Chapter: Signals

1. Concepts and Overview

1. A signal is a notification to a process that an event has occurred.

2. Signal are nothing but *software Interrupts.*

3. Origination of signals,
   - Kernel
   - From one process to another process
   - Process can send the signal to itself.

4. Types of Events
   - A hardware exception occurred.
     - ✔ Examples:
       - Divide by Zero
       - Segmentation Fault
   - The user typed one of the terminal special characters that generate signals.
     - ✔ Examples:
       - interrupt character (Control + c)
       - suspend character (Control-Z)
   - A software event occurred.
     - ✔ Examples:
       - Child process terminated
       - Input is available.

5. Each signal is defined as a unique (small) integer, starting sequentially from 1.
   - These integers are defined in <signal.h>
   - Actual numbers used for each signal vary across implementations, that's why macros are used.
   - Example:

- ✔ when the user types the interrupt character, SIGINT (signal number 2) is delivered to a process.
- ✔ $kill -l , gives the listings of all the signals.

6. Signal: Categories.
   - Traditional / Standard signals numbering from 1 to 31.
   - Real-time Signals, rest

7. Signals: Life Cycle
   - Generated
   - Pending
   - Delivered

8. 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).
   - process's signal mask—a set of signals whose delivery is currently blocked.
     - ✔ Why? To ensure that a segment of code is not interrupted by the delivery of a signal.
     - ✔ If a signal is generated while it is blocked, it remains pending until it is later unblocked (removed from the signal mask).

9. Upon delivery of a signal, a process carries out one of the following default actions, depending on the signal:
   - The signal is ignored
   - The process is terminated (killed)
   - A core dump file is generated, and the process is terminated
   - The process is stopped
   - process is resumed

10. Disposition of the signal.
    - a program can change the action that occurs when the signal is delivered.
    - A program can set one of the following dispositions for a signal:
      - ✔ The default action should occur.
      - ✔ The signal is ignored.
      - ✔ A signal handler is executed.

2. Changing Signal Dispositions: signal()

1. Two ways of changing the disposition of a signal: signal() and sigaction().
   - The signal() system call, provides a simpler interface than sigaction().
   - sigaction() provides functionality that is not available with signal().
   - there are variations in the behavior of signal() across linux implementations which mean that it should never be used for establishing signal handlers in portable programs.
2. signal() is implemented in glibc as a library function layered on top of the sigaction() system call.
3. Refer:
   - /home/satya/Desktop/LI/3-IPC/4-Signals/2-Practice/2_signal_handler.c

3. Introduction to Signal Handlers

1. A signal handler (signal catcher) is a function that is called when a specified signal is delivered to a process.
2. Refer:
   - Signal delivery and handler execution fig:20-1, page-399
3. When the kernel invokes a signal handler, it passes the number of the signal that caused the invocation as an integer argument to the handler.
4. Refer:
   - /home/satya/Desktop/LI/3-IPC/4-Signals/2-Practice/3_multiple_signal_handler.c

4. Sending Signals: kill()

1. One process can send a signal to another process using the kill() system call, which is the analog of the kill shell command.
2. Refer:
   - /home/satya/Desktop/LI/3-IPC/4-Signals/2-Practice/4_process_process_sig_kill.c
3. If no process matches the specified pid, kill() fails and sets errno to ESRCH ("No such process").

5. Other Ways of Sending Signals: raise()

1. raise(): A process to sending a signal to itself
   - In a single-threaded program, a call to raise() is equivalent to the following call to kill(): kill(getpid(), sig);

2. When a process sends itself a signal using raise() (or kill()), the signal is delivered immediately.

6. Changing Signal Dispositions: sigaction()

1. The sigaction() system call is an alternative to signal() for setting the disposition of a signal.
2. The sigaction() provides greater flexibility.
3. sigaction() is more portable than signal() when establishing a signal handler.
4. Using sigaction(), one can block the reception of specified signals while your handler runs, and to retrieve a wide range of data about the system and process state at the moment a signal was raised.
5. PROTOTYPE: Sigaction

<table>
<tr><td>System Call</td><td>Sigaction: examine and change a signal action</td></tr>
<tr><td>Prototype</td><td>int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);</td></tr>
<tr><td>Parameters</td><td></td></tr>
<tr><td>signum</td><td>The signal and can be any valid signal except SIGKILL and SIGSTOP.</td></tr>
<tr><td>act</td><td>1. If act is non-NULL, the new action for signal signum is installed from act.</td></tr>
<tr><td>oldact</td><td>1. If oldact is not NULL, the call stores the previous (or current, if act is NULL) behavior of the given signal there.</td></tr>
<tr><td>Possibilities</td><td>
<table>
<tr><th>Sl.No.</th><th>act</th><th>oldact</th><th>Remarks</th></tr>
<tr><td>1</td><td>NON-NULL</td><td>NULL</td><td>New action will be taken place</td></tr>
<tr><td>2</td><td>NULL</td><td>NON-NULL</td><td>Old action will be preserved</td></tr>
<tr><td>3</td><td>NULL</td><td>NULL</td><td>Default action of the signal</td></tr>
<tr><td>4</td><td>NON-NULL</td><td>NON-NULL</td><td>New action will be taken place</td></tr>
</table>
</td></tr>
<tr><td>Return Value</td><td>returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.</td></tr>
</table>

6. The sigaction structure is defined as something like:

<table>
<tr><td>Structure</td><td>

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

</td></tr>
<tr><td>Member-1</td><td>

```
void (*sa_handler)(int);
```

1. Pointer to the signal handler.
2. May be SIG_DFL for the default action, SIG_IGN to ignore this signal
</td></tr>
<tr><td>Member-2</td><td>

```
void (*sa_sigaction)(int, siginfo_t *, void *);
```

</td></tr>
</table>

	1. If SA_SIGINFO is specified in sa_flags, then sa_sigaction (instead of sa_handler) specifies the signal-handling function for signum. 2. This function receives - The signal number as its first argument. - a pointer to a siginfo_t as its second argument. - a pointer to a ucontext_t (cast to void *) as its third argument.
Member-3	sigset_t sa_mask; 1. specifies a mask of signals which should be blocked during execution of the signal handler. 2. In addition, the signal which triggered the handler will be blocked, unless the SA_NODEFER flag is used.
Member-4	int sa_flags; 1. specifies a set of flags which modify the behavior of the signal. 2. It is formed by the bitwise OR of zero or more flags(See Manual)

7. Refer:
   - /home/satya/Desktop/LI/3-IPC/4-Signals/1-Demos/4-AdvancedSignalHandling/*