Tercera Prueba de Programación
"Simulando Paralelismo"

Descripción del Problema

El problema consiste en implementar un mecanismo de distribución del tiempo de CPU para simular el paralelismo en computadoras que tienen un único CPU real.

Suponemos que el CPU real trabaja en unidades de tiempo que denominaremos ciclos de CPU. Una **tarea** será un programa a ejecutar y que necesita para ello una cierta cantidad de ciclos de CPU (específicas de cada tarea). Ver más abajo el código del tipo `Tarea`.

Un **organizador** es un programa que simula la existencia de varios CPU virtuales (ver definición del tipo `CPU`) asignando una cantidad de ciclos de CPU real a cada CPU virtual. A un organizador se le pide que cree un CPU virtual mediante el método `CPU CreaCPU(long ciclos)`.

Cuando se le pide al organizador añadir una tarea para ejecutar debe indicársele en cuál de los CPU virtuales se quiere ejecutar. Las tareas a ejecutar en un mismo CPU virtual se ejecutan en el orden en que fueron añadidas las mismas. Método `void AdicionaTarea(CPU cpu, Tarea tarea)`.

El organizador ejecutará las tareas según el orden en que fueron creados los CPUs virtuales. Cuando un CPU virtual consuma los ciclos de CPU real que le corresponden el organizador pasará al siguiente CPU virtual continuando de manera circular. De este modo una tarea ejecutándose en un CPU virtual puede ver interrumpida su ejecución si el CPU virtual consumió la cantidad de ciclos de CPU real que puede ejecutar en cada turno. El CPU virtual reanudará la ejecución de la tarea cuando le vuelva a llegar su turno.

Se le puede pedir al organizador poner a **"dormir"** a un CPU virtual (método `Duerme(CPU cpu)`) eso significa que cuando le llegue el turno a dicho CPU no se ejecutará ninguna tarea de él sino que se pasará al siguiente CPU. Cuando se le pida al organizador que **"despierte"**(método `Despierta(CPU cpu)`) a un CPU dormido no se reanudará su ejecución hasta que siguiendo el orden a éste le vuelva a tocar su turno de CPU real.

El método `IEnumerable<Tarea> Ejecuta(long ciclos)` devuelve todas las tareas que van terminando, si se ejecutan la cantidad de ciclos de CPU que se indica como parámetros (note que esta secuencia puede ser vacía).

El método `Tarea ProximaTarea()` ejecuta continuamente hasta que finalice alguna tarea, retornando esta. Si no se puede ejecutar ningún ciclo de CPU retorna `null`, esto puede deberse a que todo CPU esté dormido o vacío.

Ejemplo

La Tabla de más abajo nos muestra la existencia de 3 CPUs virtuales, el primero creado con 75 ciclos en cada turno, el segundo con 20 ciclos y el siguiente con 30 ciclos. El primer número de cada tarea indica la cantidad de ciclos ejecutados de esa tarea y el segundo indica la cantidad total de ciclos que necesita dicha tarea. Se indica en amarillo el CPU que está ejecutando.

Si en la situación a continuación se aplicase el método `ProximaTarea` se debe devolver la tarea **C** ya que la tarea M consumiría 20 ciclos y le quedarían 30 para terminar, luego la tarea R consumiría 30 ciclos y le quedarían 10 para terminar, luego se pasaría a la tarea C que consumiría 50 ciclos terminando la tarea C y 25 ciclos ejecutando la tarea A.

Tiempo de cada asignación de CPU	Tarea (tiempo ejecutado/tiempo total)	Tarea (tiempo ejecutado/tiempo total)	Tiempo por consumir
CP1 (75)	C 50/100	A 0/175	
CP2 (20)	M 70/120	B 0/60	20
CP3 (30)	R 40/80	K 0/110	
ProximaTarea devuelve C			
CP1 (75)	A 0/175		25
CP2 (20)	M 90/120	B 0/60	
CP3 (30)	R 70/80	K 0/110	

Si se aplicase ahora el método `Ejecuta(200)`se ejecutan los 25 por consumir en el primer CPU quedando la tarea `A` con 25 de 175 ciclos ejecutados y se pasa al siguiente CPU, la tarea `M` consume los 20 ciclos y quedaría con 110/120, se pasa al siguiente CPU, la tarea `R` consume 10 de los 30 ciclos de dicho CPU y termina, la tarea `K` consume 20 y queda en 20/110. Se han ejecutado ya un total de 75 ciclos.

Se pasa de nuevo al primer CPU, `A` consume los 75 ciclos y queda en 100/175, se han ejecutado ya un total de 150, `M` termina consumiendo 10 y `B` consume los otros 10 quedando en `B` 10/60, se han ejecutado ya 170, se pasa al siguiente CPU y `K` consume los 30 restantes quedando en `K` 50/110. De modo que se devuelve como respuesta un `IEnumerable` con las tareas `R` y `M`.

La tabla muestra la secuencia de distribución, se indica en amarillo la situación de partida y en verde las tareas que van terminando.

Tiempo de cada asignación de CPU	Tarea (tiempo ejecutado/tiempo total)	Tarea (tiempo ejecutado/tiempo total)	Tiempo por consumir
CP1 (75)	A 0/175		25
CP2 (20)	M 90/120	B 0/60	
CP3 (30)	R 70/80	K 0/110	
Tiempo total a ejecutar	200		
CP1 (75)	A 25/175		

CP2 (20)	M 110/120	B 0/60	
CP3 (30)	R 80/80	K 20/110	
Tiempo total a ejecutar	125		
CP1 (75)	A 100/175		
CP2 (20)	M 120/120	B 10/60	
CP3 (30)	K 50/110		
Estado final			
CP1 (75)	A 100/175		75
CP2 (20)	B 10/60		
CP3 (30)	K 50/110		

Implementación

En la biblioteca `Weboo.TercerExamen` se encuentran los siguientes tipos, que utilizará en su implementación.

```
public class Tarea
{
    public Tarea(string nombre, long duracion) { //... }

    public string Nombre { get;}
    public int Duracion { get;}
}
```

Esta clase representa una abstracción del concepto de **tarea**. La propiedad `Duracion` especifica la cantidad de ciclos de CPU necesarios para ejecutar esta tarea. La propiedad `Nombre` simplemente es un nombre para identificar cada tarea.

```
public class CPU
{
    public CPU(long ciclosCPU) { //... }

    public long CiclosCPU { get;}
}
```

Esta clase representa una abstracción del concepto de **CPU virtual**. La propiedad `CiclosCPU` representa la máxima cantidad de ciclos del procesador real que el organizador otorgará de forma continua a este CPU virtual cuando le toque su turno.

```
public interface IOrganizador
{
    /// Devuelva la cantidad de CPU virtuales creados.
    int TotalCPU { get; }

    /// Crea un nuevo CPU virtual.
    CPU CreaCPU(long ciclosCPU);

    /// Adiciona una tarea al CPU virtual especificado.
    void AdicionaTarea(CPU cpu, Tarea tarea);

    /// Devuelve la próxima que termina. Si no hay tareas o todos los CPU están
```

```
        /// interrumpidos, entonces devuelve null.
        Tarea ProximaTarea();

        /// Interrumpe la ejecución del CPU virtual, de forma que
        /// cuando le toque el turno a este CPU el organizador pase
        /// directamente al siguiente.
        void Duerme(CPU cpu);

        /// Reanuda la ejecución del CPU virtual, de forma que
        /// cuando le toque el turno a este CPU funcione de forma habitual.
        void Despierta(CPU cpu);

        /// Devuelve todas las tareas que terminan su ejecución después de esa cantidad
        /// de ciclos de CPU.
        IEnumerable<Tarea> Ejecuta(long ciclos);

        /// Devuelve todas las tareas asignadas a ese CPU (sin quitarlas de la cola
        /// de ejecución).
        IEnumerable<Tarea> TareasEnCPU(CPU cpu);
    }
```

Todos los métodos que reciban algún parámetro con valor `null` deben lanzar una excepción de tipo `ArgumentNullException`. Todos los métodos que reciban un parámetro de tipo `CPU` que no haya sido obtenido con una llamada previa al método `CreaCPU` deben lanzar una excepción de tipo `ArgumentException`. Todos los métodos que reciban un parámetro de tipo `long` con valor menor o igual que cero (<= 0) deben lanzar una excepción de tipo `ArgumentException`.

En la plantilla que usted recibirá se encuentra una clase nombrada `MiOrganizador` que implementa la interfaz `IOrganizador`. Usted debe realizar su solución sobre esta clase. Además, recibirá una aplicación de consola con un código de ejemplo con el que puede probar su solución. Este código no garantiza la correctitud de su respuesta.