Uso de semáforos para la resolución de carreras críticas

Asier Cabo Lodeiro y Hugo Gilsanz

1. Introducción

El problema de las carreras críticas ocurre cuando múltiples procesos acceden y modifican datos compartidos simultáneamente, lo que puede llevar a resultados inconsistentes. Como se vio en el ejercicio anterior, tras forzarlo colocando adecuadamente llamadas a la función sleep, el problema del productor-consumidor en el que se comparte en memoria un buffer sobre el que ambos trabajan, puede llevar a que estos sucesos tengan lugar. Para resolver este problema en sistemas concurrentes, se utilizan mecanismos de sincronización como los semáforos. En este informe, se analiza el uso de semáforos en la solución al problema anterior, asegurando el acceso controlado a la memoria compartida.

2. Problemática y posible solución

2.1 Descripción del problema

El problema productor-consumidor involucra dos tipos de procesos: el **productor**, que genera elementos y los coloca en un buffer compartido, y el **consumidor**, que retira estos elementos para procesarlos. Sin un mecanismo de sincronización, pueden ocurrir condiciones de carrera, como el acceso simultáneo a posiciones del buffer o la lectura de datos inconsistentes.

2.2 Incorporación de semáforos

Para solucionar el problema, se implementan tres semáforos.

- **VACIAS**: controla la cantidad de espacios disponibles en el buffer.
- **LLENAS**: indica la cantidad de elementos disponibles para ser consumidos.
- **MUTEX**: garantiza la exclusión mutua en la sección crítica.

3. Estructura y Análisis de los códigos

Como se están empleando como base los códigos del apartado anterior (véase el informe *Problema del productor-consumidor con memoria compartida*), en este apartado solo se describirán las actualizaciones con respecto a dichos programas y su funcionamiento.

3.1 Código del Productor (**prod_sem.c**)

1. Generación de un elemento (carácter aleatorio en mayúsculas).
2. Espera a que haya espacio disponible en el buffer (sem_wait(VACIAS)).
3. Espera acceso exclusivo a la memoria compartida (sem_wait(MUTEX)).
4. Inserta el elemento en el buffer.
5. Libera el acceso a la memoria compartida (sem_post(MUTEX)).
6. Aumenta la cantidad de elementos disponibles (sem_post(LLENAS)).
7. Simula un retraso de tiempo aleatorio en la producción con sleep.

Cabe destacar que también es el productor quien se encarga de eliminar los semáforos previos por seguridad y los creados por el programa (sem_unlink); además de crearlos (sem_open) para que el productor solo tenga que abrirlos.

3.2 Código del Consumidor (**cons.c**)

1. Espera a que haya elementos disponibles en el buffer (sem_wait(LLENAS)).
2. Espera acceso exclusivo a la memoria compartida (sem_wait(MUTEX)).
3. Extrae un elemento del buffer.
4. Libera el acceso a la memoria compartida (sem_post(MUTEX)).
5. Indica que hay un espacio libre en el buffer (sem_post(VACIAS)).
6. Procesa el elemento.
7. Simula un retraso de tiempo aleatorio en el consumo con sleep.

3.3 Implementación en C

Se implementa en C utilizando la biblioteca pthread.h y semaphore.h, junto con shm_open para la memoria compartida. Se definen las estructuras de datos, se inicializan los semáforos y se gestiona la memoria compartida adecuadamente para la comunicación entre procesos.

4. Instrucciones de Compilación y Ejecución

En referencia a este apartado véase el informe anterior (*Problema del productor-consumidor con memoria compartida*) con la única diferencia de que en este caso no es necesario interrumpir manualmente los códigos ya que los bucles son finitos, y evidentemente el nombre de los archivos es diferente, pero no las opciones de compilación.

5. Conclusiones

La solución basada en semáforos garantiza que el productor y el consumidor operen de manera sincronizada, evitando problemas como la sobreescritura de datos o la lectura de valores inexistentes. Se logra una ejecución eficiente y segura en un entorno concurrente, demostrando la importancia de los semáforos en la resolución de condiciones de carrera.