BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)
IS F462 – Network Programming
Lab 5

In today's lab we will do some example programs on fifos and System V message queues and semaphores.

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

Consider the above program done using 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

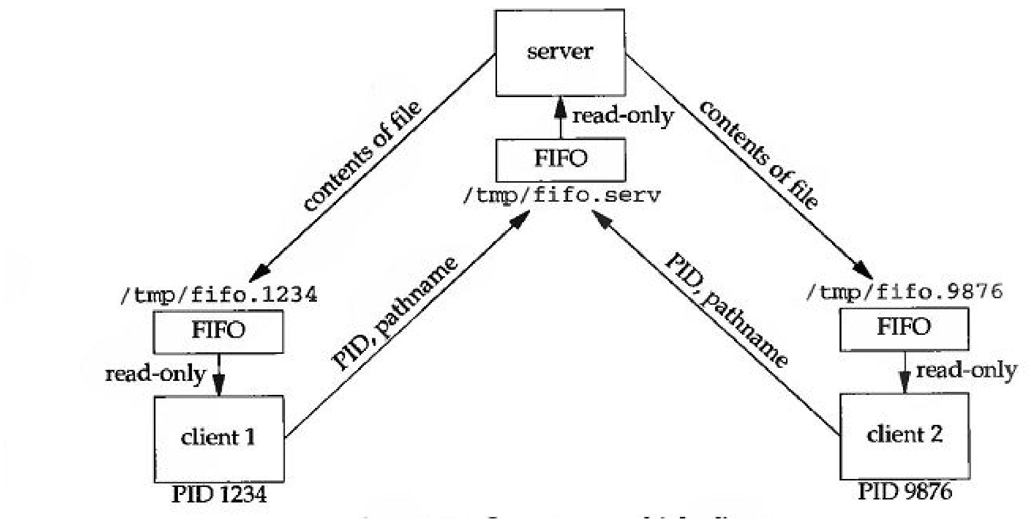
   `getconf PIPE_BUF /`

   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.

<u>Message Queues</u>

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.

Semaphores:

4. Creating and initializing a semaphore

Semaphores are used to synchronize the access to a shared data among multiple processes. The following program shows us on how to create and initialize a semaphore. The complete program is in semget.c

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

```
#define key (2000) //change this to your id, only digits: 2009100

main ()
{
  int id;
  id = semget (key, 1, IPC_CREAT | 0666);
  if (id < 0)
    printf ("Semaphore is not created\n");
  printf("Semaphore successfully created with id %d\n", id);
return;
}
```

Q?

1. Compile and run the above program. The list of semaphores that can be viewed by your account can be listed by running the command `$ipcs -s`.
2. Run the command $ipcs –s –i <semid> and check what is the value of semaphore.
3. the default value of the semaphore is not useful for practical purposes. It has to be set to the available number of resources. Suppose if we have 6 resources with us, how do we initialize? Use the following line to initialize the semaphore. Modify the above program with the following line. Complete program is available in seminit.c

   ```
   semctl (semid, 0, SETVAL, 6)
   ```

   after running the program, check the modified value using the above command.
4. find out what is the maximum value a semaphore can take.
5. how do we initialize for binary semaphores and counting semaphores?
6. As discussed in the class, the fact that creating and initializing operations not being done atomically, it may lead to deadlock situations. Write a piece of code that ensures in any process that is accessing the semaphore only after initialization.
7. use semctl() call to remove the semaphore in your program.

5. operating with semaphores:

Considering that we have created the semaphore and initialized it, let us see how to utilize it. the following program synchronizes the parent and child in printing the lines. View the file semopPC.c. This is an example of binary semaphores.

Q?

1. Compile and run the above program. What did you observe? Is the child and parent printing lines one after another? Understand how it happens.

2. Change the initial value of the semaphore 5 and check the out put. Justify the output. Remove the sleep() and check.
3. Create one more child running the same code as the first child and check the output. Justify.

6. Producer-consumer problem using semaphores:

Consider a producer-consumer problem. In this problem, producers supplies goods only when availability of goods is zero and at one time he supplies 10 goods. Consumer will consume goods only when goods are available. The Semaphores are used to synchronize the producer and consumer demands. The programs are available as producer.c and consumer.c. This is an example of counting semaphores. Change the key value to something of your idno. Otherwise you may get weird results.

Q?

1. Run both the programs in separate terminals. Observe how the semaphore is used to control the access to objects. if you stop producer.c program from running, how does consumer behave? Also use ipcs to see the semaphores created in the system.
2. Run consumer program in more than one terminal and see the sem value. This is the case when we have multiple consumers with single producer.
3. Change the producer behaviour. Producer will not wait until it becomes zero rather he will keep adding 1 object at a time. Suppose that the storage capacity for storing the goods is only N. that means producer can't produce and store more than N. How do you ensure this using semaphores? [Hint: think of using one more semaphore]
4. Try to run program in 3 in multiple terminals. That is multiple producers and multiple consumers. Justify the output.
5. Suppose that the items are not use and throw. They are re-usable. That is when a consumer takes an item he returns it after sometime say 2 secs. Modify the program to work for this.
6. Suppose while running as in 5, kill some consumers using kill command and observe the sem values. Did the consumers return the acquired values? Use SEM_UNDO flag and check.

End of lab 4