IPC:Interrupts and Signals: <signal.h>

In this section will look at ways in which two processes can communicate. When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Signals are software generated interrupts that are sent to a process when a event happens. Signals can be synchronously generated by an error in an application, such as `SIGFPE` and `SIGSEGV`, but most signals are asynchronous. Signals can be posted to a process when the system detects a software event, such as a user entering an interrupt or stop or a kill request from another process. Signals can also be come directly from the OS kernel when a hardware event such as a bus error or an illegal instruction is encountered. The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future. Most signals cause termination of the receiving process if no action is taken by the process in response to the signal. Some signals stop the receiving process and other signals can be ignored. Each signal has a default action which is one of the following:

- The signal is discarded after being received
- The process is terminated after the signal is received
- A core file is written, then the process is terminated
- Stop the process after the signal is received

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

Macros are defined in `<signal.h>` header file for common signals.

These include:

SIGHUP 1 /* hangup */
SIGINT 2 /* interrupt */
SIGQUIT 3 /* quit */
SIGILL 4 /* illegal instruction */
SIGABRT 6 /* used by abort */
SIGKILL 9 /* hard kill */
SIGALRM 14 /* alarm clock */
SIGCONT 19 /* continue a stopped process */
SIGCHLD 20 /* to parent on child stop or exit */

Signals can be numbered from 0 to 31.

Sending Signals -- `kill(), raise()`

There are two common functions used to send signals

`int kill(int pid, int signal)` - a system call that send a `signal` to a process, `pid`. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes.

`kill()` returns 0 for a successful call, -1 otherwise and sets `errno` accordingly.

`int raise(int sig)` sends the signal sig to the executing program. `raise()` actually uses `kill()` to send the signal to the executing program:

```
kill(getpid(), sig);
```

There is also a UNIX command called kill that can be used to send signals from the command line - see `man` pages.

NOTE: that unless caught or ignored, the `kill` signal terminates the process. Therefore protection is built into the system.

Only processes with certain access privileges can be killed off.

Basic rule: *only processes that have the same user can send/receive messages*.

The `SIGKILL` signal cannot be caught or ignored and will always terminate a process.

For example`kill(getpid(),SIGINT);` would send the interrupt signal to the id of the calling process.

This would have a similar effect to `exit()` command. Also `ctrl-c` typed from the command sends a `SIGINT` to the process currently being.

`unsigned int alarm(unsigned int seconds)` -- sends the signal `SIGALRM` to the invoking process after seconds seconds.

Signal Handling -- `signal()`

An application program can specify a function called a signal handler to be invoked when a specific signal is received. When a signal handler is invoked on receipt of a signal, it is said to catch the signal. A process can deal with a signal in one of the following ways:

- The process can let the default action happen
- The process can block the signal (some signals cannot be ignored)
- the process can catch the signal with a handler.

Signal handlers usually execute on the current stack of the process. This lets the signal handler return to the point that execution was interrupted in the process. This can be changed on a per-signal basis so that a signal handler executes on a special stack. If a process must resume in a different context than the interrupted one, it must restore the previous context itself

Receiving signals is straighforward with the function:

`int (*signal(int sig, void (*func)()))()` -- that is to say the function `signal()` will call the `func` functions if the process receives a signal `sig`. Signal returns a pointer to function `func` if successful or it returns an error to `errno` and -1 otherwise.

`func()` can have three values:

`SIG_DFL`
: -- a pointer to a system default function `SID_DFL()`, which will terminate the process upon receipt of `sig`.

`SIG_IGN`
: -- a pointer to system ignore function `SIG_IGN()` which will disregard the `sig` action (UNLESS it is `SIGKILL`).

A function address
: -- a user specified function.

`SIG_DFL and SIG_IGN` are defined in `signal.h` (standard library) header file.

Thus to ignore a `ctrl-c` command from the command line. we could do:

```
signal(SIGINT, SIG_IGN);
```

TO reset system so that `SIGINT` causes a termination at any place in our program, we would do:

```
signal(SIGINT, SIG_DFL);
```

So lets write a program to trap a `ctrl-c` but not quit on this signal. We have a function `sigproc()` that is executed when we trap a `ctrl-c`. We will also set another function to quit the program if it traps the `SIGQUIT` signal so we can terminate our program:

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

void sigproc(void);

void quitproc(void);

main()
{               signal(SIGINT, sigproc);
                signal(SIGQUIT, quitproc);
                printf("ctrl-c disabled use ctrl-\\ to
quit\n");
                for(;;); /* infinite loop */}

void sigproc()
{               signal(SIGINT, sigproc); /*  */
                printf("you have pressed ctrl-c \n");}
```

```
void quitproc()
{                  printf("ctrl-\\ pressed to quit\n");
               exit(0); /* normal exit status */
}
```

`sig_talk.c` -- complete example program

Let us now write a program that communicates between child and parent processes using `kill()` and `signal()`.

`fork()` creates the child process from the parent. The `pid` can be checked to decide whether it is the child (== 0) or the parent (pid = child process id).

The parent can then send messages to child using the pid and `kill()`.

The child picks up these signals with `signal()` and calls appropriate functions.

An example of communicating process using signals is `sig_talk.c`:

```
/* sig_talk.c --- Example of how 2 processes can talk */
/* to each other using kill() and signal() */
/* We will fork() 2 process and let the parent send a few */
/* signals to it`s child  */

/* cc sig_talk.c -o sig_talk  */

#include <stdio.h>
#include <signal.h>

void sighup(); /* routines child will call upon
sigtrap */
void sigint();
void sigquit();

main()
{ int pid;

  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

   if (pid == 0)
```

```
     { /* child */
       signal(SIGHUP,sighup); /* set function calls
*/
       signal(SIGINT,sigint);
       signal(SIGQUIT, sigquit);
       for(;;); /* loop for ever */
     }
  else /* parent */
     {  /* pid hold id of child */
       printf("\nPARENT: sending SIGHUP\n\n");
       kill(pid,SIGHUP);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGINT\n\n");
       kill(pid,SIGINT);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGQUIT\n\n");
       kill(pid,SIGQUIT);
       sleep(3);
     }
}

void sighup()

{  signal(SIGHUP,sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

void sigint()

{  signal(SIGINT,sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

void sigquit()

{ printf("My DADDY has Killed me!!!\n");
  exit(0);
}
```

Other signal functions

There are a few other functions defined in `signal.h`:

`int sighold(int sig)` -- adds `sig` to the calling process's signal mask

`int sigrelse(int sig)` -- removes `sig` from the calling process's signal mask

`int sigignore(int sig)` -- sets the disposition of `sig` to `SIG_IGN`

`int sigpause(int sig)` -- removes `sig` from the calling process's signal mask and suspends the calling process until a signal is received

Zombie Processes

If a child process terminates while its parent is calling a wait function, the child process vanishes and its termination status is passed to its parent via the wait call. But what happens when a child process terminates and the parent is not calling wait ? Does it simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, is becomes a zombie process.

A zombie process is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children.The wait functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then calls wait . If the child process has not terminated at that point, the parent process will block in the wait call until the child process finishes. If the child process finishes before the parent process calls wait , the child process becomes a zombie.When the parent process calls wait , the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.
What happens if the parent does not clean up its children? They stay around in the system, as zombie processes.The program in Listing 3.6 forks a child process, which terminates immediately and then goes to sleep for a minute, without ever cleaning up the child process.

(zombie.c) Making a Zombie Process

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
pid_t child_pid;
/* Create a child process. */
child_pid = fork ();
if (child_pid > 0) {
/* This is the parent process. Sleep for a minute. */
sleep (60);
}
else {
/* This is the child process. Exit immediately. */
exit (0);
}
return 0;
}
```

Try compiling this file to an executable named make-zombie . Run it, and while it's still running, list the processes on the system by invoking the following command in another window:
% ps -e -o pid,ppid,stat,cmd 3.4

Process Termination

This lists the process ID, parent process ID, process status, and process command line. Observe that, in addition to the parent make-zombie process, there is another make-zombie process listed. It’s the child process; note that its parent process ID is the process ID of the main make-zombie process.The child process is marked as <defunct> , and its status code is Z, for zombie.

What happens when the main make-zombie program ends when the parent process exits, without ever calling wait ? Does the zombie process stay around? No—try running ps again, and note that both of the make-zombie processes are gone.When a program exits, its children are inherited by a special process, the init program, which always runs with process ID of 1 (it’s the first process started when Linux boots).