

Problema del productor-consumidor con mutexes y variables de condición

JAIME ROMERO MARTÍNEZ, SALVIA SISA CORTÉS

Sistemas Operativos II

Grupo 05

{jaime.romero, salvia.sisa}@rai.usc.es

I. INTRODUCCIÓN

En este informe se tratará la **implementación en C del problema del productor-consumidor con hilos**, utilizando mutexes y variables de condición, tomando como referencia la solución que propone *Andrew S. Tanenbaum* en su libro *Modern Operating Systems*.

El problema del **productor-consumidor** (o del búfer limitado) consiste en **dos procesos** (productor y consumidor) que comparten un **búfer de tamaño fijo**. Básicamente, el productor se dedica a colocar elementos en el búfer mientras que el consumidor los retira del mismo. Podríamos decir que un **mutex** es una versión simplificada de un semáforo pero sin la habilidad de contar, ya que se trata de una variable que solo puede encontrarse en **dos estados**: abierto o cerrado. Esta variable es esencial para la correcta gestión de las regiones críticas. Por otro lado, para una solución más completa, además de mutexes, se debe utilizar otro mecanismo de sincronización: las **variables de condición**. Estas hacen que los hilos se bloqueen debido al incumplimiento de alguna condición, lo que logra que la espera y bloqueo de un hilo se realicen de forma atómica.

El **reto** de este programa reside principalmente en lograr una **sincronización** entre todos los hilos de productores y consumidores, ya que es de suma importancia resguardar cada región crítica y, por lo tanto, hacer un buen uso de los mutexes.

II. IMPLEMENTACIÓN

El código se denomina `prod_cons.c`, y recibe como **parámetros de línea de comandos** tres valores: el número de productores (P), el número de consumidores (C) y el tamaño del buffer (N). Se debe compilar con gcc, con la opción de `-lpthread`, y al ejecutarlo, cada hilo solicitará al usuario un **valor máximo de tiempo de espera**. Después, la ejecución del programa prosigue hasta que todos los hilos terminan sus tareas.

Para esta solución hemos utilizado **una barrera**, para poder sincronizar el inicio de todos los productores y consumidores de forma que estos esperen a que el usuario introduzca el tiempo de espera máximo para la totalidad de los hilos. También utilizamos **dos mutexes**: uno para gestionar el acceso al búfer común (`mutexBuffer`, referido en el *Tanenbaum* como `elMutex`) y otro para que los `printfs` de solicitud de tiempo máximo no se superpongan (`mutexPrintf`). Además, necesitamos **dos variables de condición**, una para el productor y otra para el consumidor, con el objetivo de que estos esperen cuando el búfer esté lleno o vacío.

En cuanto al **búfer**, como los hilos comparten un espacio de direcciones común, este es simplemente una **cadena de texto compartida**. Para su manipulación también hemos declarado la variable entera `lenBuffer` para llevar la **cuenta de los elementos** del buffer. Esto es debido a que si calculásemos cada vez la longitud del búfer con `strlen()`, tendríamos que asegurar que este siempre termine en el carácter nulo, además de resultar menos eficiente, ya que lo que hace es recorrer toda la cadena (complejidad $O(N)$).

En el **main** se realizan las siguientes funciones:

- Comprobación y parseo de los parámetros de entrada.
- Inicialización de la barrera con `pthread_barrier_init()`
- Inicialización de los mutexes y las variables de condición con `pthread_mutex_init()` y `pthread_cond_init()`.
- Reserva de memoria para el buffer (`malloc(N)`) e inicialización del mismo al carácter nulo y de `lenBuffer` a 0.
- Declaración de los vectores de productores y de consumidores.
- Creación de los productores y consumidores (con `pthread_create()`) y espera por los mismos (`pthread_join()`).
- Liberación de recursos mediante la destrucción de la barrera, de los mutexes y de las variables de condición, y el `free()` del búfer.

A. Productor

El **bucle principal** del productor (Figura 1) transcurre de la siguiente forma:

1. Dado que va a modificar el búfer (región crítica), **bloquea el *mutex*** que regula el acceso al mismo (ln 2).
2. Comprueba si el búfer no está vacío. Si es así, **espera a que el consumidor lo vacíe**, mediante la variable de condición `condProductor`, que libera el *mutex* mientras espera y lo vuelve a adquirir al ser despertado (línea 3).
3. Una vez el búfer está libre, **duerme** un tiempo aleatorio (ln 4).
4. **Introduce el ítem actual** en el búfer compartido mediante la función `colocarItem` (ln 5).
5. Muestra un mensaje por pantalla indicando qué productor ha producido qué ítem (ln 6).
6. **Avisa a todos los consumidores** de que hay un nuevo ítem en el búfer usando la variable de condición `condConsumidor` (ln 7).
7. Finalmente, **libera el *mutex*** para que los otros hilos puedan acceder al búfer (ln 8).

```
1 for (int i = 0; i < strlen(datosLeidos); i++){
2     pthread_mutex_lock(&mutexBuffer);
3     while (strlen(buffer) != 0) pthread_cond_wait(&condProductor, &mutexBuffer);
4     sleep(rand() % tiempoEsperaMax);
5     colocarItem(datosLeidos[i]);
6     printf("[PRODUCTOR %d] He intrducido el item %c\n", numProd, datosLeidos[i]);
7     pthread_cond_broadcast(&condConsumidor);
8     pthread_mutex_unlock(&mutexBuffer);
9 }
10 fclose(archivo);
11 printf("[PRODUCTOR %d] He terminado\n", numProd);
12 pthread_exit(NULL);
13 }
```

Figura 1: Bucle principal del productor

B. Consumidor

El **bucle principal** del consumidor (Figura 2) transcurre de la siguiente forma:

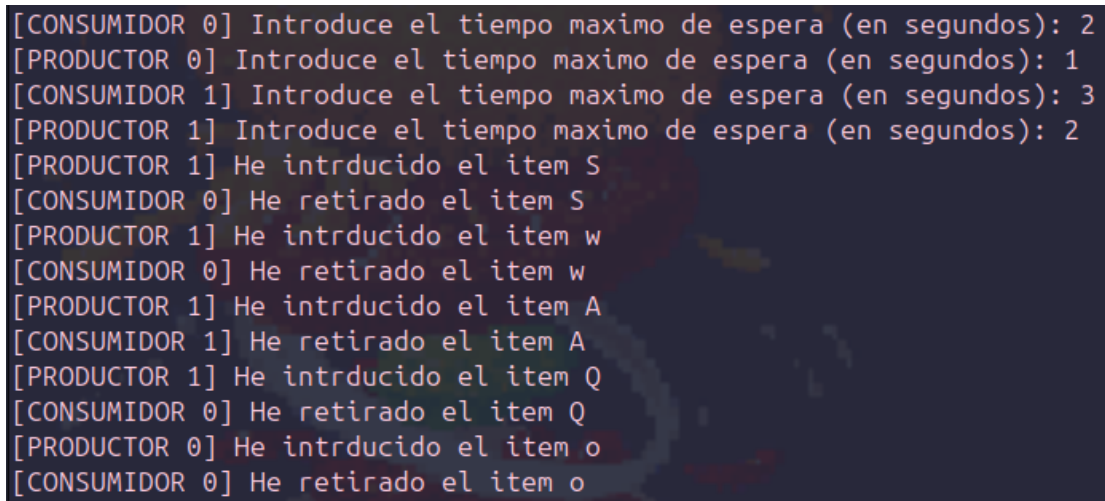
1. **Bloquea el *mutex*** para acceder de forma segura al búfer compartido (línea 2).
2. Mientras el búfer esté vacío y no se hayan recibido todos los terminadores (*), **espera una señal** de que hay nuevos ítems disponibles (líneas 3-4).
3. Si el búfer sigue vacío y ya se han recibido todos los terminadores, desbloquea el *mutex* y finaliza el bucle (líneas 5-7).
4. **Duerme** durante un número aleatorio de segundos (*sleep*) (línea 9).
5. **Extrae el primer ítem** del búfer (línea 10), decrementa el tamaño (línea 11), desplaza el resto de ítems (línea 12) y ajusta el búfer (línea 13).
6. Si el ítem leído es un *, incrementa el **contador de terminadores recibidos** (línea 14).
7. **Notifica a los productores** que pueden introducir nuevos ítems, usando la variable de condición *condProductor* (línea 15).
8. Si ya se han recibido todos los terminadores, también **notifica al resto de consumidores** para que terminen (líneas 16-17).
9. **Libera el *mutex*** (línea 18).
10. Escribe el ítem consumido en el **archivo de salida** (línea 19).
11. Imprime por pantalla qué ítem ha retirado qué consumidor (línea 20).

```
1 while (1){
2     pthread_mutex_lock(&mutexBuffer);
3     while (lenBuffer == 0 && terminadoresRecibidos < P)
4         pthread_cond_wait(&condConsumidor, &mutexBuffer);
5     if (lenBuffer == 0 && terminadoresRecibidos >= P) {
6         pthread_mutex_unlock(&mutexBuffer);
7         break;
8     }
9     sleep(rand() % tiempoEsperaMax);
10    char item = buffer[0];
11    lenBuffer--;
12    memmove(buffer, buffer+1, lenBuffer);
13    buffer[lenBuffer] = '\0';
14    if (item == '*') terminadoresRecibidos++;
15    pthread_cond_broadcast(&condProductor);
16    if (terminadoresRecibidos >= P)
17        pthread_cond_broadcast(&condConsumidor);
18    pthread_mutex_unlock(&mutexBuffer);
19    fputc(item, archivo);
20    printf("[CONSUMIDOR %d] He retirado el item %c\n", numCons, item);
21
22 }
```

Figura 2: Bucle principal del consumidor

III. EJECUCIÓN DEL PROGRAMA

Al ejecutar el programa (Figura 3) para **varios números tanto de productores como de consumidores**, vemos que este funciona correctamente, evitando las potenciales carreras críticas gracias al uso de los mutexes y las variables de condición. Además, al comprobar los ficheros de salida vemos que efectivamente, los items son consumidos en su totalidad y en el orden correcto de acuerdo al funcionamiento del búfer como una estructura FIFO.



```
[CONSUMIDOR 0] Introduce el tiempo maximo de espera (en segundos): 2
[PRODUCTOR 0] Introduce el tiempo maximo de espera (en segundos): 1
[CONSUMIDOR 1] Introduce el tiempo maximo de espera (en segundos): 3
[PRODUCTOR 1] Introduce el tiempo maximo de espera (en segundos): 2
[PRODUCTOR 1] He intrducido el item S
[CONSUMIDOR 0] He retirado el item S
[PRODUCTOR 1] He intrducido el item w
[CONSUMIDOR 0] He retirado el item w
[PRODUCTOR 1] He intrducido el item A
[CONSUMIDOR 1] He retirado el item A
[PRODUCTOR 1] He intrducido el item Q
[CONSUMIDOR 0] He retirado el item Q
[PRODUCTOR 0] He intrducido el item o
[CONSUMIDOR 0] He retirado el item o
```

Figura 3: *Ejemplo de inicio de ejecución*

IV. CONCLUSIÓN

La implementación del problema del productor-consumidor implementada demuestra una **correcta sincronización entre múltiples hilos** mediante el uso de **mutexes** y **variables de condición**. Se ha respetado la exclusión mutua en las regiones críticas y se ha garantizado que no haya condiciones de carrera ni accesos concurrentes indebidos al búfer.

La solución incorpora una barrera para una inicialización de los hilos ordenada, y permite un control sobre el tiempo de espera simulado, lo cual facilita la observación del comportamiento concurrente. La decisión de implementar el búfer como una cadena de texto compartida y controlar su longitud mediante una variable adicional (`lenBuffer`) mejora la eficiencia y claridad del código.

En conjunto, este programa proporciona una **implementación robusta y segura** del problema del productor-consumidor, aplicando mecanismos de sincronización.