BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

IS F462 – Network Programming

Lab#3

Topic: Pipes, FIFO, Message Queue

Note: Please use programs under *code* directory supplied with this sheet. Do not copy from this sheet.

Please practice the given programs in the lab. Questions given can be answered after the lab.

Pipes

//pipe.c

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

#define MSGSIZE 16
main ()
{
  int i;
  char *msg = "How are you?";
  char inbuff[MSGSIZE];
  int p[2];
  pid_t ret;
  pipe (p);
  ret = fork ();
  if (ret > 0)
    {
      i = 0;
      while (i < 10)
        {
          write (p[1], msg, MSGSIZE);
          sleep (2);
          read (p[0], inbuff, MSGSIZE);
          printf ("Parent: %s\n", inbuff);
          i++;
        }
    exit(1);
    }
  else
    {
      i = 0;
      while (i < 10)
        {
```

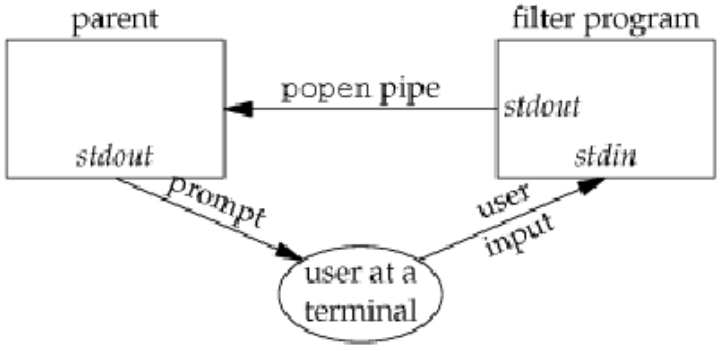
```
          sleep (1);
          read (p[0], inbuff, MSGSIZE);
          printf ("Child: %s\n", inbuff);
          write (p[1], "i am fine", strlen ("i am fine"));
          i++;
        }
    }
  exit (0);
}
```

Q?

a. Check the output of the above program. Observe that using one pipe we can communicate both ways but in only one direction at a time.
b. Remove one of the sleep statements and see the output.
c. Remove both the sleep statements and see the output.
d. Try to make the above program synchronized i.e. only when the child completes its writing, parent writes data; child doesn't write until parent completes writing.

1. Application of Pipe

The program needs to read only the upper case letters even though the user may enter the lowercase letters. For that the following design is considered. The filter program reads the characters from terminal and converts all of them into upper case and writes to output. Filter program is invoked by the program using popen().




```
/*First compile filter program gcc filter.c –o filter*/

/*filter.c*/
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
void
err_sys (char *str)
{
```

```
  perror (str);
  exit (-1);
}

int
main (void)
{
  int c;

  while ((c = getchar ()) != EOF)
    {
      if (islower (c))
       c = toupper (c);
      if (putchar (c) == EOF)
       err_sys ("output error");
      if (c == '\n')
       fflush (stdout);
    }
  exit (0);
}



/*parent.c*/
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

void err_sys(char* str)
{
       perror(str);
              exit(-1);
              }

#define MAXLINE 80
  int
main (void)
{
  char line[MAXLINE];
  FILE *fpin;

  if ((fpin = popen ("./filter", "r")) == NULL)
    err_sys ("popen error");
  for (;;)
    {
      fputs ("prompt> ", stdout);
      fflush (stdout);
      if (fgets (line, MAXLINE, fpin) == NULL)  /* read from pipe */
       break;
      if (fputs (line, stdout) == EOF)
       err_sys ("fputs error to pipe");
    }
  if (pclose (fpin) == -1)
    err_sys ("pclose error");
  putchar ('\n');
  exit (0);
}
```

Q?

1. Observe the usage of popen. Using pipes we can filter the data that is coming from terminal into the program.
2. modify the program so that parent can accept only numbers. User may enter anything. But the program should read only the numbers.

2. Pipelines

Consider the following program for executing `ls –l|wc –l`.

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

main ()
{
  int i;
  int p[2];
  pid_t ret;
  pipe (p);
  ret = fork ();

  if (ret == 0)
    {
      close (1);
      dup (p[1]);
      close (p[0]);
      execlp ("ls", "ls", "-l", (char *) 0);
    }

  if (ret > 0)
    {
      close (0);
      dup (p[0]);
      close (p[1]);
      wait (NULL);
      execlp ("wc", "wc", "-l", (char *) 0);
    }
}
```

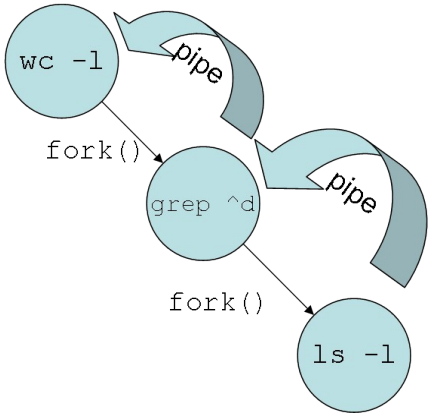
Q?

a. Is wait required here? Why or why not?

b. Is it required to close unused ends of pipe? Suppose if you don’t close p[1] in parent will it have any effect? Find out.

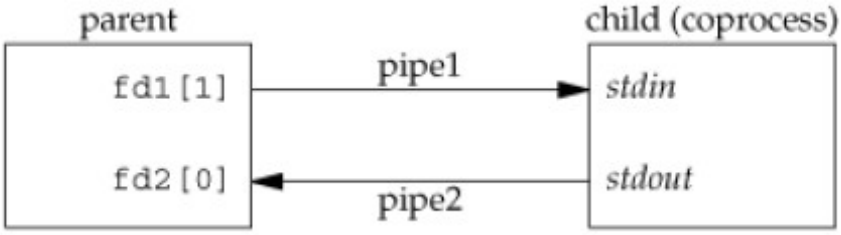
c. dup2() system call can be used instead of dup(). Find out more on dup2() using man dup2 command. Dup2 is considered safer version of dup(). Can you see why is it so?

d. Modify the above program to execute `ls –l| grep ^d |wc –l`. The output of this should be the number of directories in the current directory.



3. Coprocesses:

The diagram shows the use of a child process loading an executable. Here the noticeable point is parent is supplying the input and the same parent is reading the output from the child. This requires two pipes to make the duplex communication possible. In such cases the child process is called as coprocess. The co-process can do variety of tasks like spell checking, validation, sorting, etc ...



In the following example, we use a coprocess to validate emails. The emailValidate.c program is follows.

```
#include <stdio.h>
#include <string.h>
#define MAXSIZE 100

main()
{
char buf[MAXSIZE];int n;
while((n=read(0, buf, MAXSIZE))>0)
{
        buf[n]='\0';
```

```
        if(strstr(buf,"@")>0)
          if(strstr(buf,".")>0)
                write(1,"1\n",2);
           else write(1,"-2\n",3);
        else write(1,"-3\n",3);
}
}
```

It takes the string from stdin and writes the result to stdout. Now consider the following parent process.

```
main ()
{
  int p1[2], p2[2];
  int ret;
  FILE *fpi, *fpo;
  char line[MAXSIZE], result[MAXSIZE];
   pipe (p1);
  pipe (p2);
  ret = fork ();
   if (ret == 0)
    {
      close (p1[1]);
      close (p2[0]);
         dup2 (p1[0], 0);
      dup2 (p2[1], 1);
      execl ("./validateEmail", "validateEmail", (char *) 0);
       }
  else
    {
      close (p1[0]);
      close (p2[1]);
      fpi = fopen ("emails.txt", "r");
      fpo = fopen ("emails_validation.txt", "w");
```

```
while (fgets (line, MAXSIZE, fpi) != NULL)
    {
     write (p1[1], line, strlen (line));
     read (p2[0], result, MAXSIZE);
     fprintf (fpo, "%s,%d\n", line, atoi (result));
    }}}
```

The parent process takes help from the co-process validateEmail to check the emails.

Q?

1. Execute the above program. First compile `validateEmailCoprocess.c` to validateEmail executable. `gcc validateEmailCoprocess.c –o validateEmail` Then compile coprocess_parent.c and run it. The emails.txt sample input is available here. Double click it to open. Emails.txt. Now parent is running, if we kill validateEmail child process by sending SIGKILL signal, find out what would happen? Modify the parent to recognize such unexpected situations and respond positively.
2. Suppose the input is in order of millions. We want to concurrently process the validation. Modify the above program so that the there can be several coprocesses concurrently processing.
3. Consider the following alteration in the validateEmail.c file. Compile it to validateEmail executable and run the parent.

```
main ()

{

  char buf[MAXSIZE];

  while (fgets (buf, MAXSIZE, stdin) != NULL)

    {

      if (strstr (buf, "@") > 0)

if (strstr (buf, ".") > 0)

  printf ("1\n");

else

  printf ("-2\n");

      else

printf ("-3\n");
```

```
        }
    }
```

What did you notice? Why the program doesn't work? Find out. [hint: Effect of using low-level system calls like read(), write() with standard i/o functions like printf(), scanf() etc]

FIFO

//fifo.c

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

#define MSGSIZE 16
main ()
{
  int i;
  char *msg = "How are you?";
  char inbuff[MSGSIZE];
int rfd,wfd;

if(mkfifo("fifo",O_CREAT|O_EXCL|0666)<0)
        if(errno=EEXIST)
                perror("fifo");

   pid_t ret;
  ret = fork ();
   if (ret > 0)
```

```
  {
   rfd=open("fifo", O_RDONLY);
   wfd=open("fifo",O_WRONLY);
        i = 0;
    while (i < 10)
        {
          write (wfd, msg, MSGSIZE);
          sleep (2);
          read (rfd, inbuff, MSGSIZE);
          printf ("child: %s\n", inbuff);
          i++;
        }
  exit(1);
  }
 else
  {
   rfd=open("fifo", O_RDONLY);
   wfd=open("fifo",O_WRONLY);
    i = 0;
    while (i < 10)
        {
          sleep (1);
          read (rfd, inbuff, MSGSIZE);
          printf ("parent: %s\n", inbuff);
          write (wfd, "i am fine", strlen ("i am fine"));
          i++;
        }
  }
 exit (0);
}
```

Q?

1. Run the above program. Did you get any output? Find out why the

program is blocking. Correct it and run again

2. You can see that using FIFO, we can communicate in both directions but only one direction at a time. Modify the above program such that it will generate a SIGPIPE signal. See the following table for conditions in which it can occur.
3. The configuration values of PIPE_BUF can be obtained using the following call

   ```
   long fpathconf(int filedes, int name);
   ```

   ```
   filedes refers to pipe's descriptor and name refers to _PC_PIPE_BUF
   ```

   or use the following command

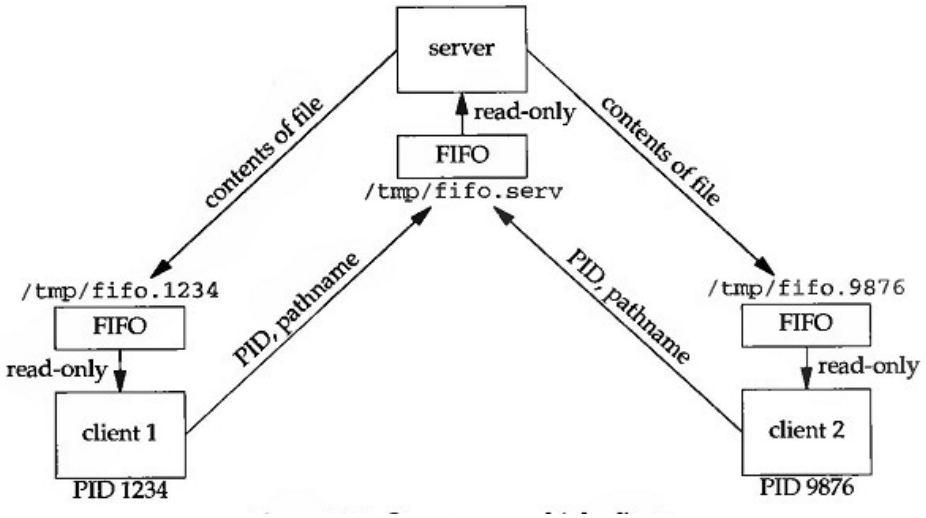
   ```
   getconf PIPE_BUF /
   ```

   After finding the PIPE_BUF value, write data having size more than the PIPE_BUF value more than once into the pipe/fifo. And see the effect.
4. O_NONBLCOK flag is set so that the program will not be put to sleep waiting for data. In FIFO it is done while using open call to open FIFO.

   ```
   writefd = Open(FIFO1, O_WRONLY | O_NONBLOCK, 0);
   ```

Chat server usng FIFOs or Named Pipes:

Consider the following diagram.



The above design enables exchanging messages among multiple processes or users in the system. Any user can write the message which includes the destination pid, source pid, and the message. The server receives the message and puts it into the fifo of the destination process. While sending the reply, the same thing happens. There are two files chatfifo_client.c and chatfifo_server.c

Q?

1. Execute the file chatfifo_server.c in one terminal. Open two other terminals and run the chatfifo_client.c in each of them. Try to chat using these two terminals. Do you find some bursts of inputs and outputs? Why does that happen? [Hint: for opening fifo, we need synchronization, otherwise it get blocke]

2. Does this program allow for deadlocks?

3. Can we use O_NONBLOCK flag to make it proper? If yes try it out by modifying the above program.

4. Think of signals can help here, if we want to let processes know just-in-time when the message is put on fifo? Modify program in 1 to mke it happen.

Message Queues

1. **Creating and using message queues:**

Consider the following program msgget.c which creates a message queue.

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

main(){
int queue_id = msgget(IPC_PRIVATE, 0600);
if (queue_id == -1) {
   perror("msgget");
   exit(1);
}
}
```

Q?

1. Compile and run the above program. List the message queues created in your system using the command *ipcs*. Do you notice the message queue that you have just created?

2. Run the above program several times and then run ipcs command. Do you find there are many message queues created now? Check the value of key. How did it create so many mqs? Is it the problem with not using IPC_CREAT and IPC_EXCL?
3. Consider the program below.

```
main(){

int queue_id = msgget(5678, IPC_CREAT|IPC_EXCL|0600);

if (queue_id == -1) {

    perror("msgget");

    exit(1);

}

}
```

   Execute the above program. Now check with ipcs. Do you identify the message queue created just now? Check the key value. Of course it is in octagonal format. Run the above program several times and see the output. What makes it give the error message?

4. use ftok() function to generate a unique key for your message queue.
5. delete the message queue using the msgctl() with IPC_RMID command or using ipcrm on command-line.

2. **Chat server using message queues (Use of multiplexing):**

Consider the example where a single server receives messages from many clients and prints them to console. There are three files namely key.h, msgq_client.c and msgq_server.c.

key.h: edit the path mentioned according to the prithvi directory you are working in
#define MSGQ_PATH "/home/students/…./f2007363/msgq_server.c "

msgq_client.c: client which sends messages

msgq_server.c: server which receives them and prints them on console.


msg_client.c


msg_server.c

Q?

1. **Multiplexing messages:** Compile server and client. Run one instance of server in one putty terminal. And run many instances of clients from other putty terminals. After sending some messages from client, if you kill server, does it affect clients? Keep sending messages from clients. And if you run server again after some time, what does server show? What do you understand from this?

2. here the server is simply displaying the messages it receives. It is not actually doing the job of chatting application. Just like in fifos. Modify the server to route the messages to suitable clients. And modify the clients to read from the queue and disply.

3. since we are using single queue, there must be a deadlock situation. Find out the maximum bytes the message queue can take up using the sys call msgctl(). Then simulate a situation where deadlock can occur.

3. **Configuring a Message Queue**

When the message queue is created, it is created with default settings. In the following example we look at how to retrieve these settings and modify them.

```
/*msgqstats.c*/int main ()

{
  int msgid, ret;
  key_t key;
  struct msqid_ds buf;
  key = ftok ("msgqstats.c", 'A');
  msgid = msgget (key, IPC_CREAT|0620);
  /* Check successful completion of msgget */
  if (msgid >= 0)
    {
      ret = msgctl (msgid, IPC_STAT, &buf);
      if (ret == 0)
```

```
        {
         printf ("Number of messages queued: %ld\n", buf.msg_qnum);
         printf ("Number of bytes on queue : %ld\n", buf.msg_cbytes);
         printf ("Limit of bytes on queue  : %ld\n", buf.msg_qbytes);
         printf ("Last message writer (pid): %d\n", buf.msg_lspid);
         printf ("Last message reader (pid): %d\n", buf.msg_lrpid);
         printf ("Last change time         : %s", ctime (&buf.msg_ctime));
         if (buf.msg_stime)
           {
            printf ("Last msgsnd time         : %s",  ctime (&buf.msg_stime));
           }
         if (buf.msg_rtime)
           {
            printf ("Last msgrcv time         : %s",  ctime (&buf.msg_rtime));
           }
        }
}
 return 0;
}
```

Q?

1. Compile and run the above program. Find out the default size of a message queue. The same information can be obtained by running the command `$ipcs –q –i <msgqid>`
2. Write a line in the above program that would print the permissions set for the message queue? Check the slides for accessing that particular member.
3. modify the above program such that it will create a message queue for a given key.
4. Suppose we need to increase the maximum size limit of the queue. Compile and run the following program. The complete program is in msgqconf.c

```
int

main ()

{
```

```
  int msgid, ret;
  key_t key;
  struct msqid_ds buf;
  key = ftok ("msgqstats.c", 'A');
  msgid = msgget (key, IPC_CREAT|0620);
  /* Check successful completion of msgget */
  if (msgid >= 0)
    {
      ret = msgctl (msgid, IPC_STAT, &buf);
      buf.msg_qbytes = 4096;
      ret = msgctl (msgid, IPC_SET, &buf);
      if (ret == 0)
       {
         printf ("Size successfully changed for queuec %d.\n", msgid);
       }
       else
              perror("msgctl:");
    }
       else
              perror("msgget:");
  return 0;
}
```

Check the output of this program. Cross check with `ipcs –q –i <msgqid>`. Find out what is the maximum size a message queue can be allotted?

5. Similarly modify the program to modify the userid, permissions etc.

/* End of Lab 3 */