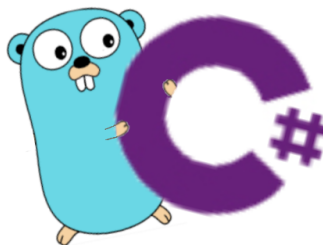


Lenguajes de Programación

Concurrencia



LEANDRO RODRÍQUEZ LLOSA
LAURA V. RIERA PÉREZ
MARCOS M. TIRADOR DEL RIEGO

Tercer año. Ciencias de la Computación.
Facultad de Matemática y Computación, Universidad de La Habana, Cuba

Noviembre 2022

I. CONCURRENCIA Y PARALELISMO

La *concurrencia* es la ejecución simultánea de varias hebras¹, pero esta simultaneidad puede ser solo en apariencia. Los procesos tienen lugar en el mismo tiempo, pero la ejecución de todos ellos no ocurre en el mismo instante. En un momento dado solo se ejecuta un programa, y este lo hace de forma secuencial en un tiempo limitado establecido para realizar sus operaciones. Si este no termina su ejecución dentro de su tiempo, es puesto en espera, dándole paso al próximo, y es resumido una vez vuelva a tocar su tiempo. Sea T el tiempo total de ejecución de dos programas concurrentes P_1 y P_2 :

Concurrencia

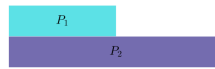


$$T = T(P_1) + T(P_2)$$

En el *paralelismo* la ejecución ocurre en el mismo instante físico, los cálculos se realizan de forma verdaderamente simultánea. Para maximizar el uso de múltiples procesadores o núcleos, presentes en las CPU modernas, el procesamiento en paralelo dividirá el trabajo entre varios subprocesos, cada uno de los cuales puede ejecutarse de forma independiente en un núcleo diferente. Paralelismo implica concurrencia, pero no se cumple el recíproco. Sea ahora T el tiempo total de ejecución de dos programas paralelos P_1 y P_2 :

¹Una hebra es una ejecución secuencial de instrucciones.

Paralelismo



$$T = \max(T(P_1), T(P_2))$$

En la programación concurrente ocurre con frecuencia que las hebras que se ejecuten necesiten sincronizar e intercambiar información en algún momento, lo cual se hace usualmente a través de memoria compartida. Esto puede causar problemas dado que varios procesos estarán realizando modificaciones concurrentemente sobre la misma memoria. Múltiples hebras se encuentran en una *condición de carrera* si el resultado de su ejecución depende del orden en que se ejecutan las instrucciones que componen cada hebra.

Se denomina *sección crítica* a la porción de código de una hebra en la que se accede a un recurso compartido y que puede entrar en una condición de carrera, por lo que no debe ser accedido por más de un proceso o hilo en ejecución a la vez. Se necesita un mecanismo de sincronización en la entrada y salida de la sección crítica para asegurar la utilización exclusiva del recurso. Los recursos destinados a lograr este comportamiento se denominan de exclusión mutua o *mutex*. Los más comunes son los candados, monitores y semáforos.

I. Locks en C#

Los candados garantizan acceso exclusivo a un recurso compartido. La sincronización se logra poniéndole el candado a una variable. En C# esto se logra mediante la palabra reservada `lock`.

El candado de exclusión mutua es adquirido para un objeto dado por `lock`, se ejecuta el bloque de código dentro de su cuerpo y luego se libera el candado. Mientras un hilo mantenga un candado, este puede volver a adquirirlo y liberarlo. Cualquier otro subproceso no puede adquirir el candado y debe esperar hasta que este sea liberado. A continuación se muestra la forma de declarar un candado:

```
lock (x) //x is an expression of reference type
{
    // Block of code...
}
```

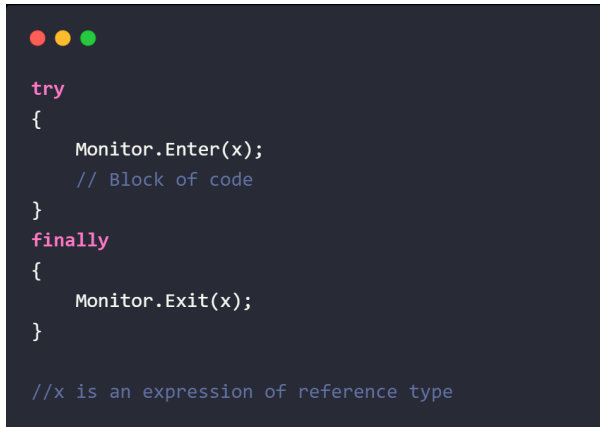
`lock` controla toda una sección sin dejar libertad para adquirir o liberar un recurso basado en una lógica más propia del problema a solucionar.

II. Monitors en C#

Los monitores permiten sincronizar el acceso a una región de código tomando y liberando un bloqueo en un objeto en particular, de manera más fluida que en los candados. En C# se dispone de la clase `System.Threading.Monitor` para este propósito.

Un monitor se asocia a un objeto bajo demanda y se puede llamar directamente desde cualquier contexto. No se puede crear una instancia de esta clase. Sus métodos son todos estáticos y a cada uno se le pasa el objeto sincronizado que controla el acceso a la sección crítica.

Posee los métodos `Enter`, que adquiere un candado para un objeto y marca el comienzo de una sección crítica; y `Exit` que libera el bloqueo de un objeto, marcando el final de la sección crítica protegida por el objeto bloqueado. Se puede lograr entonces el comportamiento de un bloque `lock` mediante un bloque `try-finally` que utilice los métodos `Enter` y `Exit` como se muestra a continuación:



```
try
{
    Monitor.Enter(x);
    // Block of code
}
finally
{
    Monitor.Exit(x);
}

//x is an expression of reference type
```

Además cuenta con los métodos `Wait` que libera el bloqueo de un objeto para permitir que otros subprocesos bloqueen y accedan al mismo; el subproceso que llama a este método espera mientras otro subproceso accede al objeto; `Pulse` y `PulseAll`, quienes envían una señal a los subprocesos en espera (uno y todos respectivamente), notificándoles que el estado del objeto bloqueado ha cambiado y que el propietario del bloqueo está listo para liberarlo. El subproceso en espera se coloca en la cola de subprocesos listos del objeto para que eventualmente pueda recibir el bloqueo para el mismo.

III. Semaphores en C#

Los semáforos limitan la cantidad de subprocesos que pueden acceder a un recurso o conjunto de recursos al mismo tiempo. En C# pueden ser encontrados en la clase `System.Threading.Semaphore`.

Un semáforo cuenta con dos propiedades fundamentales: `Count`, que indica el número de hilos que pueden ingresar al semáforo en este momento; y `InitialCount`, que indica la cantidad máxima de hilos que pueden ingresar al semáforo. Los subprocesos ingresan al semáforo llamando al método `WaitOne()`, y liberan el semáforo llamando al método `Release()`.

El `Count` en un semáforo se reduce cada vez que un subproceso ingresa al semáforo y se incrementa cuando un subproceso libera el semáforo. Cuando el `Count` es cero, las solicitudes posteriores se bloquean hasta que otros subprocesos liberan el semáforo; no hay un orden predeterminado en el que los subprocesos bloqueados entren en el semáforo. Cuando todos los subprocesos han liberado el semáforo, el `Count` estará en el valor máximo especificado cuando se creó el semáforo (`InitialCount`).

Un subproceso puede ingresar el semáforo varias veces llamando al método `WaitOne()` repetidamente. Para liberar algunas o todas estas entradas, el subproceso puede llamar a la sobrecarga del método `Release()` sin parámetros varias veces, o puede llamar a la sobrecarga del método `Release(int)` que especifica la cantidad de entradas que se liberarán. No se aplica la identidad del subproceso en las llamadas a `WaitOne()` o `Release()`, por lo que es responsabilidad

del programador asegurarse de que los subprocesos no liberen el semáforo más veces de las requeridas.

Un semáforo puede ser de dos tipos: local o nombrado del sistema. Un semáforo local existe solo dentro de su proceso, pudiendo ser utilizado por cualquier subproceso en este que tenga una referencia al mismo. Por otro lado, si se crea un semáforo utilizando un constructor que acepta un nombre, se asocia con un semáforo del sistema operativo de ese nombre, siendo visible en todo el sistema operativo y pudiendo utilizarse para sincronizar las actividades de los procesos. Se puede usar el método `OpenExisting` para abrir un semáforo nombrado del sistema existente.

En la figura siguiente se muestra una implementación de la clase Semaphore en C# usando la clase Monitor:

```
public class MySemaphore
{
    // Reference object for locks
    private static object LockObj = new object();

    // Number of threads that can enter the semaphore currently
    public int Count { get; set; }
    // Maximum number of threads that can enter the semaphore
    public int Capacity { get; set; }
    // Name of the semaphore (nullable)
    public string? Name { get; set; }

    public MySemaphore(int initial_entries, int maximum_entries)
    {
        if (maximum_entries < initial_entries)
            throw new System.Exception("Number of threads in semaphore cannot be greater than its capacity.");

        this.Count = initial_entries;
        this.Capacity = maximum_entries;
    }

    public MySemaphore(int initial_entries, int maximum_entries, string name)
    {
        if (maximum_entries < initial_entries)
            throw new System.Exception("Number of threads in semaphore cannot be greater than its capacity.");

        this.Count = initial_entries;
        this.Capacity = maximum_entries;
        this.Name = name;
    }

    // A thread enters the semaphore
    public void WaitOne()
    {
        lock (LockObj)
        {
            if (this.Count == 0)
            {
                Console.WriteLine("Semaphore is full, waiting...");
                System.Threading.Monitor.Wait(LockObj);
            }
            this.Count--;
        }
    }

    // An N quantity of threads release the semaphore
    public int Release(int N = 1)
    {
        lock (LockObj)
        {
            this.Count += N;
            if (this.Count > this.Capacity)
            {
                throw new Exception("Semaphore is empty, nothing to release.");
            }
            System.Threading.Monitor.PulseAll(LockObj);
        }
        return this.Count - N;
    }
}
```

iv. Barriers en C#

Permite que múltiples tareas trabajen de manera cooperativa en un algoritmo en paralelo a través de múltiples fases. Ninguna hebra puede comenzar una nueva fase del algoritmo hasta que las restantes hebras no hayan alcanzado la barrera (todas hallan completado la fase anterior).

El constructor más simple de la clase toma por argumento un entero, que representará el máximo número de participantes (hilos) que pueden estar esperando por la barrera en cada fase del algoritmo. Los métodos `AddParticipant()` y `RemoveParticipant()` permiten aumentar en una unidad la cantidad la cantidad máxima de participantes.

El método fundamental de la clase es `SignalAndWait()`. Este es invocado por una hebra cuando llega a un punto en que está lista para moverse a la siguiente fase. Este advierte a la instancia de *Barrier* de que un nuevo participante ha llegado a la barrera. Cuando hayan llegado una cantidad de participantes igual al número máximo entonces la barrera entra en la post-phase. Aquí es donde incrementa el número de la fase, y le da la señal a todos los participantes de que pueden entrar en la nueva fase. Adicionalmente mediante el otro constructor de *Barrier* se puede establecer una acción que se ejecutará en cada post-phase.

El último método relevante que posee es el `Dispose()` que permite liberar los recursos usados por la barrera cuando esta no se necesite usar más. Si se intenta usar la barrera después de que esta haya sido "disposed", lanzará una excepción del tipo *ObjectDisposedException*. Los otros tipos de excepciones lanzadas por los métodos de la clase son *InvalidOperationException* y *BarrierPostPhaseException*. La primera de estas ocurre cuando en la barrera están esperando más participantes que el máximo permitido o cuando se invoca alguno de sus métodos estando en la post-phase. La segunda ocurre cuando la acción establecida a ejecutar entre fases lanza una excepción. Esta es capturada y envuelta (wrapped) en una excepción del tipo mencionada y relanzada en cada uno de los hilos (participantes).

Para implementar la clase *Barrier* se hizo uso de la clase *Monitor*. Para ello en el cuerpo del método `SignalAndWait()` se usa `Monitor.Wait` para hacer que la hebra que lo ejecuta espere a que el resto llegue a la barrera. Cada vez que llega una hebra nueva se incrementa la cantidad de participantes que han llegado. Cuando se alcanza el máximo lo primero que se ejecuta la acción prefijada (si alguna), desde la última hebra en llegar. Entonces se incrementa el número de la fase y se invoca `Monitor.PulseAll` para que todas las hebras reanuden su ejecución. Sin embargo, se tuvo mucha sutileza para manejar la mayor cantidad de casos posibles en que pudieran ocurrir errores de sincronización, como podría ser la muerte por inanición. Se maneja cada situación de la manera más correcta posible según criterio de los autores, y se lanza la excepción adecuada en cada caso, según lo descrito sobre estas anteriormente.

v. Countdowns en C#

Los countdowns permiten sincronizar ejecuciones de múltiples hebras mediante cuentas regresivas. Dado el número de eventos que tienen que ocurrir para realizar una acción, se notifica cada vez que termine uno de estos eventos disminuyendo el contador, y se emite una señal cuando la cuenta llegue a cero. En C# esta funcionalidad se implementa con la clase `System.Threading.CountdownEvent`.

Cuenta con distintas propiedades: `CurrentCount` que obtiene el número de señales restantes necesarias para establecer el evento, `InitialCount` que obtiene el número de señales requeridas inicialmente para configurar el evento, y `IsSet` que indica si el recuento actual del objeto `CountdownEvent` ha llegado a cero, para emitir la señal.

Sus métodos fundamentales son `AddCount(Int32)` que incrementa el recuento actual de

`CountdownEvent` en un valor especificado, `Reset(Int32)` que restablece la propiedad `InitialCount` a un valor especificado, `Signal(Int32)` quien registra múltiples señales con `CountdownEvent`, disminuyendo el valor de `CurrentCount` en la cantidad especificada, y `Wait()`, encargado de bloquear el subproceso actual hasta que se establece `CountdownEvent`.

En la figura siguiente se muestra una implementación de la clase `CountdownEvent` en C# usando la clase `Monitor`:

```
public class MyCountdownEvent
{
    // Number of remaining signals to set the event
    private int counter;
    // Initial number of signals needed to set the event
    private int init_counter;
    // Indicates if the number of remaining signals has reached zero
    private bool counter_equals_zero;

    // Reference object for locks
    private static object LockObj = new object();

    public MyCountdownEvent(int init_count)
    {
        this.init_counter = init_count;
        this.counter = init_count;

        if (init_count == 0)
            this.counter_equals_zero = true;
        else
            this.counter_equals_zero = false;
    }

    // Propierties (readonly)
    public int CurrentCount => counter;
    public int InitialCount => init_counter;
    public bool IsSet => counter_equals_zero;

    // Increments the CountdownEvent's current count by N.
    public void AddCount(int N = 1)
    {
        lock (LockObj)
            this.counter += N;
    }

    // Decrements the CountdownEvent's current count by N, and indicates weather the event was set or not
    public bool Signal(int N = 1)
    {
        lock (LockObj)
        {
            this.counter -= N;

            if (this.counter == 0)
            {
                this.counter_equals_zero = true;
                Monitor.PulseAll(LockObj);
            }
        }
        return this.counter_equals_zero;
    }

    // Sets the CountdownEvent's current count to N.
    public void Reset(int N = -1)
    {
        lock (LockObj)
        {
            if (N < 0)
                this.counter = this.InitialCount;
            else
                this.counter = N;

            if (this.counter == 0)
            {
                this.counter_equals_zero = true;
                Monitor.PulseAll(LockObj);
            }
        }
    }
}
```


vi. Propuesta en Go para la sincronización en la concurrencia

vii. Problema de los 5 Filósofos

El problema presentado en esta sección es un problema clásico para ejemplificar los problemas de sincronización, el cual fue tomado de ([?, Epígrafe 17.7.2.1]).

Descripción: Cinco filósofos se pasan la vida comiendo y pensando, sentados en una mesa circular con un tazón común en el centro que contiene “infinitos” espaguetis y con un tenedor colocado entre cada par de filósofos. Cada filósofo alterna a su antojo el comer y el pensar, sin comunicarse con los demás. Para comer, cada filósofo necesita de los dos tenedores (el que tiene a su izquierda y el que tiene a su derecha) por lo que puede ocurrir que tome uno de los dos tenedores y tenga que esperar por el otro que está ocupado por su vecino. Al terminar de comer el filósofo debe colocar de vuelta los tenedores sobre la mesa. Puede ocurrir una situación en que cada filósofo tenga ocupado el tenedor a su izquierda y al querer tomar el tenedor a su derecha este esté ocupado por el filósofo a su derecha quien también está en la misma situación y, así, de modo circular. Entonces todos están bloqueados sin poder comer por falta de un tenedor.

Problema: Diseñar un método con el cual los filósofos puedan alternar entre comer y pensar eternamente, sin previo conocimiento de cuándo los otros filósofos desean comer y, claro, sin tener que aumentar la cantidad de tenedores porque es obvio que si cada uno tiene dos tenedores no habría problema.

vii.1. Solución en C#

En [1, Epígrafe 17.7.2.2] podemos encontrar una solución parcial al problema. La misma consiste en usar un mediador (mesero) que determine en cada momento quién puede o no usar los tenedores según los soliciten. De esta forma se elimina la posibilidad de interbloqueo y por tanto que el programa quede esperando infinitamente. Sin embargo, se dice que es una solución parcial ya que no elimina la posibilidad de muerte por inanición. La propuesta dada consiste en que solo un filósofo pueda pedirle al mesero ambos tenedores, en cada momento. Sin embargo la decisión de qué filósofo es atendido primero queda en manos del sistema operativo, y no sabemos como se comporta este, pudiendo darse el caso en que deje a uno de los filósofos esperando sin ser atendido nunca, o relativamente pocas veces.

Para evitar ese problema en este trabajo se propuso contar con una cola en la que los filósofos se van colocando cuando necesitan comer. La idea es que cuando un filósofo decida comer este revise la cola a ver si no hay nadie que quiera alguno de sus dos mismos tenedores (y que tendría por ende prioridad), así como revisa a quienes están comiendo para ver que no estén ya ocupados. En caso de que coger los tenedores sea consistente con lo anterior, este los toma y se sienta a comer. Si no, se coloca al final de la línea. Por supuesto, mientras un filósofo está comiendo sus tenedores están bloqueados para el resto. Cuando un filósofo termina de comer este avisa al mesero quién revisa la cola por si alguien estaba esperando por alguno de esos tenedores. La cola se revisa por orden de llegada (FIFO) y de este modo se garantiza que come primero siempre el que primero decidió comer.

Para implementar esto usaremos la clase `Monitor`, que interactuará sobre los objetos de tipo tenedor, para bloquearlos y desbloquearlos según se estén usando. Entonces cuando un filósofo debe comenzar a comer, se revisa si los tenedores correspondientes no están bloqueado (en uso) y si además la cantidad de gente en la cola esperando por cada uno de estos es cero. Si estas condiciones se cumplen se puede proceder a comer sin problemas, ya que no se afecta a más nadie. Se bloquean entonces los dos tenedores que se usarán. Si este no es el caso entonces esta instancia de filósofo se adiciona a la cola, y además se bloquea la ejecución de su hebra mediante

`Monitor.Wait` , mientras espera una señal de `Monitor` correspondiente al objeto `Ticket` (campo de la clase `Philosopher`).

Por otro lado tenemos en la hebra principal (disjunta de las 5 correspondientes a los filósofos) un bucle infinito que en cada iteración revisa la cola para darle los tenedores libres al que primero los necesite, haciendo `Monitor.Pulse` a su `Ticket` correspondiente. Sin embargo, si se hace esto consecutivamente se caería en un problema de interbloqueo. Además, es innecesario revisar la cola dos veces seguidas si no se han liberado tenedores nuevos. Entonces la ejecución de esta hebra queda bloqueada nuevamente por la clase `Monitor` , aplicada sobre el objeto `Waiter` , hasta que algún filósofo termine de comer y de la señal de desbloqueo.

vii.2. Go

vii.3. Comparación

REFERENCIAS

- [1] Miguel Katrib. *Empezar a programar. Un enfoque multiparadigma con C#*. Editorial UH. 2020. La Habana.
- [2] [Microsoft Learn: System.Threading Namespace in .NET](#)