Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

For more information please consult

'Advanced Programming in the UNIX® Environment, 3rd Edition, W. Richard Stevens and Stephen A. Rago, Addison Wesley'

Sections 10.1–10.10

Signals

- Signals are **software interrupts** which provide a way of handling **asynchronous events** (i.e., events that occur at what appear to be random times to the process)
  - A user at a terminal typing the interrupt key to stop a program
  - The next program in a pipeline terminating prematurely
  - A process trying to access an invalid memory address

- Every signal has a name that corresponds to a **positive integer** (the **signal number**)
  - All signal names begin with **SIG** and are all defined in **<signal.h>**
  - Two user-defined signals, named **SIGUSR1** and **SIGUSR2**, are available for use in application programs

Signals

- Every signal has a default action which for most signals is to terminate the process. We can redefine these default actions by telling the kernel to:
  - **Ignore the signal**
  - **Catch the signal**
- Ignoring some of the signals that are generated by a hardware exception (such as illegal memory reference or divide by 0) can lead a process to an undefined behavior
- Two signals, **SIGKILL** and **SIGSTOP**, cannot be ignored
  - The reason for this is to provide the kernel with a surefire way of either killing or stopping any process

Signals

- Example of signals are:
  - **SIGINT** is generated by the terminal when we type the interrupt key (often DELETE or Control-C)
  - **SIGTSTP** is generated by the terminal when we type the suspend key (often Control-Z)
  - **SIGSEGV** indicates that the process has made an invalid memory reference
  - **SIGTERM** is the default termination signal sent by the kill() command
  - **SIGKILL** provides the system administrator with a sure way to kill any process (this signal cannot be caught or ignored)
  - **SIGSTOP** stops a process like **SIGTSTP** but it cannot be caught or ignored
  - **SIGABRT** is generated by calling the abort function
  - **SIGALRM** is generated when a timer set with the alarm function expires
  - **SIGCHLD** is generated whenever a child process terminates or stops (by default, it is ignored by the parent)

Setting a Signal Handler

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
// * redefines the default action (signal handler) for a
//   signal
// * the new default action can be:
//   - SIG_IGN, to ignore the signal
//   - SIG_DFL, to set the default signal action
//   - a user-defined signal handler function to be called
//     when the signal occurs
// * returns the previous signal handler function if
//   successful, SIG_ERR on error
```

Setting a Signal Handler

```
#include <signal.h>

#define SIG_ERR (void (*)()) -1
#define SIG_DFL (void (*)()) 0
#define SIG_IGN (void (*)()) 1
// * possible signal handler declarations
```

Setting a Signal Handler: Example

```
void sig_usr(int signo) {
  if (signo = = SIGUSR1) printf("received SIGUSR1\n");
}
int main() {
  if (signal(SIGUSR1, sig_usr) = = SIG_ERR) { /* signal error */ }
  for ( ; ; ) pause();
}

Running example: $ ./a.out &     // start process in background
                 [1] 7216        // shell prints job number and PID
                 $ kill -USR1 7216  // send signal SIGUSR1
                 received SIGUSR1   // process catches the signal
                 $ kill 7216        // send signal SIGTERM
                 [1]+ Terminated ./a.out  // process terminates
```

Sending Signals

```
#include <signal.h>

int kill(pid_t pid, int signo);
// * to send a signal to a process
// * returns 0 if successful, 1 on error

int pause(void);
// * suspends the calling process until a signal is caught
// * returns if the signal handler executes and also returns
```

Handling Signals

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
// * sets a timer that will expire at a specified time in the
//   future, generating the SIGALRM signal when it expires
// * if the signal is ignored or didn't catch, its default
//   action is to terminate the process
// * returns 0 if no alarm is set or the number of seconds
//   until previously set alarm
```

Handling Signals: Example

```
void sig_alrm(int signo) {
  // nothing to do, just return to wake up the pause
}

int naive_sleep(int numsecs) {
  if (signal(SIGALRM, sig_alrm) = = SIG_ERR)
    return(numsecs);   // signal failed, return no slept time
  alarm(numsecs);      // start the timer
  pause();             // next caught signal wakes us up
  return(alarm(0));    // turn off timer, return unslept time
}
```