Práctica 2: Señales y semáforos.

Autores: Marcos Aarón Bernuy, Kevin de la Coba Malam. Pareja 04 del grupo 2922.

Ejercicio 1: Comando kill de Linux.

a) *Buscar en el manual la forma de acceder a la lista de se˜nales usando el comando kill. Copiar en la memoria el comando utilizado.*

Kill -l

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

SIGKILL tiene un valor igual a 9.

SIGSTOP tiene un valor igual a 19.

Ejercicio 2: Envío de Señales.

a) *Completar el programa en C anterior para reproducir de forma limitada la funcionalidad del comando de shell kill con un formato similar:*
$ ./ sig_kill -< signal >
El programa debe recibir dos parámetros: el primero, representa el identificador numérico de la señal a enviar; el segundo, el PID del proceso al que se enviara la señal.

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

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?*

La terminal no reacciona hasta que se envía SIGCONT, al enviar SIGCONT aparecen los caracteres escritos.

Ejercicio 3: Captura de Señales.

a) *¿La llamada a sigaction supone que se ejecute la función manejador?*

No lo supone ya que el programa puede recibir la señal o no.

b) *¿Se bloquea alguna señal durante la ejecución de la función manejador?*

Sí, se bloquea la señal que se captura, a parte de esta ninguna otra.

c) *¿Cuándo aparece el printf en pantalla?*

Cuando el proceso recibe la señal SIGINT.

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?*

El programa se cierra. Por defecto el proceso detiene su ejecución y bifurca a la rutina de tratamiento de la señal (SIG_DFL), este tratamiento consiste en terminar el proceso y en algunos casos, generar un fichero core.

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?*

No se pueden capturar todas, SIGKILL y SIGSTOP son especiales ya que son las encargadas de detener y matar procesos. El sistema operativo puede necesitar matar un proceso. Es una cuestión de seguridad.

Ejercicio 4: Captura de SIGINT Mejorada.

a) *En esta versión mejorada del programa del Ejercicio 3, ¿en qué líneas se realiza realmente la gestión de la señal?*

Aunque exista un manejador el print de que se ha recibido la señal se ejecuta en las líneas 28-31 (fuera del manejador).

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

Porque si no, el proceso desde el manejador no sería capaz de decir que ha recibido la señal, la única forma que tiene de hacerlo es modificando esa variable para que cuando salga pueda hacer el print.

Ejercicio 5: Bloqueo de Señales.

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

El programa no reacciona, no hace nada cuando recibe cualquiera de las señales SIGUSR1 o SIGUSR2.

Al recibir SIGINT el programa se cierra.

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?

Cuando finaliza la espera el programa imprime "Fin del programa". Esto sucede porque al hacer sleep el proceso está dormido y las señales se quedan en espera, a parte se quedan en espera porque tenemos la máscara de señales. Al despertar las señales siguen bloqueadas por la máscara, al poner la máscara antigua, las señales llegan al proceso y vemos en pantalla el mensaje "Señal definida por el usuario 1".

Ejercicio 6: Bloqueo de Señales.

a) ¿Qué sucede si, mientras se realiza la cuenta, se envía la señal SIGALRM al proceso?

La cuenta finaliza y el programa acaba pasando por el manejador.

b) ¿Qué sucede si se comenta la llamada a sigaction?

La ejecución del programa finaliza, pero no pasa por el manejador, se imprime por pantalla “Temporizador”. Se ejecuta el manejador establecido por el sistema. Al ejecutar sigaction se hace un “link” entre la señal y el manejador, como no se ejecuta esta función el manejador sigue siendo el establecido por el sistema operativo.

Ejercicio 7: Creación y Eliminación de Semáforos.

a) ¿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?

Se puede hacer justo después de hacer open, ya que el proceso ejecutor lo mantendrá abierto. Una vez todos los procesos hagan close, el semáforo se cerrará.

Ejercicio 8: Semáforos y Señales.

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

La llamada a sem_wait no se ejecuta con éxito, se ignora el semáforo. Sem_wait devuelve error, EINTR.

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

No sucede nada, el proceso se queda esperando a que el valor del semáforo decremente.

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.

Para garantizar que no haya error en sem_wait podemos hacer un loop que haga continuamente sem_wait. Si sem_wait devuelve error, debemos comprobar si el error es EINTR, si lo es, seguimos en el loop.

Ejercicio 9: Procesos Alternos.

a) *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.

A: Vacío.

B: sem_post(sem1);

B: sem_wait(sem2);

C: sem_post(sem1);

D: sem_wait(sem1);

E: sem_post(sem2);

F: Vacío

Ejercicio 10: Concurrencia y Sincronización de Procesos.

a) *Creación de los procesos:*
- *El programa recibirá como argumento el número de procesos concurrentes totales, NUM_PROC.*
- *El primer proceso generará otro proceso, que a su vez generará otro, y así hasta alcanzar NUM_PROC procesos, siguiendo el esquema siguiente:*

Para la creación de los procesos no nos complicamos mucho, simplemente era hacer una "escalera" de procesos. Mediante un loop un proceso creaba a otro proceso y se salía, el hijo de este se quedaba en el loop hasta crear otro proceso y así sucesivamente hasta llegar al número deseado de procesos.

b) *Sincronización con señales:*
- *En cada ciclo, cada proceso enviará la señal SIGUSR1 a su proceso hijo (salvo el último proceso, que se la enviará al primero).*
- *Tras ello, imprimirá por pantalla el número de ciclo y su PID.*
- *Por último, el proceso entrará en espera no activa hasta que su proceso padre (o el último proceso, en el caso del primero) le envíe una señal y comenzará el ciclo de nuevo.*

Siguiendo el manual signal-safety tratamos de implementar los manejadores de forma que estos fuesen lo más cortos y seguros posibles, para evitar que el sistema no quede en un estado inconsciente. La acción de cada señal se ejecuta en el código principal, no en el manejador en sí, en este únicamente tenemos una variable global para que cuando entre al manejador sea modificada y cuando salga de este, se pueda comprobar si ha sido modificada o no, en dicho caso se reestablecería el valor de la variable.

En cuanto a cómo el padre se comunica con su hijo, lo que hicimos fue que, al crear los procesos, el padre antes de salir del bucle guardase el PID de su hijo. Con este esquema tendríamos que todos los procesos guardan un PID exceptuando el último, que no tiene hijo. Como el último proceso no tiene hijo, guardamos antes del bucle el PID del primer proceso y así el último proceso envía la señal correspondiente al primer proceso. Esto se ve reflejado en send_signal, una función que creamos para simplificar código.

c) *Terminación:*
   - *El ciclo terminará cuando el primer proceso (y solo él) reciba la señal SIGINT. En ese momento, enviará la señal SIGTERM a su proceso hijo.*
   - *Cada proceso, al recibir la señal SIGTERM, se la reenviará a su proceso hijo (si lo tiene), liberará los recursos y terminará.*

Lo primero que hicimos fue otro manejador, la estructura de este es igual a la de SIGUSR1 (se modifica una variable global). Después mediante máscaras, hicimos que el proceso hijo solo pudiese recibir SITERM y SIGUSR1, y el padre solo podía recibir SIGUSR1 y SIGINT.

En cuanto el primer proceso recibe la señal SIGINT, este envía una señal SIGTERM a su hijo, iniciándose así una cadena hasta llegar al último proceso, en este momento el último proceso termina su ejecución liberando los recursos utilizados, después su padre termina su ejecución, esto se hace gracias a un wait específico para que los procesos esperen a su hijo. De esta forma el primer proceso solo se termina su ejecución cuando todos los demás han terminado su ejecución.

d) *Protección de zonas críticas:*
   - *Es necesario garantizar que el ciclo es robusto, de manera que no se pierda ninguna señal.*
   - *Para ello, se deberán bloquear las señales y realizar las esperas no activas con sigsuspend.*

Aquí lo que hicimos fue una máscara que bloquease todas las señales posibles para después ejecutar sigsuspend con otras máscaras. De esta forma, el hijo por ejemplo mientras se ejecuta tendría todas las señales bloqueadas en una cola, pero en cuanto llegase a sigsuspend se desbloquearían las señales que la máscara usada en sigsuspend tiene libres (SIGUSR1 y SIGTERM). Así evitamos que el programa entre en un estado de espera del que no puede salir porque la señal puede llegar antes de iniciar la espera.

e) *Temporización:*
   - *En el caso de que transcurran 10s de ejecución, el primer proceso procederá a terminar el ciclo enviando la señal SIGTERM a su proceso hijo.*

Para esto establecimos otro nuevo manejador para la señal SIGALRM que en cuanto pasan 10 segundos comienza la cadena de SIGTERM's.

f) *Análisis de la ejecución:*
   - *Analizar los mensajes que se imprimen por pantalla. ¿Se están ejecutando los procesos en el orden que les corresponde? ¿Hay garantías de que sea así?*

   No se ejecutan en el orden que corresponde. Es posible que la ejecución salga en orden, pero no tenemos garantías de esto ya que esto depende del manejador.

   - *Repetir el análisis anterior, pero tras introducir una espera aleatoria en cada proceso, después de enviar la señal al siguiente proceso pero antes de imprimir por pantalla. ¿Están ahora ordenados los mensajes? ¿Por qué?*

   Sí ahora los mensajes están ordenados independientemente del manejador. Esto es debido a que al hacer sleep damos tiempo a que si o si, el proceso que envía una señal, se suspenda esperando otra señal, antes de que su hijo envíe una señal.

g) *Sincronización con semáforos:*
   - *Añadir los mecanismos necesarios, usando semáforos, para que los procesos impriman los mensajes en orden, de tal forma que un proceso no imprima su mensaje hasta que el proceso anterior haya impreso el suyo (a pesar de que le haya enviado ya la señal).*

Para sincronizar los procesos con semáforos decidimos crear 2 semáforos binarios, uno específico para los prints y otro para coordinar los ciclos. El primer semáforo es muy simple, simplemente cuando un proceso quiere escribir este hace wait, al terminar de escribir este hace post. El segundo semáforo coordina el ciclo, esto lo hacemos haciendo que el primer proceso no envíe ninguna señal hasta que el último proceso se lo permita.