INFORME PRÁCTICA 2
SOPER

Álvaro Castillo García

Pablo Ernesto Soëtard

Universidad Autónoma de Madrid

Escuela Politécnica Superior

Ingeniería Informática bilingüe, grupo 2292

Sistemas Operativos

ÍNDICE

EJERCICIO 1

a)

El comando para encontrar la información en el manual es: man kill.
Para acceder a la lista de de señales se debe escribir la bandera -l; es decir kill -l.

b)

La señal SIGKILL tiene el número 9, mientras que SIGSTOP es el número 19.

EJERCICIO 2

a)

El programa que describe este ejercicio se encuentra en el archivo *ejercicio_kill.c*

Se usan las librerias standard: <stdlib.h> y <stdio.h>. También está incluida <string.h> que permitirá imprimir en la salida de error la sentencia que devuelva kill(). La librería <signal.h> es necesaria para que esta última función se pueda ejecutar.

b)

Si se intenta escribir en la otra terminal podemos comprobar que no aparece lo que escribimos. Tras ejecutar el programa con la señal SIGCONT, vemos como aparece repentinamente todo lo que hemos escrito, mientras que la terminal estaba “bloqueada”. Observamos como se vacía el buffer de stdin, que antes permanecía lleno.

EJERCICIO 3

a)

La llamada a sigaction, establece la acción a realizar cuando se reciba una señal, pero no implica que la función manejador se vaya a ejecutar según termine sigaction.

b)

Se bloquea la propia señal del manejador, en este caso SIGINT a no ser que se especifique SA_NODEFER en la mascara de señales.

c)

Según presionamos las teclas CTRL+C aparece en la pantalla: Señal número 2 recibida. Esto se debe a que justo en ese momento, el proceso recibe la señal de SIGINT y llama a la función manejador, que imprimirá esa frase en la terminal.

EJERCICIO 4

a)

El programa ejecutará el manejador por defecto que normalmente hará que acabe de manera abrupta, si el programador no ha especificado un manejador para esa señal.

b)

Se pueden capturar todas las señales menos la 9 (SIGKILL) y la 19 (SIGSTOP), ya que estas señales no pueden ser bloqueadas por sigaction debido a que el SO necesita poder detener cualquier proceso.

EJERCICIO 5

a)

A pesar de que haya un manejador para las señales, este simplemente asigna un valor a una variable. Realmente la gestión de la señal se realiza desde la función main en las líneas 28, 29, 30 y 31.

b)

Esta variable global está permitida, ya que está declarada como sig_atomic_t, lo que provoca que se pueda acceder a ella en presencia de interrupciones asíncronas generadas por señales. Además, a través de esta variable podemos comunicar el manejador con el main para que realice una cosa u otra dependiendo de si recibe o no una señal.

EJERCICIO 6

a)

Al recibir SIGUSR1 o SIGUSR2 el programa ignora las señales, al haber sido bloqueadas con anterioridad. Por el contrario al recibir SIGINT, el programa acaba directamente, sin imprimir “Fin del programa”.

b)

Al acabar el programa y haber recibido la señal SIGUSR1 durante el sleep(10), se imprime “Señal definida por el usuario 1” y finaliza el programa sin imprimir “Fin del programa”. Esto se debe a que la señal es bloqueada durante el sleep(10) pero cuando este acaba y se restaura la máscara original, esta señal que estaba en espera es recibida y ejecuta el manejador por defecto que hace que el proceso finalice, imprimiendo el mensaje anteriormente mencionado.

EJERCICIO 7

a)

Si mientras se realiza la cuenta, se envía la señal SIGALARM al proceso, este terminará, imprimiendo por pantalla: El temporizador virtual llegó al final. Esto ocurre, ya que al enviar externamente esta señal, estamos provocando que el proceso llame al manejador por defecto para SIGALARM, en vez de ejecutar la función determinada por el programador, que estaba esperando a que la función alarm() enviase la señal.

b)

Si se comenta la llamada a sigaction, cuando acabe la cuenta de la alarma y se envíe la señal de SIGALARM, se ejecutará el manejador por defecto para esa señal. En este caso, se termina el programa, mientras imprime en pantalla: Temporizador.

EJERCICIO 8

a)

El programa que describe este ejercicio se encuentra en el archivo *ejercicio_prottemp.c*

Se usan las librerias standard: <stdlib.h> y <stdio.h>. También se incluyen <unistd.h>, <sys/types.h> y <sys/wait.h> para poder crear procesos hijos y coordinarlos con el padre. La librería <signal.h> es necesaria para todas las funciones que manejan las señales.

Para suspender el proceso tras la llamada a alarm(), hemos credo un manejador para la señal SIGALRM. Este manejador cuando se ejecuta simplemente cambia el valor de una variable global usada como bandera. El proceso padre quedará en un bulce while, realizando la función sigsuspend(&mask), y cuando le llegue una señal, comprobará si el valor de la variable global ha cambiado, con el fin de ver si la señal recibida ha sido SIGALRM u otra distinta.

Hemos usado el mismo método en los hijos, con el manejador de SIGTERM. De esta manera, nos aseguramos, que el proceso continúe sólo cuando reciba una determinada señal. En caso contrario y dependiendo de la señal, el proceso finalizará.

b)

Al establecer el contador de señales podemos observar como el número de señales SIGUSR2 es menor que N. No se puede garantizar de ninguna manera que el número de señales SIGUSR2 recibidas sea igual a N, esto se debe a que al recibir una señal SIGUSR2, el manejador se ejecuta y si mientras se está ejecutando se recibe otra señal SIGUSR2, esta queda bloqueada hasta que el actual manejador se termine de ejecutar. Pero si a su vez, llega una tercera señal SIGUSR2 mientras se ejecuta el manejador de la primera, esta señal queda desechada y nunca ejecuta su manejador.

EJERCICIO 9

Sí, sería posible cambiar de sitio a la llamada sem_unlink(). La primera posición correcta, sería tras la llamada a sem_open(), para que cuando el contador de procesos asociados al semáforo llegue a 0 (los dos procesos realizan sem_close()), este se elimine.

EJERCICIO 10

a)

Si se envía la señal SIGINT, se ejecuta el manejador y se imprime “Fin de la espera”. La llamada a sem_wait() se ejecuta, hasta que es interrumpida por la señal recibida y su manejador. En ese momento, sem_wait() devuelve -1, ya que, al ser una función atómica y no haberse ejecutado como tal, el semáforo queda con el mismo valor y la llamada devuelve un error (-1).

b)

Si se ignora la señal con SIG_IGN, el proceso quedará en espera. Si se envía la señal SIGINT, el proceso no la hará caso, y seguirá esperando. Quedará parado en el mismo estado, a no ser que reciba otra señal distinta y acabe con el proceso.

c)

Se podría hacer una comprobación de lo que ha devuelto sem_wait() y en el caso de que no sea 0 que se la vuelva a llamar hasta que devuelva 0, esto querrá decir que se ha conseguido realizar el Down del semáforo correctamente.

EJERCICIO 11

A → Vacío

B → sem_post(sem1);

sem_wait(sem2);

C → sem_post(sem1);

D → sem_wait(sem1);

E → sem_post(sem2);

sem_wait(sem1);

F → Vacío

Cuando se crean ambos semáforos, sus valores se inician a 0. Para que se imprima el 1 antes que el 2, debemos añadir un sem_wait(sem1) antes de la llamada a printf(“2”), ya que la impresión del número 2, tendrá que esperar a que se incremente el semáforo 1 para ejecutarse. Se imprimirá entonces el número 1, ya que no posee ninguna restricción. Tras esta línea de código debemos incrementar el valor del semáforo 1 (sem_post(sem1)), para que el número 2 se pueda imprimir. También debemos llamar a sem_wait(sem2), para que no escriba el número 3 antes que el 2, y se quede bloqueado. Esto ocurrirá hasta que se incremente el valor del segundo semáforo, tras imprimir el número 2. Para ello, se escribirá sem_post(sem2) y se realizará un decremento del semáforo 1 (sem_wait(sem1)), para que el número 4 no se imprima hasta que el 3 no haya acabado. En ese momento, se dará paso a la instrucción printf(“3”). Por último, tras la impresión del 3, se usa un sem_post(sem1), que desbloqueará el printf(“4”), y acabará el programa.

EJERCICIO 12

El programa que describe este ejercicio se encuentra en el archivo *ejercicio_prottemp_mejorado.c*

Se usan las librerias standard: <stdlib.h>, <stdio.h>, así como la de manejo de errores <errno.h> y de strings <string.h>. También se incluyen <unistd.h>, <sys/types.h> y <sys/wait.h> para poder crear procesos hijos y coordinarlos con el padre. La librería <signal.h> es necesaria para todas las funciones que manejan las señales. Por otra parte se incluyen <semaphore.h> y <fcntl.h> para poder utilizar los semáforos.

Para realizar las lecturas y escrituras en el archivo datos.txt, de manera controlada y ordenada, se ha utilizado un semáforo (sem1). El proceso padre quedará en un bucle mientras que la variable global del manejador de alarma no esté activada o hasta que lea en el fichero que todos los hijos han escrito.

Hemos creado un archivo llamado *funciones_prottemp_mejorado.c*, con su correspondiente header (*funciones_prottemp_mejorado.h*), en el que incluimos las funciones comunes a los procesos o alguna que ha de realizarse varias veces. Encontramos así: *long long suma_numeros()* , función que sumará los números entre el 1 y pid del proceso/10; *void escribir_inicio(),* función que inicializa el archivo con dos ceros; *void escribir_resultado(sem_t *sem1,long long suma),* función que escribirá el resultado de cada proceso hijo en el fichero y *double comprobar_resultado(sem_t *sem1, long long * resultado),* que leerá del fichero los dos datos.

Hemos decidido usar un long long como tipo de variable para guardar la suma que se encuentra en el fichero.

EJERCICIO 13

El programa que describe este ejercicio se encuentra en el archivo *ejercicio_prottemp_mejorado_op.c*

Las funciones comunes con el ejercicio 12, están en el archivo *funciones_prottemp_mejorado.c*

Se usan exactamente las mismas librerías que en el ejercicio 12.

Para resolver este ejercicio, hemos decidido, crear un semáforo n-ario. Este semáforo (sem2), se inicializa a 0, y cada vez que uno de los hijos escribe en el archivo, se incrementa con sem_post(sem2). El padre quedará en un bucle while, comprobando si el valor de sem2 es igual a el número de hijos que ha lanzado. En el momento en el que esos números coincidan, el padre lee el archivo, manda la señal SIGTERM a los hijos y los espera. De esta manera, quzás se consuman más recursos que con otro algortimo, ya que el padre queda en un bucle comprobando constantemente su condición, pero nos aseguramos que este lee el archivo cuando todos los hijos hayan escrito.

Como en el enunciado no se especifica nada, el proceso padre queda primero suspendido hasta que le llega la señal de alarma y luego ya realizará todo lo demás.

EJERCICIO 14

a)

El programa que describe este apartado se encuentra en el archivo *ejercicio_lect_escr.c*

Se usan las librerias standard: <stdlib.h>, <stdio.h>, así como la de manejo de errores <errno.h> y de strings <string.h>. También se incluyen <unistd.h>, <sys/types.h> y <sys/wait.h> para poder crear procesos hijos y coordinarlos con el padre. La librería <signal.h> es necesaria para todas las funciones que manejan las señales. Por otra parte se incluyen <semaphore.h> y <fcntl.h> para poder utilizar los semáforos.

Para que el programa acabe, el usuario ha de mandar la señal SIGINT al proceso padre a través del programa ejercicio_kill. Si desde la terminal pulsa las teclas Ctrl+C, el proceso padre también finalizará, pero el programa contará con varias pérdidas de memoria y errores de valgrind. Esto se debe a que en los hijos no se ha introducido un manejador para la señal SIGINT (ya que no se especifica en el enunciado) y por tanto acaban inmediatamente, sin llegar al final del código ni liberar memoria.

b)

Al ejecutar el programa con SECS = 0 y N_READ = 1 se puede observar como se realiza una lectura completa y una escritura completa intercalándose cada 1 segundo. Esto se debe a que solo hay un proceso lector, por lo que cuando uno de los dos (lector o escritor) acaba de realizar su operación, el otro se ejecuta, poniendo en espera al anterior, y así sucesivamente.

c)

Al ejecutar el programa con SECS = 1 y N_READ = 10 se puede observar como se producen 10 lecturas completas y a continuación una escritura completa, esto se repite hasta que llega una señal SIGINT al padre. Esto se debe a que en cuanto uno de los procesos lectores se le permite escribir, este pone en espera al escritor, y entonces todos empiezan a leer, pero cuando todos han acabado de leer y se activa al escritor, todos permanecen bloqueados hasta que el escritor acabe de escribir, y así sucesivamente.

d)

Al ejecutar el programa con SECS = 0 y N_READ = 10 se puede observar como solo se producen 10 lectura completas cada segundo, y cuando se recibe la interrupción se ejecuta una escritura. Esto se debe a que el contador de lecturas nunca tiene el tiempo suficiente para bajar a 0 y así liberar a la escritura, por lo que nunca se escribirá hasta que todos los lectores hayan acabado y por ende el contador de lecturas esté a cero.

e)

Esto produciría que se generasen lecturas y escrituras de forma desordenada, es decir, normalmente se producirán muchas más lecturas que escrituras, y puede darse el caso de que no se den escrituras.

Esto se debe a que para que se pueda realizar una escritura, el contador de lectores tiene que estar a 0, por lo que esto es cada vez menos probable cuanto más lectores haya.