BITS, Pilani - Hyderabad Campus
Operating Systems (CS F372)

Tutorial - 5

The objective in this tutorial is to learn about some IPC mechanisms in Linux.

Message Queues:

Message queues provide an additional technique for IPC (apart from pipes that you already know). The main advantage of using Message Queues is that they provide "Asynchronous Communication Protocols" i.e. the sender and the receiver do not need to be active at the same time. The messages sent by a process are stored at a location which can be read at a later time by the receiver. Basic Message Passing IPC lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

For sending/receiving messages from a queue you need to first create it. The creation is done by the *msgget()* function which looks as follows :

int msgget(key_t key, int msgflg);

What is a key ? How is it created using *ftok()*?

$ man ftok

Once the queue has been created, you can use *msgsnd()* to send messages into the queue. A message typically consists of two parts. For example the structure below can be used to send and receive messages.

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

It is possible to send only one-byte arrays into a queue. Let us send one using the *msgsnd()* system call. The function looks as follows:

*int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);*

What does each of the parameters of the msgsnd() call stand for ?

Now, let us look into the receiving part. It is done using the *msgrcv()* function which looks as follows:

*int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);*

How is a message queue deleted using the *msgctl()* command ?

The *msgctl()* call has the following syntax:

*int msgctl(int msqid, int cmd, struct msqid_ds *buf);*

Let's create two programs which will communicate amongst themselves through message queues:

producer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("producer.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }
    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }
    printf("Enter lines of text, or press ^D to quit:\n");

    buf.mtype = 1;

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
        int len = strlen(buf.mtext);

        /* ignore newline at end, if it exists */
        if (buf.mtext[len-1] == '\n')
            buf.mtext[len-1] = '\0';
        if (msgsnd(msqid, &buf, len+1, 0) == -1)  /* +1 for '\0' */
            perror("msgsnd");
    }
    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }
    return 0;
}
```

consumer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};
```

```
int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("producer.c", 'B')) == -1) { //need say key as producer
        perror("ftok");
        exit(1);
    }
    if ((msqid = msgget(key, 0644)) == -1) {// connect to the queue
        perror("msgget");
        exit(1);
    }
    printf("Consumer: ready to receive messages!\n");

    while(1) {
        if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("Consumer: \"%s\"\n", buf.mtext);
    }
    return 0;
}
```

2. Shared Memory:

Shared memory is another way by which two or more processes can communicate. The basic idea is that one program will create a shared memory portion where it will put a certain amount of data while the other process will read it. The method used for creating and sending messages by using shared memory is similar to that of message queues.
The creation of the shared memory space is done using *shmget()* which looks as follows:

int shmget(key_t key, size_t size, int shmflg);

Before sharing any message, the process has to attach/connect itself to that shared memory space. That is done via the *shmat()* call which looks as follows:

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

After that, you can just do a *strncpy()* or any other cpy() statement to copy the message into that memory location which is returned by *shmat()*. The other process can then read it. Similar to message queues, each process needs to detach itself from the shared memory once the job is done and then destroy the occupied memory block. This is achieved through *shmdt()* and *shmctl()* calls.

*int shmdt(void *shmaddr);*
void shmctl(shmid, IPC_RMID, NULL);

Complete the following program where a parent and a child communicate through shared memory. Parent writes all letters of the alphabet ‘a to z’ in shared memory, and the child reads it.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid;
```

```
    char *shmPtr;
    int n;

    if (fork() == 0) {
        sleep(5); // To wait for the parent to write
        if( (shmid = shmget(2041, 32, 0)) == -1 )
            exit(1);
        shmPtr = shmat(shmid, 0, 0);
        if (shmPtr == (char *) -1)
            exit(2);
        printf ("\nChild Reading ....\n\n");
        for (n = 0; n < 26; n++)
            putchar(shmPtr[n]);
        putchar('\n');
    } else  {
        if( (shmid = shmget(2041, 32, 0666 | IPC_CREAT)) == -1 )
            exit(1);
        shmPtr = shmat(shmid, 0, 0);
        if (shmPtr == (char *) -1)
            exit(2);
        for (n = 0; n < 26; n++)
            shmPtr[n] = 'a' + n;
        printf ("Parent Writing ....\n\n") ;
        for (n = 0; n < 26; n++)
            putchar(shmPtr[n]);
        putchar('\n');
        wait(NULL);
        if( shmctl(shmid, IPC_RMID, NULL) == -1 ){
            perror("shmctl");
            exit(-1);
        }
    }
    return 0;
}
```

Modify the above program such that there are two children reading the above pattern that the parent writes. The first child reads the first two characters (ab) only, whereas the second child reads all the third character (c) onwards.

Supporting commands

```
man svipc # ipcs ipcrm
```

Homework:

Modify the shared memory based implementation such that

- After forking the child, parent reads data from user and writes to shared memory.
- The child converts the case for characters and prints them. Remember to handle non-alphabets (new line) properly.
- Before starting the child, make sure that the parent has created shared memory and initialized it will all zero’s.