Lab # 07

IPC-II (inter-process communication)

In this lab we will look at the following IPC techniques in detail.

i. **Pipes**
ii. **Message Passing**

1. Pipes

Pipes are another IPC technique for processes to communicate with one another. It can also be used by two threads within the same process to communicate.

There are three files that are open all the time for input and output purposes. These are:
1. The standard input
2. The standard output
3. The standard error

And that they have a file-descriptor of 0, 1, and 2 respectively. Whenever a program needs to display some output (via cout or printf), it will write that output to the standard output file descriptor. This will in return be displayed on the monitor. Whenever a program needs to take some input from the keyboard, it will take its input from the standard input file descriptor. Similarly, whenever an error needs to be displayed, that error will be sent to the standard error file descriptor. These files are linked-up internally to peripheral devices such as keyboard, monitor, etc. However, these linkages can be changed to point to something else. Pipes work by doing exactly that! So a pipe will redirect the standard output of one process to become the standard input of another the rest of communication is done using the following rules:

The pipe will be a buffer region in main memory which will be accessible by only two processes. One process will read from the buffer while the other will write to it. One process cannot read from the buffer unless and until the other has written to it.

1.1 Pipe On the Shell

Run the following command: pstree As you will notice, the output is too long to fit in the screen. Now run the command with:

pstree | less

Using the up and down arrows you will notice that you can browse through the output which was otherwise not visible in the first command which we gave. To exit, press q.

The | is the symbol for pipe and as you would have guessed, pstree and less are two processes. In this usage, the standard output of pstree has become the standard input of the less program. Try the following command:

pstree | grep bash

Again, you will see that the output of above command is much different from just pstree command. What the above command should print is only those lines of text from the pstree output in which the keyword bash appears. Hence, pstree and grep are two separate processes. But the standard output of the pstree command has become the standard input of the grep program. The pstree command writes out its output to the grep process. The grep process receives it, searches for keyword, formats output, and then displays the result.

1.2 Pipe System Call

The following example shows communication between two processes using pipe system call

Example 1:

```
#include <unistd.h>
int main()
{
int pid;                  // for storing fork() return
int pfd[2];               // for pipe file descriptors
char aString[20];         // Temporary storage
pipe(pfd);                // create our pipe
pid = fork();             // create child process
if (pid == 0)            // For child
{
 write(pfd[1], "Hello", 5);              // Write onto pipe
 close(pfd[0]);
 }
else // For parent
{
read(pfd[0], aString, 5);                // Read from pipe
close (pfd[1]);
}
}
```

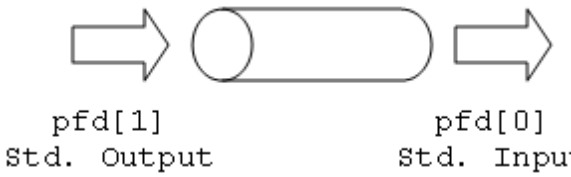

Figure 1: Child Process

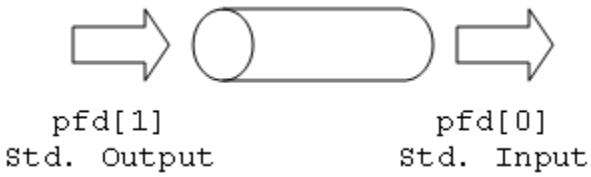

Figure 2: Parent Process

Example 2:

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
int main()
{
int pfd[2];
pipe(pfd);
if (fork() == 0)
{
close(pfd[1]);
dup2(pfd[0], 0);
close(pfd[0]);
execlp("wc", "wc", (char *) 0);
}
else
{
close(pfd[0]);
dup2(pfd[1], 1);
close(pfd[1]);
execlp("ls", "ls", (char *) 0);
}
}
```

2. Message Passing

Communication takes place by exchange of messages

If P & Q wish to communicate, they need to:

- Establish communication link between them
- Communication link can be uni/bi directional, and associated with a single pair of communicating processes
- Exchange messages via send(message), receive(message)

OS Message Queue is a linked list of messages. Queue identified by message queue identifier.

```
struct msg {
long mtype;
char mtext[MSGLENGTH]; };
```

This struct must be included in each process sharing messages.

Type = 0 receives next msg
Type = +ive receives next msg where type matches
Type = -ive receives 1st msg where type < abs(-ive)

Example 3:

Process 1 (Sending Message)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#define MSGSZ 128
typedef struct msgbuf {
         long    mtype;
         char    mtext[MSGSZ];
         } message_buf;

main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
    size_t buf_length;
    key = 1234;
    msqid = msgget(key, msgflg );
    sbuf.mtype = 1;
    strcpy(sbuf.mtext, "Did you get this?");
    buf_length = strlen(sbuf.mtext) + 1 ;
    msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT);
}
```

Process 2 (Receiving Message)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ 128
typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;
main()
{
```

```
    int msqid;
    key_t key;
    message_buf  rbuf;
    key = 1234;
    msqid = msgget(key, 0666);
    msgrcv(msqid, &rbuf, MSGSZ, 1, 0);
    printf("%s\n", rbuf.mtext);
}
```