Opcional 2: Problema del Productor-Consumidor con Procesos

Pablo Mouriño Lorenzo, César Poza González

April 1, 2025

1 Introducción

Este informe documenta la implementación del problema del **productor-consumidor** con múltiples procesos. Se permite definir un número arbitrario de productores y consumidores, los cuales interactúan mediante **memoria compartida** y **semaféros POSIX**.

2 Implementación

La implementación consta de los siguientes elementos principales:

2.1 Memoria Compartida

Se usa la función `mmap()` para crear un espacio de memoria compartida que contiene:

- Un buffer de tamaño fijo `N=8` que actúa como una **cola LIFO**.
- Un contador `nElementosBuffer` que indica el número de elementos actuales en el buffer.

2.2 Manejo de Procesos

Se utilizan `fork()` para crear procesos productores y consumidores, los cuales acceden concurrentemente al buffer.

2.3 Uso de Semáforos

Los semáforos POSIX controlan la concurrencia:

- `vacias`: Indica los espacios libres en el buffer.
- `llenas`: Indica los elementos disponibles para consumir.

- `mutex`: Garantiza la exclusión mutua en las secciones críticas.

Los semáforos se inicializan con `sem_open()` y se destruyen con `sem_unlink()` al final.

3 Ejecución

La implementación actual del main recibe como argumentos el número de productores y el número de consumidores.
En la figura 1 observamos un ejemplo de ejecución en terminal Linux para 2 productores y 3 consumidores.
Cada productor genera una letra mayúscula aleatoria y la inserta en el buffer. Los consumidores extraen las letras en orden LIFO y las procesan.

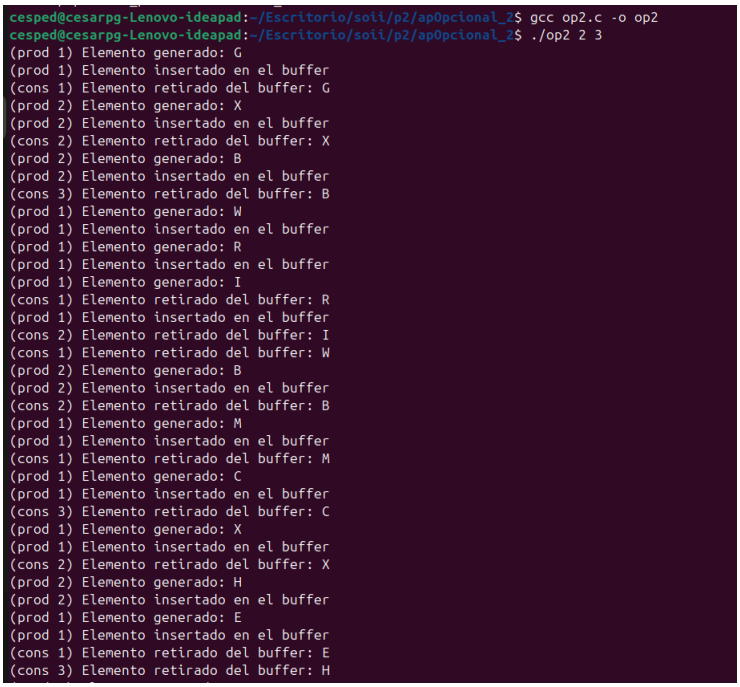

Figure 1: Ejecución con 2 procesos productores y 3 consumidores

A continuación, en la figura 2, se muestra otro ejemplo de ejecución para 4 productores y 4 consumidores, se observa como los procesos se sincronizan correctamente trabajando todos a la vez.

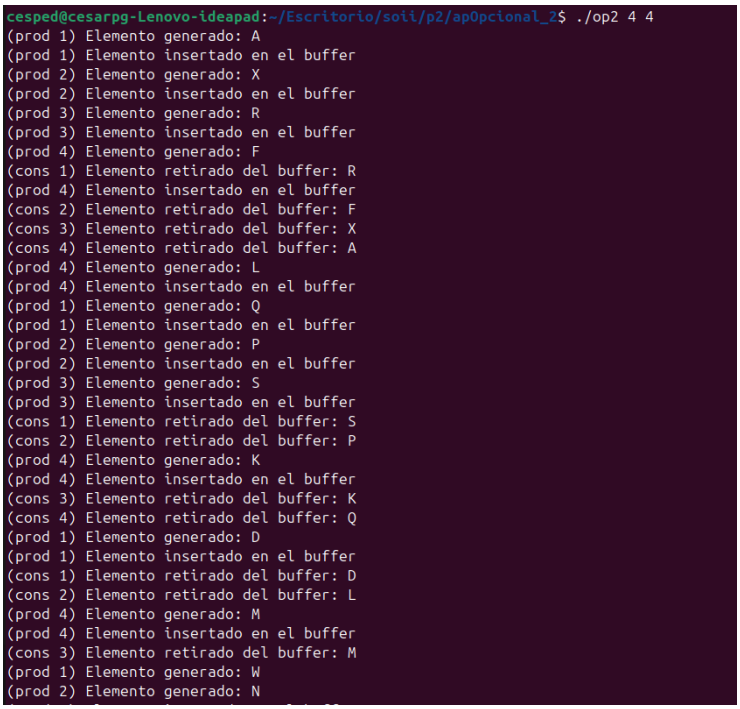

Figure 2: Ejecución con 4 procesos productores y 4 consumidores

4 Resultados y Conclusiones

Se comprobó que la implementación permite la correcta sincronización entre múltiples procesos. Se observaron las siguientes conclusiones:

- La memoria compartida y los semáforos garantizan la integridad de los datos.
- La concurrencia se maneja eficientemente, evitando condiciones de carrera.
- La ejecución con múltiples productores y consumidores permite una carga de trabajo equilibrada.
- El uso de `sleep()` con valores aleatorios simula la variabilidad en los tiempos de producción y consumo.

En futuros trabajos, se podría extender la implementación para admitir buffers circulares o un mecanismo de registro de eventos más detallado.