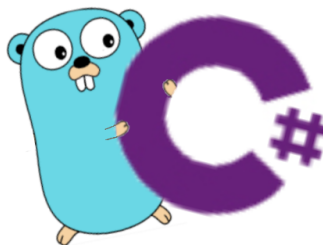


Lenguajes de Programación

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

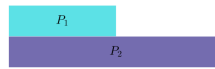


$$T = T(P_1) + T(P_2)$$

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...
}
```

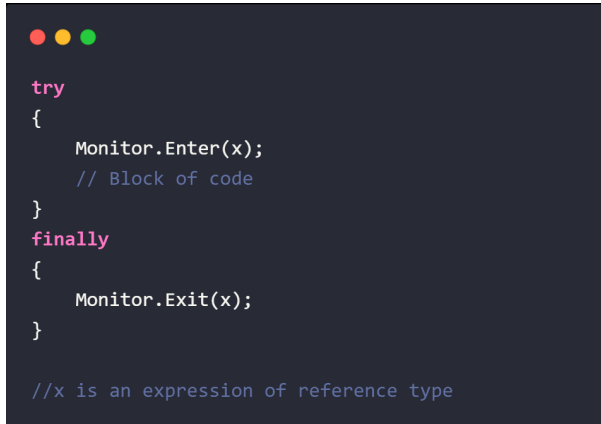
`lock` controla toda una sección sin dejar libertad para adquirir o liberar un recurso basado en una lógica más propia del problema a solucionar.

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.

Esto es importante pues si, por ejemplo, un semáforo tiene un recuento máximo de dos y tanto el hilo A como el hilo B entran en el semáforo, si un error de programación en el subproceso B hace que llame a `Release` dos veces, ambas llamadas se realizan correctamente. El conteo en el semáforo estará lleno, y cuando el subproceso A eventualmente llame a `Release`, se lanzará una excepción `SemaphoreFullException`.

Decidir si
dejo el ejem-
plo para la
expo

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.

En esta sincronización de barrera, tenemos varios subprocesos que trabajan en un solo algoritmo. El algoritmo funciona en fases. Todos los subprocesos deben completar la fase 1 y luego pueden continuar con la fase 2. Hasta que todos los subprocesos no completen la fase 1, todos los subprocesos deben esperar a que todos los subprocesos lleguen a la fase 1.

v. Countdowns en C#

Representa una primitiva de sincronización que emite una señal cuando su cuenta llega a cero.

```
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. Solución a los filósofos

vii.1. Go

vii.2. C#

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](#)