Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Bloque Temático 2: Gestión de Procesos
Unidad Temática 3

Seminario 3

Llamadas al sistema UNIX para procesos

fSO

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA

Llamadas al sistema UNIX para procesos

- Objetivos:
  - Conocer los **servicios** proporcionados por el Sistema Operativo **UNIX para crear procesos**
  - Presentar **ejercicios en lenguaje C** con llamadas al sistema
  - Conocer el **concepto de señal**
- Bibliografía:
  - "UNIX Programación Práctica", Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 968-880-959-4

Contenido

- Identificación de procesos
- Creación de procesos en UNIX
- Espera de procesos
- Terminación de procesos
- Señales

Llamadas al Sistema UNIX para Procesos

	Procesos
fork	Creación de un proceso hijo
exit	Terminación del proceso en ejecución
wait	Espera la terminación de un proceso
exec	Cambia imagen de memoria por la de un ejecutable (ejecuta programa)
getpid	Obtiene atributos de un proceso
getppid	Obtiene atributos del proceso padre

	Señales
kill	Enviar una señal
alarm	Generar una alarma (señal de reloj)
sigaction	Permite instalar un manejador de señales
sigsuspend	Suspende a un proceso mientras espera que ocurra una señal
sigemptyset	Iniciar una máscara para que no tenga señales seleccionadas
sigfillset	Iniciar una máscara para que contenga todas las señales
sigaddset	Poner una señal específica en un conjunto de señales
sigdelset	Quitar una señal específica en un conjunto de señales
sigismember	Consultar si una señal pertenece a un cto. de señales
sigprocmask	Examinar/modificar máscara de señales

Contenido

- **Identificación de procesos**
- Creación de procesos en UNIX
- Espera de procesos
- Terminación de procesos
- Señales

Identificación de procesos

- **PID del proceso**
  - Cada proceso tiene un identificador o ID
  - El proceso creador es el *padre*, mientras que el proceso creado es el *hijo*. Para conocerlos:
    - PID: identidad del proceso con **`getpid()`**
    - PPID: identidad del proceso padre

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

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

```
/*** ej1_getpid.c *****/
#include <stdio.h>
#include <unistd.h>

int main(void)
{
   printf("\nID del proceso: %ld\n", (long)getpid());

   printf("ID del padre: %ld\n", (long)getppid());

   while(1);

   return 0;
}
```

```
$ gcc –o ej1 ej1_getpid.c
$ ./ej1 &
[1] 2959
$
ID del proceso: 2959
ID del padre: 1060
$ ps -l
  UID   PID  PPID      F CPU PRI NI      SZ    RSS WCHAN    S    ADDR TTY          TIME CMD
  501  1060  1059    4006   0  31  0 2435548   1088 -      S  ffffff80136e3d50 ttys000    0:00.06 -bash
  501  2959  1060    4006   0  31  0 2434832    340 -      R  ffffff80140d8300 ttys000    0:04.65 ./eje1
```

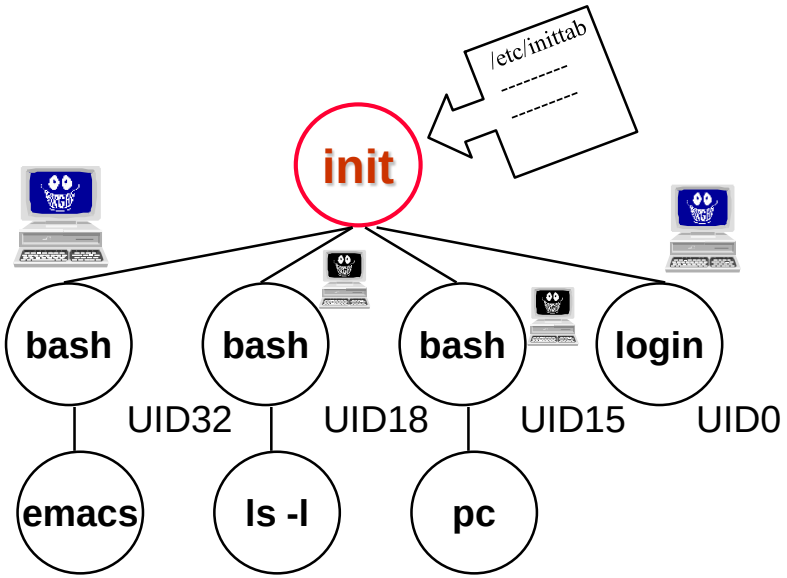
Contenido

- Identificación de procesos
- **Creación de procesos**
- Espera de procesos
- Terminación de procesos
- Señales

Creación de procesos en UNIX

- Unix utiliza un mecanismo de creación por copia
  - El proceso **hijo es una réplica exacta de** su proceso **padre**
  - El proceso **hijo hereda** la mayoría de **atributos** del proceso padre:
    - imagen de memoria
    - UID, GID
    - directorio actual
    - descriptores de ficheros abiertos
  - Unix asigna a cada proceso un identificador PID en el momento de su creación
  - Todo proceso conoce el identificador de su proceso padre: PPID
  - La ejecución del hijo es concurrente e independiente
  - En UNIX existe una **jerarquía de procesos**

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



Creación de procesos en UNIX

- **`fork()`:** creación de procesos

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void)
```

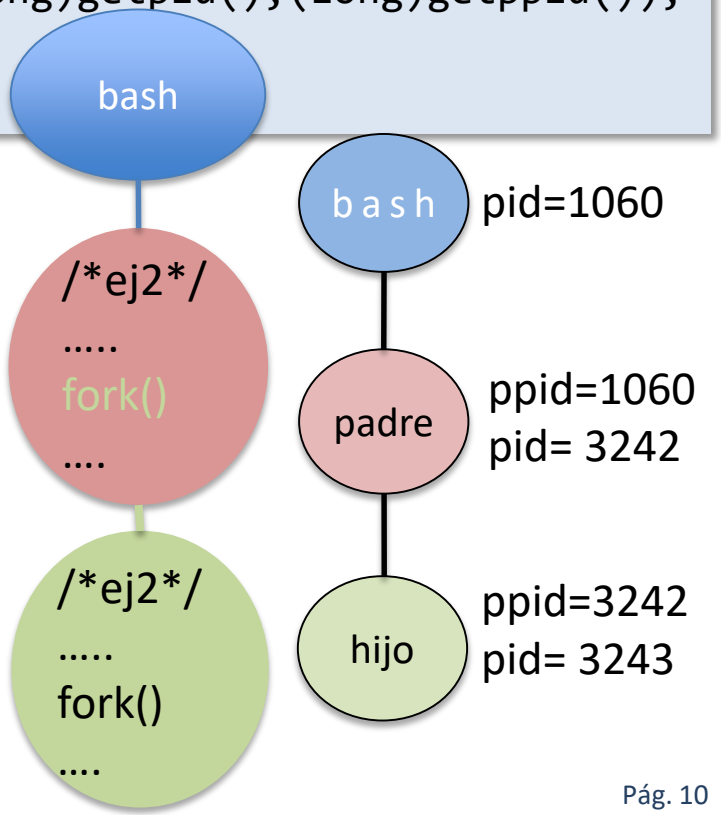
  - **Descripción**
    - Crea un proceso hijo que es un "clon" del padre: hereda gran parte de sus atributos.
    - Atributos heredables: todos excepto **`PID`**, **`PPID`**, señales pendientes, tiempos/contabilidad.
  - **Valor de retorno**
    - 0 al hijo
    - **`PID`** del hijo al padre
    - -1 al padre si error
  - **Errores**
    - Insuficiencia de recursos para crear el proceso

Creación de procesos en UNIX

Después de **fork()**, padre e hijo continúan su ejecución con la siguiente instrucción a **fork()**

```
/**ej2_fork.c **/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{ printf("Proceso %ld crea otro proceso\n", (long)getpid());
  fork();
  printf("Proceso %ld con padre %ld\n",(long)getpid(),(long)getppid());
  sleep(5);
  return 0;
}
```

```
$ ps
  PID TTY           TIME CMD
 1060 ttys000    0:00.07 -bash
$ gcc -o ej2 ej2_fork.c
$ ./ej2
Proceso 3242 crea otro proceso
Proceso 3242 con padre 1060
Proceso 3243 con padre  3242
$
```



Creación de procesos en UNIX

- Padre e hijo ejecutan código distinto

```
// ej3_fork.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
     pid_t val;
     int var = 0;
     printf("PID antesfork(): %d\n", (long) getpid());
     val=fork();
     if (val > 0) {
       printf(“Padre PID: %d\n", (long) getpid());
       var++;
     } else {
       printf(”HijoPID: %d\n", (long) getpid());
     }
     printf("Proceso [%d]-> var=%d\n", (long) getpid(),
var);
     return 0;
}
```

¿Cuántos procesos imprimen este mensaje?

¿Qué valor/es de “var” se muestran?

- Ejemplo

```
// ej4_fork.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

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

     switch (pid) {
     case -1:
       printf("No se ha podido crear el proceso hijo\n");
       break;
     case 0:
       printf("Soy el hijo con PID %ld y PPID %ld\n", (long)getpid(), (long)getppid());
       break;
     default:
       printf("Soy el padre con PID %ld y mi hijo es %d\n", (long)getpid(), pid);
     }
     sleep(5);
     return 0;
}
```

```
$ gcc –o ej4 ej4_fork.c
$ ./ej4
Soy el padre con PID 3702 y mi hijo es 3704
Soy el hijo con PID 3704 y PPID 3702
```

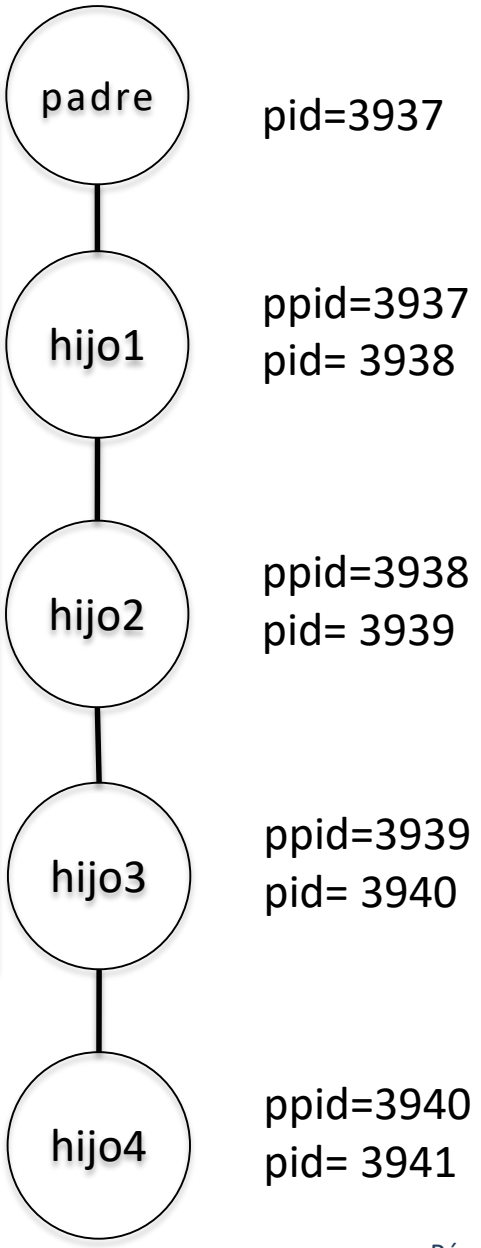
- Creación de procesos en cascada

```
// ej5_proc_chain.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

#define NPROCESOS 4

int main(void)
{ pid_t pid;
  int i;
   for (i=0; i<NPROCESOS; i++) {
     pid=fork();
     if (pid!=0)  break;
     printf("Soy el hijo con PID %ld con padre %ld\n",
                 (long)getpid(), (long)getppid());
   }
  sleep(5);
  return 0;
}
```

Variantes

```
if (pid > 0)
if (pid==0)
if (pid<0)
```

```
$ ./ej5_proc_chain
Soy el hijo con PID 3938 con padre 3937
Soy el hijo con PID 3939 con padre 3938
Soy el hijo con PID 3940 con padre 3939
Soy el hijo con PID 3941 con padre 3940
```



Creación de procesos

- **exec()**
  - La llamada **fork()** crea un hijo que es una copia del proceso que lo llama
  - Para cambiar el código de un proceso se utilizan las llamadas **exec()**
  - Existen diferentes versiones en función de los parámetros a especificar.

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ...,
          const char *argn, char * /*NULL*/);
int execlp(const char *file, const char *arg, ... ,
           const char *argn, char * /*NULL*/);
int execle(const char *path, const char *arg, ...,
           const char *argn, char * /*NULL*/, char * const envp[]);

int execv(const char *path, char *const argv[] );
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

  - **Variante l:** los argumentos se proporcionan por separado
  - **Variante v**: los argumentos se proporcionan con un puntero a vector
  - **Variante p**: se busca la ruta de **file** en el **PATH**
  - **Variante e**: se proporciona el entorno al proceso mediante **envp**, no lo hereda de su padre

Creación de Procesos

- **`exec ()`**
  - Cambia la imagen de memoria de un proceso por la definida en un fichero ejecutable.
  - El fichero ejecutable se expresa dando su nombre **`file`** o su ruta completa **`path`**.
  - Algunos atributos del proceso se conservan y, en particular:
    - El manejo de señales, excepto las señales capturadas para las que se toma la acción por defecto.
    - El **`PID`**, **`PPID`**
    - Tiempos (contabilidad)
    - Los descriptores de fichero
    - El directorio de trabajo, el directorio raíz, la máscara del modo de creación de ficheros.
  - Si el bit **`SETUID`** del fichero ejecutable está activado, **`exec`** pone como **`UID`** efectivo del proceso al **`UID`** del propietario del fichero ejecutable.
    - Ídem con el bit **`SETGID`**
  - Errores
    - Fichero no existente o no ejecutable
    - Permisos
    - Argumentos incorrectos
    - Memoria o recursos insuficientes
  - Valor de retorno
    - Si EXEC retorna al programa que lo llamó es que ha ocurrido un error; el valor de retorno es –1.

La llamada exec( )

- Ejemplo: El proceso hijo ejecuta el “ls”

```
// ej6_exec.c
#include <stdio.h>
#include <sys/types.h>

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

      char* argumentos [] = { "ls", "-l", 0 };
     char*  orden [] = { "ls", "-l", 0 };

      switch (pid) {
      case -1:
            printf("No se ha podido crear el proceso hijo\n");
            break;
      case 0:
            printf("Soy el hijo con PID %ld y voy a listar el directorio\n", (long)getpid());
            if (execvp("ls",argumentos)==-1){
                  printf("Error en exec\n");
                  exit(0);
            }
            break;
      default:
            printf("Soy el padre con PID %ld y mi hijo es %d.\n", (long)getpid(), pid);
      }
      return 0;

}
```

Variantes:

```
execl(“/bin/ls”, “ls”,”-l”,NULL)
```

Contenido

- Identificación de procesos
- Creación de procesos
- **Espera de procesos**
- Terminación de procesos
- Señales

Espera de procesos

- Un padre debe esperar hasta que el hijo finalice:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- Llamada **`wait`**
  - Detiene la ejecución del proceso invocante hasta que un hijo termine , o hasta que el proceso invocante reciba una señal
  - En **`status`** guarda el estado devuelto por el hijo. Existen macros para analizarlo.
  - Retorno:
    - Devuelve el PID del hijo,
    - -1 si error o no hay hijos

Espera de procesos

- **`wait()`**: espera terminación

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- **`wait()/waitpid()`**: Suspende la ejecución del proceso que le invoca, hasta que finaliza alguno de los hijos (**`wait`**) o un hijo en concreto (**`waitpid`**).
- Si existe un hijo zombie, **`wait`** finaliza inmediatamente. Si no, se detiene.
- Cuando **`status`** no es el puntero nulo, contiene:
  - Hijo termina con **`exit`**:

  - Hijo termina por señal:

- Ejemplo (ej7_wait.c)

```
// ej7_wait.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

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

      switch (pid) {
      case -1:
            printf("No se ha podido crear el proceso hijo\n");
            break;
      case 0:
            printf("Soy el hijo con PID %ld y mi padre es %ld\n", (long)getpid(), (long)getppid());
            sleep(20);
            printf("Ya he terminado\n");
            break;
      default:
            printf("Soy el padre con PID %ld y mi hijo es %d. Esperándolo ...\n", (long)getpid(), pid);
            if (wait(&status)!=-1)
                  printf("Mi hijo ha terminado normalmente\n");
      }
      return 0;
}
```

Espera de procesos

- Llamada **`waitpid()`**

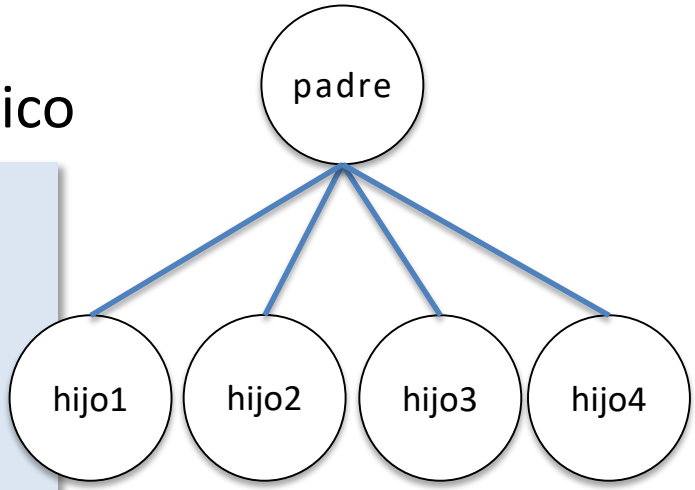
```
pid_t  waitpid(pid_t pid, int *status, int options);
```

  - Esperar a un hijo en particular
  - Parámetros:
    - **`pid`**: PID del hijo a esperar. Si pid vale -1, se espera al primero que acabe (como wait).
    - **`status`**: estado del hijo que retorna
    - **`options`**: por ejemplo, **`WNOHANG`** hace que la llamada sea no bloqueante. En la asignatura utilizaremos normalmente la versión bloqueante de esta llamada: campo **`options`** igual a 0.
    - Valor de retorno: Si es 0, no ha terminado ningún proceso (versión no bloqueante). Si es -1, indica error. Si es mayor que 0, entonces el valor devuelto es el pid del proceso hijo retornado.

Llamada **waitpid()**

- Ejemplo: creación de procesos en abanico

```
// ej8_waitpid.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

#define NPROCESOS 4

int main(void)
{
      pid_t pid[NPROCESOS];
      int i, status;

      for (i=0; i<NPROCESOS; i++) {
            pid[i]=fork();
            if (pid[i]==0){
                  printf("Soy el hijo %ld con padre %ld\n",
                        (long)getpid(), (long)getppid());
                  sleep((i+1)*5);
                  exit(0);
            }
      }
      //Ahora a esperar al tercer hijo
      if (waitpid(pid[2],&status,0)==pid[2])
            printf("Mi tercer hijo ya ha terminado\n");
      return 0;
}
```




```
$gcc –o ej8 ej8_waitpid.c
```

Contenido

- Identificación de procesos
- Creación de procesos
- Espera de procesos
- **Terminación de procesos**
- Señales

`exit()`

- Un proceso termina completamente cuando:
  - El proceso en sí finaliza (normal o anormalmente), y
  - Su padre ha realizado una llamada a wait()
- La terminación normal de un proceso se realiza invocando la llamada a **`exit`**()

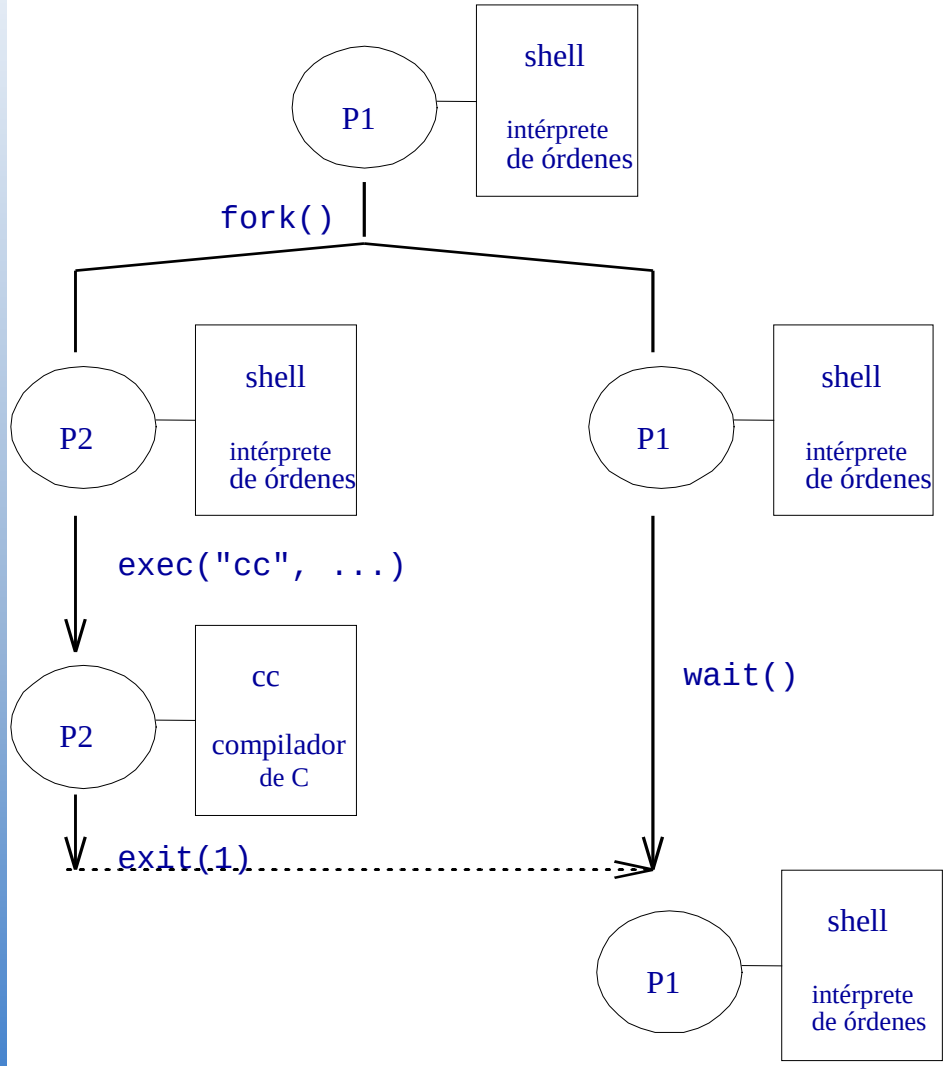
```
void exit (int status)
```

  - El valor del parámetro **`status`** se utiliza para comunicar al proceso padre la forma en que el proceso hijo termina.
  - Por convenio, este valor suele ser 0 si el proceso termina correctamente y cualquier otro valor en caso de terminación anormal.
  - El proceso padre puede obtener este valor a través de la llamada al sistema **`wait`**.

Terminación de procesos

- **Terminación anormal**:
  - El proceso termina por iniciativa del sistema operativo al detectar alguna condición de error (violación de límites, errores aritméticos) o por iniciativa de algún otro proceso
    - Señales
- **Proceso zombie:** Si el proceso finaliza antes de que su padre llame a **`wait()`**
- **Proceso huérfano**: Si el proceso padre termina antes que el hijo
  - Un proceso huérfano es **adoptado** por el proceso **`init()`**

- El shell de Unix (estructura simple)

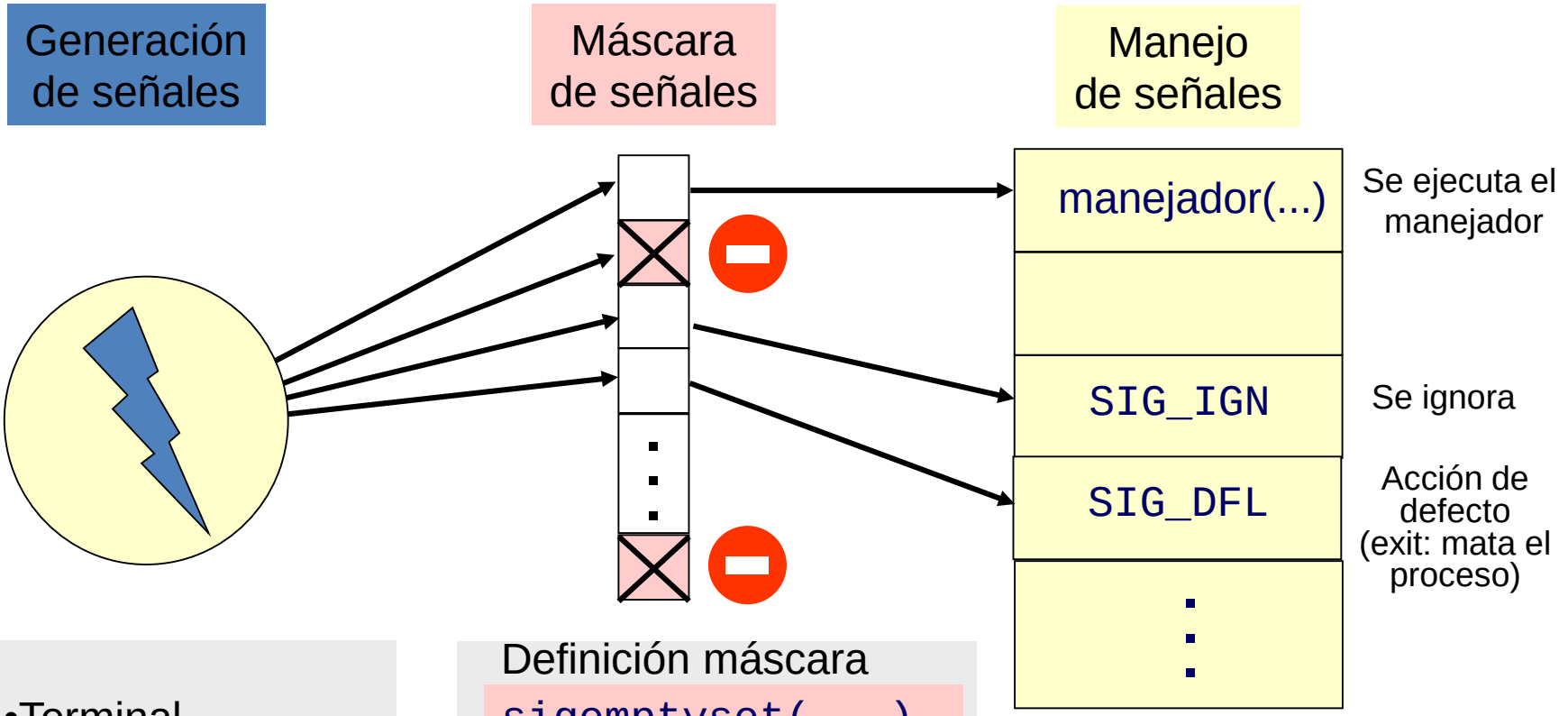



```
while(TRUE) {
 imprimir_prompt();
 leer_orden(orden, param);

 p=fork();                          /* crear hijo */

 if (p != 0) {                      /* código del padre */
  waitpid(-1, &status, 0);          /* esperar hijo */

 }else {                            /* código del hijo */
  exec(orden, params, 0); /* cambiar imagen */
                                     /* de memoria */
  error("No se puede ejec. la orden");
  exit(1);
 }
}
```

Contenido

- Identificación de procesos
- Creación de procesos
- Espera de procesos
- Terminación de procesos
- **Señales**

Señales



- Terminal
- Errores
- Software

```
kill(...)
alarm(..)
```

Definición máscara

```
sigemptyset(...)
sigfillset(..)
sigaddset(..)
sigdelset(..)
sigismemeber(...)
```

Establecimiento máscara

```
sigprocmask(...)
```

Establecer manejador

```
sigaction(...)
```

Espera de señales

```
pause(...)
sigsuspend(…)
```

Señales

- Una **señal** es el **mecanismo** que usa el SO para **informar a los procesos de** determinados **sucesos**
  - La llamada **`wait`** detiene al proceso invocante hasta que un hijo termine o se detenga, ***o hasta que el proceso invocante reciba una señal***
- Todas siguen el mismo patrón:
  - Se genera debido a la ocurrencia de un evento
  - Se suministra al proceso
  - Debe recibir un tratamiento por parte de una rutina de tratamiento por defecto o bien por una específica definida por el proceso
- Una señal puede:
  - **Manejarse:** hay que instalar el manejador
  - **Enmascararse:** se difiere su tratamiento un tiempo
  - **Ignorarse:** no quiere ser informado de su ocurrencia