MEMORIA PRÁCTICA 2

SISTEMAS OPERATIVOS

Rubén García de la Fuente
Elena Cano Castillejo
Pareja 04
Grupo 2202

<u>Ejercicio 1:</u> Comando kill de Linux. *(0,20 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.)*

Ejecutamos `man kill` y leyendo las instrucciones vemos que para ver la lista tenemos que ejecutar `kill -l`.

b) **¿Qué número tiene la señal SIGKILL? ¿Y la señal SIGSTOP?** *(0,10 ptos.)*

Podemos observar en la lista de señales que SIGKILL tiene el valor 9 y SIGSTOP el valor 19.

<u>Ejercicio 2:</u> Envío de Señales. *(0,50 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.** *(0,40 ptos.)*

Escribimos un programa con los requisitos especificados cuyo código es el siguiente:

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

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

    if (argc < 3) {
        fprintf(stdout, "Se esperaban 2 argumentos de entrada\n");
        return 1;
    }

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

    if (kill(pid, sig) == -1) {
        perror("kill");
        return 1;
    }

    return 0;
}
```

Primero comprobamos en el programa que haya como mínimo dos argumentos de entrada (si hay más los ignoramos). A continuación guardamos en dos variables locales los argumentos que hemos pasado como entrada (en sig hacemos el atoi

con argv[1]+1 para no incluir el - que hemos pasado en el comando) y ejecutamos la función kill con los argumentos guardados comprobando que al ejecutarlo no da errores.

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? *(0,10 ptos.)*

Abrimos des terminales y a través del programa creado enviamos una señal SIGSTOP a la otra, cuando intentamos escribir en esa terminal simplemente no podemos hacerlo, no acepta ninguna tecla que enviemos.
Cuando enviamos la señal SIGCONT a la terminal anteriormente parada, la terminal interpreta todo lo que hallamos escrito anteriormente. Por lo que si hemos escrito un comando y le hemos dado a ENTER estando parada, al hacer que continúe se ejecutará la sentencia.

<u>Ejercicio 3:</u> Captura de SIGINT. Dado el siguiente código en C, correspondiente al fichero "ejercicio_captura.c": *(0,30 ptos.)*

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

No, ya que el manejador se ejecutará siempre y cuando sigaction reciba la señal esperada para poder ejecutarlo, si nunca recibe dicha señal entonces el manejador nunca se ejecutará.

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

Este no ocurre debido a que hemos declarado `sigemptyset(&(act.sa_mask))`, lo cual inicializa el conjunto de máscaras de señales como vacío, por lo que las señales no se bloquean. La única señal que se bloquea durante la ejecución del manejador es la propia señal que está tratando la rutina manejadora, en este caso la señal SIGINT.

c) ¿Cuando aparece el printf en pantalla? *(0,10 ptos.)*

El printf aparece siempre que se ejecute el manejador, es decir, cuando el programa reciba la señal SIGINT y el manejador se ejecute, imprimirá por pantalla el mensaje indicado, puesto que hay un fflush a continuación del printf nos aseguramos de que stdout limpie el buffer e imprima el mensaje justo a continuación.

<u>Ejercicio 4:</u> Captura de Señales. *(0,30 ptos.)*

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

   Al ejecutar una señal que no tiene un manejador esta ejecutará la acción por defecto a la que está asociada. Es decir, que se ejecutará el manejador del propio sistema operativo y realizará las acciones indicadas.

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?** *(0,20 ptos.)*

   No se pueden capturar todas las señales, en general la función sigaction() puede capturar cualquier señal válida; sin embargo, las señales SIGKILL y SIGSTOP no pueden ser capturadas y ejecutarán la acción por defecto que están asignadas por el sistema operativo, que son matar y parar un proceso respectivamente.

<u>Ejercicio 5:</u> Captura de SIGINT Mejorada. Dado el siguiente código en C, correspondiente al fichero "ejercicio_captura_mejorado.c": *(0,20 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.)*

   La gestión de la señal se hace en este caso cuando se ejecuta la sentencia `if(got_signal)`, esto es debido a que cuando se recibe la señal SIGINT, el manejador únicamente cambia la variable got_signal de 0 a 1 y, una vez se vuelve a la ejecución normal de la tarea del programa, este comprueba si la variable got_signal está a 1 y realiza las acciones estipuladas.

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

   Puesto que la variable got_signal, que además es de tipo static volatile, se usa no solo en la rutina principal, si no también en el manejador, y todas tienen que ser capaces de leer y escribir el valor de esta variable para el correcto funcionamiento del programa.

<u>Ejercicio 6:</u> Bloqueo de Señales. Dado el siguiente código en C, correspondiente al fichero "ejercicio_sigset.c": *(0,30 ptos.)*

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

   Ejecutamos nuestro programa y al intentar mandar las señales SIGUSR1 o SIGUSR2 desde otra terminal. No sucede nada debido a que al incluirlas en la máscara están bloqueadas y no se ejecuta el manipulador correspondiente.

Sin embargo al enviar la señal SIGINT al programa y dado que no está en la máscara ésta se ejecuta interrumpiendo el curso del programa.

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?** *(0,20 ptos.)*

Realizamos la modificación correspondiente en el programa y, a continuación y desde otra terminal, enviamos la señal SIGUSR1 en los 10 segundos en los que el programa hace sleep(). Cuando la espera finaliza se muestra el siguiente mensaje por pantalla:

```
Señal definida por el usuario 1
```

Esto es debido a que la señal se envía en los 10 segundos de espera, pero la señal está bloqueada. En cuanto se redefine la máscara y se recupera la original, las señales se SIGUSR1 y SIGUSR2 se desbloquean y es cuando realmente se envía la señal al proceso y realiza las acciones indicadas.

<u>Ejercicio 7:</u> Gestión de la Alarma. Dado el siguiente código en C, correspondiente al fichero "ejercicio_alarm.c": *(0,20 ptos.)*

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

Si se envía la señal SIGALRM al proceso, el programa ejecutará la rutina manejador_SIGALRM definida dentro del propio programa. Por lo que instantáneamente imprimirá el mensaje indicado dentro de la rutina y terminará el proceso.

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

Al comentar la llamada a sigaction, no inicializamos el manejador para la señal SIGALRM, por lo que al ser enviada esta señal al proceso, se ejecutará la rutina por defecto para manejarla.
En este caso, se imprime por pantalla el siguiente mensaje:

```
Temporizador
```

A continuación, el proceso finaliza.

<u>Ejercicio 8:</u> Señales, Protección y Temporalización. *(2,50 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. *(2,30 ptos.)*

Escribimos un programa con los requisitos especificados cuyo código es el siguiente:

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

void manejador_SIGNAL(int sig) {
    return;
}

int main(int argc, char** argv) {
    struct sigaction act;
    sigset_t mask;
    int N;
    int T;
    pid_t pid;
    pid_t *cpid;
    int i, suma;

    /* COMPORBAMOS EL NÚMERO DE ARGUMENTOS */

    if (argc < 3) {
        fprintf(stdout, "Se esperaban 2 argumentos de entrada\n");
        exit(EXIT_FAILURE);
    }

    N = atoi(argv[1]);
    T = atoi(argv[2]);

    if ((cpid = malloc(N*sizeof(pid_t))) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
```

```
/* CREAMOS LAS MÁSCARAS */

sigemptyset(&(act.sa_mask));
act.sa_flags = 0;
act.sa_handler = manejador_SIGNAL;
if (sigaction(SIGALRM, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
if (sigaction(SIGUSR2, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
if (sigaction(SIGTERM, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}

/* CREAMOS LOS HIJOS */

for (i=0; i<N; i++) {
    if ((pid = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (!pid)
        break;
    else
        cpid[i] = pid;
}

/* CÓDIGO DEL HIJO */

if (pid == 0) {
    sigfillset(&mask);
    sigdelset(&mask, SIGTERM);

    for (i=1, suma=0; i<getpid()/10; i++) {
        suma += i;
    }
    fprintf(stdout, "(PID: %d) = %d\n", getpid(), suma);

    if (kill(getppid(), 12) == -1) {
        perror("kill");
        exit(EXIT_FAILURE);
    }
    sigsuspend(&mask);

    fprintf(stdout, "Finalizado %d\n", getpid());
    free(cpid);
    exit(EXIT_SUCCESS);
}

/* CÓDIGO DEL PADRE */

else {
    sigfillset(&mask);
    sigdelset(&mask, SIGALRM);
```

```
        alarm(T);
        sigsuspend(&mask);

        for (i=0; i<N; i++) {
            if (kill(cpid[i], 15) == -1) {
                perror("kill");
                exit(EXIT_FAILURE);
            }
        }

        fprintf(stdout, "Finalizado Padre\n");
        for (i=0; i<N; i++) {
            wait(NULL);
        }
        free(cpid);
        exit(EXIT_SUCCESS);
    }
}
```

En primer lugar realizamos un bucle for en el que el padre crea N hijos, además el padre va guardando los pid de los procesos hijo en un array.
A continuación, (si es el hijo), creamos una máscara vacía y definimos sigaction para la señal SIGTERM (la que va a recibir del padre). Se hace la suma de los números desde 1 hasta su pid/10, se imprime, se envía la señal SIGUSR2 al padre y se suspende a la espera de la señal SIGTERM, tras la cual finaliza.
Si es el padre, se define el manejador para SIGALRM y SIGUSR2, define la función alarm para T segundos y se suspende a la espera de la señal SIGALRM. Después envía la señal SIGTERM a todos sus hijos, y finaliza esperando a que lo hagan todos sus hijos.
En todos los casos, puesto que queremos “ignorar” sin bloquearla hacemos referencia al manejador vacío.

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? *(0,20 ptos.)*

Podemos ver que el número de señales en comparación con N muy bajo, al establecer un contador vemos que el número es 1.
Esto es así porque, en general, los hijos mandan todas las señales SIGUSR2 antes de que el padre vuelva a ejecutarse tras la suspensión del sigsuspend (debido a la rapidez del procesador). Por lo tanto Posix indica que hay pendiente por enviar al padre la señal SIGUSR2 (pero no tiene en cuenta el número de veces que se ha enviado la petición de envío de esta señal). Por tanto al salir el padre va a manejar la señal enviada una única vez y continuará normalmente con su ejecución.
No hay garantía de que se reciba ese número de señales, precisamente por lo explicado en el apartado anterior. Posix no tiene almacenado el número de señales que ha de enviar al programa.

<u>Ejercicio 9:</u> Creación y Eliminación de Semáforos. Dado el siguiente código en C: ¿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? *(0,20 ptos.)*

La función sem_unlink consiste en eliminar el nombre del semáforo y marcarlo para que el SO libere los recursos cuando el contador de procesos que están usando el semáforo llegue a 0. Sin embargo, el semáforo sólo se borrará cuando todos los procesos que lo usan utilicen la función sem_close.
Por tanto, y en este caso, la función puede escribirse justo después de la función sem_open ya que tanto los procesos padre e hijo podrán seguir accediendo a él hasta que ambos ejecuten la función sem_close.

<u>Ejercicio 10:</u> Semáforos y Señales. Dado el siguiente código en C: *(0,30 ptos.)*

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

   Cuando enviamos la señal SIGINT, el proceso ejecuta el manejador y, a continuación se desbloquea y sigue ejecutando el resto del programa, que en este caso es un pritnf.
   La llamada a sem_wait no se ejecuta con éxito, la razón es que el semáforo está a 0, cuando se ejecuta esta función, la llamada se bloquea hasta que se puede decrementar el semáforo (se ha realizado antes un sem_post), o un manejador de señal interrumpe la llamada. En este segundo caso, la función sem_wait no cambia el valor del semáforo (por lo que sigue valiendo 0), y devuelve -1.

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

   Al modificar el programa cuando llamamos a SIGINT se ejecutara nuestro manejador, sin embargo este tiene instrucciones de ignorar la señal. Por lo tanto cuanto intenta hacer un down del semáforo se queda a la espera de que en algún momento se haga un up o el manejador de señal interrumpa la llamada del semáforo y el programa continúe la ejecución, al realizar la llamada a SIGINT el manejador la ignora por lo que el semáforo se queda a la esperando al incremento de este que nunca llega y por lo tanto no se ejecuta la línea que imprime `fin de la espera` ni realza el `sem_unlink`.

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.** *(0,10 ptos.)*

   Una posible solución es ejecutar `sigfillset(&set)` y a continuación `sigprockmask(SIG_BLOCK, &set, NULL)` justo antes de ejecutar el sem_wait. De esta manera estamos bloqueando todas las señales que podemos bloquear antes de realizar la llamada.

Además podríamos hacer `sigprockmask(SIG_UNBLOCK, &set, NULL)` después para volver a desbloquear las señales tras el sem_wait, de esta manera conseguimos que se envíen las señales que se habrían "enviado" mientras se estaba realizando la llamada.

<u>Ejercicio 11:</u> Procesos Alternos. Dado el siguiente código en C: *(0,30 ptos.)*
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

```
/* Rellenar Código A */: Vacío
/* Rellenar Código B */: sem_post(sem1); sem_wait(sem2);
/* Rellenar Código C */: sem_post(sem1);

/* Rellenar Código D */: sem_wait(sem1);
/* Rellenar Código E */: sem_post(sem2); sem_wait(sem1);
/* Rellenar Código F */: Vacío
```

Comienza a ejecutarse el código, como queremos que el 1 se imprima primero, no escribimos nada en el código A; sin embargo, hacemos un down del sem1 en el padre (por si está en ejecución) y se queda bloqueado, por lo que pasa a ejecutarse el hijo que imprime el 1 y hace un up del semáforo en el que se había bloqueado el padre. A continuación hace un down del sem2 y entonces se bloquea el hijo, puesto que sem2 estaba inicializado a 0.
Pasa a ejecutarse el padre e imprime el número 2. Ahora el padre hace un up del sem2 en el cual estaba bloqueado el hijo y hace un down del sem1 (que ahora está a 0 tras haber realizado un down anteriormente) bloqueándose y pasando a ejecutarse el hijo que imprime el número 3.
El hijo hace un up del sem1 permitiendo así continuar al padre y que finalmente se imprima el número 4.
Ambos, padre e hijo, hacen un sem_close de los dos semáforos y el padre llaman a las funciones sem_unlink para liberar los recursos empleados por ambos semáforos cuando ya no los esté usando ningún proceso.

<u>Ejercicio 12:</u> 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: *(2,30 ptos.)*

- **Los procesos hijo leerán 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.**

Código del programa:

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

#define SEM "lectura_escritura"

static volatile sig_atomic_t got_signal = 0;

void manejador_SIGALRM(int sig) {
    got_signal = 1;
}

void manejador_SIGNAL(int sig) {
    return;
}

int main(int argc, char** argv) {
    sem_t *sem = NULL;
    struct sigaction act;
    sigset_t mask;
    int N;
    int T;
```

```
pid_t pid;
pid_t *cpid;
int i, suma;
FILE *f;
int ln1, ln2;

/* COMPORBAMOS EL NÚMERO DE ARGUMENTOS */

if (argc < 3) {
    fprintf(stdout, "Se esperaban 2 argumentos de entrada\n");
    exit(EXIT_FAILURE);
}

N = atoi(argv[1]);
T = atoi(argv[2]);

if ((cpid = malloc(N*sizeof(pid_t))) == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}

/* CREAMOS LAS MÁSCARAS */

sigemptyset(&(act.sa_mask));
act.sa_flags = 0;
act.sa_handler = manejador_SIGALRM;
if (sigaction(SIGALRM, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
act.sa_handler = manejador_SIGNAL;
if (sigaction(SIGUSR2, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
if (sigaction(SIGTERM, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}


/* CREAMOS EL SEMÁFORO */

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

/* CREAMOS LOS HIJOS */

for (i=0; i<N; i++) {
    if ((pid = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (!pid)
        break;
    else
```

```
        cpid[i] = pid;
}

/* CÓDIGO DEL HIJO */

if (pid == 0) {
    sigfillset(&mask);
    sigdelset(&mask, SIGTERM);

    for (i=1, suma=0; i<getpid()/10; i++) {
        suma += i;
    }

    /* ENTRADA A ZONA CRÍTICA */

    sem_wait(sem);

    if ((f = fopen("data.txt", "r+")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    fscanf(f, "%d\n%d", &ln1, &ln2);
    ln1++;
    ln2+=suma;
    fseek(f, 0, SEEK_SET);
    fprintf(f, "%d\n%d", ln1, ln2);
    fclose(f);

    sem_post(sem);

    /* SALIDA DE ZONA CRÍTICA */

    if (kill(getppid(), 12) == -1) {
        perror("kill");
        exit(EXIT_FAILURE);
    }
    sigsuspend(&mask);

    free(cpid);
    sem_close(sem);
    exit(EXIT_SUCCESS);
}

/* CÓDIGO DEL PADRE */

else {
    sigfillset(&mask);
    sigdelset(&mask, SIGALRM);
    sigdelset(&mask, SIGUSR2);

    if ((f = fopen("data.txt", "w+")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    fprintf(f, "%d\n%d", 0, 0);
    fclose(f);

    alarm(T);
```

```
        sem_post(sem);

        sem_unlink(SEM);

        /* SUSPENDEMOS HASTA QUE SE RECIBA SIGALRM O SIGUSR2 */
        /* SI SE RECIBE SIGALRM SALIMOS DEL BUCLE Y TERMINAMOS CON LOS PROCESOS
           HIJOS */
        /* SI SE RECIBE SIGUSR2 COMPROBAMOS QUE TODOS LOS HIJOS HAYAN ESCRITO EN
           EL FICHERO */

        do {
            sigsuspend(&mask);
            if (got_signal == 1)
                break;

            /* ENTRADA A ZONA CRÍTICA */

            sem_wait(sem);

            if ((f = fopen("data.txt", "r")) == NULL) {
                perror("fopen");
                exit(EXIT_FAILURE);
            }
            fscanf(f, "%d\n%d", &ln1, &ln2);
            fclose(f);

            sem_post(sem);

            /* SALIDA DE ZONA CRÍTICA */

        } while(ln1 != N);

        if (got_signal == 1)
            fprintf(stdout, "Falta trabajo\n");
        else
            fprintf(stdout, "Han acabado todos, resultado: %d\n", ln2);

        for (i=0; i<N; i++) {
            if (kill(cpid[i], 15) == -1) {
                perror("kill");
                exit(EXIT_FAILURE);
            }
        }

        for (i=0; i<N; i++) {
            wait(NULL);
        }
        free(cpid);
        sem_close(sem);
        exit(EXIT_SUCCESS);
    }
}
```

En primer lugar el proceso irá creando los procesos hijos, una vez creados el padre abrirá el fichero data.txt con el modo w+ por lo que si el fichero no existiera se crearía uno. Además

hará un up del semáforo que controla el acceso al fichero, el cual es la zona crítica de nuestro programa.

Los hijos a la hora de acceder al fichero para leer los datos harán un down del semáforo, por lo que si el padre aún no le he dado tiempo a crear el fichero o si ya hay otro proceso leyendo o escribiendo en el fichero, este se quedará a la espera de que hagan un up. De esta forma controlaremos la concurrencia de lectura y escritura. Una vez el proceso hijo acceda a la zona crítica leerá los datos del fichero y los incrementará correspondientemente para luego reescribirlos en el data.txt. Hará un up del semáforo saliendo así de la zona crítica y permitiendo a otro proceso acceder. Una vez hecho esto, los hijos enviarán la señal SIGUSR2 al padre y esperarán hasta que este les envíe la señal SIGTERM. Una vez la reciban, imprimirán por pantalla 'Finalizado' y su pid, y harán close del semáforo utilizado para gestionar la zona de lectura y escritura.

Por otro lado el padre una vez creado todos los hijos e inicializado el fichero llamará a la función alarm para que se envíe la señal SIGALRM tras T segundos. Mientras tanto se suspenderá, pudiendo tratar unicamente las señales SIGUSR2 Y SIGALRM. Al recibir la primera se ejecutará el manejador que hará que el programa continúe su curso, entonces leerá del fichero el número de procesos que han escrito en él y comprobará que es el mismo que el número total de procesos hijos creados. Si no fuera el mismo número, se volverá a suspender hasta que todos los hijos hayan escrito en el fichero. Durante el bucle seguiremos controlando con el semáforo el acceso a la zona crítica. Finalmente cuando salga del bucle enviará la señal SIGTERM a todos sus hijos que acabarán correctamente y los irá recogiendo haciendo un wait() por cada uno de ellos. Después imprimirá que todos han acabado y el resultado de estos, y para acabar cerrará el semáforo.
En el caso de que la señal SIGALRM llegue antes de que acabara de ejecutarse el programa, entraría a ejecutarse el correspondiente manejador que cambiará la concidción necesaria para que el padre salga del bucle, imprima 'Falta trabajo' y, antes de terminar, envíe la señal necesaria para que los hijos terminen.

<u>Ejercicio 13:</u> Comunicación entre Procesos. 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. *(Opcional; 1,00 ptos.)*
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.**

Código del programa:

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

#define SEM1 "lectura_escritura"
#define SEM2 "contador"

static volatile sig_atomic_t got_signal = 0;

void manejador_SIGALRM(int sig) {
    got_signal = 1;
}

void manejador_SIGNAL(int sig) {
    return;
}

int main(int argc, char** argv) {
    sem_t *sem1 = NULL;
    sem_t *sem2 = NULL;
    struct sigaction act;
    sigset_t mask;
    int N;
    int T;
    pid_t pid;
    pid_t *cpid;
    int i, suma;
    FILE *f;
    int ln1, ln2;
    int semvalue;

    /* COMPORBAMOS EL NÚMERO DE ARGUMENTOS */

    if (argc < 3) {
        fprintf(stdout, "Se esperaban 2 argumentos de entrada\n");
        exit(EXIT_FAILURE);
    }

    N = atoi(argv[1]);
    T = atoi(argv[2]);

    if ((cpid = malloc(N*sizeof(pid_t))) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
```

```
}

/* CREAMOS LAS MÁSCARAS */

sigemptyset(&(act.sa_mask));
act.sa_flags = 0;
act.sa_handler = manejador_SIGALRM;
if (sigaction(SIGALRM, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
act.sa_handler = manejador_SIGNAL;
if (sigaction(SIGUSR2, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
if (sigaction(SIGTERM, &act, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}


/* CREAMOS LOS SEMÁFOROS */

if ((sem1 = sem_open(SEM1, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0)) == SEM_FAILED) {
    perror("sem_open1");
    exit(EXIT_FAILURE);
}
if ((sem2 = sem_open(SEM2, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0)) == SEM_FAILED) {
    perror("sem_open2");
    exit(EXIT_FAILURE);
}

/* CREAMOS LOS HIJOS */

for (i=0; i<N; i++) {
    if ((pid = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (!pid)
        break;
    else
        cpid[i] = pid;
}

/* CÓDIGO DEL HIJO */

if (pid == 0) {
    sigfillset(&mask);
    sigdelset(&mask, SIGTERM);

    for (i=1, suma=0; i<getpid()/10; i++) {
        suma += i;
    }

    /* ENTRADA A ZONA CRÍTICA */
```

```
    sem_wait(sem1);

    if ((f = fopen("data.txt", "r+")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    fscanf(f, "%d\n%d", &ln1, &ln2);
    ln1++;
    ln2+=suma;
    fseek(f, 0, SEEK_SET);
    fprintf(f, "%d\n%d", ln1, ln2);
    fclose(f);

    sem_post(sem2);
    sem_post(sem1);

    /* SALIDA DE ZONA CRÍTICA */

    if (kill(getppid(), 12) == -1) {
        perror("kill");
        exit(EXIT_FAILURE);
    }
    sigsuspend(&mask);

    free(cpid);
    sem_close(sem1);
    sem_close(sem2);
    exit(EXIT_SUCCESS);
}

/* CÓDIGO DEL PADRE */

else {
    sigfillset(&mask);
    sigdelset(&mask, SIGALRM);
    sigdelset(&mask, SIGUSR2);

    if ((f = fopen("data.txt", "w+")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    fprintf(f, "%d\n%d", 0, 0);
    fclose(f);

    alarm(T);

    sem_post(sem1);

    sem_unlink(SEM1);
    sem_unlink(SEM2);

    /* SUSPENDEMOS HASTA QUE SE RECIBA SIGALRM O SIGUSR2 */
    /* SI SE RECIBE SIGALRM SALIMOS DEL BUCLE Y TERMINAMOS CON LOS PROCESOS
       HIJOS */
    /* SI SE RECIBE SIGUSR2 COMPROBAMOS QUE TODOS LOS HIJOS HAYAN ESCRITO EN
       EL FICHERO */

    do {
```

```
            sigsuspend(&mask);
            if (got_signal == 1)
                break;

            /* ENTRADA A ZONA CRÍTICA */

            sem_wait(sem1);

            sem_getvalue(sem2, &semvalue);

            sem_post(sem1);

            /* SALIDA DE ZONA CRÍTICA */

        } while(semvalue != N);

        if (got_signal == 1)
            fprintf(stdout, "Falta trabajo\n");
        else {
            if ((f = fopen("data.txt", "r")) == NULL) {
                perror("fopen");
                exit(EXIT_FAILURE);
            }
            fscanf(f, "%d\n%d", &ln1, &ln2);
            fclose(f);
            fprintf(stdout, "Han acabado todos, resultado: %d\n", ln2);
        }

        for (i=0; i<N; i++) {
            if (kill(cpid[i], 15) == -1) {
                perror("kill");
                exit(EXIT_FAILURE);
            }
        }

        for (i=0; i<N; i++) {
            wait(NULL);
        }

        free(cpid);
        sem_close(sem1);
        sem_close(sem2);
        exit(EXIT_SUCCESS);
    }
}
```

En este caso para no establecer la comunicación entre padre e hijo mediante un fichero lo que hemos hecho ha sido crear un nuevo semáforo N-ario el cual inicializamos a 0. Cada vez que se ejecute un proceso hijo se incrementará este nuevo semáforo en uno (lo que en realidad estamos haciendo es usarlo como contador). Al mandar la señal al proceso padre que estará suspendido, comprobará el valor del semáforo y si no coincide con el número de hijos creados volverá a suspenderse. Únicamente cuando el número de hijos coincida con el valor del semáforo el padre saldrá del bucle. Será entonces cuando lea los datos del fichero y envíe la respectiva señal a sus hijos para que todos finalicen correctamente y sean recogidos.

<u>Ejercicio 14:</u> Problema de Lectores–Escritores. *(2,40 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:** *(2,00 ptos.)*
   - **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.**

Código del programa:

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

#define SEM1 "sem_lectura"
#define SEM2 "sem_escritura"
#define SEM3 "sem_lectores"

static volatile sig_atomic_t got_signal_padre = 0;
static volatile sig_atomic_t got_signal_hijo = 0;

void manejador_SIGINT(int sig) {
    got_signal_padre = 1;
}

void manejador_SIGTERM(int sig) {
    got_signal_hijo = 1;
}

int main(int argc, char** argv) {
    sem_t *sem1 = NULL;
    sem_t *sem2 = NULL;
    sem_t *sem3 = NULL;
    struct sigaction act;
```

```
    sigset_t mask;
    int N_READ;
    int SECS;
    pid_t pid;
    pid_t *cpid;
    int i;
    int lectores;

    /* COMPORBAMOS EL NÚMERO DE ARGUMENTOS */

    if (argc < 3) {
        fprintf(stdout, "Se esperaban 2 argumentos de entrada\n");
        exit(EXIT_FAILURE);
    }

    N_READ = atoi(argv[1]);
    SECS = atoi(argv[2]);

    if ((cpid = malloc(N_READ*sizeof(pid_t))) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    /* CREAMOS LOS MANEJADORES */

    sigemptyset(&(act.sa_mask));
    act.sa_flags = 0;
    act.sa_handler = manejador_SIGINT;
    if (sigaction(SIGINT, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
    act.sa_handler = manejador_SIGTERM;
    if (sigaction(SIGTERM, &act, NULL) < 0) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    /* CREAMOS LOS SEMÁFOROS */

       if ((sem1 = sem_open(SEM1, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 1)) ==
SEM_FAILED) {
        perror("sem_open1");
        exit(EXIT_FAILURE);
    }
       if ((sem2 = sem_open(SEM2, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 1)) ==
SEM_FAILED) {
        perror("sem_open2");
        exit(EXIT_FAILURE);
    }
       if ((sem3 = sem_open(SEM3, O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0)) ==
SEM_FAILED) {
        perror("sem_open3");
        exit(EXIT_FAILURE);
    }

    /* CREAMOS LOS HIJOS */
```

```
for (i=0; i<N_READ; i++) {
    if ((pid = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (!pid)
        break;
    else
        cpid[i] = pid;
}

/* CÓDIGO DEL HIJO */

if (pid == 0) {
    while(got_signal_hijo == 0) {

        /* ENTRADA A ZONA CRÍTICA 1 */
        /* ENTRADA A ZONA CRÍTICA 2 */

        sem_wait(sem1);
        sem_post(sem3);
        sem_getvalue(sem3, &lectores);
        if (lectores == 1)
            sem_wait(sem2);
        sem_post(sem1);

        /* SALIDA DE ZONA CRÍTICA 1 */

        fprintf(stdout, "R-INI <%d>\n", getpid());
        sleep(1);
        fprintf(stdout, "R-FIN <%d>\n", getpid());

        /* ENTRADA A ZONA CRÍTICA 1 */

        sem_wait(sem1);
        sem_wait(sem3);
        sem_getvalue(sem3, &lectores);
        if (lectores == 0){
            sem_post(sem2);
        }
        sem_post(sem1);

        /* SALIDA DE ZONA CRÍTICA 1 */
        /* SALIDA DE ZONA CRÍTICA 2 */

        sleep(SECS);
    }
    free(cpid);
    sem_close(sem1);
    sem_close(sem2);
    sem_close(sem3);
    exit(EXIT_SUCCESS);
}

/* CÓDIGO DEL PADRE */

else {
    while(got_signal_padre == 0) {
```

```
            /* ENTRADA A ZONA CRÍTICA */

            sem_wait(sem2);

            fprintf(stdout, "W-INI <%d>\n", getpid());
            sleep(1);
            fprintf(stdout, "W-FIN <%d>\n", getpid());

            sem_post(sem2);

            /* SALIDA DE ZONA CRÍTICA */

            sleep(SECS);
        }

        for (i=0; i<N_READ; i++) {
            kill(cpid[i], 15);
        }

        for (i=0; i<N_READ; i++) {
            wait(NULL);
        }

        free(cpid);
        sem_close(sem1);
        sem_close(sem2);
        sem_close(sem3);
        sem_unlink(SEM1);
        sem_unlink(SEM2);
        sem_unlink(SEM3);
        exit(EXIT_SUCCESS);
    }
}
```

Para este programa creamos 3 semáforos, el primero indicará cuando puede leer un proceso, el segundo indicará cuándo puede escribir el proceso padre y el último lo utilizaremos como contador indicando el número de lectores.
En primer lugar definimos los manejadores (que cambiarán la condición necesaria para que los procesos puedan salir del bucle infinito), creamos los semáforos y a todos los hijos.
Los hijos entran en una zona crítica para incrementar en uno el número de lectores (sem3), a continuación ven el valor de sem3 (el número de lectores), si es 1 (todavía no hay nadie leyendo) decrementan el semáforo de escritura. Si el padre estaba escribiendo, entonces este ya había decrementado el semáforo en su momento y el lector no puede empezar a leer hasta que no salga el escritor. Si el lector entra en la zona de lectura entonces el escritor no puede escribir, pero sí que pueden entrar todos los lectores que quieran a la zona de lectura.
Cabe destacar que, aunque en todo momento se controla la concurrencia, después de enviarse la señal SIGINT y mientras el programa hace lo necesario para finalizar no se controlan las salidas por pantalla del programa y puede pueden solaparse algunos resultados. Esto es debido a que una señal puede hacer que un proceso salga del sem_wait.

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

Se producen tanto lecturas como escrituras debido a que al tener un único hijo cuando este salga de la lectura y quiera volver a entrar el padre tambíen estará esperando a entrar en la zona crítica y tendrán más o menos las mismas posibilidades para acceder.

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

Se siguen produciendo lecturas, sin embargo el número de escrituras ha descendido notoriamente respecto al apartado anterior. Esto se debe a que ahora el número de hijos es mayor y por lo tanto es más complicada que en la zona crítica no haya lectores para que pueda acceder el padre. Sin embargo, gracias a ese segundo, que espera cada hijo antes de volver a solicitar entrar, es suficiente para que en determinados puntos la zona crítica quede vacía y pueda acceder el escritor.

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

En este caso casi nunca se producen escrituras debido a que al ser 10 hijos cada vez que acaban solicitan inmediatamente volver a entrar a la zona crítica. Esto hace muy complicado que haya un momento en el que con mucha suerte no haya ninguno leyendo y pudiera acceder el escritor. Pero por lo general esto no ocurrirá y solo se producirán lecturas.

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? *(0,10 ptos.)*

Depende de cuántos hijos creemos, al crear solo uno se producen principalmente escrituras mientras que cuando aumenta el número de hijos se producirán únicamente lecturas. Esto se debe a que los hijos entran en bucle y no dan lugar a que haya ningún momento en el que la zona crítica esté vacía, por lo que es casi imposible que se produzca ninguna escritura.