

Solución al problema del productor-consumidor con semáforos

SALVIA SISA CORTÉS, EZEQUIEL SOTO SEOANE

Sistemas Operativos II

Grupo 05

{salvia.sisa, ezequiel.soto}@rai.usc.es

I. INTRODUCCIÓN

En este informe se tratará de resolver el problema del productor-consumidor con espera activa mediante el uso de **semáforos**.

El problema del **productor-consumidor**, ya comentado en el primer informe, se puede resolver utilizando **semáforos**. Un semáforo, tal y como lo propuso *Dijkstra*, es una **variable entera** que contabiliza el número de señales de wakeup y las almacena. Así, un semáforo tiene **dos operaciones** que aseguran que el proceso de comprobar el valor del número de elementos del buffer, modificarlo y dormir si es necesario se ejecuten de forma **atómica**:

- **down**: si el valor del semáforo es mayor que 0, lo **disminuye** y permite al proceso continuar; y si es 0, el proceso se va a **dormir** sin completar la operación down.
- **up**: **incrementa** el valor del semáforo y, si hay un **proceso inactivo** en ese semáforo, le permite acabar su operación down

Así, el uso de semáforos en el problema del búfer limitado nos permite evitar la pérdida de llamadas a wakeup, lo que previene varios problemas, como que ambos procesos se queden dormidos para siempre.

II. IMPLEMENTACIÓN

Para resolver el problema del productor-consumidor con **semáforos** necesitamos tres de ellos: uno para contabilizar el número de ranuras **llenas** (llenadas), otro para contabilizar las ranuras **vacías** (vacías), y otro para asegurar la **exclusión mutua** (mutex). En C, estos son declarados como punteros a estructuras de tipo `sem_t`, de la librería `semaphore.h`.

Al igual que se comentó en el anterior informe, también necesitamos **dos programas**, uno para el productor y otro para el consumidor, llamados `prod_sem.c` y `cons_sem.c`, además de una zona de memoria compartida en la que se almacena el búfer (y que se crea con `mmap()`). Cabe destacar que tanto el productor como el consumidor tienen sus propias funciones para añadir y extraer elementos del búfer así como para crearlos y consumirlos, pero no se comentarán en este informe para evitar redundancia innecesaria ya que no hubo cambios en la implementación de las mismas con respecto a la comentada en el informe anterior.

A. Productor

Para la implementación del productor, primero se **eliminan** los semáforos del kernel con `sem_unlink()` para una mayor seguridad. Después se **crean e inicializan** con `sem_open()`: el semáforo vacías a N, el llenas a 0 y el mutex a 1.

En el bucle principal, que se ejecuta un **número fijo** de veces (`MAX_ITER`), primero se llama a `sleep` con un número entre 0 y 3 para forzar situaciones en las que el productor vaya a **distinta velocidad** que el consumidor y así el búfer se pueda llenar o vaciar. Después, el productor produce un ítem y ejecuta dos operaciones down (en vacías y en mutex), mediante la función `sem_wait()` para decrementar la cuenta de las ranuras vacías, y para indicar que va a **entrar en la región crítica**. Seguidamente, inserta el elemento producido y ejecuta un up sobre mutex (para volver a permitir acceso a la región crítica a otros procesos) y sobre llenas (para incrementarle una unidad, puesto que se insertó un elemento).

Por último, se imprime el búfer final para poder analizarlo y se liberan recursos (la región de memoria, los semáforos...).

B. Consumidor

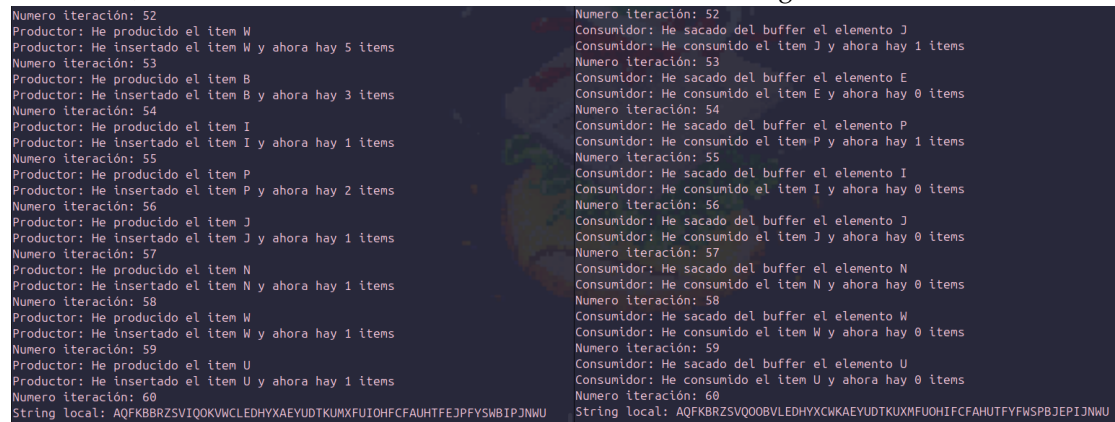
La implementación del consumidor es análoga a la del productor, con la diferencia de que el consumidor **no crea los semáforos** (ya que esta función la realiza el productor), sino que simplemente los abre. Además, por razones evidentes, el consumidor tampoco elimina los semáforos antes de abrirlos, ya que no tendría sentido.

En el **bucle principal** del consumidor, además del sleep aleatorio, nos encontramos con que primero este realiza un `sem_wait()` (es decir, un down) a llenas, para **decrementarle** una unidad, ya que se va a extraer un elemento del búfer, y a mutex, para indicar que va a acceder a la región crítica. Seguidamente **saca el elemento** y ejecuta un `sem_post()` (es decir, up) sobre mutex, para indicar que ya acabó de **acceder a la región crítica**, y sobre vacías, para sumarle una unidad, ya que ahora hay una ranura más vacía.

Por último, cuando el bucle termina, y al igual que en el productor, se imprime el resultado por pantalla y se **liberan recursos**.

III. EJECUCIÓN

Para ejecutar el programa, se compilaron con **gcc** y se ejecutaron ambos en **terminales diferentes**, con **60 iteraciones** como máximo, obteniéndose los siguientes **resultados**:



```
Numero iteración: 52
Productor: He producido el item W
Productor: He insertado el item W y ahora hay 5 items
Numero iteración: 53
Productor: He producido el item B
Productor: He insertado el item B y ahora hay 3 items
Numero iteración: 54
Productor: He producido el item I
Productor: He insertado el item I y ahora hay 1 items
Numero iteración: 55
Productor: He producido el item P
Productor: He insertado el item P y ahora hay 2 items
Numero iteración: 56
Productor: He producido el item J
Productor: He insertado el item J y ahora hay 1 items
Numero iteración: 57
Productor: He producido el item N
Productor: He insertado el item N y ahora hay 1 items
Numero iteración: 58
Productor: He producido el item W
Productor: He insertado el item W y ahora hay 1 items
Numero iteración: 59
Productor: He producido el item U
Productor: He insertado el item U y ahora hay 1 items
Numero iteración: 60
String local: AQFKBBRZSVIQOKVWCLEDHYXAEYUDTKUMXFUIOHFCFAUHTFEJPFYSWBIPJNWU

Numero iteración: 52
Consumidor: He sacado del buffer el elemento J
Consumidor: He consumido el item J y ahora hay 1 items
Numero iteración: 53
Consumidor: He sacado del buffer el elemento E
Consumidor: He consumido el item E y ahora hay 0 items
Numero iteración: 54
Consumidor: He sacado del buffer el elemento P
Consumidor: He consumido el item P y ahora hay 1 items
Numero iteración: 55
Consumidor: He sacado del buffer el elemento I
Consumidor: He consumido el item I y ahora hay 0 items
Numero iteración: 56
Consumidor: He sacado del buffer el elemento J
Consumidor: He consumido el item J y ahora hay 0 items
Numero iteración: 57
Consumidor: He sacado del buffer el elemento N
Consumidor: He consumido el item N y ahora hay 0 items
Numero iteración: 58
Consumidor: He sacado del buffer el elemento W
Consumidor: He consumido el item W y ahora hay 0 items
Numero iteración: 59
Consumidor: He sacado del buffer el elemento U
Consumidor: He consumido el item U y ahora hay 0 items
Numero iteración: 60
String local: AQFKBRZSVQOQBVLEDHYXCWKAIEYUDTKUMXFUOHIFCFAHUTFYFWSBPJEPIJNWU
```

Ambos programas imprimen su **string local**, que contiene todos los elementos producidos/consumidos, con el objetivo de identificar carreras críticas. A continuación se muestran las strings locales del productor y del consumidor:

```
Productor: AQFKBBRZSVIQOKVWCLEDHYXAEYUDTKUMXFUIOHFCFAUHTFEJPFYSWBIPJNWU
Consumidor: AQFKBRZSVQOQBVLEDHYXCWKAIEYUDTKUMXFUOHIFCFAHUTFYFWSBPJEPIJNWU
```

IV. CONCLUSIONES

Como se puede observar en el apartado anterior, **el uso de semáforos no garantiza la exclusión mutua**, por lo que se siguen produciendo carreras críticas pero con **menor frecuencia**. Esto se ve reflejado en las **strings locales** tanto del productor como del consumidor, ya que **no son consistentes** entre si: por ejemplo, el productor introduce el caracter B en la iteración 5, seguido de una R, pero el consumidor consume dos veces esa B, en sus iteraciones 5 y 6. Esto es una clara muestra de que ocurrió una carrera crítica y no se actualizó correctamente el valor de llenas, por lo que el consumidor extrajo dos veces el mismo ítem. Aún así, la solución de **implementar semáforos es preferible** frente al simple uso de espera activa y una variable compartida, ya que se reduce el número de carreras críticas considerablemente.