FECHA DE ENTREGA: SEMANA DEL 18 AL 22 DE MARZO
(HORA LÍMITE DE ENTREGA: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)

La segunda práctica se va a desarrollar en 3 semanas con el siguiente cronograma:

- Semana 1: Señales.
- Semanas 2 y 3: Semáforos y concurrencia.

Los ejercicios correspondientes a esta segunda práctica se van a clasificar en:

- **APRENDIZAJE**, se refiere a ejercicios que es altamente recomendable realizar para el correcto seguimiento de los conocimientos teóricos que se presentan en la unidad didáctica.
- **ENTREGABLE**, estos ejercicios son obligatorios y deben entregarse correctamente implementados y documentados.

SEMANA 1

Señales

Las señales son una forma limitada de comunicación entre procesos. Como comparación se puede decir que las señales son a los procesos lo que las interrupciones son al procesador. Cuando un proceso recibe una señal, detiene su ejecución, bifurca a la rutina de tratamiento de la señal (que está en el mismo proceso) y luego, una vez finalizado, sigue la ejecución en el punto que había bifurcado anteriormente.

Las señales son usadas por el núcleo para notificar sucesos asíncronos a los procesos, como por ejemplo:

1. Si se pulsa `Ctrl+C`, el núcleo envía la señal de interrupción `SIGINT`.
2. Excepciones de ejecución.

Por otro lado, los procesos pueden enviarse señales entre sí mediante la función `kill`, siempre y cuando tengan el mismo UID.

Cuando un proceso recibe una señal puede reaccionar de tres formas diferentes:

- Ignorar la señal, con lo cual es inmune a la misma.
- Invocar a la rutina de tratamiento de la señal por defecto. Esta rutina no la codifica el programador, sino que la aporta el núcleo. Por lo general suele provocar la terminación del proceso mediante una llamada a `exit`. Algunas señales no solo provocan la terminación del proceso, sino que además hacen que el núcleo genere, dentro del directorio de trabajo actual del proceso, un fichero llamado `core` que contiene un volcado de memoria del contexto del proceso.

- Invocar a una rutina propia que se encarga de tratar la señal. Esta rutina es invocada por el núcleo en el supuesto de que esté montada, y será responsabilidad del programador codificarla para que tome las acciones pertinentes como tratamiento de la señal.

Cada señal tiene asociado un número entero y positivo y, cuando un proceso le envía una señal a otro, realmente le está enviando ese número. En el fichero de cabecera `<signal.h>` están definidas las señales que puede manejar el sistema; la Tabla 1 muestra un breve resumen extraído de la sección 7 del manual de `signal` (que se puede ver ejecutando `man 7 signal`).

Tabla 1: Señales UNIX.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/breakpoint trap
SIGABRT	6	Core	Abort signal from abort(3)
SIGBUS	7	Core	Bus error (bad memory access)
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGUSR1	10	Term	User-defined signal 1
SIGSEGV	11	Core	Invalid memory reference
SIGUSR2	12	Term	User-defined signal 2
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGSTKFLT	16	Term	Stack fault on coprocessor (unused)
SIGCHLD	17	Ign	Child stopped or terminated
SIGCONT	18	Cont	Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at terminal
SIGTTIN	21	Stop	Terminal input for background process
SIGTTOU	22	Stop	Terminal output for background process
SIGURG	23	Ign	Urgent condition on socket (4.2BSD)
SIGXCPU	24	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	25	Core	File size limit exceeded (4.2BSD); see setrlimit(2)
SIGVTALRM	26	Term	Virtual alarm clock (4.2BSD)
SIGPROF	27	Term	Profiling timer expired
SIGWINCH	28	Ign	Window resize signal (4.3BSD, Sun)
SIGIO	29	Term	I/O now possible (4.2BSD)
SIGPWR	30	Term	Power failure (System V)
SIGSYS	31	Core	Bad system call (SVr4); see also seccomp(2)

Envío de Señales

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

- Manda señales entre procesos.
- Argumentos:
  - `pid` es el PID del proceso al que se enviará la señal:

        - `pid>0`, es el PID del proceso al que le enviamos la señal.
        - `pid=0`, la señal es enviada a todos los procesos que pertenecen al mismo grupo que el proceso que la envía.
        - `pid=-1`, la señal es enviada a todos aquellos procesos cuyo identificador real es igual al identificador efectivo del proceso que la envía. Si el proceso que la envía tiene identificador efectivo de superusuario, la señal es enviada a todos los procesos, excepto al proceso 0 (swapper) y al proceso 1 (init).
        - `pid<-1`, la señal es enviada a todos los procesos cuyo identificador de grupo coincide con el valor absoluto de `pid`.
    - `sig` es un entero con el número de la señal que queremos enviar. Si `sig=0` (señal nula) se efectúa una comprobación de errores, pero no se envía ninguna señal.
- En todos los casos, si el identificador efectivo del proceso no es el del superusuario o si el proceso que envía la señal no tiene privilegios sobre el proceso que la va a recibir, la llamada a `kill` falla.
- Retorno: `0` si el envío es correcto, `-1` en caso de error.

Ejercicio 1 **(APRENDIZAJE)** Ejecuta en línea de comando la orden `kill` con la opción `-l`:

```
$  kill -l
```

¿Qué se obtiene?

Ejercicio 2 **(ENTREGABLE)(1 pto)** Escribe un programa en C, `ejercicio2.c`, que creará 4 procesos hijos en un bucle de forma paralela. Cada hijo, imprimirá el mensaje `"Soy el proceso hijo <PID>"`, después dormirá 30 segundos, imprimirá el mensaje `"Soy el proceso hijo <PID> y ya me toca terminar."`. Tras imprimir este mensaje, el hijo finalizará su ejecución. El padre, tras crear un hijo, dormirá 5 segundos y después enviará la señal `SIGTERM` al hijo que acaba de crear. A continuación, creará el siguiente hijo.

¿Qué mensajes imprime cada hijo? ¿Por qué?

Tratamiento de Señales

```
int sigaction(int signum, const struct sigaction *act, struct sigaction
 *oldact);
```

- Establece la captura de señales. Cambia el manejador al definido (puede ser una función, ignorar o comportamiento por defecto), siempre que la señal especificada se pueda capturar.
- Argumentos:
    - `signum` es un entero con el número de la señal que se va a capturar.
    - `act` es el puntero a una estructura de tipo `sigaction` que define el comportamiento cuando se reciba la señal. En concreto, tiene los siguientes campos:
        - `void (*sa_handler)(int)`, la acción que se tomará al recibir la señal, y puede tomar tres clases de valores:

- `SIG_DFL`, indica que la acción a realizar cuando se recibe la señal es la acción por defecto asociada a la señal (manejador por defecto). Por lo general, esta acción consiste en terminar el proceso y en algunos casos también incluye generar un fichero `core`.
- `SIG_IGN`, indica que la señal se debe ignorar.
- La dirección de la rutina de tratamiento de la señal (manejador suministrado por el usuario); la declaración de esta función debe ajustarse al modelo:

  ```
  void handler (int sig [, int code, struct sigcontext *scp]);
  ```

  Cuando se recibe la señal `sig`, el núcleo es quien se encarga de llamar a la rutina `handler` pasándole el parámetro `sig` (el número de la señal). La llamada a la rutina `handler` es asíncrona, lo cual quiere decir que puede darse en cualquier instante de la ejecución del programa. De hecho, existen una serie de buenos hábitos a la hora de implementar funciones manejadoras, basadas solo en una serie de operaciones básicas y la llamada a funciones seguras (para más información, consultar `man 7 signal-safety`) que garantizan que el sistema no quede en un estado inconsistente. Una vez ejecutado el manejador, la ejecución continúa en el punto en el que bifurcó al recibir la señal, saliendo además de cualquier llamada bloqueante al sistema.

  Los valores `SIG_DFL`, `SIG_IGN` y `SIG_ERR` son direcciones de funciones ya que los debe poder gestionar `signaction`:

  ```
  #define SIG_DFL ((void (*) ( ))  0)
  #define SIG_IGN ((void (*) ( ))  1)
  #define SIG_ERR ((void (*) ( )) -1)
  ```

  La conversión explícita de tipo que aparece delante de las constantes `0`, `1` y `-1` fuerza a que estas constantes sean tratadas como direcciones de inicio de funciones. Estas direcciones no contienen ninguna función, ya que en todas las arquitecturas UNIX son zonas reservadas para el núcleo. De hecho, la dirección `-1` no tiene existencia física.

  - `void (*sa_sigaction)(int, siginfo_t *, void *)`, la acción que se tomará al recibir la señal pero definida con este prototipo extendido, que recibe información adicional además del número de la señal (se pueden ver más detalles consultando el manual).
  - `sigset_t sa_mask`, una máscara de señales adicionales que se bloquearán durante la ejecución del manejador.
  - `int sa_flags`, banderas para modificar el comportamiento.

- `oldact` es el puntero a una estructura de tipo `sigaction` donde se almacenará la acción establecida previamente.

- Retorno: `0` si todo es correcto, `-1` en caso de error.

Ejemplo de uso:

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

/* manejador: rutina de tratamiento de la señal SIGINT. */
void manejador(int sig) {
    printf("Señal número %d recibida \n", sig);
    fflush(stdout);
}

int main(void) {
    struct sigaction act;

    act.sa_handler = manejador;
    sigemptyset(&(act.sa_mask));
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    while(1) {
        printf("En espera de Ctrl+C (PID = %d)\n", getpid());
        sleep(9999);
    }
}
```

Al ejecutar este programa, cada vez que se pulsa `Ctrl+C` aparece el mensaje por pantalla. Para volver a restaurar el comportamiento por defecto asociado a `Ctrl+C` tras la primera captura, se puede utilizar la bandera `SA_RESETHAND` en `sa_flags`.

Ejercicio 3 **(ENTREGABLE)(0,5 ptos)** Dado el código de ejemplo anterior, contesta a las siguientes cuestiones:

a) ¿La llamada a `signaction` supone que se ejecute la función `manejador`?

b) ¿Cuándo aparece el `printf` en pantalla?

c) ¿Qué ocurre por defecto cuando un programa recibe una señal y no la tiene capturada?

d) Modifica el programa anterior en un nuevo `ejercicio3d.c` que capture la señal `SIGKILL` y cuya función manejadora escriba `"He conseguido capturar SIGKILL"`. ¿Por qué nunca sale por pantalla `"He conseguido capturar SIGKILL"`?

Espera de Señales

```c
int pause(void);
```

- Bloquea al proceso que lo invoca hasta que llegue una señal. No permite especificar el tipo de señal por la que se espera. Solo la llegada de cualquier señal no ignorada ni enmascarada sacará al proceso del estado de bloqueo.
- Retorno: `-1`. En otras llamadas al sistema, esta terminación es una condición de error, en este caso es su forma correcta de operar.

Ejemplo de uso:

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

/* manejador_SIGTERM: saca un mensaje por pantalla y termina el proceso. */
void manejador_SIGTERM(int sig) {
    printf("Terminación del proceso %d a petición del usuario \n", getpid());
    fflush(stdout);
    exit(EXIT_SUCCESS);
}

/* manejador_SIGUSR1: presenta un número aleatorio por pantalla. */
void manejador_SIGUSR1(int sig) {
    printf("%d\n", rand());
    fflush(stdout);
}

int main(void) {
    struct sigaction act;

    sigemptyset(&(act.sa_mask));
    act.sa_flags = 0;

    /* Se arma la señal SIGTERM. */
    act.sa_handler = manejador_SIGTERM;
    if (sigaction(SIGTERM, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    /* Se arma la señal SIGUSR1. */
    act.sa_handler = manejador_SIGUSR1;
    if (sigaction(SIGUSR1, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    /* Se muestra el PID para facilitar el uso de kill.
También se podría usar killall. */
    printf("PID: %d. Se esperan las señales SIGUSR1 o SIGTERM\n", getpid());

    while(1)
        pause(); /* Bloquea el proceso hasta que llegue una señal. */
}
```

Para ejecutar este programa y que muestre los números aleatorios, debemos enviarle la señal `SIGUSR1` con la orden `kill`:

```
$  kill -s USR1 <PID>
```

Podemos terminar la ejecución enviándole la señal `SIGTERM` desde otra terminal:

```
$  kill -15 <PID>
```

Ejercicio 4 **(ENTREGABLE)(2 ptos)** Escribe un programa en C, `ejercicio4.c`, que satisfaga los siguientes requisitos para realizar una competición entre procesos que tratan de responder lo antes posible a una señal:

- El proceso padre generará un proceso hijo, que será el proceso gestor.
- El proceso gestor creará `N_PROC` procesos hijos (los participantes en la competición) en un bucle, esperando tras crear cada proceso a que éste le notifique que está preparado enviándole la señal `SIGUSR2`.

- Cada participante en la carrera, una vez activo y con la señal armada, imprimirá un mensaje y avisará al gestor mediante la señal `SIGUSR2`.
- El proceso gestor, cuando haya creado los `N_PROC` procesos hijos y éstos estén listos para la competición, avisará al proceso padre de que está todo listo enviándole la señal `SIGUSR2`.
- El proceso padre mandará al grupo entero de procesos la señal `SIGSUSR1` (que será el inicio de la competición).
- Cuando los participantes en la carrera reciban la señal `SIGUSR1`, pintarán un mensaje de que ya han capturado la señal, y terminarán.
- Cuando el proceso gestor reciba `SIGUSR1`, terminará su ejecución sin dejar hijos huérfanos.
- El proceso padre esperará a que el proceso gestor termine y acabará él también.

Protección de Zonas Críticas

Máscaras

En ocasiones puede interesarnos proteger determinadas zonas de código contra la llegada de alguna señal. La máscara de señales de un proceso define un conjunto de señales cuya recepción será bloqueada. Bloquear una señal es distinto de ignorarla. Cuando un proceso bloquea una señal, ésta no será enviada al proceso hasta que se desbloquee o la ignore, lo que puede ayuda a garantizar que no se produzcan condiciones de carrera. Si el proceso ignora la señal, ésta simplemente se deshecha.

La máscara que indica las señales bloqueadas en un proceso o hilo determinado es un objeto de tipo `sigset_t` al que llamamos conjunto de señales y está definido en `<signal.h>`. Aunque `sigset_t` suele ser un tipo entero donde cada bit está asociado a una señal, no necesitamos conocer su estructura y estos objetos pueden ser manipulados con funciones específicas para activar y desactivar los bits correspondientes.

```
int sigfillset(sigset_t *set);
```

- Incluye en el conjunto referenciado por `set` todas las señales definidas. Si utilizamos este conjunto como máscara todas las señales serán bloqueadas.
- Argumentos:
  - `set` es un puntero al objeto `sigset_t` que se va a modificar.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

```
int sigemptyset(sigset_t *set);
```

- Excluye del conjunto referenciado por `set` todas las señales, desbloqueando así su recepción si utilizamos este conjunto como máscara.
- Argumentos:
  - `set` es un puntero al objeto `sigset_t` que se va a modificar.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

```
int sigaddset(sigset_t *set, int signum);
```

- Incluye en el conjunto referenciado por `set` la señal `signum`.

- Argumentos:
  - `set` es un puntero al objeto `sigset_t` que se va a modificar.
  - `signum` es el número de la señal que se va a añadir.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

```
int sigdelset(sigset_t *set, int signum);
```

- Excluye del conjunto referenciado por `set` la señal `signum`.
- Argumentos:
  - `set` es un puntero al objeto `sigset_t` que se va a modificar.
  - `signum` es el número de la señal que se va a excluir.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

```
int sigismember(const sigset_t *set, int signum);
```

- Comprueba si la señal `signum` forma parte del conjunto referenciado por `set`.
- Argumentos:
  - `set` es un puntero al objeto `sigset_t` en el que se va a buscar la señal.
  - `signum` es el número de la señal que se va a buscar.
- Retorno: `1` si la señal es miembro del conjunto, `0` si la señal no es miembro del conjunto, `-1` en caso de error.

Una vez creado un conjunto de señales podemos modificar la máscara de señales bloqueadas en un proceso con la siguiente función.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- Modifica la máscara de señales.
- Argumentos:
  - `how` es un entero que establece cómo se modificará la señal, pudiendo tomar los siguientes valores:
    - `SIG_BLOCK`, la máscara resultante es el resultado de la unión de la máscara actual y el conjunto.
    - `SIG_SETMASK`, la máscara resultante es la indicada en el conjunto.
    - `SIG_UNBLOCK`, la máscara resultante es la intersección de la máscara actual y el complementario del conjunto. Es decir, las señales incluidas en el conjunto quedarán desbloqueadas en la nueva máscara de señales.
  - `set` es un puntero al conjunto que se utilizará para modificar la máscara de señales.
  - `oldset` es el puntero al conjunto donde se almacenará la máscara de señales anterior, de manera que podremos usar este conjunto para restaurarla con posterioridad.

- Si el puntero `set` vale `NULL` es equivalente a consultar el valor de la máscara actual sin modificarla. En el caso de que el proceso sea multihilo se deberá utilizar en lugar de `sigprocmask` la función equivalente `pthread_sigmask`.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

Ejemplo de uso:

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

int main(void) {
    sigset_t set, oset;

    /* Máscara que bloqueará la señal por error en coma flotante y por
violación de segmento. */
    sigemptyset(&set);
    sigaddset(&set, SIGFPE);
    sigaddset(&set, SIGSEGV);

    /*Bloqueo de las señales SIGFPE y SIGSEV en el proceso. */
    if (sigprocmask(SIG_BLOCK, &set, &oset) < 0) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    while(1) {
        printf("En espera de señales (PID = %d)\n", getpid());
        printf("SIGFPE y SIGSEGV están bloqueadas\n");
        pause();
    }
}
```

`int sigpending(sigset_t *set);`

- Devuelve el conjunto de señales bloqueadas que se encuentran pendientes de entrega al proceso.
- Argumentos:
  - `set` es un puntero al conjunto en el que se va a incluir la lista de señales bloqueadas pendientes de entrega.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

La Ilustración 1 muestra un resumen del manejo de señales realizado por el sistema.

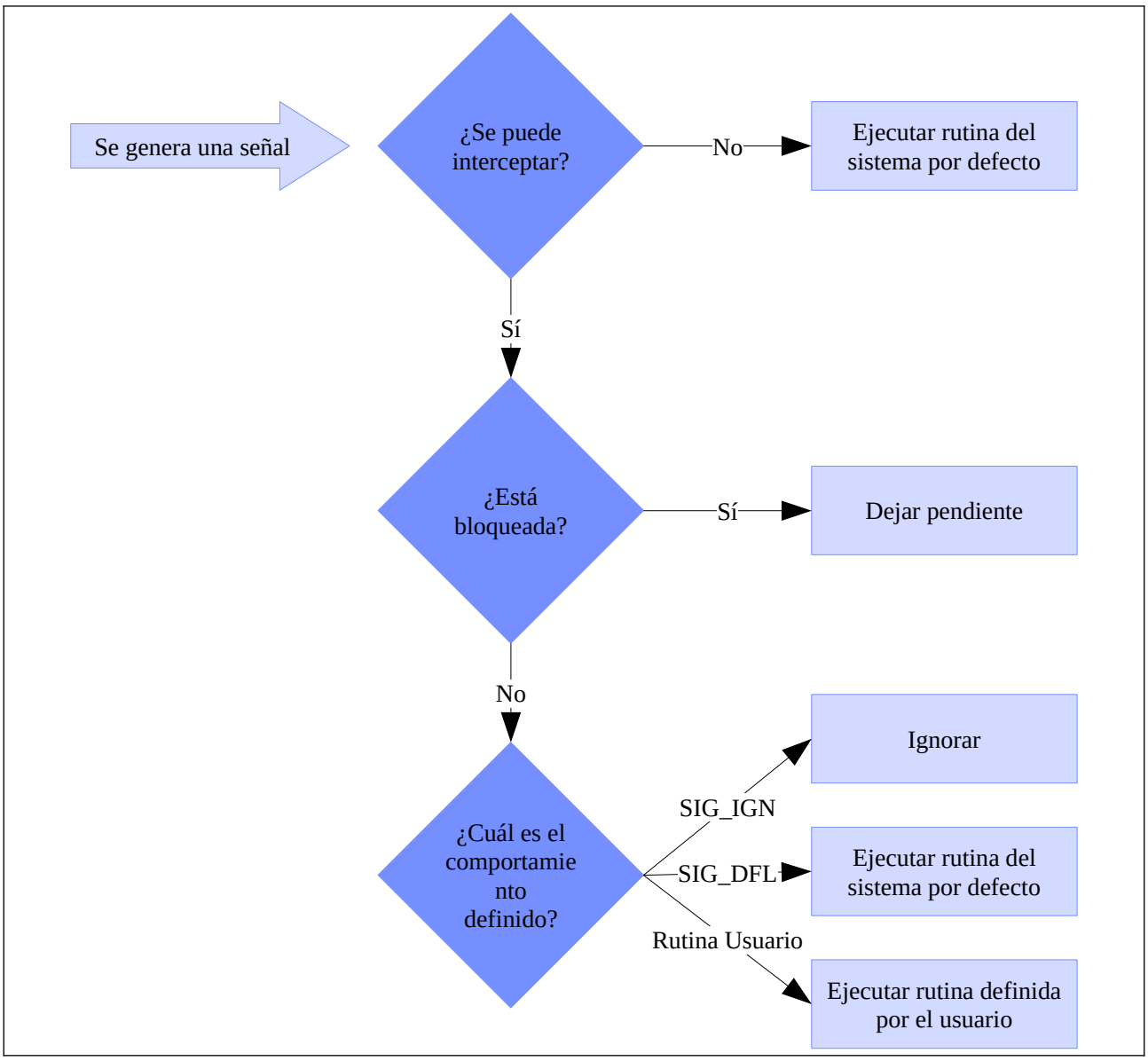

Ilustración 1: Recepción y manejo de señales.

Espera de Señales

```
int sigsuspend(const sigset_t *mask);
```

- Permite bloquear un proceso hasta que se reciba alguna de las señales deseadas.
- Argumentos:
  - `mask` es un puntero al conjunto que contiene la máscara con la que se va a realizar la esperar.
- De forma atómica, sustituye la máscara actual de señales (es decir, las señales bloqueadas) por la que recibe y queda en espera no activa, similar a la de la función `pause`. Si con `pause` podíamos parar un proceso en espera de la primera señal que se reciba, con `sigsuspend` podemos seleccionar la señal por la que se espera.
- Retorno: `-1`.

Servicios de Temporización

En determinadas ocasiones puede interesarnos que el código se ejecute de acuerdo con una temporización determinada. Esto puede venir impuesto por las especificaciones del programa, donde una respuesta antes de tiempo puede ser tan perjudicial como una respuesta con retraso.

`unsigned int alarm(unsigned int seconds);`

- Establece que se enviará una señal `SIGALRM` al cabo de un cierto tiempo.
- Argumentos:
  - `seconds` es un entero que determina la duración de la espera (en segundos).
- Esta llamada activa un temporizador que inicialmente toma el valor `seconds` segundos y que se decrementará en tiempo real. Cuando hayan transcurrido los `seconds` segundos, el proceso recibirá la señal `SIGALRM`. El valor de `seconds` está limitado por la constante `MAX_ALARM`, definida en `<sys/param.h>`. En todas las implementaciones se garantiza que `MAX_ALARM` debe permitir una temporización de al menos 31 días. Si `seconds` toma un valor superior al de `MAX_ALARM`, se trunca al valor de esta constante.

  Para cancelar un temporizador previamente declarado, haremos la llamada `alarm(0)`.

  Después de la llamada a `fork`, una alarma no es heredada por un proceso hijo.

  Es importante recalcar que `alarm` y `sleep` pueden interferir entre ellos, así que en general no hay que mezclarlos.
- Retorno: número de segundos hasta la siguiente alarma (que se sobrescribe), `0` si no había ninguna otra alarma.

Ejercicio 5 **(APRENDIZAJE)** Estudia qué hace el siguiente programa.

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

#define SECS 10

/* manejador_SIGALRM: saca un mensaje por pantalla y termina el proceso. */
void manejador_SIGALRM(int sig) {
    printf("\nEstos son los numeros que me ha dado tiempo a contar\n");
    exit(EXIT_SUCCESS);
}

int main(void) {
    struct sigaction act;
    long int i;

    sigemptyset(&(act.sa_mask));
    act.sa_flags = 0;

    /* Se arma la señal SIGALRM. */
    act.sa_handler = manejador_SIGALRM;
    if (sigaction(SIGALRM, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    if (alarm(SECS))
        fprintf(stderr, "Existe una alarma previa establecida\n");
```

```
    for (i=0;;i++) {
        fprintf(stdout, "%10ld\r", i);
        fflush(stdout);
    }

    fprintf(stdout, "Fin del programa\n");
    exit(EXIT_SUCCESS);
}
```

Herencia entre Procesos Padre-Hijo

Después de la llamada a la función `fork`, el proceso hijo:

- Hereda la máscara de señales bloqueadas.
- Tiene vacía la lista de señales pendientes.
- Hereda las rutinas de manejo.
- No hereda las alarmas.

Tras una llamada a una función `exec`, el proceso lanzado:

- Hereda la máscara de señales bloqueadas.
- Mantiene la lista de señales pendientes.
- No hereda las rutinas de manejo.
- Hereda las alarmas.

Ejercicio 6 **(ENTREGABLE)(1 pto)** Dado el siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>

#define N_ITER 5

int main (void) {
    pid_t pid;
    int counter;

    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        while(1) {
            for (counter = 0; counter < N_ITER; counter++) {
                printf("%d\n", counter);
                sleep(1);
            }
            sleep(3);
        }
    }
    while (wait(NULL) > 0);
}
```

a) Modifica el código del enunciado en un nuevo `ejercicio6a.c` con los siguientes requisitos:

- El hijo, justo después de ser creado, establecerá una alarma para ser recibida dentro de 40 segundos.
- Justo antes de comenzar cada bucle de imprimir números en un proceso hijo, las señales `SIGUSR1`, `SIGUSR2` y `SIGALRM` deben quedar bloqueadas.
- Al finalizar el bucle de impresión de números, y antes de la espera de 3 segundos, se desbloqueará la señal `SIGALRM` y `SIGUSR1`.
- ¿Qué sucede cuando el hijo recibe la señal de alarma?

b) Modifica el código del enunciado en un nuevo `ejercicio6b.c` con los siguientes requisitos:

- El padre enviará la señal `SIGTERM` al proceso hijo cuando hayan pasado 40 segundos de la creación del hijo.
- El proceso hijo, al recibir la señal `SIGTERM`, imprimirá el mensaje `"Soy <PID> y he recibido la señal SIGTERM"` y finalizará su ejecución.
- Cuando el hijo haya finalizado su ejecución, el padre finalizará.

SEMANAS 2 Y 3

Semáforos

Los semáforos son un mecanismo de sincronización provisto por el sistema operativo. Permiten paliar los riesgos del acceso concurrente a recursos compartidos, y básicamente se comportan como variables enteras que tienen asociadas una serie de operaciones atómicas.

Existen dos tipos básicos de semáforos:

- **Semáforo binario**: solo puede tomar dos valores, `0` y `1`. Cuando está a `0` bloquea el acceso del proceso a la sección crítica, mientras que cuando está a `1` permite el paso (poniéndose además a `0` para bloquear el acceso a otros procesos posteriores). Este tipo de semáforos es especialmente útil para garantizar la exclusión mutua a la hora de realizar una tarea crítica. Por ejemplo, para controlar la escritura de variables en memoria compartida, de manera que solo se permita que un proceso esté en la sección crítica mientras que se están modificando los datos.
- **Semáforo N-ario**: puede tomar valores desde `0` hasta `N`. El funcionamiento es similar al de los semáforos binarios. Cuando el semáforo está a `0`, está cerrado y no permite el acceso a la sección crítica. La diferencia está en que puede tomar cualquier otro valor positivo además de `1`. Este tipo de semáforos es muy útil para permitir que un determinado número de procesos trabaje concurrentemente en alguna tarea no crítica. Por ejemplo, varios procesos pueden estar leyendo simultáneamente de la memoria compartida, mientras que ningún otro proceso intente modificar datos.

Los semáforos tienen asociadas dos operaciones fundamentales, caracterizadas por ser atómicas (es decir, se completan o no como una unidad, no permitiéndose que otros procesos las interrumpan a la mitad). Éstas son:

- **Down**: consiste en la petición del semáforo por parte de un proceso que quiere entrar en la sección crítica. Internamente el sistema operativo comprueba el valor del semáforo, de forma que si está a un valor mayor que `0` se le concede el acceso a la sección crítica (por

ejemplo escribir un dato en la memoria compartida) y se decrementa de forma atómica el valor del semáforo. Por otro lado, si el semáforo está a `0`, el proceso queda bloqueado (sin consumir tiempo de CPU, entra en una espera no activa) hasta que el valor del semáforo vuelva a ser mayor que `0` y obtenga el acceso a la sección crítica.

- **Up**: consiste en la liberación del semáforo por parte del proceso que ya ha terminado de trabajar en la sección crítica, incrementando de forma atómica el valor del semáforo en una unidad. Por ejemplo, si el semáforo fuera binario y estuviera a `0` pasaría a valer `1`, y se permitiría el acceso a algún otro proceso que estuviera bloqueado en espera de conseguir acceso a la sección crítica.

Las funciones para gestionar los semáforos POSIX en C para UNIX están incluidas en el fichero de cabecera `<semaphore.h>`. Además, será necesario enlazar el programa con la biblioteca de hilos, es decir se debe añadir a la llamada al compilador `gcc` el parámetro `-pthread`. Se puede obtener un resumen de las funcionalidades de estos semáforos con el comando `man 7 sem_overview`.

Creación y Eliminación de Semáforos

En POSIX existen dos tipos básicos de semáforos, semáforos sin nombre y semáforos con nombre:

- Los **semáforos sin nombre** tienen que estar localizados en una región de memoria accesible a todos los hilos o procesos que los vayan a utilizar.
- Los **semáforos con nombre** pueden ser accedidos por cualquier hilo o proceso que conozca su nombre (una cadena del tipo `/nombre`, que en muchos casos será simplemente una constante definida en el código).

En esta práctica nos centraremos en los semáforos con nombre cuya creación y eliminación se acomete con las funciones siguientes.

```
sem_t *sem_open(const char *name, int oflag);

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int
 value);
```

- Inicializa y abre un semáforo con nombre. Si se especifica que se quiere crear el semáforo hay que usar el prototipo extendido.
- Argumentos:
  - `name` es una cadena que contiene el nombre del semáforo.
  - `oflag` es un entero que determina la operación que se va a realizar. Por ejemplo, para crear semáforos se usa `O_CREAT` (con o sin `O_EXCL`).
  - `mode` es una variable de tipo `mode_t` que especifica los permisos con los que se crea el semáforo.
  - `value` es un entero que indica el valor con el que se va a crear el semáforo.
- Retorno: puntero al semáforo (tipo `sem_t`) si todo es correcto, `SEM_FAILED` en caso de error.

```
int sem_close(sem_t *sem);
```

- Cierra un semáforo con nombre, liberando los recursos que el proceso tuviera asignado para ese semáforo. Sin embargo, el semáforo seguirá accesible a otros procesos (no se borrará).
- Argumentos:
  - `sem` es un puntero al semáforo que se va a cerrar.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

```
int sem_unlink(const char *name);
```

- Elimina un semáforo con nombre (la eliminación efectiva se producirá cuando todos los procesos que tengan el semáforo abierto lo cierren, o cuando terminen su ejecución).
- Argumentos:
  - `name` es una cadena que contiene el nombre del semáforo que se va a eliminar.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

Operaciones con Semáforos

```
int sem_post(sem_t *sem);
```

- Desbloquea un semáforo (“Up”), incrementando su valor.
- Argumentos:
  - `sem` es un puntero al semáforo que se va a incrementar.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

```
int sem_wait(sem_t *sem);
```

- Bloquea un semáforo (“Down”), decrementando su valor.
- Argumentos:
  - `sem` es un puntero al semáforo que se va a decrementar.
- A la hora de decrementar un semáforo también pueden ser interesantes las funciones `sem_trywait` (intento no bloqueante de decrementar el semáforo) y `sem_timedwait` (intento bloqueante de decrementar el semáforo con límite de tiempo).
- Retorno: `0` si todo es correcto, `-1` en caso de error.

```
int sem_getvalue(sem_t *sem, int *sval);
```

- Obtiene el valor de un semáforo.
- Argumentos:
  - `sem` es un puntero al semáforo cuyo valor se va a comprobar.
  - `sval` es un puntero al entero donde se almacenará el valor del semáforo.
- Retorno: `0` si todo es correcto, `-1` en caso de error.

Ejercicio 7 **(APRENDIZAJE)** Estudia el siguiente código. Durante la ejecución en zona protegida del padre, comprueba si localizas el semáforo en el sistema de archivos (normalmente en `/dev/shm`).

```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>

#define SEM "/example_sem"

void imprimir_semaforo(sem_t *sem) {
    int sval;

    if (sem_getvalue(sem, &sval) == -1) {
        perror("sem_getvalue");
        sem_unlink(SEM);
        exit(EXIT_FAILURE);
    }

    printf("Valor del semáforo: %d\n", sval);
    fflush(stdout);
}

int main(void) {
    sem_t *sem = NULL;
    pid_t pid;

    if ((sem = sem_open(SEM, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0)) == SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }

    imprimir_semaforo(sem);
    sem_post(sem);
    imprimir_semaforo(sem);
    sem_post(sem);
    imprimir_semaforo(sem);
    sem_wait(sem);
    imprimir_semaforo(sem);

    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        sem_wait(sem);
        printf("Zona protegida (hijo)\n");
        sleep(5);
        printf("Fin zona protegida (hijo)\n");
        sem_post(sem);

        sem_close(sem);
        exit(EXIT_SUCCESS);
    }
    else {
        sem_wait(sem);
        printf("Zona protegida (padre)\n");
        sleep(5);
        printf("Fin zona protegida (padre)\n");
        sem_post(sem);

        sem_close(sem);
        sem_unlink(SEM);
        wait(NULL);
```

```
        exit(EXIT_SUCCESS);
    }
}
```

Concurrencia

Ejercicio 8 **(ENTREGABLE)(2,75 ptos)** Escribe un programa en C, `ejercicio8.c`, que resuelva el problema de lectores–escritores (dando prioridad a los lectores) usando el siguiente algoritmo:

```
Lectura() {
    Down(sem_lectura);
    lectores++;
    if (lectores == 1)
        Down(sem_escritura);
    Up(sem_lectura);

    Leer();

    Down(sem_lectura);
    lectores--;
    if (lectores == 0)
        Up(sem_escritura);
    Up(sem_lectura);
}

Escritura() {
        Down(sem_escritura);

        Escribir();

        Up(sem_escritura);
}
```

Para el conteo de procesos lectores se puede utilizar un semáforo adicional (que simulará ser una variable entera compartida entre procesos).

En concreto, el programa satisfará los siguientes requisitos:

- El proceso padre creará `N_READ` procesos hijo que serán los lectores, mientras que el padre será el escritor.
- El proceso de lectura se simulará imprimiendo `"R-INI <PID>"`, durmiendo durante un segundo, e imprimiendo `"R-FIN <PID>"`.
- El proceso de escritura se simulará imprimiendo `"W-INI <PID>"`, durmiendo durante un segundo, e imprimiendo `"W-FIN <PID>"`.
- Cada proceso escritor/lector se dedicarán a repetir en un bucle el proceso de escritura/lectura (debidamente protegido) y después dormirá durante `SECS` segundos.
- Cuando el proceso padre reciba la señal `SIGINT` enviará la señal `SIGTERM` a todos los procesos hijos, esperará a que terminen y acabará liberando todos los recursos.

Contesta a las siguientes cuestiones:

a) ¿Qué pasa cuando `SECS=0` y `N_READ=1`? ¿Se producen lecturas? ¿Se producen escrituras? ¿Por qué?

b) ¿Qué pasa cuando `SECS=1` y `N_READ=10`? ¿Se producen lecturas? ¿Se producen escrituras? ¿Por qué?

c) ¿Qué pasa cuando `SECS=0` y `N_READ=10`? ¿Se producen lecturas? ¿Se producen escrituras? ¿Por qué?

d) ¿Qué pasa si los procesos escritores/lectores no duermen nada entre escrituras/lecturas (si se elimina totalmente el `sleep` del bucle)? ¿Se producen lecturas? ¿Se producen escrituras? ¿Por qué?

Ejercicio 9 **(ENTREGABLE)(2,75 ptos)** Escribe un programa en C, `ejercicio9.c`, que satisfaga los siguientes requisitos para simular carreras aleatorias sencillas entre procesos:

- El proceso padre creará `N_PROC` procesos hijo que serán los participantes en la carrera y que tendrán asociado un identificador de `0` a `N_PROC-1`.
- Dentro de un bucle, cada participante en la carrera escribirá en un fichero, protegido para acceso exclusivo, su identificador y dormirá un tiempo aleatorio de máximo una décima de segundo (usando `usleep`).
- El proceso padre, cada segundo, recontará las veces que cada participante en la carrera ha escrito en el fichero, almacenará estas cantidades, las imprimirá por pantalla y reiniciará el fichero (todo ello protegiendo adecuadamente el acceso).
- Si en el recuento se detecta que algún proceso ha llegado a más de 20 escrituras, el proceso padre avisará del final de la carrera y del ganador, mandará la señal `SIGTERM` a todos los procesos hijo para que acaben, esperará a que terminen y acabará liberando todos los recursos.