Signals 2: System Calls

1. Overview
2. **System Calls**
   - **unreliable vs. reliable API**
   - **setting disposition: signal() and sigaction()**
   - **blocking: sigprocmask(), etc.**
   - **sending: kill(), etc.**
3. Signal Handlers
4. Synchronization
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

Signal System Calls

There a large number of system calls dealing with signals:

- **signal** – set signal disposition
- **sigaction** – set signal disposition
- **sigemptyset** – set no signals in signal set/mask
- **sigfillset** – set all signals in signal set/mask
- **sigaddset** – add signal to signal set/mask
- **sigdelset** – remove signal from signal set/mask
- **sigismember** – check if signal in signal set
- **sigprocmask** – change signal mask (blocked signals)
- **sigpending** – check the pending blocked signals

Signal System Calls (contd.)

continuing:

- **pause** – suspend process until receive signal
- **sigsuspend** – suspend process until receive signal
- **sigwait,sigwaitinfo** – wait for queued signals
- **sigtimedwait** – wait for queued signals, with time limit
- **kill,sigqueue** – send a signal to a process(es)
- **raise** – send a signal to self
- **sleep** – suspend process for n seconds or signal
- **alarm** – send alarm signal to self after n seconds
- **abort** – send abort signal to self

Unreliable vs. Reliable Signals

The original UNIX signal API left many behaviors unspecified and unable to be set as desired (with different UNIXes having different behaviors).

Among the key issues:

- does a signal handler *stay in effect* after it is invoked or not? (not resetting is called *"mousetrap behavior"*)
- can a signal handler be *interrupted* by the *same signal* (so another call to that handler)?
- can a signal handler be *interrupted* by a *different signal* (so another signal handler)?
- is an *interrupted syscall* automatically *restarted* or not?

Because you could not rely on the behavior you would get across UNIXes, this is often called the **unreliable** signal interface.

Unreliable vs. Reliable Signals (contd.)

`signal()` is the older, **unreliable** signals API syscall.

`sigaction()` is the newer, **reliable** signals API syscall.

The newer signals API both specifies default behaviors for the above issues and provides mechanisms for changing those behaviors.

Because it is simpler, `signal()` can still be used to set *default or ignore disposition*.

However, because its behavior is "unreliable," *it should never be used to setup a signal handler*.

Setting Disposition: signal()

`signal()` is the older, *unreliable* API to set signal disposition:
`sighandler_t signal(int signum, sighandler_t handler)`

- `sighandler_t` is defined as:
  - `typedef void (*sighandler_t)(int)`
- `signum` is the signal (symbol) whose disposition is being changed
- `handler` is the name of a **signal handler** function or one of the special values:
  - `SIG_DFL` (default disposition)
  - `SIG_IGN` (ignore disposition)
- returns the previous value of the signal handler, else `SIG_ERR`

An example call might be: `signal(SIGCHLD,SIG_IGN);`

Setting Disposition: signal() (contd.)

`signal()` is what we call a **higher-order function**, as it takes a function as one of its parameters.

In C, you pass a function as an argument by giving the function's name, but what is actually passed is a *pointer to the function*.

This is why type `sighandler_t` is specified as it is, the notation "(***funcname*)" denotes a **function pointer**.

As already noted, `signal()` should be used only to set default or ignore disposition—*it should not be used to setup a handler*.

Thus `signal()` should be called with second arguments that are only either `SIG_DFL` or `SIG_IGN`.

Setting Disposition: sigaction()

`sigaction()` is the newer, *reliable* API to set signal disposition:

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

- `signum` is the signal (symbol) whose disposition is being changed
- `act` contains the (new) disposition settings
- `oldact` receives the old disposition settings
  (so they can be easily restored)
- `oldact` can be `NULL` if the old disposition is no longer needed
- returns 0 on success, -1 on error

Setting Disposition: sigaction() (contd.)

`sigaction()` makes use of the `sigaction` struct type:

```
struct sigaction {
  void     (*sa_handler)(int);   /* Disposition/handler specification */
  void     (*sa_sigaction)(int, siginfo_t *, void *);
                                 /* Alternative "siginfo handler" */
  sigset_t sa_mask;              /* Signal mask during handler run*/
  int      sa_flags;             /* Flags/options */
  void     (*sa_restorer)(void); /* Obsolete */
}
```

Setting Disposition: sigaction() (contd.)

Example of registering a handler using `sigaction()`:

```
struct sigaction act;

memset(&act,0,sizeof(act));
act.sa_handler = sigint_handler;  //Name of handler function
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGCHLD);
act.sa_flags = SA_RESTART;

sigaction(SIGINT,&act,NULL)
```

Setting Disposition: sigaction() (contd.)

The default behavior for the key issues mentioned earlier are:

- a signal handler *stays in effect* after it is invoked (i.e., not "mousetrap behavior)
- a signal handler cannot be *interrupted* by the *same signal*
- a signal handler can be *interrupted* by a *different signal*
- an *interrupted syscall* is *not* automatically *restarted*

These behaviors can all be changed by including appropriate `sa_flags` and/or `sa_mask` values in the sigaction struct.

Setting Disposition: sigaction() (contd.)

Key values for the sigaction struct `sa_flags` field:

- `SA_RESETHAND` – restore default disposition once the signal handler has been invoked ("mousetrap behavior")
- `SA_NODEFER` – do not block signal while its own handler is being run (default is to block even signal if not in `sa_mask`)
- `SA_RESTART` – automatically restart interrupted system calls
- `SA_SIGINFO` – use alternative handler format

Signal Blocking

The asynchronous delivery of a signal can cause problems:

- the signal might interrupt a *"slow"* syscall like `read()`
- termination might interrupt a sequence of program steps, leaving a database (or similar) in an inconsistent state
- child might signal parent before parent has updated its state to reflect `fork()`

Thus, there may be **critical sections** in programs where we do not want particular (or even any) signals to be delivered.

Most signals can be **blocked**: held by the kernel for possible later delivery (when unblocked).

A process' **signal mask** is its current set of blocked signals.

Signal Blocking (contd.)

Two signals *cannot be blocked*: `SIGKILL` and `SIGSTOP`.

(They also cannot be ignored.)

This is why a sure way to terminate a process using the `kill` command is to use `-9` or `-SIGKILL` to send `SIGKILL`.

Note that with *standard signals*, only *one copy* of a blocked signal is maintained by the kernel, even if multiple instances of the signal were generated while the signal was blocked.

Also, there are no guarantees that blocked signals will be delivered in the order they were generated once they are unblocked.

Blocking: sigprocmask

A process' **signal mask** of blocked signals is changed using `sigprocmask`:

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
```

- changes the *signal mask* based on a **signal set**
- `how` determines what changes to make:
  - `SIG_BLOCK` – add signals in `set` to signal mask
  - `SIG_UNBLOCK` – remove signals in `set` from signal mask
  - `SIG_SETMASK` – set the signal mask to be `set`
- `set` is the signal set that specifies the signals to act on
- `oldset` stores the old signal mask (so it can be restored)

Blocking: sigprocmask (contd.)

Example of setting the signal mask to block only the SIGUSR signals:

```
sigset_t sset_siguser;

sigemptyset(&sset_siguser);
sigaddset(&sset_siguser,SIGUSR1);
sigaddset(&sset_siguser,SIGUSR2);

sigprocmask(SIG_SETMASK,&sset_siguser,NULL);
```

Blocking: sigprocmask (contd.)

Signal sets can be manipulated with several functions:

- `sigemptyset()`
- `sigfillset()`
- `sigaddset()`
- `sigdelset()`
- `sigismember()`

Blocking: sigpending

Sometimes one may want to check if there are any *pending blocked signals*, such as before unblocking signals.

This can be done with `sigpending`:

```
int sigpending(sigset_t *set)
```

- `set` receives the set of pending signals, as a *signal set*

Sending Signals: kill

A process can send a signal to another process using `kill`:
`int kill(pid_t pid, int sig)`

- `pid` can be one of:
  - a positive integer – send to PID of `pid`
  - 0 – send to every process in *process group* of caller
  - -1 – send to every process have permission to send to
  - a negative integer – send to *process group* `|pid|`
- `sig` is the signal to send, else if 0 "no signal is sent, but error checking is still performed; this can be used to check for the existence of a process ID or process group ID"

To be allowed to signal another process, the sender must have the same real or effective UID as the target process' real UID, or else have `root` as its effective UID (really `CAP_KILL`).

Sending Signals: sigqueue and raise

`sigqueue` is an alternative to `kill()`:

```
int sigqueue(pid_t pid, int sig, const union sigval value)
```

The primary reason to use `sigqueue()` is that it can be used to pass additional info to a handler ("**siginfo handlers**").

Siginfo handlers are installed by using `sigaction()`'s `SA_SIGINFO` flag and `sa_sigaction` struct field.

`raise` allows a process to *signal itself*:

```
int raise(int sig)
```

Abnormal Termination: abort

`abort` can be used by a process to terminate itself *abnormally*:
`void abort(void)`

- first unblocks `SIGABRT` then raises that signal for the calling process
- results in the abnormal termination unless `SIGABRT` is caught and the handler does not return (e.g., `siglongjmp()`)
- if `SIGABRT` is ignored or caught by a handler that returns, the process will still terminate: default disposition for `SIGABRT` will be restored and signal raised for a second time.
- never returns