UNIX SYSTEM CALL - STAT()

Aim:
To Execute a Unix Command in a 'C' program using stat() system call.

Algorithm:
1. Start the program
2. Declare the variables for the structure stat
3. Allocate the size for the file by using malloc function
4. Get the input of the file whose statistics want to be founded
5. Repeat the above step until statistics of the files are listed
6. Stop the program

Program:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdlib.h>
int main(void)
{
        char *path, path1[10];
        struct stat *nfile;
        nfile=(struct stat *)malloc(sizeof(struct stat));
        printf("Ente the name of the file :");
        scanf("%s", path1);
        stat(path1,nfile);

        printf("user id : %d \n", nfile->st_uid);
        printf("block size : %ld \n", nfile->st_blksize);
        printf("last  access time : %ld \n", nfile->st_atime);
        printf("last  modification : %ld \n", nfile->st_atime);
        printf("production mode : %d \n", nfile->st_mode);
        printf("size of file : %ld \n", nfile->st_size);
        printf("number of links : %ld \n", nfile->st_nlink);
}
```

Result:
Thus the program for stat system call has been executed successfully.

UNIX SYSTEM CALL - EXEC()

Aim:

To Execute a Unix Command in a 'C' program using exec() system call.

Algorithm:

1. Start the program
2. Declare the necessary variables
3. Use the prototype execv (filename,argv) to transform an executable binary file into process
4. Repeat this until all executed files are displayed
5. Stop the program.

Program:

```
#include<stdio.h>
#include <stdlib.h>
int fork();
int execv();
int wait();

int main(int argc,char ** argv)
{
        int pid;
        char *arg[]={"/bin/ls -l",0};
        printf("\nParent process");
        pid=fork();
        if(pid==0)
        {
                execv("/bin/ls",arg);
                printf("\nChild process");
        }
        else
        {
                wait();
                printf("\nParent process");
                exit(0);
        }
}
```

Result:

Thus the program for exec() system call has been executed successfully.

UNIX SYSTEM CALL – OPENDIR(), READDIR()

Aim:

To write a C program to display the files in the given directory

Algorithm:

1. Start the program
2. Declare the variable to the structure dirent (defines the file system-independent director and also for DIR
3. Specify the directory path to be displayed using the opendir system call
4. Check for the existence of the directory and read the contents of the directory using readdir system call (returns a pointer to the next active directory entry)
5. Repeat the above step until all the files in the directory are listed
6. Stop the program

Program:

```
#include<stdio.h>
#include<dirent.h>
#include <stdlib.h>
struct dirent *dptr;
int main(int argc, char *argv[])
{
        char buff[256];
        DIR *dirp;
        printf("Enter directory name: ");
        scanf("%s",buff);
        if((dirp=opendir(buff))==NULL)
        {
                printf("Error");
                exit(1);
        }
        while(dptr=readdir(dirp))
        {
                printf("%s\n",dptr->d_name);
        }
        closedir(dirp);
}
```

Result:

Thus the program has been executed successfully.

UNIX SYSTEM CALL - OPEN, READ, WRITE

Aim:

To implement UNIX I/O system calls open, read, write.

Algorithm:

1. Create a new file using creat command (Not using FILE pointer).
2. Open the source file and copy its content to new file using read and write command.
3. Find size of the new file before and after closing the file using stat command.

Program:

```
#include<fcntl.h> //file control
#include<sys/types.h>
#include<sys/stat.h>
#include<stdlib.h>
#include <stdio.h>
int close();
int lseek();
int read();
int write();
static char message[]="hai Hello world";
int main()
{
int fd;
char buffer[80];
fd=open("new2file.txt",O_RDWR|O_CREAT|O_EXCL,S_IREAD|S_IWRITE);
if(fd!=-1)
{
printf("new2file.txt opened for read/write access\n");
write(fd,message,sizeof(message));
lseek(fd,0l,0);
if(read(fd,buffer,sizeof(message))==sizeof(message))
printf("\"%s\" was written to new2file.txt\n",buffer);
else
printf("***Error readind new2file.txt***\n");
}
else
{
close(fd);
printf("***new2file.txt already exists***\n");
exit(0);
}
}
```

Result:

IPC - SHARED MEMORY

Aim:
To implement the interprocess communication using shared memory

Algorithm:
1. Start the program

2. Declare the necessary variables

3. shmat() and shmdt() are used to attach and detach shared memory segments. They are prototypes as follows:

void *shmat(int shmid, const void *shmaddr, int shmflg); int shmdt(const void *shmaddr);

4. shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr

5. Shared1.c simply creates the string and shared memory portion.

6. Shared2.c attaches itself to the created shared memory portion and uses the string (printf)

7. Stop the program.

Program:

Program 1

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#define SHMSZ 27
int main()
{
        char c;
        int shmid,i;
        key_t key;
        char *shm, *s;
        key = 5678;
        if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0)
        {
                perror("shmget");
                exit(1);
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
        {
                perror("shmat");
                exit(1);
        }
```

```
        for (s = shm,i=0;i<26; s++,i++) {
                *s='A'+i;
        }
        *s='\0';
        shmdt(shm);
        return 0;
}
```

Program 2

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#define SHMSZ 27
int main()
{
        int shmid;
        key_t key;
        char *shm, *s;
        key = 5678;
        if ((shmid = shmget(key, SHMSZ, 0666)) < 0)
        {
                perror("shmget");
                exit(1);
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
        {
                perror("shmat");
                exit(1);
        }
        for (s = shm; *s != '\0'; s++)
                putchar(*s);
        putchar('\n');
        *shm = '*';
        shmdt(shm);
        exit(0);
}
```

Result:

Thus the program has been executed successfully and the output is verified.

IPC - MESSAGE PASSING

Aim:
To implement the interprocess communication using message passing.

Algorithm:
1. Start the program

2. Create two files msgsend and msgrecv.c

3. Msgsend creates a message queue with a basic key and message flag msgflg = IPC_CREAT | 0666 -- create queue and make it read and appendable by all, and sends one message to the queue.

4. Msgrecv reads the message from the queue

5. A message of type (sbuf.mtype) 1 is sent to the queue with the message ``Did you get this?"

6. The Message queue is opened with msgget (message flag 0666) and the *same* key as message_send.c

7. A message of the *same* type 1 is received from the queue with the message ``Did you get this?" stored in rbuf.mtext.

8. Stop the program.

Program:

Message_send

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include<stdlib.h>
#define MSGSZ 128
typedef struct msgbuf
{
        long mtype;
        char mtext[MSGSZ];
} message_buf;
int main()
{
        int msqid;
        char msg[MSGSZ];
        printf("\n enter the message : ");
        scanf("%s",msg);
        int msgflg = IPC_CREAT | 0666;
        key_t key;
        message_buf sbuf;
        size_t buf_length;
```

```
        key = 1234;
        (void) fprintf(stderr, "\nmsgget: Calling msgget(%#x,\%#o)\n", key, msgflg);
        if ((msqid = msgget(key, msgflg )) < 0)
        {
                perror("msgget");
                exit(1);
        }
        else
        (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
        sbuf.mtype = 1;
        (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
        (void) strcpy(sbuf.mtext, msg);
        (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
        buf_length = strlen(sbuf.mtext) + 1 ;
        if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
                printf ("%d, %ld, %s, %ld\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
                perror("msgsnd");
                exit(1);
        }
        else
                printf("Message: \"%s\" Sent\n", sbuf.mtext);
        exit(0);
}
```

Message receive

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSZ 128
typedef struct msgbuf {
long mtype;
char mtext[MSGSZ];
} message_buf;
void main()
{
        int msqid; key_t;
        key;
        message_buf rbuf;
        key = 1234;
        if ((msqid = msgget(key, 0666)) < 0)
        {
                perror("msgget");
                exit(1);
        }
        if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0)
        {

                perror("msgrcv");
                exit(1);
        }
```

```
        printf("%s\n", rbuf.mtext);
        exit(0);
}
```

Result:

Thus the program has been executed successfully and the output is verified.

SEMAPHORE: PRODUCER CONSUMER PROBLEM

Aim:
To write a program to solve the Producer Consumer Problem using Semaphores

Algorithm:
1. START the program
2. If the request is for Producer
    a. Get the count of number of items to be produced
    b. Add the count with current Buffer size
    c. If the new Buffer size is full
        Don't produce anything; display an error message
        Producer has to wait till any consumer consume the existing item
    d. Else
        Produce the item
        Increment the Buffer by the number of item produced
3. If the request is for Consumer
    a. Get the count of number of items to be consumed
    b. Subtract the count with current Buffer size
    c. If the new Buffer size is lesser than 0
        Don't allow the consumer to consume anything; display an error message
        Consumer has to wait till the producer produce new items
    d. Else
        Consume the item
        Decrement the Buffer by the number of item consumed
4. STOP the program

Program:

```
#include<stdio.h>
#include<stdlib.h>
int n=0,buffersize=0,currentsize=0;

void producer()
{
        printf("\nEnter number of elements to be produced: ");
        scanf("%d",&n);
        if(buffersize >= (currentsize+n))
        {
                currentsize+=n;
                printf("%d Elements produced by producer where buffersize is %d\n", currentsize,
                buffersize);
        }
        else
                printf("\nBuffer is not sufficient\n");
}

void consumer()
```

```
{
        int x;
        printf("\nEnter no. of elements to be consumed: ");
        scanf("%d",&x);
        if(currentsize>=x)
        {
                currentsize-=x;
                printf("\nNumber of elements consumed: %d, Number of Elements left: %d", x,
currentsize);
        }
        else
        {
                printf("\nNumber of Elements consumed should not be greater than Number of
Elements produced\n");
        }
}

void main()
{
        int c;
        printf("\nEnter maximum size of buffer:");
        scanf("%d",&buffersize);
        do
        {
                printf("\n1.Producer 2.Consumer 3.Exit");
                printf("\nEnter Choice:");
                scanf("%d",&c);
                switch(c)
                {
                case 1:
                        if(currentsize >= buffersize)
                                printf("\nBuffer is full. Cannot produce");
                        else
                                producer();
                        break;
                case 2:
                        if(currentsize == 0)
                                printf("\nBuffer is Empty. Cannot consume");
                        else
                                consumer();
                        break;
                default:
                        exit(1);
                break;
                }
         }
         while(c!=3);
}
```

Result:

Thus the program ~~to implement the SJF (Shortest Job First) CPU scheduling Algorithm~~ was written, executed and the output was verified successfully.