Opcional 1: Problema del Productor-Consumidor con Threads

Pablo Mouriño Lorenzo, César Poza González

April 1, 2025

1 Introducción

El problema del productor-consumidor es un problema clásico de sincronización en sistemas operativos. Se trata de gestionar el acceso a un buffer compartido entre dos entidades:

- El productor genera datos y los coloca en el buffer.
- El consumidor extrae datos del buffer y los procesa.

El objetivo es evitar condiciones de carrera y garantizar la sincronización mediante semáforos.

2 Implementación

Se ha desarrollado una solución utilizando hilos (`pthread`) y semáforos POSIX (`sem_t`). El código está compuesto por:

- Un buffer de tamaño fijo (N=8) compartido entre productor y consumidor.
- Semáforos para la exclusión mutua y sincronización de acceso al buffer.
- Funciones para la producción, inserción, extracción y consumo de elementos.
- Un productor y un consumidor ejecutados en hilos separados.

Los semáforos utilizados cumplen los siguientes propósitos:

- **vacias**: Controla la cantidad de espacios disponibles en el buffer. Se inicializa en `N`.
- **llenas**: Controla la cantidad de elementos en el buffer. Se inicializa en 0.
- **mutex**: Garantiza la exclusión mutua en el acceso al buffer compartido.

El flujo de ejecución del programa es el siguiente:

1. El productor genera un carácter aleatorio y espera a que haya espacio disponible en el buffer.
2. Una vez que hay espacio, el productor inserta el carácter en el buffer y actualiza los semáforos.
3. El consumidor espera a que haya elementos en el buffer para consumir.
4. Cuando hay un elemento disponible, el consumidor lo extrae del buffer y lo almacena en su variable local.
5. Ambos procesos incluyen retardos aleatorios (`sleep(rand() % 4)`) para simular concurrencia real.

2.1 Código en C

El siguiente fragmento muestra la estructura principal del código:

Listing 1: Código del Productor-Consumidor con Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 8
#define ITERACIONES 60

char buffer[N];
int nElementosBuffer = 0;
sem_t vacias, llenas, mutex;

void* productor(void* arg) {
    char elemento;
    for (int i = 0; i < ITERACIONES; i++) {
        elemento = 'A' + rand() % 26;
        sem_wait(&vacias);
        sem_wait(&mutex);
        buffer[nElementosBuffer++] = elemento;
        sem_post(&mutex);
        sem_post(&llenas);
        sleep(rand() % 4);
    }
    return NULL;
}

void* consumidor(void* arg) {
```

```
    char elemento;
    for (int i = 0; i < ITERACIONES; i++) {
        sem_wait(&llenas);
        sem_wait(&mutex);
        elemento = buffer[--nElementosBuffer];
        sem_post(&mutex);
        sem_post(&vacias);
        sleep(rand() % 4);
    }
    return NULL;
}

int main() {
    pthread_t hiloProductor, hiloConsumidor;
    sem_init(&vacias, 0, N);
    sem_init(&llenas, 0, 0);
    sem_init(&mutex, 0, 1);
    pthread_create(&hiloProductor, NULL, productor, NULL);
    pthread_create(&hiloConsumidor, NULL, consumidor, NULL);
    pthread_join(hiloProductor, NULL);
    pthread_join(hiloConsumidor, NULL);
    sem_destroy(&vacias);
    sem_destroy(&llenas);
    sem_destroy(&mutex);
    return 0;
}
```

3 Ejecución

En la figura 1 observamos un ejemplo de ejecución desde una terminal Linux.
Para compilar y ejecutar el código en un entorno Linux:

```
gcc -o productor_consumidor programa.c -lpthread -lrt
./productor_consumidor
```

4 Resultados y Conclusiones

El experimento realizado muestra que el uso de semáforos permite coordinar correctamente el acceso al buffer compartido. Sin la sincronización adecuada, se podrían presentar problemas como:

- **Condiciones de carrera**: Si varios hilos intentaran modificar el buffer simultáneamente sin protección, podrían producir resultados inconsistentes.

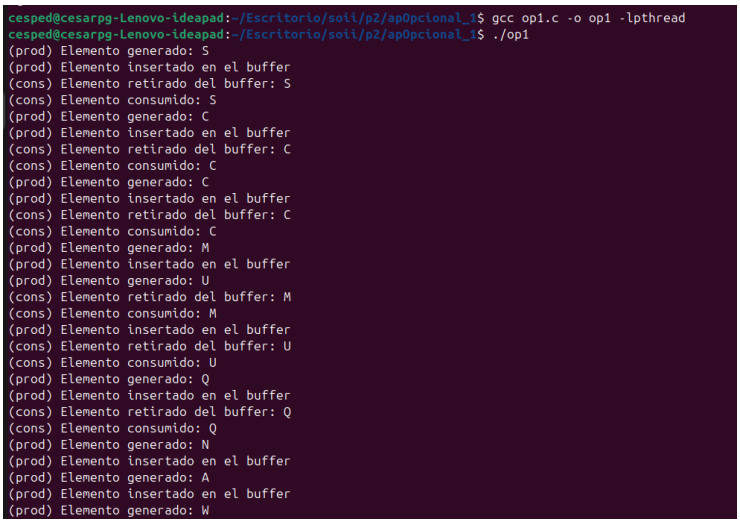

Figure 1: Ejemplo de ejecución

- **Lecturas y escrituras incorrectas**: Sin semáforos, un consumidor podría intentar leer datos de un buffer vacío o un productor podría sobrescribir datos sin ser consumidos.
- **Deadlocks**: En una mala implementación, los hilos podrían quedar bloqueados indefinidamente sin progresar.

La implementación presentada evita estos problemas al utilizar semáforos de exclusión mutua y sincronización, asegurando una ejecución ordenada y eficiente del productor y el consumidor. Además, la introducción de retardos aleatorios en la ejecución de los hilos permite observar la concurrencia en acción, validando la eficacia del mecanismo de sincronización.