5 Inter-process communication

Is simplicity best
Or simply the easiest?

Martin L. Gore

There are many applications in which processes need to cooperate with each other. This is always the case, for example, if processes have to share a resource (such as a printer). It is important to make sure that no more than one process is accessing the resource – that is, sending data to the printer – at any given time. This situation is known as a *race condition* and communication between processes must prevent it. However, eliminating race conditions is only one possible use of inter-process communication, which we take in this book to mean simply the exchange of information between processes on one or more computers.

There are many different types of inter-process communication. They differ in a number of ways, including their efficiency. The transfer of a small natural number between two processes could be effected, for example, by one of these generating a matching number of child processes and the other counting them.

This example, which is not meant entirely seriously, is of course very unwieldy and slow and would not be considered. *Shared memory* can provide a faster and more efficient answer to the problem.

A variety of forms of inter-process communication can be used under LINUX. These support resource sharing, synchronization, connectionless and

connection-oriented data exchange or combinations of these. Synchronization mechanisms are used to eliminate the race conditions mentioned above.

Connectionless and connection-oriented data exchange differ from the first two variants by different semantic models. In these models, a process sends messages to a process or a specific group of processes.

In connection-oriented data exchange, the two parties to the communication must set up a connection before communication can start. In connectionless data exchange a process simply sends data packets, which may be given a destination address or a message type, and leaves it to the infrastructure to deliver them. The reader will already be familiar with these models from everyday life: when we make a telephone call we are using a connection-oriented data exchange model, and when we send a letter we rely on a connectionless model.

It is possible to implement one concept (for example, semaphores) based on another (for example, connectionless data exchange). LINUX implements all the forms of inter-process communication possible between processes in the same system by using shared resources, kernel data structures and the 'wait queue' synchronization mechanism. Although semaphores are available in the kernel for synchronization, these themselves rely on wait queues.

LINUX processes can share memory by means of the System V shared memory facility. The file system has been implemented from the start to allow files and devices to be used by several processes at the same time. To avoid race conditions when files are accessed, various file locking mechanisms can be used. System V semaphores can be used as a synchronization mechanism between processes in a computer.

Signals are the simplest variant of connectionless data exchange. They can be understood as very short messages sent to a specific process or process group (*see* Chapter 3). In this category, LINUX still provides message queues and the datagram sockets in the INET address family. The datagram sockets are based on the UDP section of the TCP/IP code and can be used so as to be transparent to the network (*see* Chapter 8).

The available methods for connection-oriented data exchange are *pipes*, *named pipes* (also known in the literature as FIFOs),[1] UNIX *domain sockets* and *stream sockets* in the INET address family. Stream sockets are the interface to the TCP part of the network and are used to implement services such as FTP and TELNET, among others. These are also examined in Chapter 8. The use of the socket program interface does not always amount to inter-process communication, as the opposite number on the network need not be a process. It could for example be a program in an operating system, with no process concept.

[1] FIFO stands for 'First In, First Out', which describes the action of a pipe very well.

Table 5.1 Types of inter-process communication supported by LINUX.

	Kernel	Processes	Network
Resource sharing	Data structures, buffers	System V shared memory, files, anonymous `mmap`	
Synchronization method	Wait queues, semaphores	System V semaphores, file locking, lock files	
Connectionless data exchange	Signals	Signals, System V message queues, UNIX domain sockets in datagram mode	Datagram sockets (UDP)
Connection-oriented data exchange		Pipes, named pipes, UNIX domain sockets in stream mode	Stream sockets (TCP)

5.2 Communication via files

Communication via files is in fact the oldest way of exchanging data between programs. Program A writes data to a file and program B reads the data out again. In a system in which only one program can be run at any given time, this does not present any problems.

In a multi-tasking system, however, both programs could be run as processes at least quasi-parallel to each other. Race conditions then usually produce inconsistencies in the file data, which result from one program reading a data area before the other has completed modifying it, or both processes modifying the same area of memory at the same time.

The situation therefore calls for locking mechanisms. The simplest method, of course, would be to lock the whole file. For this LINUX, like other UNIX derivatives, offers a range of facilities. More common and more efficient, however, is the practice of locking file areas. This locking of file access can be either *mandatory* or *advisory*. Advisory locking allows reading and writing to the file to continue after the lock has been set.

However, lockings mutually exclude each other, depending on the semantics determined by their respective types. Mandatory locking blocks read and write operations throughout the entire area. With advisory locking, all processes accessing the file for read or write operations have to set the appropriate lock and release it again. If a process does not keep to this rule, inconsistencies are possible. However, mandatory locking provides no better protection against malfunctions within processes: if processes have write authorization to a file, they can produce inconsistencies by writing to unlocked areas. The problems produced by faulty programs when mandatory locking is employed are extremely critical, because the locked files cannot be modified as long as the process in question is still running. Since version 2.0, LINUX supports mandatory locking, but the corresponding kernel configuration parameter is by default disabled. For the reasons given above and as POSIX 1003.1 does not require mandatory locking, this is perfectly acceptable.

5.3 Pipes

Pipes are the classical method of inter-process communication under UNIX. Users of UNIX should not be unfamiliar with a command line such as

```
% ls -l | more
```

Here, the shell runs the processes `ls` and `more`, which are linked via a pipe. The first process writes data to the pipe, and the second process then reads it.

Another variant of pipes consists of *named pipes*, also known as FIFOs (pipes also operate on the 'First In, First Out' principle). In the following pages, the terms 'named pipe' and 'FIFO' will be used interchangeably. Unlike pipes, FIFOs are not temporary objects existing only as long as one file descriptor is still open for them. They can be set up in a file system using the command

```
mkfifo pathname
```

or

```
mknod pathname p
```

```
% mkfifo fifo
% ls -l fifo
prw-r--r--   1 kunitz   users          0 Feb 27 22:47 fifo|
```

Linking the standard inputs and outputs of two processes is a little more complicated with FIFOs.

```
% ls -l >fifo & more <fifo
```

There are obviously many similarities between pipes and FIFOs, and these are exploited by the LINUX implementation. The inodes have the same specific components for pipes and FIFOs.


```
struct pipe_inode_info {
   struct wait_queue * wait; /* a wait queue */
   char * base;              /* address of FIFO buffer */
   unsigned int start;       /* offset for current area */
   unsigned int len;         /* length of current area */
   unsigned int lock;        /* lock */
   unsigned int rd_openers;  /* number of processes
                              * currently opening the
                              * pipe/FIFO for read access */
   unsigned int wr_openers;  /* ditto for write access */
   unsigned int readers;     /* number of processes
                              * reading at this moment */
   unsigned int writers;     /* ditto, writing */
};
```

The system call *pipe* creates a pipe, which involves setting up a temporary inode and allocating a page of memory to `base`. The call returns one file descriptor for reading and one for writing: this is achieved by the use of separate file operation vectors.

For FIFOs there is an `open` function which allocates the page in memory and returns a file descriptor that has been assigned an operation vector with read and write operations. Its behaviour is summarized in Table 5.3.

FIFOs and pipes use the same read and write operations, with the memory assigned to the pipe/FIFO interpreted as a circular buffer to which `len` bytes have been written, starting at `start`, without yet having been read back. These operations always take into account whether the `O_NONBLOCK` for the descriptor has been set or not: if it is set, the read and write operations must not block. Unless the number of bytes to be written exceeds the internal buffer size for the pipe (4096 bytes as default), the write operation must be carried out atomically – that is, if a number of processes are writing to the pipe/FIFO, byte sequences for the individual write operations are not interrupted. The semantics implemented in LINUX are shown in Tables 5.4 and 5.5.

5.5 System V IPC

As long ago as 1970, the classical forms of inter-process communication – semaphores, message queues and shared memory – were implemented in a special variant of UNIX. These were later integrated into System V and are now known as System V IPC. LINUX supports these variants, although they are not included in POSIX. At present, shared memory is the only way of allowing more than one process to access the same area of memory under LINUX. The original LINUX implementation was produced by Krishna Balasubramanian, but it has been modified by Eric Schenk, Bruno Haible and Bjorn Ekwall.

5.5.1 Access permissions, numbers and keys

In System V IPC, objects are created in the kernel. These must be assigned unique identifiers to ensure that operations activated by the user process are carried out on the right objects. The simplest form of identifier is a number: these numbers are dynamically generated and returned to the process generating the object. A process entirely separate from the creator process cannot access the object, as it does not know the number. In a case of this sort, the two processes will have to agree a static key by which they can reference the IPC object. The C library offers the `ftok` function, which generates a unique key from a filename and a character. A special key is `IPC_PRIVATE`, which guarantees that no existing IPC object is referenced. Access to objects generated using `IPC_PRIVATE` is only possible via their object numbers.

As with System V, access permissions are managed by the kernel in the structure `ipc_perm`.

```
struct ipc_perm
{
    key_t  key;
    ushort uid;   /* owner */
    ushort gid;   /* owner */
    ushort cuid;  /* creator */
    ushort cgid;  /* creator */
    ushort mode;  /* access modes */
    ushort seq;   /* counter, used to calculate the identifier */
};
```

If a process accesses an object, the routine `ipcperms()` is called, once again using the standard UNIX access flags for the user, the group and others. The superuser, of course, has access at all times. If the uid for the process attempting access matches that of the owner or the creator, the user access permissions are checked. The same applies to checks on group access permissions.

5.5.3 Message queues

Messages consist of a sequence of bytes. In addition, IPC messages in System V include a type code. Processes send messages to the message queues and can receive messages, restricting reception to messages of a specified type if required. Messages are received in the same order in which they are entered in the message queue. The basis of the implementation under LINUX is the structure `msqid_ds`.

As with semaphores, functions are now required for initialization, for sending and receiving messages, for returning information and for releasing message queues. Although the operations to be performed are relatively simple, access protection and the updating of statistical data make things more complicated. The relevant library functions call the system call *ipc*, which passes on the call to the appropriate kernel functions. The function `sys_msgget()` creates a message queue, using the standard parameters for the IPC get functions.

```
int sys_msgget (key_t key, int msgflg);
```

The parameter `key` is a mandatory key and `msgflg` is the same as for the flags in `semget()` (*see* Table 5.6). Messages are sent using the function `sys_msgsnd()`.

```
struct msgbuf {
   long mtype;         /* message type */
   char mtext[1];      /* text of message */
};

int sys_msgsnd (int msqid, struct msgbuf *msgp, int msgsz,
                int msgflg);
```

The parameter `msgsz` is the length of the text in `mtext` and must be no greater than `MSGMAX`. The process blocks if the new number of bytes in the message queue exceeds the value in the component `msg_qbytes`, the permitted maximum. It only resumes processing once other processes have read messages from the queue or when non-blocked signals are sent to the process. Blocking can be prevented by setting the flag `IPC_NOWAIT`.

A message can be read back from the queue by means of `sys_msgrcv()`.

```
int sys_msgrcv (int msqid, struct msgbuf *msgp, int msgsz,
                long msgtyp, int msgflg);
```

The messages to be received are specified in `msgtyp`. If the value is zero, the first message in the queue is selected. For a value greater than zero, the first message of the given type in the message queue is read.

5.5.4 Shared memory

Shared memory is the fastest form of inter-process communication. Processes using a shared section of memory can exchange data by the usual machine code commands for reading and writing data. In all other methods this is only possible by recourse to system calls to copy the data from the memory area of one process to that of the other. The drawback to shared memory is that the processes need to use additional synchronization mechanisms to ensure that race conditions do not arise. Faster communication is only achieved by increased programming effort. Performing the synchronization via other system calls makes for a portable implementation, but reduces the speed advantage. Another possibility would be to exploit the machine code instructions for conditional setting of a bit in the processors for different architectures: these instructions set a bit depending on its value. As this occurs within a machine code instruction, the operation cannot be halted by an interrupt. These instructions provide a very simple and quick way of implementing a system of mutual exclusion. It has already been explained in Section 4.2.2 how complex the shared use of memory areas is. As since version 2.0 it has become possible, with `mmap()`, to map memory areas that can be written to by several processes, this mechanism too can be used to implement shared memory applications.

As in the other IPC variants of System V, a shared segment of memory is identified by a number, which refers to a `shmid_ds` data structure. This segment can be mapped to the user segment in the virtual address space by a process with the aid of an attach operation, and the procedure can be reversed with a detach operation. For simplicity, we will refer to the memory managed by the `shmid_ds` structure as a segment, although this term is already used for

By calling `sys_shmget()` a process can create or set up a reference to a segment.

```
int sys_shmget (key_t key, int size, int shmflg);
```

The parameter `size` specifies the size of the segment. If the segment has already been set up, the parameter may be smaller than the actual size. The flags listed in Table 5.6 may again be set in the parameter `shmflg`.

This function only initializes the `shmid_ds` data structure. No pages in memory are allocated to the segment at this stage. The `shm_pages` field in the `shmid_ds` data structure contains only blank entries after `sys_shmget()` is called.

By far the most important function when using shared memory is `sys_shmat()`. This maps the segment to the process's user segment.

```
int sys_shmat (int shmid, char *shmaddr, inst shmflg,
               ulong *raddr);
```


The function `sys_shmdt()` deletes a mapped page from the user segment of a process.

```
int sys_shmdt (char *shmaddr);
```

The `sys_shmctl()` function is a counterpart to the functions `sys_semctl()` and `sys_msgctl()` mentioned earlier.

```
int sys_shmctl() (int shmid, int cmd, struct shmid_ds *buf);
```

A call to this function using the `IPC_INFO` command will return the maximum

5.5.5 The `ipcs` and `ipcrm` commands

One drawback to the System V IPC is that testing and developing programs that make use of it can easily give rise to the problem whereby IPC resources remain present after the test programs have been completed, when this was in no way intended. The `ipcs` command allows the user to investigate the situation and to delete the resources in question using `ipcrm`.

For example, a program may have set up three semaphore arrays. Information can be obtained via `ipcs` on the shared memory segments, semaphore arrays and message queues to which the user has access.

```
% ipcs

------ Shared Memory Segments --------
shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
semid      owner      perms      nsems      status
1152       kunitz     666        1
1153       kunitz     666        1
1154       kunitz     666        1

------ Message Queues --------
msqid      owner      perms      used-bytes  messages
```

These semaphore arrays can now be deleted (one at a time) using `ipcrm`. The command can also be used analogously for message queues and shared memory segments.

```
% ipcrm sem 1153
resource deleted
% ipcs
```

5.6 IPC with sockets

So far, we have only looked at forms of inter-process communication supporting communication between processes in one computer. The socket programming interface provides for communication via a network as well as locally on a single computer. The advantage of this interface is that it allows network applications to be programmed using the long-established UNIX concept of file descriptors. A particularly good example of this is the INET daemon. The daemon waits for incoming network service requests and then calls the appropriate service program with the socket descriptor as standard input and output. For very simple services, the program called need not contain a single line of network-relevant code.

In this chapter, we limit ourselves to the use and implementation of UNIX domain sockets. Sockets for the INET domain will be covered in Chapter 8.

5.6.1 A simple example

Similar to FIFOs, UNIX domain sockets enable programs to exchange data the connection-oriented way. The following example illustrates how this works. The same `include` files are used for both the client and the server program.

```
/* sc.h */

#include <sys/types.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SERVER "/tmp/server"
```

The job of the client is to send a message to the server along with its process number and to write the server's response to the standard output.

```
/* cli.c - client, connection-oriented model */
#include "sc.h"

int main(void)
{
   int sock_fd;
   struct sockaddr_un unix_addr;
   char buf[2048];
   int n;

   if ((sock_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
   {
      perror("cli: socket()");
      exit(1);
   }

   unix_addr.sun_family = AF_UNIX;
   strcpy(unix_addr.sun_path, SERVER);

   if (connect(sock_fd, (struct sockaddr*) &unix_addr,
               sizeof(unix_addr.sun_family) +
               strlen(unix_addr.sun_path)) < 0)
   {
      perror("cli: connect()");
      exit(1);
   }
```

```
        sprintf(buf, "Hello Server, this is %d.\n", getpid());
        n = strlen(buf) + 1;

        if (write(sock_fd, buf, n) != n)
        {
           perror("cli: write()");
           exit(1);
        }

        if ((n = read(sock_fd, buf, 2047)) < 0)
        {
           perror("cli: read()");
           exit(1);
        }

        buf[n] = '\0';
        printf("Client received: %s\n", buf);

        exit(0);
    }
```

First a socket file descriptor is created with `socket()`. Then the address of the server is generated; for UNIX domain sockets this consists of a filename – in our example this is `/tmp/server`. The client then attempts to set up a connection to the server using `connect()`. If this is successful, it is possible to send data to the server using perfectly standard read and write functions. To be precise, the client does this by sending the message

`Hello Server, this is` *process number of client.*

To enable the server to reply, we need a few more lines of C program.

```
/* srv.c - server, connection-oriented model */

#include <signal.h>
#include "sc.h"

void stop()
{
   unlink(SERVER);
   exit(0);
}

void server(void)
{
```



```
int sock_fd, cli_sock_fd;
struct sockaddr_un unix_addr;
char buf[2048];
int n, addr_len;
pid_t pid;
char *pc;

signal(SIGINT, stop);
signal(SIGQUIT, stop);
signal(SIGTERM, stop);

if ((sock_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
{
   perror("srv: socket()");
   exit(1);
}

unix_addr.sun_family = AF_UNIX;
strcpy(unix_addr.sun_path, SERVER);
addr_len = sizeof(unix_addr.sun_family) +
           strlen(unix_addr.sun_path);

unlink(SERVER);

if (bind(sock_fd, (struct sockaddr *) &unix_addr,
         addr_len) < 0)
{
   perror("srv: bind()");
   exit(1);
}
if (listen(sock_fd, 5) < 0)
{
   perror("srv: client()");
   unlink(SERVER); exit(1);
}

while ((cli_sock_fd =
        accept(sock_fd, (struct sockaddr*) &unix_addr,
               &addr_len)) >= 0)
{
   if ((n = read(cli_sock_fd, buf, 2047)) < 0)
   {
      perror("srv: read()");
      close(cli_sock_fd);
      continue;
   }
```

```
            buf[n] = '\0';
            for (pc = buf; *pc != '\0' && (*pc < '0' || *pc > '9');
                 pc++);

            pid = atol(pc);

            if (pid != 0)
            {
               sprintf(buf, "Hello Client %d, this is the Server.\n",
                       pid);
               n = strlen(buf) + 1;

               if (write(cli_sock_fd, buf, n) != n)
               perror("srv: write()");
            }

            close(cli_sock_fd);
         }

         perror("srv: accept()");
         unlink(SERVER);
         exit(1);
      }

      int main(void)
      {
         int r;
         if ((r = fork()) == 0)
         {
            server();
         }

         if (r < 0)
         {
            perror("srv: fork()");
            exit(1);
         }

         exit(0);
      }
```

The server calls `fork()` and terminates its run. The child process continues running in the background and installs the handling routine for interrupt signals. Once a socket file descriptor has been opened, the server's own address is bound to this socket and a file is created under the pathname given in the address. By limiting the access rights to this file, the server can reduce the number of users able to communicate with it. A client's `connect` call is only successful if this file exists and the client possesses the necessary access rights. The call to `listen()` is necessary to inform the kernel that the process is now ready to accept connections at this socket. It then calls `accept()` to wait. If a connection is set up by a client using `connect()`, `accept()` will return a new socket file descriptor. This will then be used to receive messages from the client and reply to them. The server simply writes back:

`Hello Client` *process number of client*`, this is the Server.`

The server then closes the file descriptor for this connection and again calls `accept()` to offer its services to the next client.

The read and write operations usually block on the socket descriptor if either no data are present or there is no more space in the buffer. If the `O_NONBLOCK` flag has been set with `fcntl()`, these functions do not block.

Since version 2.0 it is possible to use UNIX domain sockets under LINUX in connectionless mode by means of the functions `sendto()` and `recvfrom()`.