Práctica 2

FECHA DE ENTREGA: DEL 16 DE MARZO AL 20 DE MARZO
(HORA LÍMITE: UNA HORA ANTES DEL COMIENZO DE LA CLASE DE PRÁCTICAS)

SEMANA 1: SEÑALES.
- Introducción a las Señales
- Envío de Señales
- Tratamiento de Señales
- Servicio de Temporización
- Herencia entre Procesos Padre-Hijo

SEMANAS 2–3: SEMÁFOROS Y CONCURRENCIA.
- Introducción a los Semáforos
- Creación y Eliminación de Semáforos
- Operaciones con Semáforos
- Concurrencia

SEMANA 1: 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 finalizado, 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.

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 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`.

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, 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

Señales desde Shell

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

0,20 ptos.

Ejercicio 1: Comando `kill` de Linux.

0,10 ptos. a) Buscar en el manual la forma de acceder a la lista de señales usando el comando `kill`. Copiar en la memoria el comando utilizado.

0,10 ptos. 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 consultarle 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.

0,50 ptos.

Ejercicio 2: Envío de Señales.

0,40 ptos. a) Escribir un programa en C ("`ejercicio_kill.c`") que reproduzca de forma limitada la funcionalidad del comando de shell `kill` con un formato similar: `ejercicio_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.

0,10 ptos. 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 reciba la señal. En concreto, tiene los siguientes campos:

- `void (*sa_handler)(int)` es una función que define 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).
- `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 ésta 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***.

0,30 ptos.

Ejercicio 3: Captura de `SIGINT`. Dado el siguiente código en C, correspondiente al fichero "`ejercicio_captura.c`":

```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);
    }
}
```

0,10 ptos. a) ¿La llamada a `sigaction` supone que se ejecute la función `manejador`?

0,10 ptos. b) ¿Se bloquea alguna señal durante la ejecución de la función `manejador`?

0,10 ptos. c) ¿Cuándo aparece el `printf` en pantalla?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

0,30 ptos.

Ejercicio 4: Captura de Señales.

0,10 ptos. a) ¿Qué ocurre por defecto cuando un programa recibe una señal y no tiene instalado un manejador?

0,20 ptos. b) Escribir un programa que capture todas las señales (desde la `1` hasta la `31`) usando el manejador del Ejercicio 3. ¿Se pueden capturar todas las señales? ¿Por qué?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

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`*).*

0,20 ptos.

Ejercicio 5: Captura de `SIGINT` Mejorada. Dado el siguiente código en C, correspondiente al fichero "`ejercicio_captura_mejorado.c`":

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

static volatile sig_atomic_t got_signal = 0;

/* manejador: rutina de tratamiento de la señal SIGINT. */
void manejador(int sig) {
    got_signal = 1;
}

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());
        if(got_signal) {
          got_signal = 0;
          printf("Señal recibida.\n");
        }
        sleep(9999);
    }
```

```
34 }
```

0,10 ptos. a) En esta versión mejorada del programa del Ejercico 3, ¿en qué líneas se realiza realmente la gestión de la señal?

0,10 ptos. b) ¿Por qué, en este caso, se permite el uso de variables globales?

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, ésta 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, ésta 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`, para inicializar el conjunto como vacío (sin señales).
- `sigfillset`, para inicializar el conjunto como lleno, añadiendo todas las señales.
- `sigaddset`, para añadir una señal a un conjunto.
- `sigdelset`, para eliminar una señal de un conjunto.
- `sigismember`, para comprobar si una señal pertenece a un conjunto.

Una vez que se tenga definida una máscara de señales, ésta 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.

Nota. *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).*

Nota. *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.

0,30 ptos. **Ejercicio 6: Bloqueo de Señales.** Dado el siguiente código en C, correspondiente al fichero "`ejercicio_sigset.c`":

```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 SIGUSR1 y SIGUSR2. */
    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    sigaddset(&set, SIGUSR2);

    /* Bloqueo de las señales SIGUSR1 y SIGUSR2 en el proceso. */
    if (sigprocmask(SIG_BLOCK, &set, &oset) < 0) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    printf("En espera de señales (PID = %d)\n", getpid());
    printf("SIGUSR1 y SIGUSR2 están bloqueadas\n");
    pause();

    printf("Fin del programa\n");
    exit(EXIT_SUCCESS);
}
```

0,10 ptos. a) ¿Qué sucede cuando el programa anterior recibe `SIGUSR1` o `SIGUSR2`? ¿Y cuando recibe `SIGINT`?

0,20 ptos. 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?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Espera de Señales

En el programa del Ejercicio 6 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

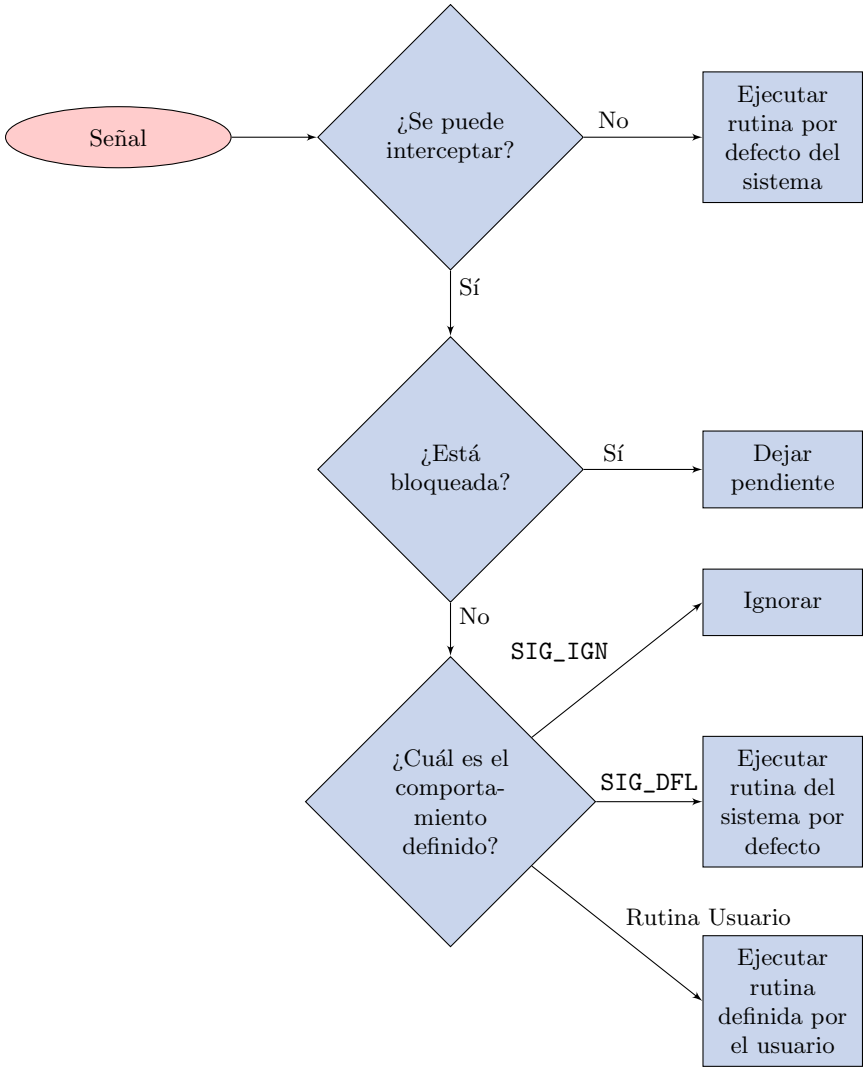

Figura 1: Recepción y manejo de señales.

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.

Nota. *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`.

Nota. *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.

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.

Nota. *Es importante recalcar que* `alarm` *y* `sleep` *pueden interferir entre ellos, así que en general no hay que mezclarlos.*

Nota. *Existen formas más potentes de establecer temporizadores en C aparte de* `alarm`*, como por ejemplo la API proporcionada por POSIX:*

- `timer_create`.
- `timer_settime`.
- `timer_gettime`.
- `timer_getoverrun`.
- `timer_delete`.

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`.

0,20 ptos.

Ejercicio 7: Gestión de la Alarma. Dado el siguiente código en C, correspondiente al fichero "`ejercicio_alarm.c`":

```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);
    }
```

```
27
28      if (alarm(SECS)) {
29          fprintf(stderr, "Existe una alarma previa establecida\n");
30      }
31
32      fprintf(stdout, "Comienza la cuenta (PID=%d)\n", getpid());
33      for (i=0;;i++) {
34          fprintf(stdout, "%10ld\r", i);
35          fflush(stdout);
36      }
37
38      fprintf(stdout, "Fin del programa\n");
39      exit(EXIT_SUCCESS);
40  }
```

0,10 ptos. a) ¿Qué sucede si, mientras se realiza la cuenta, se envía la señal `SIGALRM` al proceso?

0,10 ptos. b) ¿Qué sucede si se comenta la llamada a `sigaction`?

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

2,50 ptos. **Ejercicio 8: Señales, Protección y Temporarización.**

2,30 ptos. a) Escribir un programa en C ("`ejercicio_prottemp.c`") que satisfaga los siguientes requisitos:

- El programa tendrá dos parámetros de entrada, `N` y `T`. El programa debe crear `N` hijos que deberán ejercutarse de forma concurrente.
- Una vez creados los hijos, el proceso padre quedará suspendido durante un tiempo `T`. Debe utilizarse la función `alarm` para controlar ese tiempo. ***No se utilizará*** `pause`.
- Una vez transcurridos `T` segundos, el proceso padre enviará la señal `SIGTERM` a todos los hijos, imprimirá "`Finalizado Padre`", y finalizará sin dejar huérfanos.
- Cada proceso hijo realizará un trabajo que consiste en sumar todos los números enteros entre 1 y su PID dividido por 10 (PID/10) e imprimir una línea con su PID y el resultado del trabajo.
- Al acabar el trabajo enviarán al proceso padre la señal `SIGUSR2` y quedarán en suspenso hasta recibir la señal `SIGTERM` del padre. Tras recibir la señal, imprimirá "`Finalizado`" junto con su PID.

Además del programa, es necesario comentar en la memoria las decisiones de diseño que se han tomado para implementar la suspensión del proceso padre y garantizar el control sobre la recepción de señales.

0,20 ptos. b) Establecer un contador de señales recibidas en el manejador de la señal `SIGUSR2`. ¿Cuántas se reciben en comparación con `N`? ¿Hay alguna garantía de que se reciba ese número de señales? ¿Por qué?

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.

SEMANAS 2–3: SEMÁFOROS Y CONCURRENCIA.

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 realicen 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 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. 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 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.

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 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 `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. Al hacer `sem_open`, 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`.

0,20 ptos.

Ejercicio 9: Creación y Eliminación de Semáforos. Dado el siguiente código en C, correspondiente al fichero "`ejercicio_sem.c`":

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

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

#define SEM_NAME "/example_sem"

void imprimir_semaforo(sem_t *sem) {
    int sval;
    if (sem_getvalue(sem, &sval) == -1) {
        perror("sem_getvalue");
        sem_unlink(SEM_NAME);
        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_NAME, 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_NAME);
```

```
62
63          wait(NULL);
64          exit(EXIT_SUCCESS);
65      }
66  }
```

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

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

0,30 ptos.

Ejercicio 10: Semáforos y Señales. Dado el siguiente código en C, correspondiente al fichero "`ejercicio_sem_signal.c`":

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

#define SEM_NAME "/example_sem"

void manejador(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;

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

    printf("Entrando en espera (PID=%d)\n", getpid());
    sem_wait(sem);
    printf("Fin de la espera\n");
    sem_unlink(SEM_NAME);
}
```

0,10 ptos. a) ¿Qué sucede cuando se envía la señal `SIGINT`? ¿La llamada a `sem_wait` se ejecuta con éxito? ¿Por qué?

0,10 ptos. b) ¿Qué sucede si, en lugar de usar un manejador vacío, se ignora la señal con `SIG_IGN`?

0,10 ptos.

c) Describir los cambios que 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.

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Concurrencia

Procesos Alternos

0,30 ptos.

Ejercicio 11: Procesos Alternos. Dado el siguiente código en C, correspondiente al fichero "`ejercicio_alternar.c`":

```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_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) {
        /* Rellenar Código A */
        printf("1\n");
        /* Rellenar Código B */
        printf("3\n");
        /* Rellenar Código C */

        sem_close(sem1);
        sem_close(sem2);
```

```
    }
    else {
        /* Rellenar Código D */
        printf("2\n");
        /* Rellenar Código E */
        printf("4\n");
        /* Rellenar Código F */

        sem_close(sem1);
        sem_close(sem2);
        sem_unlink(SEM_NAME_A);
        sem_unlink(SEM_NAME_B);
        wait(NULL);
        exit(EXIT_SUCCESS);
    }
}
```

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
```

Describir de forma razonada las llamadas a los semáforos utilizadas.

Nota. *No es necesario entregar código, tan solo contestar a las preguntas en la memoria.*

Concurrencia y Comunicación entre Procesos

2,30 ptos.

Ejercicio 12: Concurrencia. Mejorar el programa resultante del Ejercicio 8 para crear un nuevo programa (“`ejercicio_prottemp_mejorado.c`”) de manera que el proceso padre lea de un fichero el resultado final del trabajo ejecutado por los hijos y pueda imprimirlo en pantalla. Se deberán cumplir los siguientes requisitos:

- Los procesos hijo leeran de un mismo fichero llamado `data.txt` dos líneas, cada una con un número. La primera línea contiene el número de procesos que han escrito su resultado. La segunda línea contiene la suma del trabajo realizado por los procesos anteriores.
- El proceso padre creará o abrirá el fichero `data.txt` e inicializará las dos líneas a 0.
- Los procesos hijos, tras leer los datos, modificarán el fichero `datos.txt` sumando 1 al dato de la primera línea, y el resultado de su trabajo al dato de la segunda línea.
- Las acciones anteriores las realizará cada proceso hijo al terminar su trabajo, y antes de mandar la señal `SIGUSR2` al padre.
- La señal `SIGUSR2` será utilizada por el padre como indicación de que debe verificar el estado del fichero `data.txt`. La verificación consiste en comprobar si el trabajo de todos los hijos ha sido incluido en el fichero. Cuando la verificación sea positiva el padre enviará a los hijos la señal `SIGTERM` sin esperar a que finalice el tiempo `T`.
- Si se ha cumplido el tiempo `T`, el padre imprimirá “`Falta trabajo`”. En caso contrario, imprimirá una línea “`Han acabado todos, resultado:`” junto con el resultado.
- Se debe tener especial cuidado con posibles fallos debidos a la concurrencia de lectura

y escritura.

+1,00 ptos.

Ejercicio 13: Comunicación entre Procesos (Opcional). En el Ejercicio 12 la comunicación entre procesos se ha implementado a través de escritura y lectura en un fichero. El proceso padre sabe que todos los hijos han acabado cuando verifica que el número de procesos que han escrito en el fichero coincide con el número de hijos que ha lanzado. El objetivo de este ejercicio es modificar el programa para crear uno nuevo, `ejercicio_prottemp_mejorado_op.c`, que permita saber al padre que todos los hijos han realizado el trabajo sin necesidad de leer esa información en el fichero. Es decir, cuando el padre abra el fichero todos los hijos habrán acabado el trabajo y habrán terminado de escribir en el fichero.
Los requisitos a cumplir son:

- El proceso padre solo accederá al fichero `data.txt` cuando esté seguro de que todos los hijos han acabado el trabajo.
- El proceso padre enviará la señal `SIGTERM` de finalización a los hijos después de haber leído en el fichero el resultado final.

Algunas opciones que puede valorar el alumno:

- Uso de semáforos N-arios.
- Los procesos hijos podrán implementar un modelo de comunicación adicional para mandarse señales entre ellos.

Lectores–Escritores

Uno de los posibles algoritmos para resolver el problema de lectores–escritores (en este caso, dando prioridad a los lectores), es el siguiente:

```
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);
}
```

2,40 ptos.

Ejercicio 14: Problema de Lectores–Escritores.

2,00 ptos. a) Escribir un programa en C ("`ejercicio_lect_escr.c`") que implemente el algoritmo de lectores–escritores. 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.

0,10 ptos. b) ¿Qué pasa cuando `SECS=0` y `N_READ=1`? ¿Se producen lecturas? ¿Se producen escrituras? ¿Por qué?

0,10 ptos. c) ¿Qué pasa cuando `SECS=1` y `N_READ=10`? ¿Se producen lecturas? ¿Se producen escrituras? ¿Por qué?

0,10 ptos. d) ¿Qué pasa cuando `SECS=0` y `N_READ=10`? ¿Se producen lecturas? ¿Se producen escrituras? ¿Por qué?

0,10 ptos. e) ¿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?