Sistemas Operativos. Práctica 2.

Pablo Cuesta Sierra y Álvaro Zamanillo Sáez

30 de marzo de 2021

Índice

Ejercicio 1

a) La lista de señales se obtiene con el comando:

```
kill -l
```

b) `SIGKILL` corresponde al número 9, y `SIGSTOP`, al número 19. En este caso lo podemos ver directamente usando: `kill -l <Nombre_de_la_señal>`

Ejercicio 2

a) Basta con añadir el siguiente código en el hueco señalado:

```
if(kill(pid, sig) != 0){
    perror("kill");
    exit(EXIT_FAILURE);
}
```

b) Si mandamos la señal `SIGSTOP`, al intentar excribir en la terminal, el texto que se introduce por el teclado, no aparece. Cuando mandamos la señal `SIGCONT`, el texto que habíamos intentado insertar antes

aparece de golpe y la terminal vuelve a la normalidad. De hecho, si excribimos algún comando y presionamos enter después de haberle mandado la señal `SIGSTOP`, en cuanto se mande la señal `SIGCONT`, se ejecuta el mismo de inmediato.

Ejercicio 3

a) La llamada a `sigaction` no supone que se ejecute la función `manejador`, sino que establece que esa es la función que este proceso ejecutará para manejar la interrupción causada por la señal indicada en el primer argumento de `sigaction`.

b) En este caso, la única señal que se bloquea durante la ejecución de `manejador` es la señal que ha provocado su ejecución: `SIGINT`. Esto se debe a que `act.sa_mask` no contiene ninguna señal.

c) Lo primero que aparece en pantalla es el `printf` del bucle. Luego el proceso se bloquea en la orden `sleep(9999)`, y cuando se manda la señal `SIGINT` con el teclado, aparece el `printf` de la función `manejador`. `sleep`, es interrumpida por la ejecución del manejador, por lo que se vuelve a ejecutar el `printf` del bucle.

d) Si modificamos el programa para que no capture `SIGINT`, cuando pulsemos `Ctrl+C`, terminará la ejecución. Esto se debe a que si un proceso recibe una señal y no la captura, se ejecutará el manejador por defecto, el cual, en general, termina el proceso.

e) Las señales `SIGKILL` y `SIGSTOP` no pueden ser capturadas, ya que sólo las puede manejar el núcleo. Si no fuera así, el sistema operativo no tendría posibilidad de parar un proceso que no responde.

Ejercicio 4

a) La gestión de la señal se realiza cuando el programa llega al bloque `if(got_signal){...}`

b) Porque el manejador no puede recibir ningún argumento (aparte de la señal), por tanto la variable que indica que se ha recibido la señal tiene que ser global.

Ejercicio 5

a) Si se envia SIGUSR1 o SIGUSR2, no ocurre nada, ya que las señales están bloqueadas y, por ende, quedan pendientes. En cambio, si se envía la señal SIGINT, el proceso finaliza puesto que esta no está bloqueada.

b) Cuando acaba la espera, finaliza el proceso. No se imprime el mensaje. Si mandamos una de estas señales que están bloqueadas, en cuanto finaliza la espera y se desbloquean las señales, que estaban pendientes, se lleva a cabo la rutina que la maneja, que termina el programa sin que se imprima el mensaje de despedida.

Ejercicio 6

a) Si se le manda al proceso la señal `SIGALRM` antes de que terminen los 10 segundos, ocurre lo mismo que si se acabara de terminar el tiempo, se ejecuta el manejador establecido por `sigaction` (que finaliza el proceso tras imprimir un mensaje).

b) Si no se ejecuta `sigaction`, al recibir la señal `SIGALRM`, se atiende con la rutina por defecto, que imprime el mensaje '`Temporizador`' y finaliza el proceso.

Ejercicio 7

`sem_unlink` se puede llamar en cualquier momento después de que se cree el semáforo. Como pone en el manual, `sem_unlink` borra el nombre del semáforo, de modo que si algún otro proceso quiere abrir un semáforo con ese mismo nombre desupés de que se llame a `sem_unlink`, tendrá que crearlo. Sin ambargo, estos dos procesos (padre e hijo) tendrán acceso a este semáforo hasta que lo cierren.

Ejercicio 8

a) Si se realiza una llamada a `sem_wait` y el semáforo está a cero, esta llamada bloquea el proceso hasta que el semáforo aumente o hasta que algún manejador interrumpa esta espera. Esto último es lo que pasa en este caso.

b) Como mandamos que se ignore la señal, ningún manejador interrumpe la llamada a `sem_wait`. En este caso, la función `sem_wait` no llega a devolver, por lo que no se ejecutará la orden `printf`.

c) Habría que sustituir la línea en la que se llama a `sem_wait` por: `while(sem_wait(sem)==-1);`. De este modo, si un manejador interrumpe la llamada, la función devolverá error (-1) y se volverá a realizar la espera.

Ejercicio 9

La siguiente sería una posible solución. Ambos semáforos se inicializan a 0 en el código dado, por lo que cualquier `sem_wait` que se realize a cada uno de ellos por primera vez bloqueará el proceso hasta que el otro le de permiso a continuar (con `sem_post`).

En resumen, cuando el padre tiene que esperar algo del hijo, llama a `sem_wait(sem1)` y cuando le tiene que dar permiso al hijo a continuar, llama a `sem_post(sem2)`. El hijo incrementa (`sem_post`) el semáforo `sem1` cuando ya da permiso al padre a continuar, y decrementa (`sem_wait`) el semáforo `sem2` cuando espera que el padre haga algo.

```
32      if (pid == 0) {
33          printf("1\n");
34          sem_post(sem1); /*up para que el padre imprima 2*/
35          sem_wait(sem2); /*esperar a que el padre imprima 2*/
36          printf("3\n");
37          sem_post(sem1); /*up para que el padre imprima 4*/
38
39          sem_close(sem1);
40          sem_close(sem2);
41      }else {
42          sem_wait(sem1); /*esperar a que el hijo imprima 1*/
43          printf("2\n");
44          sem_post(sem2); /*up para que el hijo imprima 3*/
45          sem_wait(sem1); /*esperar a que el hijo imprima 3*/
46          printf("4\n");
47
48          sem_close(sem1);
49          sem_close(sem2);
50          sem_unlink(SEM_NAME_A);
51          sem_unlink(SEM_NAME_B);
52          wait(NULL);
53          exit(EXIT_SUCCESS);
54      }
```

Ejercicio 10

Para mencionar los procesos, usaremos la notación del esquema que aparece en el enunciado: P_1, P_2, ... , $P_{\texttt{NUM\_PROC}}$.

a) Creación de los procesos

La parte de creación de procesos se realiza en un bucle *for*. Como queremos que los procesos se lancen en cascada, en cada iteración del bucle, el proceso padre sale y el hijo se queda (una iteración más) para realizar una nueva llamada a `fork()`. De este manera conseguimos tener `NUM_PROC` procesos, cada uno hijo del anterior (excepto el primero).

Como los manejadores se heredan, las llamadas a `sigaction` las realizamos al inicio del todo. De esta forma todos los procesos tienen manejadores para las señales `SIGUSR1`, `SIGTERM`, `SIGALRM` y `SIGINT`. No obstante, queremos que la señal `SIGINT` solo la pueda recibir el proceso 1, por eso el proceso 2 la añade al *set* de señales a bloquear en `sigsuspend`. Este *set* es una variable de tipo `sigset_t`, por lo que los sucesores de P_2 (es decir, todos los procesos que no son P_1) la heredan.

b) y c) Bucle de los ciclos

Hemos hecho una función que contiene el bucle que cada proceso hace para los ciclos. En cada iteración, se hace una llamada a `sigsuspend`, donde el proceso se bloquea hasta que recibe una señal. El manejador modificará una de las variables globales (`got_end` o `got_sigusr1`), por lo que, dependiendo de cuál se reciba, se hará una de las dos posibles acciones: mandar `SIGUSR1` al siguiente proceso e imprimir una cadena por *stdout*, o mandar `SIGTERM` al siguiente proceso y salir del bucle.

Todos los procesos, al salir del bucle, cierran el semáforo, realizan un `wait(NULL)` –ya que, como mucho, cada proceso tiene solamente un hijo– y terminan.

d) Protección de zonas críticas

Para que los procesos no hagan esperas activas, utilizamos `sigsuspend` en cada iteración del bucle, con la cual los procesos se bloquean hasta que reciben una señal. Tenemos que evitar, por tanto, que las señales se reciban cuando el proceso no ha llegado todavía a `sigsuspend`. Para conseguir esto, antes de establecer los manejadores en el proceso inicial (y antes de la creación de cualquier otro proceso), bloqueamos estas cuatro señales, guardando la antigua máscara para después utilizarla en la llamada a `sigsuspend`, donde sí permitimos estas señales. Sin embargo, los procesos que no son P_1 no deben recibir `SIGINT`, por lo que en cuando se crea P_2, añadimos a la antigua máscara esta señal, de modo que será bloqueada incluso dentro de `sigsuspend` para todos los procesos distintos del primero.

Nos aseguramos de que en cada iteración del bucle de los ciclos se recibe únicamente una señal bloqueando dentro del manejador las señales `SIGUSR1`, `SIGALRM`, `SIGINT` y `SIGTERM`. Esto es porque `sigsuspend` no reestablece la máscara del proceso hasta que el manejador haya terminado.

Si no bloqueáramos estas señales, sería posible que, durante la ejecución de un manejador, se recibiera otra de ellas. Sin embargo, cada iteración del bucle solo trata una, y el proceso se bloquearía de nuevo en `sigsuspend` sin haber tratado una señal, la cual se estaría perdiendo.

e) Temporización

Para la temporización que se propone, tenemos que llamar a `alarm(10)` antes de que el primer proceso cree a su hijo. En general, los hijos no heredan alarmas, por lo que esta alarma solamente la recibirá el primer proceso. Además, tenemos que establecer un manejador para la señal `SIGALRM`, y darle en el manejador el mismo tratamiento que a `SIGINT`, ya que el resultado de recibir cualquiera de estas dos señales es el mismo.

f) Análisis de la ejecución

Dentro de cada ciclo los procesos imprimen desordenados. No obstante, los `printf` de ciclos distintos no se intercalan. No hay ninguna garantía de que vaya a ocurrir esto ya no que estamos imponiendo ninguna condición que obligue a un proceso a esperar a otro.

De hecho, cuando añadimos una espera aleatoria vemos que ahora sí se empiezan a intercalar mensajes de ciclos distintos. Con esta nueva ejecución confirmamos lo dicho de que no había garantía alguna del orden de ejecución (y por ende del orden en el que aparecen los mensajes).

g) Sincronización con semáforos

La sincronización con semáforos se encarga únicamente de que el orden de impresión por *stdout* sea el deseado.

Solamente es necesario emplear un semáforo binario de forma que cuando un proceso quiera mandar la señal `SIGUSR1` al siguiente e imprimir en *stdout*, tenga que esperar hasta que el proceso anterior haya terminado de escribir por pantalla.

Con el uso de este semáforo binario estamos protegiendo la sección crítica de mandar señal y escribir, asegurándonos que solo un proceso estará ejecutando esa parte, y evitando así que el siguiente proceso imprima antes que el que le manda `SIGUSR1`.