POSIX.1 Signals

- Even under **Linux**, the implementation of ***signal()*** changed between **libc5** and **glibc** libraries.

- Originally the ***signal()*** system call was used to influence the behavior of signals. The problem with the original implementation of signals was that they were unreliable, signals could be lost.

- A process also had to reinstall a signal handler every time it was invoked.

- Recall the mechanism for the ***signal()*** call:

  typedef void (*signal_handler_type)(int signo);

  signal_handler_type signal (int signum, signal_handler_type sighand);

- The first line declares a pointer to a function that takes a signal value as a n argument and returns NULL.

- The ***signal()*** system call expects a signal (e.g. **SIGHUP**) and a pointer to the signal handler to use. It returns the old signal handler that was registered for that signal.

- Consider what ***signal()*** does if a signal is received and the signal handler for that signal is already running:

  1. Go ahead and invoke the signal handler. Not a good idea because most signal handlers are not designed to handle two instances of the same signal at the same time.

  2. Hold off on delivery of the second signal until the signal handler has finished handling the first signal.

  3. Just before running the signal handler for the first time, reset the signal handler for the signal to **SIG_DFL**. This allows the signal handler to reregister itself right away if it wants to handle new receptions of that signal. It can also reregister itself as the very last thing it does if it prefers not to be run simultaneously.

- While #3 may seem like a good idea, it is in fact a very bad idea. The main problem lies with the fact that there is a time window between the kernel resetting the signal handler to SIG_DFL and the signal handler getting a chance to reregister itself.

- If a second instance of is received within this window of vulnerability, then the default behavior will occur, and there is nothing an application can do about it.

- There are two different types of system calls, slow and fast.

- Fast system calls are those that are guaranteed to return, while slow system calls can remain blocked for long periods of time.

- Examples of fast system calls are ***stat()*** and ***select(***) (this one is considered fast because it takes a timeout value as an argument).

- Example of a slow and fast system call is ***read()***. This call is **fast** on a **regular file** but **slow** on a **pipe**.

- Some slow' system calls like read's from pipes can be interrupted by a signal from the kernel.

- They return the error code EINTR, and the program now has the responsibility to issue the system call again if needed.

- This is not always a desired behavior and some systems allow an automatic restart of these slow system calls on a per-signal basis (option **SA_RESTART**, a Berkeley extension, with ***sigaction()***).

- The table shown below summarizes the signals that every POSIX-compliant system must support.

Signal	Default Action	Description
SIGABRT	terminate w/core	Abnormal termination, caused by ***abort()***.
SIGALRM	terminate	Timeout, caused e.g. by ***alarm()***.
SIGFPE	terminate w/core	Erroneous arithmetic operation (e.g. divide by zero).
SIGHUP	terminate	Hangup on controlling terminal.
SIGILL	terminate w/core	Invalid hardware instruction.
SIGINT	terminate	Interactive attention signal (interrupt).
SIGKILL	terminate	Termination. Cannot be caught or ignored.
SIGPIPE	terminate	Write on a pipe that is not open for reading by another process.
SIGQUIT	terminate w/core	Interactive termination signal (quit).
SIGSEGV	terminate w/core	Invalid memory reference.
SIGTERM	terminate	Termination signal.
SIGUSR1	terminate	Application-define signal 1
SIGUSR2	terminate	Application-define signal 2

- The system calls shown below are used for manipulating signals above.

Function	Description
kill()	Sends a signal to a specified process
raise()	Sends a signal to the current process
alarm()	Schedules an alarm
pause()	Suspends process execution
sigaction()	Tells the kernel how to deliver a signal to a process
sigprocmask()	Examines and changes blocked signals
sigpending()	Examines pending signals
sigsuspend()	Process suspends itself and waits for a signal (any signal)

- The data type that the signal functions rely on is called ***sigset_t***. It represents a set of signals.

- Every signal is either contained in or absent from each ***sigset_t*** object you use. This allows programs to easily pass a list of signals to the kernel.

- Here is how these signal masks are defined in a program:

  // signal masks
  sigset_t newmask, oldmask, zeromask, testmask;

- The signal API provides a set of functions that must be used to manipulate ***sigset_t*** objects.

  int sigemptyset(sigset_t *set);

  int sigfillset(sigset_t *set);

  int sigaddset(sigset_t *set, int signum);

  int sigdelset(sigset_t *set, int signum);

  int sigismember(const sigset_t *set, int signum);

- The first argument to all of these functions is a **pointer** to the **signal set** that is being manipulated.

- ***sigemptyset*** initializes the signal set given by set to empty, with all signals excluded from the set.

- ***sigfillset*** adds every signal to the set.

- ***sigaddset*** and ***sigdelset*** add and delete respectively signal **signum** from **set**. This allows a program to build sets that contain exactly the signals they need.

- ***sigismember*** tests whether **signum** is a member of **set**.

- The following code fragment illustrates the use of these system calls:

  ```
  sigemptyset(&zeromask);          // clear all signals in signal mask
  sigemptyset(&newmask);           // generate signal mask with SIGINT
  sigaddset(&newmask, SIGINT);
  ```

- The next step is to catch a signal. The ***sigaction*** structure is used to deliver a signal to the program:

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

- **sa_handler** specifies the action to be associated with ***signum*** and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.

- **sa_mask** gives a mask of signals which should be blocked when the signal handler is being run.

- The **sa_flags** field is a bitmask of various flags logically ORed together, the combination of which specifies the kernel's behavior when the signal is received (see man pages).

- The **sa_restorer** field is obsolete and should not be used.

- The ***sigaction()*** system call is used to tell the kernel how it should deliver a particular signal to the process.

- The sample program provided illustrates the use of a simple signal handler for two signals, **SIGINT** (^C) and **SIGTSTP** (^D).

- A process can examine or modify its process signal mask with the ***sigprocmask*** system call.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

- The ***how*** argument specifies the manner in which the set is changed, and consists of one of the following values:

SIG_BLOCK
The resulting set will be the union of the current set and the signal set pointed to by *set*.

SIG_SETMASK
The resulting set will be the signal set pointed to by *set*.

SIG_UNBLOCK
The resulting set will be the intersection of the current set and the complement of the signal set pointed to by *set*.

- Keep in mind that some signals, such as SIGSTOP and SIGKILL, cannot be blocked. If an attempt is made to block these signals, the system ignores the request without reporting an error.