Práctica 2

FECHA DE ENTREGA: DEL 17 DE MARZO AL 21 DE MARZO
(HORA LÍMITE: ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)

Nota. *La práctica consta de dos tipos de ejercicios diferentes. Los ejercicios cortos son ejercicios de aprendizaje sin puntuación asociada, que no es necesario entregar porque se evaluarán a través del examen correspondiente de evaluación continua. La calificación de la entrega corresponderá únicamente al ejercicio de codificación, incluyendo tanto el código como la documentación.*

Señales

Introducción a las 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 finalizada, sigue la ejecución en el punto en que había bifurcado anteriormente.

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

- Si se pulsa Ctrl+C, el núcleo envía la señal de interrupción `SIGINT`.
- Excepciones de ejecución, como la señal `SIGSEGV` que indica un fallo de segmentación.

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:

1. Ignorar la señal, con lo cual es inmune a la misma.
2. 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.

3. 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 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`.

Tabla 1: Lista de señales UNIX (la lista actualizada se encuentra en el manual de `signal`).

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
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

Señales desde shell

Linux dispone de un comando que permite mandar señales a procesos: el comando `kill`.

Ejercicio 1: Comando `kill` de Linux.

a) Buscar en el manual la forma de acceder a la lista de señales usando el comando `kill`.

b) ¿Qué número tiene la señal `SIGKILL`? ¿Y la señal `SIGSTOP`?

Señales desde C

El programador dispone a su vez de una función con el mismo nombre, `kill`, que permite el envío de señales a procesos, y cuyos detalles pueden consultarse en el manual. Esta función recibe como primer argumento, `pid`, el PID del proceso al que se enviará la señal, con las siguientes peculiaridades:

- Si `pid` > 0, se envía la señal al proceso con PID `pid`.
- Si `pid` $= 0$, se envía la señal al grupo de procesos del proceso que realiza la llamada.
- Si `pid` $= -1$, se envía la señal a todos los procesos para los que se tiene permiso.
- Si `pid` < -1, se envía la señal a todos los procesos cuyo identificador de grupo es $-$`pid`.

El segundo argumento es el número de la señal que se enviará, `sig`. Si `sig` $= 0$ (señal nula) se efectúa una comprobación de errores, pero no se envía ninguna señal.

Ejercicio 2: Envío de señales. Dado el siguiente código en C:

sig_kill.c

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

int main(int argc, char *argv[]) {
  int sig;
  pid_t pid;

  if (argc != 3) {
    fprintf(stderr, "Usage: %s -<signal> <pid>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  sig = atoi(argv[1] + 1);
  pid = (pid_t)atoi(argv[2]);

  /* Complete the code. */

  exit(EXIT_SUCCESS);
}
```

a) Completar el programa en C anterior, de manera que reproduzca de forma limitada la funcionalidad del comando de shell `kill` con un formato similar:

```
$ ./sig_kill -<signal> <pid>
```

El programa debe recibir dos parámetros: el primero, `<signal>` representa el identificador numérico de la señal a enviar; el segundo, `<pid>`, el PID del proceso al que se enviará la señal.

b) Probar el programa enviando la señal `SIGSTOP` de una terminal a otra (cuyo PID se puede averiguar fácilmente con el comando `ps`). ¿Qué sucede si se intenta escribir en la terminal a la que se ha enviado la señal? ¿Y después de enviarle la señal `SIGCONT`?

Tratamiento de señales

Captura de señales

Para poder modificar el comportamiento de un proceso al recibir una señal, el programador puede hacer uso de la función `sigaction`, cuya documentación detallada puede ser consultada en la correspondiente página de manual, para **capturar la señal**. Esta función recibe como primer argumento el número de la señal que se va a capturar, `signum`. El segundo argumento, `act`, es un puntero a una estructura de tipo `sigaction` que define el comportamiento cuando se recibe la señal. En concreto, tiene los siguientes campos:

- `void (*sa_handler)(int)`, una función que define la acción que se tomará al recibir la señal, que 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).
- `void (*sa_sigaction)(int, siginfo_t *, void *)`, la acción que se tomará al recibir la señal pero definida con este prototipo extendido, que se usa cuando se incluye `SA_SIGINFO` como bandera. Este prototipo permite recibir 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 (la señal que se captura se bloquea por defecto, salvo que se indique lo contrario).
- `int sa_flags`, banderas para modificar el comportamiento.

El último argumento, `oldact`, es el puntero a una estructura de tipo `sigaction` donde se almacenará la acción establecida previamente, para poder recuperarla más adelante.

Cuando se recibe la señal capturada, el núcleo es quien se encarga de llamar a la rutina manejadora, pasándole como parámetro el número de la señal. Una vez ejecutada la rutina manejadora y si esta no ha producido la terminación del proceso, la ejecución continúa en el punto en el que bifurcó al recibir la señal. En el caso de rutinas manejadoras definidas por el usuario, ***se sale además de la mayoría de llamadas bloqueantes al sistema***.

Ejercicio 3: Captura de señales. Dado el siguiente código en C:

sig_capture.c

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

/* Handler function for the signal SIGINT. */
void handler(int sig) {
  printf("Signal number %d received\n", sig);
  fflush(stdout);
}

int main(void) {
  struct sigaction act;
```

```
15
16    act.sa_handler = handler;
17    sigemptyset(&(act.sa_mask));
18    act.sa_flags = 0;
19
20    if (sigaction(SIGINT, &act, NULL) < 0) {
21      perror("sigaction");
22      exit(EXIT_FAILURE);
23    }
24
25    while (1) {
26      printf("Waiting Ctrl+C (PID = %d)\n", getpid());
27      sleep(9999);
28    }
29  }
```

a) ¿La llamada a `sigaction` supone que se ejecute la función `handler`?
b) ¿Se bloquea alguna señal durante la ejecución de la función `handler`?
c) ¿Cuándo aparece el `printf` en pantalla?
d) Modificar el programa anterior para que no capture `SIGINT`. ¿Qué sucede cuando se pulsa Ctrl + C? En general, ¿qué ocurre por defecto cuando un programa recibe una señal y no tiene instalado un manejador?
e) A partir del código anterior, escribir un programa que capture todas las señales (desde la `1` hasta la `31`) usando el mismo manejador. ¿Se pueden capturar todas las señales? ¿Por qué?

Manejadores de señales

En el código del Ejercicio 3, la actividad principal del programa es un `sleep`, por lo que parece que una interrupción del mismo no tendría ninguna consecuencia. Sin embargo, en general el programa estará realizando alguna tarea (que puede ser crítica, como el control de una máquina) y la señal le notificará que debe responder de alguna manera (por ejemplo, acabando sus operaciones).

La llamada a la rutina manejadora es asíncrona, lo cual quiere decir que puede darse en cualquier instante de la ejecución del programa, por lo que el código del manejador se va a ejecutar en algún momento sobre el que *a priori* el programador no tiene control. La prioridad de dicho código puede ser baja, pero el sistema operativo lo ejecutará en cuanto reciba la señal. Por eso en muchos casos es recomendable que el manejador simplemente controle que se ha recibido la señal y que el código relacionado con la misma se ejecute dentro del flujo normal del programa (que sí se tiene controlado).

Nota. *Existen una serie de buenos hábitos a la hora de implementar funciones manejadoras, de manera que se basen solo en una serie de operaciones básicas y la llamada a funciones seguras que garantizan que el sistema no quede en un estado inconsistente (se puede ver más información en el manual de* `signal-safety`*).*

Ejercicio 4: Captura de `SIGINT` mejorada. Dado el siguiente código en C:

`sig_capture_improved.c`

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
```

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

static volatile sig_atomic_t got_signal = 0;

/* Handler function for the signal SIGINT. */
void handler(int sig) { got_signal = 1; }

int main(void) {
  struct sigaction act;

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

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

  while (1) {
    printf("Waiting Ctrl+C (PID = %d)\n", getpid());
    if (got_signal) {
      got_signal = 0;
      printf("Signal received.\n");
    }
    sleep(9999);
  }
}
```

a) En esta versión mejorada del programa del Ejercicio 3, ¿en qué líneas se realiza realmente la gestión de la señal?
b) ¿Por qué, en este caso, se permite el uso de variables globales?

Nota. *Con lo estudiado hasta el momento, ya se puede realizar el Ejercicio 10a) y el Ejercicio 10b).*

Protección de zonas críticas

En ocasiones parte del código de un programa puede ser tan crítico que no se quiera permitir que se interrumpa por la llegada de una señal durante su ejecución; para este tipo de situaciones se pueden utilizar las **máscaras de señales**.

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, esta no será enviada al proceso hasta que se desbloquee o la ignore, lo que puede ayudar a garantizar que no se produzcan condiciones de carrera. Si el proceso ignora la señal, esta simplemente se desecha.

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

A continuación se lista el conjunto de funciones que permiten modificar un conjunto de señales (los detalles pueden ser consultados en el manual de `sigsetops`):

- `sigemptyset`: inicializa el conjunto como vacío (sin señales).
- `sigfillset`: inicializa el conjunto como lleno, añadiendo todas las señales.
- `sigaddset`: añade una señal a un conjunto.
- `sigdelset`: elimina una señal de un conjunto.
- `sigismember`: comprueba si una señal pertenece a un conjunto.

Una vez que se tenga definida una máscara de señales, esta puede aplicarse al proceso mediante una llamada a la función `sigprocmask`. Esta función recibe como primer argumento un entero, `how`, indicando cómo se modificará la máscara del proceso:

- `SIG_BLOCK` para que la máscara resultante sea el resultado de la unión de la máscara actual y el conjunto pasado como argumento.
- `SIG_SETMASK` para que la máscara resultante sea la indicada en el conjunto.
- `SIG_UNBLOCK` para que la máscara resultante sea 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).

El segundo argumento, `set`, es un puntero al conjunto que se utilizará para modificar la máscara de señales. El último argumento, `oldset`, es el puntero al conjunto donde se almacenará la máscara de señales anterior, de manera que se pueda restaurar con posterioridad. Es importante recordar que hasta que no se realiza la llamada a la función `sigprocmask` no se produce cambio alguno en la máscara del proceso, simplemente se ha modificado una variable de un programa.

Notas.

- *En el caso de que el proceso sea multihilo se deberá utilizar en lugar de* `sigprocmask` *la función equivalente* `pthread_sigmask` *para manipular la máscara de señales (que es independiente para cada hilo).*
- *En determinadas situaciones puede ser interesante consultar qué señales bloqueadas se encuentran pendientes de entrega al proceso. Esto puede hacerse mediante la función* `sigpending`.

A modo de resumen, la Figura 1 incluye un diagrama de flujo del manejo de señales realizado por el sistema.

Ejercicio 5: Bloqueo de señales. Dado el siguiente código en C:

sig_sigset.c

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

int main(void) {
  sigset_t set, oset;

  /* Mask to block signals SIGUSR1 and SIGUSR2. */
  sigemptyset(&set);
  sigaddset(&set, SIGUSR1);
  sigaddset(&set, SIGUSR2);

  /* Blocking of signals SIGUSR1 and SIGUSR2 in the process. */
  if (sigprocmask(SIG_BLOCK, &set, &oset) < 0) {
    perror("sigprocmask");
    exit(EXIT_FAILURE);
```

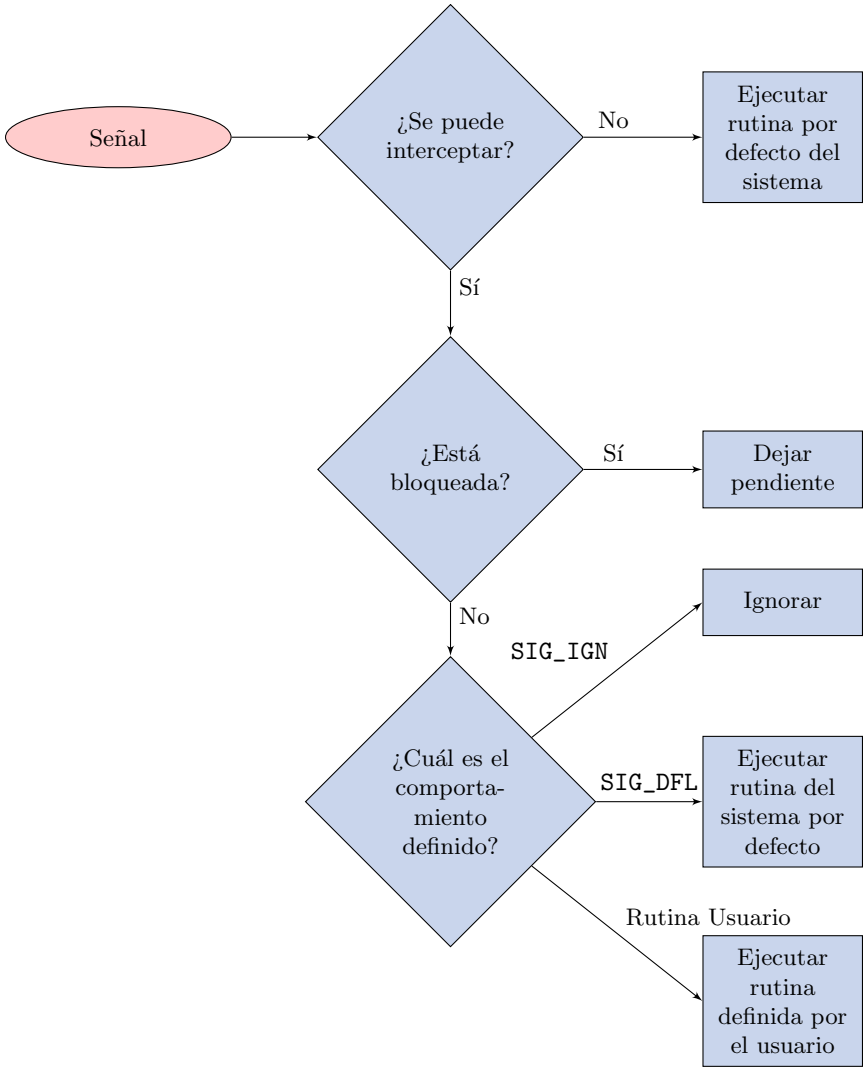

Figura 1: Recepción y manejo de señales.

```
19    }
20
21    printf("Waiting signals (PID = %d)\n", getpid());
22    printf("SIGUSR1 and SIGUSR2 are blocked\n");
23    pause();
24
25    printf("End of program\n");
26    exit(EXIT_SUCCESS);
27 }
```

a) ¿Qué sucede cuando el programa anterior recibe `SIGUSR1` o `SIGUSR2`? ¿Y cuando recibe `SIGINT`?

b) Modificar el programa anterior para que, en lugar de hacer una llamada a `pause`, haga una llamada a `sleep` para suspenderse durante 10 segundos, tras la que debe restaurar la máscara original. Ejecutar el programa, y durante los 10 segundos de espera, enviarle `SIGUSR1`. ¿Qué sucede cuando finaliza la espera? ¿Se imprime el mensaje de despedida? ¿Por qué?

Espera de señales

En el programa del Ejercicio 5 se realizaba una **espera no activa** mediante la función `pause`, que deja al proceso en espera hasta que reciba una señal.

Existe una forma más potente de esperar la llegada de señales: la función `sigsuspend`. Esta función recibe como argumento un puntero a un conjunto de señales, `mask`, que representa la máscara de señales con la que se va a realizar la espera. De forma atómica, la llamada a `sigsuspend` 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`. De esta manera, si con `pause` se paraba un proceso en espera de la primera señal que se recibe, con `sigsuspend` se puede seleccionar la señal por la que se espera. Una vez que termina la espera, se restituye la máscara de señales inicial.

Notas.

- *Se puede encontrar más información respecto a los problemas de usar* `pause` *y las ventajas de usar* `sigsuspend` *en los siguientes enlaces:*
  - *Problems with* `pause`*.*
  - *Using* `sigsuspend`*.*
- *El uso más apropiado de la función* `sigsuspend` *no solo requiere el bloqueo de todas las señales menos aquellas que se quieren esperar, sino también que se bloqueen las señales que se quieren esperar antes de realizar la llamada a* `sigaction` *(usando, para esto, la función* `sigprocmask`*), desbloqueándolas con* `sigsuspend`*. De esta forma, y gracias a que las funciones anteriores son atómicas, no puede darse el caso de que el proceso reciba la señal tras* `sigaction` *(y, por tanto, la trate correctamente y vuelva a su ejecución normal) pero antes de* `sigsuspend`*, comando con el que entraría en un estado de espera del que no podría salir, puesto que la señal ya habría llegado antes de iniciar la espera.*
- *Con lo estudiado hasta el momento, ya se puede realizar el Ejercicio 10c).*

Servicio de temporización

En determinadas ocasiones puede interesar que el código se ejecute de acuerdo con una **temporización** determinada. La función `alarm` provoca que la señal `SIGALRM` se envié al propio proceso al cabo de un cierto tiempo indicado en segundos como parámetro.

Los detalles sobre el comportamiento de `alarm` se pueden consultar en la correspondiente página del manual.

Notas.

- *Es importante recalcar que las funciones* `alarm` *y* `sleep` *pueden interferir entre ellas, así que en general no hay que mezclarlas.*
- *Existen formas más potentes de establecer temporizadores en C aparte de* `alarm`*, como por ejemplo la API proporcionada por POSIX. Esta API permite, entre otras muchas cosas, fijar un número arbitrario de temporizadores, seleccionar el reloj a usar, repetir periódicamente los temporizadores y visualizarlos en el directorio del proceso dentro del sistema de archivos* `/proc`*. Las funciones para estos temporizadores son:*
  - `timer_create`*.*
  - `timer_settime`*.*
  - `timer_gettime`*.*
  - `timer_getoverrun`*.*
  - `timer_delete`*.*

Ejercicio 6: Gestión de la alarma. Dado el siguiente código en C:

sig_alarm.c

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

#define SECS 10

/* The handler shows a message and ends the process. */
void handler_SIGALRM(int sig) {
  printf("\nThese are the numbers I had time to count\n");
  exit(EXIT_SUCCESS);
}

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

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

  /* The handler for SIGALRM is set. */
  act.sa_handler = handler_SIGALRM;
  if (sigaction(SIGALRM, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
  }

  if (alarm(SECS)) {
    fprintf(stderr, "There is a previously established alarm\n");
  }

  fprintf(stdout, "The count starts (PID=%d)\n", getpid());
  for (i = 0;; i++) {
    fprintf(stdout, "%10ld\r", i);
    fflush(stdout);
  }

  fprintf(stdout, "End of program\n");
  exit(EXIT_SUCCESS);
}
```

a) ¿Qué sucede si, mientras se realiza la cuenta, se envía la señal `SIGALRM` al proceso?
b) ¿Qué sucede si se comenta la llamada a `sigaction`?

Nota. *Con lo estudiado hasta el momento, ya se puede realizar el Ejercicio 10d) y el Ejercicio 10e).*

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 de la familia `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.

Semáforos

Introducción a los 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.

Se suelen distinguir dos tipos de semáforos:

- **Semáforo binario**: solo puede tomar dos valores, 0 y 1. Cuando está a 0 bloquea el paso del proceso, 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 se impida que más de un proceso realice la escritura al mismo tiempo.
- **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 paso del proceso. La diferencia está en que puede tomar otros valores positivos 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). Estas son:

- **Down**: consiste en la petición del semáforo por parte de un proceso. Internamente, el sistema operativo comprueba el valor del semáforo, de forma que si está a un valor mayor que 0 se le concede el paso (por ejemplo, para 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.
- **Up**: consiste en la liberación del semáforo por parte del proceso, incrementando de forma atómica el valor del semáforo en una unidad. Por ejemplo, si el semáforo estuviera a 0 pasaría a valer 1, y se permitiría el acceso a algún otro proceso que estuviera bloqueado

en espera de conseguir acceso.

La Figura 2 muestra cómo usar un semáforo para garantizar que un proceso no ejecute un fragmento de código (el bloque «`Code 2`») hasta que otro proceso haya terminado de ejecutar otro fragmento (el bloque «`Code 1`»). Por otro lado, la Figura 3 ilustra el uso de un semáforo para garantizar la exclusión mutua, de forma que solo un proceso pueda estar en una cierta región de código (su bloque «`Critic`») al mismo tiempo. En este ejemplo, el proceso 1 es el primero en alcanzar la región crítica.

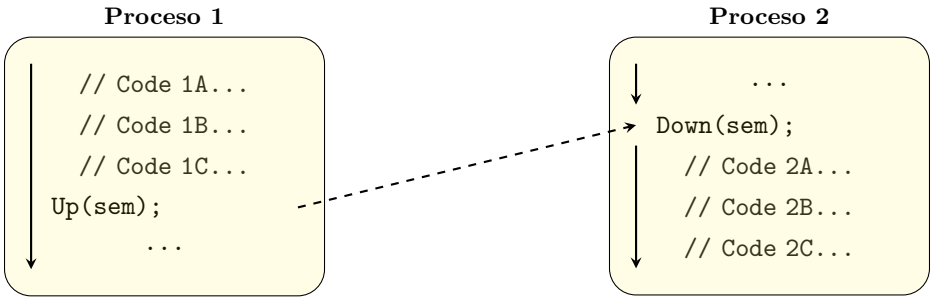

Figura 2: Funcionamiento de un semáforo (***inicializado a*** 0) que garantiza que un cierto fragmento de código en un proceso se ejecuta antes que otro fragmento en otro proceso.

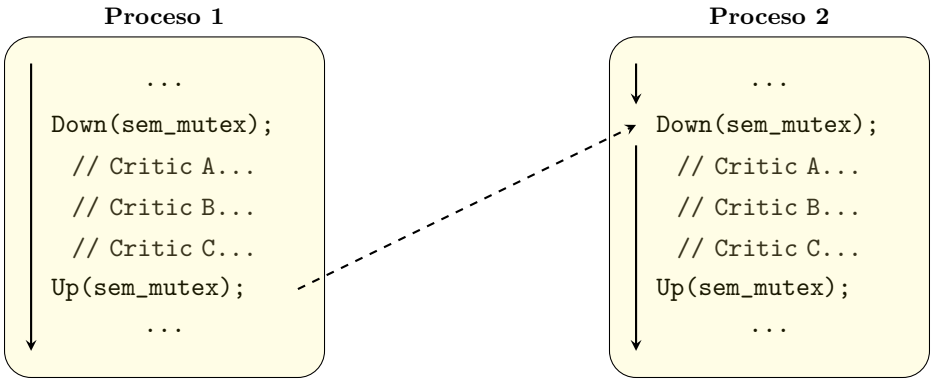

Figura 3: Funcionamiento de un semáforo (***inicializado a*** 1) para garantizar la exclusión mutua.

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 `-lpthread` (o `-pthread`). Se puede obtener un resumen de las funcionalidades de estos semáforos en el manual de `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).

Para crear y/o abrir un semáforo con nombre se utiliza la función `sem_open` (que comparte la semántica con la función `open` para ficheros). El primer argumento de esta función es el nombre del semáforo que se va a abrir (o crear). El segundo argumento es un entero que determina la operación que se va a realizar (por ejemplo, para crear semáforos se usa `O_CREAT`, con o sin la opción `O_EXCL`). En el caso de que se vaya a crear el semáforo, es necesario usar el prototipo extendido con dos argumentos adicionales, uno de tipo `mode_t` para indicar los permisos del nuevo semáforo, y otro de tipo `int` con el valor inicial del semáforo recién creado. El retorno de esta función es un puntero a `sem_t`, con el que se puede manipular posteriormente el semáforo.

La función `sem_close` 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).

Por último, la función `sem_unlink` elimina un semáforo con nombre (de forma similar a lo que hace `unlink` con los ficheros). Es importante entender su uso. Normalmente cuando un proceso muere se liberan los recursos asociados al mismo. Sin embargo, elementos como los semáforos pertenecen al sistema operativo y pueden compartirse por varios procesos. Al llamar a `sem_open` se solicita al sistema operativo que cree un nuevo semáforo, o bien que abra un semáforo ya existente con cierto nombre, y el número de procesos asociado al semáforo se incrementa en uno. Al hacer `sem_close`, se reduce el número de procesos asociados al semáforo en uno. Para que el sistema operativo libere los recursos asociados al semáforo se llama a `sem_unlink`, para que al llegar el contador de procesos asociados al semáforo a cero, sus recursos sean liberados por el sistema operativo. Hay que tener en cuenta que al llamar a `sem_unlink` no tiene por qué borrarse el semáforo inmediatamente, pero su nombre sí, por lo que una nueva llamada a `sem_open` con el nombre del semáforo creará un semáforo nuevo.

Nota. *Como se indica en el manual de* `sem_overview`*, en Linux los semáforos con nombre se crean en un sistema de ficheros virtual, montado normalmente en el directorio* `/dev/shm`*.*

Operaciones con semáforos

Para realizar la operación Up con los semáforos POSIX, se usará la función `sem_post`, mientras que para realizar la operación Down se usará la función `sem_wait`. Se pueden consultar los detalles en las páginas correspondientes del manual, donde además se puede ver información sobre las funciones de apoyo `sem_trywait` (no bloqueante) y `sem_timedwait` (bloqueo limitado en el tiempo). Para conocer el estado de un semáforo se dispone de la función `sem_getvalue`.

Ejercicio 7: Creación y eliminación de semáforos. Dado el siguiente código en C:

sem_create.c

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

#define SEM_NAME "/example_sem"

void sem_print(sem_t *sem) {
  int sval;
  if (sem_getvalue(sem, &sval) == -1) {
```

```
    perror("sem_getvalue");
    sem_unlink(SEM_NAME);
    exit(EXIT_FAILURE);
  }
  printf("Semaphore value: %d\n", sval);
  fflush(stdout);
}

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

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

  sem_print(sem);
  sem_post(sem);
  sem_print(sem);
  sem_post(sem);
  sem_print(sem);
  sem_wait(sem);
  sem_print(sem);

  pid = fork();
  if (pid < 0) {
    perror("fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    sem_wait(sem);
    printf("Critical region (child)\n");
    sleep(5);
    printf("End of critical region (child)\n");
    sem_post(sem);
    sem_close(sem);
    exit(EXIT_SUCCESS);
  } else {
    sem_wait(sem);
    printf("Critical region (parent)\n");
    sleep(5);
    printf("End of critical region (parent)\n");
    sem_post(sem);
    sem_close(sem);
    sem_unlink(SEM_NAME);

    wait(NULL);
    exit(EXIT_SUCCESS);
  }
}
```

¿Podría modificarse el sitio de llamada a `sem_unlink`? En caso afirmativo, ¿cuál sería la primera posición en la que se sería correcto llamar a `sem_unlink`?

Ejercicio 8: Semáforos y señales. Dado el siguiente código en C:

sem_signal.c

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

#define SEM_NAME "/example_sem"

void handler(int sig) { return; }

int main(void) {
  sem_t *sem = NULL;
  struct sigaction act;

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

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

  /* The handler for SIGINT is set. */
  act.sa_handler = handler;
  if (sigaction(SIGINT, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
  }

  printf("Starting wait (PID=%d)\n", getpid());
  sem_wait(sem);
  printf("Finishing wait\n");
  sem_unlink(SEM_NAME);
}
```

a) ¿Qué sucede cuando se envía la señal `SIGINT`? ¿La llamada a `sem_wait` se ejecuta con éxito? ¿Por qué?

b) ¿Qué sucede si, en lugar de usar un manejador vacío, se ignora la señal con `SIG_IGN`?

c) ¿Qué cambios habría que hacer en el programa anterior para garantizar que no termine salvo que se consiga hacer el Down del semáforo, lleguen o no señales capturadas?

Concurrencia

Procesos alternos

Ejercicio 9: Procesos alternos. Dado el siguiente código en C:

conc_alternate.c

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

#define SEM_NAME_A "/example_sem_1"
#define SEM_NAME_B "/example_sem_2"

int main(void) {
  sem_t *sem1 = NULL;
  sem_t *sem2 = NULL;
  pid_t pid;

  if ((sem1 = sem_open(SEM_NAME_A, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0)
    ) ==
      SEM_FAILED) {
    perror("sem_open");
    exit(EXIT_FAILURE);
  }
  if ((sem2 = sem_open(SEM_NAME_B, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0)
    ) ==
      SEM_FAILED) {
    perror("sem_open");
    exit(EXIT_FAILURE);
  }

  pid = fork();
  if (pid < 0) {
    perror("fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    /* Insert code A. */
    printf("1\n");
    /* Insert code B. */
    printf("3\n");
    /* Insert code C. */

    sem_close(sem1);
    sem_close(sem2);
  } else {
    /* Insert code D. */
    printf("2\n");
    /* Insert code E. */
    printf("4\n");
    /* Insert code F. */

    sem_close(sem1);
```

```
51      sem_close(sem2);
52      sem_unlink(SEM_NAME_A);
53      sem_unlink(SEM_NAME_B);
54      wait(NULL);
55      exit(EXIT_SUCCESS);
56    }
57  }
```

Rellenar el código correspondiente a los huecos `A`, `B`, `C`, `D`, `E` y `F` (alguno de ellos puede estar vacío) con llamadas a `sem_wait` y `sem_post` de manera que la salida del programa sea:

```
1
2
3
4
```

Nota. *Con lo estudiado hasta el momento, ya se puede realizar el Ejercicio 10 completo.*

Ejercicio de codificación

Continuando con la implementación de la red de mineros basados en **Blockchain**, un aspecto crucial en estos sistemas es la validación de las soluciones obtenidas. Una manera de abordar esta tarea es exigiendo que cada solución encontrada sea validada por una mayoría de los mineros que componen la red antes de ser aceptada.

Como se describe a continuación, se implementará un sistema de votación multiproceso, donde en cada iteración un proceso se presente como «candidato», y el resto determinen (por el momento, de forma aleatoria) si aceptan o no su propuesta. La sincronización requerida se realizará mediante señales y semáforos.

10,00 ptos. **Ejercicio 10: Sistema de votación multiproceso.** Escribir un programa en C que implemente un sistema de votación multiproceso. El ejercicio se divide en distintas partes diferenciadas, de manera que se puedan ir implementado y probando secuencialmente.

2,00 ptos. a) Arranque del sistema y proceso Principal:

- El sistema se debe ejecutar con dos parámetros:

  ```
  ./voting <N_PROCS> <N_SECS>
  ```

  donde `voting` es el nombre del ejecutable, `<N_PROCS>` es el número de procesos que participarán en la votación y `<N_SECS>` es el número máximo de segundos que estará activo el sistema.
- El proceso Principal creará tantos procesos Votante como se haya especificado, y almacenará la información del sistema (al menos los PID de los procesos Votante) en un fichero.
- Cuando el sistema esté listo, el proceso Principal enviará la señal `SIGUSR1` a todos los procesos Votante.
- Cuando Principal reciba la señal de interrupción `SIGINT`, enviará a los procesos Votante la señal `SIGTERM` para que terminen liberando sus recursos. Cuando terminen, liberará los recursos del sistema y terminará su ejecución mostrando el mensaje `"Finishing by signal"`.

2,00 ptos.

b) Procesos Votante:
- Al arrancar, los procesos Votante quedarán en espera de que el proceso Principal les avise de que el sistema está listo.
- Cuando reciban la señal `SIGUSR1` leerán la información del sistema para poder enviar señales a todos los procesos implicados, y comenzarán a iterar:
  - Los procesos competirán por convertirse en el proceso Candidato (en una suerte de condición de carrera), de forma que el primero que lo solicite a través de señales será el proceso Candidato.
  - El resto de procesos Votante quedarán en espera de que arranque la votación, para elegir si aceptan o no al candidato. Esta votación se realizará a través de un fichero compartido por todos los procesos.
  - Cuando la votación esté lista, el proceso Candidato enviará la señal `SIGUSR2` a los procesos Votante para que registren su voto. Tras esto, esperará a que finalice la votación (es decir, a que se hayan registrado todos los votos en el fichero) alternando la comprobación del fichero con esperas no activas de 1 ms.
  - Cuando los procesos Votante reciban la señal de votar, generarán un voto aleatorio (sí o no) y lo registrarán en el fichero. Una vez votado, entrarán en una espera no activa hasta la siguiente ronda.
  - Una vez finalice la votación, el proceso Candidato mostrará los resultados con una cadena con el formato `"Candidate %d => [ Y Y N Y N ] => Accepted"`/`"Candidate %d => [ Y N N N Y ] => Rejected"`, donde el número indica el PID del proceso candidato, las letras `'Y'`/`'N'` el sentido del voto de cada proceso, y la candidatura se acepta en el caso de que el número de votos positivos sea mayor que el de negativos. Tras esto, realizará una espera no activa de 250 ms y arrancará una nueva ronda enviando la señal `SIGUSR1`.
- Los procesos Votante terminarán, liberando todos los recursos necesarios, cuando reciban la señal `SIGTERM`.

2,00 ptos.

c) Protección de zonas críticas:
- Es necesario garantizar que el sistema es robusto, de manera que no se pierda ninguna señal.
- Para ello, se deberán bloquear las señales (puede que también durante la ejecución del manejador) y realizar las esperas no activas con `sigsuspend`.

1,00 ptos.

d) Temporización:
- En el caso de que transcurra el número máximo de segundos especificados como argumento del programa, el proceso Principal procederá a terminar el sistema enviando la señal `SIGTERM` a los procesos Votante, y terminará su ejecución con el mensaje `"Finishing by alarm"`.

1,00 ptos.

e) Análisis de la ejecución:
- ¿Se produce algún problema de concurrencia en el sistema descrito?
- ¿Es sencillo, o siquiera factible, organizar un sistema de este tipo usando únicamente señales?

2,00 ptos.

f) Sincronización con semáforos:
- Añadir los mecanismos necesarios, usando semáforos, para que no se produzcan problemas de concurrencia, de forma que solo haya un proceso Candidato, y que no se pierda información en los accesos al fichero.

El esquema global del programa se resume en la Figura 4.

Se deberán tener en cuenta los siguientes aspectos: (a) entrega de la documentación en formato `.pdf` requerida en la normativa de prácticas, (b) control de errores, (c) liberación de los semáforos, (d) espera del proceso padre a sus procesos hijo, y (e) garantizar que no se producen bloqueos por pérdida de señales.
Algunos ejemplos de ejecución del sistema final son los siguientes:

```
$ ./voting 20 1
Candidate 19427 => [ N Y Y N N Y Y N Y Y Y Y N Y Y Y Y Y N Y ] => Accepted
Candidate 19426 => [ Y Y Y Y Y N Y N N Y Y N Y Y N N N N Y Y ] => Accepted
Candidate 19427 => [ Y Y Y N Y Y N Y N Y N Y N N Y N N N N N ] => Rejected
Candidate 19428 => [ N N Y N N N Y Y Y N Y Y N N Y Y N Y N N ] => Rejected
Finishing by alarm
```

```
$ ./voting 10 5
Candidate 19585 => [ N Y N Y Y Y Y Y Y N ] => Accepted
Candidate 19584 => [ Y Y N N N Y N Y Y N ] => Rejected
Candidate 19586 => [ N Y Y Y N Y Y N Y N ] => Accepted
Candidate 19587 => [ Y Y N Y Y Y Y N Y N ] => Accepted
Candidate 19584 => [ Y Y N Y N N Y Y Y Y ] => Accepted
Candidate 19584 => [ Y N N Y Y Y Y N Y N ] => Accepted
Candidate 19584 => [ Y N N Y N Y Y N Y N ] => Rejected
Candidate 19585 => [ N Y N N Y Y N Y N Y ] => Rejected
Candidate 19585 => [ N Y Y N N N N N Y N ] => Rejected
^CFinishing by signal
```

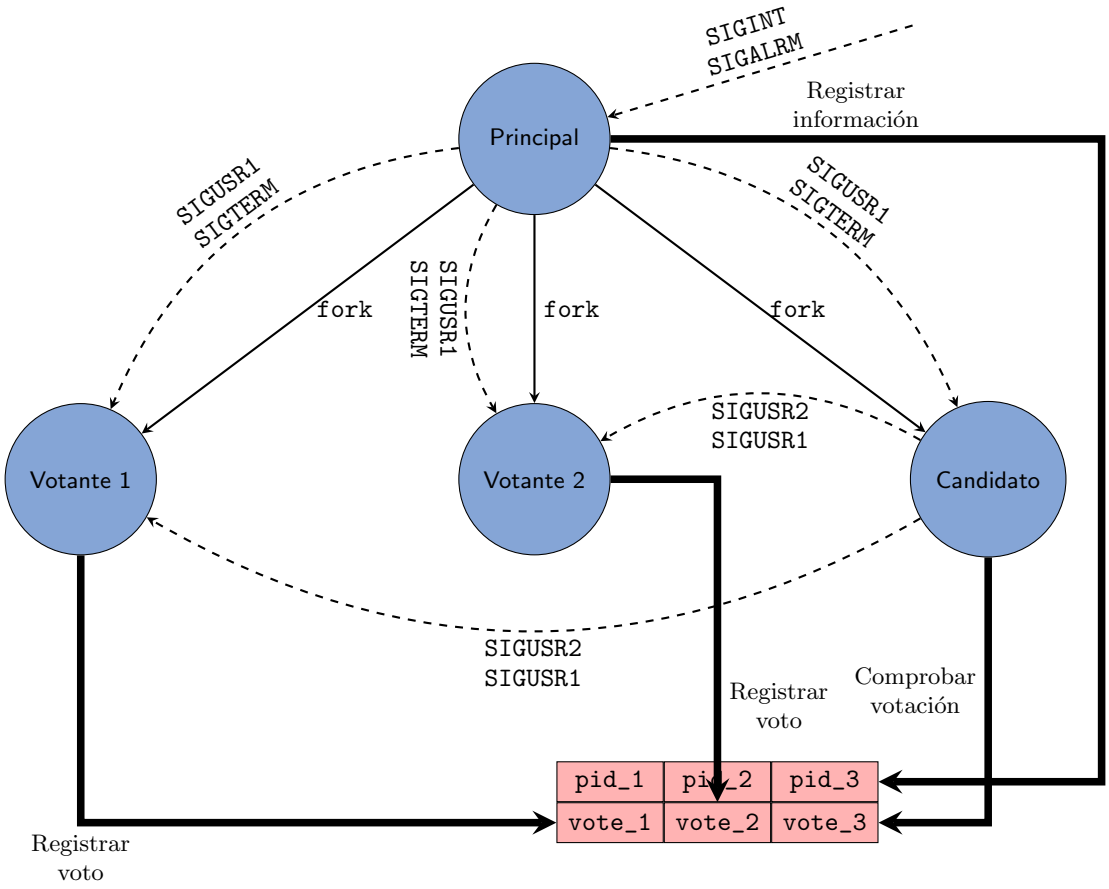

Figura 4: Esquema global del funcionamiento del sistema de votación, en una ronda en la que el proceso Votante 3 actúa de proceso Candidato.