Sistemas Operativos. Práctica 2.

Pablo Cuesta Sierra y Álvaro Zamanillo Sáez

26 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`. Como el manejador no termina el proceso, éste continúa por la orden siguiente a la que ejecutó por última vez, es decir, vuelve a empezar el bucle, imprimiendo el `printf` del `while`.

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)

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, no se imprime el mensaje tras la espera.

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 for parent to print 2*/
35          sem_wait(sem2); /*wait for parent to print 2*/
36          printf("3\n");
37          sem_post(sem1); /*up for parent to print 4*/
38
39          sem_close(sem1);
40          sem_close(sem2);
41      }else {
42          sem_wait(sem1); /*wait for child to print 1*/
43          printf("2\n");
44          sem_post(sem2); /*up for parent to print 3*/
45          sem_wait(sem1); /*wait for child to print 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