Linux/Unix Signals

- Signals are simple forms 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., divide-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 the owner of a process can send a signal to it.

Signal Handling

Default handling of a signal can be:

- **Term** ... terminate the process.
- **Ign** ... ignored; the signal does nothing.
- **Core** ... terminate the process and dump memory image to a 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 \textit{(Term by default)}
- SIGABRT ... generated by \texttt{abort()} \textit{(Core by default)}
- SIGFPE ... floating point exception \textit{(Core by default)}
- SIGSEGV ... invalid memory reference \textit{(Core by default)}

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

Signal Handlers

\textit{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

\begin{center}
\textbf{Signal}
\end{center}

\begin{center}
\textbf{signal()} — Installing a signal handler, the old way
\end{center}

```
#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 \texttt{signum} (e.g. SIGINT)
- \texttt{handler} can be one of...
  - SIG_IGN ... ignore signal \texttt{signum}
  - SIG_DFL ... use default handler for \texttt{signum}
  - a user-defined function for \texttt{signum} signals
  - function type must be \texttt{void \(\texttt{(int)}\)}
- returns previous value of signal handler, or SIG_ERR
sigaction() — installing a signal handler, the new way

#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 ...

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)

Signal Handlers

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

```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);
    // loop waiting for signal
    // 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>

unsigned int sleep(unsigned int seconds);

- sleep() suspended the caller for **seconds** of real-time
- efficient way to wait for an event such as an 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
#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`

https://www.cse.unsw.edu.au/~cs1521/21T2/COMP1521 21T2 — Signals 13 / 17

Example: ignoring a signal

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

https://www.cse.unsw.edu.au/~cs1521/21T2/COMP1521 21T2 — Signals 14 / 17

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 ctrl-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`

https://www.cse.unsw.edu.au/~cs1521/21T2/COMP1521 21T2 — Signals 15 / 17
#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 cntrl-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");
}

Example: another simple signal handler

#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");
}