signal are simple form of interprocess-communication

signals can be generated from a variety of sources

- from another process via `kill()`
- from the operating system (e.g. timer)
- from within the process (e.g. system call)
- from a fault in the process (e.g. div-by-zero)

processes can define how they want to handle signals

- using the `signal()` library function (simple)
- using the `sigaction()` system call (powerful)

signal SIGKILL always terminates receiving processes

only owner of a processes can send signal to it
Default handling of signal can be:

- **Term** ... terminate the process
- **Ign** ... ignored; the signal does nothing
- **Core** ... terminate the process and dump memory image to file named core
- **Stop** ... pause the process
- **Cont** ... continue the process (if paused)

Processes can choose to ignore a signal.

Processes can set a custom *signal handler* for signal.

... except for SIGKILL and SIGSTOP, which cannot be caught, blocked, or ignored.

See `man 7 signal` for details of signals and default handling.
Operating System-Generated Signals

Signals from internal process activity, e.g.

- SIGILL ... illegal instruction \(\text{ Term }\) by default
- SIGABRT ... generated by \textit{abort()} \(\text{ Core }\) by default
- SIGFPE ... floating point exception \(\text{ Core }\) by default
- SIGSEGV ... invalid memory reference \(\text{ Core }\) by default

Signals from external process events, e.g.

- SIGHUP ... hangup detected on controlling terminal/process
- SIGINT ... interrupt from keyboard (ctrl-c) \(\text{ Term }\) by default
- SIGPIPE ... broken pipe \(\text{ Term }\) by default
- SIGCHLD ... child process stopped or died \(\text{ Ign }\) by default
- SIGTSTP ... stop typed at tty (ctrl-z) \(\text{ Stop }\) by default
Signal Handlers

*Signal Handler* = a function invoked in response to a signal

- knows which signal it was invoked by
- needs to ensure that invoking signal (at least) is blocked
- carries out appropriate action; may return
 signaling() — installing a signal handler, the old way

#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

- old way to create signal handler - do not use in new code
- set how to handle a signal signum (e.g. SIGINT)
- handler can be one of ...
  - SIG_IGN ... ignore signal signum
  - SIG_DFL ... use default handler for signum
  - a user-defined function for signum signals
    - function type must be void (int)
- returns previous value of signal handler, or SIG_ERR
#include <signal.h>

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

- set how to handle a signal `signum` (e.g. SIGINT)  
- `act` defines how signal should be handled  
- `oldact` saves a copy of how signal was handled  
- if `act->sa_handler` == SIG_IGN, signal is ignored  
- if `act->sa_handler` == SIG_DFL, default handler is used  
- on success, returns 0; on error, returns -1 and sets `errno`

For much more information: man 2 sigaction
Details on struct sigaction...

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

- **void (**sa_handler)(int)**
  - pointer to a handler function, or SIG_IGN or SIG_DFL
- **void (**sa_sigaction)(int, siginfo_t *, void **)**
  - pointer to handler function; used if SA_SIGINFO flag is set
  - allows more context info to be passed to handler
- **sigset_t sa_mask**
  - a mask, where each bit specifies a signal to be blocked
- **int sa_flags**
  - flags to modify how signal is treated
    - (e.g., don’t block signal in its own handler)
Details on siginfo_t ...

typedef struct {
    int    si_signo;  /* signal number of signal being handled */
    int    si_code;   /* signal code - more information about why */
    pid_t  si_pid;    /* process ID of sending process */
    uid_t  si_uid;    /* user ID of owner of sending process */
    void  *si_addr;   /* address of faulting instruction */
    int    si_status; /* exit value for process termination */
    /* ... */
} siginfo_t;

System-dependent; these are (a subset of) mandated fields.
Waiting for an event ... the dumb way

```
#include <signal.h>

void signal_handler(int signum) {
    printf("signal number %d received\n", signum);
}

int main(void) {
    struct sigaction action = {.sa_handler = signal_handler};
    sigaction(SIGUSR1, &action, NULL);
    printf("I am process %d waiting for signal %d\n", getpid(), SIGUSR1);
    // loop waiting for signal
    // bad consumes CPU/electricity/battery
    // sleep would be better
    while (1) {
    }
}
```

source code for busy_wait_for_signal.c

https://www.cse.unsw.edu.au/~cs1521/21T2/
#include <unistd.h>

unsigned int sleep(unsigned int seconds);

- sleep() suspended the caller for **seconds** of real-time
- efficient way to wait for an event such as a signal
- allows operating system to run other processes
Example: waiting for an event

```c
#include <signal.h>

void signal_handler(int signum) {
    printf("signal number %d received\n", signum);
}

int main(void) {
    struct sigaction action = {.sa_handler = signal_handler};
    sigaction(SIGUSR1, &action, NULL);
    printf("I am process %d waiting for signal %d\n", getpid(), SIGUSR1);
    // suspend execution for 1 hour
    sleep(3600);
}
```

source code for wait_for_signal.c
```c
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);

- send signal number `sig` to process number `pid`
- if successful, return 0; on error, return -1 and set `errno`

```c
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <signal> <pid>\n", argv[0]);
        return 1;
    }
    int signal = atoi(argv[1]);
    int pid = atoi(argv[2]);
    kill(pid, signal);
}
```
```c
#include <signal.h>
int main(void) {
    // catch SIGINT which is sent if user types cntrl-d
    struct sigaction action = {.sa_handler = SIG_IGN};
    sigaction(SIGINT, &action, NULL);
    while (1) {
        printf("Can't interrupt me, I'm ignoring ctrl-C\n");
        sleep(1);
    }
}
```

source code for ignore_control.c
Example: a simple signal handler

```c
#include <signal.h>
void ha_ha(int signum) {
    printf("Ha Ha!\n"); // I/O can be unsafe in a signal handler
}

int main(void) {
    // catch SIGINT which is sent if user types cntrl-d
    struct sigaction action = {.sa_handler = ha_ha};
    sigaction(SIGINT, &action, NULL);
    while (1) {
        printf("Can't interrupt me, I'm ignoring ctrl-C\n");
        sleep(1);
    }
}
```

source code for laugh_at_control_c.c
Example: another simple signal handler

```c
#include <signal.h>

int signal_received = 0;

void stop(int signum) {
    signal_received = 1;
}

int main(void) {
    // catch SIGINT which is sent if user types ctrl-C
    struct sigaction action = {.sa_handler = stop};
    sigaction(SIGINT, &action, NULL);
    while (!signal_received) {
        printf("Type ctrl-c to stop me\n");
        sleep(1);
    }
    printf("Good bye\n");
}
```

source code for stop_with_control_c.c

https://www.cse.unsw.edu.au/~cs1521/21T2/COMP1521 21T2 — Signals
Example: catching an internal error with a signal handler

```c
#include <signal.h>
#include <stdlib.h>

void report_signal(int signum) {
    printf("Signal %d received\n", signum);
    printf("Please send help\n");
    exit(0);
}

int main(int argc, char *argv[]) {
    struct sigaction action = {.sa_handler = report_signal};
    sigaction(SIGFPE, &action, NULL);
    // this will produce a divide by zero
    // if there are no command-line arguments
    // which will cause program to receive SIGFPE
    printf("%d\n", 42/(argc - 1));
    printf("Good bye\n");
}
```

source code for catch_error.c