Interprocess Communications

- It is often desirable to construct systems that consist of several cooperating processes rather than just a single, monolithic program.
- There are several possible reasons for this: a single program might, for example, be too large for the machine it is running on, either in terms of physical memory or available address space, etc.
- UNIX is rich in interprocess communication mechanisms. We will discuss the three most widely used of these facilities: **signals**, **pipes** and **FIFO**s.

Signals

- Signals are notifications sent to a process in order to notify it of events of interest. By their nature, they interrupt whatever the process is doing force it to handle them immediately.
- In that sense signals represent a very limited form of interprocess communication.
- Suppose you are running a UNIX command that seems likely to take a long time:

  $ gcc verybigprog.c

- Then you realize that something is wrong, and the command will eventually fail. To save time, you stop the command by hitting the current interrupt key, which is often **DEL** or **CTRL-C.**
- The command will be terminated and you will be returned to the shell prompt.
- What actually happens is this: the part of the kernel responsible for keyboard input sees the interrupt character. The kernel then sends a signal called **SIGINT** to all processes that recognize the terminal as their control terminal. This includes the invocation of ***gcc***.
- When ***gcc*** receives the signal, it performs the default action associated with **SIGINT** and terminates.
- Programs can also elect to 'catch' **SIGINT**, which means they perform a special interrupt routine whenever the user presses interrupt.
- For example, suppose a program file has become corrupted in some way and contains illegal machine instructions.
- When a user executes the program, the kernel will detect the attempt to execute an illegal instruction and send the process the signal **SIGILL** (the ILL here stands for 'illegal') to terminate it.
- The resulting dialogue could look like:

  $ badprog
  Illegal instruction - core dumped

- As well as being sent from the kernel to a process, signals can be sent from process to process. This is easiest to show with the kill command.

- For example, suppose a programmer starts off a long running command in the background and then decides to terminate it.:

  $ cc verybigprog.c&
  1098

- ***kill*** can be used to send the signal **SIGTERM** to the process. Like **SIGINT**, **SIGTERM** will terminate a process. The process-id must be given to the kill command as an argument:

  $ kill 1098
  1098 terminated

- Signals then, provide a simple method for transmitting software interrupts to UNIX processes.

- Because of their nature, signals tend to be used for handling abnormal conditions rather than the straightforward transmission of data between processes.

- There are three actions that can take place when a signal is delivered to a process: it can be ignored; or the process can be terminated (with or without core dumping); or a handler function can be called.

Signal names

- Signal names are defined in the standard header file **signal.h**. The following is a partial list of standard signals:

- **SIGHUP**
  - **Hangup**. This is sent by the kernel to all processes attached to a control terminal when that terminal is disconnected.

- **SIGINT**
  - **Interrupt**. Sent by the kernel to all processes associated with a terminal when a user hits the interrupt key.

- **SIGQUIT**
  - **Quit**. Similar to **SIGINT**, this is sent by the kernel when the user hits the quit key associated with the terminal. The default value for the quit key is ASCII FS or **CTRL-**.
  - Unlike **SIGINT**, **SIGQUIT** will result in what is described as abnormal termination.

- **SIGILL**
  - **Illegal instruction**. This is sent by the operating system when a process attempts to execute an illegal instruction. Results in abnormal termination.

- **SIGTRAP**
    - **Trace trap**. Special signal used by debuggers.
- **SIGFPE**
    - **Floating-point exception**. This is sent by the kernel when a floating-point error occurs. Causes abnormal termination.
- **SIGKILL**
    - **Kill**. This is a rather special signal that is sent from one process to another to terminate the receiver. Since all other signals may be ignored or caught, it is important that there exists one signal that is guaranteed to terminate a process "in extremis". This is it.
- **SIGSYS**
    - **Bad argument to a system call**. Sent by the kernel when a process passes an invalid argument to a system call that cannot be handled via the normal system call error recovery mechanisms.
- **SIGPIPE**
    - Write on a pipe with no one to read it.
- **SIGALARM**
    - **Alarm clock**. This is normally sent by the kernel to a process after a timer has expired. The timer is usually set by the ***alarm*** system call.
- **SIGTERM**
    - Software termination signal. This signal is provided by the system for use by ordinary programs.
- **SIGUSR1** and **SIGUSR2**
    - Like **SIGTERM**, these are never sent by the kernel and may be used for whatever purpose a user wishes.

Handling Signals With The *signal()* System Call

- All of the signals will, by default, terminate the receiving process.
- For simple programs this perfectly adequate. It allows the user to stop a program that goes astray by typing one of the interrupt or quit keys.
- However, in larger programs, unexpected signals can cause problems. It makes no sense, for example, to allow a program to be halted by a careless press of an interrupt key during an important data transfer.
- Fortunately, a process can choose the way it responds when it receives a particular signal with the signal call, which is used to associate a particular action with a named signal.
- The ***signal()*** function is used as follows:

```
#include <stdio.h>
#include <signal.h>    // signal name macros, and the signal() prototype

// The signal handler
void catch_int(int sig_num)
{
    //re-set the signal handler again to catch_int, for next time
    signal(SIGINT, catch_int);

    // and print the message
    printf("Don't do that!");
    fflush(stdout);
}
    .
    .
    .


    // set the INT (Ctrl-C) signal handler to 'catch_int'
    signal (SIGINT, catch_int);
```

- The first parameter identifies the signal in question. It can be set to any of the signal names defined previously.

- ***signal()***'s second parameter ***catch_int*** describes the action to be associated with the signal. Generally It can take three values:

    1. The address of a function returning int. ***catch_int*** will be executed when the specified signal is received. Control will be passed to **catch_int** as soon as the process receives the signal, whatever part of the program it is executing.

       When **catch_int** returns, control will be passed back to the point at which the process was interrupted.

    2. **SIG_IGN** A special symbolic name, which simply means 'ignore this signal'. In future, the process will do just that.

    3. **SIG_DFL** Another symbolic name, which restores the system's default action.

<u>Example 1: catching SIGINT</u>

- This example shows how a signal can be caught and also sheds more light on the underlying signal mechanism.

- The program simply associates a function called ***catch_int*** with **SIGINT**, then executes a series of ***sleep*** and ***printf*** statements.

- If left alone ***sigex*** produces the following output:

    $ sigex
    sleep call #1
    sleep call #2
    sleep call #3
    sleep call #4
    Exiting

- The user can, however, interrupt the progress of sigex by typing the interrupt key:

    $ sigex
    sleep call #1
    ctrl-c (user presses interrupt key)

    CATCHINT: signo=2
    CATCHINT: returning

    sleep call #2
    sleep call #3
    sleep call #4
    Exiting

Example 2: ignoring SIGINT

- Suppose we want a process to ignore the interrupt signal SIGINT. All we need to do is include the following line in the program:

  signal (SIGINT, SIG_IGN);

- After this is executed the interrupt key will be ineffective. It can be enabled again with

  signal (SIGINT, SIG_DFL);

- Sending signals with ***kill()***
- In practice, ***kill*** is used for one of these purposes:
  - To terminate one or more processes, usually with **SIGTERM**, but sometimes with **SIGQUIT** or **SIGIOT** so that a core dump will be obtained.
  - To test the error-handling code of a new program by simulating signals such as **SIGFPE** (floating-point exception).
- Usage

```
#include <signal.h>

int pid, sig, retval;
        .
        .
retval = kill (pid, sig);
```

- The first parameter **pid** determines the processes, or process, to which the signal sig will be sent. Normally pid will be a positive integer and in this case it will be taken to be an actual process-id.
- So the statement,

  kill (1234, SIGTERM);

  means send the signal SIGTERM to the process with process-id 1234, i.e., terminate that process.
- The ***pid*** parameter to kill can take other values which have special meanings:
- If ***pid*** is zero, the signal will be sent to all processes that belong to the same process group as the sender. This includes the sender.