

# Sistemas Operativos. Práctica 2.

Pablo Cuesta Sierra y Álvaro Zamanillo Sáez

27 de marzo de 2021

## Índice

Ejercicio 1 . . . . .	1
Ejercicio 2 . . . . .	1
Ejercicio 3 . . . . .	1
Ejercicio 4 . . . . .	2
Ejercicio 5 . . . . .	2
Ejercicio 6 . . . . .	2
Ejercicio 7 . . . . .	2
Ejercicio 8 . . . . .	2
Ejercicio 9 . . . . .	3
Ejercicio 10 . . . . .	4
a) Creación de los procesos . . . . .	4
b) y c) Bucle de los ciclos . . . . .	4
d) Protección de zonas críticas . . . . .	4
e) Temporización . . . . .	4
f) Análisis de la ejecución . . . . .	5
g) Sincronización con semáforos . . . . .	5

## 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 escribir 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 escribimos 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) 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 envía `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 después de que se llame a `sem_unlink`, tendrá que crearlo. Sin embargo, 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 realice 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:

$P_1, P_2, \dots, P_N$ , donde  $N = \text{NUM\_PROC}$ .

### a) Creación de los procesos

Para cada proceso, tenemos que asegurarnos de que se establece el manejador para las señales que puede recibir, y debe haber una variable (`sigset_t susp_mask` inicializada con `sigfillset(&susp_mask)`) que permita que se reciban estas señales durante `sigsuspend`. En resumen:

- Señales que puede recibir  $P_1$ : `SIGUSR1`, `SIGINT` y `SIGALRM`
- Señales que pueden recibir los otros procesos: `SIGUSR1`, `SIGTERM`.

Como los manejadores se heredan y las variables (en este caso `susp_mask`) también, podemos hacer lo siguiente:

- Antes de crear un hijo, el  $P_1$  establece manejador para `SIGUSR1` y la quita de `susp_mask`.
- Tras crear  $P_2$ , en  $P_1$  se establece el manejador para las señales que solamente va a recibir este (`SIGALRM` y `SIGINT`), y las quitamos de `susp_mask`. Inmediatamente después, se manda la primera señal al proceso  $P_2$  y se imprime el primer mensaje por `stdout`.
- En  $P_2$ , antes de crear un hijo, se establece el manejador para `SIGTERM` y se quita esta señal de `susp_mask`. Tras esto, hay un bucle *for*, donde cada proceso crea un hijo y sale del bucle hasta que hay un total de `NUM_PROC` procesos.

### 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

– pregunta

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** (en este primer proceso), 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 print 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).

*Nota: Se puede establecer alguna garantía, por ejemplo: el proceso número  $j \in \{0, 1, \dots, NUM\_PROC - 1\}$  debe imprimir su mensaje del ciclo  $k$  antes de que el proceso número  $(j + 1) \% NUM\_PROC$  imprima el mensaje del ciclo  $(k + 1)$ . Pero como ya se ha comentado, nada asegura que los mensajes se impriman en el orden que corresponde.*

## 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.