Sincronización de procesos con mutexes

Guillermo Arcos Salgado, Pedro Vidal Villalba

Sistemas operativos II

{guillermo.arcos, pedro.vidal.villalba}@rai.usc.es

22 de abril de 2024

I. Introducción

En el desarrollo de esta práctica se ha programado una versión del conocido problema del productor-consumidor usando hilos, mutexes y variables de condición.

Se crearán una cantidad P de productores y una cantidad C de consumidores, que no solo llevarán a cabo su tarea principal indicada por el rol, sino que tendrán que realizar una actividad secundaria: la contribución a la suma de todos los valores de las posiciones pares o impares (dependiendo si son productores o consumidores, respectivamente) de un determinado `array` (de tamaño $110 \cdot (P + C)$). Cada hilo solo podrá sumar de cada vez un valor aleatorio de entre 2 y 4 elementos.

Durante la ejecución, cada uno de los hilos debe producir o consumir un elemento del buffer, según su rol, y posteriormente intentar contribuir a la suma del array, si es que puede; independientemente de que contribuya o no, repetirá estas tareas en bucle un determinado número de veces.

Además, el `buffer` para la tarea principal es de 12 elementos y cada productor debe producir 18 elementos. Al acabar estos, debe asegurarse que los consumidores vacíen completamente la cola de items producidos. Finalmente, el programa no termina hasta que se sumen completamente todas las posiciones correspondientes del `array` para la tarea secundaria.

II. Aspectos de la implementación

En cuanto a la tarea principal, se ha optado por la construcción de un `stack` compartido por productores y consumidores con el `buffer`, un mutex y dos variables de condición (una para productores y otra para consumidores) asociados para regular el acceso al mismo, y un contador para su gestión, entre otros campos. Tanto el `buffer` como el contador, serán solo accedidos dentro de las regiones críticas protegidas, excepto en casos excepcionales pero también controlados (por ejemplo, comprobar si el `buffer` está vacío una vez hayan terminado todos los productores, así que el contador ya no puede aumentar).

Otro campo de especial interés de este `stack` es el del flag `production_finished`, para informar a los consumidores que ya no se van a producir más elementos. La modificación de este campo no produce carreras críticas debido a que solo es el hilo principal el que modifica su valor.

Cuando la producción termine, lo cual determina el hilo principal cuando detecta que todos los productores han acabado, este flag se pone a 1 y se despierta a todos los consumidores que pudieran estar dormidos dentro de la región crítica por su variable de condición, esperando a que el `buffer` deje de estar vacío, utilizando la función `pthread_cond_broadcast()`. Una vez despiertan, estos comprueban si el `buffer` está vacío nuevamente; en caso de estarlo y si la producción ya terminó, simplemente liberan el `mutex` y terminan.

En cuanto a la tarea secundaria, se tiene una estructura `Array` con los campos necesarios para llevar a cabo la contribución a la suma por parte de cada rol (cada uno tendrá su propia estructura pero comparten el mismo `array` sobre el que se realiza la operación). Uno de sus campos es el `mutex` privado para cada estructura, que permitirá evitar carreras críticas mediante el uso de la función `pthread_mutex_trylock()` para que si el consumidor o productor correspondiente intenta aportar a la suma pero ya otro se encuentra en esa región crítica, simplemente pasen a

la siguiente tarea sin quedarse bloqueados esperando.

Además, para esta tarea no será necesario el uso de variables de condición, pues no es posible que un hilo no pueda aportar a la suma una vez dentro de la región crítica, excepto que la suma ya haya terminado, en cuyo caso simplemente debe mostrar el resultado y centrarse en la tarea de producir o consumir.

III. Adición de llamadas a funciones sleep

Se añaden al código una serie de llamadas a la función `sleep` para conseguir diversos propósitos. Para ello, el programa acepta por entrada 6 argumentos, que son el tiempo en segundos para los `sleeps` en las funciones (en orden): producción de un item, inclusión de un ítem en el `buffer`, contribución al sumatorio por parte de productores, extracción de un ítem, consumición de un ítem y contribución al sumatorio por parte de consumidores.

- Para que los productores terminen su tarea principal antes que la suma de los elementos pares del `array`, se puede introducir los argumentos: 0 0 1 0 0 0. De esta forma, se hace que un productor, cuando tenga que hacer la contribución, se bloquée en la región crítica durante 1 segundo, impidiendo que otros productores puedan acceder, de forma que pasan a la tarea de producir de inmediato.
- Para que los productores terminen antes la suma de los elementos pares que su tarea principal, se puede introducir los argumentos: 0 1 0 0 0 0. De esta forma se duerme a cada productor dentro de la región crítica a la hora de producir un elemento, haciendo que los demás productores se queden esperando bloqueados por el `mutex` para producir. Ahora bien, cuando el que estaba ocupando la región crítica salga, entonces podrá entrar a la de la contribución de forma inmediata (de forma que el `trylock` no va a impedirle entrar), pues el resto se encuentran esperando por el `mutex` del `stack`. Además, en el caso de que un productor acabe de producir, puede ayudar a contribuir a la suma de forma mucho más rápida por esta misma razón.
- Para que los consumidores terminen su tarea principal antes que la suma de todos los elementos impares del `array`, se introducen los argumentos 0 0 0 0 0 1. La explicación es análoga a la del primer caso con los productores.
- El caso contrario al anterior (que consumidores acaben la suma antes que su tarea principal) es difícil que ocurra con un número tan pequeño de items a producir como 18. Esto es debido a que, si bien con productores uno de ellos podía terminar antes que el resto teniendo vía libre para dedicarse exclusivamente a la tarea de sumar, en los consumidores esto no ocurre, ya que no terminan hasta que terminen todos los productores y no esté el `buffer` vacío. No obstante, con los argumentos 0 0 0 0 1 0 si se puede conseguir que la suma acabe antes de que **algunos** de los consumidores acabe de consumir. Esto se debe al hecho de que el `trylock` de la suma para los consumidores, por lo general, no va a impedirles hacer la suma. Aún así, esta no es causa suficiente como para hacer que la suma acabe antes de que todos acaben de consumir.
- Si entendemos por cantidad de trabajo el número de iteraciones que realiza cada hilo, no podemos cambiar la cantidad de trabajo realizada por cada uno de ellos, pues la cantidad total de elementos a añadir va a ser siempre igual que la cantidad de elementos a eliminar. Ahora bien, si entendemos la cantidad de trabajo como el trabajo que tendría por delante un consumidor o productor de forma individual, contando el tiempo consumido durante las llamadas a `sleep()`, podemos hacer que un consumidor tenga más trabajo que un productor (con 0 0 0 0 1 0) manteniendo el `buffer` casi siempre lleno (pues se consumirá muy lentamente, acumulándose elementos en el `buffer`). Análogamente, podemos lograr que esté casi siempre vacío (con 1 0 0 0 0 0), y sean los productores los que tienen que realizar más trabajo por elemento que los consumidores.