

Práctica 2: Sincronización de procesos con semáforos

Pedro y Pablo Seijo

7 de abril de 2024

1. Ejercicio 1

Para forzar las condiciones de carrera, modificamos el código introduciendo sleeps en el productor y el consumidor. Estos sleeps simulan retrasos en la producción y el consumo de elementos, lo que puede llevar a situaciones de carrera si no se sincronizan adecuadamente. A continuación se describen las condiciones de carrera que se pueden producir:

1.1. Carrera Crítica Productor

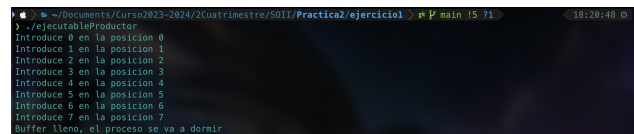
Para esta carrera introducimos un sleep en esta parte del código:



```
1  for(i=0; i<100; i++){
2
3      if(buffer[N]==N){
4          printf("Buffer lleno, el proceso se va a dormir\n");
5          while(buffer[N]==N);
6      }
7
8      sleep(15); //Simulamos un tiempo de espera para la carrera
9
10     //item=produce_item();
11     insert_item(i);
12 }
```

Figura 1: Código Productor Carrera Crítica

Y la ejecución es la siguiente:



```
~/Documents/Unib2023-2024/2Cuatrimestre/S011/Practica2/ejercicio1 > ./ejecutableProductor
Introduce 0 en la posición 0
Introduce 1 en la posición 1
Introduce 2 en la posición 2
Introduce 3 en la posición 3
Introduce 4 en la posición 4
Introduce 5 en la posición 5
Introduce 6 en la posición 6
Introduce 7 en la posición 7
Buffer lleno, el proceso se va a dormir
```

Figura 2: Carrera Crítica Productor

Aquí, el productor intenta llenar el buffer con nuevos elementos. Al igual que el consumidor, el productor también emplea una espera activa para verificar el estado del buffer (`while(buffer[N]==N);`). Si el buffer está lleno, el productor se pone a dormir durante 15 segundos, lo que representa un retraso en la producción de nuevos elementos. Este sueño forzado puede hacer que el productor sea más lento que el consumidor en ciertos momentos, variando la dinámica de producción y consumo y posiblemente llevando a condiciones de carrera si el consumidor se acelera inesperadamente.

1.2. Carrera Crítica Consumidor

Para esta carrera introducimos un sleep en esta parte del código:

```

1
2 void consume_item() {
3     // Verificar si hay elementos para consumir
4     if (buffer[N] > 0) {
5         // Decrementar la cuenta de elementos en el buffer
6         buffer[N]--;
7
8         // Introducimos un retardo aleatorio para simular el tiempo de consumo
9         sleep(rand() % 5);
10        sleep(15); // Para probar el caso en que el buffer se llena y el productor se duerme
11
12        int posConsumido = buffer[N];
13        int consumido = buffer[posConsumido];
14
15        // Marcamos la posición como vacía o consumida
16        buffer[posConsumido] = -1;
17
18        // Imprimir información del elemento consumido
19        printf("Valor %d consumido (Posición: %d)\n", consumido, posConsumido);
20
21    }
22 }

```

Figura 3: Código Carrera Crítica Consumidor

Y la ejecución es la siguiente:

```

Abrir archivo en el editor (cmd + clic) | i:ciol > main !6 74 | 18:52:01
o > ./ejecutableConsumidor
Buffer vacío, el proceso se va a dormir
Valor 0 consumido (Posición: 0)
Buffer vacío, el proceso se va a dormir

```

Figura 4: Carrera Crítica Consumidor

De la misma manera en la situación contraria pasa lo siguiente:

- El consumidor entra en un estado de espera al detectar que el buffer está vacío. Esta espera se simula mediante una instrucción de suspensión (sleep), durante la cual no hay consumo de elementos.
- Paralelamente, el productor se activa y comienza a llenar el buffer. Debido a que no hay sincronización, no está consciente de la actividad del consumidor.
- Si el consumidor sale de su estado de espera y comienza a consumir un elemento al mismo tiempo que el productor está insertando uno, se accede simultáneamente al buffer. Sin mecanismos de sincronización, ambos procesos pueden leer y escribir en la misma posición del buffer o modificar la cuenta de elementos de forma concurrente.
- Esta simultaneidad puede resultar en que el productor sobrescriba un valor que aún no ha si-

do consumido o que el consumidor consuma un elemento que el productor cree que ha sido insertado con éxito. Además, la cuenta del buffer (buffer[N]) podría ser incrementada o decrementada incorrectamente, causando desbordamientos o subconsumos.

Teniendo en cuenta que una carrera crítica se refiere a un sistema donde su comportamiento final depende de la secuencia o del tiempo de los procesos, y no está controlado por el SO o el programa, sin los controles adecuados, la carrera crítica puede llevar a condiciones de carrera, donde los procesos compiten por recursos compartidos y pueden producir resultados inesperados o indeseados.

2. Ejercicio 2

La salida del programa es la siguiente:

```

~/Documents/Curso2021-2024/2Cuatrimestre/S011/Practica2/ejercicio2 > ./main
19:56:40
./ejecutableConsumidor2
Consumido el valor 5 de la posición 3, suma de valores en el buffer: 17
Consumido el valor 10 de la posición 4, suma de valores en el buffer: 12
Consumido el valor 0 de la posición 4, suma de valores en el buffer: 2
Consumido el valor 2 de la posición 4, suma de valores en el buffer: 4

```

Figura 5: Salida del programa de consumidor

```

~/Doc/C/2Cuatrimestre/S011/Practica2/ejercicio2 > ./main
19:57:23
./ejecutableProductor2
Producido el valor 2 en la posición 7
Producido el valor 0 en la posición 6
Producido el valor 0 en la posición 5
Producido el valor 10 en la posición 4
Producido el valor 5 en la posición 3
Producido el valor 0 en la posición 4
Producido el valor 2 en la posición 4
Producido el valor 6 en la posición 4

```

Figura 6: Salida del programa de productor

En este ejercicio se ha implementado una solución al problema del productor-consumidor utilizando memoria compartida y semáforos para gestionar el acceso concurrente a un buffer. Este buffer es de tipo entero (int) y funciona como una pila LIFO (Last In, First Out), donde el último elemento insertado es el primero en ser retirado. La implementación se ha realizado siguiendo estos criterios:

- El buffer tiene un tamaño de $N = 8$ elementos.

- La función `produce_item()` genera un entero aleatorio entre 0 y 10 para simular la producción de un elemento.
- La función `insert_item(int item)` inserta el elemento producido en el buffer siguiendo la política LIFO.
- La función `consume_item()` retira un elemento del buffer, suma este elemento con todos los valores que haya en el buffer en ese momento, y muestra por pantalla el valor consumido y el resultado de la suma.

Se han utilizado semáforos para resolver las carreras críticas, asegurando que sólo un proceso acceda al buffer en un momento dado y gestionando el llenado y vaciado del buffer de manera sincronizada. Los semáforos utilizados son:

- Un semáforo llamado **VACIAS**, inicializado en N , que indica el número de posiciones vacías en el buffer.
- Un semáforo llamado **LLENAS**, inicializado en 0, que indica el número de posiciones llenas en el buffer.
- Un semáforo de exclusión mutua, **MUTEX**, inicializado en 1, que asegura el acceso exclusivo al buffer durante las operaciones de inserción y retirada.

Para evitar la interrupción del programa en cada ejecución y permitir la observación de la dinámica productor-consumidor, se ha establecido un número finito de iteraciones (100) y se han incluido llamadas a la función `sleep()` con valores aleatorios entre 0 y 3 segundos, fuera de las regiones críticas, para simular variaciones en las velocidades de producción y consumo.

Finalmente, para asegurar la limpieza y evitar que los semáforos permanezcan activos en el kernel después de la ejecución, se utilizan las funciones `sem_close()` y `sem_unlink()` al final del programa.

Esta solución demuestra la efectividad de los semáforos para gestionar la sincronización en problemas de concurrencia, permitiendo un control deta-

llado sobre el acceso a recursos compartidos y evitando condiciones de carrera, mientras se mantiene una secuencia ordenada y predecible de operaciones de producción y consumo.

3. Ejercicio 3

Ahora se cambia la implementación para utilizar hilos en lugar de procesos. La salida del programa es la siguiente:

```

$ ./productorConsumidorHilos
Producido: 8 en posición: 7
Consumido: 8, Suma en el buffer: 8
Producido: 0 en posición: 7
Consumido: 0, Suma en el buffer: 0
Producido: 9 en posición: 7
Consumido: 9, Suma en el buffer: 9
Producido: 10 en posición: 7
Consumido: 10, Suma en el buffer: 10
Producido: 4 en posición: 7
Consumido: 4 en posición: 6
Consumido: 4, Suma en el buffer: 8
Producido: 0 en posición: 6
Consumido: 0, Suma en el buffer: 4
Producido: 10 en posición: 6
Consumido: 10, Suma en el buffer: 14

```

Figura 7: Salida del programa con hilos

3.1. Definición y Variables Globales

El tamaño del buffer se define como $N = 8$, utilizando un arreglo de enteros para simular el buffer. Los semáforos `vacias`, `llenas`, y `mutex` se utilizan para controlar el número de posiciones vacías, el número de posiciones llenas, y asegurar el acceso exclusivo al buffer, respectivamente.

3.2. Funciones Clave

- `produce_item()`: Genera un elemento aleatorio entre 0 y 10.
- `insert_item(int item)`: Inserta un elemento en el buffer según la política LIFO, decrementando previamente el índice del buffer.
- `remove_item()`: Elimina un elemento del buffer e imprime el elemento consumido junto con la suma de los valores actuales en el buffer.

3.3. Hilos y Sincronización

La implementación utiliza dos tipos de hilos: productores y consumidores. Cada productor genera elementos aleatorios y los inserta en el buffer, mientras que cada consumidor retira elementos del buffer. La sincronización se logra mediante el uso de semáforos, donde:

- Los semáforos `vacias` y `llenas` gestionan el número de posiciones vacías y llenas en el buffer, respectivamente.
- El semáforo `mutex` asegura el acceso exclusivo al buffer para insertar o remover elementos.

Los hilos productores esperan que haya posiciones vacías para producir, mientras que los consumidores esperan que el buffer contenga elementos para consumir. Esto asegura que no se produzcan condiciones de carrera y que el acceso al buffer sea seguro y ordenado.

4. Conclusiones

[7-8]