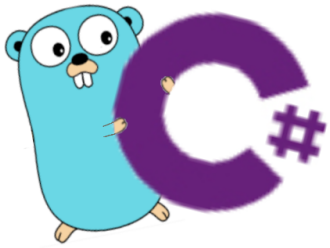
Lenguajes de Programación

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 hebras (una hebra es una ejecución secuencial de instrucciones) pero esta simultaneidad puede ser solo en apariencia. Cada programa se ejecuta de forma secuencial, el sistema operativo es el encargado de administrar y distribuir el tiempo que el procesador le da a cada tarea. La vida de los procesos comparten el mismo tiempo, pero la ejecución de todos ellos no ocurre en el mismo instante.

Se comparte el tiempo de ejecución, dándole a cada proceso una suma de tiempo limitada para ejecutarse. Solo un proceso se ejecuta en un instante dado, y si no completa su operación dentro de su tiempo, el proceso se coloca en pausa, y se le da lugar a otro proceso para iniciar o resumirse, hasta que le vuelva a tocar su tiempo.

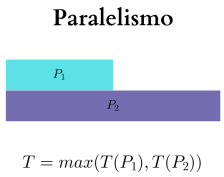
Sea T el tiempo total de ejecución de dos programas concurrentes P_1 y P_2:

Concurrencia

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

En la computación paralela, la ejecución ocurre en el mismo instante físico, los cálculos se realizan de forma verdaderamente simulánea. 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.

Sea T el tiempo total de ejecución de dos programas paralelos P_1 y P_2:



Paralelismo implica concurrencia, pero no se cumple el recíproco.

En la programación concurrente es común wue las hebras que se ejecuten necesiten sincronizar e intercambiar información enn algún momento, lo cual se hace usualmente a través de memoria compartida, pudiendo dar lugar a varios problemas.

I. Monitors en C#

Proporcionan un mecanismo que sincroniza el acceso a los objetos.

Los monitores garantizan que, a la vez, solo se estén ejecutando dentro de un monitor una sola hebra.

La clase Monitor permite sincronizar el acceso a una región de código tomando y liberando un bloqueo en un objeto determinado llamando a los métodos Monitor.Enter, Monitor.TryEnter y Monitor.Exit . Los bloqueos de objeto proporcionan la capacidad de restringir el acceso a un bloque de código, normalmente denominado sección crítica. Aunque un subproceso posee el bloqueo de un objeto, ningún otro subproceso puede adquirir ese bloqueo. También puede usar la Monitor clase para asegurarse de que ningún otro subproceso pueda acceder a una sección del código de aplicación que ejecuta el propietario del bloqueo, a menos que el otro subproceso ejecute el código mediante un objeto bloqueado diferente.

II. Semáforos en C#

Un semáforo limita la cantidad de subprocesos que pueden acceder a un recurso o conjunto de recursos al mismo tiempo. En C# puede ser encontrado en la clase `System.Threading.Semaphore`.

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. Por ejemplo, si un semáforo tiene un recuento máximo de dos y que

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.

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. Cada objeto Semaphore es un semáforo local independiente. Por otro lado, si se crea un objeto Semaphore 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 puediendo utilizarse para sincronizar las actividades de los procesos. Se puede usar el método OpenExisting para abrir un semáforo nombrado del sistema existente.

```
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;
    }
}
```

III. 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.

IV. 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)
            {
```