Sistemas Operativos
Práctica 3

Blanca Martín · Fernando Villar
Grupo 2201 · Pareja 10

04/04/2017

Índice

1. Introducción

En este documento se discutirán los ejercicios propuestos en la tercera práctica de Sistemas Operativos (grupo 2201), que orbitan alrededor de los conceptos relacionados con memoria compartida y semáforos: creación, uso, control y liberación.

2. Listado de código fuente

- ejercicio2.c
- semaforos.c
- semaforos.h
- ejercicio5.c
- ejercicio6.c

Todo ello regulado por el archivo "Makefile", que genera automáticamente los archivos .o y los ejecutables para cada ejercicio, y que cuenta con la posibilidad de efectuar un comando **make clean** para eliminar todos los archivos generados. Los ejercicios 5 y 6 toman también el objeto semaforos.o para implementar dicha librería de semáforos. La documentación completa del código fuente se presenta en el archivo PDF aportado en este mismo directorio, `"Documentación código fuente (Doxygen).pdf"`.

3. Ejercicio 2

El fallo de planteamiento se resume en el hecho de que, según cómo decida el planificador de procesos actuar, existe la posibilidad de que se produzcan cambios inesperados en la información que está en memoria compartida. Por ejemplo, podría suceder la siguiente situación:

- Tras 3 segundos, el primer proceso hijo solicita que se introduzca un nombre por pantalla.
- Tras 4 segundos, el segundo proceso hijo solicita que se introduzca un nombre por pantalla.
- Tras 5 segundos, el usuario introduce el primer nombre, e inmediatamente después, introduce el segundo.
- Antes de leer de la memoria compartida, se guarda en ella el primer nombre y el segundo justo a continuación, de manera que de memoria compartida se lee el segundo nombre cuando nosotros esperaríamos leer el primero.

Esta circunstancia puede también suceder con el ID, y en general con cualquier información que resida en la memoria compartida. A pesar de todo, la probabilidad de que esto suceda en circunstancias reales es ínfima, porque el tiempo que tarda el usuario en introducir el segundo nombre es suficiente para que el planificador ejecute todo lo que tenga que ejecutar del proceso correspondiente al primer nombre.

```
sliezzan@debian-sliezzan:~/Documents/OperatingSystems/SOPER/03_P3/Desarrollo$ ./ej_2_3/ejercicio2 5

Introduzca el nombre de un cliente:

Introduzca el nombre de un cliente:
A
Introduzca el nombre de un cliente:
lber
Introduzca el nombre de un cliente:
to
Nombre de usuario: Alberto      ID: 1
Usuario2
Nombre de usuario: Usuario2     ID: 2

Introduzca el nombre de un cliente:
Juan
Nombre de usuario: Juan ID: 3
Luis
Nombre de usuario: Luis ID: 4
Usuario5
Nombre de usuario: Usuario5     ID: 5
```

Figura 1: Ejecución del ejercicio 2

4. Ejercicio 4

A la realización de este ejercicio corresponden los archivos **semaforos.c** y **semaforos.h**, donde se implementa la biblioteca de semáforos requerida. Cabe mencionar que para mantener el hecho de que las funciones **up** y **down** sean atómicas, se han creado objetos de tipo **union semun** locales en cada función, de manera que diferentes planificaciones no alteren los valores de las variables sem_op, y que por tanto no alteren la manipulación de los semáforos.

Lo que sí que puede suceder, con **UpMultiple_Semaforo** y **DownMultiple_Semaforo**, es que entre medias de la actualización de los distintos semáforos con los que interactúan, otras funciones sean llamadas desde otro sitio y alteren los valores de dichos semáforos en mitad de la ejecución. Sin embargo, como suponemos en todo momento que los semáforos actúan con independencia, esta circunstancia no supone ningún problema.

5. Ejercicio 5

Este ejercicio corresponde a las pruebas sobre la biblioteca de semáforos creada en el apartado anterior para comprobar su correcto funcionamiento. En este caso, se realizan las siguientes acciones:

- Se crean dos arrays, uno correspondiente a los valores iniciales de los semáforos (**arg.array**, todos a 1) y otro en el que introduciremos los índices para implementar **UpMultiple_Semaforo** y **DownMultiple_Semaforo** (**active**).
- Se inicializan los semáforos, todos a 1.
- Se realiza un down al primer semáforo.
- Se realizan downs a todos los semáforos restantes (indicados por el array active).
- Se realiza un up al primer semáforo.
- Se realizan ups a todos los semáforos restantes (indicados por el array active).

En todo momento se ejecuta la función **semctl** para comprobar el estado de los semáforos.

```
sliezzan@debian-sliezzan:~/Documents/OperatingSystems/SOPER/03_P3/Desarrollo$ ./ej_5_6/ejercicio5
Creación correcta. ID 262152.
Inicialización correcta.
Down_Semaforo correcto (0).
Valor del semáforo 0: 0
Valor del semáforo 1: 1
Valor del semáforo 2: 1
Valor del semáforo 3: 1
Valor del semáforo 4: 1
DownMultiple_Semaforo correcto (1-4).
Valor del semáforo 0: 0
Valor del semáforo 1: 0
Valor del semáforo 2: 0
Valor del semáforo 3: 0
Valor del semáforo 4: 0
Up_Semaforo correcto (0).
Valor del semáforo 0: 1
Valor del semáforo 1: 0
Valor del semáforo 2: 0
Valor del semáforo 3: 0
Valor del semáforo 4: 0
UpMultiple_Semaforo correcto (1-4).
Valor del semáforo 0: 1
Valor del semáforo 1: 1
Valor del semáforo 2: 1
Valor del semáforo 3: 1
Valor del semáforo 4: 1
Borrado correcto.
```

Figura 2: Ejecución del ejercicio 5

6. Ejercicio 6

Para la implementación de la funcionalidad requerida en el ejercicio 6, vamos a tener en cuenta las siguientes circunstancias:

1. **Procesos implicados:**
   - Lanzaremos **3 procesos**, en concreto, desde el padre se crearán dos procesos hijo en paralelo.
   - El padre será el encargado de regular la actividad del programa.
   - El primer hijo hará de productor.
   - El segundo hijo hará de consumidor.
2. **Memoria compartida:** Se creará una región de memoria compartida para los tres procesos en la que se guardará una estructura con:
   - Un buffer de caracteres de tamaño 26 para las letras del alfabeto.
   - El valor del lugar del buffer donde hay que leer/escribir (número de caracteres escritos).
   - Un valor que regula la terminación de los procesos productor y consumidor.
3. **Semáforos:** Se creará un array de tres semáforos para:
   - Controlar el acceso a lectura/escritura del buffer.
   - Controlar que no se lea de un buffer vacío.
   - Controlar que no se escriba en un buffer lleno.

Por tanto, el funcionamiento del programa será, básicamente, el siguiente:

- **Productor:** Si tiene acceso a lectura/escritura y el buffer no está lleno, escribirá en el buffer la siguiente letra según la posición donde se encuentre el valor que guardamos en la estructura (en la posición 0 se imprimirá una 'a', en la posición 1 se imprimirá una 'b', etc.), moverá una posición dicho valor hacia la derecha y registrará con el semáforo correspondiente que en el buffer cabe una posición menos (o lo que es lo mismo, que se puede leer de una más).
- **Consumidor:** Si tiene acceso a lectura/escritura y el buffer no está vacío, disminuirá en una unidad la posición del valor que guardamos en la estructura y leerá de esa posición. Posteriormente, reflejará los cambios en la capacidad de lectura/escritura en el semáforo correspondiente.

Y, en consecuencia, el primer semáforo se inicializará a 1 para que sólo un proceso acceda a la sección crítica donde se efectúa la lectura/escritura; el segundo semáforo se inicializará a 0 para que el consumidor no lea si el buffer está vacío; y el tercer semáforo se inicializará a 26 para que, después de escribir 26 caracteres, no se pueda escribir más. La disposición de las funciones **Up** y **Down** se pueden observar en el código y regulan exactamente lo que acabamos de comentar.

Cuando pasa un tiempo determinado (en nuestro código lo hemos implementado con un **sleep(3)**), el padre cambia el segundo valor de la estructura, de manera que la siguiente iteración de los bucles de productor y consumidor no se produce, salen de las funciones y sus procesos finalizan. El padre espera a la terminación de los dos hijos y libera todo lo reservado en la primera parte del programa, para finalmente terminar él también su ejecución.

NOTA: Se ha colocado un pequeño delay tras cada iteración de los bucles de productor y consumidor para facilitar la comprensión del funcionamiento del programa. Cabe mencionar que este delay es el mismo en ambos bucles y por tanto se tiende a que por cada iteración de productor haya otra de consumidor; sin embargo, en la práctica, si bien de una forma lenta, se puede apreciar que normalmente el productor se ejecuta más veces que el consumidor. Este hecho lo atribuimos a que, probablemente, las políticas de planificación de nuestros sistemas operativos dan prioridad a los procesos más antiguos, y en nuestro caso, el primer hijo es el productor, por lo que se ejecutaría con más frecuencia.