Experiment #1
Multi-Processing Environment Under Unix/Linux
Process Management for Real-Time Applications

1. Prerequisite

ENCS 538, C programming language, basics of inter-process communications under Unix/Linux. Please refer to section 4 for a detailed list of prerequisites.

2. Introduction

The realization and control of embedded system needs some tools to manage multitasks and concurrent processes. This experiment aims to introduce some of these concepts using Linux as multi-tasking environment.

3. Objectives

- To learn how to create and manage processes on a multi-tasking environment.
- To learn how to establish inter-task communication between the different tasks.
- To learn and implement different control algorithms in embedded systems using software and programming in a multi-tasking environment.

4. List of prerequisites

Review the lecture notes that you have got during the ENCS538 course (Real-Time Applications and Embedded Systems). Mainly, you need to focus on the following items :

- Unix/Linux file system information.
- The C-language programming techniques.
- Process creation, process ID, parent process ID.
- Ending a process, waiting on a process.
- Systems calls: exec, execl, execlp, system, fork.
- Signal and signal management calls.
- Primitive communication between processes (lock files, signals), pipes and fifos.
- Better inter-process communication techniques: pipes, fifos, message queues, semaphores, shared memory.
- Basic Unix/Linux commands relative to process management (e.g. ls, ps, kill).

5. Background

5.1 Process Creation Concepts

Processes are the primitive units for allocation of system resources. Each process has its own **address space** and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized **hierarchically**. Each process has a **parent process**, which explicitly arranged to create it. The processes created by a given parent are called its **child processes**. A child inherits many of its attributes from the parent process.

A **process ID number** names each process. A unique process ID is allocated to each process when it is created. The lifetime of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.

Processes are created with the **fork()** system call (so the operation of creating a new process is sometimes called **forking a process**). The child process created by fork is a copy of the original parent process, except that it has its own process ID.

After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling **wait()** or **waitpid()**. These functions give you limited information about why the child terminated--for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the return value from fork to tell whether the program is running in the parent process or the child process.

When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action. A process that is waiting for its parent to accept its return code is called a **zombie process**. If a parent dies before its child, the child (**orphan process**) is automatically adopted by the original "**init**" process whose PID is **1**.

5.2 File Execution

A child process can execute another program using one of the **exec** functions (See Appendix A). The program that the process is executing is called its **process image**. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

- The process ID and the parent process ID.
- Session and process group membership.
- Real user ID and group ID, and supplementary group IDs.

- Current working directory and root directory. In the GNU system, the root directory is not copied when executing a setuid program; instead the system default root directory is used for the new program.
- File mode creation mask.
- Process signal mask.
- Pending signals.
- Elapsed processor time associated with the process; see section Processor Time.

5.3 Signals

Programs must sometimes deal with unexpected or unpredictable events, such as:

- a floating point error.
- a power failure.
- an alarm clock "ring".
- the death of a child process.
- a termination request from a user (i.e., a Control-C).
- a suspend request from a user (i.e., a Control-Z).

These kind of events are sometimes called ***interrupts***, as they must interrupt the regular flow of a program in order to be processed. When UNIX recognizes that such an event has occurred, it sends the corresponding process a ***signal***.

The kernel isn't the only one that can send a signal; any process can send any other process a signal, as long as it has permissions.

A programmer may arrange for a particular signal to be **ignored** or to be processed by a special piece of code called a **signal handler**. In the latter case, the process that receives the signal suspends its current flow of control, executes the signal handler, and then resumes the original flow of control when the signal handler finishes.

Signals inform processes of the occurrence of asynchronous events. *Every type of signal has a* **handler** *which is a function*. All signals have default handlers which may be replaced with user-defined handlers. The default signal handlers for each process usually terminate the process or ignore the signal, but this is not always the case.

Signals may be sent to a process from another process, from the **kernel**, or from **devices** such as terminals. The **^C, ^Z, ^S** and **^Q** terminal commands all generate **signals** which are sent to the **foreground process** when pressed.
The kernel handles the delivery of signals to a process. Signals are checked for whenever a process is being rescheduled, put to sleep, or re-executing in user mode after a system call.

5.4 Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.

2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls **fork**, and the pipe is used between the parent and the child.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one to the standard input of the next using a pipe.

A pipe is created by calling the **pipe** function.

```
#include <unistd.h>

int pipe(int filedes[2]); // Returns: 0 if OK, −1 on error
```

Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

Pipes are implemented using UNIX domain sockets in 4.3BSD, 4.4BSD, and Mac OS X 10.3. Even though UNIX domain sockets are full duplex by default, these operating systems hobble the sockets used with pipes so that they operate in half-duplex mode only.

POSIX.1 allows for an implementation to support full-duplex pipes. For these implementations, filedes[0] and filedes[1] are open for both reading and writing.

Two ways to picture a half-duplex pipe are shown in Figure 1 The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

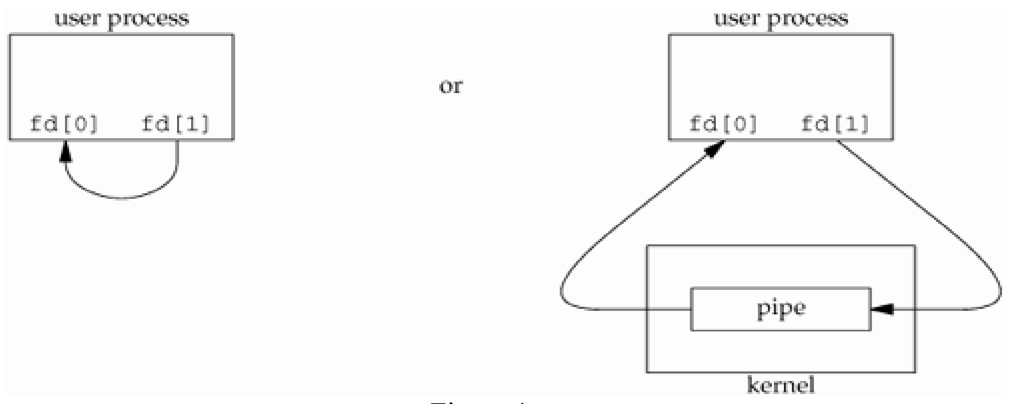
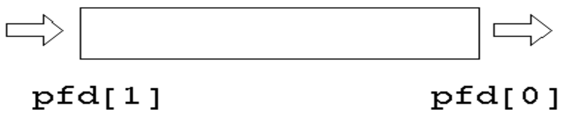


Figure 1

Fork and a pipe

A pipe in a single process is next to useless. Normally, the process that calls **pipe** then calls **fork**, creating an IPC channel from the parent to the child or vice versa. Figure 2 shows this scenario.

Before fork



After fork

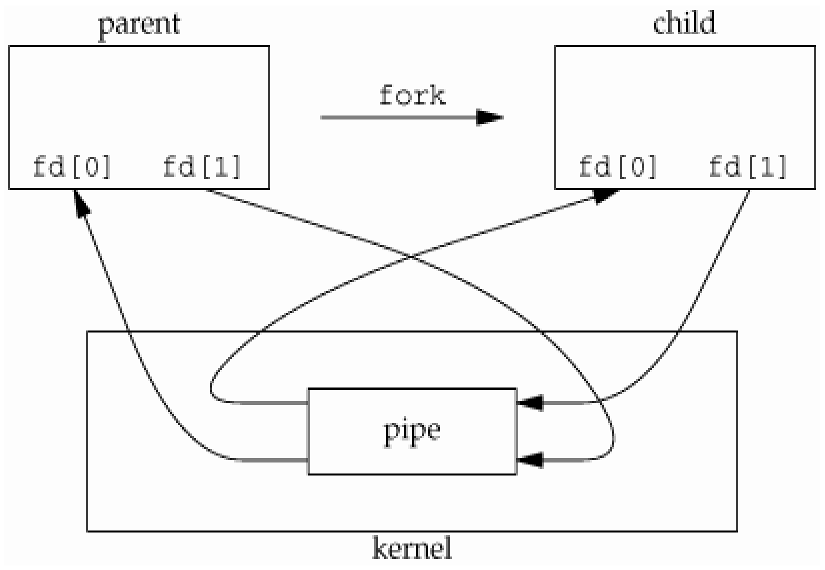


Figure 2

This gives *two* **read ends** and *two* **write ends**. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.
Either process can write into the pipe, and either can read from it. Which process will get what is not known.

Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end. Then it will become a simple pipeline again as shown in Figure 3.

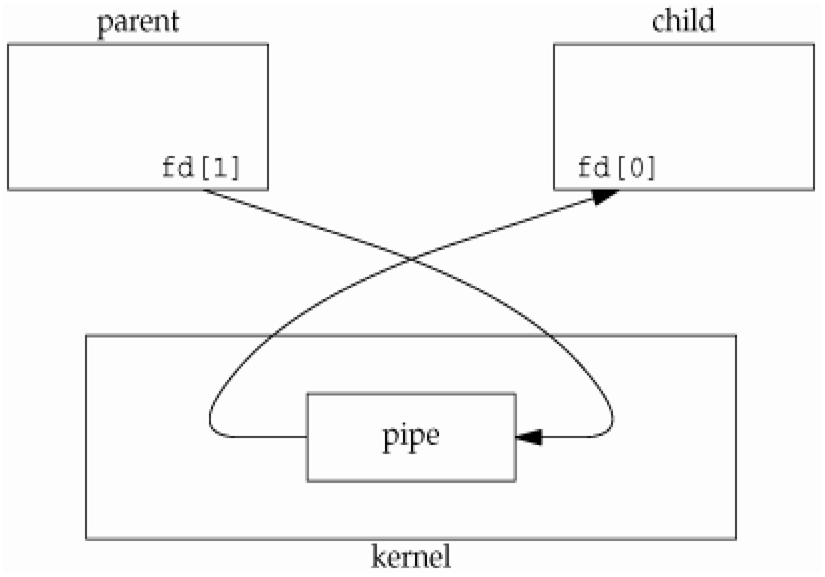


Figure 3

6. Procedure

6.1 Part A: Process creation and termination

1. Type the following program in a text editor. Name the file **create.c**, compile it using **gcc** to generate the executable create and run it. Notice the function **getpid()** which returns the process ID and the function **getppid()** which returns the parent process ID.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        printf("My PID is %d\n", getpid());
        printf("My parent PID is %d\n", getppid());

        while ( 1 );
}
```

2. Execute the command **ps -ef** and make sure the process is active. Notice that the process parent **ID** is the **PID** of the shell window where the process has been launched.
3. Terminate the process create using the command **kill PID**. Replace **PID** by the current process **ID**.
4. Execute the command **ps -ef** and make sure the process has ceased to exist.
5. Type the following program in a text editor. Name the file **fork1.c**, compile it using **gcc** to generate the executable **fork1** and run it in the background:

```
#include <stdio.h>
#include <unistd.h>

int main( )
{
      int i;
      /*
      * Generating 5 children processes
      */
      for ( i = 0; i < 5; i++ ) {
      if ( fork() == 0 ) {
         printf("Child: My PID is %d\n", getpid());
         while ( 1 );
      }
      else
         {
                 printf("In the PARENT process\n");

                 if ( i == 4 )
                 while ( 1 );
         }
        }
        return 0;
}
```

To run the program **fork1** in the background, do the following:

```
> ./fork1  &
```

6. Execute the command **ps -ef** and notice how many processes that hold the name **fork1** are active. Determine which one is the parent using the **PPID**.
7. Terminate the parent process **fork1** using the command **kill PID**. Replace **PID** by the current process ID. Notice the new **parent ID** for the forked children.
8. Terminate all **fork1** processes using the command **kill PID** as explained above.
9. Type the following program in a text editor. Name the file **exec1.c**, compile it using **gcc** to generate the executable **exec1** and run it.

```
/*
* Running the cat utility via an exec  system call
*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[ ])
{
     if ( argc != 2 ) {
     printf("Usage: %s fileName\n", argv[0]);
     exit(-1);
}

     execlp("/bin/cat", "cat", argv[1], (char *) NULL);
     perror("exec failure ");
     exit(-2);
}
```

10. Run the program exec1 by giving it any text file as an argument.
11. Explain the main difference between the fork and exec function calls.

TODO:

It will be given during the lab based on material covered in this section.

6.2 Part B: Inter-process communication

Signals

1. Execute **man sigset**, **man sighold**, **man sigrelse**, **man sigpause**, **sigignore** and check what these functions do.
2. Check the different signal types such as **SIGABRT** (6), **SIGALRM** (14), **SIGKILL** (9), **SIGQUIT** (3), **SIGINT** (2), **SIGUSR1** and **SIGUSR2**. Execute **kill -l** to check the other signal types.

3. Check the command **kill** that you can call from a terminal and the function **kill(pid, sig)** that can be called from within your C-file. Note that the function **kill(pid, sig)** is used to send a signal to a process or a group of processes.
4. Type the following program that catches the signals **SIGINT** and **SIGQUIT** and displays the signals numbers. Name the **file signals1.c.**

```
/*
* Catching signals with sigset
*/
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
        int      i;
        void signal_catcher(int);
        if ( sigset(SIGINT, signal_catcher) == SIG_ERR )
         {
                perror("Sigset cannot set SIGINT");
                exit(SIGINT);
         }
        if ( sigset(SIGQUIT, signal_catcher) == SIG_ERR )
        {
                perror("Sigset cannot set SIGQUIT");
                exit(SIGQUIT);
  }


        for ( i = 0; ; ++i )
        {
                printf("%i\n", i);
                sleep(1);
        }
}

void signal_catcher(int the_sig)
{
        printf("\nSignal %d received.\n", the_sig);

        if ( the_sig == SIGQUIT )
        exit(1);
}
```

5. Compile **signals1.c** using the **gcc** compiler to create the executable **signals1** and run it.
6. Note that the signals **SIGINT** and **SIGQUIT** can be sent to the **signals1** process by typing **Ctrl-C** and **Ctrl-\** respectively.

7. Type the following program that catches **signal SIGUSR2** and **holds SIGUSR1**. Name the file **signals2.c**.

```
/*
* Catching signals with sigset
*/
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main(void)
{
    void sigset_catcher(int);

    sighold(SIGUSR1);
    sigset(SIGUSR2, sigset_catcher);
    printf("Waiting for signal\n");
    pause();
    printf("Done\n");
    exit(0);
}

void sigset_catcher(int n)
{
    printf("\nReceived signal %d will release SIGUSR1\n", n);
    sigrelse(SIGUSR1);
    printf("SIGUSR1 released!\n");
}
```

8. Compile **signals2.c** using the **gcc** compiler to create the executable **signals2** and run it.
9. In a different terminal, execute the command **kill -USR2 pid** and check the process output. Replace **pid** by the actual process **pid**.
10. As a remark, note that the signals **SIGKILL** and **SIGSTOP** cannot be ignored or held.
11. Type the following program and name the file **alarm.c**. Compile **alarm**.c to create the executable alarm and run it.

```
/*
* Testing the alarm signal (SIGALRM)
*/
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int val = 0;

int main(void)
{
        int     i;
        void  signal_catcher(int);

        alarm(1);

        if ( sigset(SIGALRM,  signal_catcher)  == SIG_ERR  )
        {
                perror("Sigset  cannot  set  SIGALRM");
                exit(SIGINT);
        }

        while  ( 1  );
}

void  signal_catcher(int  the_sig)
{
        val++;
        printf("val  = %d\n",  val);
        alarm(1);
}
```

12. Check that you have created a timer that is being updated once every second.

TODO:

It will be given during the lab based on material covered in this section.

Pipes

1. Execute **man pipe**, **man popen**, **man pclose**, **man read**, **man write**, **man close** and check what these functions do.
2. Execute the following command:

   ```
   > ps -ef | grep  anyProcess
   ```

   Where **anyProcess** represents any process currently active on your system. Notice the output that you get when executing the above command and compare it to the output you get when executing the command:

   ```
   > ps -ef
   ```

   Note that the symbol **|** represents a pipe. Explain briefly how does it function.

3. Type the following program that forks a parent and a child processes and establishes a communication channel through a **pipe**. Name the file **pipe1.c**:

   ```
   /*
   *  Using  a  pipe  to  send data  from  parent  to  a  child
   */

   #include <stdio.h>
   #include <unistd.h>
   ```

```
#include <stdlib.h>
#include <string.h>

int  main(int  argc,  char *argv[])
{
          int        f_des[2];
          static  char  message[BUFSIZ];

          if  ( argc  !=  2  )
          {
                    fprintf(stderr,  "Usage:  %s  message\n",  *argv);
                    exit(1);
          }

          if ( pipe(f_des)  == -1  )
          {
                    perror("Pipe");
                    exit(2);
          }

          switch  ( fork()  )
          {
                    case -1: perror("Fork"); exit(3);

                    case 0:  /*  In  the  child  */
                    close(f_des[1]);
                    if ( read(f_des[0],  message,  BUFSIZ)  !=  -1  )
                    {
                              printf("Message  received  by  child:  [%s]\n",  message);
                              fflush(stdout);
                    }
                    else
                    {
                               perror("Read");
                               exit(4);
                     }

                    break;

          default:  /*  In  the  parent  */
                    close(f_des[0]);
                    if ( write(f_des[1],  argv[1],  strlen(argv[1]))  !=  -1  )
                    {
                              printf("Message  sent  by  parent:  [%s]\n",  argv[1]);
                              fflush(stdout);
                    }
                    else
                     {
                     perror("Write"); exit(5);
                    }
          }
          exit(0);
}
```

4. Compile **pipe1.c** using the **gcc** compiler to create the executable **pipe1** and run it.
5. Explain how the communication between the parent and child processes took place.
6. Type the following program that makes use of the function **popen**. Name the file **popen1.c**:

```
/*
* Using the popen and pclose I/O commands
*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>

int main(int argc, char *argv[])
{
        FILE  *fin,*fout;
        char    buffer[PIPE_BUF];
        int       n;

        if ( argc < 3 )
        {
                fprintf(stderr, "Usage: %s cmd1 cmd2\n", argv[0]);
                exit(1);
        }

                fin= popen(argv[1], "r");
                fout = popen(argv[2], "w");

        while ( (n = read(fileno(fin), buffer, PIPE_BUF)) > 0 )

        write(fileno(fout), buffer, n);

        pclose(fin);
        pclose(fout);
        exit(0);
}
```

7. Compile **popen1.c** using the **gcc** compiler to create the executable **popen1**. Execute the following command:

   > **popen1 "cat popen1.c" "grep include"**

   Explain the output that you get.

8. Execute the command:

   > **cat popen1.c | grep include**

   Compare the output that you get in steps 7 and 8 above.

TODO:

It will be given during the lab based on material covered in this section.

Appendix A. Process Creation and Execution

A.1 Program compilation and Process management

- To compile a program , use **gcc** command to generate the executable file. See the following example:
  > gcc example.c -o example

- To use a debugger (like the ddd debugger), you need to tell the compiler **gcc** to create the symbols table for the program. You do that by using the option **-g** with the compiler as follows:
  > gcc -g example.c -o example.

- To run the program, you call the executable file as follows:
  > ./ example

- To run the executable file using debugger, execute the following command:
  > ddd example

A.2 Process Identification:

The **pid_t** data type represents process IDs which is basically a signed integer type **(int).** You can get the process **ID** of a process by calling **getpid**(). The function **getppid()** returns the process ID of the parent of the current process (this is also known as the parent process ID). Your program should include the header files ‘**unistd.h**’ and ‘**sys/types.h**’ to use these functions.

- Function: pid_t getpid (void)
  The getpid() function returns the process ID of the current process.

- Function: pid_t getppid (void)
  The getppid() function returns the process ID of the parent of the current process.

A.3 Process Completion:

The functions described in this section are used to **wait** for a child process to terminate or stop, and determine its status. These functions are declared in the header file "**sys/wait.h**".

- Function: pid_t wait (int *status_ptr)
  wait() will force a parent process to wait for a child process to stop or terminate. **wait**() return the **pid** of the child or **-1** for an error. The exit status of the child is returned to **status_ptr**.

- Function: void exit (int status)

  exit() terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the status value.

By convention, a **status of 0** means *normal* termination. Any other value indicates an *error* or *unusual* occurrence. Many standard library calls have errors defined in the **sys/stat.h** header file. We can easily derive our own conventions.

If the child process must be guaranteed to execute before the parent continues, the **wait** system call is used. A call to this function causes the parent process to wait until one of its child processes exits. The **wait** call returns the **process id** of the child process, which gives the parent the ability to wait for a particular child process to finish.

A process may suspend for a period of time using the sleep command

- Function: unsigned int sleep (seconds)

A.4 File Execution:

A child process can execute another program using one of the **exec** functions. The program that the process is executing is called its **process image**. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

This section describes the exec family of functions, for executing a file as a process image. You can use these functions to make a child process execute a new program after it has been forked.

The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file "**unistd.h**".

- Function: int   execv (const char *filename, char *const argv[ ])

The **execv()** function executes the file named by filename as a new process image. The **argv** argument is an *array of null-terminated strings* that is used to provide a value for the **argv** argument to the main function of the program to be executed. The last element of this array must **be a null pointer**. By convention, the first element of this array is the file name of the program sans directory names.

The environment for the new process image is taken from the environ variable of the current process image.

- Function: int   execl (const char *filename, const char *arg0, ...)

This is similar to **execv**, but the **argv** strings are specified individually instead of as an array. *A null pointer must be passed as the last such argument*.

- Function: int   execvp (const char *filename, char *const argv[ ])

The **execvp** function is similar to **execv**, except that it searches the directories listed in the **PATH** environment variable to find the full file name of a file from filename if filename does not contain a slash.

This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen. Shells use it to run the commands that user's type.

- Function: int   execlp (const char *filename, const char *arg0, ...)

This function is like **execl**, except that it performs the same file name searching as the **execvp** function.

Appendix B. Signals

B.1 Types Of Signals:

There are 31 different signals defined in "/usr/include/signal.h". A programmer may choose for a particular signal to trigger a user-supplied signal handler, trigger the default kernel-supplied handler, or be ignored.

Some signals are widely used, while others are extremely obscure and used by only one or two programs. The following list gives a brief explanation of each signal:

SIGHUP
Hangup. Sent when a terminal is hung up to every process for which it is the control terminal. Also sent to each process in a process group when the group leader terminates for any reason. This simulates hanging up on terminals that can't be physically hung up, such as a personal computer.

SIGINT
Interrupt. Sent to every process associated with a control terminal when the interrupt key **(Control-C)** is hit. This action of the interrupt key may be suppressed or the interrupt key may be changed using the **stty** command. Note that suppressing the interrupt key is completely different from ignoring the signal, although the effect (or lack of it) on the process is the same.

SIGTSTP
Interrupt. Sent to every process associated with a control terminal when the interrupt key **(Control-Z)** is hit. This action of the interrupt key may be suppressed or the interrupt key may be changed using the **stty** command. Note that suppressing the interrupt key is completely different from ignoring the signal, although the effect (or lack of it) on the process is the same.

SIGQUIT
Quit. Similar to SIGINT, but sent when the quit key (normally Control-\) is hit. Commonly sent in order to get a core dump.

SIGILL
Illegal instruction. Sent when the hardware detects an illegal instruction. Sometimes a process using floating point aborts with this signal when it is accidentally linked without the **-f** option on the cc command. Since C programs are in general unable to modify their instructions, this signal rarely indicates a genuine program bug.

SIGTRAP
Trace trap. Sent after every instruction when a process is run with tracing turned on with **ptrace**.

SIGIOT
I/O trap instruction. Sent when a hardware fault occurs, the exact nature of which is up to the implementer and is machine-dependent. In practice, this signal is preempted by the standard subroutine abort, which a process calls to commit suicide in a way that will produce a core dump.

SIGEMT
Emulator trap instruction. Sent when an implementation-dependent hardware fault occurs. Extremely rare.

SIGFPE
Floating-point exception. Sent when the hardware detects a floating-point error, such as a floating point number with an illegal format. Almost always indicates a program bug.

SIGKILL
Kill. The one and only sure way to kill a process, since this signal is always fatal (can't be ignored or caught). To be used only in emergencies; **SIGTERM** is preferred.

SIGBUS
Bus error. Sent when an implementation-dependent hardware fault occurs. Usually means that the process referenced at an odd address data that should have been word-aligned.

SIGSEGV
Segmentation violation. Sent when an implementation-dependent hardware fault occurs. Usually means that the process referenced data outside its address space. Trying to use NULL pointers will usually give you a SIGSEGV.

SIGPIPE
Write on a pipe not opened for reading. Sent to a process when it writes on a pipe that has no reader. Usually this means that the reader was another process that terminated abnormally. This signal acts to terminate all processes in a pipeline: When a process terminates abnormally, all processes to its right receive an end-of-file and all processes to its left receive this signal. Note that the standard shell (**sh**) makes each process in a pipeline the parent of the process to its left. Hence, the writer is not the reader's parent (it's the other way around), and would otherwise not be notified of the reader's death.

SIGALRM
Alarm clock. Sent when a process's alarm clock goes off. The alarm clock is set with the alarm system call.

SIGTERM
Software termination. The standard termination signal. It's the default signal sent by the kill command, and is also used during system shutdown to terminate all active processes. A program should be coded to either let this signal default or else to clean up quickly (e.g., remove temporary files) and call exit.

SIGUSR1
User defined signal 1. This signal may be used by application programs for inter-process communication. This is not recommended however, and consequently this signal is rarely used.

SIGUSR2
User defined signal 2. Similar to SIGUSR1.

SIGPWR
Power-fail restart. Exact meaning is implementation-dependent. One possibility is for it to be sent when power is about to fail (voltage has passed, say, 200 volts and is falling). The process has a very brief time to execute. It should normally clean up and exit (as with SIGTERM). If the process wishes to survive the failure (which might only be a momentary voltage drop), it can clean up and then sleep for a few seconds. If it wakes up it can assume that the disaster was only a dream and resume processing. If it doesn't wake up, no further action is necessary.