24/04/2018

SISTEMAS OPERATIVOS
PRÁCTICA 3

Mihai Blidaru

Lucía Fuentes Villodres

Grupo 2262

Ejercicio 2:

1. Explica en qué falla el planteamiento de este ejercicio:

En este ejercicio no se está proponiendo el empleo de semáforos, pero hay varios procesos que quieren realizar operaciones de lectura y escritura en la estructura que está siendo usada como recurso compartido, por lo tanto se puede producir interbloqueo de procesos.

2. Implementa un mecanismo (ejercicio2_solved.c) para solucionar este problema basado en tu conocimiento de la asignatura.

Hemos planteado una solución añadiendo un semáforo, de tal forma que cuando un proceso hijo quiera escribir en la estructura haga un down, y hasta que el proceso padre no lea el nombre que el hijo acaba de registrar, no se vuelve a subir el valor del semáforo. De esta forma evitamos que dos procesos hijos escriban a la vez en la estructura.

```
ID zona de memoria compartida: 589824
[Hijo 1]Intoduce nombre del cliente: a
Nombre usuario = a
ID = 1
[Hijo 0]Intoduce nombre del cliente: b
Nombre usuario = b
ID = 2
```

Ejercicio 3:

Elecciones de diseño:

Para este ejercicio hemos supuesto que hay un límite máximo de productos que el productor va a producir y que el consumidor va a consumir. Está definido por la macro LIMITE.

Para abordar el problema, necesitamos tres semáforos:

- Un mutex, que va a ser empleado tanto por el productor como por el consumidor y se utiliza para asegurarnos de que los dos procesos no acceden simultáneamente al buffer que almacena los ítems.
- Un fillCount, que está inicializado a 0 porque registra el número de elementos que hay almacenados en el buffer. El consumidor lo baja cuando consume un producto y el productor lo sube cuando produce, de esta forma nos aseguramos que el consumidor no consume un producto si el buffer está vacío.
- Un emptyCount, inicializado al tamaño máximo del buffer, que contabiliza el número de espacios disponibles para guardar ítems. El consumidor lo sube cuando consume y el productor lo baja cuando produce, para asegurarnos de que el productor no produce más ítems de los que caben en el buffer.

El productor va a producir ítems hasta que llegue al máximo y el consumidor va a consumirlos. Debido a que no podemos predecir la cola de los semáforos porque depende de la gestión de procesos del sistema, algunas veces el productor va a producir una serie de productos seguidos que luego va a consumir el consumidor, o se van a ir alternando. En esta traza de ejecución se puede apreciar:

```
+ [K]
Consumido item: K
+ [L]
Consumido item: L
+ [M]
Consumido item: M
+ [N]
Consumido item: B
+ [O]
Consumido item: O
+ [P]
+ [Q]
Consumido item: P
+ [R]
Consumido item: Q
+ [S]
Consumido item: R
+ [T]
Consumido item: T
```

Ejercicio 4:

Podemos comprobar efectivamente que la ejecución del programa es la planteada por el enunciado, imprimimos el status de salida de los hilos y miramos que en el fichero de salida hay una serie de números menores de 1000 separados por espacios:

```
Thread 1 exited with status 0
Thread 2 exited with status 0
```

fichero | bash - "luciafuentes- | ejercicio4.c

```
1  125 606 636 717 950 532 107 62 129 239 853 474 520 904 43 288 715 8 514 888 670 451 851 56 249
```

Ejercicio 5:

Decisiones de diseño:

Antes de entrar en la ejecución de cadena_montaje.c, hay un archivo generator.c que sirve para generar el fichero de entrada del programa, con una serie de caracteres aleatorios con el tamaño que pasemos como argumento. Si queremos un fichero de entrada f1.txt con 100 caracteres aleatorios solo tendremos que ejecutar ./generator f1.txt 100.

Una vez tenemos el fichero podemos ejecutar cadena_montaje.c. Como bien dice en el enunciado, tenemos 3 procesos A, B y C. El único proceso que se va a llamar desde el programa principal es el proceso A.

El proceso A va a lanzar el proceso B, y va a enviar mensajes de longitud aleatoria (máximo 2048 bytes), posteriormente va a cerrar el fichero de lectura y esperar que el proceso B acabe.

El proceso B va a lanzar al proceso C, y va a comenzar a leer los mensajes que envía el proceso A. Va a procesar cada mensaje, llamando a la función procesar que se encarga de la conversión de las letras del abecedario, y posteriormente va a enviar la información en forma de mensajes al proceso C. Se pondrá a la espera a que el proceso C termine.

El proceso C se va a encargar de leer los mensajes de B y volcarlos en el fichero que hayamos pasado como argumento. Cuando termina de leer todos los mensajes terminará, con él el proceso B y el proceso A en el main. Finalmente comprobamos la salida con la función comprobar_salida, que imprimirá el mensaje de “Salida correcta” si todo ha ido bien.

```
luciafuentes:~/workspace/practica3 $ ./generator f1.txt 100
luciafuentes:~/workspace/practica3 $ ./cadena_montaje f1.txt f2.txt
Proceso C terminado correctamente con codigo 0
Proceso B terminado correctamente con codigo 0
Proceso A terminado correctamente con codigo 0
Salida correcta
```

f1.txt

```
1  nWlrBbmQBhCDarzOwKkYHIDdqSCDX
```

f2.txt

```
1  oXmsCcnRCiDEbsaPxLlZIJEerTDEY
```