Signals

Signals are software interrupts

- Signals are *software interrupts* caused by HW or SW
  - HW event *(user) - HW - kernel - process*
    - STRG-C -> SIGINT *
    - DIV by zero -> SIGFPE
  - SW event *(user) - process - kernel - process*
    - kill(1,2) function
    - write to pipe with no reader -> SIGPIPE
    - termination of countdown -> SIGALRM

* Every signal has a name with prefix SIG (see signal(3) on BSD and signal(7) on Linux) defined in <signal.h>

Signals are asynchronous events

- Signals are *asynchronous events*, i.e.,
  - Signals occur at what happens to be random times to the process.
  - The process can not simply test a variable (such as errno) to see whether a signal has occured.
  - Instead, the process has to tell the kernel *"if and when this signal occurs, do the following."*
  - -> signal handler

Signals are an early concept

- Signals *have been provided since the early versions* of UNIX
  - First versions were unreliable
    - Signals could get lost
    - Difficult to turn off selected signals when executing critical regions of code.
  - POSIX.1 standardized the *reliable-signal routines*.

Disposition of signals (signal handlers)

- We can tell the kernel to do one of three things:
  - **Ignore the signal.**
  - **Catch the signal.**
  - **Let the default action apply.**

Disposition of signals (assoc. action)

- **Ignore the signal.**
  - Works for most signals.
  - SIGKILL and SIGSTOP can never be ignored.
- **Catch the signal.**
  - Tell the kernel to call a function of ours.
  - Our function can do whatever we want to handle the exception.
- **Let the default action apply.**
  - Every signal has a default action.
  - Often the default action is to terminate the process.

Signal Handlers

```
#include <signal.h>

void (*
signal(int sig, void (*func)(int)))(int);

/* or in the equivalent but easier to read typedef'd version: */

typedef void (*sig_t) (int);

sig_t
signal(int sig, sig_t func);
```

- signal(3) is the 'historic' way of registering a signal handler.
- It often is a simplified interface to the more general, 'new' sigaction(2) facility.
- Beware to not use signal(3) on systems, were this is not the case and use sigaction(2) from the start there.

Signal Handlers

```
#include <signal.h>

void (*
signal(int sig, void (*func)(int)))(int);

/* or in the equivalent but easier to read typedef'd version: */

typedef void (*sig_t) (int);

sig_t
signal(int sig, sig_t func);
```

- 1. argument **sig** is the name of the signal the handler should be installed for.
- 2. argument **func** (the signal handler)
  - SIG_DFL - default action *
  - SIG_IGN - ignore the signal **
  - Address of the handler function in our program to be executed once signal is delivered.
- Returns: previous signal handler if OK, SIG_ERR on failure.

* See signal(2) BSD, signal(7) on Linux for default actions.
** All signals but SIGSTOP and SIGKILL. They always use their default action.

Problems using historic signal function

- Determination of currently installed signal handler changes its status to default action.
- Signal is reset to its default action each time the signal occured.
- No way to block a signal (and handle it at a later time) *
- -> **New reliable signal concept**
  - sigaction(2) and friends, i.e., sig*(2)
  - Helmut Herold: Linux/Unix Systemprogrammierung, Chapter 13 - Signale (basically a german version of Rago/Stevens, Adv. Prog. in the UNIX Environment)
    IP restricted copy on our webserver

* *"Prevent the following signals from occurring, but remember if they do occur"* != ignoring signals

Signals and process creation

- fork(2)
  - child inherits all signal handlers
    (child starts with a copy of parent's memory image)
- exec(3)
  - default action applies
    (former addresses of handlers probably have no meaning)
  - However, formerly ignored signals are still ignored in the exec'ed process.

Reentrant functions

- Situation: A signal being caught is handled by a process.
- Normal sequence of instructions is temporarily interrupted.
- Process continues to execute signal handler.
- Once signal handler returns normal sequence is continued.

- In the signal handler, we can not tell when process has been interrupted.
- Assume it was in the middle of malloc(3) ...
- -> Only reentrant functions are allowed in signal handlers. *

* sigaction(2) on BSD, Tab 13.4 in HH

alarm(2) function

- alarm allows us to set a timer that will expire at a specified time in the future (argument).
- When the timer expires, the SIGALRM signal is generated.
- If we ignore or do not catch the signal, its default action is to terminate the process.

```
#include <unistd.h>

unsigned int
alarm(unsigned int seconds);
```

- Returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

Reliable-Signals Terminology

- A signal is *generated* for a process (or *sent* to a process) when the event that causes the signal occurs.
  - hardware exception
  - software condition
  - terminal-generated signal
  - call to the kill(2) function
- A signal is *delivered* to a process when the action for a signal is taken.
- During the time between the generation and its delivery, the signal is said to be *pending*.

Reliable-Signals Terminology (cont.)

- A process has the option of *blocking* the delivery of a signal.
- *signal mask* defines the set of signals currently blocked.
  - Think of this mask as having a bit for each possible signal.
  - If set the signal is blocked.
- *signal set* denotes POSIX.1 data type sigset_t
  - Numbers of signals may exceed the number of bits in an integer. POSIX.1 defines a data type, sigset_t, that holds a *signal set*. The *signal mask* is stored in one of these *signal sets*.

Small example

Further reading

- For Linux (GNU C library manual)
  - http://www.gnu.org/software/libc/manual/html_mono/libc.html#Signal-Handling
- HH, Chap 13
  - http://www.inf.uni-konstanz.de/dbis/teaching/ss09/os/restricted/hh13_signals.pdf
- APUE, Chap 10