**Linux/Unix Signals**

- Signals 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 process can send signal to it.

**Signal Handling**

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 `SIGKILL` and `SIGSTOP` 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 (Term by default).
- `SIGABRT` ... generated by `abort()` (Core by default).
- `SIGFPE` ... floating point exception (Core by default).
- `SIGSEGV` ... invalid memory reference (Core by default).

Signals from external process events, e.g.
- `SIGHUP` ... hangup detected on controlling terminal/process.
- `SIGINT` ... interrupt from keyboard (ctrl-c) (Term by default).
- `SIGPIPE` ... broken pipe (Term by default).
- `SIGCHLD` ... child process stopped or died (Ign by default).
- `SIGTSTP` ... stop typed at tty (ctrl-z) (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.
Installing A Signal Handler - the Old Way

```c
#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

Installing A Signal Handler - the New Way

```c
#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

Signal Handlers

Details on struct `sigaction` ...

- `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` ...

- `si_signo` ... signal being handled
- `si_errno` ... any `errno` value associated with signal
- `si_pid` ... process ID of sending process
- `si_uid` ... user ID of owner of sending process
- `si_status` ... exit value for process termination
- etc. etc. etc.

For more details: bits/types/siginfo_t.h (system-dependent)
Waiting For an Event - The Dumb Way

```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);
    // bad consumes CPU/electricity/battery
    // sleep would be better
    while (1) {
    }
}
```

source code for busy_wait_for_signal.c

Waiting For an Event - The Smart Way

```c
#include <unistd.h>
n 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

```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);
    sleep(3600);
}
```

source code for wait_for_signal.c

sending signals with kill()

```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);
}
```

source code for send_signal.c
#include <signal.h>

```c
int main(void) {
    // catch SIGINT which is sent if user types ctrl-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.c

---

#include <signal.h>

```c
#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