SIGNALS

Introduction to Signals

Suppose a program is running in a while(1) loop and you press
Ctrl+C key. The program dies. How does this happens?

- User presses Ctrl+C
- The tty driver receives character, which matches intr
- The tty driver calls signal system
- The signal system sends SIGINT(2) to the process
- Process receives SIGINT(2)
- Process dies

Actually by pressing <ctrl+c>, you ask the kernel to send SIGINT
to the currently running foreground process. To change the key
combination you can use stty(1) or tcsetattr(2) to replace
the current intr control character with some other key combination

Signal is a software interrupt delivered to a process by OS because:

The process did something (SIGFPE (8), SIGSEGV (11), SIGILL (4))

The user did something (SIGINT (2), SIGQUIT (3), SIGTSTP (20))

One process wants to tell another process something (SIGCHILD (17))

Signals are usually used by OS to notify processes that some event
has occurred, without these processes needing to poll for the event

Whenever a process receives a signal, it is interrupted from whatever it
is doing and forced to execute a piece of code called signal handler.
When the signal handler function returns, the process continues
execution as if this interruption has never occurred

A signal handler is a function that gets called when a process receives
a signal. Every signal may have a specific handler associated with it. A
signal handler is called in asynchronous mode. Failing to handle
various signals, would likely cause our application to terminate, when it
receives such signals

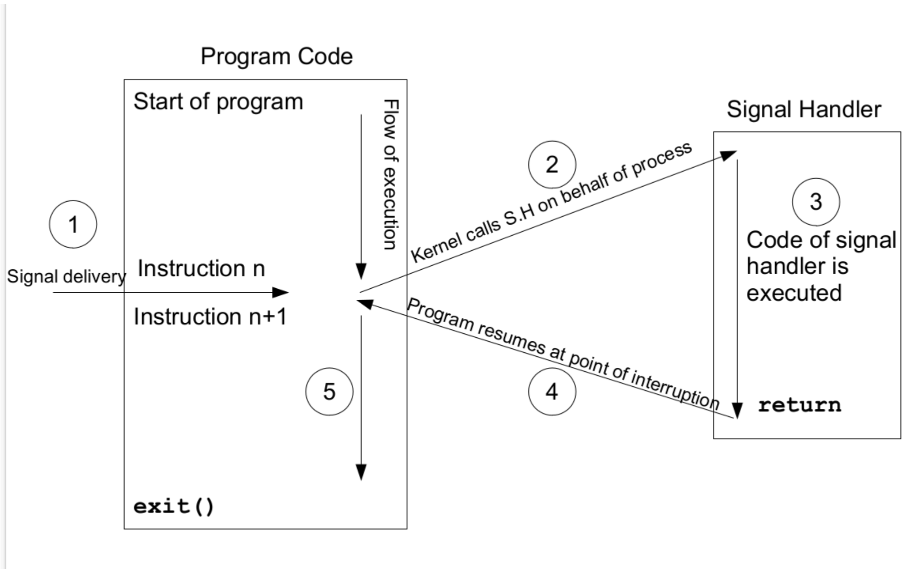
Synchronous & Asynchronous Signals

Signals may be generated synchronously or asynchronously

Synchronous signals pertains to a specific action in the program and is delivered (unless blocked) during that action.
Examples:
- Most errors generate signals synchronously
- Explicit request by a process to generate a signal for the same process

Asynchronous signals are generated by the events outside the control of the process that receives them. These signals arrive at unpredictable times during execution.
Examples include:
- External events generate requests asynchronously
- Explicit request by a process to generate a signal for some other process

Signal Delivery and Handler Execution



Signal Numbers and Strings

Every signal has a symbolic name and an integer value associated with it, defined in /usr/include/asm-generic/signal.h

You can use following shell command to list down the signals on your system:

$ kill -l   =>(32 standard and 32 real time signals)

Linux supports 32 real time signals from SIGRTMIN (32) to SIGRTMAX (63). Unlike standard signals, real time signals have no predefined meanings, are used for application defined purposes. The default action for an un-handled real time signal is to terminate the receiving process. See also $ man 7 signal

we can write Handlers for all these signals except 2 which
are called non cachable signals

1.SIGKILL
2.SIGSTOP

Sending Signals to Processes

A signal can be issued in one of the following ways:

1.Using Key board

<Ctrl+c> gives SIGINT(2)
<Ctrl+\> gives SIGQUIT(3)
<Ctrl+z> gives SIGTSTP(20)

2.Using Shell command

```
kill -<signal> <PID>          OR           kill -<signal> %<jobID>
```

If no signal name or number is specified then default is to send
SIGTERM(15) to the process

Do visit man pages for jobs, ps, bg and fg commands

bg gives SIGTSTP(20) while fg gives SIGCONT(18)

Signal Disposition

Upon delivery of a signal, a process carries out one of the following
default actions, depending on the signal: [$man 7 signal]

1. The signal is ignored; that is, it is discarded by the kernel and has no
effect on the process. (The process never even knows that it occurred)

2. The process is terminated (killed). This is sometimes referred to as
abnormal process termination, as opposed to the normal process
termination that occurs when a process terminates using exit()

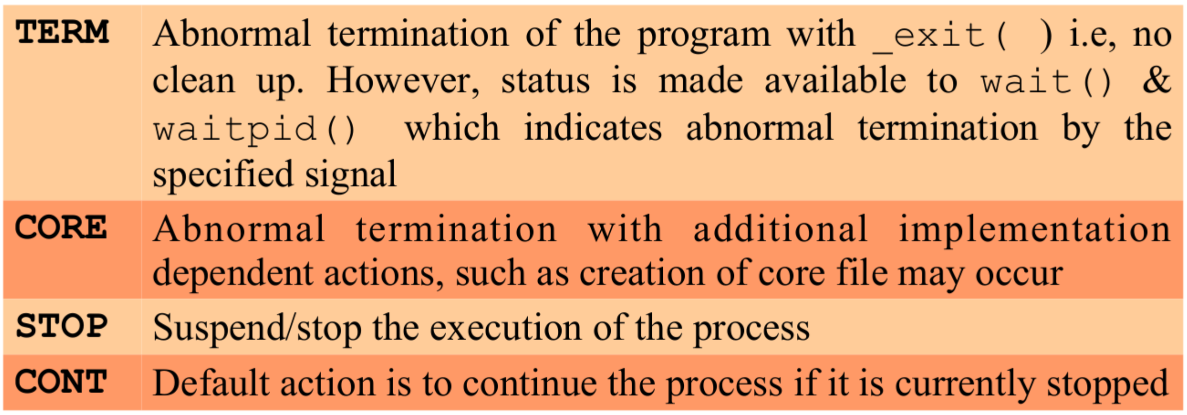
3. A core dump file is generated, and the process is terminated. A core
dump file contains an image of the virtual memory of the process,
which can be loaded into a debugger in order to inspect the state of the
process at the time that it terminated

4. The process is stopped—execution of the process is suspended
(SIGSTOP, SIGTSTP)

5. Execution of the process is resumed which was previously stopped
(SIGCONT, SIGCHLD)

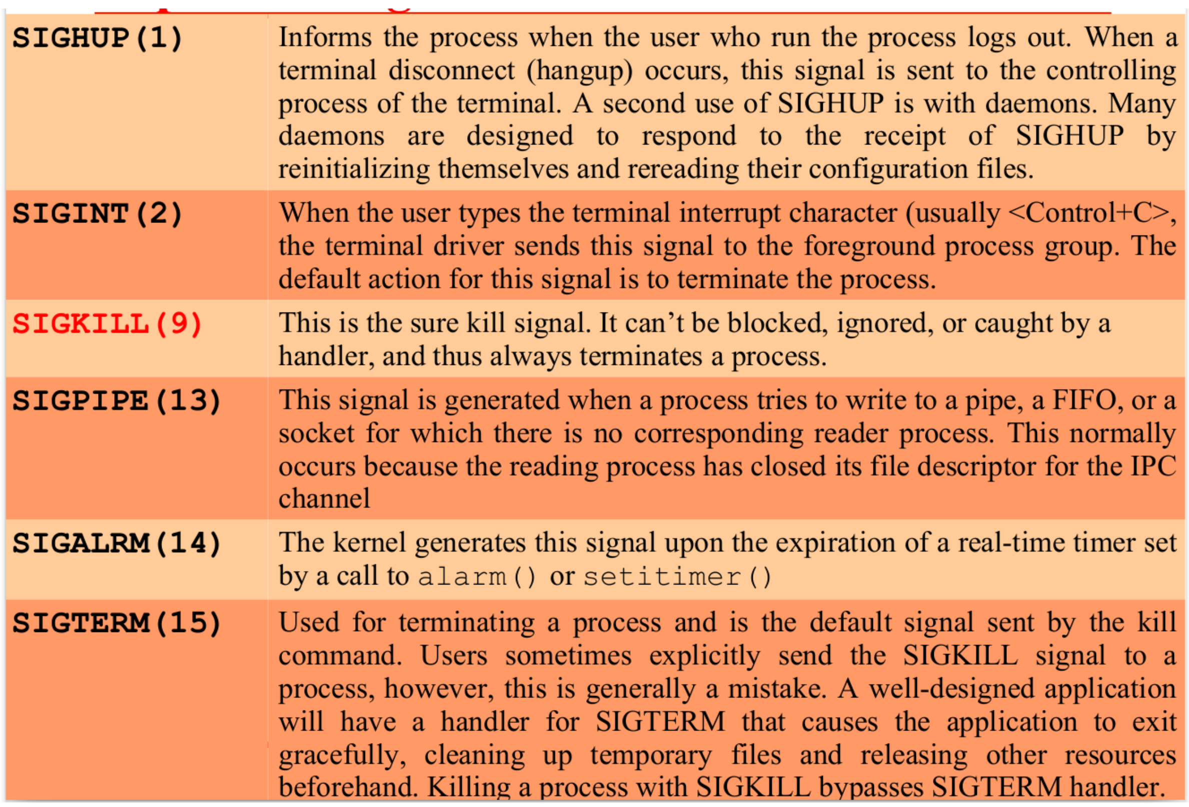
Each signal has a current disposition which determines how the process behave when the OS delivers it the signal

If you install no signal handler, the run time environment sets up a set of default signal handlers for your program. Different default actions for signals are:



TERM	Abnormal termination of the program with `_exit( )` i.e, no clean up. However, status is made available to `wait()` & `waitpid()` which indicates abnormal termination by the specified signal
CORE	Abnormal termination with additional implementation dependent actions, such as creation of core file may occur
STOP	Suspend/stop the execution of the process
CONT	Default action is to continue the process if it is currently stopped

Important Signals:-



SIGHUP(1)	Informs the process when the user who run the process logs out. When a terminal disconnect (hangup) occurs, this signal is sent to the controlling process of the terminal. A second use of SIGHUP is with daemons. Many daemons are designed to respond to the receipt of SIGHUP by reinitializing themselves and rereading their configuration files.
SIGINT(2)	When the user types the terminal interrupt character (usually <Control+C>, the terminal driver sends this signal to the foreground process group. The default action for this signal is to terminate the process.
SIGKILL(9)	This is the sure kill signal. It can't be blocked, ignored, or caught by a handler, and thus always terminates a process.
SIGPIPE(13)	This signal is generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process. This normally occurs because the reading process has closed its file descriptor for the IPC channel
SIGALRM(14)	The kernel generates this signal upon the expiration of a real-time timer set by a call to `alarm()` or `setitimer()`
SIGTERM(15)	Used for terminating a process and is the default signal sent by the kill command. Users sometimes explicitly send the SIGKILL signal to a process, however, this is generally a mistake. A well-designed application will have a handler for SIGTERM that causes the application to exit gracefully, cleaning up temporary files and releasing other resources beforehand. Killing a process with SIGKILL bypasses SIGTERM handler.

SIGQUIT(3)	When the user types the quit character (Control+\) on the keyboard, this signal is sent to the foreground process group. Using SIGQUIT in this manner is useful with a program that is stuck in an infinite loop or is otherwise not responding. By typing Control-\ and then loading the resulting core dump with the gdb debugger and using the backtrace command to obtain a stack trace, we can find out which part of the program code was executing
SIGILL(4)	This signal is sent to a process if it tries to execute an illegal (i.e., incorrectly formed) machine-language instruction module
SIGFPE(9)	Generate by floating point Arithmetic Exception
SIGSEGV(11)	Generated when a program makes an invalid memory reference. A memory reference may be invalid because the referenced page doesn't exist (e.g., it lies in an unmapped area somewhere between the heap and the stack), the process tried to update a location in read-only memory (e.g., the program text segment or a region of mapped memory marked read-only), or the process tried to access a part of kernel memory while running in user mode. In C, these events often result from dereferencing a pointer containing a bad address. The name of this signal derives from the term segmentation violation

Default Behavior: Stop

SIGSTOP(19)	This is the sure stop signal. It can't be blocked, ignored, or caught by a handler; thus, it always stops a process
SIGTSTP(20)	This is the job-control stop signal, sent to stop the foreground process group when the user types the suspend character (usually <Control+Z>) on the keyboard.. The name of this signal derives from "terminal stop"

Default Behavior: Cont

SIGCHILD(17)	This signal is sent (by the kernel) to a parent process when one of its children terminates (either by calling `exit()` or as a result of being killed by a signal). It may also be sent to a process when one of its children is stopped or resumed by a signal
SIGCONT(18)	When sent to a stopped process, this signal causes the process to resume (i.e., to be rescheduled to run at some later time). When received by a process that is not currently stopped, this signal is ignored by default. A process may catch this signal, so that it carries out some action when it resumes

Masking of Signals

A signal is generated by some event. Once generated, a signal is later delivered to a process, which then takes some action in response to the signal. Between the time it is generated and the time it is delivered, a signal is said to be pending. Normally, a pending signal is delivered to a process as soon as it is next scheduled to run, or immediately if the process is already running (e.g., if the process sent a signal to itself). There can be at most one pending signal of any particular type, i.e., standard signals are not queued

Sometimes, however, we need to ensure that a segment of code is not interrupted by the delivery of a signal. To do this, we can add a signal to the process's signal mask—a set of signals whose delivery is currently blocked. If a signal is generated while it is masked/blocked, it remains pending until it is later unmasked or unblocked (removed from the signal mask)

Sending Signals to Processes in a C Program

kill() System call

int kill(pid_t pid, int sig);

One process can send a signal to another process using the kill() system call

The pid argument identifies one or more processes to which the signal specified by sig is to be sent

If sig is zero then normal error checking is performed but no signal is sent. Used to determine if a specified process still exists. If it doesn't exist, a -1 is returned & errno is set to ESRCH

If no process matches the specified pid, kill() fails and sets errno to ESRCH

Four different cases determine how pid is interpreted:

If pid > 0, the signal is sent to the process with the process ID specified by first argument

If pid == 0, the signal is sent to every process in the same process group as the calling process, including the calling process itself

If pid < –1, the signal is sent to every process in the process group whose PGID equals the absolute value of pid

If pid == –1, the signal is sent to every process for which the calling process has permission to send a signal, except init and the calling process. If a privileged process makes this call, then all processes on the system will be signaled, except for these last two

raise() Library call

int raise(int sig);

Sometimes, it is useful for a process to send a signal to itself. The raise() function performs this task

In a single-threaded program, a call to raise() is equivalent to the following call to kill():

kill(getpid(), sig);

When a process sends itself a signal using raise() or kill(), the signal is delivered immediately (i.e., before raise() returns to the caller)

Note that raise() returns a nonzero value (not necessarily –1) on error. The only error that can occur with raise() is EINVAL, because sig was invalid

abort() Library call

void abort();

The abort() function terminates the calling process by raising a SIGABRT signal. The default action for SIGABRT is to produce a core dump file and terminate the process. The core dump file can then be used within a debugger to examine the state of the program at the time of the abort() call

abort() function never returns

pause() System call

int pause();

The pause() system call causes the invoking process/thread to sleep until a signal is received that either terminates it or causes it to call a signal catching function

The pause() function only returns when a signal was caught and the signal-catching function returned. In this case pause() returns -1, and errno is set to EINTR

alarm() System call

unsigned int alarm(unsigned int seconds);

The alarm() system call is used to ask the OS to send calling process a special signal SIGALARM(14) after a given number of seconds. If seconds is zero no new alarm is scheduled

This function returns the previously registered alarm clock for the process that has not yet expired, i.e., the number of seconds left for that alarm clock is returned as the value of this function. Previously registered alarm clock is replaced by new value

UNIX like systems do not operate as real-time systems, so your process might receive this signal after a longer time than requested. Moreover, there is only one alarm clock per process. Can be used for following purposes:

To check timeouts (e.g., wait for user input up to 30 seconds, else exits)
To check some conditions on a regular basis (e.g., if a server has not responded in last 30 seconds, notify the user and exits)

Adding a Delay: using sleep()

int sleep(unsigned int secs);
int usleep(useconds_t usec);
int nanosleep(const struct timespec* req,struct timespec* rem);

These calls causes the calling thread to sleep (suspend execution) either until the number of specified in seconds specified in the argument have elapsed or until a signal arrives which is not ignored

Returns zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler

```
struct timespec {

       time_t tv_sec;                  /* seconds */
       long tv_nsec;                   /* nanoseconds */
};
```

signal() System call

sighandler_t signal(int signum, void (*sh)(int));

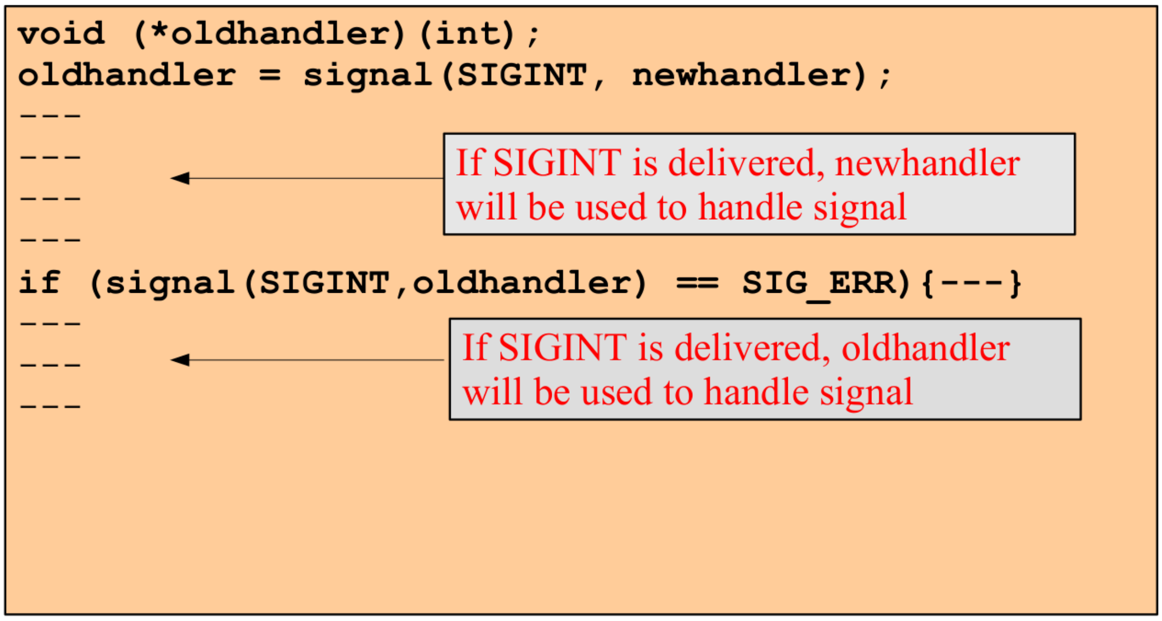
To change the disposition of a particular signal a programmer can use the signal() system call, which installs a new signal handler for the signal with number signum

The second parameter can have three values
i) SIG_IGN: the signal is ignored
ii) SIG_DFL: the default action associated with signal occur
iii) A user specified function address, which is passed an integer argument and returns nothing

The signal() system call returns the previous signal handler, or SIG_ERR on error

The signals SIGKILL and SIGSTOP cannot be caught. Moreover, the behavior of a process is undefined after it ignores SIGFPE, SIGILL or SIGSEGV signal that was not generated by kill() or raise() functions

Handling Signals



```
void (*oldhandler)(int);
oldhandler = signal(SIGINT, newhandler);
---
---
---
---
if (signal(SIGINT,oldhandler) == SIG_ERR){---}
---
---
---
```

If SIGINT is delivered, newhandler will be used to handle signal

If SIGINT is delivered, oldhandler will be used to handle signal

Masking Signals using sigprocmask()

Avoiding Race Conditions Using Signal Mask

One of the problems that might occur when handling a signal, is the occurrence of a second signal while the signal handler function is executing

A process can temporarily prevent signals from being delivered, by blocking/masking it, while it is doing some thing critical, or while it is executing inside a signal handler

Every process has a signal mask that defines the set of signals currently blocked for that process. One bit for each possible signal. If a bit is ON, that signal is currently blocked

Since it is possible for the number of signals to exceed to number of bits in an integer, therefore, POSIX.1 defines a data type called sigset_t that holds a signal set of a process

When a process blocks a signal, the OS doesn’t deliver signal until the process unblocks the signal. However, when a process ignores a signal, signal is delivered and the process handles it by throwing it away

Remember, after a fork(), child process inherits its parent mask

Functions related to Signal Sets

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

To create a process signal mask, you need to create a variable of type sigset_t. The sigemptyset() function initializes a signal set to contain no members, while the sigfillset() function initializes a set to contain all signals.
After initialization, individual signals can be added to a set using sigaddset() and removed using sigdelset().
There are two ways to initialize a signal set:

You can initially specify it to be empty with sigemptyset() and then add specified signals individually using sigaddset()

You can initially specify it to be full with sigfillset() and then delete specified signals individually using sigdelset()

Setting the Process Signal Mask

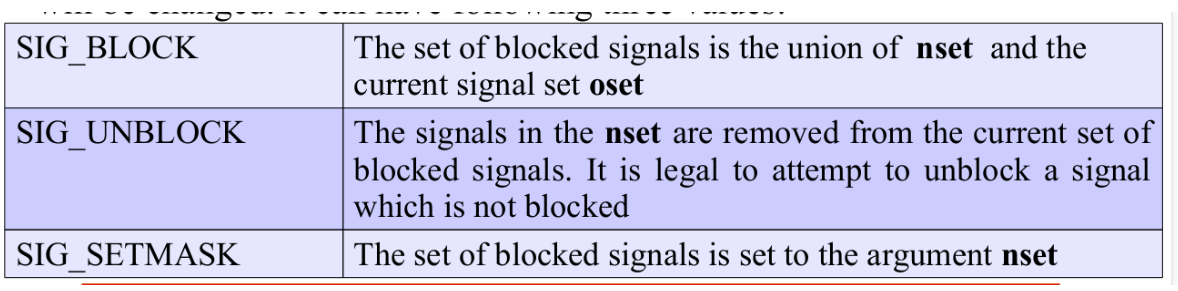
int sigprocmask(int how,const sigset_t* nset, sigset_t* oset);

The sigprocmask() allows us to get the existing signal mask or set a new signal mask of a process

The second argument specifies the new signal mask. It it is NULL, then the signal mask is unchanged

The third argument will store the old mask of the process. This is useful when we want to restore the previous masking state once we're done with our critical section

The first argument how actually determines how the process signal mask will be changed. It can have following three values:



SIG_BLOCK	The set of blocked signals is the union of **nset** and the current signal set **oset**
SIG_UNBLOCK	The signals in the **nset** are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked
SIG_SETMASK	The set of blocked signals is set to the argument **nset**

Ignoring Signals, Masking Signals and Writing SHs using sigaction()

Limitations of signal() System call

Using the signal() call, we cannot determine the current disposition of a signal without changing the disposition.
Example: If we want to determine the current disposition of SIGINT, we can't do it without changing the current disposition

sighandler_t oldHandler = signal(SIGINT, &newHandler);

If we use signal(), to register a handler for a signal, it is possible that after we entered the signal handler, but before we managed to mask all the signals using sigprocmask(), we receive another signal, which WILL be called

There are a lot of variations in the behavior of signal() call across UNIX implementations

sigaction() System call

```
int sigaction(int signum, const struct sigaction* newact,
                    struct sigaction* oldact);
```

Although sigaction() is somewhat more complex to use than signal(), it gives following advantages over signal():

sigaction() allows us to retrieve the disposition of a signal without changing it, and to set various attributes controlling precisely what happens when a signal handler is invoked

sigaction() is more portable than signal()

The first argument signum identifies the signal whose disposition we want to retrieve or change

The second argument newact is a pointer to a structure specifying a new disposition for the signal. If we are interested only in finding the existing disposition of the signal, then we can specify NULL for this argument

The third argument oldact is used to return information about the signal’s previous disposition. If we are not interested in this information, then we can specify NULL for this argument

The structures pointed to by third and fourth argument to sigaction is of following type:

```
struct sigaction {
        void (*sa_handler)(int);
        sigset_t sa_mask;
        int sa_flags;
};
```

The sa_handler field specifies the pointer to the handler function

The sa_mask field specifies the process signal mask to be set while this signal is being handled

The sa_flags field contains flags that effect signal behavior, normally it is set to zero