```
                          31 30          I           1   0
                         --------------------------------
      signal bits        |   |   |   . . |1 | . .   |  |  | 0=no signal,1=signal
                         --------------------------------

                         --------------------------------
      signal masks       |   |   |   . . |0 | . .   |  |  | 0=no BLOCK, 1=BLOCK
                         --------------------------------

                         --------------------------------
      signal handler     |   |   |   . . |0 | . .   |  |  | 0=DEFault, 1=IGNore
                         -------------------------------- other=catcher address
```

Fig. 6.1 Signal Handling Diagram

kernel mode finds an unblocked signal, it clears the signal bit to 0 and tries to handle the signal by the handler function in the signal handler array. A 0 entry means DEFault, a 1 means IGNore and other values means a preinstalled catcher function in user space.

6.3.5 Install Signal Catchers

A process may use the system call

```
        int r = signal(int signal_number, void *handler);
```

to change the handler function of a selected signal number, except SIGKILL(9) and SIGSTOP(19), which can not be changed. The installed handler, if not 0 or 1, must be the entry address of a signal catcher function in user space of the form

```
        void catcher(int signal_number){.............}
```

The signal() system call is available in all Unix-like systems but it has some undesirable features.

(1). Before executing the installed signal catcher, the signal handler is usually reset to DEFault. In order to catch the next occurrence of the same signal, the catcher must be installed again. This may lead to a race condition between the next signal and reinstalling the signal handler. In contrast, sigaction() automatically blocks the next signal while executing the current catcher, so there is no race condition.
(2). Signal() can not block other signals. If needed, the user must use sigprocmask() to explicitly block/unblock other signals. In contrast, sigaction() can specify other signals to be blocked.
(3). Signal() can only transmit a signal number to the catcher function. Sigaction() can transmit addition information about the signal.
(4). Signal() may not work for threads in a multi-threaded program. Sigaction() works for threads.
(5). Signal() may vary across different versions of Unix. Sigaction() is the POISX standard, which is more portable.

For these reasons, signal() has been replaced by the POSIX **sigaction()** function. In Linux (Bovet and Cesati 2005), sigaction() is a system call. It has the prototype

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

The sigaction structure is defined as

```
    struct sigaction{
       void     (*sa_handler)(int);
       void     (*sa_sigaction)(int, siginfo_t *, void *);
       sigset_t sa_mask;
       int      sa_flags;
       void     (*sa_restorer)(void);
    };
```

The most important fields are:

- **sa_handler:** This is the pointer to a handler function that has the same prototype as a handler for signal().
- **sa_sigaction:** This is an alternative way to run the signal handler. It has two additional arguments beside the signal number where the `siginfo_t` * provides more information about the received signal.
- **sa_mask:** allows to set signals to be blocked during execution of the handler.
- **sa_flags:** allows to modify the behavior of the signal handling process. To use the `sa_sigaction` handler, sa_flags must be set to `SA_SIGINFO`.

For a detailed description of the sigaction structure fields, see the sigaction manual page. The following shows a simple example of using the sigaction() system call.

Example 6.6 Example use of sigaction()

```
/****** sigaction.c file *******/
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handler(int sig, siginfo_t *siginfo, void *context)
{
  printf("handler: sig=%d from PID=%d UID=%d\n",
                   sig, siginfo->si_pid, siginfo->si_uid);
}

int main(int argc, char *argv[])
{
  struct sigaction act;
  memset(&act, 0, sizeof(act));
  act.sa_sigaction = &handler;
  act.sa_flags = SA_SIGINFO;
  sigaction(SIGTERM, &act, NULL);
  printf("proc PID=%d looping\n”);
  printf(“enter kill PID to send SIGTERM signal to it\n", getpid());
  while(1)(
    sleep(10);
  }
}
```

While the program is running, the reader may enter **kill PID** from another X-terminal, which sends a SIGTERM(15) signal to the process. Any signal would wake up the process even if it is in the sleep state, allowing it to handle the signal. In the signal handler we read two fields from the siginfo parameter to show the signal sender's PID and UID, which are unavailable if the signal handler was installed by the signal() system call.

(6). A process may use the system call

```
int r = kill(pid, signal_number);
```

to send a signal to another process identified by pid. The sh command

```
kill -s signal_number pid
```

uses the kill system call. In general, only related processes, e.g. those with the same uid, may send signals to each other. However, a superuser process (uid=0) may send signals to any process. The kill system call uses an invalid pid, to mean different ways of delivering the signal. For example, pid=0 sends the signal to all processes in the same process group, pid=-1 for all processes with pid>1, etc. The reader may consult Linux man pages on signal/kill for more details.

6.4 Signal Processing Steps

(7). A process checks signals and handles outstanding signals when it is in kernel mode. If a signal has a user installed catcher function, the process first clears the signal, fetches the catcher's address and, for most trap related signals, resets the installed catcher to DEFault. Then it manipulates the return path in such a way that it returns to execute the catcher function in user mode. When the catcher function finishes, it returns to the original point of interruption, i.e. from where it lastly entered kernel mode. Thus, the process takes a detour to execute the catcher function first. Then it resumes normal execution.

(8). Reset user installed signal catchers: User installed catcher functions for trap related signals are intended to deal with trap errors in user code. Since the catcher function is also executed in user mode, it may commit the same kind of errors again. If so, the process would end up in an infinite loop, jumping between user mode and kernel mode forever. To prevent this, the Unix kernel typically resets the handler to DEFault before letting the process execute the catcher function. This implies that a user installed catcher function is good for only one occurrence of the signal. To catch another occurrence of the same signal, the catcher must be installed again. However, the treatment of user installed signal catchers is not uniform as it varies across different versions of Unix. For instance, in BSD Unix the signal handler is not reset but the same signal is blocked while executing the signal catcher. Interested readers may consult the man pages of signal and sigaction of Linux for more details.

(9). Signal and Wakeup: There are two kinds of SLEEP processes in the Unix/Linux kernel; sound sleepers and light sleepers. The former are non-interruptible, but the latter are interruptible by signals. If a process is in the non-interruptible SLEEP state, arriving signals (which must originate from hardware interrupts or another process) do not wake up the process. If it is in the interruptible SLEEP state, arriving signals will wake it up. For example, when a process waits for terminal inputs, it sleeps with a low priority, which is interruptible, a signal such as SIGINT will wake it up.