:3

Allocation of Resources 0:2

Enter the Max of Resources 1:2

Allocation of Resources 1:0

Enter the Max of Resources 2:2

Allocation of Resources 2:0

Enter Max and Allocated Resources for P 2 :
Enter the Max of Resources 0:9

Allocation of Resources 0:3

Enter the Max of Resources 1:0

Allocation of Resources 1:0

Enter the Max of Resources 2:2

Allocation of Resources 2:2

Enter Max and Allocated Resources for P 3 :
Enter the Max of Resources 0:2

Allocation of Resources 0:2

Enter the Max of Resources 1:2

Allocation of Resources 1:1

Enter the Max of Resources 2:2

Allocation of Resources 2:1

Enter Max and Allocated Resources for P 4 :
Enter the Max of Resources 0:4

Allocation of Resources 0:0

Enter the Max of Resources 1:3

Allocation of Resources 1:0

Enter the Max of Resources 2:3

Allocation of Resources 2:2

1.Input
2.New Request
3.Safe State or Not

4.Print
5.Exit
Enter your choice:3

p1p3p4p0p2
The System is in Safe State

1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:4

Number of Process:5

Number of Resources:3

Pid	Max	Allocated	Need
P0 :	7 5 3	0 1 0	7 4 3
P1 :	3 2 2	2 0 0	1 2 2
P2 :	9 0 2	3 0 2	6 0 0
P3 :	2 2 2	2 1 1	0 1 1
P4 :	4 3 3	0 0 2	4 3 1

Available:
3 3 2
1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:2

Requesting Process ID:
1

Number of Request for Resource 0:1

Number of Request for Resource 1:0

Number of Request for Resource 2:2

p1p3p4p0p2
Request committed.

1.Input
2.New Request
3.Safe State or Not
4.Print

5.Exit
Enter your choice:3

p1p3p4p0p2
The System is in Safe State

1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:4

Number of Process:5

Number of Resources:3

Pid	Max	Allocated	Need
P0 :	7 5 3	0 1 0	7 4 3
P1 :	3 2 2	3 0 2	0 2 0
P2 :	9 0 2	3 0 2	6 0 0
P3 :	2 2 2	2 1 1	0 1 1
P4 :	4 3 3	0 0 2	4 3 1

Available:
2 3 0
1.Input
2.New Request
3.Safe State or Not
4.Print
5.Exit
Enter your choice:2

Requesting Process ID:
2

Number of Request for Resource 0:3

Number of Request for Resource 1:2

Number of Request for Resource 2:2

The Process Exceeds its Max needs: Terminated