Carreras críticas en el problema del productor-consumidor

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 el problema del productor-consumidor con **espera activa** mediante el uso de dos programas (`prod.c` y `cons.c`).

El problema del **productor-consumidor** (o del búfer limitado) consiste en **dos procesos** (productor y consumidor) que comparten un **búfer de tamaño fijo**. Básicamente, el productor se dedica a colocar elementos en el búfer mientras que el consumidor los retira del mismo. Cuando uno de los dos **no puede realizar su función** (colocar o retirar), se va a **dormir** (con un `sleep`) y **espera a que el otro proceso lo despierte** (con un `wakeup`). Para llevar un conteo de los elementos que hay en el búfer, ambos procesos comparten la variable cuenta, que es actualizada por los procesos según introduzcan o saquen elementos del búfer.

El **problema** ocurre cuando la variable cuenta no es actualizada correctamente, ya que al no tener el acceso restringido, es susceptible a **carreras críticas**, lo que acaba ocasionando que tanto el productor como el consumidor se queden dormidos para siempre.

II. Implementación

Para la implementación, como se comentó anteriormente, se crearon **dos programas**, uno para el productor y otro para el consumidor, llamados `prod.c` y `cons.c`. Ambos procesos **comparten una región de memoria** (mediante el uso de la llamada al sistema `mmap()`), en la que se almacena una **estructura** `datos_compartidos` que contiene dos elementos: el **búfer de caracteres** `buffer[N]` y el **número de elementos** que contiene el búfer `num_elementos`.

A. Productor

El productor realiza **dos funciones principales**:

- `produce_item()`: **genera una letra mayúscula aleatoria** y la añade a su **string local** (que es una cadena de texto en la que se van almacenando todos los items creados).
- `insert_item()`: **coloca el item** (char) generado por la función anterior **en el búfer compartido**.

Dado que la implementación elegida fue la **espera activa**, el productor produce items y comprueba constantemente si el **número de elementos** actual del búfer es igual al **tamaño máximo** del mismo. En caso positivo, como no puede insertar más elementos en el búfer, simplemente **se queda esperando** hasta que tenga espacio para insertar uno o más caracteres.

B. Consumidor

El consumidor realiza **dos acciones** complementarias al productor:

- `remove_item()`: **retira el caracter de la cima del búfer**, ya que este se comporta como una estructura **LIFO**.
- `consume_item()`: **añade el caracter** obtenido del búfer a su **string local**.

Al igual que en el caso del productor, debido a que se usa la **espera activa**, el consumidor comprueba en cada iteración que haya **al menos un elemento** en el búfer; en caso contrario, **espera** hasta que el productor coloque algún ítem en el búfer.

III. Ejecución de los programas

Para poder **ver mejor las carreras críticas** que sufren tanto el productor como el consumidor, se introdujeron llamadas a la función `sleep` de forma que **la región crítica dure más tiempo**. Concretamente, fueron añadidos un `sleep(1)` en cada programa para poder visualizar mejor la ejecución de los mismos, además de un **sleep en el consumidor** para facilitar que ocurran carreras críticas.

Para la experimentación, se **compilaron** los programas con gcc y se ejecutaron en **dos terminales distintas**, de forma que comenzaran la ejecución de forma casi simultánea (el productor empezó un poco antes). Así, se obtuvo el **resultado** de ejecución que se muestra a continuación en la Figura 1, donde se resalta cómo ocurre la carrera crítica:

```
Productor: He producido el item A
Productor: He insertado el item A en la posición 0 y ahora hay 1 items
~Buffer: A
~String local: A

Productor: He producido el item U
Productor: He insertado el item U en la posición 0 y ahora hay 1 items
~Buffer: U
~String local: AU

Productor: He producido el item Y
Productor: He insertado el item Y en la posición 1 y ahora hay 2 items
~Buffer: UY
~String local: AUY

Productor: He producido el item D
Productor: He insertado el item D en la posición 1 y ahora hay 2 items
~Buffer: UD
~String local: AUYD

Productor: He producido el item O
Productor: He insertado el item O en la posición 2 y ahora hay 3 items
~Buffer: UDO
~String local: AUYDO

Productor: He producido el item T
Productor: He insertado el item T en la posición 2 y ahora hay 3 items
~Buffer: UDT
~String local: AUYDOT
```

```
Consumidor: He sacado del buffer el elemento U de la posicion 0
Consumidor: He consumido el item U y ahora hay 1 items
~Buffer: UY
~String local: U

Consumidor: He sacado del buffer el elemento D de la posicion 1
Consumidor: He consumido el item D y ahora hay 2 items
~Buffer: UDO
~String local: UD

Consumidor: He sacado del buffer el elemento T de la posicion 2
Consumidor: He consumido el item T y ahora hay 3 items
~Buffer: UDTB
~String local: UDT

Consumidor: He sacado del buffer el elemento Y de la posicion 3
Consumidor: He consumido el item Y y ahora hay 4 items
~Buffer: UDTYC
~String local: UDTY

Consumidor: He sacado del buffer el elemento W de la posicion 4
Consumidor: He consumido el item W y ahora hay 5 items
~Buffer: UDTYWN
~String local: UDTYW

Consumidor: He sacado del buffer el elemento X de la posicion 5
Consumidor: He consumido el item X y ahora hay 6 items
~Buffer: UDTYWXV
~String local: UDTYWX
```

Figura 1: *Ejecución del problema productor-consumidor*

Como podemos observar, el productor inserta el **item Y en la posición 1**, pero después, debido a una **carrera crítica** en la actualización de la variable compartida que almacena el número de elementos del buffer (`num_elementos`), el productor **saca de la posición 1 del buffer el elemento D, en vez de el Y**.

IV. Conclusión

Tal y como se destaca en el resultado de ejecución anterior, observamos que se producen **carreras críticas** debido a los **sleeps estratégicos** introducidos tanto en el productor como en el consumidor. Podemos concluir que estas ocurren debido a que el acceso a la variable(`num_elementos` (llamada `cuenta` en el *Tanembaum*) **no está restringido**, el productor y el consumidor la actualizan pisándose el uno al otro, ocasionando incongruencias en sus operaciones de insertar y extraer, que dependen directamente de esa variable y afectan al búfer compartido.

Por lo tanto, logramos nuestro objetivo de estudiar las carreras críticas que se producen en el problema del productor-consumidor