Module 7: Inter-Process Communication

Processes execute to accomplish specified computations. An interesting and innovative way to use a computer system is to spread a given computation over several processes. The need for such communicating processes arises in parallel and distributed processing contexts. Often it is possible to partition a computational task into segments which can be distributed amongst several processes. Clearly, these processes would then form a set of communicating processes which cooperate in advancing a solution. In a highly distributed, multi-processor system, these processes may even be resident on different machines. In such a case the communication is supported over a network. A comprehensive coverage of distributed systems is beyond the scope of this book. There are texts like Tanenbaum and Steen [8] which are exclusively devoted to this topic. All the same, we shall study some of the basics to be able to do the following:

- How to spawn (or create) a new process.
- How to assign a task for *exec*ution to this newly spawned process.
- A few mechanisms to enable communication amongst processes.
- Synchronization amongst these processes.

In most cases an IPC package is used to establish inter-process communication. Depending upon the nature of the chosen IPC, the package sets up a data structure in kernel space. These data structures are often persistent. So once the purpose of IPC has been fulfilled, this set-up needs to be deleted (a clean up operation). The usage pattern of the IPC package in a system (like system V Unix) can be seen by using explicit commands like *ipcs*. A user can also remove any unused kernel resources by using a command like *ipcrm.*

For our discussions here, we shall assume Unix like environment. Hopefully, the discussion here offers enough information to partially satisfy and raise the level of curiosity about the distributed computing area.

7.1 Creating A New Process: The *fork()* System Call

One way to bring in a new process into an existing *exec*ution environment is to *exec*ute *fork()* system call. Just to recap how system calls are handled, the reader may refer to Figure 7.1. An application raises a system call using a library of call functions. A system call in turn invokes its service (from the kernel) which may result in memory allocation, device communication or a process creation. The system call *fork()* spawns a new process which, in fact, is a copy of the parent process from where it was invoked!! The newly spawned process inherits its parent's *exec*ution environment. In Table 7.1 we list some of the attributes which the child process inherits from its parent.

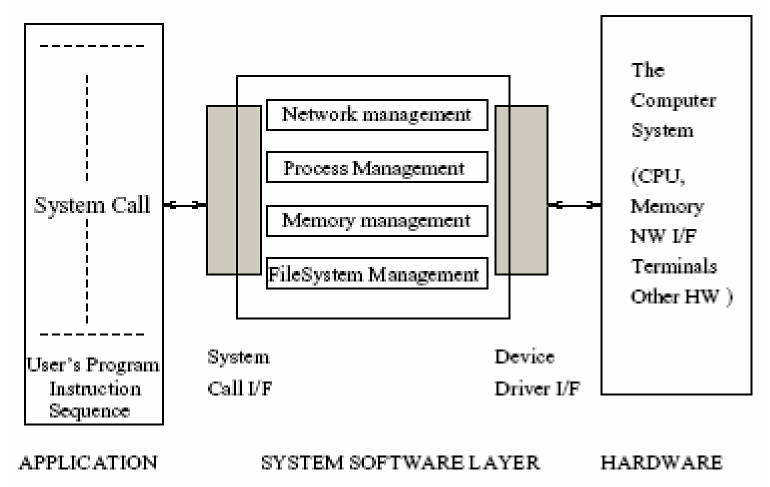
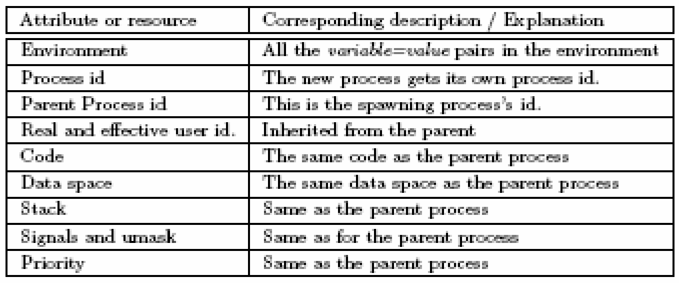


Figure 7.1: Processing system calls.

Note that a child process is a process in its own right. It competes with the parent process to get processor time for *exec*ution. In fact, this can be easily demonstrated (as we shall later see). The questions one may raise are:

- Can one identify when the processor is *exec*uting the parent and when it is *exec*uting the child process?
- What is the nature of communication between the child and parent processes?



Attribute or resource	Corresponding description / Explanation
Environment	All the *variable=value* pairs in the environment
Process id	The new process gets its own process id.
Parent Process id	This is the spawning process's id.
Real and effective user id.	Inherited from the parent
Code	The same code as the parent process
Data space	The same data space as the parent process
Stack	Same as the parent process
Signals and umask	Same as for the parent process
Priority	Same as the parent process

Table 7.1: Properties inherited by a newly forked process.

The answer to the first question is yes. It is possible to identify when the parent or child is in *exec*ution. The return value of *fork()* system call is used to determine this. Using the return value, one can segment out the codes for *exec*ution in parent and child. We will show that in an example later.

The most important communication from parent to child is the *exec*ution environment which includes data and code segments. Also, when the child process terminates, the parent process receives a signal. In fact, a signal of the termination of a child process, is one feature very often exploited by programmers. For instance, one may choose to keep parent process in wait mode till all of its own child processes have terminated. Signaling is a very powerful inter-process communication mechanism (using *signal*s) which we shall learn in Section 7.3.5. The following program demonstrates how a child process may be spawned.

The program: Demonstration of the use of *fork()* system call

```
main()
{ int i, j;
if ( fork() ) /* must be parent */
{ printf("\t\t In Parent \n");
printf("\t\t pid = %d and ppid = %d \n\n", getpid(), getppid());
for (i=0; i<100; i=i+5)
{ for (j=0; j<100000; j++);
printf("\t\t\t In Parent %d \n", i);
}
wait(0); /* wait for child to terminate */
printf("In Parent: Now the child has terminated \n");
}
else
{ printf("\t In child \n");
printf("\t pid = %d and ppid = %d \n\n", getpid(), getppid() );
for (i=0; i<100; i=i+10)
{ for (j=0; j<100000; j++);
printf("\t In child %d \n", i);
} } }
```

The reader should carefully examine the structure of the code above. In particular, note how the return value of system call *fork()* is utilized. On perusing the code we note that, the code is written to *exec*ute in different parts of the program code for the child and the parent. The program makes use of true return value of *fork()* to print “In parent", i.e. if the parent process is presently *exec*uting. The dummy loop not only slows down the *exec*ution but also ensures that we obtain interleaved outputs with a manageable number of lines on the viewing screen.

Response of this program:

```
[bhatt@iiitbsun IPC]$./a.out
In child
pid = 22484 and ppid = 22483
In child 0
In child 10
In child 20
In Parent
pid = 22483 and ppid = 22456
In Parent 0
In Parent 5
In Parent 10
In Parent 15
In child 30
.......
.......
In child 90
In Parent 20
In Parent 25
......
......
In Parent: Now the child has terminated;
```

Let us study the response. From the response, we can determine when the parent process was *exec*uting and when the child process was *exec*uting. The final line shows the result of the *exec*ution of line following wait command in parent. It *exec*utes after the child has fallen through its code. Just as we used a wait command in the parent, we could have also used an exit command explicitly in the child to exit its *exec*ution at any stage. The command pair wait and exit are utilized to have inter-process communication. In particular, these are used to synchronize activities in processes. This program demonstrated how a process may be spawned. However, what one would wish to do is to spawn a process and have it *exec*ute a planned task. Towards this objective, we shall next populate the child code segment with a code for a specified task.

7.2 Assigning Task to a Newly Spawned Process

By now one thing should be obvious: if the child process is to *exec*ute some other code, then we should first identify that *exec*utable (the one we wish to see *exec*uted). For our example case, let us first generate such an *exec*utable. We compile a program entitled *get_int.c* with the command line cc *get_int.c -o int.o*. So, when *int.o exec*utes, it reads in an integer.

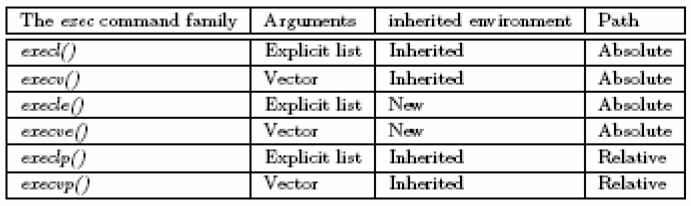
The program to get an integer :

```
#include <stdio.h>
#include <ctype.h>
int get_integer( n_p )
int *n_p;
{ int c;
int mul, sign;
int integer_part;
*n_p = 0;
mul = 10;
while( isspace( c = getchar() ) ); /* skipping white space */
if( !isdigit(c) && c != EOF && c != '+' && c != '-' )
{ /* ungetchar(c); */
printf("Found an invaild character in the integer description \n");
return 0;
}
if (c == '-') sign = -1.0;
if (c == '+') sign = 1.0;
if (c == '-' || c == '+' ) c = getchar();
for ( integer_part = 0; isdigit(c); c = getchar() )
{ integer_part = mul * integer_part + (c - '0');
};
*n_p = integer_part;
if ( sign == -1 ) *n_p = - *n_p;
if ( c == EOF ) return (*n_p);
}
main()
{ int no;
int get_integer();
printf("Input a number as signed or unsigned integer e.g. +5 or -6 or 23\n");
get_integer(&no);
printf("The no. that was input was %d \n", no);
}
```

Clearly, our second step is to have a process spawned and have it *exec*ute the program *int.o.* Unix offers a way of directing the *exec*ution from a specified code segment by using an *exec* command. In the program given below, we spawn a child process and populate its code segment with the program *int.o* obtained earlier. We shall entitle this program as *int_wait.c.*

Program *int_wait.c*

```
#include <stdio.h>
main()
{
if (fork() == 0)
{ /* In child process execute the selected command */
execlp("./int.o", "./int.o", 0);
printf("command not found \n"); /* execlp failed */
fflush(stdout);
```



The *exec* command family	Arguments	inherited environment	Path
execl()	Explicit list	Inherited	Absolute
execv()	Vector	Inherited	Absolute
execle()	Explicit list	New	Absolute
execve()	Vector	New	Absolute
execlp()	Explicit list	Inherited	Relative
execvp()	Vector	Inherited	Relative

Table 7.2: The *exec* command description.

```
exit(1);
```

```
}
else
{ printf("Waiting for the child to finish \n");
wait(0);
printf("Waiting over as child has finished \n");
}
}
```

To see the programs in action follow the steps:

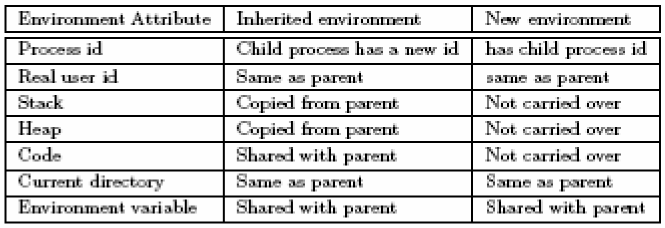
1. cc *get_int.c* -o *int.o*
2. cc *int_wait.c*
3. ./a.out

The main point to note here is that the forked child process gets populated by the code of program *int.o* with the parent *int_wait.c*. Also, we should note the arguments communicated in the *exec* command line.

Before we discuss some issues related to the new *exec*ution environment, a short discussion on *exec* command is in order. The *exec* family of commands comes in several flavors. We may choose an *exec* command to *exec*ute an identified *exec*utable defined using a relative or absolute path name. The *exec*() command may use some other arguments as well. Also, it may be *exec*uted with or without the inherited *exec*ution environment.

Most Unix systems support *exec* commands with the description in Table 7.2. The example above raises a few obvious questions. The first one is: Which are the properties the child retains after it is populated by a different code segment? In Table 7.3 we note that the process ID and user ID of the child process are carried over to the implanted process. However, the data and code segments obtain new information. Though, usually, a child process inherits open file descriptors from the parent, the implanted process may have some restrictions based on file access controls.

With this example we now have a way to first spawn and then populate a child process with the code of an arbitrary process. The implanted process still remains a child process but has its code independent of the parent. A process may spawn any number of child processes. However, much ingenuity lies in how we populate these processes and what form of communication we establish amongst these to solve a problem.



Environment Attribute	Inherited environment	New environment
Process id	Child process has a new id	has child process id
Real user id	Same as parent	same as parent
Stack	Copied from parent	Not carried over
Heap	Copied from parent	Not carried over
Code	Shared with parent	Not carried over
Current directory	Same as parent	Same as parent
Environment variable	Shared with parent	Shared with parent

Table 7.3: New environment description.

7.3 Establishing Inter-process Communication

In this section we shall study a few inter-process communication mechanisms. Each of these uses a different method to achieve communication amongst the processes. The first mechanism we study employs pipes. Pipes, as used in commands like *ls|more*, direct the output stream of one process to feed the input of another process. So for IPC, we need to create a pipe and identify the direction of feeding the pipe. Another way to communicate would be to use memory locations. We can have one process write into a memory location and expect the other process to read from it. In this case the memory location is a shared memory location. Finally, there is one more mechanism in which one may send a message to another process. The receiving process may interpret the message. Usually, the messages are used to communicate an event. We next study these mechanisms.

7.3.1 Pipes as a Mechanism for Inter-process Communication

Let us quickly recap the scheme which we used in section 7.2. The basic scheme has three parts: spawn a process; populate it and use the wait command for synchronization. Let us now examine what is involved in using pipes for establishing a communication between two processes. As a first step we need to identify two *exec*utables that need to communicate. As an example, consider a case where one process gets a character string input and communicates it to the other process which reverses strings. Then we have two processes which need to communicate. Next we define a pipe and connect it between the processes to facilitate communication. One process gets input strings and writes into the pipe. The other process, which reverses strings, gets its input (i.e. reads) from the pipe. Figure 7.2 explains how the pipes are used. As shown in the upper part of the figure, a pipe has an input end and an output end. One can write into a pipe from the input end and read from the output end. A pipe descriptor, therefore, has an array that stores two pointers. One pointer is for its input end and the other is for its output end. When a

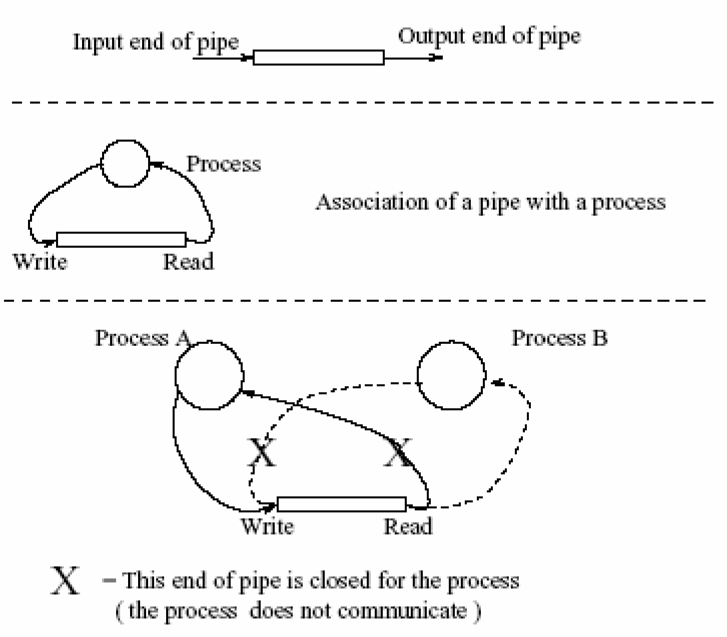

Input end of pipe
Output end of pipe
Process
Association of a pipe with a process
Write
Read
Process A
Process B
X
X
Write
Read
X – This end of pipe is closed for the process
(the process does not communicate)

Figure 7.2: Pipe as an IPC Mechanism.

process defines a pipe it gets both the addresses, as shown in the middle part of Figure 7.2. Let us suppose array pp is used to store the descriptors. pp[0] stores the write end address and pp[1] stores the read end address. Suppose two processes, Process A and Process B, need to communicate, then it is imperative that the process which writes closes its read end of the pipe and the process which read closes its write end of the pipe. Essentially, for a communication from Process A to process B the following should happen. Process A should keep its write end open and close read end of the pipe. Similarly, Process B should keep its read end open and close its write end. This is what is shown in the lower part of Figure 7.2. Let us now describe how we may accomplish this.

1. First we have a parent process which declares a pipe in it.
2. Next we spawn two child processes. Both of these would get the pipe definition which we have defined in the parent. The child processes, as well as the parent, have both the write and read ends of the pipe open at this time.
3. Next, one child process, say Process A, closes its read end and the other child process, Process B, closes its write end.
4. The parent process closes both write and read ends.
5. Next, Process A is populated with code to get a string and Process B is populated to reverse a string.

With the above arrangement the output from Process A is piped as input to Process B. The programs given below precisely achieve this.

In reading the programs, the following interpretations have to be borne in mind:

1. The pipe is defined by the declaration *pipe*(p_des).
2. The *dup* command replaces the standard I/O channels by pipe descriptors.
3. The *execlp* command is used to populate the child process with the desired code.
4. The close command closes the appropriate ends of the pipe.
5. The *get_str* and *rev_str* processes are pre-compiled to yield the required executables.

The reader should be able to now assemble the programs correctly to see the operation of the programs given below:

pipe.c

```
#include <stdio.h>
#include <ctype.h>
main()
{ int p_des[2];
pipe( p_des ); /* The pipe descriptor */
printf("Input a string \n");
if ( fork () == 0 )
{
dup2(p_des[1], 1);
close(p_des[0]); /* process-A closing read end of the pipe */
execlp("./get_str", "get_str", 0);
/*** exit(1); ***/
}
else
if ( fork () == 0 )
{ dup2(p_des[0], 0);
close(p_des[1]); /* process-B closing write end of the pipe */
execlp("./rev_str", "rev_str", 0);
/*** exit(1); ****/
}
else
{ close(p_des[1]); /* parent closing both the ends of pipe */
close(p_des[0]);
wait(0);
wait(0);
}
fflush(stdout);
}
```

get_str.c

```
#include <stdio.h>
#include <ctype.h>
void get_str(str)
char str[];
{ char c;
int ic;
c = getchar();
ic = 0;
while ( ic < 10 && ( c != EOF && c != '\n' && c != '\t' ))
{ str[ic] = c;
c = getchar();
ic++;
}
str[ic] = '\0';
return;
}
```

rev_str.c

```
void rev_str(str1, str2)
char str1[];
char str2[];
{ char c;
int ic;
int rc;
ic = 0;
c = str1[0];
while( ic < 10 && (c != EOF && c != '\0' && c != '\n') )
{ ic++;
c = str1[ic];
}
str2[ic] = '\0';
rc = ic - 1;
ic = 0;
while (rc-ic > -1)
{ str2[rc-ic] = str1[ic];
ic++;
}
return;
}
```

It is important to note the following about pipes as an IPC mechanism:

1. Unix pipes are buffers managed from within the kernel.
2. Note that as a channel of communication, a pipe operates in one direction only.
3. Some plumbing (closing of ends) is required to use a pipe.
4. Pipes are useful when both the processes are schedulable and are resident on the same machine. So, pipes are not useful for processes across networks.
5. The read end of a pipe reads any way. It does not matter which process is connected to the write end of the pipe. Therefore, this is a very insecure mode of communication.
6. Pipes cannot support broadcast.

There is one other method of IPC using special files called “named pipes". We shall leave out its details. Interested readers should explore the suggested reading list of books. In particular, books by Stevenson, Chris Brown or Leach [28], [12], [24] are recommended.

7.3.2 Shared Files

One very commonly employed strategy for IPC is to share files. One process, identified as a writer process, writes into a file. Another process, identified as a reader process, reads from this file. The write process may continually make changes in a file and the other may read these changes as these happen. Unlike other IPC methods, this method does not require special system calls. It is, therefore, relatively easily portable. Of course, for creating processes we shall use the standard system calls *fork()* and *execlp()*. Besides these, there are no other system calls needed. However, we do need code for file creation, access and operations on files. A word of caution is in order. If the reader is faster than the writer, then this method shall have errors. Similarly, if a writer continues writing then the file may grow to unbounded lengths. Both these situations result in errors. This problem of a mismatch in the speed of reader and writer is called the reader writer problem. We earlier learned to resolve similar problems using mutual exclusion of resource sharing. In this case too we can program for mutually exclusive writes and reads.

Shared file pointers: Another way to handle files would be to share file pointers instead of files themselves. Sharing the file pointers with mutual exclusion could be easily done using semaphores.

The shared file pointer method of IPC operates in two steps. In the first step, one process positions a file pointer at a location in a file. In the second step, another process reads from this file from the communicated location. Note that if the reader attempts to read a file even before the writer has written something on a file, we shall have an error. So, in our example we will ensure that the reader process sleeps for a while (so that the writer has written some bytes). We shall use a semaphore simulation to achieve mutual exclusion of access to the file pointer, and hence, to the file.

This method can be used when the two processes are related. This is because the shared file pointer must be available to both. In our example, these two processes shall be a parent and its child. Clearly, if a file has been opened before the child process is spawned, then the file descriptors created by the parent are available to the child process as well. Note that when a process tries to create a file which some other process has already created, then an error is reported.

To understand the programs in the example, it is important to understand some instructions for file operations. We shall use *lseek*() system command. It is used to access a sequence of bytes from a certain offset in the file. The first byte in the file is considered to have an offset of 0. It has the syntax long *lseek*(int fd, long offset, int arg) with the following interpretation.

- With arg = 0, the second argument is treated as an offset from the first byte in file.
- With arg = 1, the current position of the file pointer is changed to sum of the current file pointer and the value of the second argument.
- With arg = 2, the current position of the file pointer is changed to the sum of the size of file and value of the second argument. The value of the second argument can be negative as long as the overall result of the sum is positive or zero.

The example here spans three programs, a main, a reader and a writer program. Let us look at the code for the main program.

```
#include <stdio.h>
#include <fcntl.h>
#define MAXBYTES 4096
void sem_simulation();
main(argc, argv)
int argc;
char *argv[];
{/* the program communicates from parent to child using a shared file pointer */
FILE *fp;
char message[MAXBYTES];
long i;
int mess_num, n_bytes, j, no_of_mess;
int sid, status;
if ( argc < 3 )
{ fputs("Bad argument count \n", stderr);
fputs("Usage: num_messages num_bytes \n", stderr);
exit(1);
}
no_of_mess = atoi(argv[1]);
n_bytes = atoi(argv[2]);
printf("no_of_mess : %6d and n_bytes : %6d \n", no_of_mess, n_bytes );
if(n_bytes > MAXBYTES)
{ fputs("Number of bytes exceeds maximum", stderr);
exit(1);
} /* open a file before creating a child process to share a file pointer*/
else if( ( fp = fopen("./temp_file", "w+" )) == NULL )
{ fputs("Cannot open temp_file for writing \n", stderr);
exit(1);
}
/* create processes and begin communication */
switch (fork ())
{ case -1: fputs("Error in fork ", stderr);
exit( 1 );
case 0: sleep(2);
if(execlp("./readfile", "./readfile", argv[1], argv[2], NULL) == -1)
fputs("Error in exec in child \n", stderr);
exit( 1 );
default: if(execlp("./writefile", "./writefile", argv[1], argv[2], NULL) == -1)
fputs("Error in exec in parent \n", stderr);
exit( 1 );
} /* end switch */
}
```

Now we describe the reader process.

```
#include <stdio.h>
#include <fcntl.h>
#define MAXBYTES 4096
void sem_simulation()
{ if (creat( "creation", 0444) == -1)
{ fputs("Error in create \n", stderr);
system("rm creation");
}
else fputs(" No error in creat \n", stderr);
}
main (argc, argv)
int argc;
char *argv[];
```

```
{ FILE *fp;
long i;
char message[MAXBYTES];
int mess_num, n_bytes, j, no_of_mess;
int sid, status;
void sem_simulation();
no_of_mess = atoi(argv[1]);
n_bytes = atoi(argv[2]);
printf("in read_child \n");
/* read messages from the shared file */
for ( i=0; i < no_of_mess; i++ )
{ sem_simulation();
fseek(fp, i*n_bytes*1L, 0);
while((fgets(message, n_bytes+1, fp)) == NULL ) ;
fseek(fp, i*n_bytes*1L, 0);
sem_simulation();
} /* end of for loop */
exit(0);
}
```

Now let us describe the writer process.

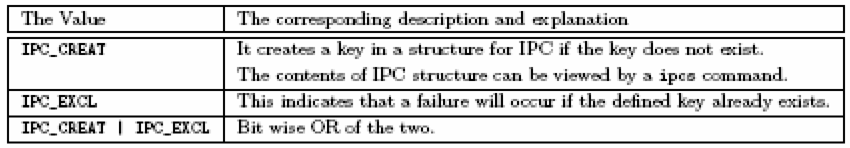
```
#include <stdio.h>
#include <fcntl.h>
#define MAXBYTES 4096
void sem_simulation()
{ if (creat( "creation", 0444) == -1)
{ fputs("Error in create \n", stderr);
system("rm creation");
}
else fputs(" No error in create \n", stderr);
}
main (argc, argv)
int argc;
```

```
char *argv[];
{ FILE *fp;
long i, j, status, message_num;
char message[MAXBYTES];
int n_bytes, no_of_mess;
void sem_simulation();
no_of_mess = atoi(argv[1]);
n_bytes = atoi(argv[2]);
printf("in parent with write option \n");
printf("no_of_mess : %6d n_bytes : %6d \n");
for ( i=0; i < no_of_mess; i++ )
{ /* Create a message with n_bytes */
message_num = i;
for ( j = message_num; j < n_bytes; j++ )
message[j] = 'd';
printf("%s \n", message);
/* Use semaphore to control synchronization, write to end of file */
sem_simulation();
fseek(fp, 0L, 2);
while( ( fputs(message, fp) ) == -1)
fputs("Cannot write message", stderr );
fseek(fp, 0L, 2);
sem_simulation();
}
wait(&status);
unlink("creation");
unlink("./temp_file");
fclose(fp);
}
```

The shared file pointer method is quite an elegant solution and is often a preferred solution where files need to be shared. However, many parallel algorithms require that "objects" be shared. The basic concept is to share memory. Our discussion shall now veer to IPC using shared memory communication.

7.3.3 Shared Memory Communication

Ordinarily, processes use memory areas within the scope of virtual memory space. However, memory management systems ensure that every process has a well-defined and distinct data and code area. For shared memory communication, one process would write into a certain commonly accessed area and another process would read subsequently from



The Value	The corresponding description and explanation
IPC_CREAT	It creates a key in a structure for IPC if the key does not exist. The contents of IPC structure can be viewed by a ipcs command.
IPC_EXCL	This indicates that a failure will occur if the defined key already exists.
IPC_CREAT | IPC_EXCL	Bit wise OR of the two.

Table 7.4: Creating a shared data mechanism.

that area. One other point which we can debate is: do the processes have to be related? We have seen that a parent may share a data area or files with a child. Also, by using the *exec*() function call we may be able to populate a process with another code segment or data. Clearly, the shared memory method can allow access to a common data area even amongst the processes that are not related. However, in that case an area like a process stack may not be shareable. Also, it should be noted that it is important that the shared data integrity may get compromised when an arbitrary sequence of reads and writes occurs. To maintain data integrity, the access is planned carefully under a user program control. That then is the key to shared memory protocol.

The shared memory model has the following steps of execution.

1. First we have to set up a shared memory mechanism in the kernel.
2. Next an identified \safe area" is attached to each of the processes.
3. Use this attached shared data space in a consistent manner.
4. When finished, detach the shared data space from all processes to which it was attached.
5. Delete the information concerning the shared memory from the kernel.

Two important *.h* files in this context are: *shm.h* and *ipc.h* which are included in all the process definitions. The first step is to set up shared memory mechanism in kernel. The required data structure is obtained by using *shmget()* system call with the following syntax.

int shmget(key_t key, int size, int flag);

The parameter key_t is usually a long int. It is declared internally as key_t key. key_t is an alias defined in sys/types.h using a typedef structure. If this key is set to IPC_PRIVATE, then it always creates a shared memory region. The second parameter, size is the size of the sh-mem-region in bytes. The third parameter is a combination of usual file access permissions of r/w/e for o/g/w with the interpretation of non-zero constants as explained in Table 7.4.

A successful call results in the creation of a shared memory data structure with a defined id. This data structure has the following information in it.

```
struct shmid_ds
{ struct ipc_perm shm_perm;
int shm_seg_segsz /* size of segments in bytes */
struct region *shm_reg; /* pointer to region struct */
char pad[4]; /* for swap compatibility */
ushort shm_lpid; /* pid of last shmop */
ushort shm_cpid; /* pid of creator */
ushort shm_nattch; /* used for shm_info */
ushort shm_cnattch; /* used for shm_info */
time_t shm_atime; /* last attach time */
time_t shm_dtime; /* last detach time */
time_t shm_ctime; /* last change time */
}
```

Once this is done we would have created a shared memory data space. The next step requires that we attach it to processes that would share it. This can be done using the system call *shmat()*. The system call *shmat()* has its syntax shown below.

*char *shamt(int shmid, char *shmaddr, int shmflg);*

The second argument should be set to zero as in (char *)0, if the kernel is to determine the attachment. The system uses three possible flags which are: SHM_RND, SHM_RDONLY and the combination SHM_RND | SHM_RDONLY. The SHM_RDONLY flag indicates the shared region is read only. Otherwise, it is both for read and write operations. The flag SHM_RND requires that the system enforces use of the byte address of the shared memory region to coincide with a double word boundary by rounding.

Now that we have a well-defined shared common area, reading and writing can be done in this shared memory region. However, the user must write a code to ensure locking of the shared region. For instance, we should be able to block a process attempting to write while a reader process is reading. This can be done by using a synchronization method such as semaphores. In most versions of Unix, semaphores are available to enforce mutual exclusion. At some stage a process may have finished using the shared memory region. In that case this region can be detached for that process. This is done by using the *shmdt()* system call. This system call detaches that process from future access. This information is kept within the kernel data-space. The system call *shmdt()* takes a single argument, the address of the shared memory region. The return value from the system call is rarely used except to check if an error has occurred (with -1 as the return value). The last step is to clean up the kernel's data space using the system call *shmctl().* The system call *shmctl()* takes three parameters as input, a shared memory id, a set of flags, and a buffer that allows copying between the user and the kernel data space.

A considerable amount of information is pointed to by the third parameter. A call to *shmctl()* with the command parameter set to IPC_STAT gives the following information.

- User's id
- Creator's group id
- Operation permissions
- Key
- segment size
- Process id of creator *
- Current number of attached segments in the memory.
- Last time of attachment
- User's group id
- Creator's id
- Last time of detachment
- Last time of change
- Current no. of segments attached
- Process id of the last shared memory operation

Now let us examine the *shmget()* system call.

int shmget(key_t key, int region_size, int flags);

Here key is a user-defined integer, the size of the shared region to be attached is in bytes. The flags usually turn on the bits in IPC_CREAT. Depending upon whether there is key entry in the kernel's shared memory table, the *shmget()* call takes on one of the following two actions. If there is an entry, then *shmget()* returns an integer indicating the position of the entry. If there is no entry, then an entry is made in the kernel's shared memory table. Also, note that the size of the shared memory is specified by the user. It, however, should satisfy some system constraints which may be as follows.

```
struct shminfo
{ int shmmax,       /* Maximum shared memory segment size 131072 for some */
shmmin,             /* minimum shared memory segment size 1 for some */
shmni,              /* No. of shared memory identifiers */
shmseg,             /* Maximum attached segments per process */
shmall;             /* Max. total shared memory system in pages */
};
```

The third parameter in *shmget()* corresponds to the flags which set access permissions as shown below:

400 read by user ...... Typically in shm.h file as constant SHM_R

200 write by user ....Typically in shm.h file as constant SHM_W

040 read by group

020 write by group

004 read by others

002 read by others ......All these are octal constants.

For example, let us take a case where we have read/write permissions by the user's group and no access by others. To be able to achieve this we use the following values.

SHM_R | SHM_W | 0040 | IPC_CREAT as a flag to a call to *shmget().*

Now consider the *shmat()* system call.

*char *shmat(int shmid, char *address, int flags);*

This system call returns a pointer to the shared memory region to be attached. It must be preceded by a call to *shmget*(). The first argument is a *shmid* (returned by *shmget*()). It is an integer. The second argument is an address. We can let the compiler decide where to attach the shared memory data space by giving the second argument as (char *) 0. The flags in arguments list are to communicate the permissions only as SHM_RND and SHM_RDONLY. The *shmdt()* system call syntax is as follows:

*int shmdt(char * addr);*

This system call is used to detach. It must follow a call *shmat()* with the same base address which is returned by *shmat()*. The last system call we need is *shmctl()*. It has the following syntax.

*int shmctl(int shmid, int command, struct shm_ds *buf_ptr);*

The *shmctl()* call is used to change the ownership and permissions of the shared region. The first argument is the one earlier returned by *shmget()* and is an integer. The command argument has five possibilities:

- IPC_STAT : returns the status of the associated data structure for the shared memory pointed by buffer pointer.
- IPC_RMID : used to remove the shared memory id.
- SHM_LOCK : used to lock
- SHM_UNLOCK : used to unlock
- IPC_SET : used to set permissions.

When a region is used as a shared memory data space it must be from a list of free data space. Based on the above explanations, we can arrive at the code given below.

```
include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#define MAXBYTES 4096 /* Maximum bytes per shared segment */
main(argc, argv)
int argc;
char *argv[];
{ /* Inter process communication using shared memory */
char message[MAXBYTES];
```

```
int i, message_num, j, no_of_mess, nbytes;
int key = getpid();
int semid;
int segid;
char *addr;
if (argc != 3) { printf("Usage : %s num_messages");
printf("num_of_bytes \n", argv[0]);
exit(1);
}
else
{ no_of_mess = atoi(argv[1]);
nbytes = atoi(argv[2]);
if (nbytes > MAXBYTES) nbytes = MAXBYTES;
if ( (semid=semget( (key_t)key, 1, 0666 | IPC_CREAT ))== -1)
{ printf("semget error \n");
exit(1);
}
/* Initialise the semaphore to 1 */
V(semid);
if ( (segid = shmget( (key_t) key, MAXBYTES, 0666 |
IPC_CREAT ) ) == -1 )
{ printf("shmget error \n");
exit(1);
}
/*if ( (addr = shmat(segid, (char * )0,0)) == (char *)-1) */
if ( (addr = shmat(segid, 0, 0)) == (char *) -1 )
{ printf("shmat error \n");
exit(1);
}
switch (fork())
{ case -1 : printf("Error in fork \n");
exit(1);
case 0 : /* Child process, receiving messages */
for (i=0; i < no_of_mess; i++)
if(receive(semid, message, sizeof(message)));
exit(0);
default : /* Parent process, sends messages */
for ( i=0; i < no_of_mess; i++)
{ for ( j=i; j < nbytes; j++)
message[j] = 'd';
if (!send(semid, message, sizeof(message)))
printf("Cannot send the message \n");
} /* end of for loop */
} /* end of switch */
} /* end of else part */
}
/* Semaphores */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
int sid;
cleanup(semid, segid, addr)
int semid, segid;
char *addr;
{ int status;
/* wait for the child process to die first */
/* removing semaphores */
wait(&status);
semctl(semid, 0, IPC_RMID, 0);
shmdt(addr);
shmctl(segid, 0, IPC_RMID, 0);
};
```

```
P(sid)
int sid;
{ /* Note the difference in this and previous structs */
struct sembuf *sb;
sb = (struct sembuf *) malloc(sizeof(struct sembuf *));
sb -> sem_num = 0;
sb -> sem_op = -1;
sb -> sem_flg = SEM_UNDO;
if( (semop(sid, sb, 1)) == -1) puts("semop error");
};
V(sid)
int sid;
{ struct sembuf *sb;
sb = (struct sembuf *) malloc(sizeof(struct sembuf *));
sb -> sem_num = 0;
sb -> sem_op = 1;
sb -> sem_flg = SEM_UNDO;
if( (semop(sid, sb, 1)) == -1) puts("semop error");
};
/* send message from addr to buf */
send(semid, addr, buf, nbytes)
int semid;
char *addr, *buf;
int nbytes;
{ P(semid);
memcpy(addr, buf, nbytes);
V(semid);
}
/* receive message from addr to buf */
receive(semid, addr, buf, nbytes)
int semid;
char *addr, *buf;
```

```
int nbytes;
{ P(semid);
memcpy(buf, addr, nbytes);
V(semid);
}
```

From the programs above, we notice that any process is capable of accessing the shared memory area once the key is known to that process. This is one clear advantage over any other method. Also, within the shared area the processes enjoy random access for the stored information. This is a major reason why shared memory access is considered efficient. In addition, shared memory can support many-to-many communication quite easily. We shall next explore message-based IPC.

7.3.4 Message-Based IPC

Messages are a very general form of communication. Messages can be used to send and receive formatted data streams between arbitrary processes. Messages may have types. This helps in message interpretation. The type may specify appropriate permissions for processes. Usually at the receiver end, messages are put in a queue. Messages may also be formatted in their structure. This again is determined by the application process. Messages are also the choice for many parallel computers such as Intel's hyper-cube. The following four system calls achieve message transfers amongst processes.

- *msgget()* returns (and possibly creates) message descriptor(s) to designate a message queue for use in other systems calls.
- *msgctl()* has options to set and return parameters associated with a message descriptor. It also has an option to remove descriptors.
- *msgsnd()* sends a message using a message queue.
- *msgrcv()* receives a message using a message queue.

Let us now study some details of these system calls.

msgget() system call : The syntax of this call is as follows:

int msgget(key_t key, int flag);

The *msgget()* system call has one primary argument, the key, a second argument which is a flag. It returns an integer called a qid which is the id of a queue. The returned qid is an index to the kernel's message queue data-structure table. The call returns -1 if there is an error. This call gets the resource, a message queue. The first argument key_t, is defined in sys/types.h file as being a long. The second argument uses the following flags:

- MSG_R : The process has read permission
- MSG_W : The process has write permission
- MSG_RWAIT : A reader is waiting to read a message from message queue
- MSG_WWAIT : A writer is waiting to write a message to message queue
- MSD_LOCKED : The msg queue is locked
- MSG_LOCKWAIT : The msg queue is waiting for a lock
- IPC_NOWAIT : Described earlier
- IPC_EXCL : ....

In most cases these options can be used in bit-ored manner. It is important to have the readers and writers of a message identify the relevant queue for message exchange. This is done by associating and using the correct qid or key. The key can be kept relatively private between processes by using a *makekey()* function (also used for data encryption). For simple programs it is probably sufficient to use the process id of the creator process (assuming that other processes wishing to access the queue know it). Usually, kernel uses some algorithm to translate the key into qid. The access permissions for the IPC methods are stored in IPC permissions structure which is a simple table. Entries in kernel's message queue data structures are C structures. These resemble tables and have several fields to describe permissions, size of queue, and other information. The message queue data structure is as follows.

```
struct meqid_ds
{ struct ipc_perm meg_perm; /* permission structure */
struct msg *msg_first; /* pointer to first message */
struct msg *msg_last; /* ........... last ..........*/
ushort msg_cbytes; /* no. of bytes in queue */
ushort msg_qnum; /* no. of messages on queue */
ushort msg_qbytes; /* Max. no. of bytes on queue */
ushort msg_lspid; /* pid of last msgsnd */
ushort msg_lrpid; /* pid of the last msgrcv */
time_t msg_stime; /* last msgsnd time */
time_t msg_rtime; /* .....msgrcv................*/
```

time_t msg_ctime; / last change time */*

}

There is one message structure for each message that may be in the system.

struct msg

*{ struct msg *msg_next; /* pointer to next message */*

long msg_type; / message type */*

ushort msg_ts; / message text size */*

ushort msg_spot; / internal address */*

Note that several processes may send messages to the same message queue. The “type" of message is used to determine which process amongst the processes is the originator of the message received by some other process. This can be done by hard coding a particular number for type or using process-id of the sender as the msg_type. The *msgctl()* function call: This system call enables three basic actions. The most obvious one is to remove message queue data structure from the kernel. The second action allows a user to examine the contents of a message queue data structure by copying them into a buffer in user's data area. The third action allows a user to set the contents of a message queue data structure in the kernel by copying them from a buffer in the user's data area. The system call has the following syntax.

*int msgctl(int qid, int command, struct msqid_ds *ptr);*

This system call is used to control the resource (a message queue). The first argument is the qid which is assumed to exist before call to *msgctl()*. Otherwise the system is in error state. Note that if *msgget()* and *msgctl()* are called by two different processes then there is a potential for a \race" condition to occur. The second argument command is an integer which must be one of the following constants (defined in the header file sys/msg.h).

- IPC STAT: Places the contents of the kernel structure indexed by the first argument, qid, into a data structure pointed to by the third argument, ptr. This enables the user to examine and change the contents of a copy of the kernel's data structure, as this is in user space.
- IPC SET: Places the contents of the data structures in user space pointed to by the third argument, ptr, into the kernel's data structure indexed by first argument qid, thus enabling a user to change the contents of the kernel's data structure. The only

fields that a user can change are msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes.

- IPC RMID : Removes the kernel data structure entry indexed by qid.

The msgsnd() and msgrcv() system calls have the following syntax.

*int msgsnd(int qid, struct msgbuf *msg_ptr, int message_size, int flag);*

*int msgrcv(int qid, struct msgbuf *msg_ptr, int message_size, int msgtype, int flag);*

Both of these calls operate on a message queue by sending and receiving messages respectively. The first three arguments are the same for both of these functions. The syntax of the buffer structure is as follows.

struct msgbuf{ long mtype; char mtext[1]; }

This captures the message type and text. The flags specify the actions to be taken if the queue is full, or if the total number of messages on all the message queues exceeds a prescribed limit. With the flags the following actions take place. If IPC_NOWAIT is set, no message is sent and the calling process returns without any error action. If IPC_NOWAIT is set to 0, then the calling process suspends until any of the following two events occur.

1. A message is removed from this or from other queue.
2. The queue is removed by another process. If the message data structure indexed by qid is removed when the flag argument is 0, an error occurs (*msgsnd()* returns -1).

The fourth arg to *msgrcv()* is a message type. It is a long integer. The type argument is used as follows.

- If the value is 0, the first message on the queue is received.
- If the value is positive, the queue is scanned till the first message of this type is received. The pointer is then set to the first message of the queue.
- If the value is -ve, the message queue is scanned to find the first message with a type whose value is less than, or equal to, this argument.

The flags in the *msgrcv()* are treated the same way as for *msgsnd().*

A successful *exec*ution of either *msgsnd()*, or *msgrcv()* always updates the appropriate entries in msgid_ds data structure. With the above explanation, let us examine the message passing program which follows.

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/msg.h>
main(argc, argv)
int argc;
char *argv[];
{ int status, pid, pid1;
if (( pid=fork())==0) execlp("./messender", "messender", argv[1], argv[2], 0);
if (( pid1=fork())==0) execlp("./mesrec", "mesrec", argv[1], 0);
wait(&status); /* wait for some child to terminate */
wait(&status); /* wait for some child to terminate */
}
```

Next we give the message sender program.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main(argc, argv)
int argc;
char *argv[];
/* This is the sender. It sends messages using IPC system V messages queues.*/
/* It takes two arguments : */
/* No. of messages and no. of bytes */
/* key_t MSGKEY = 100; */
/* struct msgformat {long mtype; int mpid; char mtext[256]} msg; */
{
key_t MSGKEY = 100;
struct msgformat { long mtype;
int mpid;
char mtext[256];
} msg;
int i ;
int msgid;
int loop, bytes;
extern cleanup();
loop = atoi(argv[1]);
bytes = atoi(argv[2]);
printf("In the sender child \n");
for ( i = 0; i < bytes; i++ ) msg.mtext[i] = 'm';
printf("the number of 'm' s is : %6d \n", i);
msgid = msgget(MSGKEY, 0660 | IPC_CREAT);
msg.mtype = 1;
msg.mpid = getpid();
/* Send number of messages specified by user argument */
for (i=0; i<loop; i++) msgsnd(msgid, &msg, bytes, 0);
printf("the number of times the messages sent out is : %6d \n", i);
/* Cleaning up; maximum number queues 32 */
for (i=0; i<32; i++) signal(i, cleanup);
}
cleanup()
{ int msgid;
msgctl(msgid, IPC_RMID, 0);
exit(0);
}
```

|Now we give the receiver program listing.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main(argc, argv)
int argc;
char *argv[];
/* The receiver of the two processes communicating message using */
/* IPC system V messages queues. */
/* It takes two arguments: No. of messages and no. of bytes */
/* key_t MSGKEY = 100; */
/* struct msgformat {long mtype; int mpid; char mtext[256]} msg; */
```

```
{
key_t MSGKEY = 100;
struct msgformat { long mtype;
int mpid;
char mtext[256];
} msg;
int i, pid, *pint;
int msgid;
int loop, bytes;
msgid = msgget(MSGKEY, 0777);
loop = atoi(argv[1]);
bytes = atoi(argv[2]);
for ( i = 0; i <= bytes; i++ )
{ printf("receiving a message \n");
msgrcv(msgid, &msg, 256, 2, 0);
} }
```

If there are multiple writer processes and a single reader process, then the code shall be somewhat along the following lines.

```
if ( mesg_type == 1) { search mesg_queue for type 1; process msg_type type 1 }
.
.
if ( mesg_type == n) { search mesg_queue for type n; process msg_type type n }
```

The number and size of messages available is limited by some constant in the IPC package.

In fact this can be set in the system V IPC package when it is installed. Typically the constants and structure are as follows.

```
MSGPOOL 8
MSGMNB 2048                      /* Max. no. of bytes on queue */
MSGMNI 50                        /* No. of msg. queue identifiers */
MSGTQL 50                        /* No. of system message headers */
MSGMAP 100                       /* No. of entries in msg map */
MSGMAX ( MSGPOOL *1024 )         /* Maximum message size */
```

MSGSSZ 8 / Message segment size */*

*MSGSEG ((MSGPOOL *1024) / MSGSSZ) /* No. of msg. segments */*

Finally, we may note that the message queue information structure is as follows.

struct msginfo{int msgmap, msgmax, msgmnb, msgmni, msgssz, msgtql; ushort msgseg}

From the programs above, it should be obvious that the message-based IPC can also be used for merging multiple data streams (multiplexing). As messages carry senders' id it should also be possible to do de-multiplexing. The message type may also capture priorities. Prioritizing messages can be very useful in some application contexts. Also, note that the communicating parties need not be active at the same time. In our program descriptions we used signals. Note that signals too, are messages! Signals are important and so we shall discuss these in the next subsection.

7.3.5 Signals as IPC

Within the suite of IPC mechanisms, signals stand out for one very good reason. A signal, as a mechanism, is one clean way to communicate asynchronous events. In fact, we use signals more often than any other means of IPC. Every time we abort a program using ^c, a signal is generated to break. Similarly, if an unexpected value for a pointer is generated, we have core dump and a segmentation fault recognized. When we change a window size, a signal is generated. Note that in all these examples, an event happens within the process or the process receives it as an input. In general, a process may send a signal to another process. In all these situations the process receiving a signal needs to respond. We shall first enumerate typical sources of signal, and later examine the possible forms of responses that are generated. Below we list the sources for signal during a process *exec*ution:

1. From the terminal: Consider a process which has been launched from a terminal and is running. Now if we input the interrupt character, ^c, from the keyboard then we have a signal SIGINT initiated. Suppose, we have disconnect of the terminal line (this may happen when we may close the window for instance), then there is a signal SIGHUP to capture the hanging up of the line.
2. From window manager: This may be any of the mouse activity that may happen in the selected window. In case of change of size of the window the signal is SIGWINCH.

3. From other subsystems: This may be from memory or other subsystems. For instance, if a memory reference is out of the process's data or code space, then there shall be a signal SIGSEGV.
4. From kernel: The typical usage of time in processes can be used to set an alarm. The alarm signal is SIGALARM.
5. From the processes: It is not unusual to kill a child process. In fact, sometimes we may kill a job which may have entered an infinite loop. There may be other reasons to abort a process. The typical kill signal is SIGKILL. One of the uses is when a terminal hangs, the best thing to do is to log in from another terminal and kill the hanging process. One may also look upon the last case as a shell initiated signal. Note that a shell is it self a process.

Above we have noted various sources from where signals may be generated. Usually this helps to define the signal type. A process may expect certain types of signals and make a provision for handling these by defining a set of signal handlers. The signal handlers can offer a set of responses which may even include ignoring certain signals! So next, we shall study the different kind of signal responses which processes may generate.

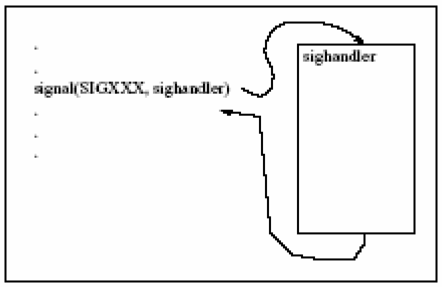


Figure 7.3: Processing signals.

In Figure 7.3 we see a program statement *signal*(SIGXXX, sighandler) to define how this process should respond to a signal. In this statement SIGXXX identifies the signal and sighandler identifies a signal service routine. In general, a process may respond to a given signal in one of the following ways.

1. Ignore it: A process may choose to ignore some kinds of signal. Since processes may receive signals from any source, it is quite possible that a process would authenticate the process before honoring the signal. In some cases then a process may simply ignore the signal and offer no response at all.

2. Respond to it: This is quite often the case in the distributed computing scenarios where processes communicate to further computations in steps. These signals may require some response. The response is encoded in the signal handler. For instance, a debugger and the process being debugged would require signal communication quite often. Another usage might be to advise a clean-up operation. For instance, we need to clean-up following the shared memory mode of IPC. Users of Java would recognize that response for exception handling falls in the same category.
3. Reconfigure: This is required whenever system services are dynamically reconfigured. This happens often in fault-tolerant systems or networked systems. The following is a good example of dynamic configuration. Suppose we have several application servers (like WebSphere) provisioning services. A dispatcher system allocates the servers. During operations, some server may fail. This entails redeployment by the dispatcher. The failure needs to be recognized and dispatching reconfigured for future.
4. Turn on/off options: During debugging as well as profiling (as discussed in chapter on “Other Tools") we may turn some options \On" or \Off" and this may require some signals to be generated.
5. Timer information: In real-time systems, we may have several timers to keep a tab on periodic events. The ideas is to periodically generate required signals to set up services, set alarms or offer other time-based services.

In this chapter we examined the ways to establish communication amongst processes. Its a brief exposure. To comprehend the distributed computing field, it is important to look up the suggested reading list. Interested readers should explore PVM (parallel virtual machine) [30] and MPI (message passing interface) [31] as distributed computing environments.