TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

LAB REPORT ON.
IPC.

Lab No.: 3

Experiment Date: 2078/9/30

Submission Date: 2078/10/14.

Submitted By:

Name: Soyog Dhakal

Roll No.: 075BCT092.

Group: BCT-D

Submitted To:

Department of
Electronics and computer
Engineering

Title: Inter-process communication (IPC)

Theory:

IPC stands for inter-process communication. IPC is very important in UNIX. It might not look as trivial but suppose we have two processes running in memory, a child process and parent process. The requirement is such that parent has to wait till a key is pressed from within the child and then the parent has to exit.

Inter process communication is the mechanism provided by the operating system that allows process. to communicate with each other. This communication could involve a process letting another process know that some event has occured or the transferring of data from one process to another.

Synchronization in interprocess Communication.

Synchronization is a necessary part of interprocess communicati
It is either provided by the interprocess control mechanism or handled by the communicating processes. Some of the methods to provide Synchronization are as follows:

1. Semaphore:

It is a variable that ~~celebrates~~ controls the access to common resources by multiple process. The two types of semaphores are binary semaphores and counting semaphores.

2. Mutual Exclusion

It requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.

· Barrier.

A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.

Spinlock

This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operations even though it is active.

Approaches to Enterprocess Communication.

The different approaches to implement interprocess communication are given as follows:

· Pipe

A pipe is a data channel that is unidirectional. Two pipes can be used to create two way data channel between two processes This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.

Socket

The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating system use sockets for interprocess communication.

File

A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple process can access a file as required. All operating systems use files for.

data storage.

· Signal

Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one processes to another. Normally, signals are not used to transfer data but are used for remote commands between processes.

Shared memory

Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the process can communicate with each other. All POSIX systems, as well as Windows operating system use shared memory.

Message Queue

Multiple process can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipent retrieves them. Message queues are quite useful for interprocess communication and are used by most operating system.

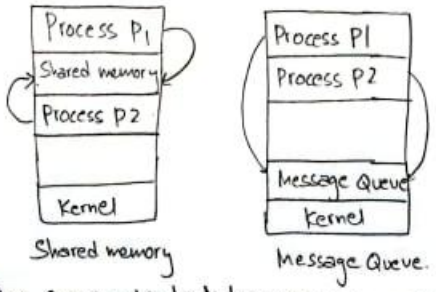


Fig: Approaches to Interprocess Communication

PROGRAM1:

The program provided doesn't work and the parent process waits indefinitely for the child process to execute. This is because each parent and child process has its own memory space for the variables and thus is independent of what happens in other process.
The solution for this is:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char c;
    int pfd[2];
    if (pipe(pfd) > 0)
    {
        printf("error");
    }
    if (!fork())
    {
        close(pfd[0]);
        printf("press a key");
        scanf("%c", &c);
        write(pfd[1], &c, 1);
        exit(0);
    }
    else
    {
        close(pfd[1]);
        read(pfd[0], &c, 1);
        printf("I received the character %c",c);
        exit(0);
    }
}
```

```
press a keyD
I received the character D[1] + Done
gine-In-aizmc0r2.ilk" 1>"/tmp/Microsoft-
```

Program2:

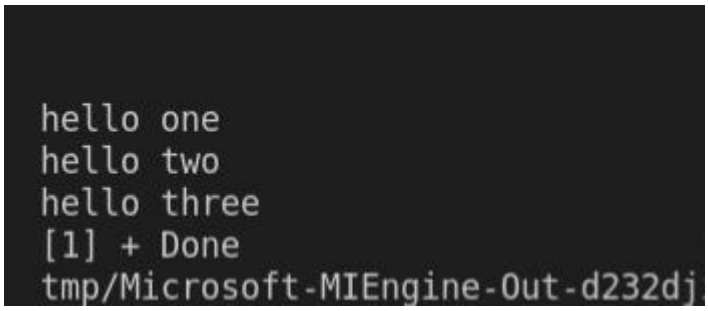
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main()
{
    int pfd[2];
    if (pipe(pfd) < 0)
        printf("error ");
    if (!fork())
    {
        char data;
        printf("I’m child\n");
        printf("press any key to exit……\n");
        scanf("%c",&data);
        write(pfd[1], &data, 1);
        printf("child exiting\n");
    }
    else
    {
        char data;
        read(pfd[0], &data, 1);
        printf("I’m parent\n");
        printf("received %c from child\n", data);
        printf("parent exiting……\n");
        exit(0);
    }
}
```

```
I’m child
press any key to exit……
k
child exiting
I’m parent
received k from child
parent exiting……
[1] + Done                    "/
```

In this program pipe is used to pass a character from child process to the parent process. The read is a blocking call s the parent process waits till it receives a character from child process. Also, in above output we observe that parent finished execution earlier so the child process become orphan and its last print statement occurs at new command line.

Program3:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#define msgsz 16
void main() {
    char *msg1= “hello one”);
    char *msg2= “hello two”);
    char *msg3= “hello three”);
    char inbuf[msgsz];
    int p[2],j;
    pipe(p);
    write(p[1],msg1,msgsz);
    write(p[1],msg2,msgsz);
    write(p[1],msg3,msgsz);
    for(j=0;j<3;j++){
        read(p[0],inbuf,msgsz);
        printf(“%s\n”,inbuf);
    }
    exit(0);
}
```



In the given program, both ends of pipe is used by same process first to write some characters and then to read those written characters. This can be modified as follows to write from the child process and read it from parent.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#define msgsz 16

void main()
{
    char *msg1 = "hello one";
    char *msg2 = "hello two";
    char *msg3 = "hello three";
    char inbuf[msgsz];
    int p[2];
    pipe(p);
    if(!fork())
    {
        printf("\n message sent from child:%s",msg1);
        fflush(stdout);
        write(p[1],msg1,msgsz);
        printf("\nMessage sent from child:%s",msg2);
        fflush(stdout);
        write(p[1],msg2,msgsz);
        printf("\nmessage sent from child:%s",msg3);
```

```
        fflush(stdout);
        write(p[1],msg3,msgsz);
    }
    else{
        char inbuff[msgsz];
        for(int i=0;i<3;i++){
            read(p[0],inbuf,msgsz);
            printf("\nMessage received in parent: %s",inbuf);
            fflush(stdout);
        }
    }
}
```

```
 message sent from child:hello one
Message received in parent: hello one
Message sent from child:hello two
Message received in parent: hello two
message sent from child:hello three
Message received in parent: hello three[1] + Dor
icrosoft-MIEngine-In-r05cle1l.4rw" 1>"/tmp/Micro
```

Message can be sent from parent and then received from child just by changing !fork() to fork() in the if statement in the above program code.

Program4:

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

void main()
{
    int p[2],pid;
    pipe(p);
    pid = fork();
    if (pid == 0)
        printf("in the child p[0] is % d p[1] is % d\n", p[0], p[1]);
    else
        printf("in the parent p[0] is % d p[1] is % d\n", p[0], p[1]);
}
```

```
in the parent p[0] is  3 p[1] is  4
in the child p[0] is  3 p[1] is  4
[1] + Done                       "/usr
tmp/Microsoft-MIEngine-Out-twxzwlno.fs
```

Swapping contents within parent and child process in program 2 we get:

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<sys/wait.h>
void main()
{
    int pfd[2];
    if (pipe(pfd) < 0)
        printf("error ");
    if (!fork())
    {
        char data;
        printf("I'm parent\n");
        printf("press any key to exit……\n");
        scanf("%c",&data);
        write(pfd[1], &data, 1);
        wait(NULL);
        printf("parent exiting\n");
    }
    else
    {
        char data;
        read(pfd[0], &data, 1);
        printf("I'm child\n");
        printf("received %c from parent\n", data);
        printf("child exiting……\n");
        exit(0);
    }
}
```

```
I'm parent
press any key to exit……
i
parent exiting
I'm child
received i from parent
child exiting……
[1] + Done                       "/u
tmp/Microsoft-MIEngine-Out-hxyz2wyv.
```

Here it is observed that the parent process exits first making the child process orphan which leads the shell to think that the program has been completed but then the output from child process appears afterwards. To rectify this problem a wait system call should be inserted in parent process before exiting.

Program5:

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#define msgsz 16

void main() {
    char *msg="hello1";
    char inbuf[msgsz];
    int p[2],pid,j;
    pipe(p);
    pid=fork();
    if(pid>0)
    {
        close(p[0]);
        write(p[1],msg1,msgsz);
    }
    if(pid==0)
    {
        close(p[1]);
        read(p[0],inbuf,msgsz);
        printf("%s\n",inbuf);
    }
    exit(0);
}
```

```
hello1
[1] + Done                    "/usr/
tmp/Microsoft-MIEngine-Out-pswmmzom.r4(
suyog@suyog-VirtualBox:~/Desktop/OS lab
```

After modification

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#define msgsz 16

int main() {
    char *msg = "hello1";
    char inbuf[msgsz];
    int p[2], pid, j;
    pipe(p);
    pid = fork();
    if (pid > 0) {
        if (close(p[0]) == 0) {
            printf("Read file descriptor closed successfully!\n");
        }
        if (read(p[0], inbuf, msgsz) == -1) {
```

```
            printf("Unable to read from pipe!\n");
        } else {
            printf("Successfully read content from pipe!\n");
            printf("In parent, read data : %s\n", inbuf);
        }
        write(p[1], msg, msgsz);
    }
    if (pid == 0) {
        if (close(p[1]) == 0) {
            printf("Write file descriptor closed successfully!\n");
        }
        if (write(p[1], msg, msgsz) == -1) {
            printf("Unable to write to pipe!\n");
        } else {
            printf("Successfully written content to pipe!\n");
        }
        read(p[0], inbuf, msgsz);
        printf("In child, data read : %s\n", inbuf);
    }
    exit(0);
}
```

```
Read file descriptor closed successfully!
Unable to read from pipe!
Write file descriptor closed successfully!
Unable to write to pipe!
In child, data read : hello1
```

Discussion

The pipe system call allocates a message pipe and places file descriptor (which are represented by integer number, one for reading and one for writing) in the length two array parameter passed. This file descriptor remains same on parent and child processes as the child process is forked after the pipe system call. Thus the write and read call reads or writes from same "file" in both parent and child processes as the file descriptor passed in both read/write calls are identical. This also means that if communication is not properly designed same process may write the message and read the same message. Also if a process terminated but other process is still waiting for message to be sent by the terminated process then it may wait indefinitely if the waiting process didn't close its writing file descriptor.

Conclusion

Hence, inter process communication between two independently running child and parent processes can be achieved by using read and write function calls intermediated through use of message pipe via pipe system call.