Implementación del problema del productor-consumidor con mutexes y variables de condición

Francisco Javier Cardama Santiago
Universidad de Santiago de Compostela
Santiago de Compostela, España
franciscojavier.cardama@rai.usc.es

I. Introducción

En el problema del productor-consumidor a estudiar se tiene un número arbitrario de productores y consumidores trabajando sobre un **buffer**, el cual se implementa mediante una estructura FIFO circular.

El problema principal es determinar **cuál** es la región crítica y **cómo** acotarla mediante mutexes y variables de condición.

En este informe se tratará brevemente la implementación de cola FIFO en el segundo apartado. Posteriormente, se discutirán dos tipos de implementaciones en el apartado III, acotando de forma distinta la región crítica.

II. Implementación de la cola FIFO

En esta sección se tratará la implementación concreta de la *cola FIFO* circular de una forma resumido, debido a que esto resultará de interés para la posterior determinación de las regiones críticas.

La estructura FIFO circular se compone de las siguientes variables:

- **inicio:** variable de tipo entero que indica el índice anterior a la posición del primer elemento de la cola.
- **final:** variable de tipo entero que indica el índice donde se encuentra el último elemento insertado en la cola.
- **número de elementos**: variable de tipo entero que indica el número de elementos actuales que hay en la cola.

De esta forma, cada vez que se inserte un elemento en la cola, se incrementará la variable *final* en una unidad al igual que el número de producciones.

En el caso contrario, en el de la eliminación de elementos de la cola, se incrementará la variable *inicio* en una unidad, mientras que el número de producciones se decrementará en uno.

Siguiendo esta implementación, la comprobación de situaciones como 'colaVacia' o 'colaLlena' se vuelve más sencilla, en las que simplemente se mira si la variable 'número de elementos' sea igual a 0 o al tamaño de la cola, respectivamente.

III. Implementación del problema del productor-consumidor

Para realizar la implementación del problema del productor-consumidor se van a tener en cuenta dos tipos distintos de soluciones, en los que se estudiarán las posibles regiones críticas que puedan surgir.

III-A. Implementación con una única región crítica

En esta solución se va a interpretar que existe una única región crítica **común** a *ambos tipos de procesos* (productor y consumidores), la cual será el propio buffer. Esto quiere decir que solo podrá trabajar sobre el buffer (producir o consumir) un único proceso al mismo tiempo.

Por lo tanto, para poder realizar esta solución será necesaria una variable de tipo *mutex* para el acceso a la región crítica y dos variables de condición, una para cada tipo de proceso, para gestionar cuando un productor o consumidor se deben bloquear / despertar.

```
1 void productor(HiloProductor* hilo){
2   int i;
3   int item;
4
5   for(i = 0; i < hilo->numProducciones; i++){
```

```
    item = producir();

    pthread_mutex_lock(&mutexRegion);

    while(colaLlena(buffer)){
      pthread_cond_wait(&condProductor, &mutexRegion);
    }

    insertarBufferTime(&buffer, item, hilo->tiempo);

    if(numElementos(buffer) == 1){
      pthread_cond_signal(&condConsumidor);
    }

    pthread_mutex_unlock(&mutexRegion);
  }

  pthread_exit(EXIT_SUCCESS);
}
```

Función del productor en C.

```
void consumidor(HiloConsumidor* hilo){
  int item;

  while(1){

    pthread_mutex_lock(&mutexRegion);

    if(obtenerProducciones(buffer) == 0){
      pthread_cond_broadcast(&condConsumidor);
      pthread_mutex_unlock(&mutexRegion);
      pthread_exit(EXIT_SUCCESS);
    }

    while(colaVacia(buffer)){
      pthread_cond_wait(&condConsumidor, &mutexRegion);
      if(obtenerProducciones(buffer) == 0){
        pthread_mutex_unlock(&mutexRegion);
        pthread_exit(EXIT_SUCCESS);
      }
    }

    item = sacarBufferTime(&buffer, hilo->tiempo);
    incrementarProducciones(&buffer, -1);

    imprimirBuffer(buffer);

    if(numElementos(buffer) == tamano(buffer) - 1){
      pthread_cond_signal(&condProductor);
    }
    pthread_mutex_unlock(&mutexRegion);
  }
}
```

Función del consumidor en C.

En esta implementación, se puede observar que tanto el productor como el consumidor están buscando el acceso a la misma región crítica y, esto, tiene las siguientes implicaciones.

1. Cuando el tipo de proceso actual no puede continuar su ejecución, lleva a cabo un `pthread_cond_wait`. Esto se puede deber a que o bien la cola está **llena** o bien está **vacía**, por lo tanto no tiene sentido que se *libere la región crítica* y entre otro proceso del mismo tipo, debido a que se volverá a quedar bloqueado. Esta es una de las primeras implicaciones de utilizar una única región crítica **común** a *productores* y *consumidores*.
2. Debido a la anterior implicación, puede haber **más de un proceso** del mismo tipo cuando sólo se puede ejecutar uno a la vez.
3. Que haya varios procesos del mismo tipo bloqueados implica que cuando se desbloqueen puede que el resto de procesos ya hayan hecho su trabajo y su única solución sea **finalizar su ejecución**.

4. Como sólo se ejecuta un `pthread_cond_signal` en cada cambio de estado (de **vacía** a **no vacía** o de **llena** a **no llena**), pudiendo darse el caso de que la gran parte del trabajo esté siendo realizada por **un único proceso** de cada tipo.

En resumen, el primer problema que se da con esta implementación es que se pueden tener a procesos de un **mismo tipo** *bloqueados* mientras otro proceso del mismo tipo tiene la región crítica, cuando los anteriores fueron bloqueados debido a que no podían continuar su ejecución.

La segunda implicación es que sólo puede estar trabajando un único proceso en la región crítica, indiferentemente de que sea un consumidor o un productor, ya que la región crítica es compartida. Esto implica que el tiempo de trabajo se desaproveche considerablemente.

III-B. Implementación mediante tres regiones críticas

En esta implementación, se procederá a dividir la **región crítica común** en regiones críticas específicas para cada proceso, para evitar las implicaciones de la anterior versión.

En vez de considerar como una única región el **buffer**, esta se divide en tres nuevas, una que bloquee el acceso a la variable **'inicio'**, otra a la variable **'final'** y una última la variable **'número de elementos'**.

Esto es interesante debido a que los productores solo 'insertan' en el Buffer, por lo tanto, solo incrementan la variable 'final' del buffer y en ninguna situación se ven influenciados por el valor que tome la variable 'inicio'.

En el caso de los consumidores pasa lo contrario, estos solo 'sacan' elementos del buffer y, por lo tanto, solo modifican la variable 'inicio' del buffer sin verse afectados por las modificaciones de la variable 'final'.

Por lo tanto de esta forma se consigue que cada tipo de proceso tenga una **región crítica exclusiva**, pudiendo haber un productor y un consumidor trabajando al mismo tiempo.

La única zona donde interfiere una variable que puede ser modificada por cualquier tipo de proceso, es en las comprobaciones de tipo 'colaLlena' y 'colaVacia' donde se utiliza la variable *'número de elementos'*. Esta situación no es especialmente peligrosa, debido a que para que se de la condición de 'colaVacia' un consumidor tiene que haber decrementado la variable y al contrario en la situación de 'colaLlena', un productor tendría que haberla incrementado y al tener una región crítica para cada tipo de proceso este problema se minimiza en gran medida.

Aún así, hay situaciones en las que se puede producir una carrera crítica, por ejemplo en el caso en que un consumidor compruebe que la cola está vacía, esté a punto de ejecutar su *pthread_cond_wait* y justo le llegue ya la señal de que ya no está vacía, perdiéndose de esta forma el *pthread_cond_signal* y quedándose el proceso bloqueado hasta la siguiente señal. Por lo tanto, como es necesario que con la implementación aportada se solucione el problema en todas sus situaciones, se añade una nueva región crítica para la variable **'número de elementos'**.

```
1  void productor(HiloProductor* hilo){
2    int i;
3    int item;
4
5    for(i = 0; i < hilo->numProducciones; i++){
6
7      item = producir();
8
9      pthread_mutex_lock(&mutexProd);
10
11     pthread_mutex_lock(&mutexDespertar);
12
13     while(colaLlena(buffer)){
14       pthread_cond_wait(&condDespertar, &mutexDespertar);
15     }
16
17     pthread_mutex_unlock(&mutexDespertar);
18
19     insertarBufferTime(&buffer, item, hilo->tiempo);
20
21     pthread_mutex_lock(&mutexDespertar);
22
23     if(numElementos(buffer) == 1){
24       pthread_cond_signal(&condDespertar);
25     }
```

```
26
27      pthread_mutex_unlock(&mutexDespertar);
28
29      pthread_mutex_unlock(&mutexProd);
30    }
31
32    pthread_exit(EXIT_SUCCESS);
33  }
```

Función del productor en C.

```
1  void consumidor(HiloConsumidor* hilo){
2    int item;
3
4    while(1){
5
6      pthread_mutex_lock(&mutexConsum);
7
8      if(obtenerProducciones(buffer) == 0){
9        pthread_mutex_unlock(&mutexConsum);
10       pthread_exit(EXIT_SUCCESS);
11     }
12
13     pthread_mutex_lock(&mutexDespertar);
14     while(colaVacia(buffer)){
15       pthread_cond_wait(&condDespertar, &mutexDespertar);
16     }
17     pthread_mutex_unlock(&mutexDespertar);
18
19     item = sacarBufferTime(&buffer, hilo->tiempo);
20     incrementarProducciones(&buffer, -1);
21
22     pthread_mutex_lock(&mutexDespertar);
23     if(numElementos(buffer) == tamano(buffer) - 1){
24       pthread_cond_signal(&condDespertar);
25     }
26     pthread_mutex_unlock(&mutexDespertar);
27
28     pthread_mutex_unlock(&mutexConsum);
29   }
30 }
```

Función del consumidor en C.

Para esta implementación serán necesarias tres variables mutex, una para la región crítica de los productores, otra para la de los consumidores y por última una para el acceso a la variable *'número de elementos'*. Por otra parte, sólo será necesaria una variable de condición, debido a que los procesos se bloquean siempre dentro de la región crítica del 'número de elementos' y nunca se va a dar el caso de que haya un **consumidor** y un **productor** bloqueados a la vez, no como en las situaciones que se daban en la implementación anterior.

Esto se debe a que la región crítica que se libera al realizar el *pthread_cond_wait* es la del 'número de elementos', la cual se bloquea dentro de la región crítica del tipo de proceso concreto, y esta va a seguir perteneciendo al proceso que se queda bloqueado, de esta forma se solucionan los problemas de la anterior versión cuando, por ejemplo, un productor se bloqueaba se diese la posibilidad de que entrasen más productores y se bloqueasen. Por lo tanto, con esta nueva implementación se consigue que cuando un proceso se quede bloqueado, este, sea despertado por el tipo de proceso opuesto inmediatemente en el momento en el que se da la condición de que pueda seguir trabajando, de esta forma se consigue poder usar una única variable de condición, debido a que si el buffer **no está vacío** no habrá ningún consumidor bloqueado y si el buffer **no está lleno** no habrá ningún productor bloqueado.

Todas estas mejoras surgen gracias al estar en la situación de tener una región crítica para cada tipo de proceso, debido a que en la anterior versión era necesario desbloquear la región crítica común para que pudiese entrar un proceso del tipo opuesto para despertarlo, ya que, si no se hace se daría un interbloqueo. En esta situación es indiferente que un consumidor se quede esperando a un productor sin liberar su región crítica, ya que el productor puede trabajar indiferentemente del estado de la región crítica del consumidor y viceversa.

IV. CONCLUSIONES

Concluyendo, la segunda implementación tratada da lugar a varias ventajas con respecto a la primera implementación las cuales se indican a continuación.

Primera implementación.

Ventajas

- El código es más fácil de entender, dando lugar a que se produzcan menos errores a la hora de implementarlo.

Desventajas

- Solo puede estar trabajando a la vez un único proceso, indifirentemente de su tipo.
- Debido a que puede que haya procesos del mismo tipo bloqueados y otro ejecutándose, se puede dar el caso de que siempre trabajen los mismos procesos
- Se puede dar la situación de que haya varios procesos del mismo que se encuentren bloqueados cuando podrían continuar su ejecución.

Segunda implementación.

Ventajas

- Puede haber un proceso de cada tipo trabajando a la vez sobre el buffer.
- La competición para trabajar sobre el buffer solo se gestiona en quien toma el mutex antes y en ningún caso depende de cuando se haya bloqueado o cuántos haya bloqueados.
- En ningún caso habrá procesos del mismo tipo bloqueados en caso de que las condiciones para que su ejecución continúe se cumplan.

Desventajas

- El código es más complicado de entender, debido a que hay que comprobar que no ocurran carreras críticas, dando lugar a un mayor número de errores a la hora de la implementación.

REFERENCIAS

[1] Andrew S. Tanenbaum, "Sistemas Operativos Modernos," Tercera Edición, Prentice Hall, 2008.