

# Práctica 1: Procesos y señales



POSIX y glibc

---



Javier Barbero Gómez

Grado en Ingeniería Informática  
Sistemas Operativos

# Participar mediante Blackboard



 **Javier Barbero Gomez**  
Moderador 



---



 Ausente  Cerrar sesión



---

**Comentarios**

 Feliz  Triste

 Sorprendido  Confundido

 Más rápido  Más lento

 De acuerdo  En desacuerdo



## Sobre las clases prácticas

- Reparto de grupos
- No se registrará la asistencia
- No hay entregas
- Examen práctico

# Sobre las clases prácticas

1. Procesos y señales (esta práctica)
2. Hilos
3. Comunicación y sincronización de hilos mediante semáforos
4. Comunicación y sincronización entre procesos mediante memoria compartida y semáforos

- Thinstation a través de SSH
- SFTP
- Entorno local *UNIX-like*

## Objetivo de esta práctica

- Aprender sobre POSIX y su relevancia actual
- Aprender sobre el papel de POSIX dentro de sistemas GNU/Linux: la librería *glibc*
- Aprender a crear y gestionar procesos en sistemas que siguen el estándar POSIX

- Complementar el contenido de la práctica con *consultas bibliográficas* (web, biblioteca...)
- De cara al examen, es recomendable explorar *otros ejemplos*
- Seguir un estilo de programación *claro y consistente*

# El estándar POSIX





# El estándar POSIX

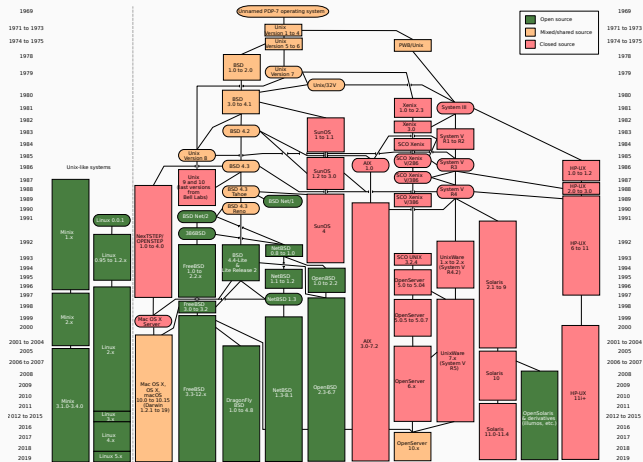
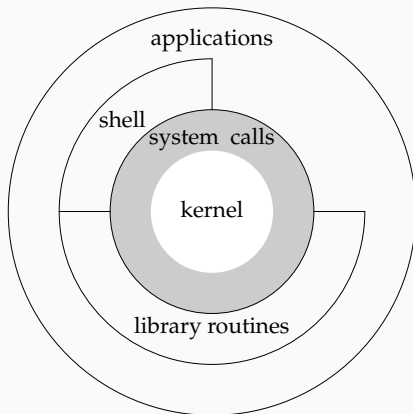
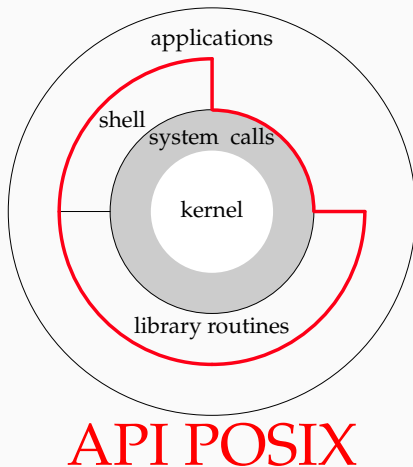


Figura 1: Diagrama de la historia de los SO basados en UNIX (CC BY-SA)



**Figura 2:** Arquitectura del sistema operativo UNIX (extraído de *Advanced Programming in the UNIX Environment, 3rd edition*)



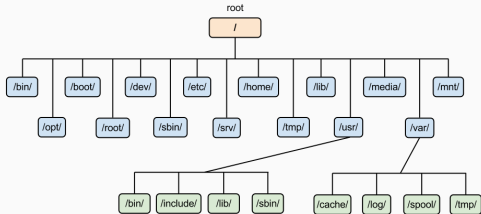
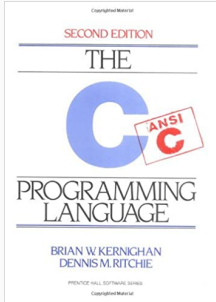
## API POSIX

**Figura 2:** Arquitectura del sistema operativo UNIX (extraído de *Advanced Programming in the UNIX Environment, 3rd edition*)

- POSIX.1-2017
- IEEE Std 1003.1-2017
- The Open Group Technical Standard Base Specifications, Issue 7

<http://pubs.opengroup.org/onlinepubs/9699919799/>

# El estándar POSIX



```
Terminal
-rwxr-xr-x 1 bin 18296 Jun 8 1979 fack
-rwxr-xr-x 1 bin 1458 Jun 8 1979 getty
-rwxr-xr-x 1 root 45 Jun 8 1979 group
-rwxr-xr-x 1 bin 2482 Jun 8 1979 init
-rwxr-xr-x 1 bin 8484 Jun 8 1979 mkfs
-rwxr-xr-x 1 bin 3542 Jun 8 1979 mkmod
-rwxr-xr-x 1 bin 3578 Jun 8 1979 mount
-rwxr-xr-x 1 root 141 Jun 8 1979 passwd
-rwxr-xr-x 1 bin 366 Jun 8 1979 rc
-rwxr-xr-x 1 bin 336 Jun 8 1979 rfs
-rwxr-xr-x 1 bin 3794 Jun 8 1979 runcnt
-rwxr-xr-x 1 bin 634 Jun 8 1979 update
-rwxr-xr-x 1 bin 46 Sep 22 05:45 utmp
-rwxr-xr-x 1 root 4500 Jun 8 1979 wall
# ls -l /usr/bin
-rwxr-xr-x 1 sys 53902 Jun 8 1979 /rptunix
-rwxr-xr-x 1 sys 53220 Jun 8 1979 /rptunix
-rwxr-xr-x 1 root 50990 Jun 8 1979 /runkit
-rwxr-xr-x 1 root 51982 Jun 8 1979 /r2unix
-rwxr-xr-x 1 sys 51720 Jun 8 1979 /rptunix
-rwxr-xr-x 1 sys 51274 Jun 8 1979 /rptunix
# ls -l /bin/sh
-rwxr-xr-x 1 bin 17310 Jun 8 1979 /bin/sh
```



Implementaciones en:

- GNU/Linux
  - glibc
- BSD
  - macOS
  - FreeBSD
- Windows
  - Microsoft POSIX subsystem
- ...

Implementaciones en:

- GNU/Linux
  - `glibc`
- BSD
  - macOS
  - FreeBSD
- Windows
  - Microsoft POSIX subsystem
- ...

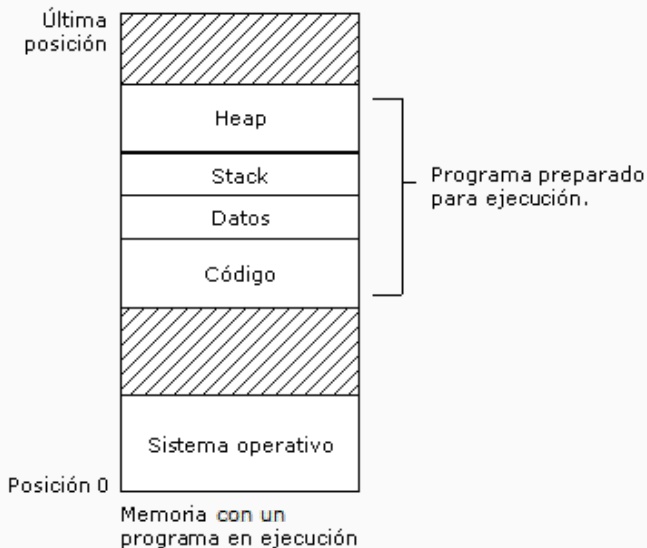
- Identificación
- Planificación y estado
- Memoria asignada
- Contexto
- Recursos asignados (ficheros, red, ...)
- Comunicación entre procesos
- Auditoría



**Figura 3:** Bloque de control de proceso (BCP)



# Procesos



## Terminar un proceso: `exit()`

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

```
void exit(int status);
```

## Terminar un proceso: `exit()`

Dos formas de terminar un proceso:

- `return` <código de salida>; desde `main`
- `exit(<código de salida>);`

## Terminar un proceso: `exit()`

Dos formas de terminar un proceso:

- `return` <código de salida>; desde `main`
- `exit(<código de salida>);`

## Identificar procesos: getpid(), getppid(), getuid()

```
#include <unistd.h>
```

```
// Devuelve el ID de proceso
```

```
pid_t getpid (void);
```

```
// Devuelve el ID de proceso del proceso padre
```

```
pid_t getppid (void);
```

```
// Devuelve el ID de usuario que
```

```
// ejecuta el proceso
```

```
uid_t getuid (void);
```

## Variables de entorno: getenv()

```
// Variable de entorno  
// (no necesita include)  
extern char **environ;  
  
#include <stdlib.h>  
  
// Obtener el valor de una variable de entorno  
// por su nombre  
char *getenv (const char *name);
```

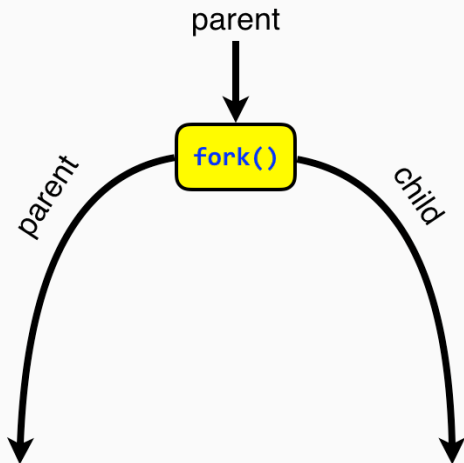
## Crear procesos: fork()

```
#include <unistd.h>
```

```
// Clonar el proceso actual
```

```
pid_t fork (void);
```

## Crear procesos: `fork()`



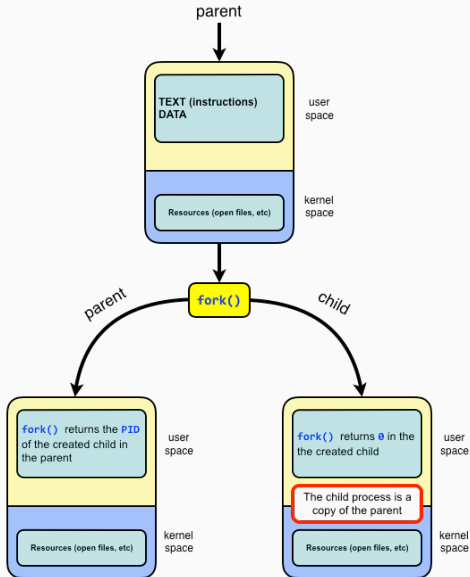


## Crear procesos: `fork()`

«Upon successful completion, `fork()` shall return **0 to the child process** and shall return **the process ID of the child process to the parent process**. Both processes shall continue to execute from the `fork()` function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and `errno` shall be set to indicate the error.»

<https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html>

# Crear procesos: `fork()`



# Ejemplo de fork()

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS

int main () {
    pid_t pid;
    ➔ pid = fork();

    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

# Ejemplo de fork()

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS

int main () {
    pid_t pid;
    ➔ pid = fork();

    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS

int main () {
    pid_t pid;
    ➔ pid = fork();

    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

# Ejemplo de fork()

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS
```

```
int main () {
    pid_t pid;
    → pid = fork();
```

pid == <PID hijo>

```
    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS
```

```
int main () {
    pid_t pid;
    → pid = fork();
```

pid == 0

```
    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

# Ejemplo de fork()

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS

int main () {
    pid_t pid;
    pid = fork();

    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        → default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

pid == <PID hijo>

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS

int main () {
    pid_t pid;
    pid = fork();

    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        → case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

pid == 0

# Ejemplo de fork()

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS

int main () {
    pid_t pid;
    pid = fork();

    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

pid == <PID hijo>

```
#include <stdio.h>
#include <unistd.h> // fork, pid_t
#include <errno.h> // errno
#include <string.h> // strerror
#include <stdlib.h> // EXIT_SUCCESS

int main () {
    pid_t pid;
    pid = fork();

    switch (pid) {
        case -1:
            printf("Error al hacer fork: %s\n", strerror(errno));
            break;
        case 0:
            printf("Soy el proceso hijo\n");
            break;
        default:
            printf("Soy el proceso padre\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

pid == 0

## Sincronizar procesos: wait(), waitpid()

```
#include <sys/wait.h>
```

```
// Esperar a que termine un proceso hijo
```

```
pid_t wait (int *wstatus);
```

```
pid_t waitpid (pid_t pid,  
               int *wstatus,  
               int options);
```



## Sincronizar procesos: `wait()`, `waitpid()`

Cuándo usar `waitpid`:

- Cuando se quiere esperar a *un proceso hijo concreto*
- Cuando se quiere especificar *alguna opción especial*
- Ambas

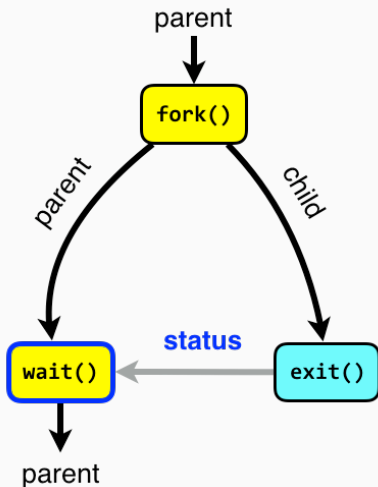
`wait` puede usarse en cualquier otro caso.

## Sincronizar procesos: `wait()`, `waitpid()`

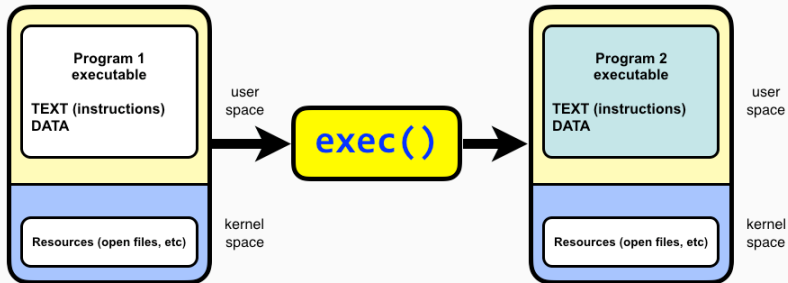
«If `wait()` returns because the status of a child process is available, this function shall return a value equal to the process ID of the child process for which status is reported. If `wait()` returns due to the delivery of a signal to the calling process, -1 shall be returned and `errno` set to `[EINTR]`. Otherwise, -1 shall be returned, and `errno` set to indicate the error.»

<https://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html>

## Sincronizar procesos: `wait()`, `waitpid()`



# Ejecutar un programa: exec



## Ejecutar un programa: exec

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ...,
    ↪ (char *)NULL);

int execlp(const char *path, const char *arg0, ...,
    ↪ (char *)NULL, char *const envp[]);

int execlp(const char *file, const char *arg0, ...,
    ↪ (char *)NULL);

int execv(const char *path, char *const argv[]);

int execve(const char *path, char *const argv[],
    ↪ char *const envp[]);

int execvp(const char *file, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const
    ↪ envp[]);
```

# Ejecutar un programa: `exec`

Variantes:

- `l`: Los parámetros son pasados como una lista variable de argumentos
- `v`: Los parámetros son pasados como un array de cadenas `char **` terminado en `(char **) NULL`
- `e`: Acepta un parámetro adicional para definir el `environ` del nuevo ejecutable
- `p`: Usar la variable de entorno `PATH` para determinar donde se encuentra el ejecutable (igual que en el terminal)
- `fexecve`: Usar un fichero abierto como ejecutable

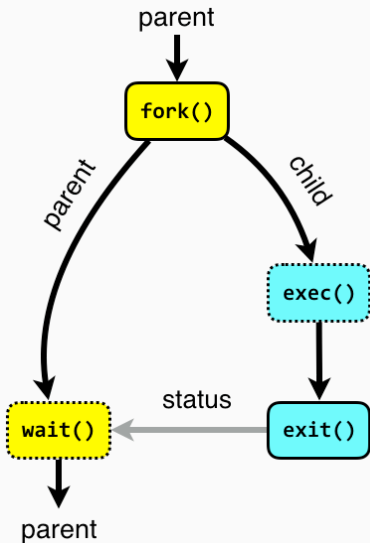
## Ejecutar un programa: exec

Ejemplos:

```
exec1("/usr/bin/firefox", "firefox", "--search",  
↪ "posix", (char *) NULL);
```

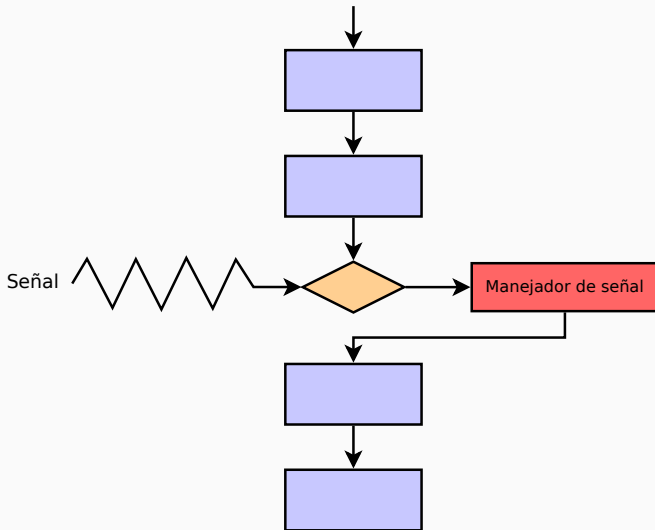
```
char *const args[] = {"firefox", "--search", "posix",  
↪ (char *) NULL};  
execvp("firefox", args);
```

## Gestión de procesos: resumen





# Gestión de señales



¿Cuándo se envía una señal a un proceso?

- Pulsar ciertas combinaciones de teclas (`Ctrl` + `C`, `Ctrl` + `Z`) en una *shell* mientras se ejecuta un proceso
- De forma programada, por el kernel
- Acceso a zona de memoria no válida
- Enviada por otro proceso (una *shell* u otro programa en C)

Consecuencias de la recepción de una señal:

- Ninguna (ignorada)
- Detener la ejecución de un proceso (sin eliminarlo)
- Reanudar la ejecución de un proceso detenido
- Matar el proceso
- Tratar de manera específica: *manejadores de señales*

Cabecera para tratamiento de señales:

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

Algunas de las señales más comunes:

- **SIGINT**: Interrupción, Ctrl + C en una terminal
- **SIGFPE**: Error de coma flotante
- **SIGTERM**: Solicitud de finalizar el proceso
- **SIGKILL**: Ídem, pero no es capturable
- **SIGSTOP** y **SIGCONT**: Detener y reanudar el proceso
- **SIGALRM**: Alarma programable
- **SIGUSR1** y **SIGUSR2**: Uso personalizado

```
#include <signal.h>  
  
// Especificar el comportamiento  
// al recibir una señal  
void (*signal(int sig, void (*func)(int)))(int);
```

- signal

*// signal*

- `signal` *// signal*
- `signal(` `)` *// es una función*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*



- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*
- `signal(int sig, (*func)( ))` *// a una función*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*
- `signal(int sig, (*func)( ))` *// a una función*
- `signal(int sig, (*func)(int))` *// con un parámetro int*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*
- `signal(int sig, (*func)( ))` *// a una función*
- `signal(int sig, (*func)(int))` *// con un parámetro int*
- `signal(int sig, void (*func)(int))` *// que devuelve void*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*
- `signal(int sig, (*func)( ))` *// a una función*
- `signal(int sig, (*func)(int))` *// con un parámetro int*
- `signal(int sig, void (*func)(int))` *// que devuelve void*
- `*signal(int sig, void (*func)(int))` *// que devuelve un puntero*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*
- `signal(int sig, (*func)( ))` *// a una función*
- `signal(int sig, (*func)(int))` *// con un parámetro int*
- `signal(int sig, void (*func)(int))` *// que devuelve void*
- `*signal(int sig, void (*func)(int))` *// que devuelve un puntero*
- `(*signal(int sig, void (*func)(int)))( )` *// a una función*



- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*
- `signal(int sig, (*func)( ))` *// a una función*
- `signal(int sig, (*func)(int))` *// con un parámetro int*
- `signal(int sig, void (*func)(int))` *// que devuelve void*
- `*signal(int sig, void (*func)(int))` *// que devuelve un puntero*
- `(*signal(int sig, void (*func)(int)))( )` *// a una función*
- `(*signal(int sig, void (*func)(int)))(int)` *// con un parámetro int*

- `signal` *// signal*
- `signal( )` *// es una función*
- `signal( sig, )` *// con un parámetro "sig"*
- `signal(int sig, )` *// de tipo int*
- `signal(int sig, func )` *// y un parámetro "func"*
- `signal(int sig, (*func) )` *// que es un puntero*
- `signal(int sig, (*func)( ))` *// a una función*
- `signal(int sig, (*func)(int))` *// con un parámetro int*
- `signal(int sig, void (*func)(int))` *// que devuelve void*
- `*signal(int sig, void (*func)(int))` *// que devuelve un puntero*
- `(*signal(int sig, void (*func)(int)))( )` *// a una función*
- `(*signal(int sig, void (*func)(int)))(int)` *// con un parámetro int*
- `void (*signal(int sig, void (*func)(int)))(int);` *// que devuelve void*

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

```
// Enviar una señal a un proceso
```

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

## Gestión de señales: programar una alarma

```
#include <unistd.h>
```

```
// Programar una alarma
```

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

- POSIX facilita la portabilidad
- Gestión de procesos:
  - Terminación: `exit()`
  - Identificación: `getpid()`, `getppid()`, `getuid()`
  - Entorno: `getenv()`
  - Creación: `fork()`
  - Ejecución: familia `exec()`
  - Sincronización: `wait()`
- Señales:
  - Capturar señales: `signal()`
  - Enviar señales: `kill()`
  - Programar una alarma: `alarm()`