Junco de las Heras & Marta Vaquerizo

SISTEMAS OPERATIVOS
PRÁCTICA 2

Ejercicio 1: Comando *kill* de Linux.

a) El comando utilizado para acceder a la lista de señales usando *kill* es:
kill –l (o también kill -L)

b) El número de la señal SIGKILL es el 9, y el de la señal SIGSTOP es el 19.

Ejercicio 2: Envío de señales.

a) El código de este apartado se encuentra en **ejercicio_kill.c**, con sus respectivos comentarios.

b) Lo que hace SIGSTOP es suspender el proceso, similar a si en la terminal hacemos Ctrl+Z, así que no se puede ni escribir ni ejecutar comandos en la terminal. Cuando recibe SIGCONT entonces se renauda el proceso, pudiendo volver a ejecutar los comandos de nuevo.

Ejercicio 3: Captura de SIGINT.

a) No, solo se ejecutará la función manejador cuando reciba la señal SIGINT (o Ctrl +C).

b) Solo bloquea SIGINT, y no se bloquea ninguna señal más, ya que el código hace **sigemptyset** de la máscara de act, es decir, que se vacía la lista de señales a bloquear, bloqueando solo la señal que activa esa máscara, es decir, SIGINT.

c) Cada vez que el proceso recibe SIGINT, se hace una llamada a la función manejador, ejecutándose el printf.

Ejercicio 4: Captura de Señales.

a) Que es el sistema el que se encarga de qué hacer con la señal recibida, por ejemplo acabando la finalización del proceso.

b) El proceso bloquea todas las señales menos la 9 (SIGKILL) y la 19 (SIGSTOP), ya que esas no se pueden capturar ni ignorar.

Ejercicio 5: Captura Mejorada de SIGINT

a) En este nuevo código, la gestión de la señal se realiza dentro del while, (35, 38), cuando se comprueba si la bandera, que se activa cuando se recibe la señal, está activada o no, realizando lo necesario.

b) En este caso se usa una variable global para que tanto el manejador como el programa puedan acceder a ella, y de esa manera poder controlar la señal.

Ejercicio 6: Bloqueo de Señales

a) Cuando el programa recibe SIGUSR1 o SIGUSR2, como están bloqueadas, el sistema no las toma en cuenta, y como se ha ejecutado la función **pause()**, el sistema sigue esperando una señal. Por el contrario, cuando recibe SIGINT, el programa termina sin imprimirse el último printf.

b) Cuando acaba la espera, no se imprime el mensaje de salida "Fin del programa". Esto se debe a que cuando se bloquea una señal, esta no se desecha, sino que espera a ser ignorada o desbloqueada, por lo que, como tras la espera se restaura la máscara del principio del programa, en la que no se bloqueaba ni a SIGUSR1 ni a SIGUSR2, estas señales quedan desbloqueadas tras la espera, y se ejecutan, dando lugar a que finalice el proceso de forma abrupta.
El código añadido es:
sigprocmask (SIG_UNBLOCK, &set, &oset);

Ejercicio 7: Gestión de Alarma

a) Si se lanza la señal SIGALRM cuando se está ejecutando el programa, el programa llama al manejador, muestra el mensaje "Estos son los números que me ha dado tiempo a contar", y acaba sin mostrar por pantalla "Fin del programa".

b) Si se comenta la llamada a **sigaction**, cuando se lanza la señal SIGALRM, el programa finaliza de forma abrupta (retorno distinto de 0) y se muestra el mensaje de "Alarm Clock" por pantalla.

Ejercicio 8: Señales, Protección, y Temporización

a) El código con los requisitos que se piden se encuentra en el fichero ***ejercicio_prottemp.c*****.** En el programa se han creado dos funciones: una es el manejador, que recibe las señales SIGALRM, SIGTERM Y SIGUSR2, y guarda un 1 en las variables globales asociadas a cada una de ellas, cuando se han recibido. Esto se puede hacer, ya que no hay problemas de conflicto de señales entre los hijos y el padre. La otra función es **f_hijo**, que es el código que ejecuta cada hijo. Esta función hace el trabajo de sumar desde 1 hasta PID/10, y luego envía con kill la señal SIGUSR2 al padre. Espera con **sigsuspend** a recibir la señal SIGTERM del padre.

La función main carga todos los datos primero, y luego prepara el manejador para capturar las tres señales. Después el padre lanza los N hijos. Tras ello, se genera una alarma, y se espera con sigsuspend los T segundos, hasta que acaba la alarma. Uno por uno, el padre envía la señal SIGTERM, con la función kill, a sus N hijos, e imprime: cuántas señales SIGUSR2 recibe, el mensaje de "Finalizado Padre", espera a todos sus hijos, y acaba.

b) En este apartado, se ha creado una variable **static volatile int num_usr2**, que se incrementa en el manejador cada vez que se recibe la señal SIGUSR2. Esto se ha utilizado para contar cuantas señales de este tipo recibe el padre. La variable num_usr2 está acotado por 1 <= num_usr2 <= N, siempre que N > 0. El caso mínimo sucede cuando un proceso lee que num_usr2 = 0, vaya a incrementarla, pero se acaba su tiempo de procesador, entonces no logra incrementarla, y pasa a otro proceso, al que le ocurre lo mismo, y así con todos los procesos. Luego, cuando el planificador le da el tiempo de procesador al primer proceso que intentó incrementar la variable, este escribe que vale 1, pero cuando se le da el tiempo a los demás procesos, les va a ocurrir lo mismo que al primero, porque para ellos num_usr2 era 0. El caso máximo se da cuando hacen correctamente el num_usr2++ en orden, es decir, no hay interrupciones por parte del planificador, y num_usr2 llega a N.

Para evitar este problema tendría que protegerse la variable num_usr2, con un semáforo por ejemplo.

Ejercicio 9: Creación y Eliminación de Semáforos.

Si, podría modificarse. El **sem_unlink** se podría poner justo después de haberlo creado, ya que como los procesos que usan el semáforo están en el mismo programa, si se hace **sem_unlink** y estos siguen usando el semáforo, este no se va a borrar hasta que ambos procesos dejen de usarlo. También se podría poner en cualquier parte del programa, pero si hay un error en algún sitio entonces hay posibilidades de que no se ejecute **sem_unlink**, por lo que el semáforo podría no borrarse, es decir, quedaría en el sistema operativo, así que es más correcto ponerlo nada más haber sido creado.

Ejercicio 10: Semáforos y Señales.

a) Cuando se lanza la señal SIGINT, el programa termina con "Fin de la espera". Además, la llamada a **sem_wait** no se ejecuta con éxito, ya que es interrumpida por la rutina del manejador, y hace que salte a la siguiente línea imprimiendo "Fin de la espera".

b) Como la señal se ignora con un **signal(SIGINT, SIG_IGN);** antes de hacer el **sem_wait(sem)** no se llama al manejador de señales ni al nuestro ni al que hay por defecto, por lo que **sem_wait** no se detiene si recibe la señal SIGINT (aunque sí se desbloquea si le llega otra señal no ignorada).

c) Sabiendo que **sem_wait** devuelve 0, en caso de que pueda modificar al semáforo, y -1 si ha sido desbloqueada porque se ha ejecutado el manejador de señales, para encubrirla, se ha decidido poner un bucle, para que se repita hasta que devuelva 0:
while(sem_wait(sem) == -1);

Ejercicio 11: Procesos Alternos.

En el hueco A, no se ha puesto nada, ya que se quiere imprimir primero, el número uno.

En el hueco D se ha puesto un **sem_wait(sem1)**, para que se bloquee el proceso padre, y se ejecute el hijo primero, imprimiendo el número uno.

En el hueco B, se ha puesto un **sem_post(sem1)**, para que se pueda ejecutar el padre, que estaba bloqueado, y así se imprimirá un dos. En este mismo hueco, también se ha puesto un **sem_wait(sem2)**, para que se bloquee el hijo mientras el padre imprime el número dos.

En el hueco E, se ha puesto un **sem_post(sem2**), para que se desbloquee el proceso hijo, y se imprima el número tres, y también se ha puesto un **sem_wait(sem1)**, para que se bloquee el padre mientras el hijo imprime el número tres.

En el hueco C, se ha puesto un **sem_post(sem1)**, para que se desbloquee el proceso padre y se imprima el número cuatro.

El hueco F queda vacío, ya que se ha impreso todo lo que se quería. Los semáforos están como al principio (con el valor de 0) y se libera su memoria.

Ejercicio 12: Concurrencia

El código de este ejercicio se encuentra en el fichero ***ejercicio_prottemp_mejorado.c***.

En este ejercicio, se ha usado el mismo código del ***ejercicio_prottemp.c***, del ejercicio 8, al que se ha añadido semáforos y la lectura a fichero.

Se ha creado una función **lee_numeros(int *n1, int *n2)**, para no repetir código y tenerlo más limpio, que lee del data.txt (o del nombre puesto en FILENAME) el n1, el \n y el n2. El \n es porquese necesita que estén los números en líneas distintas, como piden en el enunciado. También se ha creado la función **escribe_numeros(int n1, int n2)**, que escribe n1, \n y n2 en FILENAME. Ambas funciones devuelven 0, si lo han podido hacer bien, o -1 si ha habido algún fallo. En la función **f_hijo**, se ha añadido el semáforo SEM_NAME, y en la sección crítica se hace la lectura de los dos números n1, n2, luego la escritura de n1+1 (un hijo ha escrito más) y n2+sum (la suma que había antes + la suma actual), y, al final, manda la señal SIGUSR2 al padre. El envío de la señal también tiene que estar en la zona crítica porque está sumando la variable global num_usr2.
En la función main, se ha añadido el semáforo del padre, que escribe 0, 0. Luego crea a los N hijos igual que en el ejercicio 8, y en la condición de sigsuspend, se ha incluido las siguientes líneas de código:

if(num_usr2 == N) {lee_numeros, printf, imprimo= 1, break;}

Se ha llevado a cabo esta estrategia, ya que no se puede escribir un solo break. Esto se debe a que podría darse el caso en el que se tiene que num_usr2 == N, y se hace un break. Pero justo después, le quita el procesador al proceso que se está ejecutando, entonces, cuando le vuelve el turno, justo llega el final de la alarma, y se imprime por pantalla "Falta trabajo", pero en realidad no falta.

Ejercicio 13: Comunicación entre Procesos (Opcional).

El código de este ejercicio se encuentra en el fichero *ejercicio_prottemp_mejorado_op.c*.

En el ejercicio anterior, para comprobar si el padre había recibido las N señales de SIGUSR2, no se abría el fichero, sino que se miraba en el contador num_usr2, ya que esta forma era más óptima, porque cuantas más veces se abra un fichero más posibilidades hay de que haya algún fallo. Es por esto que nuestro ***ejercicio_prottemp_mejorado_op.c*** es ***ejercicio_prottemp_mejorado.c***, cambiando la condición del bucle para parar, en vez de recibir la alarma, espera a que num_usr2, un tipo de semáforo N-ario, llegue a N, gracias a la implementación que se ha hecho en el ejercicio anterior.

Ejercicio 14: Problema de Lectores-Escritores

a) El código de este ejercicio se encuentra en el fichero **ejercicio_lect_escr.c.**

En todos los apartados siguientes las respuestas absolutas son que **no se puede predecir si va a haber lecturas ni escrituras**, ya que todo depende de a qué proceso le da primero el procesador el tiempo de ejecución, por lo que no se puede saber. Las siguientes respuestas son simplemente descripciones de una ejecución de programa (al final del ejercicio está la captura de la evidencia), que probablemente es lo que va a pasar, pero con certeza no puede saber.

b) Sí, hay tanto escrituras como lecturas. El tiempo entre escribir el _INI y el _FIN es 1 segundo, y hay 1 lector y 1 escritor, así que aproximadamente los procesos de escritura y los de lectura se intercalarán. La evidencia de ejecución se encuentra al final del ejercicio.

c) Sí, hay tanto escrituras como lecturas. Hay muy pocas escrituras, debido a que como hay tantos lectores tienen más probabilidades de recibir el procesador. Evidencia de ejecución al final del ejercicio.

d) No, solo hay lecturas. Esto es debido a que no hay tiempo de **sleep** entre que acaba la lectura y vuelve en el bucle, así que al haber 10 procesos compitiendo por el procesador contra 1 de escritura, al de escritura tiene muy pocas probabilidades de que se lo den, tendría que coincidir que todas las lecturas estén acabando de leer, y el proceso de escritura reciba el procesador. Evidencia de ejecución al final del ejercicio.

e) Lo que sucede ahora se podría asemejar a esperar mucho tiempo con el **sleep(1)**. Ahora hay tanto lecturas como escrituras. La prueba de ello es que con SECS = 0 y N_READ = 10 antes no había escritura (sí que había pero tenía muy pocas probabilidades de ejecutarse), ahora poniendo un break en la parte de escritura se puede comprobar si se ejecuta la escritura o no, y apenas pasan unos segundos cuando el programa acaba, evidenciando que ha habido una escritura.