Signals

Class Notes

Signals

Signals are asynchronous messages delivered to a process or group of processes from kernel's process manager. Linux supports 64 signals. UNIX and its variants provide two different categories of signals.

1) **General purpose** (Event notification) signals
2) **Real time** (Process communication) signals

- A general purpose signals are mapped to specific system events and they are identified with a specific name and name is assigned as per the event to which they are bound.
- Set of signals assigned to job control events are triggered when specific job control operations are initiated.

General purpose signals are of five categories:

- **Job control**: SIGCONT (18), SIGSTOP (19), SIGSTP (20).

- **Termination:** These signals are used to interrupt or terminate a running process.
  SIGINT (2), SIGQUIT (3), SIGABRT (6), SIGKILL (9), SIGTERM (15).

- **Async I/O**: These signals are generated when data is available on a specific device or when kernel services wish to notify applications about resource state.
  SIGURG (23), SIGIO (29), SIGPOLL (29).

- **Timer:** These signals are generated when application chooses timers alarms.
  SIGALRM (14), SIGPROF (27), SIGVTALRM (26).

- **Error reporting**: These signals occur when application code runs into an exception.
  SIGHUP (1), SIGILL (4), SIGTRAP (5), SIGBUS (7), SIGFPE (8), SIGPIPE (13), SIGSEGV (11), SIGXCPU (24).

Real time signals are exclusively meant for process communication. These signals do not have pre assigned names. They are priority ordered (33-64).

Signal flow:

- The **source** of a signal can be
    1) Process
    2) Exception handler (error reporting signals)
    3) Kernel services (drivers)
- The **destination** of the signal would be a process always.

- Signal generation and signal delivery may not take place in sequence. Signaling subsystem may delay the delivery of a signal when destination process is not ready to receive signals. The main reasons for the signal delay are
    1) Destination process is in wait state.
    2) Destination process has explicitly disabled signal delivery.

- An application can respond to a signal using either of the following methods.
    1) Execution of kernel defined signal handler (default).
    2) Execution of application defined (specific) signal handler.
    3) Ignore.

Register application specific signal handler:

```
#include<signal.h>

signal (SIGINT,sighand);
void sighand(int signum)
{
        <body>
}
```

Ignoring the signal:

```
#include<signal.h>

signal (SIGINT,SIG_IGN);    /* ignoring the signal */

signal (SIGINT,SIG_DFL);    /* switching  back to default */
```

Sigaction:

Posix standard provides a function **sigaction()** to register signal handlers or to change signal disposition.

```
NAME
       sigaction - examine and change a signal action

SYNOPSIS
       #include <signal.h>

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

```
The sigaction structure is defined as something like:

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

- Applications can choose to block signals from being delivered while handling priority signal handlers. Sigaction provides an option of blocking chosen signals while a critical handler is in execution.

  Sample code:

```
struct  sigaction act;
sigset_t sigmask;
int rc;
rc = sigemptyset (&sigmask);
rc=sigaddset (&sigmask ,SIGQUIT);
act.sa_handler = handler;
act.sa_mask = sigmask;
sigaction (SIGALRM ,&act,NULL);
```

- If a signal is generated while its delivery is blocked, it will be stored in the process signal pending queue. Pending signals are delivered immediately when the process unblocks signal delivery.
- Linux kernel allows applications to be in either of the following states of wait states.
    1) **INTERRUPTABLE_WAIT**: while a process is in this state, it can be interrupted by signals.
    2) **UNINTERRUPTABLE_WAIT**: while a process is in this state, all signals are blocked.
- Signal handlers allocate stack frames with in stack segment of the process address space. Applications can program a signal handler to use an alternate stack for its local data allocation.

Sample code**:**

Set up alternate stack:

```
stack_t sigstack;
sigstack.ss_sp = malloc(SIGSTKSZ);
sigstack.ss_size = SIGSTKSZ;
sigstack (&sigstack ,NULL);
```

Register signal handler using sigaction with flag SA_ONSTACK:

```
sa.sa_handler = sigsegvHandler;
sa.sa_flags = SA_ONSTACK;
sigaction (SIGSEGV ,&sa ,NULL);
```

Sigprocmask:

Applications can block delivery of signals during execution of main thread.

```
#include<signal.h>
int sigprocmask (int how, const sigset_t*set ,sigset_t*oldset);
```

Sample code:

```
sigset_t s_set;
sigemptyset (&s_set);
sigaddset (&s_set,4);
sigprocmask (SIG_BLOCK | SIG_SETMASK , &s_set ,NULL);
{
body;
}
sigprocmask (SIG_UNBLOCK, &s_set ,NULL);
```

- Sigprocmask when used with unblock command clears the specified signal from the block list and delivers them if they are found in the pending queue.
- Sigprocmask can be used to extract current blocked signals. Applications can append new signals to current blocked signal list.
- Applications can overwrite current blocked signal list by setting SIG_SETMASK flag as the first argument of the sigprocmask call.
- Applications can block or ignore or change signal disposition application defined handler for any signal except 9 and 19.

When signal handlers are in execution, application's primary thread is blocked. Application can resume executing only after termination of signal handler. When implementing application-defined-signal handlers the following issues are to be considered.

- Avoid longer work (instructions) in handlers.
- Avoid blocking function calls and operations.
- Avoid accessing global data
- Avoid calling async routines

Communication signals:

```
#include<signal.h>

int sigwaitinfo (const sigset_t*set , siginfo_t*info);

int sigtimedwait (const sigset_t*set , siginfo_t*info ,const struct timespec*timeout);
```

Sample code:

Step1: Block the signals which need to handle synchronously

```
sigset_t waitset;

sigemptyset (&waitset);

sigaddset (&waitset ,SIGALRM);

sigprocmask (SIG_BLOCK, &waitset ,NULL);
```

Step2: Wait for the chosen signal to occur

```
sigfinfo_t info;

sigwaitinfo (&waitset , &info);
```

Step3: Assign timeout. Sigtimedwait puts the application into wait until specific timeout.

```
struct timespec timeout;

timeout.tv_sec = 10;

timeout.tv_nsec = 1000;
```

signalfd() creates a file descriptor that can be used to accept signals targeted at the caller. This provides an alternate to the use of signal handler or sigwaitinfo and has got the advantage that the file descriptor can be monitored by select, poll and epoll.

Sample code:

```
sigset_ t waitset;

sigemptyset (&emptyset);

sigaddset (&waitset ,SIGALRM);

sigprocmask (SIG_BLOCK ,&waitset ,NULL);

struct  signalfd_siginfo info;

read (sfd, &info ,sizeof(signalfd_siginfo));
```