Sincronización de procesos con paso de mensajes

Luis Barba Cepedello, Pedro Vidal Villalba

Sistemas operativos II

{luis.barba, pedro.vidal.villalba}@rai.usc.es

30 de abril de 2024

I. Planteamiento del problema

El objetivo de esta práctica es resolver el problema del productor-consumidor haciendo uso de paso de mensajes. La solución a implementar debe ser la propuesta por Tanenbaum. Además, se deben tener en cuenta las siguientes consideraciones:

- El consumidor debe llenar el buffer del productor antes de empezar a consumir.
- Se debe asegurar que no quedan mensajes en las colas de mensajes, ni en la del consumidor ni en la del productor.
- Se debe implementar tanto una versión LIFO como una FIFO. En la primera, el consumidor debe consumir los items en orden Last-In-First-Out; en la segunda, en First-In-First-Out.
- Se debe llamar a la función `sleep` para forzar el llenado y vaciado del buffer y comprobar que que se sigue solucionando el problema del productor-consumidor.

Como consideración final, en la solución implementada es necesario ejecutar primero el programa correspondiente al productor. Esto es porque realiza *unlink* de los buzones, en caso de que existiesen aún como remanentes de ejecuciones previas. Si se ejecutase primero el consumidor, el productor haría unlink de los buzones ya linkeados por el consumidor, y ambos programas se bloquearían por no reconocerse entre ellos.

II. Implementación de la solución

En primer lugar, es importante destacar que son necesarios dos buzones de mensajes. El primer buzón, en el que el productor escribirá y del que el consumidor leerá, actuará como el buffer. El segundo será un buzón de control, cuya función será servir al consumidor de vía de comunicación con el productor para indicarle cuándo puede producir items. Tanenbaum llama a estos buffers `productor` y `consumidor`. En nuestro caso, serán `storage1` y `storage2`, respectivamente.

Antes de comenzar la producción y consumo de items, el consumidor debe llenar el buffer de control. Con esto se consigue que, aunque el consumidor tarde en consumir algún item, el productor pueda seguir produciendo sin esperar por él, teniendo un margen de 5 items a producir. Esto se puede lograr con una simple línea de código:

```
1  for (i = 0; i < N; i++) mq_send(storage2,
       &control, sizeof(char), 0);
```

Código 1: *Llenado inicial del buffer de control*

La implementación de las funciones `producer` y `consumer` sigue el esquema que plantea la solución de Tanenbaum. Como se puede observar en el código 2, el productor comienza por producir un item, que guarda en una variable. Seguidamente, llama a la función `insert_item`, la cual recibe un mensaje del buzón de control con `mq_receive`, para asegurarse de que hay espacio en el buffer, y seguidamente inserta el item producido en el mismo con `mq_send`.

```
1  for (i = 0; i < NUM_ITEMS; i++) {
2      item = produce_item(i);
3      insert_item(item, i);
4      producer_printf("Insertado el item "
           bold("%c")" en el buffer (numero
           de elemento: "bold("%2d")")\n",
           item, i);
5  }
```

Código 2: *Implementación del productor*

Por su parte, el consumidor comienza llamando a la función `remove_item`, la cual recibe un item del buffer para posteriormente enviar un mensaje al buzón de control, indicando al productor que puede producir otro item.

```
1  for (i = 0; i < NUM_ITEMS; i++) {
2      item = remove_item(&control, &
           position);
3      consume_item(item, position);
4  }
```

Código 3: *Implementación del consumidor*

De esta forma, se asegura que el productor solo produzca items cuando tenga espacio en el buffer, y se asegura también que el consumidor únicamente consuma cuando haya algún item en el buffer.

Finalmente, para asegurar que no quedan mensajes en los buzones al acabar los programas, se ha implementado la función `clear_buffer`, la cual se deshace de todos los mensajes residuales que pudiesen quedar. Además, imprime también el número de items restantes en el buzón que esté vaciando, para asegurar que está vacío.

III. Implementaciones LIFO y FIFO

El código listado anteriormente sigue una implementación LIFO. Para la implementación del buffer con FIFO, el único cambio necesario es la prioridad con la que se mandan los items al buffer.

- En la implementación LIFO, se llama a `send` con prioridad `i`, siendo `i` el número de iteración. De esta forma, el último item en entrar tendrá la prioridad más alta y será el primero en ser consumido, resultando en un comportamiento LIFO.
- Por otro lado, para implementar el buffer con FIFO, se puede establecer la prioridad a 0, ya que los buzones de `mqueue.h` se comportan por defecto como colas. En nuestra implementación, sin embargo, hemos empleado `n-i` —siendo `n` el número de items totales a enviar— como prioridad. Esto logra lo mismo que usar 0, pero con la ventaja de que es posible conocer desde el consumidor el número del item consumido, ya que se puede extraer de la prioridad.

IV. Uso de `sleep`

Para forzar el vaciado del buffer, basta con llamar a `sleep` en el productor. Realmente, no es importante en qué lugar del bucle del productor se llama a `sleep`. Independientemente de dónde sea invocada, si el productor duerme durante cierto tiempo (por ejemplo, 1 segundo), dará tiempo al consumidor a consumir todos los items que se encuentren en el buffer. Así, el consumidor se acabará bloqueando, esperando nuevos items, y el productor tardará en producirlos. En el momento en el que lo haga, el consumidor estará listo para consumir el item y volverá a bloquearse, esperando.

Recíprocamente, para forzar el llenado del buffer, es suficiente con llamar a `sleep` en el consumidor. De nuevo, no es importante el lugar en el que se llama a la función. Si el consumidor ejecuta `sleep` en cada iteración, da tiempo al productor a llenar el buffer de items producidos. En el momento en el que esté lleno, se bloqueará, esperando el consumo de algún item y la recepción de un mensaje de control. Cuando el consumidor finalmente consume un item, el productor ya está listo para insertar el siguiente, con lo que el buffer se vuelve a llenar instantáneamente después, el productor se bloquea y el consumidor duerme.

V. Conclusiones

La motivación de esta solución viene dada por la existencia de regiones críticas. Con buffers de archivo compartido, es necesario gestionar las carreras críticas con técnicas como semáforos. Esto complica considerablemente el código y da lugar a potenciales errores del programador, quien debe controlar el acceso a las regiones críticas.

Con paso de mensajes, sin embargo, los productores y consumidores se comunican a través de mensajes que son gestionados por el kernel del sistema operativo. Esto significa que no hay una región crítica en el sentido tradicional, donde múltiples procesos e hilos compiten por el acceso a recursos compartidos. Cuando un productor produce un elemento, simplemente lo envía a una cola de mensajes gestionada por el kernel. Del mismo modo, cuando un consumidor necesita un elemento, lee un mensaje de la cola de mensajes. El kernel se encarga de gestionar el acceso a la cola de mensajes y garantizar que los mensajes se entreguen de manera segura y sin conflictos. Por lo tanto, no hay necesidad de sincronización explícita o exclusión mutua entre los productores y consumidores, ya que el kernel se encarga de gestionar la comunicación entre ellos de manera segura.

Esta solución facilita considerablemente el trabajo del programador, que ya no debe encargarse de gestionar carreras críticas. Sin embargo, debido a efectuar tantas llamadas al sistema operativo, el paso de mensajes puede resultar considerablemente más lento que otras técnicas.