TERMWORK - 1

Problem Statement

Implementing IPC using Pipes and message queues.

Objectives

1. To practice network programming in UNIX based operating systems.
2. To design & simulate the network in latest simulation tools.
3. To illustrate message controlling mechanisms.

Theory

- Process is a program in execution.
- A process can be of two types :
  - Independent process
  - Co-operating process
- An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.
- Process can communicate with each other through Shared Memory, Message Passing
- Inter-process communication (IPC) is set of interfaces, which is usually programmed in order for the programs to communicate between series of processes.

Methods in Inter-process Communication

- Pipes : This allows flow of data in one direction only. Analogous to simplex systems (Keyboard). Data from the output is usually buffered until input process receives it which must have a common origin.
- Message Queing : This allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are co-ordinated using an API.
- Sockets : This method is mostly used to communicates over a network between a client & a server. It allows for a standard connection which is computer & OS independent.

Creating child process using fork

- fork() creates a new process by duplicating the calling process.
- The new process is referred to as the child process.
- The calling process is referred to as parent process.
- The child process & the parent process run in separate memory space.
- At the time of fork() both memory spaces have the same content.
- fork() : q = fork() → return these three possible values
  - $q < 0$ → error
  - $q = 0$ → child process
  - $q > 0$ → parent process

Pipe() function

- Pipe creates a unidirectional pipe (data channel) for the communication between the two processes.
- Pipe uses two file descriptors :
  - Writing end : fd[1]
  - Reading end : fd[0]

Source Program

```c
# include< unistd.h>
# include< stdio.h>
# include < sys / types.h>
# include < sys / wait. h>

int main()
{
	int fd[2], n ;
	char buffer [100] ;
	pid_t p ;
	pipe(fd);
	p = fork();
	if (p>0)
	{
		printf(" Parent pid = %d \n", getpid());
		printf(" My child's pid is %d \n", p);
		printf(" Passing value to child \n");
		write( fd [1] , "hello \n", 6);
	}
	else
	{
		printf(" Child pid = %d \n", getpid());
		printf(" My parent's pid is %d \n", getppid());
		n = read ( fd[0], buffer , 100);
		printf(" Child received data \n");
		write(1, buffer , n);
	}
}
```

Theory

IPC using message queue

- This allows messages to be passed between processes using either a single queue or several message queue.
- This is managed by system kernel. These messages are coordinated using an API.
- Functions used:
  - i) msgget() : to create a message queue.
  - ii) msgsnd() : to add message to message queue.
  - iii) msgrcv() : to retrieve message from message queue.
  - iv) msgctl() : to delete the message.
- The msgsnd() & msgrcv() system calls are used, respectively, to send messages to, & receive messages from, a message queue. The calling process must have write permission on the message queue in order to send a message, & read permission to receive a message.
- struct msgbuf

```
struct msgbuf
{
    long mtype;     // message type, must be >0
    char mtext[1];  // message data
};
```

- int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg);
- ssize_t msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

Source Code

Receiver

```
# include <sys/ipc.h>
# include < sys/msg.h>
# include <stdio.h>
# include < stdlib.h>
# define MAX 10
long me
struct mesg_buffer
{
    long mesg_type ;
    char mesg_text [100];
} message ;

int main()
{
    key_t key ;
    int msgid ;
    key = fork ("progfile ", 65);
    msgid = msgget (key, 0666 | IPC_CREAT);
    msgrcv (msgid, &message, sizeof (message), 1, 0);
    printf(" Data Received is %s \n", message.mesg_text);
    msgctl( msgid , IPC_RMID , NULL);
    return 0 ;
}
```

Sender

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

struct mesg_buffer
{
    long mesg_type ;
    char mesg_text [100] ;
} message ;

int main()
{
    key_t key ;
    int msgid ;
    key = ftok (" progfile ", 65);
    msgid = msgget ( key , 0666 | IPC_CREAT ) ;
    message.mesg_type = 1 ;
    printf (" Write data ") ;
    fgets ( message.mesg_text , MAX , stdin ) ;
    msgsnd ( msgid , &message , sizeof (message) , 0 ) ;
    printf (" Data sent is : %s \n" , message. mesg_text ) ;
    return 0 ;
}
```

Conclusion

We successfully implemented IPC using pipes and message queues.

Outcomes

- Understood the concept of IPC.
- Implementing IPC using pipes & message queues.

References

W. Richard Stevens, Bill Fenner, Andrew M. Rodoff : " UNIX Network Programming ". Volume 1, Third Edition, Pearson 2004.