signals

characteristics

- a signal is a software interrupt
- signals notify a process that an event has occurred
- hardware can also create events, which are detected by the HW, received by the kernel and sent to the process as signals (e.g., division by zero)
- **two main categories**
  - *Standard POSIX reliable signals (POSIX.1 standard)*
  - *POSIX real-time signals (POSIX.1b standard)*
    - provide additional signals
    - signals are queued: if multiple instances of a signal are sent, the signal will be received many times
    - data may be sent together with the signals
- each signal is identified by a positive integer, defined in `<signal.h>`
- a signal is **generated by an event** and **delivered to a process**. in between the signal is said **to be pending**
- a pending signal is delivered to a process as soon as it is scheduled to run, or immediately if it is already running

some signals

name	description	default action
`SIGALARM`	timer expired	terminate
`SIGBUS`	memory access error	core dumped + terminate
`SIGINT`	terminal interrupt character (Ctrl + C)	terminate
`SIGTSTP`	terminal stop character (Ctrl + Z)	stop process
`SIGPIPE`	program tried to write to a closed channel	
`SIGTERM`	system requests the application to terminate	
`SIGKILL`	sure kill signal	terminate

signal delivery

default results

- the signal is ignored - discarded by the kernel and has no effect on the process
- the process is terminated
- a core dump file is generated and the process is terminated
- the process is stopped

what can the process do?

- **ignore the signal**
  - signal is discarded
- **block the signal**
  - signals are stored in a queue (not always) until the process unblocks them, and then they are delivered
- **handle (catch) the signal**
  - signals are redirected to a signal handler which is called
- **none os the above**
  - let the default action apply

> ⚠ **Warning**
>
> some signals cannot be ignored or handled
> `SIGKILL` always terminates the process
> `SISTOP` always stops a process

> 🖉 **Note**
>
> - When a process starts, signals are on their “default behaviour”. Some are ignored, but most are in the “non-handled, non-blocked nor ignored state”. If a signal occurs, the process will die.
> - Multiple occurrences of the same signal while a process has blocked it may not result in multiples deliveries when the signal is unblocked
>   - POSIX.1 allows the delivery of the signal either once or more than once
>   - Only queued signals guarantee the delivery of multiple occurrences
>   - Signals are always queued when POSIX real-time extensions are used

basic functions

```c
typedef void (*sighandler_t)(int);    // prototype of the handler routine

// receives an integer and returns nothing
sighandler_t signal(int signo, sighandler_t handler);
```

- signo: signal number/name
- handler:
    - can be a function: address to a function that will handle the signal
    - can be `SIG_IGN`: ignores the signal
    - can be `SIG_DFL`: restaures the dafault handler os the signal
- returns the precious handler of the signal specified or SIG_ERR on error
- It is the original API for setting the disposition of a signal and has a simple interface. However, there are variations in the implementation of signal() across different versions of Unix. The function sigaction() does not have that problems and has more functionalities - use it instead of signal(). In some systems signal() is based on sigaction() - in this case it presents no problems

```c
int kill(pid_t pid, int signo);
// sends a signal to a certain process
```

- pid: if 0, sends to all processes in the current sender process gorup (including the sender)
- signo: if 0 (`null signal`), no signal is sent and `kill()`only checks if a process can be signaled - can be used to test if a process exists
- returns 0 on success, or -1 on error

```c
int raise(int signo);
// send a signal to the process itself
// equivalent to kill(getpid(), signo);
```

- returns 0 on success, or non-zero on error

Note

if using pthreads, `raise` can be implemented as `pthread_kill(pthread_self(), signo);`

```
int pause(void);
// suspends the execution of the process until a signal is received
```

```
- always returns -1 with `errno` set to `EINTR`
```

signal sets functions

- a signal set represents multiple signals
- is used in signal functions that need signal sets
- return 0 if successful, -1 on error

```
int sigemptyset(sigset_t *set);
// initialize the set and exclude all signals

int sigfillset(sigset_t *set);
// initialize the set and include all signals

int sigaddset(sigset_t *set, int signo);
// add signal to set

int sigdelset(sigset_t *set, int signo);
// delete signal from set
```

sigprocmask function

```
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict
oset);
// the signal mask of a process is the set of signals currently blocked
from delivery to that process
```

```
// sigprocmask() changes the process signal mask (`set` parameter),
retrieves the existing mask (`oset` parameter), or both
```

```
- how:
        - SIG_BLOCK - signals specified in the signal set pointed to by
set are added to the existing signal mask
        - SIG_UNBLOCK - signals in the signal set pointed to by set are
removed from the existing signal mask
        - SIG_SETMASK - the signal set pointed to by `set` replaces the
existing signal mask
- returns 0 on success, -1 on error
```

> **Info**
>
> use `pthread_sigmask()` to manipulate threads signal masks

sigaction function

```
int sigaction(int signo, const struct sigaction *act, struct sigaction
*oact);
// sets or examines the action associated to the signal specified
```

```
- act: if non-NULL, it modifies the action
- oact: if non-NULL, the system returns the previous action fot the signal
- returns 0 on success, or -1 on error
```

```
struct sigaction {
        void (*sa_handler)(int);    // addr of signal handler (or
SIG_IGN/SIG_DFL)
        sigset_t sa_mask;           // signals to block during handler
        int sa_flags;               // signal options (see manual)
        void (*sa_restorer)(void);  // not for application use
};
```

```
- 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:
        - defines the signals to be blocked during the invocation of the
handler sa_handler
        - the signal that caused the handler to be invoked is
automatically added to the mask (is also blocked)
        - when the handler returns the sa_mask is removed
        - it specifies the set of signals that cannot interrupt this
handler
```

other functions

- other functions
  - `sigismember()` : test for membership of a set.
  - `sigandset()` : intersects sets (and)
  - `sigorset()` : union of sets (or)
  - `sigpending()` : to determine which signals are pending for a process
  - `killpg()` : send signal to a process group
  - ...
- for threads
  - `pthread_sigmask()` : signal mask for threads
  - `sigwait()` : wait for a signal
  - `pthread_kill()` : send a signal to a thread

some considerations

- after calling a standard function, it may return -1 indicating an error
  - `errno == EINTR` means that a certain routine was interrupted and has to be tried again.
  - other routines return other things.
  - if you are using signals, you must protect them against all that!

- What do you think it will happen if you receive a signal inside a signal handler??
  - in most systems, upon entering a signal handling routine, all signals of that type become blocked (i.e. they are queued - for “normal” signals, a finite set of them are queued (typically 1); for “real time signals”, all are...)

- the other signals are still processed asynchronously if they arrive.
- this behaviour is not consistent across systems. In fact, in some systems, that signal type resets to its default behaviour. This means that if, meanwhile, the program receives a signal of the same type it may die! On that type of system, the first thing that you must do is to once again set the signal handler.
- the new POSIX routines address this. Also, most system nowadays do not reset the signal handler.

	Architecture					
Signal	x86/ARM most others	Alpha/ SPARC	MIPS	Standard that specified the signal	Action	Comment
SIGABRT	6	6	6	P1990	Core	Abort signal from abort(3)
SIGALRM	14	14	14	P1990	Term	Timer signal from alarm(2)
SIGBUS	7	10	10	P2001	Core	Bus error (bad memory access)
SIGCHLD	17	20	18	P1990	Ign	Child stopped or terminated
SIGCLD	-	-	18	-	Ign	A synonym for SIGCHLD
SIGCONT	18	19	25	P1990	Cont	Continue if stopped
SIGEMT	-	7	7	-	Term	Emulator trap
SIGFPE	8	8	8	P1990	Core	Floating-point exception
SIGHUP	1	1	1	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	4	4	4	P1990	Core	Illegal Instruction
SIGINFO	-	29/-	-	-		A synonym for SIGPWR
SIGINT	2	2	2	P1990	Term	Interrupt from keyboard (^C)
SIGIO	29	23	22	-	Term	I/O now possible (4.2BSD)
SIGIOT	6	6	6	-	Core	IOT trap. A synonym for SIGABRT
SIGKILL	9	9	9	P1990	Term	Kill signal (cannot be caught, blocked or ignored)
SIGLOST	-	-/29	-	-	Term	File lock lost (unused)
SIGPIPE	13	13	13	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL				P2001	Term	Pollable event (Sys V); synonym for SIGIO
SIGPROF	27	27	29	P2001	Term	Profiling timer expired
SIGPWR	30	29/-	19	-	Term	Power failure (System V)
SIGQUIT	3	3	3	P1990	Core	Quit from keyboard
SIGSEGV	11	11	11	P1990	Core	Invalid memory reference (segmentation violation)
SIGSTKFLT	16	-	-	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	19	17	23	P1990	Stop	Stop process (cannot be caught, blocked or ignored)
SIGSYS	31	12	12	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	15	15	15	P1990	Term	Termination signal
SIGTRAP	5	5	5	P2001	Core	Trace/breakpoint trap
SIGTSTP	20	18	24	P1990	Stop	Stop typed at terminal (^Z)
SIGTTIN	21	21	26	P1990	Stop	Terminal input for background process
SIGTTOU	22	22	27	P1990	Stop	Terminal output for background process
SIGUNUSED	31	-	-	-	Core	Synonymous with SIGSYS
SIGURG	23	16	21	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	10	30	16	P1990	Term	User-defined signal 1
SIGUSR2	12	31	17	P1990	Term	User-defined signal 2
SIGVTALRM	26	26	28	P2001	Term	Virtual alarm clock (4.2BSD)
SIGWINCH	28	28	20	-	Ign	Window resize signal (4.3BSD, Sun)
SIGXCPU	24	24	30	P2001	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	25	25	31	P2001	Core	File size limit exceeded (4.2BSD); see setrlimit(2)

P1990 = signal described in POSIX.1-1990 standard

- **signals and threads**
  - Some conflicts exist between the Unix signal model that was based on processes and POSIX threads model.
  - Signal actions are process-wide – if a signal received by a thread has as default action to stop or terminate, then all threads in the process are stopped or terminated.

    - Actions are shared between all threads – when an action is changed in a thread, all threads response to the signal is changed.

    - A signal may be directed to a specific thread (thread-directed):
        - in the case of synchronous signals that result from a specific thread execution, which are received by that same thread (e.g., SIGFPE – floating point exception);
        - by using `pthread_kill()` to enable a thread to send a signal to other thread;

        - A signal is process-directed:
            - if sent from a process to other process;
        - in the case of asynchronous signals, which are received by any thread that has not blocked them (`pthread_sigmask`);

    - When a signal is delivered to a multithreaded process just one thread catches it
    - Each thread may have its own thread mask
- Signals from the Linux shell
    - Using Linux command line
        - List signals: `kill –l`
        - Send a `SIGALRM` to a process with `PID = 76543`: `kill –SIGALRM 76543`
        - Send a signal to all processes with the name `myproc`: `killall –SIGKILL myproc`
    - CTRL + C
        - When ‘^C’ is pressed the terminal will send a SIGINT to the foreground process group of the terminal