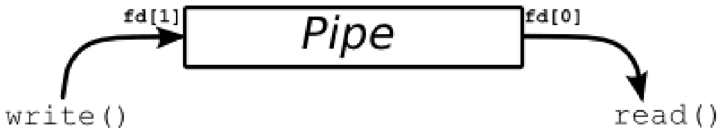
BITS, Pilani - Hyderabad Campus
Operating Systems (CS F372)

Tutorial - 4

The objective in this tutorial is to learn about Unix IPC mechanisms such as pipes, messages queues, and shared memory.

1. Pipes :

What are pipes ? What is the functionality of '|' in calls such as "ls | more" ?



A call to pipe() returns a pair of file descriptors. A basic example to create and test file descriptors is as follows :

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

int main()
{
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }

    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("read \"%s\"\n", buf);

    return 0;
}
```

Now let us see how a parent and its child can communicate with each other via a pipe.

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

int main()
```

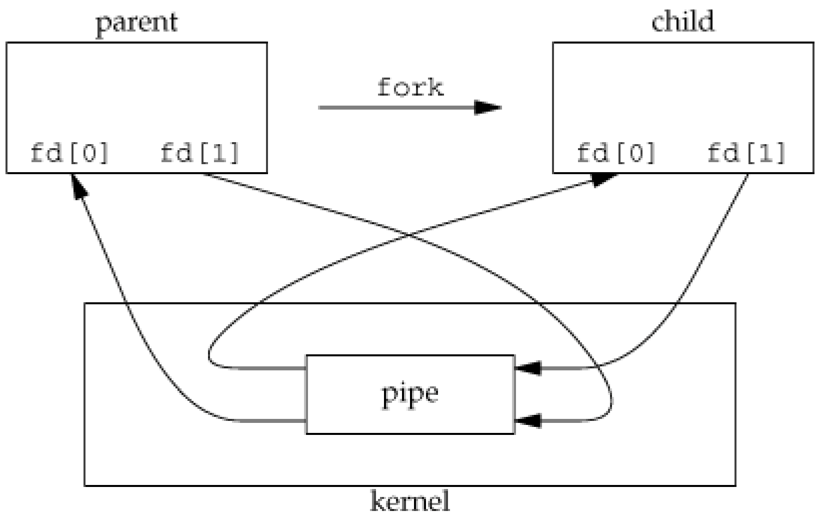
```
{
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
       close(pfds[0]); /*always close the unwanted ends of pipe */
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
       close[pfds[1]);
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

After the call to fork() following pipe(), both the parent and child share the pipe as shown in the figure below. So, we need to close the unwanted ends as in the program above.



Code to implement **"ls | wc -l"**. You need to use **dup()** or **dup2()** system call. Read man page on how to use them.

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

int main(int argc, char *argv[]) {
    pid_t pid;
    int pfd[2];
    int s;
    pipe(pfd);
    pid = fork();
    if (pid == 0) {
        dup2(pfd[1], 1);
        close(pfd[0]);
        close(pfd[1]);
       if (execlp("ls", "ls", NULL) == -1) /* first child runs "ls" */
            perror("execlp ls");
    } else {
        if (fork() == 0) {
            dup2(pfd[0], 0);
            close(pfd[0]);
            close(pfd[1]);
           if (execlp("wc", "wc", "-l", NULL) == -1) /*second child runs wc" */
                perror("execlp wc");
        } else {
            close(pfd[0]);
            close(pfd[1]);
            wait(&s);
            wait(&s); /* need to wait for both the children */
        }
    }
}
```

Homework: Write code that uses the above techniques to create child processes and two pipes to implement the command:

```
ls -l | grep 2016 | wc -l
```

2. Message Queues :

Message queues provide an additional technique for IPC. 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.

What are the other characteristics of Message Queue ?

How are message queues different from Pipes/Named Pipe ?

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 key ? How is it created using *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.

So now that we know how the queue is created and how a message looks like, 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 ?

Receiving from queue: Now that we know how to send a message, let us look into the receiving part. Quite intuitively, it is done using the *msgrcv()* function which looks as follows:

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

Last but one, 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);*

So, now let's create two programs which will communicate amongst themselves through message queues:

writer.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("writer.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }
    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }
    printf("Enter lines of text, ^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;
}
```

reader.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("writer.c", 'B')) == -1) { //same as writer.c
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, 0644)) == -1) {// connect to the queue
        perror("msgget");
        exit(1);
    }

    printf("Reader: ready to receive messages, boss.\n");

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

    return 0;
}
```

3. 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, the sharing of message is trivial. You can just to 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);

Now let’s look at a program where a parent and a child communicates through the shared memory.

```
#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;
}
```

Supporting commands

```
man svipc ; # ipcs ipcmk ipcrm
```

Extended take-home commands

Modify the shared memory based implementation such that

- Rather than sleeping for five seconds, the child detects when the data is available as well as the number of bytes in data.
- After forking the child, parent reads data from user and writes to shared memory.
- The child coverts the case for characters and prints them. Remember to handle non-alphabets (new line) properly.
- E.g. – before starting the child, make sure that the parent has created shared memory and initialized it will all zero’s.