Sistemas Operativos II

Práctica 3 - Informe

Sincronización de procesos con mutexes

Xiana Carrera Alonso
Curso 2021-2022

04/04/2022

ÍNDICE

1 Introducción

1.1 Estructura de la práctica

Esta práctica está destinada al estudio de los mutexes y de las variables de condición para la sincronización de hilos. Esto se ha realizado a través del problema del productor-consumidor, del cual en la práctica 2 ya se habían analizado las situaciones de carreras críticas y una posible implementación de una solución empleando semáforos.

El desarrollo del código se ha realizado en base al algoritmo propuesto por [**Tanenbaum**], previamente analizado en clases teóricas.

Este informe hace referencia únicamente al ejercicio 1 de la práctica, de carácter obligatorio. Para él se imponen una serie de restricciones:

- El uso de un *buffer* LIFO (una pila) de tamaño N = 10.
- El funcionamiento del código para números arbitrarios de productores (P) y consumidores (C).
- Que cada productor inserte 20 ítems en el *buffer*.
- Que aparezcan llamadas aleatorias a *sleep()* o *usleep()* fuera de la región crítica para provocar que en las ejecuciones se alcancen situaciones en las que el *buffer* esté vacío o lleno, y por tanto se empleen las variables de condición.

En el apartado 1.2 de este informe se describirá el fundamento teórico de los mutexes y de las variables de condición. A continuación, en el apartado 2, se explicará la estructura del programa desarrollado y las conclusiones obtenidas al respecto.

1.2 Explicación teórica

1.2.1 Mutexes

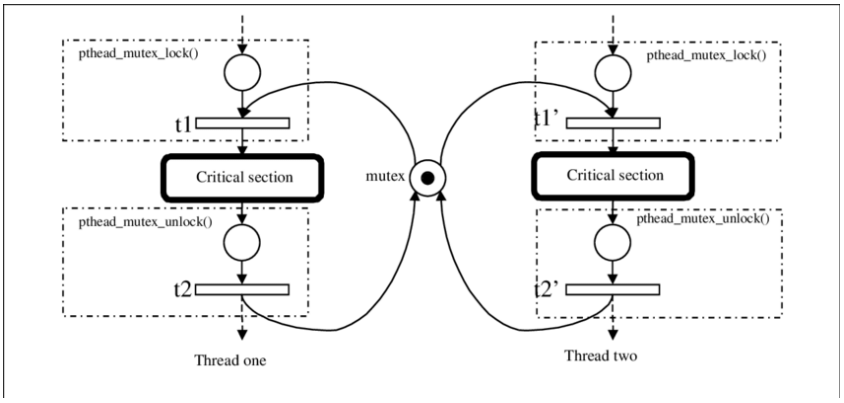

***Figura 1**. Representación del uso de un mutex en el acceso a una región crítica, por [**Kavi**].*

Un mutex es una variable que permite gestionar la exclusión mutua de hilos (del mismo o de distintos procesos) con respecto a un recurso compartido cuyo uso está delimitado por una determinada región crítica.

En este sentido, los mutexes son simplificaciones de los semáforos. Mientras que estos últimos podían tomar el valor 0 o un entero positivo, un mutex únicamente tiene dos posibles valores o estados: abierto (1, desbloqueado) y cerrado (0, bloqueado). Así, los mutexes pierden la capacidad de llevar la cuenta de condiciones, pero veremos que esto se puede suplir mediante el uso de variables auxiliares (las llamadas "variables de condición"). En el ejercicio 2 de esta práctica se estudia incluso la posibilidad de llevar a cabo este tipo de controles empleando únicamente mutexes.

Las funciones básicas al respecto que fueron empleadas en esta práctica son:

- **pthread_mutex_init()**

Su propósito es crear un mutex e inicializarlo con unos ciertos atributos, ya sea aquellos empleados por defecto o algunos escogidos específicamente por el programador.

El uso de *pthread_mutex_init()* por parte de un proceso permite que este pueda emplearlo posteriormente.

- **pthread_mutex_destroy()**

Esta función destruye un mutex, borrándolo del sistema. Se trata de una cuesstión importante de cara al manejo de recursos y a la eficiencia en el almacenamiento.

Un mutex destruido ya no podrá ser accedido por parte del proceso que lo estaba empleando. Además, únicamente se podrán destruir mutexes en estado desbloqueado.

- **pthread_mutex_lock()**

Los hilos pueden ejecutar esta función para solicitar la adquisición del mutex para sí mismos.

Si estaba libre, el hilo que lo solicita lo adquiere y el mutex pasa a estado bloqueado. Cualquier intento de acceso a él por parte de otro hilo provocará que se bloquee. De este modo, el primer hilo tendrá la seguridad de que es el único que habrá pasado la barrera del mutex que protege la región crítica.

Cuando un hilo termina de trabajar en la región crítica, tiene la responsabilidad de liberar el mutex, de forma que alguno de los hilos bloqueados que estaban esperando por él pueda continuar. Esto se realiza a través de la función *pthread_mutex_unlock().*

- **pthread_mutex_unlock()**

Esta función pasa un mutex de estado bloqueado a desbloqueado. En consecuencia, si había algún hilo dormido esperando por él, se le concederá el mutex a uno y solamente uno de ellos. El resto tendrán que seguir bloqueados. De esta forma, se tiene la seguridad de que irán accediendo a la región crítica de uno en uno, manteniendo la propiedad de exclusión mutua.

Si no había ningún hilo dormido, el mutex quedará desocupado y disponible para el primero que lo reclame.

Otra función relevante de mutexes es *pthread_mutex_trylock(),* que sigue el mismo patrón de funcionamiento que *pthread_mutex_lock(),* con la excepción de que la función no bloquea al hilo si el mutex estaba ocupado, y simplemente retorna. No obstante, no fue empleada en este programa.

La clave del funcionamiento de los mutexes es que está garantizado que sus procedimientos asociados se ejecutan de forma atómica.

1.2.2 Variables de condición

Las variables de condición permiten imponer una restricción sobre el acceso a la región crítica en relación al cumplimiento de una determinada condición, de forma adicional al control que llevan a cabo los mutexes.

Así, los hilos quedarán bloqueados no solo cuando ya hay algún otro empleando la región crítica, sino también cuando la restricción impuesta les prohíbe el acceso.

En el caso del problema del productor-consumidor, las variables de condición son empleadas para dormir a los productores cuando el *buffer* está vacío, y a los consumidores cuando está lleno. Únicamente cuando se levante la condición podrán comenzar a ejecutar la región crítica.

Las funciones de variables de condición empleadas en este programa han sido:

- **pthread_cond_init()**

De forma análoga a como lo hacía *pthread_mutex_init()* para mutexes, *pthread_cond_init()* inicializa una variable de condición, permitiendo su uso por parte del proceso.

- **pthread_cond_destroy()**

Análogamente a *pthread_mutex_destroy(), pthread_cond_destroy()* destruye la variable de condición, volviéndola inaccesible por parte del proceso y eliminándola de la lista de recursos en uso por parte del sistema.

- **pthread_cond_wait()**

El flujo habitual de uso de variables de condición es el siguiente: cuando un hilo pretende acceder a la región crítica, comprueba si se cumple la condición necesaria para ello. Si no es así, llama a *pthread_cond_wait()* en relación a la variable de condición asociada y al mutex de la región crítica. Esto provocará que el hilo quede dormido y, además, que libere el mutex, posibilitando que algún otro pueda entrar en la región crítica y alterarla de forma que se satisfaga la condición mencionada.

- **pthread_cond_signal()**

Cuando algún hilo realiza alguna acción que levanta la restricción de acceso a la región crítica a otro hilo por una determinada variable de condición, es su responsabilidad lanzar una señal para advertirle a través de *pthread_cond_signal().*

Esta función despertará a un y solamente un hilo que se encontrara esperando en un *pthread_cond_wait()* correspondiente a la variable de condición. En ese punto, el hilo despertará y solicitará acceso al mutex. Nótese que cuando lo adquiera, debería volver a comprobar la restricción, pues puede que esta vuelva a no cumplirse a causa de alguna interrupción, o que sencillamente se haya despertado por una señal y la condición no se haya verificado verdaderamente.

Si no había ningún hilo en espera en un *pthread_cond_wait(),* la señal se pierde y no tiene efecto (a diferencia de lo que ocurría con los semáforos, donde sí era relevante mantener actualizado su valor).

2 Programa

2.1 Estructura

***Figuras 2 y 3**. Código del productor y del consumidor.*

El grueso de la implementación de este ejercicio se encuentra en las funciones *producir*() y *consumir*(). En el main únicamente se realizan acciones de carácter auxiliar, tales como inicializar y destruir los mutexes y las variables de condición, o crear los hilos hijos.

En primer lugar, los consumidores habrán de repartirse las iteraciones entre ellos. Dado que habrá que consumir un total de ITEMS_BY_P * P = 20 * P elementos, el número de iteraciones por consumidor vendrá dado por el cociente de la división del anterior producto por C. Ahora bien, dado que los valores de C y P pueden escogerse al azar, no está garantizado que la división sea exacta. Se ha optado por asignar al primer consumidor las iteraciones sobrantes, de forma que este llevará a cabo un número de iteraciones igual al *cociente + resto = (ITEMS_BY_P * P / C)* + (*ITEMS_BY_P * P % C*), mientras que el resto de consumidores se encargarán solo del cociente, *ITEMS_BY_P * P / C.*

En ese punto, comienza la ejecución del bucle principal. Tanto consumidores como productores irán imprimiendo mensajes de registro sobre las acciones que toman, con la precaución de no mostrar el contenido del *buffer* ni su número de elementos, *cuenta*, mientras no están en la región crítica, pues puede que otro hilo los esté modificando y que los valores parezcan a primera vista incoherentes (aunque en realidad no es así, sencillamente estarían desactualizados). Debido a los cambios más bruscos entre hilos, se ha optado por imponer un mayor control sobre los *printf()* del programa, a fin de evitar que se corten líneas del *log* resultante. Para ello, se emplea un segundo mutex, *mutex_impr*, que restringe el acceso a las zonas de impresión. Además, se espera unos microsegundos mediante *usleep()* tras imprimir, para aumentar las probabilidades de que dé tiempo a mostrar la cadena por consola.

Tanto consumidores como productores comenzarán por solicitar acceso a la región crítica ejecutando *pthread_mutex_lock()*. Si algún otro hilo ya estaba ejecutando la región crítica, quedarán bloqueados hasta que este la libere. Si no había ninguno, continuarán con el programa. Esto garantiza la exclusión mutua entre todos los hilos (ya sean productores o consumidores).

Los productores podrán entonces preparar el elemento que insertarán. Para facilitar el estudio del buffer, cada productor introduce siempre el mismo ítem, que se calcula en función de su identificador. Además, en este punto se duerme a los productores con *sleep()*, escogiendo un entero aleatorio entre 0 y 4. Esto, junto con una llamada análoga a *sleep()* por parte de los consumidores en un punto posterior, aumentará las probabilidades de que el *buffer* llegue a vaciarse y a llenarse por completo en cada ejecución, facilitando las comprobaciones sobre el funcionamiento de las variables de condición.

No obstante, en este punto ni productores ni consumidores pueden ejecutar la región crítica directamente. Los primeros tendrán que asegurarse de que la cuenta del número de ítems sea distinta de N (llamando a *esta_buffer_lleno()*); y los segundos, de que sea distinta de 0 (llamando a *esta_buffer_vacio()*). Si no es así, deben bloquearse y abandonar temporalmente el uso del mutex, con el objetivo de que otro hilo pueda insertar/eliminar un elemento y romper la condición que les bloquea. Esto se realiza atómicamente a través de *pthread_cond_wait()*. Una vez se reciba una señal que los

despierte, procedente del *pthread_cond_signal()* de la variable de condición correspondiente (*condc* o *condp)*, volverán a comprobar la condición por si una interrupción o una señal inesperada hubieran provocado que esta vuelva a activarse. Si es así, repetirán la llamada a *pthread_cond_wait().* Nótese que retornar de esta función implica que el hilo vuelve a disponer del mutex.

En este punto, los productores ejecutan su región crítica, consistente en colocar un elemento en el *buffer* y actualizar *cuenta*, que es la verdadera variable compartida problemática. Por su parte, los consumidores retiran un elemento y también actualizan cuenta.

Justo después envían una señal a los hilos contrarios con *pthread_cond_signal().* Cabe mencionar que esto se hace sin soltar el mutex, para impedir que otro hilo entre aún en la región crítica. Si hay algún hilo dormido por una condición, se despertará a uno de ellos, que tratará de adquirir el mutex en cuanto este quede libre. Sin embargo, no se puede predecir qué hilo será despertado, ya que esta es una decisión del planificador.

Por último, se libera el mutex, de forma que el sistema operativo o bien lo deja libre (si nadie intentaba acceder a él), o bien elige un proceso bloqueado esperando por el mutex y se lo concede. En este punto, además, el consumidor muestra el ítem que eliminó y llama a *sleep()* (el punto de bloqueo es distinto al del productor para favorecer las variaciones entre ellos).

2.2 Conclusiones

Con este algoritmo, garantizamos que cualquier interrupción no afectará al control de *cuenta* o del *buffer*. El fundamento principal es que un cambio de contexto no cambiará el dueño del mutex y que sus operaciones se realizan de forma atómica.

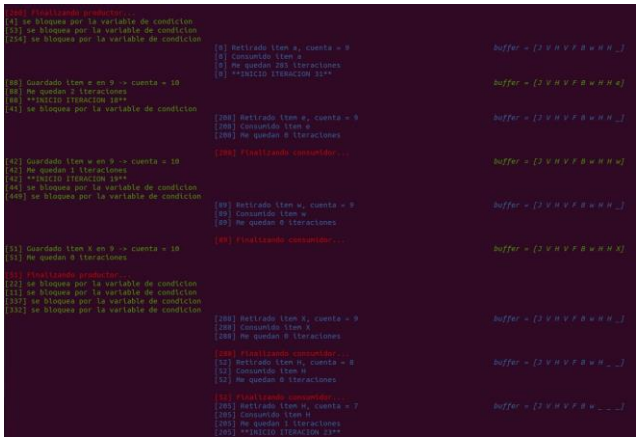

Figura 4*. Prueba con 503 productores y 407 consumidores, en la que no se observan incoherencias en el buffer.*

En todas las ejecuciones observamos que no se están produciendo carreras críticas. La variable *cuenta* se actualiza siempre de forma secuencial, y nunca quedan huecos vacíos en el *buffer*. Se puede apreciar además la importancia de los *sleep()* a la hora de generar situaciones extremas. Para desequilibrar aún más la balanza, se podría emplear *sleep()* únicamente en un hilo.

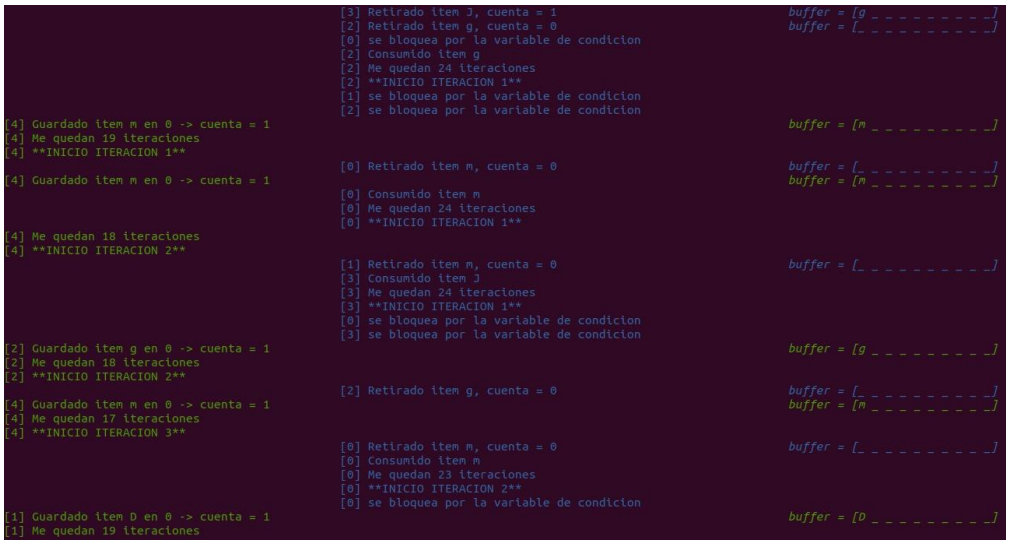

***Figura 5**. Ejecución donde el buffer se vacía repetidas veces. Obsérvese que los consumidores [0], [1] y [2] se bloquean nada más entrar en la región crítica, y deben esperar a que actúe un productor antes de continuar.*

En los casos en los que tiene lugar el vaciado o llenado del *buffer*, vemos que es imprescindible que la situación sea desbloqueada por un productor o un consumidor, respectivamente, antes de que el otro tipo de hilo pueda continuar.

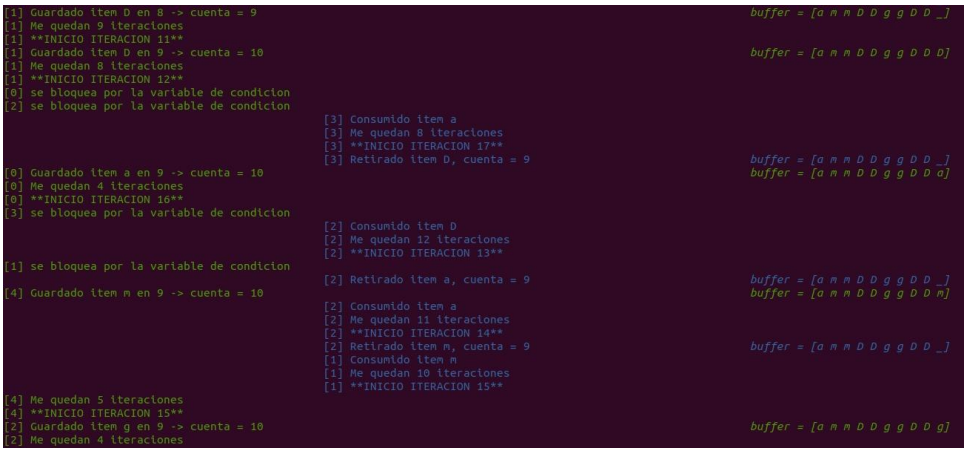

***Figura 6**. Ejecución donde el buffer se llena repetidas veces. Ahora son los productores los que quedan bloqueados. Debe ser un consumidor el que los despierte; de nada sirve que otros productores traten de acceder a la región crítica.*

Como el número de iteraciones de los consumidores es finito, tenemos también asegurado que todos ellos finalizarán eventualmente, sin que sea necesario llevar un

control adicional del número de elementos consumidos en total (ni eliminar a los hilos entre ellos una vez se alcance el máximo). Además, ningún hilo quedará bloqueado eternamente por un mutex o por una variable de condición, pues las señales de desbloqueo se encuentran dispuestas de forma simétrica, de manera que para cada bloqueo siempre hay una acción de un hilo opuesto que eventualmente desbloquea (también es importante aquí el hecho de que el número de iteraciones totales de los consumidores es igual al número de generaciones de ítems de los productores).

Estas consideraciones garantizan que el problema del productor-consumidor también es afrontable de forma segura y eficiente a través del uso de mutexes con hilos.

3 Bibliografía

[**Tanenbaum**] Tanenbaum, Andrew S. *Sistemas Operativos Modernos.* Editorial Prentice-Hall, 3ª edición (2009).

[**AfterAcademy**] The producer-consumer problem in Operating System. *After Academy*. https://afteracademy.com/blog/the-producer-consumer-problem-in-operating-system [online] última visita 28 de marzo de 2022