Inter-process Communication
The Need for Various Types of IPC Objects

In this article, we explore the basic inter-process communication (IPC) techniques that are available.

A couple of questions: What is IPC? And do normal users require it often?

IPC occurs when processes exchange data through some communication channel. For a general-purpose OS, processes do not access the memory of other processes due to protection mechanisms. So IPC is essential for when a process needs data from another process.

Implicitly, we may use IPC every day. Browsing the WWW is an example. Here, the processes involved are the local browser and the remote Web server. The communication channel is the Internet, and the protocol is HTTP. Another example is if we run the 'command pipeline' *who | wc –l* to print the number of users logged in on the system. This also is an IPC technique. Thus, we often use IPC techniques when we work.

Broadly, we can classify common or widely available IPC techniques in three groups: pipes and FIFO; message queues, shared memory and semaphores; and sockets. The first group has *process-persistent* IPC techniques—these IPC objects are available while the processes that use them are running. The second group of IPC objects is *kernel-persistent;* they are available till the system reboots, or they are explicitly deleted. The third group of IPC objects can help us communicate over a network. Groups 1 and 2 are generally used to communicate between processes on a single computer.

In the subsequent sections, we will look at algorithms and C code to use some of these IPC objects to implement a generic sender and receiver. I have provided the algorithms as pseudo code, which you can use for reference, to improve the code. I mainly focus on the core concepts and avoid explaining too many syntactic details, which you can learn from the Linux manual pages.

Note: Source code is omitted for brevity; please download it from *http://www.linuxforu.com/article_source_code/sept12/inter_process_communication.zip.*

Pipes and FIFO

In the previous section's example, the *who|wc –l* command used a communication channel called a *pipe*. A pipe has two ends (represented by descriptors)—*read* and the *write* end.

A process can read data from the *read* end and the data can be written on the *write* end. In our example, the *who* command writes to the pipe, and the command *wc –l* reads from the pipe. This can be implemented by a C program, but I leave this for you to try, with some hints at the end of this section.

Here's the algorithm for a sender and receiver using a FIFO object.

Sender

```
Create a FIFO. (If already created then proceed).

Open the FIFO for Reading/Writing.

While (1) {

      Read from Key board.

      Write data read from user to FIFO.

      If User enters "exit"
 then break the loop.

}

Delete FIFO and exit program.
```

Receiver

```
Create a FIFO. (If already created then proceed)
Open the FIFO for Reading/Writing
While (1) {
      Read data from FIFO
      Print the read data on the screen
      If received data is "exit" then break the loop.
}

Program terminates.
```

The C programs for the two blocks of pseudo-code above are in the downloaded archive as *fifo_sender.c* and *fifo_receiver.c*. Compile each, preferably naming the output as *sender* and *receiver*, run them in two different terminal windows and test them out.

Please note the function to create a FIFO is *mkfifo*. The name for the FIFO is accepted as a command-line argument to the program; the *S_IRWXU* flag is for specifying access permissions to the FIFO. You can check the man page (run *man 2 open* in a terminal) for the *open* system call, which will show you the detailed list of symbolic constants available for file access permissions. If you run an *ls*, you should see the created FIFO in the list.

Now, open the FIFO to get the file descriptor, which can be used for subsequent reading or writing to or from it. If you try to open a FIFO for reading, but no process has opened it for writing (or vice-versa), the program blocks in the call to open the FIFO. To avoid this, I have used the *O_RDWR* flag to specify opening in *read* or *write* mode. Other functions like *read/write*, which are used for files and even sockets, are known to you from our discussion of file systems. When the user enters *exit*, the sender breaks the loop, deletes the FIFO and exits.

Differences between FIFO and pipes

At the beginning of the section, I mentioned pipes, but we've just looked at an example of a FIFO. Is there any difference? Yes! Here, the communicating processes are not *'related'*, but when using a pipe, you can only communicate between *related processes*—those with common ancestors or the result of a *fork* call. This is because a FIFO has a name (and so is sometimes called a *named pipe*) whereas a pipe does not have any name. You can try rewriting this code using a pipe. A hint: please read the man pages for the *pipe* and *dup* functions. You could also try writing the *who* | *wc –l*, or could try to enhance the simple shell, from the process part of this article, to support pipes.

System V and POSIX IPC objects

The IPC objects in Group 2 are called System V IPC objects, because they originate from System V UNIX. Most systems nowadays have POSIX implementations of these—and the functions used to create and use them in each implementation (System V and POSIX) differ. We will extend the earlier simple FIFO IPC example to use a System V message queue, and also to use System V shared memory along with a POSIX semaphore.

The question now is how to *uniquely identify* an IPC object that several processes use to communicate between each other? These processes may be unrelated and may run at different times. Here, System V takes a *key*-based approach (that is a number), while POSIX takes a *name*-based approach. This means that all processes that need to communicate through a System V IPC object must know a number that's used to index the particular IPC object. On the other hand, processes communicating via POSIX IPC objects need to know the object's name, which is like a filename (though the IPC object is not an ordinary file) to uniquely identify an IPC object they can use for communication.

The POSIX interface also allows for memory-based IPC objects (stored in memory). These name-based IPC objects can be used by *threads*, since threads share a common address space. They can also be used by processes, but then the memory in which they are stored needs to be shared between processes, like say, *Shared Memory*. We will not go into the details of POSIX memory-based implementations at this stage.

Example: Sys V IPC object: message queue

Now, let's implement the sender-receiver code using a System V message queue. Here's the algorithm.

Sender

```
Generate key (say K) using ftok library function
Create/Get Access to the Message Queue (The key K is needed
here)
While (1) {
      Read from keyboard.
      Create a message.
      Send the message to the message queue
      If user has entered "exit" break the loop.
}
Delete message queue and exit program.
```

Receiver

```
Generate key (say K1)
Create/Get Access to the Message Queue (The key K1 is needed
here)
While (1) {
      Read data from the Message Queue.
      Print data on the screen
      If received message is "exit" then break loop.
}
Program terminates.
```

The C programs (*msg_sender.c* and *msg_receiver.c*) can be downloded from the e-download archive. Compile them as before and run in two terminal windows. You can use the *ipcs* command to list System V IPC objects on the system and check that the message queue has been created.

Now, let's understand the code. We used the *ftok* function to generate a key (a number), which should be unique and available to all programs that want to refer to a particular message queue. The choice of arguments (the filename '*msg_receiver.c*' and '*B*') is completely arbitrary, but you should use the same parameter in all programs using the queue. Also, the first argument should be an existing file. (You might use a file name like '*test*' before running the code, and then create the file by running the command *touch test* in the directory where you're executing the program.)

Creating or accessing an existing message queue is done by the *msgget* function, which takes the key value and some additional information. In this example, use the permission bits with the predefined constant *IPC_CREAT*, joining them with the *or* operator. The *IPC_CREAT* constant creates the queue if it does not exist. The return value is the *message queue descriptor*, which then can be used as an argument to functions operating on the queue (reading from it, writing to it or deleting it).

Note the formation of a message. It is a C language structure:

```
struct msbuf {
        long mtype;
        char mstore [80];
}
```

The first element must be of the *long* type, and it can have any type of subsequent elements. In my example, the second element is an 80-character array.

An important feature of Sys V message queues is that the receiver can selectively retrieve messages based on the first element of the message structure. This feature is not available in pipes or FIFO, where readers can only read data in first-in-first-out order.

The functions *msgsnd*, *msgrcv* and *msgctl* send and receive messages, and delete the queue respectively. I have explained the *msgsnd* arguments in code comments. They are similar to those of the *msgrcv* function, which waits for the arrival of a message in the queue; whenever it receives a message, it returns from its blocked state. Note that the original message is removed from the queue when the *msgrcv* function returns.

If multiple messages are available in the queue, which will be retrieved? Based on the fourth argument to *msgrcv*, the type fields of messages in the queue are compared, to decide which message to retrieve. In our example, the value is 1, so the first message with a type value of 1 will be retrieved. There can be other retrieval schemes based on the fourth argument, for which you can consult the man page.

Finally, we have used the *msgctl* function with the constant *IPC_RMID* to delete the message queue.

(*Please note:* Just like the *ipcs* command displays IPC objects, the *ipcrm* command can be used to delete System V IPC objects from the shell prompt.)

The other functions are System V message-queue-related functions (browse the manual pages to learn more about them).

Shared memory and semaphores

Processes normally do not share memory. The memory of a process image (like stack, global data and heap) belongs to that process, and cannot be shared, as we saw when discussing *fork* in the processes article. But you can create memory that can be shared between multiple processes.

This is a faster mechanism of communication, since after creating and attaching shared memory to a process, you do not need any special function calls to access it. It simply behaves like normal data storage (a variable or array). But if multiple processes simultaneously read or write shared memory, race conditions can occur. Avoid that by using a synchronisation object like a semaphore. (Please refer to any standard OS textbook or the Internet for more details on race conditions and semaphores, if needed.)

Let us now implement the program using shared memory (using System V functions) and a semaphore (using POSIX functions). Algorithms follow:

Sender

```
Generate key (say K1).
Create/Get Access to the Shared Memory (The key K1 is needed
here).
Attach the Shared Memory to the process address space.
Create a Semaphore (named put)
Create a Semaphore (named get)
Initialise get to 0.
Initialise put to 1.
While (1) {
      Read from Key board.
      Wait for semaphore put   to become 1
      Write User Input to Shared memory.
      Update semaphore get to 1
      If user input was “exit”then break;
}
Delete shared memory.
Delete semaphores get and put.
Program terminates.
```

Receiver

```
Generate key (say K1).
Create/Get Access to the Shared Memory (The key K1 is needed
here).
Attach the Shared Memory to the process address space.
Get Access to the Semaphore get.
Get Access to the Semaphore put.
While (1) {
         Wait for semaphore get to become 1
         Read data from Shared Memory
         Update semaphore put to 1
         If read data is “exit” then break the loop.
}
Program terminates.
```

The C code implementations are *shm_sender.c* and *shm_receiver.c*. Compile (please use the linker flag *–lrt* here) and run as before (run the sender first, since the semaphores are created in the sender).

Now let's look at the code. The same key is generated by both programs, so they can refer to the same shared memory segment (just like in the message queue example). Before you can use the shared memory, it needs to be attached to the process address space using the *shmat* function, which returns a pointer that can be used to read or write the shared memory.

To avoid race conditions on simultaneous read/writes to the shared memory, two semaphores are used. Here, the sender creates both the semaphores and initialises them. The function for creating or accessing the POSIX named semaphore is *sem_open*. The semaphore is identified by a name; I have used *get* and *put*. You may use any other name, but use the / character, so that all processes refer to the same semaphore. Here, if the reader runs first, it will be able to immediately write to the shared memory, since the initial value of the *put* semaphore is 1. After writing data, it updates the *get* semaphore to 1, so that the receiver, which was waiting on the *get* semaphore, can now read data. Similarly, now the sender waits, till the receiver successfully reads the shared memory. The semaphores ensure that no race conditions occur. After the user enters ‘exit’, the semaphores and the shared memory are deleted using the *sem_unlink* and *semctl* function by the sender.

Observations

Message queues offer more flexibility in terms of message storage and retrieval. Shared memories are fast; after creating and attaching *shm*, you don't need to call special functions to read or write data. However, you need to be cautious about possible race conditions, and use semaphores to prevent them.

I hope you enjoyed reading this introductory article and start using what I’ve featured in applications. END

References

[1] Bach, Maurice J: ‘The Design of the Unix Operating System’, PHI

[2] Stevens, W Richard: ‘Unix Network Programming: Interprocess Communication’, Vol-2, 2nd Edition, Pearson Education

[3] Stevens, W Richard, Rago, Stephen A: ‘Advanced Programming in the Unix Environment’, 2nd Edition, Pearson Education

[4] Linux Manual Pages

Acknowledgements

I would like to thank Tanmoy Bandyopadhyay and Vikas Nagpal for their help in reviewing this article.

By: Swati Mukhopadhyay

The author has more than 12 years of experience in academics and corporate training. Her interests are digital logic, computer architecture, computer networking, data structures, Linux and programming languages like C and C++. For any queries or feedback, she can be reached at *swati.mukerjee@gmail.com* or *http://swatimukhopadhyay.wordpress.com/*.