# System and Device Programming

## UNIX Signals

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

## Definition

- A **signal** is
  - A software interrupt
  - An asynchronous notification sent, by the kernel or by another process, to a process to notify it of an event that occurred
- Signals
  - Allow notify asynchronous events
    - Such as the occurrence of particular events (e.g., error conditions, memory access violations, calculation errors, illegal instructions, etc.)
  - Can be used as a limited form of inter-process communication

# Characteristics

- Available from the very first versions of UNIX
  - Originally managed in an unreliable way
- Standardized by the POSIX standard, they are now stable and relatively reliable
- Each signal has a name
  - Names start with **SIG...**
  - The file **signal.h** defines signal names
    - Unix FreeBSD, Mac OS X and Linux support 31 signals
    - Solaris supports 38 signals

# Main signals

The command **kill –l** displays a complete list of signals

| Name | Description |
|---|---|
| SIGABRT | Process abort, generated by system call abort |
| SIGALRM | Alarm clock, generated by system call alarm |
| SIGFPE | Floating-Point exception |
| SIGILL | Illegal instruction |
| SIGKILL | Kill (non maskable) |
| SIGPIPE | Write on a pipe with no reader |
| SIGSEGV | Invalid memory segment access |
| SIGCHLD | Child process stopped or exited |
| **SIGUSR1**<br>**SIGUSR2** | **User-defined signal ½**<br>**default action = terminate the process**<br>**Available for use in user applications** |

# Signal management

- Signal management goes through three phases
  - Signal generation
    - When the kernel or a source process causes an event that generate a signal
  - Signal delivery
    - A not yet delivered signal remains pending
    - At signal delivery a process executes the actions related to that signal

There is no signal queue. The kernel sets a flag in the process table

  - Reaction to a signal
    - To properly react to the asynchronous arrival of a given type of signal, a process must inform the kernel about the action that it will perform when it will receive a signal of that type

# Signal management

- Signal management can be carried out with the following system calls
  - **signal**
    - Instantiates a signal handler
  - **kill** (and **raise**)
    - Sends a signal
  - **pause**
    - Suspends a process, waiting the arrive of a signal
  - **alarm**
    - Sends a SIGALARM signal, after a preset time
  - **sleep**
    - Suspends the process for a specified amount of time (waits for signal SIGALRM)

The terms **signal** and **kill** are relatively inappropriate.
**signal** does not send a signal!!

# System call signal

```
#include <signal.h>

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

Return value:
The previous signal handler, on success
SIG_ERR, on failure

- Allow to instantiate a signal handler
  - Specifies the signal to be managed (**sig**)
  - The function use to manage it (**func**), i.e., the **signal handler**
- Arguments
  - **sig** indicates the type of signal to be caught
  - **func** specifices the address (i.e., pointer) to the function that will be executed when a

# Reaction to a signal

- The **signal** system call allows setting three different reactions to the delivery of a signal
  - Accept the default behavior
    - signal (SIGname, **SIG_DFL**)
  - Ignore signal delivery
    - signal (SIGname, **SIG_IGN**)
  - Catch the signal
    - signal (SIGname, **signalHandlerFunction**)

# Example

Setting a program to deal with 2 signals

```
...
void manager (int sig) {
  if (sig==SIGUSR1)
    printf ("Received SIGUSR1\n");
  else if (sig==SIGUSR2)
    printf ("Received SIGUSR2\n");
  else printf ("Received %d\n", sig);
  return;
}
...
int main () {
  ...
  signal (SIGUSR1, manager);
  signal (SIGUSR2, manager);
  ...
}
```

Same signal handler for more than one signal type

Both signal types must be declared

# Example

Asynchronous manipulation of SIGCHLD (with no wait)

```
static void sigChld (int signo) {
  if (signo == SIGCHLD)
    printf ("Received SIGCHLD\n");
  return;
}
...
signal(SIGCHLD, sigChld);
if (fork() == 0) {
  // child
  ...
  exit (i);
} else {
  // father
  ...
}
```

There is no
pid = wait (&code);

# System call kill

```
#include <signal.h>

int kill (pid_t pid, int sig);
```

- Send signal (**sig**) to a process or to a group of processes (**pid**)
- To send a signal to a process, you must have the rights
  - A **user** process can send signals only to processes having the same UID
  - The **superuser** can send signal to any process

# System call kill

- Arguments

| If pid is | Send sig |
|---|---|
| >0 | To process with PID equal to `pid` |
| ==0 | To all processes with GID equal to its GID (if it has the rights) |
| <0 | To all processes with GID equal to the absolute value of `pid` (if it has the rights) |
| ==–1 | To all processes (if it has the rights) |

- Return value
  - The value 0, if successful
  - The value –1, in case of error

```
int kill (pid_t pid, int sig);
```

# System call raise

```
#include <signal.h>

int raise (int sig);
```

- The **raise** system call allows a process to send a signal to itself
  - The system call
    - raise (sig)

    is equivalent to
    - Kill (getpid(), sig);

# System call pause

```
#include <unistd.h>

int pause (void);
```

- Suspends the calling process until a signal is received
- Returns after the completion of the signal handler
  - In this case the function returns -1

# System call alarm

```
#include <unistd.h>

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

Return value:
The number of
seconds remaining or 0

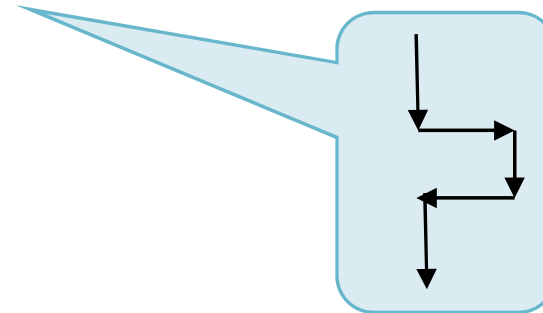
- Activate a timer (i.e., a count-down)
  - The **seconds** parameter specifies the count-down value (in seconds)
    - At the end of the countdown the signal **SIGALRM** is generated
  - If the system call is executed before the previous call has originated the corresponding signal, the count-down restarts from a new value

# Signal limitations

- Signals do not convey any information
- The **memory** of the "pending" signals is **limited**
  - Max one signal pending (sent but not delivered) per type
    - Forthcoming signals of the same type are lost
  - Signals can be ignored
- Signals require functions that must be **reentrant**
- Produce **race conditions**
- Some limitations are avoided in POSIX.4

# Reentrant functions

- A signal has the following behavior
  - The interruption of the current execution flow
  - The execution of the signal handler
  - The return to the standard execution flow at the end of the signal handler



- Consequently
  - The kernel **knows** where a signal handler returns, but the signal handler **does not know**
  - The signal handler must operate in a **compatible way** with the original execution flow

# Race conditions

- **Race condition**
  - The result of more concurrent processes working on common data depends on the execution order of the processes instructions
- Concurrent programming is subject to race conditions
  - Using signals increases the probability of race conditions
- Race condition should be avoided at all costs