Operating system
Lab 3
IPC

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

Prog 1:

```
int exflag=0;
void main()
{       char c;
        if(!fork()){
        printf("press a key");
        scanf("%c",&c);
        exflag=1;
        exit(0);
        }
        else{
        while(!exflag);
        printf("i got the character");
        exit(0);
        }
}
```

Will this work?? Try modifying the above code so that the characters can be passed in between the above processes. All "this will work" codes may not work. There are various solutions to this. One of them is pipe.

PIPE
The pipe system call accepts a pointer to an integer (actually an array of two integers). The first integer (pfd[0] can be used to read using the *read* call and the second (pfd[1]) can be used to write using the *write* call.

Prog 2:

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

Prog 3:

```
#define MSGSZ 16
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);
}
```

Comment on the above program. Now modify the above program so that the read and the write are done from the parent and the child;; and the child and the parent respectively. What is the output?

Prog 4:

```
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]);

}
```

You might observe that the values of file descriptors are the same for both the cases. In fact there are four gates connected to the pipe. Each end of the pipe is connected to both the process. A parent may read or write and a child can as well.

Now swap the contents within the parent and the child in prog 2. what will be the output. If there is some betrayal to your expectation then rectify it.

Prog 5:

```
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);
        }


}
```

The close system call has been used to close one of the file descriptors within each process. The effect is that the corresponding process can't use that file descriptor. Modify the program to make sure that it indeed is the case.