**1. SHARED MEMORY:**

Client:

//SHMClient.C

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 27

void die(char \*s)

{

perror(s);

exit(1);

}

int main()

{

int shmid;

key\_t key;

char \*shm, \*s;

key = 5678;

if ((shmid = shmget(key, MAXSIZE, 0666)) < 0)

die("shmget");

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1)

die("shmat");

//Now read what the server put in the memory.

for (s = shm; \*s != '\0'; s++)

putchar(\*s-32);

putchar('\n');

/\*

\*Change the first character of the

\*segment to '\*', indicating we have read

\*the segment.

\*/

\*shm = '\*';

puts("\nClient exiting\n");

exit(0);

}

Server:

//SHMServer.C

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define MAXSIZE 27

void die(char \*s)

{

perror(s);

exit(1);

}

int main()

{

char c;

int shmid;

key\_t key;

char \*shm, \*s;

key = 5678;

if ((shmid = shmget(key, MAXSIZE, IPC\_CREAT | 0666)) < 0)

die("shmget");

if ((shm = shmat(shmid, NULL, 0)) == (char \*) -1)

die("shmat");

/\*

\* \* Put text into the memory for the

\* other process to read.

\* \*/

s = shm;

for (c = 'a'; c <= 'z'; c++)

\*s++ = c;

/\*

\* Wait until the other process

\* changes the first character of our memory

\* to '\*', indicating that it has read what

\* we put there.

\*/

while (\*shm != '\*'){puts("\nServer waiting\n");

sleep(1);}

puts("\nServer exiting after client read data\n");

exit(0);

}

**2. Unnamed pipe**

**// C program to illustrate**

**// pipe system call in C**

**// shared by Parent and Child**

#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], pid, nbytes;

if (pipe(p) < 0)

exit(1);

/\* continued \*/

if ((pid = fork()) > 0) {

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

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

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

// Adding this line will

// not hang the program

// close(p[1]);

wait(NULL);

}

else {

// Adding this line will

// not hang the program

// close(p[1]);

while ((nbytes = read(p[0], inbuf, MSGSIZE)) > 0)

printf("% s\n", inbuf);

if (nbytes != 0)

exit(2);

printf("Finished reading\n");

}

return 0;

}

**3.Named pipe:**

Writer:

// C program to implement one side of FIFO

// This side writes first, then reads

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

int fd;

// FIFO file path

char \* myfifo = "/tmp/myfifo";

// Creating the named file(FIFO)

// mkfifo(<pathname>, <permission>)

mkfifo(myfifo, 0666);

char arr1[80], arr2[80];

while (1)

{

// Open FIFO for write only

fd = open(myfifo, O\_WRONLY);

// Take an input arr2ing from user.

// 80 is maximum length

fgets(arr2, 80, stdin);

// Write the input arr2ing on FIFO

// and close it

write(fd, arr2, strlen(arr2)+1);

close(fd);

// Open FIFO for Read only

fd = open(myfifo, O\_RDONLY);

// Read from FIFO

read(fd, arr1, sizeof(arr1));

// Print the read message

printf("User2: %s\n", arr1);

close(fd);

}

return 0;

}

Reader:

// C program to implement one side of FIFO

// This side reads first, then reads

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

int fd1;

// FIFO file path

char \* myfifo = "/tmp/myfifo";

// Creating the named file(FIFO)

// mkfifo(<pathname>,<permission>)

mkfifo(myfifo, 0666);

char str1[80], str2[80];

while (1)

{

// First open in read only and read

fd1 = open(myfifo,O\_RDONLY);

read(fd1, str1, 80);

// Print the read string and close

printf("User1: %s\n", str1);

close(fd1);

// Now open in write mode and write

// string taken from user.

fd1 = open(myfifo,O\_WRONLY);

fgets(str2, 80, stdin);

write(fd1, str2, strlen(str2)+1);

close(fd1);

}

return 0;

}

**4. Message queues:**

**sender:**

//IPC\_msgq\_send.c

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define MAXSIZE 128

void die(char \*s)

{

perror(s);

exit(1);

}

typedef struct msgbuf1

{

long mtype;

char mtext[MAXSIZE];

}msgbuf;

void main()

{

int msqid;

int msgflg = IPC\_CREAT | 0666;

key\_t key;

msgbuf sbuf;

size\_t buflen;

key = 1234;

if ((msqid = msgget(key, msgflg )) < 0) //Get the message queue ID for the given key

die("msgget");

//Message Type

sbuf.mtype = 1;

printf("Enter a message to add to message queue : ");

scanf("%[^\n]",sbuf.mtext);

getchar();

buflen = strlen(sbuf.mtext) + 1 ;

if (msgsnd(msqid, &sbuf, buflen, IPC\_NOWAIT) < 0)

{

printf ("%d, %ld, %s, %ld\n", msqid, sbuf.mtype, sbuf.mtext, buflen);

die("msgsnd");

}

else

printf("Message Sent\n");

exit(0);

}

**receiver:**

//IPC\_msgq\_rcv.c

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 128

void die(char \*s)

{

perror(s);

exit(1);

}

typedef struct msgbuf1

{

long mtype;

char mtext[MAXSIZE];

}msgbuf ;

void main()

{

int msqid;

key\_t key;

msgbuf rcvbuffer;

key = 1234;

if ((msqid = msgget(key, 0666)) < 0)

die("msgget()");

//Receive an answer of message type 1.

if (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)

die("msgrcv");

printf("%s\n", rcvbuffer.mtext);

exit(0);

}

**5. Directory system calls**

A directory can be read as a file by anyone whoever has reading permissions for it. Writing a directory as a file can only be done by the kernel. The structure of the directory appears to the user as a succession of structures named directory entries. A directory entry contains, among other information, the name of the file and the i-node of this. For reading the directory entries one after the other we can use the following functions:

#include <sys/types.h>

#include <dirent.h>

**DIR\*** opendir(**const char\*** pathname);

**struct dirent\*** readdir(**DIR\*** dp);

**void** rewinddir(**DIR\***dp);

**int** closedir(**DIR\*** dp);

The *opendir* function opens a directory. It returns a valid pointer if the opening was successful and NULL otherwise.

The *readdir* function, at every call, reads another directory entry from the current directory. The first *readdir* will read the first directory entry; the second call will read the next entry and so on. In case of a successful reading the function will return a valid pointer to a structure of type *dirent* and NULL otherwise (in case it reached the end of the directory, for example).

The *rewinddir* function repositions the file pointer to the first directory entry (the beginning of the directory).

The *closedir* function closes a previously opened directory. In case of an error it returns the value -1.

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#include <stdio.h>

#include <stdlib.h>

void listDir(char \*dirName)

{

DIR\* dir;

struct dirent \*dirEntry;

struct stat inode;

char name[1000];

dir = opendir(dirName);

if (dir == 0) {

perror ("Eroare deschidere fisier");

exit(1);

}

while ((dirEntry=readdir(dir)) != 0) {

sprintf(name,"%s/%s",dirName,dirEntry->d\_name);

lstat (name, &inode);

// test the type of file

if (S\_ISDIR(inode.st\_mode))

printf("dir ");

else if (S\_ISREG(inode.st\_mode))

printf ("fis ");

else

if (S\_ISLNK(inode.st\_mode))

printf ("lnk ");

else;

printf(" %s\n", dirEntry->d\_name);

}

}

int main(int argc, char \*\*argv)

{

if (argc != 2) {

printf ("UTILIZARE: %s nume\_dir\n", argv[0]);

exit(0);

}

printf("Continutul directorului este:\n");

listDir(argv[1]);

}