Luis Cárabe Fernández-Pedraza (luis.carabe@estudiante.uam.es)
Emilio Cuesta Fernández (emilio.cuesta@estudiante.uam.es)

SISTEMAS OPERATIVOS: PRÁCTICA 3.

Luis Cárabe Fernández-Pedraza
Emilio Cuesta Fernández

Luis Cárabe Fernández-Pedraza (luis.carabe@estudiante.uam.es)
Emilio Cuesta Fernández (emilio.cuesta@estudiante.uam.es)

*Ejercicio 2:

El programa en cuestión consiste en un proceso padre que ha de imprimir los datos de alta de un nuevo cliente (nombre e id) que previamente han sido solicitados por pantalla por sus hijos. Es necesario pasarle como único argumento el número de clientes que se van a introducir por pantalla.

Para ello se utiliza una región de memoria compartida. Esta región se crea utilizando la función “shmget”. Posteriormente se consigue acceso a ella utilizando la función “shmatt”. Si se realizan estas dos funciones desde el proceso padre, todos los hijos heredaran la variable con el valor del retorno de “shmatt” que permite acceder a la memoria. Por ello, no es necesario volver a llamar a estas funciones dentro del código de los procesos hijo.

Una vez realizado el fork(), los distintos procesos hijos que se han ido creando solicitan los datos por pantalla después de que haya transcurrido un tiempo aleatorio y, antes de finalizar, envían una señal SIGUSR1 a su padre mediante kill. Para que este tiempo se realmente aleatorio, se utiliza como semilla el pid de cada proceso, que es único. Los tiempos obtenidos se han reducido módulo 30 para que la espera no se demorara excesivamente.

Por su parte, el proceso padre imprime los datos conforme los va leyendo. Estos datos se imprimen desde el manejador de la señal SIGUSR1, previamente definido. (Se ha utilizado una variable global como puntero a la estructura de memoria compartida para que el manejador pudiera tener acceso a los datos).

El planteamiento de este ejercicio falla en varios puntos:

- En primer lugar, permite el acceso a memoria compartida de los dos procesos al mismo tiempo. Esto, en general, puede generar problemas graves. Por ejemplo, uno de los procesos que escriben los datos escribe antes de que el lector haya leído la información anterior, este se habrá perdido el set de información sobreescrito. Tal y como está planteado este ejercicio, esto es poco probable, pero no se está garantizando la exclusión mutua.

- Por otro lado, hay un problema de coordinación en los procesos. Muchos solicitan nombres de clientes al mismo tiempo (para las personas). La persona que introduce los nombres no es capaz de introducirlos al ritmo que solicita el programa. Como no se utilizan medidas de coordinación que establezcan un orden determinado, el resultado es un poco caótico, como se muestra en la imagen.

```
emi@EmilioPC:~/soper/practica3$ ./ejercicio2 5
ALTA DE UN NUEVO CLIENTE (Proceso 10990). Introduzca su nombre:
Emilio
Nombre de usuario: Emilio, ID: 1
ALTA DE UN NUEVO CLIENTE (Proceso 10987). Introduzca su nombre:
Luis
Nombre de usuario: Luis, ID: 2
ALTA DE UN NUEVO CLIENTE (Proceso 10991). Introduzca su nombre:
ALTA DE UN NUEVO CLIENTE (Proceso 10989). Introduzca su nombre:
Juan
Nombre de usuario: Juan, ID: 3
ALTA DE UN NUEVO CLIENTE (Proceso 10988). Introduzca su nombre:
Carlos
Nombre de usuario: Carlos, ID: 4
Román
Nombre de usuario: Román, ID: 5
```

El resultado no tiene por qué ser siempre así, depende en gran medida de los tiempos aleatorios generados.

Para solucionar este problema, se han utilizado los conocimientos de los que disponíamos de otras prácticas y proponemos el programa presente en el fichero "**ejercicio2b.c".** Esta solución no utiliza semáforos porque aún no los habíamos visto, pero sirviéndose de la función pause() hace que todo funcione de forma más secuencial y ordenada. Sin embargo, como el proceso padre se queda bloqueado esperando a recibir la señal de un hijo, no va generando más hijos. Esto resta velocidad al programa y en realidad no funciona exactamente igual. Una solución mejor se realiza utilizando semáforos, y hemos incluido este programa en el tester de la librería de semáforos para demostrar que funciona bien y así comprobar la eficacia de los semáforos sobre este mismo problema.
Se adjunta ahora imagen de la salida del programa de **ejercicio2b.c**

```
emi@EmilioPC:~/soper/practica3$ ./ejercicio2b 5
ALTA DE UN NUEVO CLIENTE (Proceso 11167). Introduzca su nombre:
Jose
Nombre de usuario: Jose, ID: 1
ALTA DE UN NUEVO CLIENTE (Proceso 11169). Introduzca su nombre:
Juan
Nombre de usuario: Juan, ID: 2
ALTA DE UN NUEVO CLIENTE (Proceso 11170). Introduzca su nombre:
Luis
Nombre de usuario: Luis, ID: 3
ALTA DE UN NUEVO CLIENTE (Proceso 11171). Introduzca su nombre:
Ramon
Nombre de usuario: Ramon, ID: 4
ALTA DE UN NUEVO CLIENTE (Proceso 11173). Introduzca su nombre:
Jacinto
Nombre de usuario: Jacinto, ID: 5
```

Luis Cárabe Fernández-Pedraza (luis.carabe@estudiante.uam.es)
Emilio Cuesta Fernández (emilio.cuesta@estudiante.uam.es)

*Ejercicio 4:

En este ejercicio, simplemente hemos tenido que codificar la librería semaforos.h, al darnos las funciones a implementar junto con su descripción, no hemos tenido ningún problema, ya que nos limitamos a usar las funciones de las librerías <sys/ipc.h>, <sys/sem.h> y <sys/types.h> para ayudarnos a realizar bien su cometido. Con lo cual, no hemos tenido que realizar ninguna decisión de diseño, simplemente a codificar para obtener el resultado esperado.

*Ejercicio 5:

En este ejercicio, hemos realizado un programa de prueba para comprobar la correcta funcionalidad de la librería de semáforos implementada en el ejercicio 4. Esta comprobación se basa en dos partes, primero comprobamos todas las funciones llamándolas una a una y comprobando su retorno, así como las modificaciones internas del semáforo realizadas en dichas funciones. Como queríamos ver un ejemplo más práctico de uso, utilizamos el ejercicio 2, que en un principio funcionaba mal y tras hacer una enmienda con pause conseguíamos que funcionara un poco mejor, con sus inconvenientes, como ya vimos en esta misma memoria. Pero para dar una mejor solución a ese problema y asegurar que nuestras funciones de semaforos.h son útiles, sustituimos el sleep de los procesos hijos por una llamada a down, y tras pedir datos, modificar el contador y enviar la señal SIGUSR1 llamamos a up, lo que soluciona los problemas mencionados en el Ejercicio 2, como la posibilidad de que el contador no se incremente correctamente o que dos procesos intenten hacer un scanf a la vez.

A continuación, mostramos una captura de ejemplo:

```
luis@luis-X555LJ:~/Escritorio/SOPER/practica3$ ./ejercicio5
No se han producido errores en la primera prueba de la librería "semáforos.h"
Introduce el número de procesos hijos a generar en el ejercicio2 realizado con semáforos:
3
ALTA DE UN NUEVO CLIENTE (Proceso 4762). Introduzca su nombre:
Emilio
Nombre de usuario: Emilio, ID: 1
ALTA DE UN NUEVO CLIENTE (Proceso 4763). Introduzca su nombre:
Luis
Nombre de usuario: Luis, ID: 2
ALTA DE UN NUEVO CLIENTE (Proceso 4764). Introduzca su nombre:
Paco
Nombre de usuario: Paco, ID: 3
No se han producido errores en la segunda prueba de la librería "semáforos.h"
```

Luis Cárabe Fernández-Pedraza (luis.carabe@estudiante.uam.es)
Emilio Cuesta Fernández (emilio.cuesta@estudiante.uam.es)

*Ejercicio 6:

Este ejercicio consiste en implementar el clásico problema del productor y consumidor. Para ello, es necesario utilizar tanto memoria compartida como semáforos.

Más en concreto, en este caso un proceso ha de escribir en la memoria compartida las letras del alfabeto en orden (productor) mientras que otro proceso ha de leerlas también en orden e imprimir por pantalla la letra que ha leído (consumidor). Como buffer de memoria compartida se ha decidido definir una estructura que contenga un array de 26 caracteres. Se toma 26 porque es el número de letras que hay en nuestro alfabeto descontando la Ñ.
Dada nuestra implementación, será necesario coordinar los procesos para que no accedan al fragmento de memoria compartida la mismo tiempo (Semáforo 1 en nuestro código) y además para que el proceso consumidor no intente leer en posiciones del array de caracteres que aún no han sido escritas. (Semáforo 0 en nuestro código). Por tanto, nuestro problema queda resuelto mediante el uso de una región de memoria compartida y 2 semáforos.

Tanto los semáforos como la memoria se crean e inicializan en el proceso padre del programa. Como el productor y el consumidor (hijos) tienen una copia de las variables de su proceso padre, disponen de los identificadores del set de semáforos y de la memoria compartida. Por ello, no se llama ni a "shmget", ni a "shmatt" ni a sus equivalentes para semáforos en los procesos hijos.

Para modularizar el código, se han diseñado dos funciones "*productor(shm* buffer, int semid)"* y *"consumidor(shm* buffer, int semid)"* que realizan las las labores del productor y del consumidor respectivamente. En ellas, *buffer* es el puntero a la región de memoria compartida y *semid* el identificador del set de semáforos que coordinan el programa.

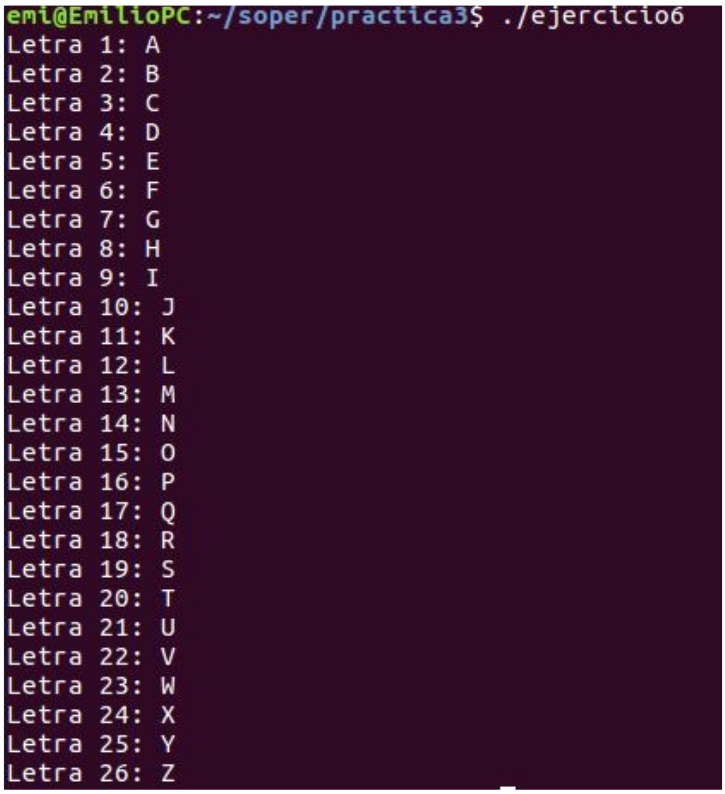
Desde el punto de vista del funcionamiento del programa, el semáforo 0 se inicializa a 0 y el semáforo 1 se inicializa a 1.

El *productor* realizará, en cada iteración de un bucle, un *Down()* del semáforo 1, que regula el acceso a memoria compartida. Posteriormente, escribirá en la posición correspondiente del array la letra pertinente y acto seguido dejará de utilizar la memoria compartida y realizará un *Up()* del semáforo 1. Justo después, incrementará el semáforo 0 mediante un *Up()*. De esta manera indica que se ha escrito algo. Mientras este semáforo tenga un valor mayor que 0, habrá letras escritas por el productor que no hayan sido leídas por el consumidor.

El *consumidor* realizará, en cada iteración de su bucle, un *Down()* del semáforo 0 en primer lugar. Esto implica que si el productor aún no ha escrito nada o si todo lo que ha escrito ya ha

sido leído por el consumidor, el proceso que consume será bloqueado hasta que pueda leer algo más. Posteriormente, cuando ya se dispone de información que leer, el proceso realiza un *Down()* del semáforo 1 para solicitar acceder a memoria compartida. Una vez en ella, se imprime la información por pantalla y, para terminar, se realiza un *Up()* del semáforo 1 para indicar que se ha dejado de utilizar la memoria compartida.

Una vez explicado el funcionamiento de nuestro programa, se adjunta una captura de su ejecución:



Nota para todos los ejercicios: si el programa aborta inesperadamente, es necesario ejecutar ipcrm.