ARCOS Group
Universidad Carlos III de Madrid

Lesson 3
Signals, exceptions and pipes

Operating Systems
Computer Science and Engineering

To remember…

Before classes	Class	After class

Prepare the prerequisites.

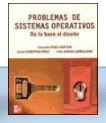
Study the material associated with the **bibliography**: slides alone are not enough.
Please ask questions (especially after study).

Exercising skills:

- Perform all **exercises**.
- Carrying out the **practice notebooks** and **the practical exercises** progressively.

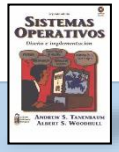
Recommended reading

Base



1. Carretero 2020:
   1. Cap. 5
2. Carretero 2007:
   1. Cap. 3.6 and 3.7
      Cap. 3.9 and 3.13

Suggested



1. Tanenbaum 2006:
   1. (es) Cap. 2.2
   2. (en) Cap.2.1.7
2. Stallings 2005:
   1. 4.1, 4.4, 4.5 and 4.6
3. Silberschatz 2006:
   1. 4

Contents

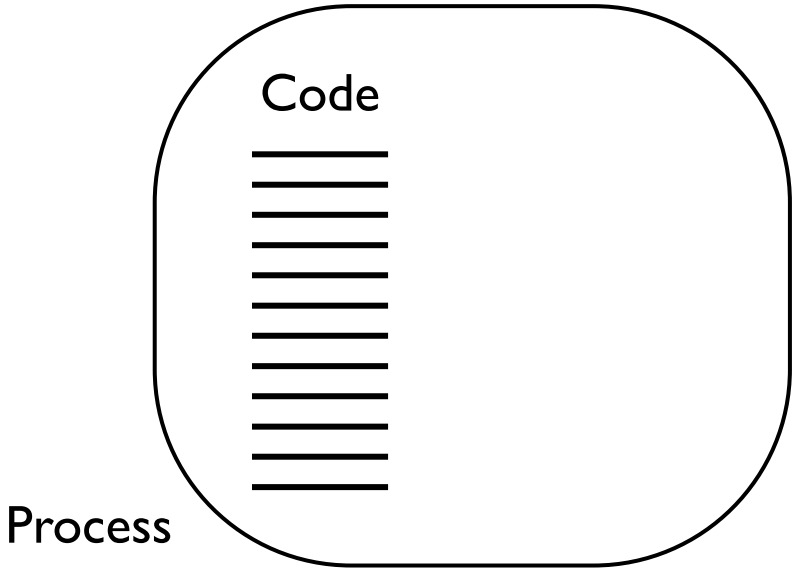
1. Signals and exceptions.
2. Timers.
3. Process environment.
4. Process communication with pipes.
   - Local message passing.

Contents

1. **Signals and exceptions**.
2. Timers.
3. Process environment.
4. Process communication with pipes.
   - Local message passing.

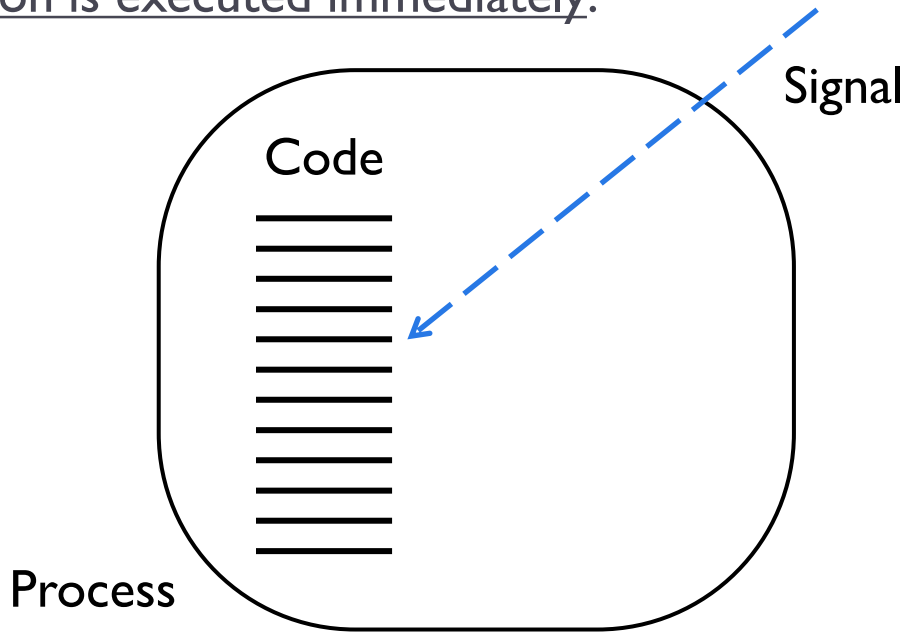
Signals: interruptions to process

- Mechanism to communicate to a process the occurrence of an event in an asynchronous manner and allow to react to such event.
  - Signals are interruptions to the process.
  - An associated processing function is executed immediately:



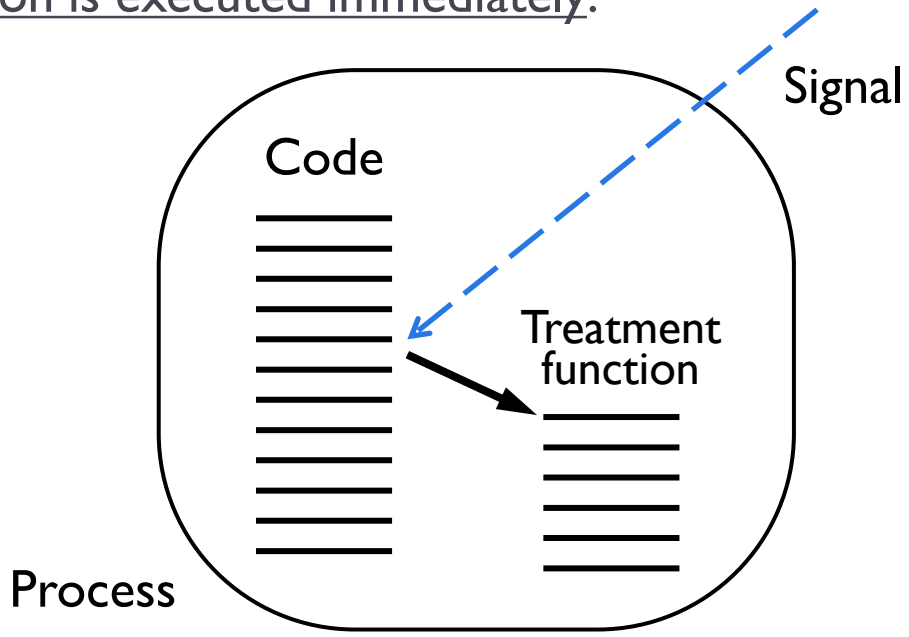
Signals: interruptions to process

- Mechanism to communicate to a process the occurrence of an event in an asynchronous manner and allow to react to such event.
  - Signals are interruptions to the process.
  - An associated processing function is executed immediately:



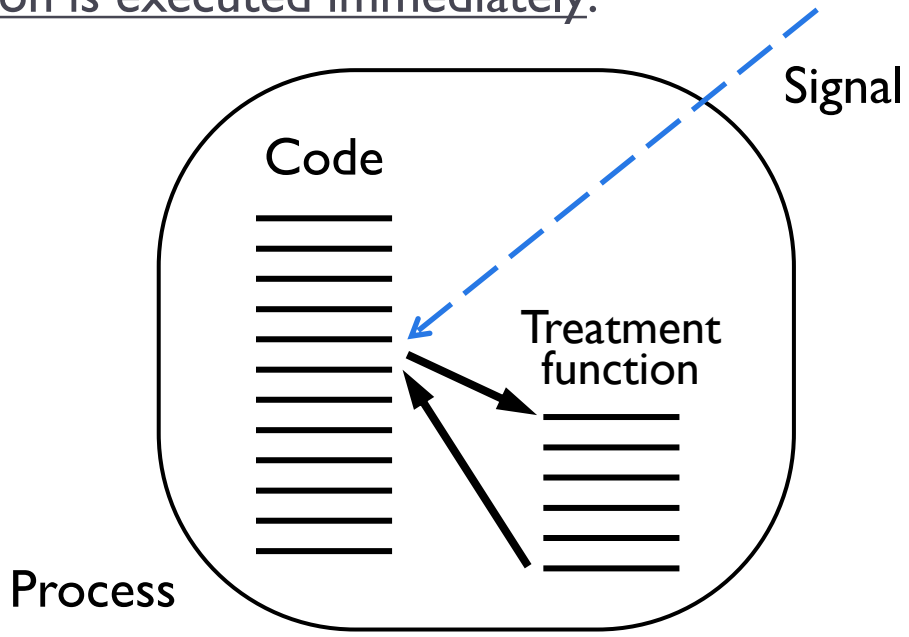
Signals: interruptions to process

- Mechanism to communicate to a process the occurrence of an event in an asynchronous manner and allow to react to such event.
  - Signals are interruptions to the process.
  - An associated processing function is executed immediately:



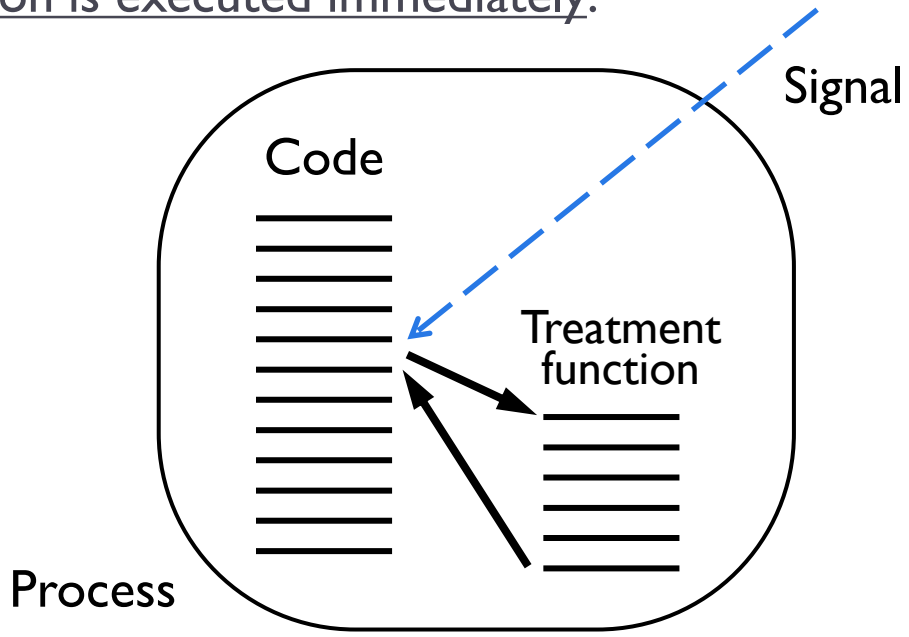
Signals: interruptions to process

- Mechanism to communicate to a process the occurrence of an event in an asynchronous manner and allow to react to such event.
  - Signals are interruptions to the process.
  - An associated processing function is executed immediately:



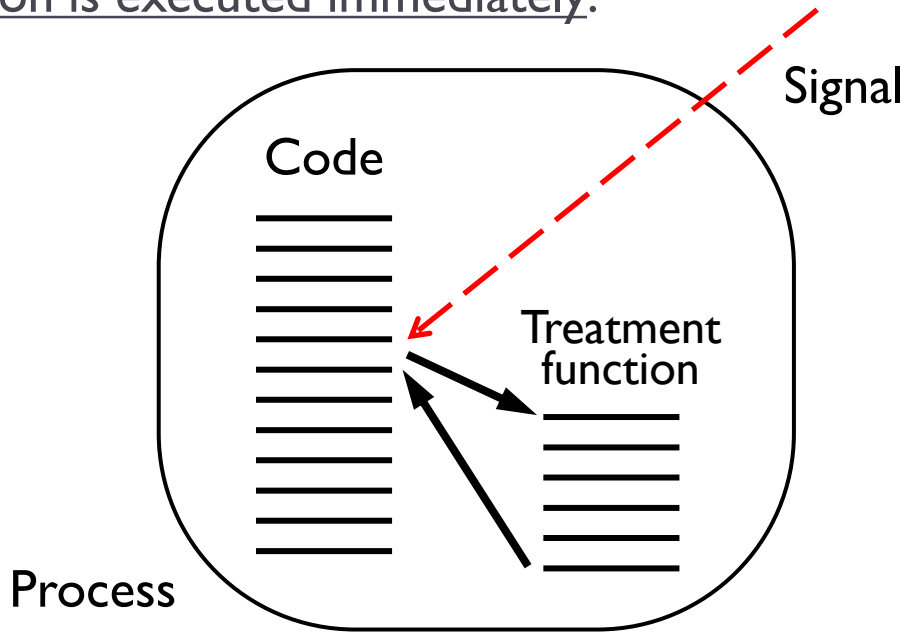
Signals: interruptions to process

- Mechanism to communicate to a process the occurrence of an event in an asynchronous manner and allow to react to such event.
  - Signals are interruptions to the process.
  - An associated processing function is executed immediately:
    - Ignore signal (being "immune")
    - Default processing (kill/ignore)
    - Invoke your own routine.



Signals: interruptions to process

- Mechanism to communicate to a process the occurrence of an event in an asynchronous manner and allow to react to such event.
  - Signals are interruptions to the process.
  - An associated processing function is executed immediately:
    - Ignore signal (being "immune")
    - Default processing (kill/ignore)
    - Invoke your own routine.
- Sending or generation from:
  - Operating system
  - Process



RECAP

Event-oriented programming

general issues

```
int main ( … )
{
    …
    On (event1, handler1) ;
    …
}
```

1) Associate the handler (handler1) to the event

RECAP

Event-oriented programming

general issues

```
void handler1 ( … )
{
}

int main ( … )
{
    …
    On (event1, handler1) ;
    …
}
```

2) Code the handler function that will handle the event

1) Associate the handler (handler1) to the event

RECAP

Event-oriented programming

general issues

```
int global1;
…

void handler1 ( … )
{
}

int main ( … )
{
    …
    On (event1, handler1) ;
    …
}
```

3) To communicate functions, global variables are used

2) Code the handler function that will handle the event

1) Associate the handler (handler1) to the event

Example: count times Ctrl-C is pressed

Old API

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

int contador = 0 ;
int do_quit = 0 ;    // false

void sig_handler ( int signal_id )
{
    if (SIGINT == signal_id) {
       printf("contador = %d\n", contador);
       contador++ ;
    }
    if (SIGQUIT == signal_id) {
       do_quit = 1;    // true
    }
}

int main ( int argc, char *argv[] )
{
    signal(SIGINT,    sig_handler) ;    // CTRL+c
    signal(SIGQUIT,  sig_handler) ;    // CTRL+\
    while(! do_quit) {}
    return 0 ;
}
```

3) To communicate functions, global variables are used

2) Code the handler function that will handle the event

1) Associate the handler (handler1) to the event

signal.h

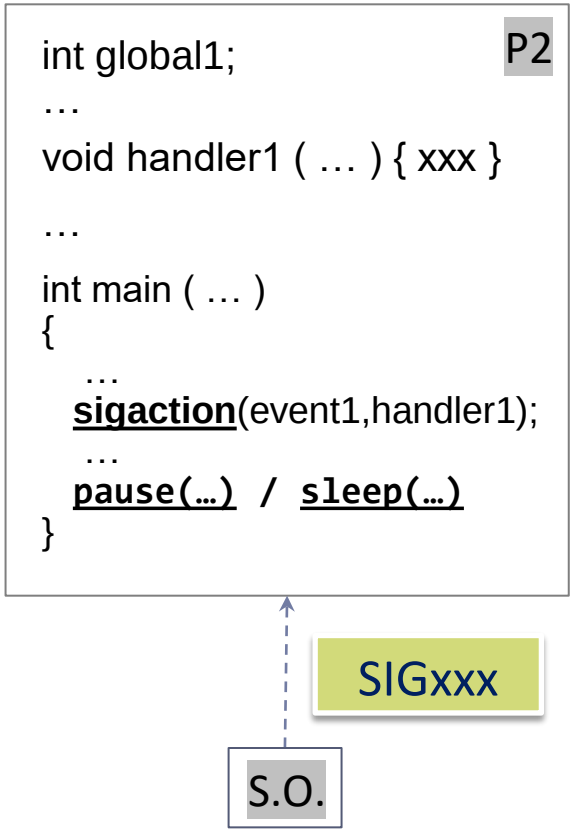
SIGILL_____illegal instruction
SIGALRM_____timer expires
SIGKILL_____kill the process
SIGSEGV_____segmentation fault
SIGUSR1 & SIGUSR2_reserved for programmer use

```
alex@potato:$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

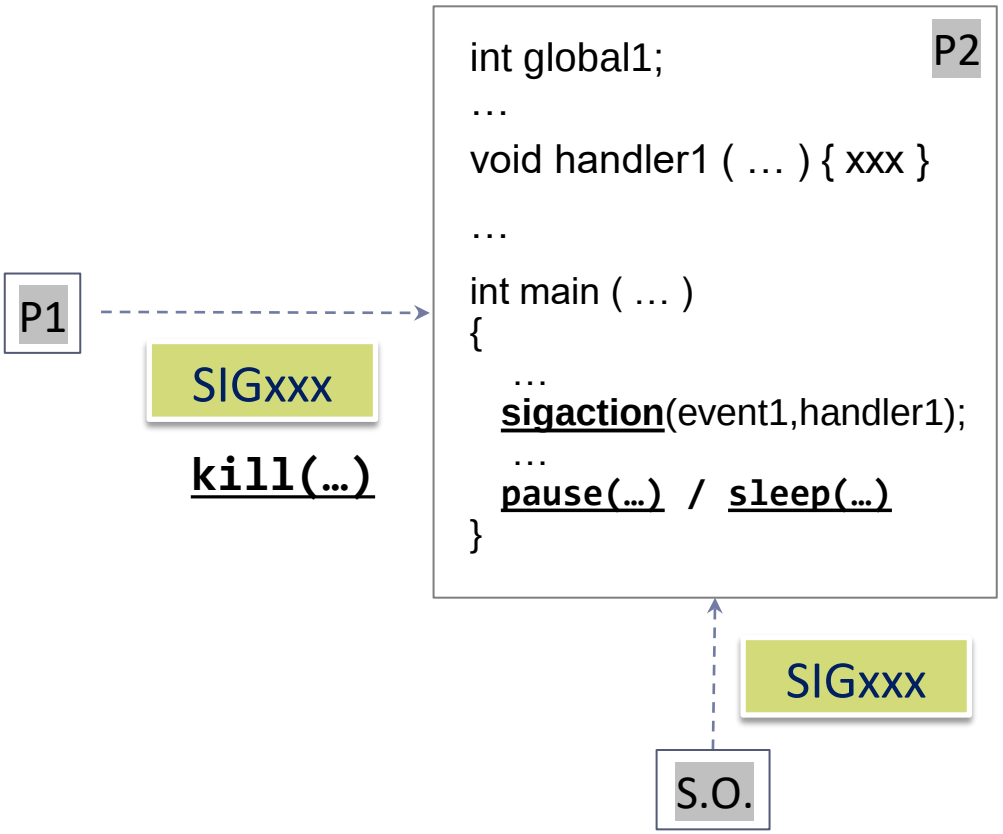
Signals: examples

- A process receives from the operating system the signal
  - SIGCHLD when a child process ends.
  - SIGILL when trying to execute an illegal machine instruction.
  - A process when it is running from a terminal as a foreground task and you press the keys:
    - Control and c simultaneously receives a SIGINT signal.
    - Control and z simultaneously receives a SIGSTOP signal.
- A process receives the SIGUSR1 signal from another process when the other executes `kill(<pid>, SIGUSR1) ;`

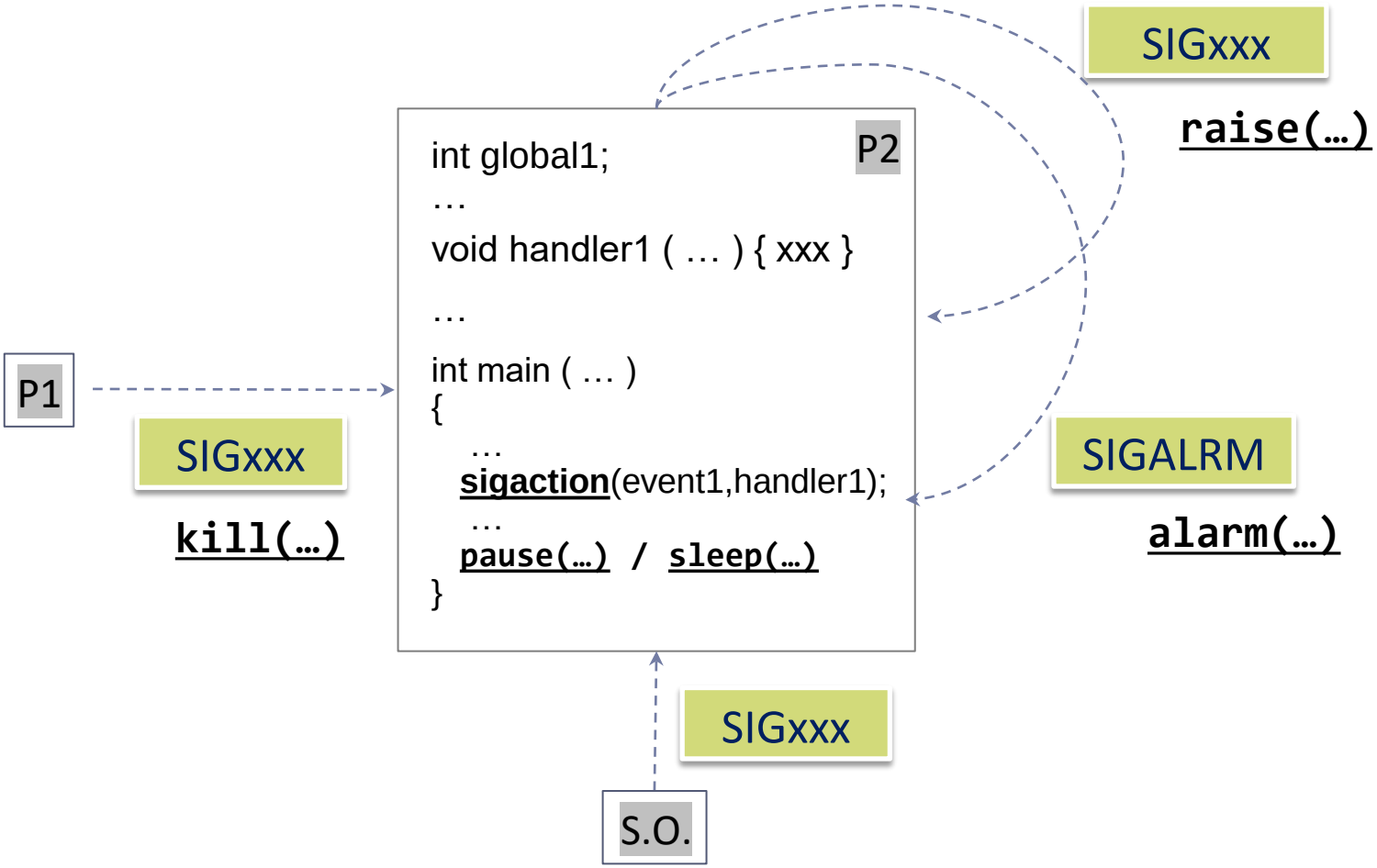
Signals: operating system to process



Signals: process to process



Signals: process to itself



Example: count times Ctrl-C is pressed

New API

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

int contador = 0 ;
int do_quit = 0 ;   // false

void sig_handler ( int signal_id )
{
    if (SIGINT == signal_id) {
       printf("contador = %d\n", contador);
       contador++ ;
    }
    if (SIGQUIT == signal_id) {
       do_quit = 1;    // true
    }
}

int main ( int argc, char *argv[] )
{
    struct sigaction act ;

    act.sa_handler = sigint_handler ;
    act.sa_flags = 0 ;                          // by default
    sigaction(SIGINT,  &act, NULL) ;  // CTRL+c
    sigaction(SIGQUIT, &act, NULL) ;  // CTRL+\
    while(! do_quit) {}
    return 0 ;
}
```

POSIX services for signal handling
kill

Service	`int kill ( pid_t pid, int sig ) ;`
Arguments	▫ **pid**: identificador de el/los proceso/s al/os que mandar la señal. ▫ **sig**: signal identifier to be sent.
Returns	▫ Zero if successful (at least one signal sent). ▫ -1 in case of error.

Description	▫ Sends to the process "pid" the signal "sig".
	▫ Special cases:
	  ▫ pid > 0 -> process with identifier == <pid>
	  ▫ pid = 0 -> to all processes with the same gid as the kill() caller.
	  ▫ pid = -1 -> to all the processes that the kill() caller can send.
	  ▫ pid < -1 -> to all processes in the process group with ID <pid>.

POSIX services for signal handling
raise

Service	`int raise (int sig ) ;`
Arguments	▫ **sig**: signal identifier to be sent.
Returns	▫ Zero if successful (at least one signal sent). ▫ -1 in case of error.
Description	▫ Sends to the process itself the signal "sig". ▫ In a single process it is equivalent to: `kill(getpid(), sig);` With multithreads: `pthread_kill(pthread_self(), sig);`

POSIX services for signal handling
pause

Service	`int pause ( void ) ;`
Arguments	▫ None.
Returns	▫ After the signal arrives and its handler is executed, pause() returns -1.

Description	▫ Blocks the process until a signal is received.
	▫ Details to remember:
	  ▫ It is not possible to specify a deadline for unblocking.
	  ▫ It does not allow to indicate the type of signal expected.
	  ▫ It does not unblock the process in case of ignored signals.

POSIX services for signal handling
sigaction

Service	`int sigaction ( int sig,`
	`struct sigaction *act,`
	`struct sigaction *oact ) ;`

Arguments	▫ **sig**: identifies the signal. ▫ **act**: pointer to structure describing the treatment to be used. ▫ **oact**: (if != NULL) will point to the old treatment used before.
Returns	▫ Zero if successful. ▫ -1 in case of error.

Description	▫ Allows you to specify the action to be performed for the "sig" signal.
	▫ The previous configuration is stored in "oact" if it is not NULL..
	▫ The sigaction struct:
	  ▫ **sa_handler**: SIG_DFL (by default, generally dies but some cases are ignored), SIG_IGN (ignore) or function pointer to be used.
	  ▫ **sa_sigaction**: alternative to sa_handler (do not use both or none).
	  ▫ **sa_mask**: mask of signals to be blocked during the interrupt handler.
	  ▫ **sa_flags**: zero by default (set of *flags*).

POSIX services for signal handling
sigaction

Service	`int sigaction ( int sig,`
	`struct sigaction *act,`
	`struct sigaction *oact ) ;`

Arguments	▫ **sig**: identifies the signal. ▫ **act**: pointer to structure describing the treatment to be used. ▫ **oact**: (if != NULL) will point to the old treatment used before.
Returns	▫ Zero if successful. ▫ -1 in case of error.
Description	▫ Allows you to specify the action to be performed for the "sig" signal. ▫ The previous configuration is stored in "oact" if it is not NULL.. ▫ The sigaction struct: ▫ **sa_handler**: SIG_DFL (by default, generally dies but some cases are ignored), SIG_IGN (ignore) or function pointer to be used. ▫ **sa_sigaction**: alternative to sa_handler (do not use both or none). ▫ **sa_mask**: mask of signals to be blocked during the interrupt handler. ▫ **sa_flags**: zero by default (set of *flags*).

API for signal sets

- `int sigemptyset ( sigset_t * set ) ;`
  - **Creates** an **empty** set of signals.
- `int sigfillset ( sigset_t * set );`
  - **Creates** a **full** set with all possible signals.
- `int sigaddset ( sigset_t * set, int signo );`
  - **Adds** a **signal** to a set of signals.
- `int sigdelset ( sigset_t * set, int signo );`
  - **Deletes** a **signal** from a set of signals.
- `int sigismember ( sigset_t * set, int signo );`
  - **Checks if** a **signal belongs to** a **set**.

Services POSIX
sleep

Service	`int sleep ( unsigned int sec ) ;`
Arguments	▫ **sec**: seconds to sleep the process (suspends its execution).
Returns	▫ Zero if all the time has elapsed or the number of seconds left to sleep if the process has been interrupted by a signal.
Description	▫ Suspends a process until a time limit expires or a signal is received

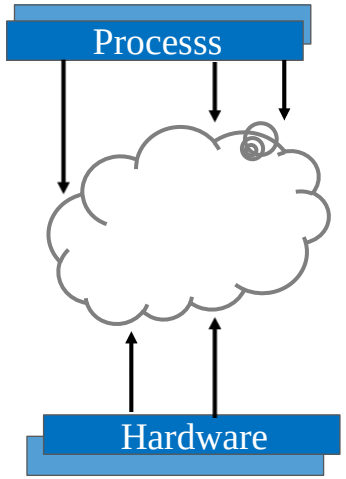
Contents

1. Signals and exceptions.
2. Timers.
3. Process environment.
4. Process communication with pipes.
   - Local message passing.

RECAP

O.S. events: Exceptions

- During boot-up.
- **After boot-up,** is executed **in response to events:**
  - **System call.**
    - { Source: "processes", Function: "Service requests" }
      - Process management
      - Memory management
      - File management
      - Device management
      - Communication
      - Maintenance
  - **Exception.**
    - { Source: "**process**", Function: "**Handle exceptions**" }
  - **Hardware interruption.**
    - { Source: "hardware", Function: "Request for hw attention." }
- In kernel processes (firewall, etc.)



Exceptions

- Hardware detects special conditions:
  - Division by zero, page fault, write to read-only page, stack overflow, etc.
- Transfers control to the O.S. for processing.
  - Save process context.
  - Switch to protected mode
  - Handler execution in the O.S.
    - Most exceptions cause the O.S. to send a signal to the process indicating the exception.
    - Many programming languages (Java, C++, Python, ...) use an exception mechanism to handle runtime errors.

Example: Java exceptions

```
public class JavaExceptionExample
{
    public static void main ( String args[] )
    {
        try
        {
            // array index
            int a[] = new int[2] ;
               a[5] = 20 ;

            // divide by cero
            int data=100 / 0 ;

            // …
        }
        catch ( ArithmeticException e )
        {
            System.out.println(e);
        }
        System.out.println("After exception\n");
    }
}
```

- In Java there is a construct for working with exceptions:
  Try { <code> }
  catch (<exception>) { <code> }
- Avoiding program code checks within frequent code:
  - Improves code clarity
  - Improved performance.

- In Java there is one class to represent an exception, with a hierarchy of subclasses
- Examples: ArrayIndexOutOfBoundsException, NullPointerException, FileNotFoundException, ArithmeticException, IllegalArgumentException

Contents

1. Signals and exceptions.
2. **Timers.**
3. Process environment.
4. Process communication with pipes.
   - Local message passing.

Timers
UNIX

- The O.S. keeps one timer per process.
  - A counter is kept in the BCP of the process for the time remaining until the timer expires.
- The O.S. sends a SIGALRM signal to the process when the current timer expires.
  - If a timer in the BCP reaches zero, the processing function is executed.
- The O.S. has an API to work with Timers.
  - *alarm(seconds)* that updates the counter at BCP.

POSIX services for timing
alarm

Service	`int alarm ( unsigned int sec ) ;`
Arguments	▫ **sec**: number of seconds after which SIGALRM will be sent.
Returns	▫ If there was no timer activated, then it returns zero. ▫ Otherwise, it returns the number of seconds left to expire the previous timer.

Description	▫ Set a new timer:
	  ▫ If there was a timer ticking, then it is removed and a new one is put in its place.
	  ▫ If the parameter is zero, then the timer is disabled.

Example 1: print message every 5 sec.

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void treat_alarmarm ( int signal_id ) {
    printf("¡ALARM!\n");
}

int main ( int argc, char *argv[] )
{
     struct sigaction act ;

     act.sa_handler = treat_alarmarm ;
     act.sa_flags = 0 ;  /* by default */
     sigaction(SIGALRM, &act, NULL) ;

     while(1) {
        alarm(5) ;
        pause() ;
     }

     return 0 ;
}
```

Example 2: timed execution (1/2)

```c
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

pid_t pid;
void treat_alarm(void) {
  kill(pid, SIGKILL);
}

int main ( int argc, char **argv )
{
  int status;
  char **arguments;
  struct sigaction act;
  arguments = &argv[1];
  pid = fork();
  switch(pid) {
    case -1: /* error */
            perror ("fork");
             exit(-1);
    case 0: /* hijo */
            execvp(arguments[0], arguments);
            perror("exec");
              exit(-1)
    default:  /* padre */
            act.sa_handler = treat_alarm;
             act.sa_flags = 0;
             sigaction(SIGALRM, &act, NULL);
            alarm(5);
            wait(&status);
    }
    return 0;
}
```

```c
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

pid_t pid;

void treat_alarm(void)
{
   kill(pid, SIGKILL);
}

main(int argc, char **argv)
{
   int    status;
   char **arguments;
   struct sigaction act;

   arguments = &argv[1];
   pid = fork();
```

Example 2: timed execution (2/2)

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

pid_t pid;
void treat_alarm(void) {
  kill(pid, SIGKILL);
}

int main ( int argc, char **argv )
{
  int status;
  char **arguments;
  struct sigaction act;
  arguments = &argv[1];
  pid = fork();
  switch(pid) {
    case -1: /* error */
            perror ("fork");
             exit(-1);
    case 0: /* hijo */
            execvp(arguments[0], arguments);
            perror("exec");
              exit(-1)
    default:  /* padre */
            act.sa_handler = treat_alarm;
             act.sa_flags = 0;
             sigaction(SIGALRM, &act, NULL);
            alarm(5);
            wait(&status);
    }
    return 0;
}
```

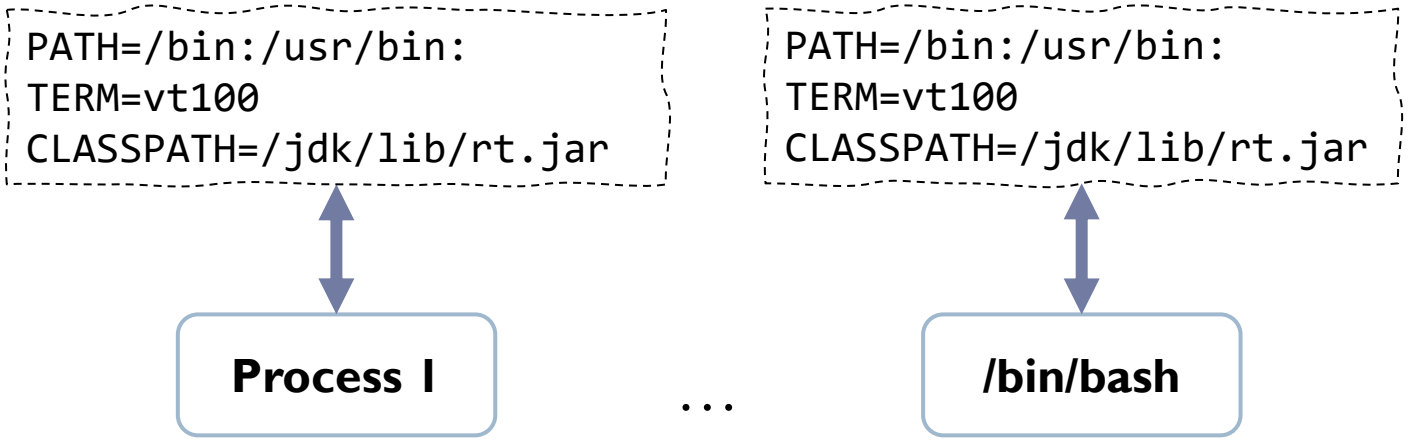
```
  switch(pid) {
    case -1: /* error fork() */
             perror ("fork");
             exit(-1);
    case 0:  /* son */
             execvp(arguments[0],
                    arguments);
             perror("exec");
             exit(-1);
    default: /* father */
          act.sa_handler = treat_alarm;
          act.sa_flags = 0;
          sigaction(SIGALRM,&act,NULL);
          alarm(5);
          wait(&status);
  }

  exit(0);
}
```

Contents

1. Signals and exceptions.
2. Timers.
3. **Process environment.**
4. Process communication with pipes.
   - Local message passing.

Environment variables

- Information accessible to a process in the form of key and value tuples.
- Mechanism for passing information to processes.
  - Process configuration aspects can be updated
  - The O.S. itself uses the environment variables (e.g., PATH)



Environment variables

- Information accessible to a process
  in the form of key and value tuples.
- Mechanism for passing information to processes.
  - Process configuration aspects can be updated
  - The O.S. itself uses the environment variables (e.g., PATH)
- It is possible to interact with environment variables from:
  - **Commands from shell (env, set, export)**
  - In some "O.S.+programming language" is accessible from main(…)
  - O.S. API (getenv, setenv, putenv)

Examples: commands for enviroment

```
alex@potato:$ env
SHELL=/bin/bash
PWD=/mnt/c/Users/alex
LOGNAME=alex
MOTD_SHOWN=update-motd
TERM=xterm-256color
HOME=/home/alex
USER=alex
LANG=C.UTF-8
…

alex@potato:$ export KEY=value
alex@potato:$ env | grep KEY
KEY=value
```

PATH_____list directories to search for binaries
PWD_____current working directory
TERM_____terminal type to be used in console

Environment variables

- Information accessible to a process
  in the form of key and value tuples.
- Mechanism for passing information to processes.
  - Process configuration aspects can be updated
  - The O.S. itself uses the environment variables (e.g., PATH)
- It is possible to interact with environment variables from:
  - Commands from shell (env, set, export)
  - **In some "O.S.+programming language" is accessible from main(…)**
  - O.S. API (getenv, setenv, putenv)

Example: main with envp

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc, char** argv, char** envp )
{
  int i ;

  for (i=0; envp[i]!=NULL; i++)
  {
     printf("%s\n", envp[i]);
  }

  return 0;
}
```

Environment variables

- Information accessible to a process
  in the form of key and value tuples.
- Mechanism for passing information to processes.
  - Process configuration aspects can be updated
  - The O.S. itself uses the environment variables (e.g., PATH)
- It is possible to interact with environment variables from:
  - Commands from shell (env, set, export)
  - In some "O.S.+programming language" is accessible from main(…)
  - **O.S. API (getenv, setenv, putenv)**

API for working with environment

- `char *getenv ( const char * var ) ;`
  - **Gets** the value of environment variable 'var'.
- `int setenv ( const char * name, const char * value, int overwrite ) ;`
  - **Modify/Add** a variable 'name' with value 'value'.
- `int puntenv ( const char * nombre );`
  - **Modifies/adds** a variable of the form "key=value".

Environment variables

- The process environment inherits the following from the parent:
  - Argument vector with which the program was invoked.
  - Environment vector, i.e. the list of variables <name, value> that the parent passes to the children.
- Passing environment variables between parent and child:
  - It is a flexible way of communicating the two and allows you to configure aspects of the child's performance (in user mode).
- The environment variables makes it possible to particularize aspects at the level of each particular process.
  - Instead of having one common configuration for the entire system.

ARCOS Group
Universidad Carlos III de Madrid

Lesson 3
Signals, exceptions and pipes

Operating Systems
Computer Science and Engineering