# 6. Signal

Signal indicated by a small integer—the only “data” received

To send signal:

```
int kill(int pid, int signum)
```

To receive signal, must register function with OS

OS will interrupt program (at any point) and run *signal handler* function

# Some Terminology & Facts

Signal is **generated** somehow

Eventually it is **delivered** to target process

Between generation & delivery, signal is **pending**

Delivery can occur only when target is running

Signal number has corresponding symbolic name (beginning with "`SIG`") listed in `/usr/include/bits/signum.h`

E.g., `SIGFPE` ("floating point exception") is signal 8

# 7 Ways to Generate Signal

1. `kill(2)` system call

2. `kill(1)` program ... just calls `kill(2)`

3. Hardware exception (e.g., `SIGSEGV`, `SIGBUS`, `SIGFPE`, `SIGILL`)

4. OS condition (e.g., `SIGURG`, `SIGPIPE`)

5. Shell translates certain keys into `kill(2)`: `SIGINT`, `SIGQUIT`

6. Process can signal itself with `raise(3)`

7. Process can `SIGALRM` itself with `alarm(3)`/`setitimer(2)`

# Signal Handling

Signal can be:

- Blocked—keep pending until signal unblocked
- Ignored—delivered & immediately dropped
- Handled—delivered & handled

To block: `sigprocmask(2)`

To ignore or handle: `sigaction(2)`

# Sigprocmask, I

Recall: in computer science, a **mask** is a sequence of bits where each bit specifies some action/information

Signal set ("sigset_t") is type specifying mask value for all possible signals

There are several functions to manipulate signal sets

After calling manipulation functions, call sigprocmask:

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

# Sigprocmask, II

"`how`" argument may be:

- `SIG_BLOCK`—add specified signals to those being blocked
- `SIG_UNBLOCK`—subtract specified signals from those being blocked
- `SIG_SETMASK`—specified signals are exactly those to be blocked

Some signals cannot be blocked (`SIGKILL`, `SIGSTOP`)

# Sigaction, I

```
int sigaction(int signal,
              struct sigaction *act,
              struct sigaction *oldact)
```

# Sigaction, II

Important fields of `struct sigaction`:

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

`void (*sa_handler)(int)` — a function, `SIG_IGN` (ignore), or `SIG_DFL` (default)

`sigset_t sa_mask` — signals to block during execution of function

# Signal Delivery

Usually signal delivered *asynchronously* — program is interrupted, signal handler function runs to completion, then program resumes

Sometimes want *synchronous* delivery — i.e., wait for signal

There are 3 ways to wait for signal:

1. `pause(3)` — don't use this!
2. `sigwait(2)`
3. `sigsuspend(2)`

# The problem with pause(3)

`pause`'s semantics: wait for signal

Possible for signal to be lost & program to get stuck:

1. Unblock signal
2. Signal delivered
3. `pause(3)` called — will never return

Programmer did not want #2 to happen between consecutive program statements #1 and #3 but unluckily it did

Need **atomic** unblock-and-pause operation

# Atomic Unblock-and-Pause

`sigwait` — wait for signal in set

`sigsuspend` — wait for signal NOT in set

`sigwait(sigset_t set, int *signal)`:

- Unblocks all signals in argument set
- Returns when one of them is delivered
- Out parameter indicates which

`sigsuspend(sigset_t mask)`:

**1.** Save blocked signal mask

**2.** Replace blocked signal mask with argument mask

**3.** Wait until some unblocked signal occurs

**4.** Restore previous blocked signal mask

**5.** Return

# Signals & Concurrency, I

Signal handling raises some of same issues as threads

Reason: signal handler is preemptively scheduled

For instance:

- Function is partially complete
- Signal is delivered & handled
- Handler calls SAME function, which runs to completion
- Function resumes and runs to completion

Function in this example is “reentered”

Function could be user-written or library

# Signals & Concurrency, II

*Only certain library functions can be called by signal handler!*

These are “signal safe”

Prohibited: any function that accesses (reads or writes) static data

Prohibited: `malloc`/`free`, `fprintf`

# Signals & Concurrency, III

Another problem: static variable `errno`

Every system call potentially writes it

Example:

**1.** Program makes system call

**2.** Call fails, `errno`=12

**3.** Before program can examine `errno`, signal occurs & handler runs

**4.** Handler makes system call

**5.** Call fails, `errno`=3

**6.** Program examines `errno` – sees 3 instead of 12

Solution: handler should save & restore `errno`

# Reentrancy

Code that is “signal safe” is **reentrant**

Reentrant code can be safely “re-entered”

With reentrant function, this scenario is OK:

**1.** Function is partially complete

**2.** Signal is delivered & handled

**3.** Handler calls SAME function, which runs to completion

**4.** Function resumes and runs to completion

Handler “re-entered” the function in step #3

# How to Write A Reentrant Function

1. Function only reads, never writes — not practical!

2. Function writes only activation-specific variables

In other words: function writes only local variables (allocated on stack), not globals (such as `errno`) nor static locals

# How to Spot A Non-Reentrant Function

In general, must understand function *implementation* to determine if it is reentrant

But some non-reentrant functions can be spotted from interface alone

Giveaway: returns pointer to static/global

Example: `asctime(3)`

There is now also `asctime_r(3)` — caller must supply buffer to accept return value

Others: `localtime(3)`, `gmtime(3)`, `ctime(3)`, `strtok(3)`, `readdir(3)`