```
/*
                    PROCESS FORKING

  In Linux we have multiple processes executing at the same time. This
  program illustrates the fork() system call which creates a new child
  process of the calling process. Go through the comments carefully
  before you compile and execute the program.

*/

#include <stdio.h>
#include <sys/ipc.h>  /* You must include this file for all activities
                         related to creation and interprocess communication.
                         IPC stands for Inter-Process Communication */

/*
   When the program begins executing from main(), it is a single process
   having a single process identifier (PID). When the fork() call is
   executed, a new process with a new PID is created. THE CODE OF THE NEW
   PROCESS IS IDENTICAL TO THE PARENT PROCESS, AND THE VALUES OF ALL
   VARIABLES IS A COPY OF THE VALUES OF THE VARIABLES OF THE PARENT AT THE
   TIME OF THE fork() CALL. The PIDs of the currently executing processes
   and the PIDs of their parents can be seen by using the "ps -ef" command
   from the command prompt.
*/

main()
{
        printf("Now there is a single process\n");
        printf("Observe what happens when we fork\n");
        printf("You can stop by pressing Control-c\n");
    printf("Press Enter to continue... \n"); getchar();

        /*
          For the parent process, the fork() system call returns the
          PID of the newly created child process. On the other hand
      the newly created child process, whose execution starts from
      the fork() statement (that is, its control flow starts from
      the fork() statement) receives the value 0 as return value
      from the fork() statement.
    */
        if (fork() == 0) {

                /*
                  We are here only if fork() returned zero. Therefore,
           this part of the code is executed only by the child
                  process.
                */
                while (1)
                  printf("\t\t\t Child executing\n ");
        }
```

```
        else {

                /*
                  We are here only if fork() returned a non-zero value.
                  Therefore this part of the code is executed only by
                  the original (or parent) process.
                */
                while (1)
                  printf("Parent executing\n");
        }
}

/*
  After execution:

  1. Did you observe how the parent and child take turns in execution?
    -- Each process in the system gets a time slice to execute after
      which it is put away until its next turn. Time is maintained by
          a counter called "timer". At the end of a time slice, the timer
      interrupts the CPU, causing the CPU to execute an interrupt
          service routine. This routine calls the process scheduler which
      selects the next process to be executed in the next time slice.

  2. Store the value returned by the fork() call in an integer variable
    and print out its value both in the parent as well as in the child.
    Does your observation match with your expectation?

*/
```

```
/*
                        PROCESS FORKING REVISITED

  This program illustrates the fork() system call in more detail.

*/

#include <stdio.h>
#include <sys/ipc.h>

main()
{
        int i ;
        int x = 10 ;
        int pid1, pid2 ;

        printf("Before forking, the value of x is %d\n", x);

        /*
          After forking, we make the parent and its two children
       increment x in different ways to illustrate that they
          have different copies of x
```

```
        */
        if ((pid1 = fork()) == 0) {

                /* First child process */
                for (i=0 ; i < 5; i++) {
                  printf("\t\t\t At first child: x= %d\n", x);
                  x= x+10;
                  sleep(1) ; /* Sleep for 1 second */
                }
        }
        else {

                /* Parent process */

                /* Create another child process */
                if ((pid2 = fork()) == 0) {

                   /* Second child process */
            for (i=0 ; i < 5; i++) {
                printf("\t\t\t\t\t\t At second child: x= %d\n", x);
                x= x+20;
                        sleep(1) ; /* Sleep for 1 second */
            }
                }
                else {

                        /* Parent process */
                        for (i=0 ; i < 5; i++) {
                                printf("At parent: x= %d\n", x);
                                x= x+5;
                                sleep(1) ; /* Sleep for 1 second */
                        }

                        /*
                           The wait() system call causes the parent
                           to wait for a child process to complete
                           its execution. The input parameter can
                           specify the PID of the child process for
                           which it has to wait.
                        */

                        wait(pid1);
                        wait(pid2);
                }
        }
}
```

```
/*
                SHARING MEMORY BETWEEN PROCESSES

   In this example, we show how two processes can share a common
   portion of the memory. Recall that when a process forks, the
```

```
   new child process has an identical copy of the variables of
   the parent process. After fork the parent and child can update
   their own copies of the variables in their own way, since they
   dont actually share the variable. Here we show how they can
   share memory, so that when one updates it, the other can see
   the change.

*/

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>    /*  This file is necessary for using shared
                            memory constructs
                        */

main()
{
        int shmid;
        int *a, *b;
        int i;

        /*
           The operating system keeps track of the set of shared memory
           segments. In order to acquire shared memory, we must first
           request the shared memory from the OS using the shmget()
           system call. The second parameter specifies the number of
           bytes of memory requested. shmget() returns a shared memory
           identifier (SHMID) which is an integer. Refer to the online
           man pages for details on the other two parameters of shmget()
        */
        shmid = shmget(IPC_PRIVATE, 2*sizeof(int), 0777|IPC_CREAT);
                        /* We request an array of two integers */

        /*
           After forking, the parent and child must "attach" the shared
           memory to its local data segment. This is done by the shmat()
           system call. shmat() takes the SHMID of the shared memory
           segment as input parameter and returns the address at which
           the segment has been attached. Thus shmat() returns a char
           pointer.
        */
        if (fork() == 0) {

                /* Child Process */

                /*  shmat() returns a char pointer which is typecast here
                   to int and the address is stored in the int pointer b. */
                b = (int *) shmat(shmid, 0, 0);

                for( i=0; i< 10; i++) {
                        sleep(1);
```

```
                printf("\t\t\t Child reads: %d,%d\n",b[0],b[1]);
        }

    }
    else {

        /* Parent Process */

        /*  shmat() returns a char pointer which is typecast here
            to int and the address is stored in the int pointer a.
            Thus the memory locations a[0] and a[1] of the parent
            are the same as the memory locations b[0] and b[1] of
            the parent, since the memory is shared.
        */
        a = (int *) shmat(shmid, 0, 0);

        a[0] = 0; a[1] = 1;
        for( i=0; i< 10; i++) {
                sleep(1);
                a[0] = a[0] + a[1];
                a[1] = a[0] + a[1];
                printf("Parent writes: %d,%d\n",a[0],a[1]);
        }
        wait();
    }
}

/*
    POINTS TO NOTE:

    In this case we find that the child reads all the values written
    by the parent. Also the child does not print the same values
    again.

    1. Modify the sleep in the child process to sleep(2). What
       happens now?

    2. Restore the sleep in the child process to sleep(1) and modify
   the sleep in the parent process to sleep(2). What happens now?

    Thus we see that when the writer is faster than the reader, then
    the reader may miss some of the values written into the shared
    memory. Similarly, when the reader is faster than the writer, then
    the reader may read the same values more than once. Perfect
    inter-process communication requires synchronization between the
    reader and the writer.

    Further note that "sleep" is not a synchronization construct.
  We use "sleep" to model some amount of computation which may
    exist in the process in a real world application.
*/
```

```
/*
                        COMMUNICATING VIA A PIPE

   In the previous example we saw that communicating through shared
   memory requires synchronization between the two processes in order
   to ensure that all values that are written by one into the shared
   memory is read (and read once) by the other. Pipe is a system
   construct which facilitates such communication. A pipe is also a
   shared buffer. When a process tries to read from an empty pipe, it
   waits until someone has written something into the pipe. When the
   pipe is full, any process attempting to write into the pipe is made
   to wait.
*/

#include <stdio.h>
#include <unistd.h>      /* Include this file to use pipes */

#define BUFSIZE 80       /* We will write lines of 80 chars into the pipe.
                            The pipe has a large capacity and can accomodate
                            many such lines.
                         */

main()
{
        int fd[2];
        int n=0;
        int i;
        char line[BUFSIZE];

        /*  A pipe is treated as a file by the system. You must have used
            fopen() to open a file. fopen() returns a "file pointer" which
            is used in fprintf(), fscanf(), fclose() etc. However, when we
            wish to perform reads and writes in blocks from a file, we use
            the system call "open" to open a file. Internally files are
            always opened using the "open" call. For each process the system
            maintains a "file descriptor table" (FDT) containing an entry
            for each file opened by that process. When a new file is opened,
            a new entry is created in the FDT, and the entry number is
            returned as an integer called "file descriptor".

            Unlike in a file, we may want to both read and write from a
            pipe at the same time. Hence when a pipe is created, two file
            descriptors are created -- one for reading the pipe and one for
            writing into the pipe. The pipe() system call requires an
            array of two integers as parameter. The system returns the file
            descriptors through this array.
        */
        pipe(fd);  /* fd[0] is for reading,
                      fd[1] is for writing
                   */
```

```
        /*  To illustrate the working of the pipe, we will make the child
            process write the integer n into the pipe and make the parent
            to read from the pipe. We put sleep in the writer process
            (in this case the child) to show that the reader process waits
            for the writer to write into the pipe.

            To write a block of bytes into a pipe (or more generally into
            a file) the write() system call is used. Similarly read() is
            used to read a block of bytes from a file. Refer to the online
            man pages for these calls.
        */

        if (fork() == 0) {

                close(fd[0]); /* The child will not read and
                                 hence we close fd[0]
                              */

                for (i=0; i < 10; i++) {

                        sprintf(line,"%d",n); /* Since write() accepts only
                                                 arrays of characters, we
                                                 first write the integer n
                                                 into the char array "line"
                                              */
                        write(fd[1], line, BUFSIZE);
                        printf("Child writes: %d\n",n);
                        n++;
                        sleep(2);
                }
        }
        else {

                close(fd[1]); /* The parent will not write and
                                 hence we close fd[1]
                              */

                for (i=0; i < 10; i++) {

                        printf("\t\t\t Parent trying to read pipe\n");
                        read(fd[0], line, BUFSIZE);
                        sscanf(line,"%d",&n); /* Read the integer from the
                                                 line of characters read
                                                 from the pipe
                                              */
                        printf("\t\t\t Parent reads: %d\n",n);
                }
        }
}
```

```
/*
                        NETWORK PROGRAMMING WITH SOCKETS
```

In this program we illustrate the use of Berkeley sockets for interprocess communication across the network. We show the communication between a server process and a client process.

Since many server processes may be running in a system, we identify the desired server process by a "port number". Standard server processes have a worldwide unique port number associated with it. For example, the port number of SMTP (the sendmail process) is 25. To see a list of server processes and their port numbers see the file /etc/services

In this program, we choose port number 6000 for our server process. Here we shall demonstrate TCP connections only. For details and for other types of connections see:

Unix Network Programming
-- W. Richard Stevens, Prentice Hall India.

To create a TCP server process, we first need to open a "socket" using the socket() system call. This is similar to opening a file, and returns a socket descriptor. The socket is then bound to the desired port number. After this the process waits to "accept" client connections.

*/

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

/* The following three files must be included for network programming */
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

                        /* THE SERVER PROCESS */

        /* Compile this program with cc server.c -o server
          and then execute it as ./server &
        */
main()
{
        int                     sockfd, newsockfd ; /* Socket descriptors */
        int                     clilen;
        struct sockaddr_in      cli_addr, serv_addr;

        int i;
        char buf[100];          /* We will use this buffer for communication */

        /* The following system call opens a socket. The first parameter
          indicates the family of the protocol to be followed. For internet
          protocols we use AF_INET. For TCP sockets the second parameter
          is SOCK_STREAM. The third parameter is set to 0 for user
```

```
  applications.
*/
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("Cannot create socket\n");
        exit(0);
}

/* The structure "sockaddr_in" is defined in <netinet/in.h> for the
  internet family of protocols. This has three main fields. The
  field "sin_family" specifies the family and is therefore AF_INET
  for the internet family. The field "sin_addr" specifies the
  internet address of the server. This field is set to INADDR_ANY
  for machines having a single IP address. The field "sin_port"
  specifies the port number of the server.
*/
serv_addr.sin_family            = AF_INET;
serv_addr.sin_addr.s_addr       = INADDR_ANY;
serv_addr.sin_port              = 6000;

/* With the information provided in serv_addr, we associate the server
  with its port using the bind() system call.
*/
if (bind(sockfd, (struct sockaddr *) &serv_addr,
                                    sizeof(serv_addr)) < 0) {
        printf("Unable to bind local address\n");
        exit(0);
}

listen(sockfd, 5); /* This specifies that up to 5 concurrent client
                      requests will be queued up while the system is
                      executing the "accept" system call below.
                   */

/* In this program we are illustrating a concurrent server -- one
  which forks to accept multiple client connections concurrently.
  As soon as the server accepts a connection from a client, it
  forks a child which communicates with the client, while the
  parent becomes free to accept a new connection. To facilitate
  this, the accept() system call returns a new socket descriptor
  which can be used by the child. The parent continues with the
  original socket descriptor.
*/
while (1) {

        /* The accept() system call accepts a client connection.
          It blocks the server until a client request comes.

          The accept() system call fills up the client's details
          in a struct sockaddr which is passed as a parameter.
          The length of the structure is noted in clilen. Note
          that the new socket descriptor returned by the accept()
```

```
                system call is stored in "newsockfd".
             */
             clilen = sizeof(cli_addr);
             newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                                &clilen) ;

             if (newsockfd < 0) {
                     printf("Accept error\n");
                     exit(0);
             }

             /* Having successfully accepted a client connection, the
                server now forks. The parent closes the new socket
                descriptor and loops back to accept the next connection.
             */
             if (fork() == 0) {

                     /* This child process will now communicate with the
                        client through the send() and recv() system calls.
                     */
                     close(sockfd);    /* Close the old socket since all
                                          communications will be through
                                          the new socket.
                                       */

                     /* We initialize the buffer, copy the message to it,
                        and send the message to the client.
                     */
                     for(i=0; i < 100; i++) buf[i] = '\0';
                     strcpy(buf,"Message from server");
                     send(newsockfd, buf, 100, 0);

                     /* We again initialize the buffer, and receive a
                        message from the client.
                     */
                     for(i=0; i < 100; i++) buf[i] = '\0';
                     recv(newsockfd, buf, 100, 0);
                     printf("%s\n", buf);

                     close(newsockfd);
                     exit(0);
             }

             close(newsockfd);
     }
}
```

```
/*                              THE CLIENT PROCESS

     Please read the file server.c before you read this file. To run this,
```

```
        you must first change the IP address specified in the line:

                serv_addr.sin_addr.s_addr = inet_addr("144.16.202.221");

        to the IP-address of the machine where you are running the server.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

main()
{
        int                     sockfd ;
        struct sockaddr_in      serv_addr;

        int i;
        char buf[100];

        /* Opening a socket is exactly similar to the server process */
        if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                printf("Unable to create socket\n");
                exit(0);
        }

        /* Recall that we specified INADDR_ANY when we specified the server
          address in the server. Since the client can run on a different
          machine, we must specify the IP address of the server.

          TO RUN THIS CLIENT, YOU MUST CHANGE THE IP ADDRESS SPECIFIED
          BELOW TO THE IP ADDRESS OF THE MACHINE WHERE YOU ARE RUNNING
          THE SERVER.
        */
        serv_addr.sin_family            = AF_INET;
        serv_addr.sin_addr.s_addr       = inet_addr("144.16.202.221");
        serv_addr.sin_port              = 6000;

        /* With the information specified in serv_addr, the connect()
          system call establishes a connection with the server process.
        */
        if ((connect(sockfd, (struct sockaddr *) &serv_addr,
                                                sizeof(serv_addr))) < 0) {
                printf("Unable to connect to server\n");
                exit(0);
        }

        /* After connection, the client can send or receive messages.
          However, please note that recv() will block when the
          server is not sending and vice versa. Similarly send() will
```

```
    block when the server is not receiving and vice versa. For
    non-blocking modes, refer to the online man pages.
  */
  for(i=0; i < 100; i++) buf[i] = '\0';
  recv(sockfd, buf, 100, 0);
  printf("%s\n", buf);

  for(i=0; i < 100; i++) buf[i] = '\0';
  strcpy(buf,"Message from client");
  send(sockfd, buf, 100, 0);

  close(sockfd);
}
```