Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Práctica 2 - Parte 2

Rafael Menéndez de Llano Rozas

Guion

2

Sincronización de procesos

En esta parte adquirirás la visión del sistema como programador haciendo uso de las llamadas al sistema POSIX para la sincronización entre procesos y entre procesos y sistema usando el mecanismo de las señales.

Comandos de control de prioridad de procesos y asignación de cores

Cada proceso tiene asignado un modificador de prioridad que se llama número *nice*, y cuanto mayor es el número *nice* menor es la prioridad del proceso (de ahí el nombre, cuando aumentamos su valor estamos beneficiando a los demás procesos). Este número *nice* se puede cambiar con el uso de un comando (`man 1 nice`) o con una llamada al sistema (`man 2 nice`).

```
#include <unistd.h>
int nice(int inc);    // valores negativos solo para el superusuario
```

El comando se utiliza o bien de forma aislada y nos dará el *nice* por defecto (0), o bien precediendo a otro comando con la opción `-n` seguida de un número que puede ir de -20 a 19 (si es negativo deja de ser "*nice"*). Si el proceso ya se está ejecutando, tenemos una versión de `nice` que es `renice`, donde además indicaremos el PID del proceso al que queremos cambiar su prioridad. También sirve para grupos de procesos.

Ejemplo:

```
$nice ./a.out
$nice -n 10 ./a.out
$renice -n 15 22772
```

1) Ejecuta el `vi` en *background*. A continuación, ejecútalo anteponiendo `nice` con un valor determinado. Haz un `ps` (o `top`) con la opción adecuada (se vio en teoría) para mostrar las prioridades de los procesos que has ejecutado. Indica si has cambiado la prioridad del proceso. Intenta hacerlo con un valor negativo en un puesto fijo y en un portátil o Shell de W10. Observa la identificación del primer vi y cambia su prioridad.

Existen también llamadas al sistema para obtener y variar la prioridad de un proceso. Para `wich` se usa el parámetro PRIO_PROCESS, para `quien` se suele poner cero (el propio proceso) y la prioridad, al igual que `nice`, varía entre -20 y 19:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int wich, id_t quien);
int setpriority(int wich, id_t quien,  int prio);
```

A continuación, se describe un ejemplo de dos procesos que intentan variar su prioridad de distinta manera, una de forma cortés y otra no.

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int main()
{
    int prioridad, control;
    pid_t pid;
    puts("Solo un proceso");
    puts("Llamo a fork");
    pid = fork();
    if (pid == 0)
    {
        prioridad=getpriority(PRIO_PROCESS, 0);
        printf("Hijo pid %d con prioridad %d \n",getpid(),prioridad);
        control=setpriority(PRIO_PROCESS, 0, 10);
        prioridad=getpriority(PRIO_PROCESS, 0);
        printf("Hijo   pid   %d   con   prioridad   %d   control   %d\n",
        getpid(),prioridad, control);
    }

    else if (pid > 0)
    {
        prioridad=getpriority(PRIO_PROCESS, 0);
        printf("Padre pid %d con prioridad %d \n",getpid(),prioridad);
        control=setpriority(PRIO_PROCESS, 0, -10);
        prioridad=getpriority(PRIO_PROCESS, 0);
        printf("Padre   pid   %d   con   prioridad   %d   control   %d\n",
        getpid(),prioridad, control);
    }
    else
         perror("El fork no ha funcionado");
}
```

Para medir la carga de trabajo de los procesos y obtener los datos de contabilidad de los mismos se suelen utilizar dos comandos que son el `top` y el `w`. El primero pone en orden de uso de CPU a todos los procesos que se están ejecutando, y los va refrescando cada cierto tiempo modificable (-d); el segundo pone la cabecera de `top` con los tiempos estáticos de uso por usuario y las medias de carga previas.

2) Descárgate el programa anterior (`prioridad.c`) y pruébalo. Ejecuta el comando `top` y el `w` y descubre y analiza los datos reflejados en pantalla. Ejecuta los comandos `free` y `uptime` , advierte las coincidencias. ¿de dónde cogen estos comandos la información que presentan?

Como se ha visto en teoría, para saber si un sistema utiliza *hyper-threading* (HP) existe el comando `lscpu`, que además nos puede ofrecer toda la información sobre la arquitectura del procesador (fabricante, frecuencia, modelo, familia, revisión), los modos de operación, el número de CPUs, su identificación, los hilos por núcleo, el número de sockets o si hay virtualización o no.

Aprovechando la información que ofrece ese comando podemos obtener o asignar la afinidad de los procesos a los cores (físicos o lógicos con HP) con el comando `taskset`. Si sólo queremos saber el número de core asignado a un proceso con un PID utilizaremos:

```
$ taskset -p PID            (nos lo dará en modo máscara)
$ taskset -cp PID           (nos lo dará en modo lista)
```

Si quisiéramos cambiar la afinidad de un proceso existente, deberíamos incluirla como argumento, de nuevo en modo máscara o lista:

```
$ taskset -p 0x1 PID        (cambiar afinidad con máscara)
$ taskset -cp 0   PID       (en forma de lista)
$ taskset -cp 0,2,5-7 PID   (la lista en otros formatos)
```

Por último, si quisiéramos lanzar un proceso con la afinidad fijada usaríamos (máscara o lista):

```
$ taskset 0x1 ejecutable    (lanzar ejecutable al procesador 0)
$ taskset -c 0 ejecutable   (lanzar ejecutable al procesador 0)
```

3) Descubre qué tipo de arquitectura tienes en el sistema. De acuerdo con esto, ejecuta un editor `vi` y páralo. Mira su PID. Indica su afinidad en modo máscara y lista. Cambia su afinidad en los dos modos y compruébalo. Por último, lanza otro `vi` en los dos modos y con diferentes *cores* y compruébalo.

Señales: sincronización de procesos

En la sesión anterior se ha visto cómo se pueden crear procesos a través de llamadas al sistema, ahora se verá como estos procesos se pueden sincronizar y comunicar; para ello disponemos de muchos mecanismos, los fundamentales son las señales (*signals*), las tuberías (*pipes*) y las tuberías con nombre (FIFO), estas dos últimas se verán en la última sesión.

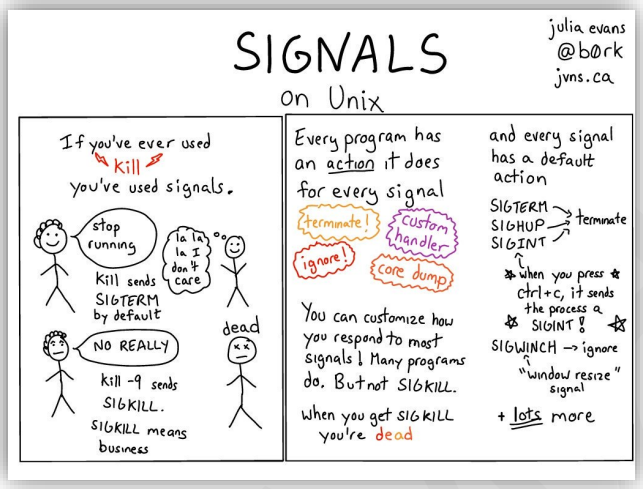
Las señales se pueden considerar como interrupciones software que, o bien indican un evento que ocurre en el sistema, o bien, una sincronización entre los procesos con mediación del S.O. Son totalmente asíncronas, es decir, cuando a un proceso le llega una señal, deja lo que está haciendo y ejecuta un código separado de tratamiento de la señal y después vuelve al punto donde dejó la ejecución. De hecho y sin saberlo, las habéis estado utilizando habitualmente con la *shell* cuando queríais abortar (control-c) un programa muy largo o que no funcionaba como esperabais o cuando usabais `kill` para matar un proceso. La secuencia de hechos era la siguiente:

1. La parte del kernel responsable del terminal ve el carácter de interrupción `control-c`.
2. El kernel envía una señal (llamada SIGINT número 2) a todos los procesos que reconocen al terminal como su terminal de control.
3. Cuando el proceso recibe la señal, realiza la acción por defecto asociada a una señal, en este caso la finalización (la *shell* también recibe esa señal, pero la ignora para poder continuar).

El otro ejemplo de utilización habitual de las señales (repasa el apartado de ejecución en *background* del guion 1) es el comando `kill` que lo que realmente hace es enviarle la señal SIGTERM (señal de terminal número 15) al proceso, que normalmente le obliga a terminar. Este comando tiene una opción

numérica que precisamente es el número de señal a enviar, así, cuando se utiliza la opción -9, realmente se estaba enviado la señal SIGKILL (señal número 9) que causa la muerte del proceso. Esta señal difiere del resto en que es la única para la que un proceso no puede estar preparado, esto se hace así por seguridad y tener un mecanismo por el que poder matar a un proceso de forma segura.

Las señales no pueden transportar datos, pero se han definido varias según el uso para el que se destinan, que es una forma indirecta de transmitir información.



Tipos de señales

Cada tipo de señal es conocida en el sistema a través de un entero. Para que el código sea más legible, existe un fichero de cabecera donde cada señal está definida por un nombre que indica el propósito para el que, normalmente, va a ser empleada. El fichero se llama `signal.h` y con la directiva `#define` se hacen las asociaciones entre los enteros y los nombres.

Las señales que están definidas en tu sistema se pueden conseguir haciendo:

```
$ man -S7 signal   (sección 7 de macros y convenios) o
$ cat /usr/include/bits/signum.h     (si no está puedes buscar el fichero)
$ kill -l
```

La numeración varía de un sistema a otro, de ahí la importancia de usar el mnemónico de la cabecera.

4) Busca las señales que tienes en tu sistema, su numeración y el total disponible. Si no existe ese camino para `signum.h` , averigua dónde está (`x86_64-linux-gnu`), usa lo aprendido en el guion 1. ¿cuáles crees que has usado?

Las señales, cuando son recibidas por un proceso, pueden causar varios efectos:

1. Terminación normal.
2. Terminación anormal.
3. Activación.
4. Parada.

Dado que el efecto por defecto es la terminación, debe existir alguna forma de recibir señales sin que esto ocurra. Es la recepción de señales.

Recepción de señales

Para recibir señales existe una llamada al sistema perteneciente al estándar del ANSI C (C89, C99 y POSIX.1-2001) denominada *signal*, que permite manejar las señales recibidas por un proceso.

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t manejador);;
```

El primer parámetro, `signum`, identifica la señal en cuestión, por tanto, puede ser cualquiera de las que has encontrado en el apartado anterior; excepto `SIGKILL`, que no puede ser tratada por motivos de seguridad (cualquier proceso en última instancia puede ser abortado). El segundo parámetro, `manejador`, es una función construida por nosotros, pero respetando el tipo definido para ella, que se ejecutará cuando llegue la señal al proceso, es como una especie de rutina de atención a una interrupción. Tiene el tipo de dato `sighandler`. Esta función describe la acción que estará asociada con la señal y puede tomar uno de los tres valores siguientes:

1. La dirección de una función de tipo `void` que toma un entero como argumento y no devuelve nada (`void (*función_manejadora)(int)`). Este argumento es el número de la señal recibida. Tan pronto como el proceso recibe la señal, el control se le pasará a la función, independientemente del punto del programa en el que se encuentre. Cuando termina, el proceso continuará por donde iba.

2. SIG_IGN. Corresponde a un símbolo especial que simplemente significa que cuando llegue la señal no se haga nada (ignorar). El símbolo está definido en el fichero de cabecera `signal.h`.

3. SIG_DFL. Es otro nombre simbólico que restaura la acción por defecto (*default*) del sistema. Normalmente será terminar y además puede hacer un *core dump*.

El valor retornado por `signal`, es un puntero a `sighandler` (una función de tipo `void` que toma un entero de argumento) al igual que el segundo parámetro. Dado que no devuelve un entero, podemos saber si la llamada ha fallado comparándola con la constante SIG_ERR también definida en `signal.h`.

Cuando se trabaja con señales hay que tener en cuenta los siguientes puntos:

1. El nombre de una función es directamente un puntero a función, es decir, una dirección a código. Algo análogo a los *arrays* pero con código, no con datos.
2. Las funciones de atención deben ser lo más cortas posibles y el código incluido debe ser re-entrante (de sólo lectura).
3. Las señales son heredadas por el proceso hijo (`fork`). Cuando se ejecuta un `exec` se toman las acciones por defecto del sistema a no ser que la señal esté ignorada. Si es así, seguirá ignorada por el proceso hijo (como se verá después, se hereda la máscara del padre).
4. Las llamadas al sistema no pueden ser interrumpidas por la llegada de una señal, salvo las lecturas/escrituras sobre terminal y las llamadas “lentas”, pensadas para ser interrumpidas, como `wait`, `sleep` y `pause`. Cuando se vuelva, será a la siguiente instrucción.
5. La función `signal` sólo actúa una vez, por lo tanto, si una señal llega dos veces y sólo se ha ejecutado una `signal` la segunda llegada de la señal no será tratada. Por eso, si se quiere tratar continuamente a una señal se deberá ejecutar la llamada `signal` en la propia función de manejo (salvo si se ignora que vale para siempre). Este comportamiento puede variar de un tipo de sistema a otro, de hecho, actualmente una llamada a `signal` sirve para siempre.

6. Las señales no pueden ser almacenadas, por lo que si llegan varias a la vez se pueden perder.
7. La llegada de una señal SIGINT a un proceso padre se propaga a todos sus hijos.

La utilización de la llamada queda reflejadas en este ejemplo donde un proceso captura la señal de interrupción SIGINT (pulsación de *control-C*) y entonces pinta un mensaje en pantalla:

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

void manejador(int);       /* prototipo manejador de senal */

int main()
{
        int i;

        /* si la interrupcion llega antes se abortara el proceso */
        if (signal(SIGINT, manejador) == SIG_ERR)
                perror("error en signal");

        /* si llega despues se ira a ejecutar la funcion */
        for(i=1;i<5;i++)
        {
                printf("Duermo %d segundo \n", i);
                sleep(1);
        }
}

void manejador(int tipo)
{
        printf("La senal cogida es de tipo %d \n",tipo);
}
```

5) Descárgate el programa `controlc.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Crea uno nuevo poniendo un `sleep` antes de `signal` y pulsa `control-c` para que veas que se aborta. Vuelve al original e indica cuántas veces puedes pulsar `control-c` antes de que acabe. ¿es más de una? Modifica el programa para que puedas pulsar varias veces `control-c` en cualquier sistema (por ejemplo, uno que no rearme el manejador) y que además la función de atención no pueda ser llamada dentro de la propia función de atención para que no se aniden.

Si deseamos que un proceso ignore la señal de interrupción SIGINT basta que introduzcamos la siguiente línea en nuestro programa:

```
signal(SIGINT, SIG_IGN);
```

para que la señal de interrupción no tenga ningún efecto. Esta es la técnica que utiliza la *shell* para evitar que los procesos en *background* sean parados cuando el usuario pulsa la tecla de interrupción. Si se desea restaurar la situación por defecto, basta con introducir:

```
signal(SIGINT, SIG_DFL);
```

Envío de señales

Si se pueden recibir señales para sincronizar procesos lo lógico es que se puedan enviar. La llamada al sistema para el envío, tiene el mismo nombre que el comando `kill` y es evidente el efecto que produce en el proceso receptor si no está preparado. El prototipo de la llamada es:

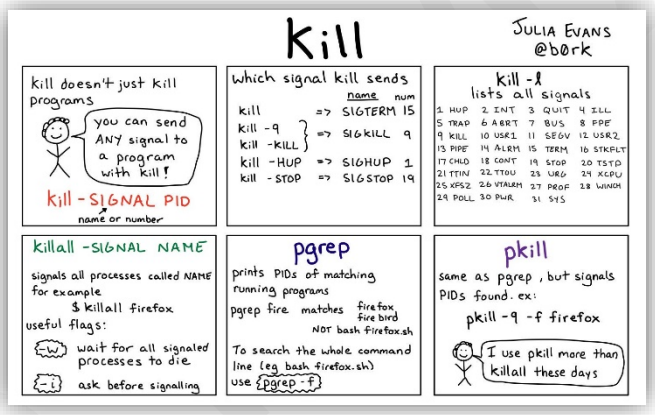
```
#include <sys/types.h>
#include <signal.h>

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

El argumento `pid` determina el proceso o procesos a los que les será enviada la señal `sig`. Por ejemplo:

```
kill(1234, SIGTERM);
```

significa "envía la señal SIGTERM al proceso cuyo identificador es 1234". Para conseguir las identificaciones de los procesos se usan las llamadas vistas en la sesión 1, no existiendo ningún impedimento para que un proceso pueda enviarse señales a sí mismo. Dado el efecto de las señales, sólo funcionan si los identificadores del usuario de ambos procesos: receptor y emisor, coinciden (salvo el *superusuario*).



Para mandar señales a un grupo de procesos, el PID puede tomar los siguientes valores:

0: Con lo que la señal es enviada a todos los procesos que tengan el mismo identificador de grupo, incluido el emisor.

-1: Si el productor tiene la identificación efectiva del *superusuario*, entonces la señal se mandará a todos los procesos del sistema con excepción de algunos protegidos. En POSIX.1, está opción está indefinida.

<0: Si es menor que cero, pero distinto de -1, la señal será enviada a todos los procesos que tengan el identificador de grupo igual su valor absoluto.

Ahora que sabemos cómo mandar y recibir señales, podemos usar las llamadas `signal` y `kill` para sincronizar dos procesos (ping-pong) que escriben por la salida estándar:

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

int veces = 0;     /* variable global */
```

```
void padre(int);   /* funciones manejadoras para padre e hijo */
void hijo(int);

int main()
{
        pid_t creado, ppid;
        signal(SIGUSR1, padre);    /* señal de usuario SIGUSR1*/


        switch(creado = fork())
        {
        case -1:
                perror("fallo en el fork");
                exit(1);
        case 0:     /* el hijo */
                signal(SIGUSR1, hijo); /* cambia la función */
                ppid = getppid();   /* pid del padre */
                for(;;)    /* infinito */
                {
                    sleep(1); /* que el padre haga el pause */
                    kill(ppid, SIGUSR1);  /* mando al padre */
                    pause();           /* espero la respuesta */
                }
        default :    /* el padre */
                for(;;)    /* infinito */
                {
                      pause();   /* espero la señal del hijo */
                      sleep(1); /* que el hijo haga pause */
                      kill(creado, SIGUSR1); /* aviso al hijo */
                }
        }
}
/* en algunos sistemas no hace falta rearmar la señal */
/* funcion de manejo de la signal para el padre */
void padre (int senal)
{
       extern int veces;
       /* ambos procesos tienen una copia de veces */
       printf("El padre cogio la signal numero %d\n", ++veces);
      /* de nuevo para recibir la siguiente signal */
      signal(senal, padre);   /* pone SIGUSR1 para el padre */
}
/* funcion de manejo de la signal para el hijo */
void hijo (int senal)
{
       extern int veces;
       /* ambos procesos tienen una copia de veces */
      printf("El hijo cogio la signal numero %d\n", ++veces);
      /* de nuevo para recibir la siguiente signal */
      signal(senal, hijo);    /* pone SIGUSR1 para el hijo */
}
```

En este ejemplo, se crean dos procesos a través de una llamada a `fork`, y la respuesta se maneja con un `switch` en vez de un `if`. Ambos procesos escriben mensajes alternativamente en la salida estándar a través de la función de interrupción (que si tuviera los mismos mensajes podría ser la misma para ambos procesos) sincronizándose mediante el uso de `kill`. Como no hay nada que hacer mientras tanto, ambos procesos tienen que estar ejecutando una instrucción `pause` para recibir la señal y despertarse; y para asegurarse que el otro llega a ella un `sleep` (ver siguiente apartado).

6) Descárgate el programa `pingpong.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Cambia el programa (comenta una línea) para que los dos procesos tengan la misma función de atención. Cambia el programa anterior y comenta las líneas donde están los pauses. Compila y ejecuta. Indica qué ocurre.

7) Crea una copia del original y haz que el proceso hijo retome la función anterior que tenía al cabo de 5 vueltas (para ello debes usar lo que devuelve `signal` y guardarlo en una variable de tipo puntero a función como este prototipo: `void (*antiguo)(int)`). Cambia el programa y al cabo de 10 vueltas haz que recupere la acción por defecto y descubre qué le ocurre.

Utilización de las llamadas de tiempo

Además de `sleep` (sección 3 del manual) y sus variantes `nanosleep` (para threads) y `usleep` (para microsegundos), existen dos llamadas fundamentales para manejar el tiempo que son las llamadas `pause` y `alarm`.

La llamada a `pause` ya ha sido utilizada y es muy simple, lo único que hace es detener la ejecución de un proceso sin gastar CPU del sistema:

```
#include <unistd.h>
int pause(void);
```

De esta instrucción sólo se puede salir si el proceso recibe una señal de cualquier tipo. Si la señal no es manejada, la ejecución terminará, si es ignorada también la ignorará `pause`, y si la señal es manejada, `pause` devolverá -1 después de la ejecución de la función.

La otra llamada del sistema es `alarm` que sirve para manejar el *timer* de un proceso y hacer que éste reciba una señal de alarma (la número 14) al cabo de los segundos especificados.

El prototipo es:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Toma como argumento un número de segundos positivo y devuelve el número con el que estaba programada antes. Si el argumento es cero se desactiva.
Advierte la diferencia con `sleep`, que bloquea al proceso. En el caso de `alarm`, el proceso sigue ejecutándose y el reloj de la alarma continuará activo incluso después de un `exec` (después de un `fork`, la alarma se desactiva para el hijo).

Como ejemplo de uso de las dos funciones, vamos a ver un programa que sirva de agenda, pintando al cabo de un tiempo un mensaje en pantalla. Este programa tomará como entradas dos argumentos, el número de minutos a esperar y el mensaje a pintar y se ejecutará automáticamente en *background* para poder seguir usando la shell. Como los argumentos se toman en forma de string, habrá que hacer una conversión de ascii a int para convertir el tiempo.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#define beep   '\007'        /* caracter de pitido */
#define TRUE   1
#define FALSE  0

int alarma = FALSE;             /* inicializacion */
/* se puede declarar antes y no poner prototipo, no toma entero */
void coger()
{
        alarma = TRUE;
}
```

```
int main(int cuenta, char *argumentos[])
{
        int segundos, creado, j;
        void coger();          /* al estar antes opcional */
        if (cuenta < 3)         /* numero de argumentos */
        {
                perror("agenda: Error argumentos");
                exit(1);    /* error 1 */
        }
              /* atoi esta en stdlib ver cuadernillo de C*/
        if ((segundos=atoi(argumentos[1])*60) <= 0)
        {
                perror("agenda: tiempo no valido");
                exit(2);    /* error 2 */
        }
        switch(creado = fork())
        {
                case -1:
                        perror("Agenda: error en fork");
                        exit(3);
                case 0:   /* en background */
                        break;
                        default :
                        printf("Agenda: process-id %d\n", creado);
                            /* se mata al padre para que libere el terminal,
                            background, el hijo sigue */
                        exit(0);
        }
        signal(SIGALRM, coger);   /* se programa la alarma */
        alarm(segundos);
        pause();         /*   es   equivalente   a      sleep(segundos)   */
         /* me despierto por alarma y compruebo que ha sido la 14 */
        if (alarma == TRUE)
        {
                putchar(beep);
                for(j=2; j < cuenta; j++)      /* mensaje */
                printf("%s ", argumentos[j]);
                printf("\n");
         }
        exit(0);
}
```

8) Descárgate el programa `agenda.c` de la página, entiende todos sus aspectos (ver comentarios), compílalo y prueba su funcionamiento.

Señales seguras

Hasta ahora se han visto el funcionamiento clásico de las señales en los sistemas compatibles con ANSI C. Este tipo de funcionamiento podía tener puntos no seguros, pongamos dos ejemplos:

```
...
signal(señal, funcion);
...
funcion()
{...
   signal(señal, funcion);
...}
```

si la segunda señal llega cuando se ejecuta la función manejadora, pero antes que la llamada a `signal`, tendremos problemas ya que el programa se abortaría (si no se rearma solo).

```
...
signal(señal, funcion);
...
while (control == 0)
   pause();
...
funcion()
{...
   signal(señal, funcion);
    control = 1;
...}
```

si la señal llega entre el `while` y el `pause` en su cuerpo, en programa se quedaría colgado en ese `pause`.

Debido a estos problemas, se cambia el funcionamiento de las señales en varios sentidos:

1. Una vez que se ha hecho una llamada a `signal`, ya no se toma la acción por defecto, con lo cual el proceso está preparado para recibir otra señal de ese tipo.
2. Dentro de la función manejadora, la llegada de otra señal de ese tipo se bloquea automáticamente.
3. Si se ejecuta una llamada al sistema y en ese momento se recibe la señal, después de ejecutar el manejador, se puede restaurar la llamada o no. Con el System V siempre, con el BSD nunca. En el POSIX.1 no está definido.

En las versiones actuales que cumplen POSIX se permite una característica más, que es la de poder bloquear determinadas señales. Dentro del bloque de control del proceso (PCB) existe una máscara que nos dirá que señales nos interesa bloquear. Esta máscara está definida en la cabecera `signal.h` como de tipo `sigset_t`, y existen varias funciones (no son llamadas al sistema) para inicializarla y modificarla:

```
int sigemptyset(sigset_t *conjunto);
int sigfillset (sigset_t *conjunto);

int sigaddset (sigset_t *conjunto, int senal);
int sigdelset (sigset_t *conjunto, int senal);

int sigismember (const sigset_t *conjunto, int  senal);
```

Las dos primeras servirán para inicializar una máscara que se pasa por referencia, la primera poniéndola a cero (ninguna señal bloqueada) y la segunda a uno (todas bloqueadas). Después hay dos funciones para añadir un bloqueo o quitarlo, que toman la máscara por referencia y el nombre o número de la señal. La quinta llamada servirá para saber si una señal está puesta a bloqueada o no (devolverá un dato de tipo verdadero o falso –`int`–).

Una vez definida la máscara, habrá que comunicárselo al sistema a través de esta llamada:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *new, sigset_t *anterior);
```

En el tercer argumento se devuelve la máscara anterior a la llamada (a no ser que se pase un NULL). Si el segundo argumento `new` tiene una dirección a una máscara válida (es distinto de NULL, si tiene NULL servirá sólo para preguntar por el conjunto que está puesto), se tomarán las acciones como indica el primer argumento `how` en forma de constantes definidas en `signal.h`:

1. SIG_BLOCK. Se supone que hay una máscara puesta en el proceso para la recepción de las señales, cuando se utiliza esta constante se está mezclando (OR) la que hay puesta con la que se utiliza en el segundo argumento.

2. SIG_UNBLOCK. En este caso el segundo argumento deberá contener las señales que queremos desbloquear.
3. SIG_SETMASK. En este caso se pone la máscara del segundo argumento sin tener en cuenta la anterior.

Existe una llamada al sistema para decirnos las señales que tiene pendiente un proceso, normalmente se ejecutará antecediendo a la función `sigismember`.

Esta llamada es:

```
int sigpending (sigset_t  *conjunto);
```

que devolverá las señales pendientes en formato de máscara.

En el siguiente ejemplo definiremos una máscara, bloquearemos la señal SIGINT y después veremos si la tenemos pendiente o no:

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

int main()
{
        sigset_t   nueva_mascara, vieja_mascara, pendiente;

        /* preparo la mascara: la borro y pongo SIGINT*/
        sigemptyset(&nueva_mascara);
        sigaddset(&nueva_mascara, SIGINT);

       /* fase 1 */
        /* comunico la mascara al sistema mezclandola OR */
        if(sigprocmask(SIG_BLOCK,&nueva_mascara, &vieja_mascara)<0)
              perror("Error al poner la mascara");

        puts("Puedes pulsar el control C, fase 1");
        sleep(5);  /* 5 segundos para pulsar */

        /* miro si han hecho control-C en los 5 segundos */
        if (sigpending(&pendiente) < 0)
              perror("error en pendiente");

        if (sigismember(&pendiente, SIGINT))
              puts("tenemos un control C pendiente");


      /* fase 2 */
        /* vuelvo a la situacion anterior */
        puts("Voy a desbloquear senal");

        if (sigprocmask(SIG_SETMASK, &vieja_mascara, NULL) < 0)
              perror("Error al volver a la situacion original");

        puts("desbloqueada senal");
        puts("puedes pulsar el control-C fase 2");
        sleep(5);   /* si pulsamos control-C terminara */
        puts("Termine sin control C");
}
```

9) Descárgate el programa `bloqueo.c` de la página, entiéndelo, compílalo y prueba su funcionamiento pulsando `control-C` en las dos fases del programa. Indica si ves alguna diferencia.

Ahora modificaremos el anterior añadiendo una función manejadora:

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

int main()
{
   int creado, ppid;
   void f_manejadora();          /* funciones manejadoras */
   sigset_t     nueva_mascara, vieja_mascara, pendiente;

   if (signal(SIGINT, f_manejadora) == SIG_ERR)
      perror("Error al senalar el control-c");

   sigemptyset(&nueva_mascara);
   sigaddset(&nueva_mascara, SIGINT);

   if (sigprocmask(SIG_BLOCK, &nueva_mascara, &vieja_mascara) < 0)
      perror("Error al poner la mascara");
   sleep(5);     /* 5 segundos para pulsar  */
   if (sigpending(&pendiente) < 0)
      perror("error en pendiente");
   if (sigismember(&pendiente, SIGINT))
      puts("tenemos un control C pendiente");

    /* volvemos a la situacion anterior */
   if (sigprocmask(SIG_SETMASK, &vieja_mascara, NULL) < 0)
      perror("Error al volver a la situacion original");
   puts("desbloqueada senal");
   sleep(5);    /* aqui si pulsamos control-c */
}

void f_manejadora()      /* solo para una vez */
{
   puts("desde la manejadora llego la senal");
   if (signal(SIGINT, SIG_DFL) == SIG_ERR)        /* 1 vez */
      perror("Error al volver por defecto");
}
```

10) Descárgate el programa `bloqueo2.c` de la página, entiéndelo, compílalo y prueba su funcionamiento pulsando `control-C` en las dos fases del programa. Date cuenta que, al poner en la funciona manejadora la opción por defecto, el programa termina si se pulsa en la segunda fase.

Hasta ahora hemos usado `signal` para recibir señales, pero la verdadera llamada POSIX para hacerlo es `sigaction`. Podríamos considerar que `signal` es un caso sencillo de esta llamada. El prototipo es:

```
int sigaction (int senal, const struct sigaction *nueva_accion,
                          struct sigaction *accion_anterior);
```

En el primer argumento se dice la señal que se quiere tratar, en el segundo la acción que queremos tomar (sin modificarlo) y en el tercero la acción que estaba antes de llamarla.
La acción viene dada por la siguiente estructura:

```
struct sigaction {
   void  (*sa_handler)(int);  /* manejador, SID_IGN o SIG_DFL  */
   sigset_t  sa_mask;         /* señales adicionales a bloquear */
   int sa_flags;              /* opciones de señal */
};
```

El primer campo es el mismo que el de la llamada a `signal`, es decir, la dirección de la función que hace de manejador, o se ignorará o se tomará la acción por defecto. El segundo campo es la máscara adicional de bloqueos, que sólo actuará cuando ponemos como primer campo la dirección de un manejador, después de que éste es ejecutado, la máscara vuelve al estado anterior (es una forma de evitar que cuando se esté ejecutando el manejador se traten otras señales) y el tercero son las opciones de la acción. Los más habituales son:

- **SA_NOCLDSTOP** ignorar los stop de los hijos si la señal es SIGCHLD.
- **SA_RESTART** se rearranca la system call interrumpida.
- La misma señal se bloquea por defecto en el manejador salvo **SA_NODEFER** (SA_NOMASK)**.**
- **SA_RESETHAND** (SA_ONESHOT) hace que sólo sirve una vez el manejador.

En el siguiente ejemplo se muestra su utilización, donde se intenta simular el comportamiento de las señales en un sistema clásico (no se rearmaban y servían para una vez):

```
#include <signal.h>
#include <stdio.h>
void coger(int);      /* prototipo manejador */

int main()
{
       int i;
       struct sigaction accion, vieja;

       /* relleno los campos de la accion */
       accion.sa_handler=coger;  /* la dirección de la función */
       sigemptyset(&accion.sa_mask);    /* no quiero mascaras */

       /* este es el comportamiento clásico las opciones con OR */
       accion.sa_flags=SA_NOCLDSTOP|SA_RESTART|SA_ONESHOT|SA_NOMASK;

       /*llamo a el equivalente a signal con la accion construida*/
       if (sigaction(SIGINT, &accion, &vieja) <0 )
              perror("error en sigact");



       /* si llega ahora se ira a ejecutar la funcion coger */
       for(i=1;i<5;i++)
       {
              printf("Duermo %d segundo \n",i);
              sleep(1);
       }
}

void coger(int tipo)
{
       printf("La senal cogida es de tipo %d \n",tipo);
}
```

11) Descárgate el programa `sigac.c` de la página, entiéndelo, compílalo y prueba su funcionamiento. Usa `man` para darte cuenta de la complejidad de la llamada al sistema, dado que la acción tiene más campos (uno de ellos es otra estructura más amplia y compleja) y que hay más opciones de las indicadas.