Problema del productor-consumidor con memoria compartida

Asier Cabo Lodeiro y Hugo Gilsanz

1. Introducción

El informe describe y analiza la implementación de un problema clásico de sincronización: el productor-consumidor. En este caso, se ha implementado en lenguaje C utilizando memoria compartida (POSIX shared memory) para la comunicación entre dos procesos independientes. Se detalla cómo ejecutar los códigos, se comentan aspectos relevantes de su funcionamiento y se extraen algunas conclusiones sobre la solución implementada.

2. Objetivos

- **Implementar un modelo productor-consumidor**: Permitir que un proceso (productor) genere datos y los inserte en un buffer compartido, mientras que otro proceso (consumidor) retira y procesa dichos datos.

- **Uso de memoria compartida**: Emplear la API de POSIX (shm_open, mmap, etc.) para establecer un área de memoria accesible por ambos procesos.

- **Simulación de sincronización**: Realizar una sincronización básica mediante el uso de contadores y “espera activa” (busy waiting) para controlar el acceso al buffer.

3. Descripción del Problema

El problema productor-consumidor consiste en dos procesos que comparten un buffer de tamaño fijo (en este caso, 8 posiciones). El **productor** genera elementos (caracteres mayúsculos aleatorios) y los coloca en el buffer, mientras que el **consumidor** retira los elementos para procesarlos. Se deben cumplir las siguientes condiciones:

- El productor no debe escribir en el buffer si este está lleno.

- El consumidor no debe retirar elementos si el buffer está vacío.

4. Estructura y Análisis de los Códigos

4.1 Código del Productor (prod.c)

- **Inicialización de la memoria compartida:**
  Se crea la memoria compartida con shm_open y se ajusta su tamaño con ftruncate. El mapeo se realiza mediante mmap para obtener un puntero a la estructura compartida.

- **Generación de datos:**
  La función produce_item() utiliza rand() para generar un carácter mayúsculo aleatorio, simulando la producción de un nuevo elemento.

- **Inserción en el buffer:**
  La función insert_item() añade el elemento generado en la posición correspondiente del buffer y aumenta el contador elementos. Se imprime un mensaje indicando la posición en la que se ha insertado el dato.

- **Sincronización básica:**
  El productor verifica si el buffer está lleno (cuando elementos alcanza el valor de N) y, en ese caso, entra en un bucle de espera activa hasta que el consumidor retire alguno de los elementos. Además, al insertar el primer elemento se envía un mensaje para “despertar” al consumidor.

- **Simulación de retraso:**
  Se utiliza la función sleep(1) para simular el tiempo de producción y facilitar la observación del comportamiento.

4.2 Código del Consumidor (cons.c)

- **Acceso a la memoria compartida:**
  Se abre la misma región de memoria compartida creada por el productor usando shm_open y mmap.

- **Extracción y consumo:**
  La función remove_item() retira el último elemento insertado en el buffer

decrementando el contador elementos. Posteriormente, la función consume_item() imprime el elemento consumido.

- **Sincronización básica:**
  Se verifica que si el buffer se encuentra vacío (cuando elementos es 0), el consumidor espera activamente hasta que el productor inserte un nuevo elemento. Del mismo modo, se notifica al productor cuando se libera un espacio en el buffer (cuando elementos llega a N-1).

- **Simulación de retraso:**
  Se introduce un retardo de 2 segundos con sleep(2) para simular el proceso de consumo, permitiendo observar claramente la interacción entre ambos procesos.

5. Instrucciones de Compilación y Ejecución

5.1 Compilación

Para compilar los dos códigos, se puede usar el compilador GCC. Se recomienda incluir la biblioteca de hilos (-pthread). A continuación, vemos un ejemplo de compilación:

```
gcc prod.c -o productor -pthread
gcc cons.c -o consumidor -pthread
```

5.2 Ejecución

1. Abra dos terminales.

En una terminal, ejecute el productor:
./productor
En la otra terminal, ejecute el consumidor:
./consumidor

Ambos procesos comenzarán a interactuar a través del buffer compartido, mostrando mensajes en la terminal que indican la inserción y extracción de elementos.

6. Comentarios sobre el Funcionamiento

- **Sincronización y Espera Activa:**
  La sincronización se maneja mediante contadores y bucles de espera activa (busy waiting). Aunque funcional para fines de demostración, este método no es el más eficiente en términos de recursos, ya que consume CPU mientras espera.

- **Uso de Memoria Compartida:**
  La utilización de shm_open y mmap permite la comunicación entre procesos sin necesidad de copiar datos entre espacios de memoria, lo que optimiza el intercambio de información. Sin embargo, es el punto donde ocurren las carreras críticas por la falta de securización del proceso.

- **Robustez y Mejoras Potenciales:**
  La solución presentada no contempla la alta probabilidad de carreras críticas que pueden llegar a ocurrir durante la ejecución de los programas. Se recomienda implementar mecanismos para el control de acceso a la memoria en las zonas críticas, como semáforos.

- **Simplicidad vs. Eficiencia:**
  La implementación es sencilla y permite entender de forma clara el funcionamiento básico de la comunicación mediante memoria compartida. Sin embargo, la espera activa y la ausencia de mecanismos de bloqueo avanzados pueden limitar su escalabilidad en entornos de producción.

7. Conclusiones

- **Demostración del Concepto:**
  La implementación cumple su propósito al demostrar de forma clara y sencilla la interacción entre procesos mediante un buffer compartido. Los mensajes impresos en la terminal permiten visualizar el flujo de datos y la sincronización básica entre el productor y el consumidor.

- **Limitaciones y Áreas de Mejora:**
  Aunque funcional, el uso de espera activa representa un desperdicio de recursos. La implementación se beneficiaría del uso de semáforos o mutexes para una sincronización más eficiente (véase *Uso de semáforos para la resolución de carreras críticas*). Además, la gestión adecuada de la memoria compartida (cerrado y eliminación) es necesaria para evitar problemas en sistemas reales.