Programmazione multithread in C#

Tecnologie e progettazione di sistemi informatici e di telecomunicazioni

Versione n. 19 del 30/12/2023

prof. Roberto FULIGNI

Istituto Tecnico Tecnologico
"Giacomo Fauser"
Novara

Programmazione multithread in C#

1. Calcolo sequenziale e parallelo

Riferimenti: *J. Albahari, Threading in C#* - Part 1: Getting Started - Introduction and Concepts

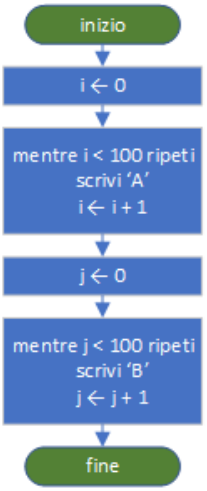
J. Albahari, Threading in C# - Part 1: Getting Started - Thread Pooling (solo “*Entering the Thread Pool via TPL*”)

Esempio n. 1.1

Visualizzare una qualunque sequenza di 200 caratteri composta da cento caratteri ‘A’ e cento ‘B’.

Esempio n. 1.1 – Algoritmo sequenziale

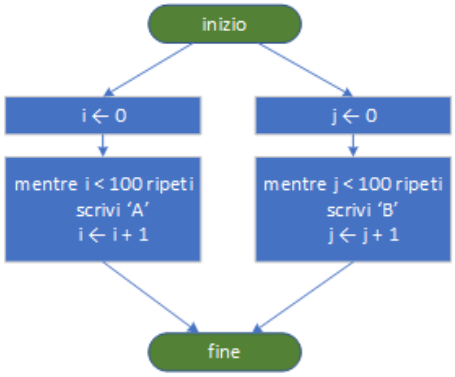
```
inizio
    i ← 0
    mentre i < 100 ripeti
       scrivi ‘A’
       i ← i + 1
    fine mentre
    j ← 0
    mentre j < 100 ripeti
       scrivi ‘B’
       j ← j + 1
    fine mentre
fine
```



Esempio n. 1.1 – Algoritmo sequenziale – Codifica C#

```
static void Main(string[] args)
{
    int i, j;

    Console.WriteLine("Algoritmo sequenziale");

    i = 0;
    while (i < 100)
    {
        Console.Write("A");
        i++;
    }

    j = 0;
    while (j < 100)
    {
        Console.Write("B");
        j++;
    }
}
```

Esempio n. 1.1 – Algoritmo parallelo



Esempio n. 1.1 – Algoritmo parallelo – Codifica in C# (uso della classe Thread)

```
static void Main(string[] args)
{
    Console.WriteLine("Algoritmo parallelo (uso della classe Thread)");

    // Crea un nuovo thread che eseguirà la procedura "StampaA"
    Thread t = new Thread(StampaA);

    t.Start();                          // Avvia il nuovo thread

    // Ora il thread principale stampa il carattere "B"
    int j;
    j = 0;
    while (j < 100)
    {
        Console.Write("B");
        j++;
    }
}

static void StampaA()
{
    int i;
    i = 0;
    while (i < 100)
    {
        Console.Write("A");
        i++;
    }
}
```

Esempio n. 1.1 – Algoritmo parallelo – Codifica in C# (uso della libreria TPL)

```
static void Main(string[] args)
{
    Console.WriteLine("Algoritmo parallelo (uso della libreria TPL)");

    // Seleziona un thread disponibile all'interno del pool di thread e lo avvia.
    // Il thread esegue la procedura "StampaA"

    Task.Factory.StartNew(StampaA);
```

```
        int j;
        j = 0;
        while (j < 100)
        {
            Console.Write("B");
            j++;
        }
    }

    static void StampaA()
    {
        int i;
        i = 0;
        while (i < 100)
        {
            Console.Write("A");
            i++;
        }
    }
```

Esempio n. 1.2

Determinare il tempo necessario per visualizzare la stringa dell’esempio precedente, nell’ipotesi che la stampa di ogni singolo carattere impieghi un tempo di 10 ms. Registrare i tempi ottenuti applicando prima l’algoritmo sequenziale, poi quello parallelo. Determinare infine il valore dello *speedup*.

Esempio n. 1.2– Algoritmo sequenziale – Codifica C#

```
    static void Main(string[] args)
    {
        int i, j;
        Stopwatch watch = new Stopwatch();

        Console.WriteLine("Algoritmo sequenziale");

        watch.Start();      // Avvio del cronometro

        i = 0;
        while (i < 100)
        {
            Console.Write("A");
            Thread.Sleep(10);       // Sospende l'esecuzione del codice per 10 ms
            i++;
        }

        j = 0;
        while (j < 100)
        {
            Console.Write("B");
            Thread.Sleep(10);
            j++;
        }

        // Arresto del cronometro e stampa del tempo impiegato

        watch.Stop();
        Console.WriteLine();
```

```
        Console.WriteLine("Tempo impiegato [ms]: {0}", watch.ElapsedMilliseconds);
    }
```

Il tempo impiegato per visualizzare la sequenza di caratteri con l’algoritmo sequenziale è T(1) = 2177 ms.

Esempio n. 1.2 – Algoritmo parallelo – Codifica in C# (uso della classe Thread)

```
static void Main(string[] args)
{
    Stopwatch watch = new Stopwatch();

    Console.WriteLine("Algoritmo parallelo (uso della classe Thread)");

    watch.Start();

    Thread t = new Thread(StampaA);
    t.Start();

    int j;
    j = 0;
    while (j < 100)
    {
        Console.Write("B");
        Thread.Sleep(10);
        j++;
    }

    t.Join();
    watch.Stop();
    Console.WriteLine();
    Console.WriteLine("Tempo impiegato [ms]: {0}", watch.ElapsedMilliseconds);

}

static void StampaA()
{
    int i;
    i = 0;
    while (i < 100)
    {
        Console.Write("A");
        Thread.Sleep(10);
        i++;
    }
}
```

Il tempo impiegato per visualizzare la sequenza di caratteri con l’algoritmo parallelo e due thread eseguiti da processori distinti è T(2) = 1091 ms.

Dalla formula dello speedup *s*:

$$s = \frac{T(1)}{T(n)}$$ (*n* è il numero totale di processori utilizzati)

Si ottiene *s* = 2177 / 1091 = 1,995.

Esempio n. 1.2 – Algoritmo parallelo – Codifica in C# (uso della libreria TPL)

```
static void Main(string[] args)
```

```
{
    Stopwatch watch = new Stopwatch();

    Console.WriteLine("Algoritmo parallelo (uso della libreria TPL)");

    watch.Start();

    Task t = Task.Factory.StartNew(StampaA);

    int j;
    j = 0;
    while (j < 100)
    {
        Console.Write("B");
        Thread.Sleep(10);
        j++;
    }

    t.Wait();

    watch.Stop();
    Console.WriteLine();
    Console.WriteLine("Tempo impiegato [ms]: {0}", watch.ElapsedMilliseconds);

}

static void StampaA()
{
    int i;
    i = 0;
    while (i < 100)
    {
        Console.Write("A");
        Thread.Sleep(10);
        i++;
    }
}
```

Il tempo impiegato per visualizzare la sequenza di caratteri con l’algoritmo parallelo e due thread e TPL è T(2) = 1096 ms. Non si osservano variazioni significative di tempo (e quindi di speedup) rispetto all’algoritmo basato sulla classe Thread.

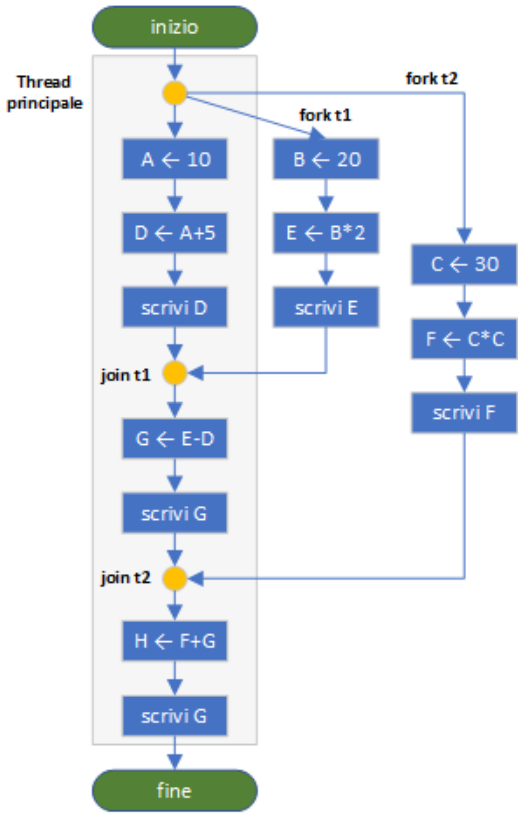
2. Costrutto *fork / join*

Esempio 2.1

Scrivere un programma concorrente che, utilizzando il costrutto *fork/join*, esegua le operazioni indicate di seguito esprimendo il massimo grado di parallelismo.

```
inizio
    A ← 10
    B ← 20
    C ← 30
    D ← A + 5
    scrivi D
    E ← B * 2
    scrivi E
    F ← C * C
    scrivi F
    G ← E – D
    scrivi G
    H ← F + G
    scrivi H
fine
```

Scomposizione dell’algoritmo mediante fork/join



Esempio 2.1 – Programma concorrente (pseudocodifica)

```
/* main */
inizio
    t1 ← fork(proc1)
    t2 ← fork(proc2)
    A ← 10
    D ← A + 5
    scrivi D
    join t1

    G ← E – D
    scrivi G

    join t2


    H ← F + G
    scrivi H
fine
```

```
procedura proc1
inizio
    B ← 20
    E ← B * 2
    scrivi E
fine
```

```
procedura proc2
inizio
    C ← 30
    F ← C * C
    scrivi F
fine
```

Esempio 2.1 – Codifica in C# (uso della classe Thread)

```
static int A, B, C, D, E, F, G, H;
static void Main(string[] args)
{
    Thread t1 = new Thread(Proc1);
    Thread t2 = new Thread(Proc2);
    t1.Start();
    t2.Start();
    A = 10;
    D = A + 5;
    Console.WriteLine("D: {0}", D);

    t1.Join();  // Il main thread attende prima la fine di t1...

    G = E - D;
    Console.WriteLine("G: {0}", G);

    t2.Join();  // ... e poi quella di t2.

    H = F + G;
    Console.WriteLine("H: {0}", H);
}

static void Proc1()
{
    B = 20;
    E = B * 2;
    Console.WriteLine("E: {0}", E);
}
static void Proc2()
{
    C = 30;
    F = C * C;
    Console.WriteLine("F: {0}", F);
}
```

Esempio 2.1 – Codifica in C# (uso della libreria TPL)

```
static int A, B, C, D, E, F, G, H;
static void Main(string[] args)
{
    // Metodo classico di avvio di un nuovo task
    Task tsk1 = Task.Factory.StartNew(Proc1);

    // In alternativa è possibile indicare direttamente le attività svolte
    // dal task utilizzando una "istruzione lambda" (statement lambda): il
    // codice risulta in questo caso più compatto.
    Task tsk2 = Task.Factory.StartNew( () => {
       C = 30;
       F = C * C;
       Console.WriteLine("F: {0}", F);
    });

    A = 10;
    D = A + 5;
    Console.WriteLine("D: {0}", D);
```

```
            tsk1.Wait();  // Attende il completamento del task tsk1

            G = E - D;
            Console.WriteLine("G: {0}", G);

            tsk2.Wait();   // Attende il completamento del task tsk2

            H = F + G;
            Console.WriteLine("H: {0}", H);


        }
        static void Proc1()
        {
            B = 20;
            E = B * 2;
            Console.WriteLine("E: {0}", E);
        }
    }
```

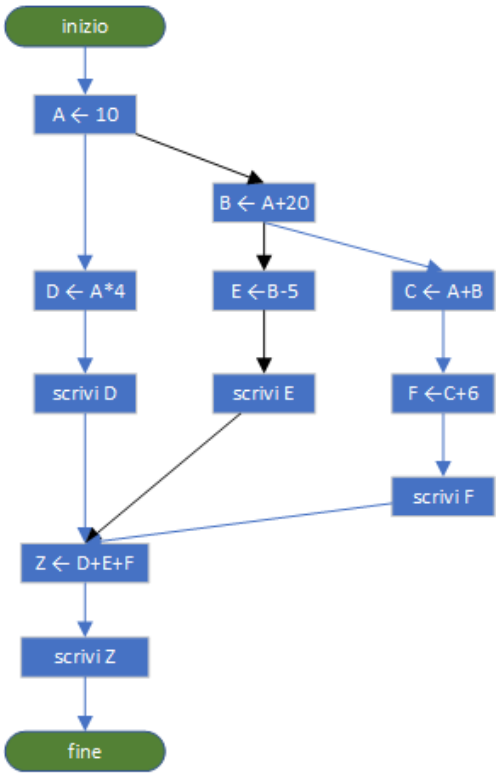
3. Costrutto join(count)

Riferimenti: *J. Albahari, Threading in C#* - Part 2: Basic Synchronization - CountdownEvent

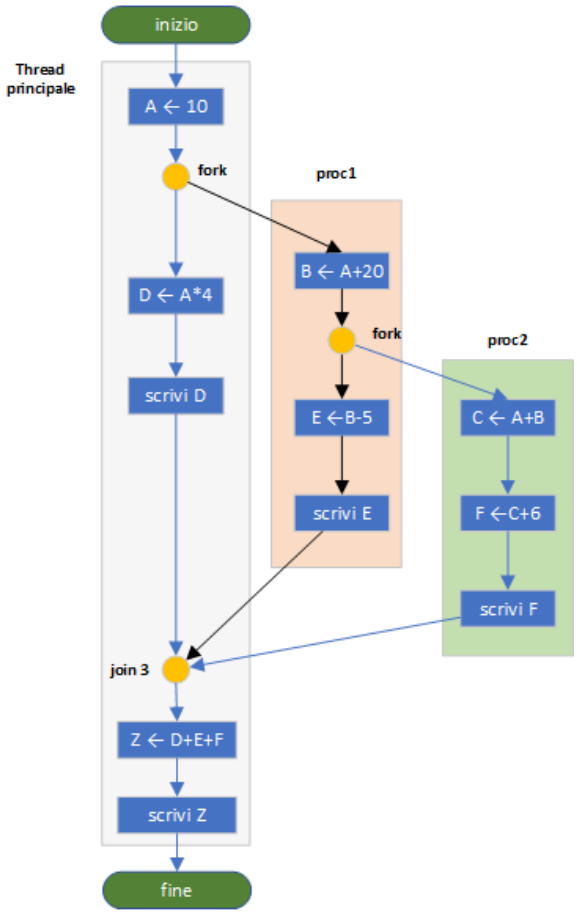
Esempio 3.1

Scrivere, utilizzando il costrutto *join(count)*, un algoritmo concorrente per il seguente diagramma delle precedenze.

Diagramma iniziale



join(count)




```
Conta ←3
inizio  /*main*/
    A ← 10
    fork(proc1)
    D ← A * 4
    scrivi D

L:  join Conta

    Z ← D + E + F
    scrivi Z
fine
```

```
procedura proc1
inizio
    B ← A + 20
    fork(proc2)
    E ← B – 5
    scrivi E

    goto L

fine
```

```
procedura proc2
inizio
    C ← A + B
    F ← C + 6
    scrivi F

    goto L

fine
```

Esempio 3.1 – Codifica in C# (Thread oppure TPL)

In C# il costrutto *join(count)* è realizzabile utilizzando uno speciale contatore costituito da un’istanza della classe *CountdownEvent*: esso è inizialmente impostato a un valore pari al numero di specificato da *count* (nel nostro caso 3) ed è decrementato di un’unità ogni volta che uno dei thread da sincronizzare termina il proprio compito. Per decrementare il contatore si usa il metodo *Signal*.

Il thread principale al termine della propria attività di elaborazione invoca il metodo *Wait* e attende che il valore del contatore si riduca a zero prima di proseguire con le ultime operazioni.

```
        static int A, B, C, D, E, F, Z;
        static CountdownEvent L = new CountdownEvent(3);
        static void Main(string[] args)
        {
            A = 10;
            Thread t = new Thread(Proc1);
            t.Start();

            D = A * 4;
            Console.WriteLine("D: {0}", D);

            L.Signal();     // Decrementa il contatore L

            L.Wait();       // Attende che il contatore L sia uguale a zero

            Z = D + E + F;
            Console.WriteLine("Z: {0}", Z);

        }

        static void Proc1()
        {
            Console.WriteLine("Procedura n. 1");
            B = A + 20;
            Thread t = new Thread(Proc2);
            t.Start();
            E = B - 5;
            Console.WriteLine("E: {0}", E);

            L.Signal();     // Decrementa il contatore L
        }
        static void Proc2()
        {
            Console.WriteLine("Procedura n. 2");
            C = A + B;
            F = C + 6;
            Console.WriteLine("F: {0}", F);

            L.Signal();     // Decrementa il contatore L
        }
    }
```

4. Costrutto *cobegin / coend*

Esempio 4.1

Scrivere un programma concorrente a partire dal seguente diagramma usando il costrutto *cobegin/coend*.

Diagramma iniziale

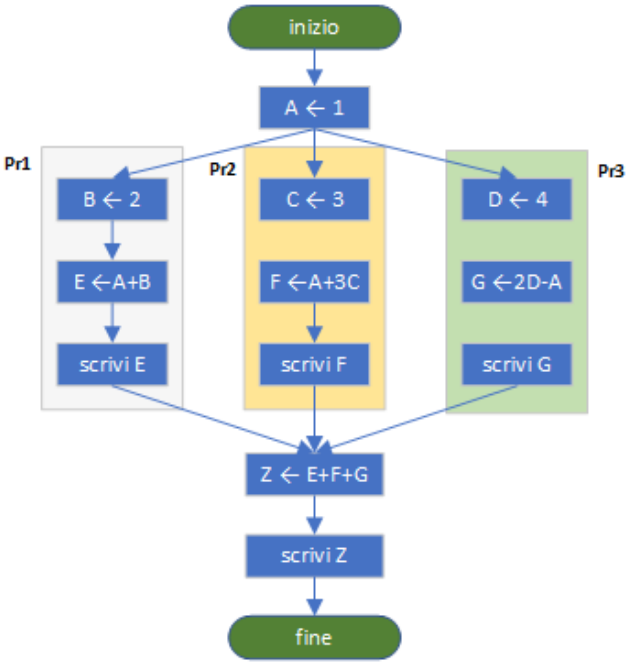
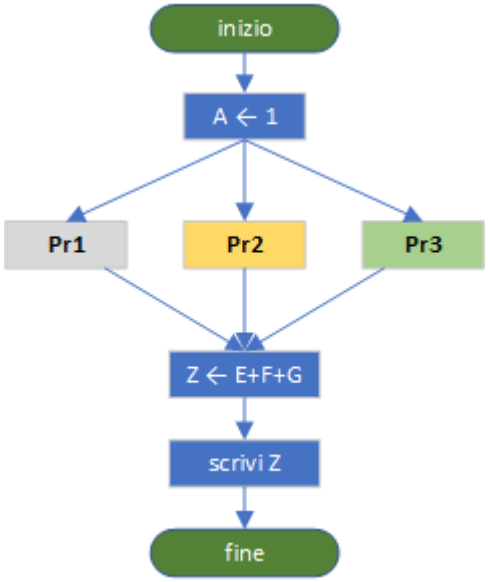


Diagramma trasformato




```
inizio
    A ← 1
    cobegin
       proc1
       proc2
       proc3
    coend
    Z ← E + F + G
    scrivi Z
fine
```

```
procedura proc1
inizio
    B ← 2
    E ← A + B
    scrivi E
fine
```

```
procedura proc2
inizio
    C ← 3
    F ← A + 3*C
    scrivi F
fine
```

```
procedura proc3
inizio
    D ← 4
    G ← 2*D - A
    scrivi G
fine
```

Esempio 4.1 – Codifica in C# (classe Parallel)

```
static int A, B, C, D, E, F, G, Z;
static void Main(string[] args)
{
    A = 1;
    // Esecuzione parallela delle tre procedure, equivalente a:
    // cobegin
    //      Proc1();
    //      Proc2();
    //      Proc3();
    // coend
```

```
        // Il metodo Invoke esegue le tre procedure, normalmente in parallelo.
        // Il main thread attende il completamento di tutte le procedure indicate.
        Parallel.Invoke(Proc1, Proc2, Proc3);

        Z = E + F + G;
        Console.WriteLine("Z: {0}", Z);
    }

    static void Proc1()
    {
        B = 2;
        E = A + B;
        Console.WriteLine("E: {0}", E);
    }
    static void Proc2()
    {
        C = 3;
        F = A + 3*C;
        Console.WriteLine("F: {0}", F);
    }
    static void Proc3()
    {
        D = 4;
        G = 2*D - A;
        Console.WriteLine("G: {0}", G);
    }
```

Esempio 4.1 – Codifica in C# (libreria TPL)

```
    static int A, B, C, D, E, F, G, Z;
    static void Main(string[] args)
    {
        A = 1;
        // Esecuzione parallela delle tre procedure, equivalente a:
        // cobegin
        //      Proc1();
        //      Proc2();
        //      Proc3();
        // coend

        Task tsk1 = Task.Factory.StartNew(Proc1);
        Task tsk2 = Task.Factory.StartNew(Proc2);
        Task tsk3 = Task.Factory.StartNew(Proc3);

        // Il main thread attende il completamento di tutti i task
        Task.WaitAll(tsk1, tsk2, tsk3);

        Z = E + F + G;
        Console.WriteLine("Z: {0}", Z);
    }

    static void Proc1()
    {
        B = 2;
        E = A + B;
        Console.WriteLine("E: {0}", E);
    }
    static void Proc2()
```

```
        {
            C = 3;
            F = A + 3 * C;
            Console.WriteLine("F: {0}", F);
        }
        static void Proc3()
        {
            D = 4;
            G = 2 * D - A;
            Console.WriteLine("G: {0}", G);
        }
    }
```

5. Programmazione multithread in WPF

Riferimenti: *Gul Md Ershad* - Update UI With WPF Dispatcher And TPL

Ammar Shaukat - Using XAML Progress Bar In WPF

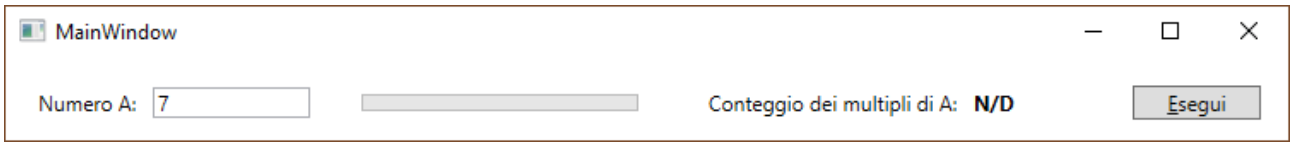
Microsoft Docs - Threading Model

Esempio 5.1

Scrivere un’applicazione WPF multithread che, richiesto in input un numero A intero e positivo, visualizzi il conteggio dei multipli di A minori o uguali a 200 000 000.

Esempio 5.1 – Interfaccia grafica XAML

La finestra principale contiene un controllo *textbox* (nome **tbA**) per l’inserimento del numero A, una *label* (**lbConteggio**) per il conteggio finale, una *progress bar* per indicare lo stato di avanzamento dell’elaborazione (**pbElab**) e un bottone (**bntEsegui**).




```
<Grid>
        <Label Content="Numero A:" HorizontalAlignment="Left" Margin="14,13,0,0"
               VerticalAlignment="Top"/>

        <TextBox x:Name="tbA" HorizontalAlignment="Left" Height="18" Margin="86,17,0,0"
               TextWrapping="Wrap" Text="7" VerticalAlignment="Top" Width="92"/>

        <Label Content="Conteggio dei multipli di A:" HorizontalAlignment="Left"
               Margin="406,13,0,0" VerticalAlignment="Top"/>

        <Label x:Name="lbConteggio" Content="N/D" HorizontalAlignment="Left"
               Margin="561,13,0,0" VerticalAlignment="Top" Width="93" FontWeight="Bold"/>

        <ProgressBar x:Name="pbElab" HorizontalAlignment="Left" Height="10"
               Margin="208,21,0,0" VerticalAlignment="Top" Width="162" Maximum="200000000"/>

        <Button x:Name="btnEsegui" Content="_Esegui" HorizontalAlignment="Left"
               Margin="659,16,0,0" VerticalAlignment="Top" Width="75" Click="btnEsegui_Click"/>
</Grid>
```

Esempio 5.1 – Codifica in C#

Poiché il calcolo è computazionalmente oneroso, decidiamo di contare i multipli attraverso un thread dedicato. Scriviamo innanzitutto la procedura da eseguire:

```
private void EseguiElaborazione()
{
    int A = 0;          // Conterrà il dato di input convertito in numero
    int conteggio = 0;

    Dispatcher.Invoke(() =>
   {
       // Converte la stringa contenuta nella textbox in un numero intero.
       // Si suppone che la conversione abbia sempre successo.
       A = Convert.ToInt32(tbA.Text);

       // Inizializzazione dei controlli grafici
       btnEsegui.IsEnabled = false;
       lbConteggio.Content = "";
       pbElab.Value = 0;
   });

    for(int m = 1; m <= N; m++)
    {
        if (m % A == 0)
            conteggio++;

        if (m % 1000000 == 0)
        {
            // Aggiornamento della progress bar ogni milione di numeri
            Dispatcher.Invoke(() =>
            {
                pbElab.Value = m;
            });
        }
    }

    // Visualizzazione del conteggio finale e ripristino del bottone
    Dispatcher.Invoke(() =>
    {
        lbConteggio.Content = conteggio.ToString();
        btnEsegui.IsEnabled = true;
    });
}
```

Nella funzione associata all’evento *Click* del bottone avviamo un nuovo task per eseguire la procedura:

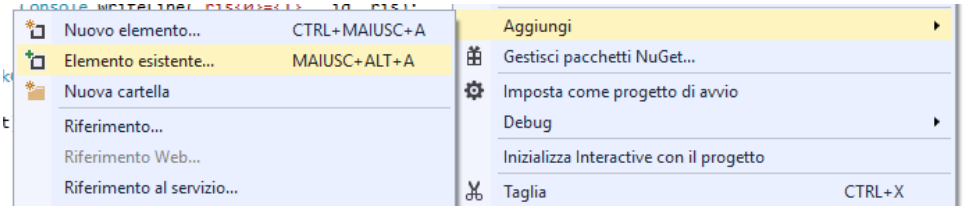
```
private void btnEsegui_Click(object sender, RoutedEventArgs e)
{
    Task.Factory.StartNew(EseguiElaborazione);
}
```

Esempio 5.2

Realizzare un’applicazione WPF che mostri, mediante una semplice animazione, la fase di decollo di un aeroplano.

Esempio 5.2 – Preparazione dell’interfaccia grafica

Dopo aver memorizzato l’immagine di un aeroplano nel file **aereo.png**, aggiungere un elemento esistente al progetto Visual Studio:



Selezionare il file **aereo.png**. Verificare la presenza, nell’elenco dei file di progetto, dell’immagine selezionata.

Aggiungere alla finestra WPF un nuovo controllo di tipo *Image* (nome **imgAereo**) e collocarla in basso a sinistra (regolare i margini: left = 10; top = 320). Impostare la proprietà *Source* = “*aereo.png*”. Completare l’interfaccia con un bottone (**btnDecolla**).


```
<Grid>
        <Image x:Name="imgAereo" HorizontalAlignment="Left" Margin="10,320,0,0"
               VerticalAlignment="Top" Stretch="UniformToFill" Height="83" Width="219"
               Source="aereo.png"/>

        <Button x:Name="btnDecolla" Content="_Decolla" HorizontalAlignment="Left"
               Margin="671,384,0,0" VerticalAlignment="Top" Width="75"
               Click="btnDecolla_Click"/>

</Grid>
```

Esempio 5.2 – Codifica in C#

```
        private void EseguiDecollo()
        {
            const double DX = 10;
            const double DY = -5;

            // Fase 1 - Si disabilita il bottone "Decolla"
            //          (non si deve iniziare un nuovo decollo
            //           prima di aver completato quello corrente)

            Dispatcher.Invoke(() =>
            {
                btnDecolla.IsEnabled = false;
            });

            // Fase 2 - Simulazione della fase di decollo
            //          Si sposta ripetutamente l'immagine modificandone i margini.
```

```
            //          La velocità dell'animazione è regolata introducendo
            //          opportuni ritardi (sleep).

            for(int i = 0; i < 76; i++)
            {
                Thread.Sleep(100);
                // Aggiornamento dell'interfaccia grafica
                Dispatcher.Invoke(() =>
                {
                    Thickness margini = imgAereo.Margin;
                    margini.Left += DX;
                    margini.Top += DY;
                    imgAereo.Margin = margini;
                });

            }

            // Fase 3 - Conclusa l'animazione, si riporta l'areo nella
            //          posizione di partenza e si riabilita il bottone.

            Dispatcher.Invoke(() =>
            {
                imgAereo.Margin = new Thickness(10, 320, 0, 0);     // Left: 10; Top: 320;
                                                                    // Bottom e Right: 0
                btnDecolla.IsEnabled = true;
            });
        }

        private void btnDecolla_Click(object sender, RoutedEventArgs e)
        {
            // Cliccando sul bottone, si esegue l'animazione con un nuovo task
            Task.Factory.StartNew(EseguiDecollo);
        }
    }
```

6. Sincronizzazione tra processi: lock e semafori

Riferimenti: *J. Albahari, threading in c#* - Part 2: Basic Synchronization – Locking (solo costrutto *lock* e sezioni *When to Lock*, *Locking and Atomicity*, *Nested Locking*, *Deadlocks*, *Semaphore*)

Esempio 6.1 – Thread non sincronizzati

Si consideri il seguente programma concorrente:

```
/* variabili globali */
var N

inizio
    N ← 0
    cobegin
       Incrementa
       Decrementa
    coend
    scrivi N
fine
```

```
procedura Incrementa
/* variabili locali */
var i, temp

inizio
    i ← 0
    mentre i<100000 ripeti
       temp ← N
       temp ← temp + 1
       N ← temp
       i ← i + 1
    fine mentre
fine
```

```
procedura Decrementa
/* variabili locali */
var i, temp

inizio
    i ← 0
    mentre i<100000 ripeti
       temp ← N
       temp ← temp - 1
       N ← temp
       i ← i + 1
    fine mentre
fine
```

Il main thread, dopo aver inizializzato a 0 la variabile condivisa N, esegue concorrentemente le procedure I*ncrementa* (aumenta il valore di N di un’unita alla volta per 100 000 volte) e D*ecrementa* (diminuisce N in un’unità per 100 000 volte). Prima di terminare, il thread stampa il contenuto finale di N.

Dato che la variabile N è incrementata e decrementata per lo stesso numero di volte, ci aspettiamo che alla fine il suo valore sia ancora 0: in realtà, l’output prodotto contiene, in generale, un risultato diverso.

Per verificare le condizioni di Bernstein, determiniamo innanzitutto il dominio e il rango delle due procedure (si noti che non sono considerate le variabili locali):

- *domain(Incrementa)*: {N} r*ange(Incrementa)*: {N}
- *domain(Decrementa)*: {N} r*ange(Decrementa)*: {N}

Poiché r*ange(incrementa)* $\cap$ *range(decrementa)* $\neq \emptyset$, le condizioni non sono soddisfatte e il risultato può essere errato a causa di sequenze di *interleaving* che generano interferenza.

Esempio 6.1 – Thread non sincronizzati (codifica in C#)

```
const int MAX_ITERAZIONI = 100000;
static int N;
static void Main(string[] args)
{
    Console.WriteLine("ACCESSO LIBERO");
    N = 0;
    Parallel.Invoke(Incrementa, Decrementa);
    Console.WriteLine("N = {0}", N);
}

static void Incrementa()
{
    int i, temp;
```

```
        i = 0;
        while (i < MAX_ITERAZIONI)
        {
            // Inizio della sezione critica (accesso libero alla variabile N)
            temp = N;
            temp = temp + 1;
            N = temp;
            // Fine della sezione critica

            i++;
        }
    }
    static void Decrementa()
    {
        int i, temp;

        i = 0;
        while (i < MAX_ITERAZIONI)
        {
            // Inizio della sezione critica (accesso libero alla variabile N)
            temp = N;
            temp = temp - 1;
            N = temp;
            // Fine della sezione critica

            i++;
        }
    }
```

Esempio 6.2 – Mutua esclusione (semaforo binario e primitive lock/unlock)

Il programma precedente può essere corretto consentendo ai due thread concorrenti di accedere alla variabile N solo in mutua esclusione. Per fare questo, si introduce un semaforo binario *S*:

```
/* variabili globali */
var N
semaforo S = 1

inizio
    N ← 0
    cobegin
       Incrementa
       Decrementa
    coend
    scrivi N
fine
```

```
procedura Incrementa
/* variabili locali */
var i, temp

inizio
    i ← 0
    mentre i<100000 ripeti
       lock(S)
       temp ← N
       temp ← temp + 1
       N ← temp
       unlock(S)
       i ← i + 1
    fine mentre
fine
```

```
procedura Decrementa
/* variabili locali */
var i, temp

inizio
    i ← 0
    mentre i<100000 ripeti
       lock(S)
       temp ← N
       temp ← temp - 1
       N ← temp
       unlock(S)
       i ← i + 1
    fine mentre
fine
```

Esempio 6.2 – Mutua esclusione – Codifica in C# (costrutto lock)

```
    const int MAX_ITERAZIONI = 100000;
    static int N;
    // La variabile locker sarà utilizzata dal programma per controllare
    // l'accesso a una sezione critica in mutua esclusione
    static readonly object locker = new object();
```

```
static void Main(string[] args)
{
    Console.WriteLine("ACCESSO IN MUTUA ESCLUSIONE (LOCK)");
    N = 0;
    Parallel.Invoke(Incrementa, Decrementa);
    Console.WriteLine("N = {0}", N);
}

static void Incrementa()
{
    int i, temp;

    i = 0;
    while (i < MAX_ITERAZIONI)
    {
        // Inizio della sezione critica (accesso alla variabile N regolato da lock)
        lock(locker)
        {
            temp = N;
            temp = temp + 1;
            N = temp;
        }
        // Fine della sezione critica

        i++;
    }
}

static void Decrementa()
{
    int i, temp;

    i = 0;
    while (i < MAX_ITERAZIONI)
    {
        // Inizio della sezione critica (accesso alla variabile N regolato da lock)
        lock (locker)
        {
            temp = N;
            temp = temp - 1;
            N = temp;
        }
        // Fine della sezione critica

        i++;
    }
}
```

Esempio 6.2 – Mutua esclusione – Codifica in C# (semaforo binario)

```
const int MAX_ITERAZIONI = 100000;

static int N;
// Il semaforo binario (impostato inizialmente a 1) sarà utilizzato dal programma
// per controllare l'accesso a una sezione critica in mutua esclusione
static SemaphoreSlim S = new SemaphoreSlim(1);

static void Main(string[] args)
{
```

```
        Console.WriteLine("ACCESSO IN MUTUA ESCLUSIONE (SEMAFORO BINARIO)");
        N = 0;
        Parallel.Invoke(Incrementa, Decrementa);
        Console.WriteLine("N = {0}", N);
    }

    static void Incrementa()
    {
        int i, temp;

        i = 0;
        while (i < MAX_ITERAZIONI)
        {
            // Attende che il semaforo sia pari a 1 (verde).
            // Quando il semaforo è verde, lo imposta a 0 (rosso) e prosegue
            S.Wait();

            // Inizio della sezione critica (accesso alla var. N regolato da semaforo)
            temp = N;
            temp = temp + 1;
            N = temp;
            // Fine della sezione critica

            // Ripristina il semaforo a 1 (verde)
            S.Release();

            i++;
        }
    }

    static void Decrementa()
    {
        int i, temp;

        i = 0;
        while (i < MAX_ITERAZIONI)
        {
            // Attende che il semaforo sia pari a 1 (verde).
            // Quando il semaforo è verde, lo imposta a 0 (rosso) e prosegue
            S.Wait();

            // Inizio della sezione critica (accesso alla var. N regolato da semaforo)
            temp = N;
            temp = temp - 1;
            N = temp;
            // Fine della sezione critica

            // Ripristina il semaforo a 1 (verde)
            S.Release();

            i++;
        }
    }
```

7. Applicazione dei semafori

Esempio 7.1 – Vincolo di precedenza

Una variabile intera globale S è condivisa da due procedure eseguite in modo concorrente:

- Procedura 1: legge da tastiera due numeri interi e li memorizza nelle variabili locali *a* e *b*; assegna alla variabile S il risultato di a + b; stampa su schermo i valori delle due variabili locali;
- Procedura 2: genera un numero intero casuale di due cifre e lo memorizza nella variabile locale *c*; stampa su schermo la variabile *c* e la somma elaborata dalla prima procedura e contenuta in S.

Diagramma delle precedenze

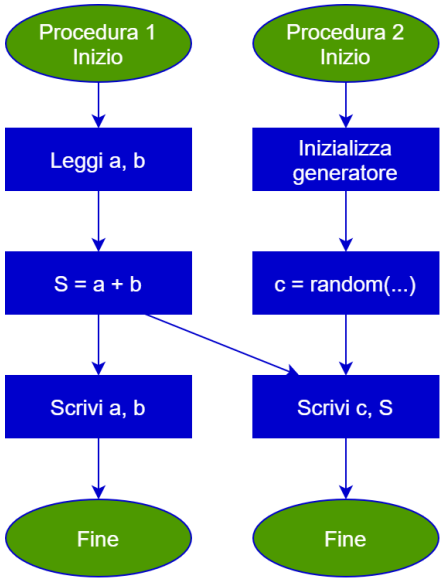
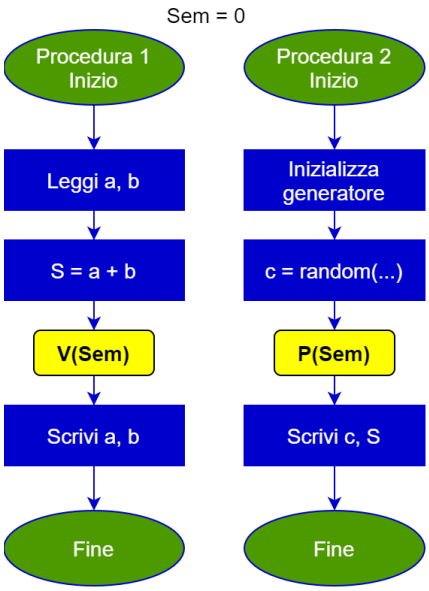


Diagramma con semaforo



Esempio 7.1 – Vincolo di precedenza (codifica in C#)

```
static int S;
static SemaphoreSlim Sem = new SemaphoreSlim(0);

static void Main(string[] args)
{
    Parallel.Invoke(Procedura1, Procedura2);
}

static void Procedura1()
{
    int a, b;
    Console.Write("Inserire il numero a: ");
    a = Convert.ToInt32(Console.ReadLine());
    Console.Write("Inserire il numero b: ");
    b = Convert.ToInt32(Console.ReadLine());
    S = a + b;

    // Pone il semaforo a 1, la seconda procedura può elaborare S
    Sem.Release();
```

```
        Console.WriteLine("a = {0}    b = {1}", a, b);
    }
    static void Procedura2()
    {
        Random r = new Random(2021);
        int c = r.Next(10, 100);

        // In attesa del calcolo della somma S da parte della prima procedura
        Sem.Wait();

        Console.WriteLine("c = {0}     S = {1}", c, S);
    }
```

Esempio 7.2 – Problema del *rendez-vous*

Dati due vettori di interi V e W, si considerino le seguenti attività da eseguire concorrentemente:

- Attività 1: memorizzazione di valori casuali nel vettore V (seed 2018) e determinazione del minimo valore contenuto nella concatenazione di V e W.
- Attività 2: memorizzazione di valori casuali nel vettore W (seed 2019) e determinazione della media aritmetica dei valori contenuti nella concatenazione di V e W.

Esempio 7.2 – Problema del *rendez-vous* (codifica in C#)

```
const int Lunghezza = 100;
static int[] V = new int[Lunghezza];
static int[] W = new int[Lunghezza];
static SemaphoreSlim mutex1 = new SemaphoreSlim(0);
static SemaphoreSlim mutex2 = new SemaphoreSlim(0);
static void Main(string[] args)
{
    Parallel.Invoke(Attivita1, Attivita2);
}
static void Attivita1()
{
    Random r = new Random(2018);
    int minimo;
    // Prima parte: inserimento di valori casuali in V
    for (int i = 0; i < Lunghezza; i++)
        V[i] = r.Next(1000);            // Numeri casuali minori di 1000

    mutex1.Release();       // Segnala la fine della prima parte di "Attività 1"
    mutex2.Wait();          // In attesa del completamento della prima parte di  "Attività 2"

    // Seconda parte: determinazione del valore di minimo
    minimo = Math.Min(V.Min(), W.Min());
    Console.WriteLine("Valore minimo contenuto in V,W: {0}", minimo);
}
static void Attivita2()
{
    Random r = new Random(2019);
    int somma;
    double media;
    // Prima parte: inserimento di valori casuali in V
    for (int i = 0; i < Lunghezza; i++)
```

```
            W[i] = r.Next(1000);

        mutex2.Release();       // Segnala la fine della prima parte di "Attività 2"
        mutex1.Wait();          // In attesa del completamento della prima parte di  "Attività 1"

        // Seconda parte: determinazione della media aritmetica
        somma = V.Sum() + W.Sum();
        media = somma / (double)(V.Length + W.Length);
        Console.WriteLine("Media degli elementi contenuti in V,W: {0:N2}", media);
    }
```

Esempio 7.3 – Problema del rendez-vous prolungato

Due thread si scambiano alcuni dati utilizzando un buffer condiviso. Il primo thread mette nel buffer un dato (costituito da un numero pari) alla volta, ma per poter scrivere il secondo dato deve attendere che l’altro thread prelevi il primo, e così via.

Esempio 7.3 – Problema del rendez-vous prolungato (codifica in C#)

```
const int NumValori = 10;
static int buffer;
// Il buffer è inizialmente vuoto
static SemaphoreSlim bufferVuoto = new SemaphoreSlim(1);
static SemaphoreSlim bufferPieno = new SemaphoreSlim(0);

static void Main(string[] args)
{
    Parallel.Invoke(Metti, Togli);
}

static void Metti()
{
    Random r = new Random(2018);
    for(int i = 0; i < NumValori; i++)
    {
        int valorePari;
        do
        {
            valorePari = r.Next(100);
        }
        while (valorePari % 2 != 0);    // Ripeti se il numero non è pari

        bufferVuoto.Wait();     // Attende che il buffer si svuoti
        Thread.Sleep(100);
        buffer = valorePari;
        Console.WriteLine("La procedura Metti ha inserito nel buffer il numero {0}",
            buffer);
        bufferPieno.Release();  // Segnala che il buffer è di nuovo pieno
    }


}
static void Togli()
{
    for (int i = 0; i < NumValori; i++)
    {
        bufferPieno.Wait();     // Attende che il buffer si riempia
```

```
            Thread.Sleep(50);
            Console.WriteLine("La procedura Togli ha estratto dal buffer il numero {0}",
                buffer);
            bufferVuoto.Release();  // Segnala che ora il buffer è vuoto
        }

    }
```

Esempio 7.4 – Mutua esclusione tra gruppi di processi

Un negozio di abbigliamento vende un solo articolo “alla moda” in quantità illimitata. Lo stilista che lavora per il negozio entra periodicamente nel locale, sostituisce l’articolo presente con uno nuovo in un tempo TS ed esce. Un cliente entra nel negozio e compra l’articolo disponibile in quel momento in un tempo TC. L’accesso al negozio può avvenire solamente nei seguenti casi:

- Lo stilista può entrare solo se non ci sono clienti, altrimenti aspetta che il negozio sia vuoto.
- Un cliente può entrare se il negozio è vuoto oppure se ci sono altri clienti. Se nel locale è presente lo stilista, il cliente attende che si completi la sostituzione dell’articolo prima di accedere al locale.

I clienti e lo stilista accedono concorrentemente e sono simulati da thread distinti.

Esempio 7.4 – Mutua esclusione tra gruppi di processi (codifica in C#)

```
const int NumClienti = 20;
const int RitardoCliente = 1300;                // ms
const int RitardoStilista = 2000;               // ms
const int DurataAcquistoCliente = 1200;         // ms
const int DurataLavoroStilista = 2000;          // ms
static readonly string[] articoli = new string[]
{
    "Stivali rossi",
    "Giacca blu",
    "Camicia bianca",
    "Cintura verde",
    "Guanti neri",
    "Cappotto grigio"
};
static SemaphoreSlim mutex = new SemaphoreSlim(1);
static int moda = 0;
static SemaphoreSlim mutexGruppo = new SemaphoreSlim(1);
static int processiGruppoDentro = 0;
static void Main(string[] args)
{
    List<Task> lista = new List<Task>();

    lista.Add(Task.Factory.StartNew(() =>
    {
        ElaboraStilista();
    }, TaskCreationOptions.LongRunning));

    for (int i = 0; i < NumClienti; i++)
    {
        int idCliente = i;
        Thread.Sleep(RitardoCliente);
        lista.Add(Task.Factory.StartNew(() =>
```

```
            {
                ElaboraCliente(idCliente);
            }, TaskCreationOptions.LongRunning));
        }

        Task.WaitAll(lista.ToArray());
    }

    static void ElaboraCliente(int id)
    {
        Console.WriteLine("Il cliente n. {0} è giunto al negozio.", id);

        mutexGruppo.Wait();
        processiGruppoDentro++;
        if (processiGruppoDentro == 1)
            mutex.Wait();
        Console.WriteLine("Il cliente n. {0} è entrato. Nel negozio ci sono ora {1} clienti.",
          id, processiGruppoDentro);
        mutexGruppo.Release();

        Thread.Sleep(DurataAcquistoCliente);
        Console.WriteLine("Il cliente n. {0} ha acquistato: {1}", id, articoli[moda]);

        mutexGruppo.Wait();
        processiGruppoDentro--;
        if (processiGruppoDentro == 0)
            mutex.Release();
        Console.WriteLine("Il cliente n. {0} è uscito. Nel negozio ci sono ora {1} clienti.",
            id, processiGruppoDentro);
        mutexGruppo.Release();
    }

    static void ElaboraStilista()
    {
        const int NumVolteStilista = 5;
        for (int i = 0; i < NumVolteStilista; i++)
        {
            Thread.Sleep(RitardoStilista);
            Console.WriteLine("Lo stilista desidera creare un nuovo articolo.", i);

            mutex.Wait();

            Console.WriteLine("Lo stilista si è messo al lavoro.", i);
            Thread.Sleep(DurataLavoroStilista);
            moda++;
            if (moda == articoli.Length)
                moda = 0;
            Console.WriteLine("***** LO STILISTA HA CREATO UN NUOVO ARTICOLO: {0}",
                articoli[moda]);

            mutex.Release();
        }

    }
```

Esempio 7.5 – Mutua esclusione tra gruppi di processi (senza starvation)

Se i thread clienti entrano nel negozio molto frequentemente, il thread stilista potrebbe non riuscire a realizzare nuovi capi di abbigliamento, rimanendo fuori ad aspettare per un tempo indefinito (starvation).

Risolvere il problema precedente in modo da prevenire l’attesa indefinita del thread stilista.

Esempio 7.5 – Mutua esclusione tra gruppi di processi (senza starvation, codifica in C#)

```
const int LimiteClientiConsecutivi = 5;   // Numero massimo di clienti che possono
                                          // accedere consecutivamente
                                          // Quando si raggiunge questo limite, il
                                          // gruppo si prepara a rilasciare
                                          // la risorsa "negozio"
const int NumClienti = 50;
const int RitardoCliente = 500;                 // ms
const int RitardoStilista = 2000;               // ms
const int DurataAcquistoCliente = 1200;         // ms
const int DurataLavoroStilista = 2000;          // ms
static readonly string[] articoli = new string[]
{
    "Stivali rossi",
    "Giacca blu",
    "Camicia bianca",
    "Cintura verde",
    "Guanti neri",
    "Cappotto grigio"
};
static int moda = 0;
// Semaforo di tipo FIFO per una gestione equa dei thread in attesa
static Semaphore mutexNegozio = new Semaphore(1, 1);
static SemaphoreSlim mutexGruppo = new SemaphoreSlim(1);
static int processiGruppoDentro = 0;

// Semaforo a conteggio per la gestione del numero massimo di clienti consecutivi
static SemaphoreSlim limiteGruppo = new SemaphoreSlim(LimiteClientiConsecutivi,
                                                      LimiteClientiConsecutivi);
static void Main(string[] args)
{
    List<Task> lista = new List<Task>();

    lista.Add(Task.Factory.StartNew(() =>
    {
        ElaboraStilista();
    }, TaskCreationOptions.LongRunning));

    for (int i = 0; i < NumClienti; i++)
    {
        int idCliente = i;
        Thread.Sleep(RitardoCliente);
        lista.Add(Task.Factory.StartNew(() =>
        {
            ElaboraCliente(idCliente);
        }, TaskCreationOptions.LongRunning));
    }
    Task.WaitAll(lista.ToArray());
}

static void ElaboraCliente(int id)
{
    Console.WriteLine("Il cliente n. {0} è giunto al negozio.", id);

    limiteGruppo.Wait();    // Se questo semaforo è pari a 0, è stato raggiunto il
```

```
                                // limite massimo di clienti consecutivi

        mutexGruppo.Wait();
        processiGruppoDentro++;
        if (processiGruppoDentro == 1)
            mutexNegozio.WaitOne();
        Console.WriteLine("Il cliente n. {0} è entrato. Nel negozio ci sono ora {1} clienti.",
                          id, processiGruppoDentro);
        mutexGruppo.Release();

        Thread.Sleep(DurataAcquistoCliente);
        Console.WriteLine("Il cliente n. {0} ha acquistato: {1}", id, articoli[moda]);

        mutexGruppo.Wait();
        processiGruppoDentro--;
       Console.WriteLine("Il cliente n. {0} è uscito. Nel negozio ci sono ora {1} clienti.", id,
                         processiGruppoDentro);
        if (processiGruppoDentro == 0)
        {
            mutexNegozio.Release();
            // Si rilascia il semaforo per un numero di volte sufficiente a consentire
            // l'accesso a un nuovo gruppo di clienti
            // Il nuovo gruppo entra eventualmente in competizione con lo stilista per
            // accedere al negozio.
            // Qualunque sia il valore attuale, il semaforo è impostato a un valore pari
            // a LimiteClientiConsecutivi
            limiteGruppo.Release(LimiteClientiConsecutivi - limiteGruppo.CurrentCount);
        }
        mutexGruppo.Release();
    }

    static void ElaboraStilista()
    {
        const int NumVolteStilista = 5;
        for (int i = 0; i < NumVolteStilista; i++)
        {
            Thread.Sleep(RitardoStilista);
            Console.WriteLine();
            Console.WriteLine("****** Lo stilista desidera creare un nuovo articolo.",
                              i);
            Console.WriteLine();

            mutexNegozio.WaitOne();

            Console.WriteLine();
            Console.WriteLine("****** Lo stilista si è messo al lavoro.", i);
            Console.WriteLine();
            Thread.Sleep(DurataLavoroStilista);
            moda++;
            if (moda == articoli.Length)
                moda = 0;
            Console.WriteLine();
            Console.WriteLine("****** LO STILISTA HA CREATO UN NUOVO ARTICOLO: {0}",
                              articoli[moda]);
            Console.WriteLine();
            mutexNegozio.Release();
        }
    }
```

8. Problema del produttore/consumatore

Esempio 8.1 – Un produttore, un consumatore, buffer singolo (codifica in C#)

```
const int NumDati = 10;
const int TempoInserimentoBuffer = 200;    // [ms]
const int TempoEstrazioneBuffer = 500;     // [ms]
static SemaphoreSlim vuoto = new SemaphoreSlim(1);
static SemaphoreSlim pieno = new SemaphoreSlim(0);
static int buffer;
static void Produci(int inizio, int fine)
{
    Stopwatch swProd = new Stopwatch();
    swProd.Start();
    for (int i = inizio; i <= fine; i++)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        int quad = i * i;
        vuoto.Wait();
        Thread.Sleep(TempoInserimentoBuffer);
        buffer = quad;
        pieno.Release();
        Console.WriteLine("Il thread produttore n. {0} ha inserito il numero {1}",
                          threadId, quad);
    }
    swProd.Stop();
    Console.WriteLine("Tempo impiegato dal produttore: {0} ms",
                          swProd.ElapsedMilliseconds);
}
static void Consuma(int quanti)
{
    Stopwatch swCons = new Stopwatch();
    swCons.Start();
    for (int i = 1; i <= quanti; i++)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        int quad;
        pieno.Wait();
        Thread.Sleep(TempoEstrazioneBuffer);
        quad = buffer;
        vuoto.Release();
        Console.WriteLine("Il thread consumatore n. {0} ha estratto il numero {1}",
                          threadId, quad);
    }
    swCons.Stop();
    Console.WriteLine("Tempo impiegato dal consumatore: {0} ms",
                          swCons.ElapsedMilliseconds);
}
static void Main(string[] args)
{
    Console.WriteLine("Un produttore, un consumatore, buffer singolo");
    Console.WriteLine();
    Stopwatch sw = new Stopwatch();
    sw.Start();
    Parallel.Invoke(
        () => Produci(1, NumDati),
        () => Consuma(NumDati)
```

```
        );
        sw.Stop();
        Console.WriteLine("Elaborazione terminata. Tempo totale trascorso: {0} ms",
                          sw.ElapsedMilliseconds);
    }
```

Esempio 8.2 – Un produttore, un consumatore, buffer circolare (codifica in C#)

```
    const int NumDati = 10;
    const int DimBuffer = 2;
    const int TempoInserimentoBuffer = 200;    // [ms]
    const int TempoEstrazioneBuffer = 500;     // [ms]
    static SemaphoreSlim vuoto = new SemaphoreSlim(DimBuffer);
    static SemaphoreSlim pieno = new SemaphoreSlim(0);
    static int[] buffer = new int[DimBuffer];
    static int metti = 0;
    static int togli = 0;
    static void Produci(int inizio, int fine)
    {
        Stopwatch swProd = new Stopwatch();
        swProd.Start();
        for (int i = inizio; i <= fine; i++)
        {
            int threadId = Thread.CurrentThread.ManagedThreadId;
            int quad = i * i;
            vuoto.Wait();
            Thread.Sleep(TempoInserimentoBuffer);
            buffer[metti] = quad;
            metti = (metti + 1) % DimBuffer;
            pieno.Release();
            Console.WriteLine("Il thread produttore n. {0} ha inserito il numero {1}",
                              threadId, quad);
        }
        swProd.Stop();
        Console.WriteLine("Tempo impiegato dal produttore: {0} ms",
                              swProd.ElapsedMilliseconds);
    }
    static void Consuma(int quanti)
    {
        Stopwatch swCons = new Stopwatch();
        swCons.Start();
        for (int i = 1; i <= quanti; i++)
        {
            int threadId = Thread.CurrentThread.ManagedThreadId;
            int quad;
            pieno.Wait();
            Thread.Sleep(TempoEstrazioneBuffer);
            quad = buffer[togli];
            togli = (togli + 1) % DimBuffer;
            vuoto.Release();
            Console.WriteLine("Il thread consumatore n. {0} ha estratto il numero {1}",
                              threadId, quad);
        }
        swCons.Stop();
        Console.WriteLine("Tempo impiegato dal consumatore: {0} ms",
                              swCons.ElapsedMilliseconds);
    }
    static void Main(string[] args)
```

```
{
    Console.WriteLine("Un produttore, un consumatore, buffer circolare a {0} celle",
                      DimBuffer);
    Console.WriteLine();
    Stopwatch sw = new Stopwatch();
    sw.Start();
    Parallel.Invoke(
        () => Produci(1, NumDati),
        () => Consuma(NumDati)
    );
    sw.Stop();
    Console.WriteLine("Elaborazione terminata. Tempo trascorso: {0} ms",
                      sw.ElapsedMilliseconds);
}
```

Esempio 8.3 – N produttori, M consumatori, buffer circolare (codifica in C#)

```
const int NumDati = 10;
const int DimBuffer = 4;
const int TempoInserimentoBuffer = 200;    // [ms]
const int TempoEstrazioneBuffer = 500;     // [ms]
static SemaphoreSlim vuoto = new SemaphoreSlim(DimBuffer);
static SemaphoreSlim pieno = new SemaphoreSlim(0);
static SemaphoreSlim mutexP = new SemaphoreSlim(1);
static SemaphoreSlim mutexC = new SemaphoreSlim(1);
static int[] buffer = new int[DimBuffer];
static int metti = 0;
static int togli = 0;
static void Produci(int inizio, int fine)
{
    Stopwatch swProd = new Stopwatch();
    int temp;

    swProd.Start();
    for (int i = inizio; i <= fine; i++)
    {
        int threadId = Thread.CurrentThread.ManagedThreadId;
        int quad = i * i;
        vuoto.Wait();

        mutexP.Wait();
        temp = metti;
        metti = (metti + 1) % DimBuffer;
        mutexP.Release();

        Thread.Sleep(TempoInserimentoBuffer);
        buffer[temp] = quad;
        pieno.Release();
        Console.WriteLine("Il thread produttore n. {0} ha inserito il numero {1}",
                          threadId, quad);
    }
    swProd.Stop();
    Console.WriteLine("Tempo impiegato dal produttore: {0} ms",
                          swProd.ElapsedMilliseconds);
}
static void Consuma(int quanti)
{
    Stopwatch swCons = new Stopwatch();
    int temp;
```

```
        swCons.Start();
        for (int i = 1; i <= quanti; i++)
        {
            int threadId = Thread.CurrentThread.ManagedThreadId;
            int quad;
            pieno.Wait();

            mutexC.Wait();
            temp = togli;
            togli = (togli + 1) % DimBuffer;
            mutexC.Release();

            Thread.Sleep(TempoEstrazioneBuffer);
            quad = buffer[temp];
            vuoto.Release();
            Console.WriteLine("Il thread consumatore n. {0} ha estratto il numero {1}",
                              threadId, quad);
        }
        swCons.Stop();
        Console.WriteLine("Tempo impiegato dal consumatore: {0} ms",
                              swCons.ElapsedMilliseconds);
    }
    static void Main(string[] args)
    {
        Console.WriteLine("Due produttori, due consumatori, buffer circolare a {0}
                          celle", DimBuffer);
        Console.WriteLine();
        Stopwatch sw = new Stopwatch();
        sw.Start();
        Parallel.Invoke(
            () => Produci(1, NumDati / 2),
            () => Produci(NumDati / 2 + 1, NumDati),
            () => Consuma(NumDati / 2),
            () => Consuma(NumDati - (NumDati / 2))
        );
        sw.Stop();
        Console.WriteLine("Elaborazione terminata. Tempo trascorso: {0} ms",
                          sw.ElapsedMilliseconds);
    }
```

9 – Problema dei lettori/scrittori

Esempio 9.1 – Prima soluzione (priorità ai lettori)

```
// Accesso in m.e. alla var. NumLettori
Semaphore mutex = new Semaphore (1)

// Accesso al buffer (lettura/scrittura)
Semaphore sincro = new Semaphore (1)

int numLettori = 0
int buffer = 100
/* main */
inizio
    leggi()
    leggi()
    scrivi()
    leggi()
    leggi()
fine


procedura leggi()
int dato
inizio
    inizioLettura()

    dato ← buffer

    fineLettura()

    print(dato)
fine

procedura inizioLettura()
inizio
    P(mutex)
    numLettori++
    if (numLettori == 1) {
       P(sincro)
    end if
    V(mutex)
fine
```

```
procedura fineLettura()
inizio
    P(mutex)
    numLettori--
    if (numLettori == 0)
       V(sincro)
    end if
    V(mutex)
fine


procedura scrivi()
int dato
inizio
    inizioScrittura()

    buffer ← random() // Numero casuale

    fineScritura()
fine

procedura InizioScrittura()
inizio
    P(sincro)
fine

procedura fineScrittura()
inizio
    V(sincro)
fine
```

Esempio 9.1 – Prima soluzione (priorità ai lettori – Codifica in C#)

```
const int MaxProcessi = 100;        // Metà lettori, metà scrittori
const int RitardoLettura = 20;      // ms
const int RitardoScrittura = 10;    // ms

// Regola l'accesso alla variabile NumLettori
static SemaphoreSlim mutex = new SemaphoreSlim(1);
// Regola l'accesso al buffer per sacrivere/leggere
static SemaphoreSlim sincro = new SemaphoreSlim(1);
static int numLettori = 0;
static int buffer = 100;

static void InizioLettura()
```

```
{
    mutex.Wait();
    numLettori++;
    if (numLettori == 1)
        sincro.Wait();
    mutex.Release();

}

static void FineLettura()
{
    mutex.Wait();
    numLettori--;
    if (numLettori == 0)
        sincro.Release();
    mutex.Release();
}

static void InizioScrittura()
{
    sincro.Wait();
}
static void FineScrittura()
{
    sincro.Release();
}

static void Leggi(int idLettore)
{
    int dato;
    InizioLettura();

    if (RitardoLettura > 0)
        Thread.Sleep(RitardoLettura);
    dato = buffer;
    Console.WriteLine("Il lettore n. {0} ha letto il buffer. Valore rilevato: {1}",
       idLettore, dato);

    FineLettura();
}

static void Scrivi(int idScrittore)
{
    int dato;
    InizioScrittura();

    if (RitardoScrittura > 0)
        Thread.Sleep(RitardoScrittura);
    dato = buffer;
    dato++;
    buffer = dato;
    Console.WriteLine("Lo scrittore n. {0} ha modificato il buffer.
      Nuovo valore: {1}", idScrittore, dato);

    FineScrittura();
}

static void Main(string[] args)
{
```

```
            List<Task> lista = new List<Task>();
            for (int i = 1; i <= MaxProcessi / 2; i++)
            {
                int id = i;
                // Avvia un nuovo task lettore
                lista.Add(Task.Factory.StartNew(() => Leggi(id),
                      TaskCreationOptions.LongRunning));
                // Avvia un nuovo task scrittore
                lista.Add(Task.Factory.StartNew(() => Scrivi(id),
                      TaskCreationOptions.LongRunning));
            }

            // Attende la fine di tutti i lettori/scrittori
            Task.WaitAll(lista.ToArray());
        }
```

Esempio 9.2 – Seconda soluzione (priorità agli scrittori)

```
Semaphore semLettura = new Semaphore (1)

// Accesso in m.e. alla var. NumLettori
Semaphore mutexL = new Semaphore (1)

// Accesso in m.e. a NumScrittori
Semaphore mutexS = new Semaphore (1)

// Accesso al buffer (lettura/scrittura)
Semaphore sincro = new Semaphore (1)

int numLettori = 0
int numScrittori = 0
int buffer = 100
/* main */
inizio
    leggi()
    scrivi()
    leggi()
    scrivi()
    leggi()
fine


procedura leggi()
int dato
inizio
    inizioLettura()

    dato ← buffer

    fineLettura()

    print(dato)
fine
```

```
procedura inizioLettura()
inizio
    P(semLettura)
    P(mutexL)
    numLettori++
    if (numLettori == 1) {
       P(sincro)
    end if
    V(mutexL)
    V(semLettura)
fine

procedura fineLettura()
inizio
    P(mutexL)
    numLettori--
    if (numLettori == 0)
       V(sincro)
    end if
    V(mutexL)
fine

procedura scrivi()
int dato
inizio
    inizioScrittura()

    buffer ← random() // Numero casuale

    fineScritura()
fine

procedura InizioScrittura()
inizio
    P(mutexS)
    numScrittori++
    if (numScrittori == 1) {
       P(semLettura)
    end if
    V(mutexS)
    P(sincro)
fine
```

```
procedura fineScrittura()
inizio
    V(sincro)
    P(mutexS)
    numScrittori--
    if (numScrittori == 0)
       V(semLettura)
    end if
    V(mutexS)
fine
```

Esempio 9.2 – Seconda soluzione (priorità agli scrittori – Codifica in C#)

```
const int MaxProcessi = 100;        // Metà lettori, metà scrittori
const int RitardoLettura = 20;      // ms
const int RitardoScrittura = 10;    // ms

// Regola l'accesso alla variabile NumLettori
static SemaphoreSlim mutexL = new SemaphoreSlim(1);
// Regola l'accesso alla variabile NumScrittori
static SemaphoreSlim mutexS = new SemaphoreSlim(1);
// Regola l'accesso al buffer per sacrivere/leggere
static SemaphoreSlim sincro = new SemaphoreSlim(1);

static SemaphoreSlim semLettura = new SemaphoreSlim(1);

static int numLettori = 0;
static int numScrittori = 0;
static int buffer = 100;

static void InizioLettura()
{
    semLettura.Wait();  // Indica un nuovo accesso in lettura
    mutexL.Wait();
    numLettori++;
    if (numLettori == 1)
        sincro.Wait();
    mutexL.Release();
    semLettura.Release();
}

static void FineLettura()
{
    mutexL.Wait();
    numLettori--;
    if (numLettori == 0)
        sincro.Release();
    mutexL.Release();
}

static void InizioScrittura()
{
    mutexS.Wait();
    numScrittori++;
    if (numScrittori == 1)
        semLettura.Wait();
    mutexS.Release();

    sincro.Wait();
}
```

```
static void FineScrittura()
{
    sincro.Release();
    mutexS.Wait();
    numScrittori--;

    if (numScrittori == 0)
        semLettura.Release();

    mutexS.Release();
}

static void Leggi(int idLettore)
{
    int dato;
    InizioLettura();

    if (RitardoLettura > 0)
        Thread.Sleep(RitardoLettura);
    dato = buffer;
    Console.WriteLine("Il lettore n. {0} ha letto il buffer. Valore rilevato: {1}",
       idLettore, dato);

    FineLettura();
}
static void Scrivi(int idScrittore)
{
    int dato;
    InizioScrittura();

    if (RitardoScrittura > 0)
        Thread.Sleep(RitardoScrittura);
    dato = buffer;
    dato++;
    buffer = dato;
    Console.WriteLine("Lo scrittore n. {0} ha modificato il buffer.
      Nuovo valore: {1}", idScrittore, dato);

    FineScrittura();
}
static void Main(string[] args)
{
    List<Task> lista = new List<Task>();
    for (int i = 1; i <= MaxProcessi / 2; i++)
    {
        int id = i;
        // Avvia un nuovo task scrittore
        lista.Add(Task.Factory.StartNew(() => Scrivi(id),
             TaskCreationOptions.LongRunning));
        // Avvia un nuovo task lettore
        lista.Add(Task.Factory.StartNew(() => Leggi(id),
             TaskCreationOptions.LongRunning));
    }

    // Attende la fine di tutti i lettori/scrittori
    Task.WaitAll(lista.ToArray());
}
```

Esempio 9.3 – Terza soluzione (lettori e scrittori gestiti equamente)

```
Semaphore semLinea = new Semaphore (1)

// Accesso in m.e. alla var. NumLettori
Semaphore mutex = new Semaphore (1)

// Accesso al buffer (lettura/scrittura)
Semaphore sincro = new Semaphore (1)

int numLettori = 0
int buffer = 100

/* main */
inizio
    leggi()
    scrivi()
    leggi()
    scrivi()
    leggi()
fine


procedura leggi()
int dato
inizio
    inizioLettura()

    dato ← buffer

    fineLettura()

    print(dato)
fine
```

```
procedura inizioLettura()
inizio
    P(semLinea)
    P(mutex)
    numLettori++
    if (numLettori == 1) {
       P(sincro)
    end if
    V(mutex)
    V(semLinea)
fine

procedura fineLettura()
inizio
    P(mutex)
    numLettori--
    if (numLettori == 0)
       V(sincro)
    end if
    V(mutex)
fine

procedura scrivi()
int dato
inizio
    inizioScrittura()

    buffer ← random() // Numero casuale

    fineScritura()
fine

procedura InizioScrittura()
inizio
    P(semLinea)
    P(sincro)
    V(semLinea)
fine

procedura fineScrittura()
inizio
    V(sincro)
fine
```

Esempio 9.3 – Terza soluzione (lettori e scrittori gestiti equamente – Codifica in C#)

```
const int MaxProcessi = 100;        // Metà lettori, metà scrittori
const int RitardoLettura = 20;      // ms
const int RitardoScrittura = 10;    // ms

// Regola l'accesso alla variabile NumLettori
static SemaphoreSlim mutex = new SemaphoreSlim(1);
// Regola l'accesso al buffer per sacrivere/leggere
static SemaphoreSlim sincro = new SemaphoreSlim(1);

static SemaphoreSlim semLinea = new SemaphoreSlim(1);

static int numLettori = 0;
```

```
static int buffer = 100;

static void InizioLettura()
{
    semLinea.Wait();
    mutex.Wait();
    numLettori++;
    if (numLettori == 1)
        sincro.Wait();
    mutex.Release();
    semLinea.Release();

}

static void FineLettura()
{
    mutex.Wait();
    numLettori--;
    if (numLettori == 0)
        sincro.Release();
    mutex.Release();
}

static void InizioScrittura()
{
    semLinea.Wait();
    sincro.Wait();
    semLinea.Release();
}
static void FineScrittura()
{
    sincro.Release();
}

static void Leggi(int idLettore)
{
    int dato;
    InizioLettura();

    if (RitardoLettura > 0)
        Thread.Sleep(RitardoLettura);
    dato = buffer;
    Console.WriteLine("Il lettore n. {0} ha letto il buffer. Valore rilevato: {1}",
      idLettore, dato);

    FineLettura();
}
static void Scrivi(int idScrittore)
{
    int dato;
    InizioScrittura();

    if (RitardoScrittura > 0)
        Thread.Sleep(RitardoScrittura);
    dato = buffer;
    dato++;
    buffer = dato;
    Console.WriteLine("Lo scrittore n. {0} ha modificato il buffer.
             Nuovo valore: {1}", idScrittore, dato);
```

```
        FineScrittura();
    }

    static void Main(string[] args)
    {
        List<Task> lista = new List<Task>();
        for (int i = 1; i <= MaxProcessi / 2; i++)
        {
            int id = i;
            // Avvia un nuovo task lettore
            lista.Add(Task.Factory.StartNew(() => Leggi(id),
                 TaskCreationOptions.LongRunning));
            // Avvia un nuovo task scrittore
            lista.Add(Task.Factory.StartNew(() => Scrivi(id),
                 TaskCreationOptions.LongRunning));
        }

        // Attende la fine di tutti i lettori/scrittori
        Task.WaitAll(lista.ToArray());
    }
```

Appendice

A1 – Acquisizione di risorse web

Spesso le moderne applicazioni elaborano dati prelevandoli dalla rete locale oppure da internet. I dati sono scaricati sotto forma di file (pagine web, documenti, immagini) oppure di stringhe (dati codificati in formati testuali come XML e JSON).

In C# è possibile scaricare le informazioni in diversi modi. Uno di questi utilizza la classe *WebClient* (namespace *System.Net*).

La classe WebClient offre due metodi:

- Metodo *DownloadFile(url)*: scarica e memorizza un file contenente la risorsa web specificata dal parametro *url*;
- Metodo *DownloadString(url)*: restituisce una stringa contenente la risorsa web specificata dal parametro *url*;

Un esempio d’uso della classe *WebClient* è il seguente:

```
try
{
    using (WebClient c = new WebClient())
    {
        const string URL1 = "https://www.netlab.fauser.edu/s/esempi/saluti.txt";
        const string URL2 = "https://www.netlab.fauser.edu/s/esempi/logofauser.png";
        string s;

        s = c.DownloadString(URL1);
        Console.WriteLine("Output: {0}", s);
        c.DownloadFile(URL2, "logo.png");
        Console.WriteLine("Immagine scaricata correttamente.");

    }
}
catch (Exception e)
{
    Console.Error.WriteLine("Errore: {0}", e.Message);
}
```

L’istanza *c*, definita con la clausola *using*, è utilizzata all’interno di un blocco *try...catch* in modo da gestire le eccezioni dovute agli errori che si possono verificare durante il trasferimento della risorsa (file inesistente, url errato, ecc.).

In un programma concorrente, i download che richiedo molto tempo possono essere inseriti in uno specifico thread oppure, in TPL, in un nuovo task: in questo caso si migliorano le prestazioni avviando il task con l’opzione `TaskCreationOptions.LongRunning`:

```
Task tsk = Task.Factory.StartNew(Proc, TaskCreationOptions.LongRunning);
```

A2 – Acquisizione di dati in formato JSON

Le risorse prelevabili dalla rete non sono composte solamente da file o semplici stringhe, ma possono dati dati strutturati come vettori, matrici, oggetti. I server che offrono questo tipo di risorsa solitamente provvedono a trasformare, prima di procedere all’invio, i dati richiesti in formato testuale, in modo da semplificarne la ricezione da parte del richiedente. Tra i formati più utilizzati ricordiamo XML (*Extensible Markup Language*) e JSON (*JavaScript Object Notation*).

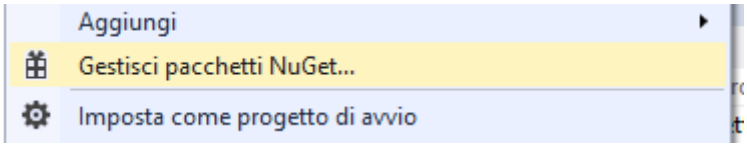
Il seguente URL: `https://www.netlab.fauser.edu/s/esempi/utente.php` permette di acquisire alcuni dati di un utente in formato JSON:

```
{"matricola":18453,"cognome":"Rossi","nome":"Mario","abilitato":true,"telefoni":["339-123456","333-654321"]}
```

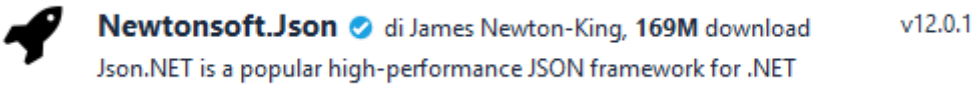
Quando un server invia un dato complesso in formato JSON, un’applicazione C# riceve una stringa di testo che deve essere convertita in un’apposita struttura dati (nel nostro caso useremo un oggetto di tipo *dinamico*).

Sebbene esistano numerosi strumenti in grado di processare stringhe JSON, noi useremo **Newtonsoft Json.Net**, un componente software famoso per la sua versatilità e semplicità d’uso. Purtroppo Json.Net non è incluso nelle librerie standard del framework .NET e deve essere installato manualmente nel progetto ricorrendo al gestore di pacchetti **NuGet**.

Per installare *Json.Net* in un progetto Visual Studio, fare clic con il tasto destro del mouse sul nome del progetto, quindi selezionare *Gestisci pacchetti NuGet*:



Fare clic sulla scheda *Sfoglia*, quindi ricercare il componente digitando *json.net*, selezionare successivamente **Netwonsoft.Json** e infine installare il componente. Al termine dell’operazione, la nuova classe è memorizzata all’interno del progetto e può essere richiamata direttamente nel codice.



Il codice seguente scarica dalla rete l’anagrafica di un utente in formato testuale (un’unica stringa in formato JSON), converte la risposta in un oggetto dinamico utilizzando *Jobject.Parse* (metodo del componente *Json.Net*), copia i singoli campi in opportune variabili e infine stampa i dati sullo schermo.

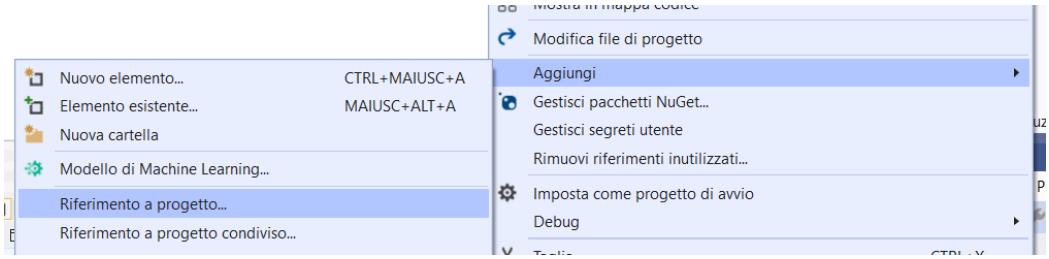
```
try
{
    using (WebClient c = new WebClient())
    {
        const string URL = "https://www.netlab.fauser.edu/s/esempi/utente.php";
        string s;
        string nome, cognome;
        int matricola;
        bool abilitato;
        string[] telefoni;

        s = c.DownloadString(URL);
        dynamic u = JObject.Parse(s);      // Richiede using Newtonsoft.Json.Linq;
        // Si copiano i dati contenuti nell'oggetto u in variabili
        // locali eseguendo, se richiesto, il cast esplicito
        nome = u.nome;
        cognome = u.cognome;
        matricola = u.matricola;
        abilitato = (bool)u.abilitato;
        telefoni = u.telefoni.ToObject<string[]>();

        Console.WriteLine("Utente: {0} {1}", nome, cognome);
        Console.WriteLine("Matricola: {0}", matricola);
        if (abilitato == true)
            Console.WriteLine("Utente abilitato");
        Console.WriteLine("Telefoni");
        for (int i = 0; i < telefoni.Length; i++)
            Console.WriteLine("   {0}", telefoni[i]);
    }
}
catch (Exception e)
{
    Console.Error.WriteLine("Errore: {0}", e.Message);
}
```

A3 – Uso della libreria “Ponte”

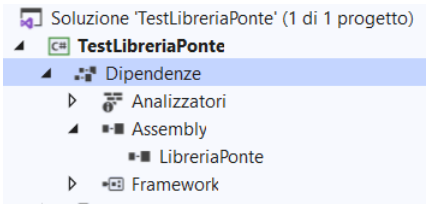
La libreria “Ponte” consente di mostrare, all’interno di un’applicazione WPF, il transito di alcuni veicoli su un ponte. Essa è un utile strumento nella risoluzione di uno degli esercizi svolti durante il corso.

Per utilizzare la libreria, è innanzitutto necessario scaricare l’apposito file compresso[1] dal sito web del corso e decomprimere il contenuto in una cartella temporanea. La cartella include il file *LibreriaPonte.dll*.

All’interno di un progetto WPF creato con Visual Studio 2019, aggiungere un nuovo riferimento alla libreria: fare click con il tasto destro del mouse sul nome del progetto, quindi scegliere **Aggiungi → Riferimento a progetto…** e sfogliare le cartelle per individuare e selezionare il file *LibreriaPonte.dll*.



Verificare che all’interno del progetto, nella sezione **Dipendenze → Assembly**, sia presente la voce **LibreriaPonte**.



Successivamente, svolgere le seguenti operazioni:

1. Inserire nella finestra WPF un nuovo controllo grafico di tipo **Border** e impostare le seguenti proprietà: *BorderThickness="1" BorderBrush="black" Width="752" Height="444" ClipToBounds="True"* . Questo controllo riproduce una piccola cornice in cui verrà visualizzato il ponte. Assicurarsi che la larghezza e l’altezza della finestra siano sufficientemente grandi per contenere il bordo e, nel caso in cui il controllo sia inserito in una griglia, che la riga e la colonna di appartenenza abbiano dimensioni adeguate.
2. Inserire all’interno del bordo precedente un controllo grafico di tipo **Canvas** e assegnargli il nome ***Canvas1***. Il codice XAML risultante è:

```
<Border BorderThickness="1" Width="752" Height="444" ClipToBounds="True"
        BorderBrush="black">

   <Canvas x:Name="Canvas1"/>

</Border>
```

1 Il file “*Libreria per il ‘Problema del Ponte’*” è scaricabile in formato .zip dalla sezione TPSIT della pagina web *http://www.fauser.edu/~fuligni/quarte.html*

3. All’interno del file .cs associato alla finestra WPF, dopo aver inserito all’inizio del codice l’istruzione `using LibreriaPonte;` dichiarare una variabile globale *p* di tipo **Ponte**:

   ```
   Ponte p;
   ```

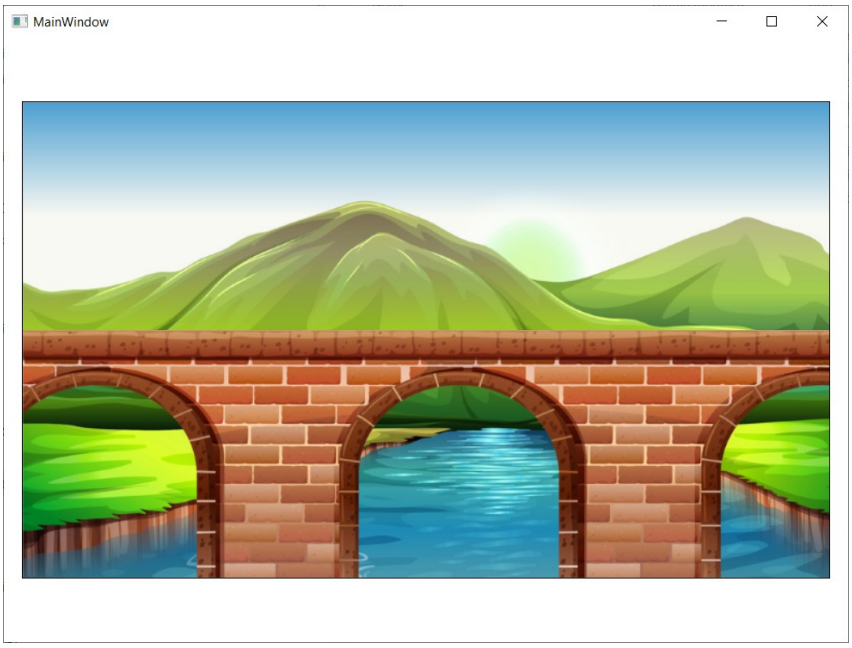
4. Inizializzare la variabile *p* nella routine associata all’evento *ContentRendered* della finestra:

   ```
   private void Window_ContentRendered(object sender, System.EventArgs e)
   {
       p = new Ponte(Canvas1);
   }
   ```

5. Rilasciare l’oggetto *p* nella routine associata all’evento *Closed* della finestra:

   ```
   private void Window_Closed(object sender, System.EventArgs e)
   {
       p.Dispose();
   }
   ```

Eseguire l’applicazione e verificare che il ponte sia visualizzato correttamente.



Transito degli autoveicoli

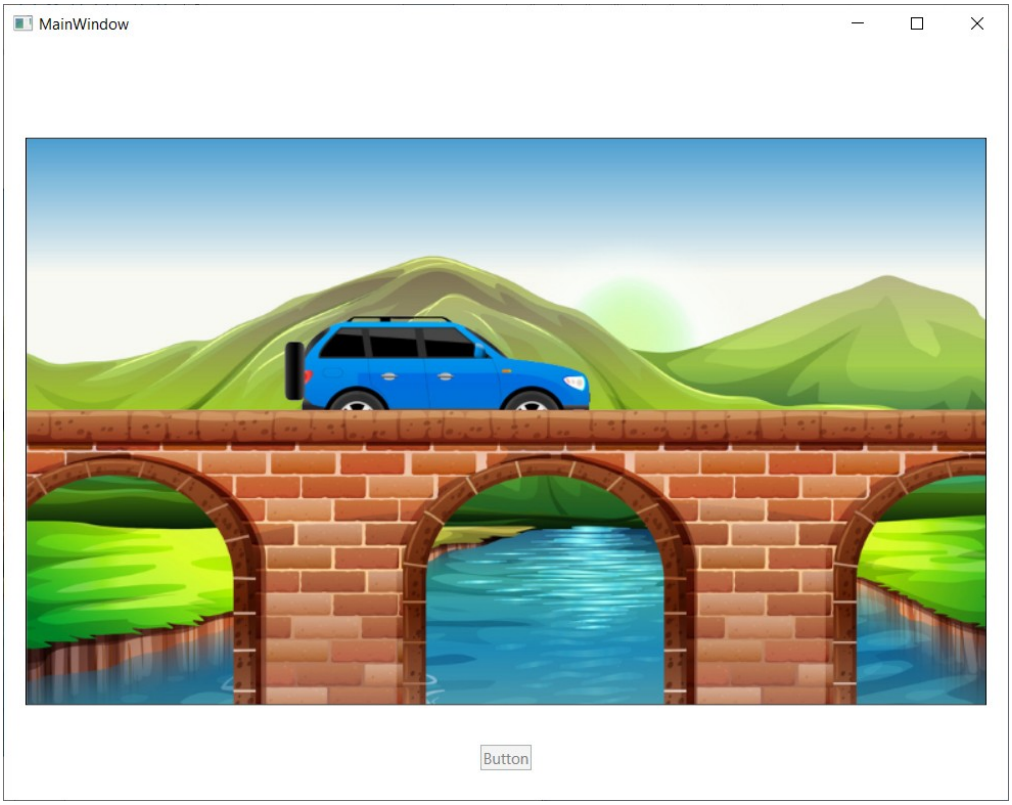
La libreria riproduce il transito di due tipi di autoveicoli: autovetture e camion. Un autoveicolo può transitare sul ponte verso destra (partendo dall’estremità sinistra del ponte) oppure verso sinistra (a partire dall’estremità destra). È possibile far transitare sul ponte più veicoli contemporaneamente, anche in entrambe le direzioni.

La classe *Ponte* dispone di due metodi *AggiungiAuto* e *AggiungiCamion* con cui si dà inizio al transito di uno dei due tipi di autoveicolo. I metodi ricevono in input la direzione di transito, la quale può assumere il valore **Ponte.Direzione.versoDestra** oppure **Ponte.Direzione.versoSinistra**. I metodi restituiscono un semaforo di tipo *SemaphoreSlim* su cui è possibile rimanere in attesa della fine del transito dell’autoveicolo.

I metodi della classe *Ponte* possono essere eseguiti dal thread UI oppure da un thread di elaborazione mediante *Dispatcher.Invoke*. La libreria utilizza thread interni per non impegnare eccessivamente il thread UI.

Il seguente codice fa transitare un’autovettura sul ponte da sinistra verso destra e ne attende la fine, disabilitando il bottone durante l’animazione:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    btn.IsEnabled = false;
    Task.Factory.StartNew(() =>
    {
        SemaphoreSlim s = null;
        Dispatcher.Invoke(() =>
        {
            s = p.AggiungiAuto(Ponte.Direzione.versoDestra);
        });

        // Attende la fine del transito (senza bloccare il thread UI)
        s.Wait();

        Dispatcher.Invoke(() =>
        {
            btn.IsEnabled = true;
        });
    });
}
```



Esercizi

1. [Soluzione **ElementiCasuali**] – Si considerino le seguenti due attività:

   a) Generazione di 5 000 000 numeri casuali compresi tra -10 e 20 (seed 2018) e memorizzazione dei valori in un array A; ordinamento crescente di A; determinazione della media aritmetica *m* dei soli valori contenuti nella prima metà di A (i primi 2 500 000 elementi); visualizzazione del messaggio "`Media degli elementi estratti da A:` " seguito dal valore di *m*.

   b) Generazione di 3 000 000 numeri casuali compresi tra 51 e 100 (seed 2019) e memorizzazione dei valori in un array B; ordinamento crescente di B; determinazione del numero *n* di elementi dispari presenti nella prima metà di B; stampa del messaggio "`Elementi dispari estratti da B:` " seguito dal valore di *n*.

   Scrivere un programma **Progr1** che esegua le due attività in modo sequenziale riportando il tempo impiegato T_1. Scrivere successivamente due programmi **Progr2** e **Progr3** che eseguano le stesse attività usando un algoritmo parallelo (basato rispettivamente sulla classe Thread e sulla libreria TPL) mostrando il tempo impiegato (rispettivamenteT_2 e T_3). Determinare lo speedup per ciascuno degli algoritmi paralleli.

   [R. m ≈ -2,746; n = 779423]

2. [Soluzione **RadiciQuadrate**] – Si vuole realizzare un'applicazione che risolva due problemi matematici utilizzando la funzione *sqrt*. Per evidenziare i vantaggi della programmazione concorrente, supporremo che il calcolo della radice quadrata di un numero sia un'operazione onerosa che richiede un tempo di elaborazione di 50 ms (nel codice si aggiungano opportuni ritardi per simulare il maggiore costo computazionale delle istruzioni contenenti una radice quadrata). L'applicazione deve richiedere l'inserimento da tastiera di due numeri interi A, B positivi (si controlli la validità dei dati immessi) e successivamente eseguire le seguenti attività:

   - Determinare le misure dei cateti $C_1 = 6.53 * A$, $C_2 = 2.85 * B$ e dell'ipotenusa I di un triangolo rettangolo.
   - Posto C = - (B + 10), determinare il Δ dell'equazione $Ax^2 + Bx + C = 0$ e le corrispondenti soluzioni reali X_1 e X_2.

   L'applicazione deve infine riportare su schermo i risultati ottenuti e il tempo totale di elaborazione (per evitare che il ritardo dovuto all'uso della tastiera influisca sulle rilevazioni, si consiglia di avviare il cronometro dopo la lettura dei dati di input).

   a) Scrivere un progetto **RQSequenziale** contenente l'applicazione richiesta utilizzando un algoritmo sequenziale.

   b) Scrivere un progetto **RQParallelo** contenente la versione concorrente dell'applicazione. Confrontare i tempi ottenuti e calcolare lo *speedup*.

   L'output prodotto dai progetti deve rispettare il seguente formato (i lati del triangolo devono essere visualizzati con due cifre decimali, le soluzioni dell'equazione con tre):

```
A: 4
B: 5

Tempo di elaborazione [ms]: 58

Risultati:
  Triangolo rettangolo di cateti 26,12; 14,25 e ipotenusa 29,75
  Soluzioni dell'equazione di secondo grado: x1 = -2,660; x2 = 1,410
```

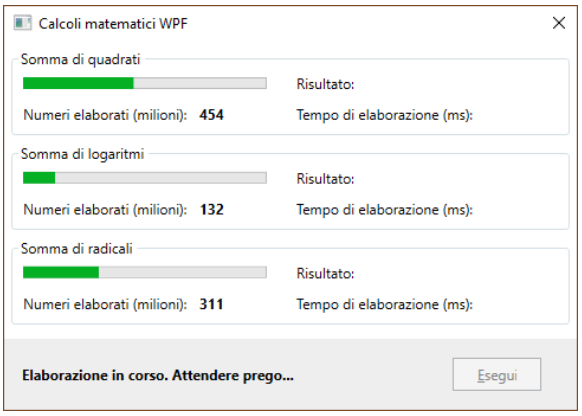
3. [Soluzione **CalcoliMatematici**] – Data la costante numerica N = 1 000 000 000, si considerino le seguenti attività di calcolo:
   - Determinazione e stampa della somma dei quadrati dei primi N numeri interi positivi: $SQ = 1^2 + 2^2 + 3^2 + ... + N^2$
   - Determinazione e stampa della somma dei logaritmi in base 10 dei primi N numeri interi positivi: $SL = \log_{10}(1) + \log_{10}(2) + \log_{10}(3) + ... + \log_{10}(N)$
   - Determinazione e stampa della somma delle radici quadrate dei primi N numeri interi positivi: SR = sqrt(1) + sqrt(2) + sqrt(3) + ... + sqrt(N)

   Ogni attività termina con la stampa del tempo impiegato per ottenere il risultato (si utilizzino tre "cronometri" distinti).

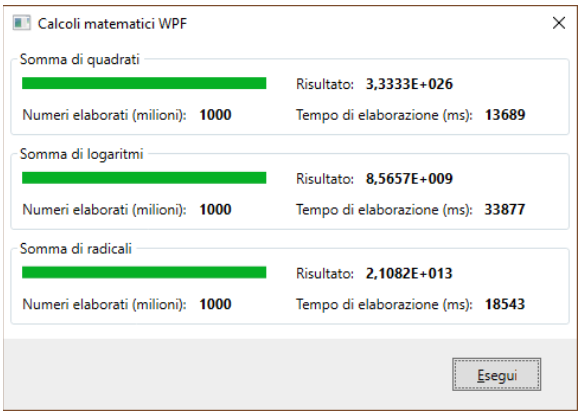
   a) Realizzare un'applicazione concorrente **CalcoliJoinCount** in C# (di tipo *console*) che esegua le tre attività in parallelo implementando il costrutto *join(count)*.

   b) Realizzare un'applicazione concorrente **CalcoliCobegin** in C# (di tipo *console*) che esegua le tre attività in parallelo implementando il costrutto *cobegin/coend*.

4. [Soluzione **CalcoliMatematiciWPF**] – Risolvere il problema precedente sviluppando un'applicazione WPF che implementi il costrutto cobegin/coend mediante la libreria TPL. Le fasi dell'elaborazione (stato di avanzamento, risultati finali) devono essere mostrate con la seguente interfaccia grafica.



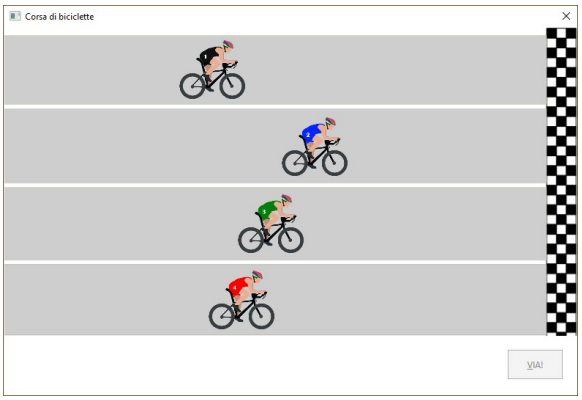
Elaborazione in corso



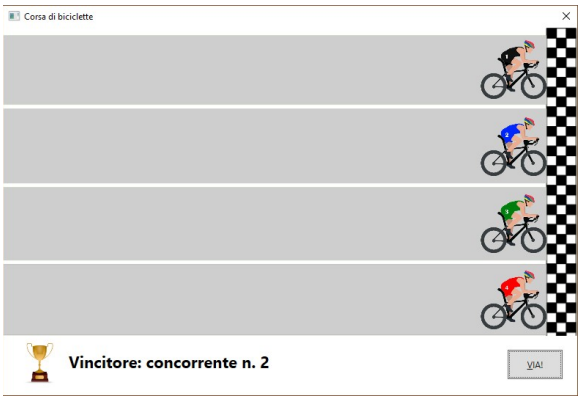
Risultati finali

5. [Soluzione **CorsaBiciclette**] – Realizzare un'applicazione WPF che simuli una gara tra quattro ciclisti. L'applicazione, dopo aver generato un vettore di quattro numeri casuali rappresentanti le velocità, simula

la gara utilizzando un thread per ciascun ciclista e riproducendo sul monitor gli spostamenti in base alle velocità assegnate. Al termine della gara, il programma comunica il codice del vincitore.



Gara in corso...



Proclamazione del vincitore

6. [Soluzione **Negozio**] – Scrivere un programma concorrente **Negozio** per simulare un'attività commerciale. Il negozio è caratterizzato da uno stato (aperto/chiuso) e da una capacità massima di 5 clienti. L'attività è composta da tre processi concorrenti:

   a) Processo **EntraCliente**: esegue ripetutamente le seguenti attività: attende 1,5 secondi, quindi se il negozio è aperto e il numero di clienti presenti in quel momento è minore della capacità massima, simula l'ingresso di un nuovo cliente nel negozio. Se il negozio è chiuso, il processo termina.

   b) Processo **EsceCliente**: esegue ripetutamente le seguenti attività: attende 5,5 secondi, quindi, se il negozio è aperto e nel locale vi è almeno un cliente, simula l'uscita di un cliente, il quale paga al commerciante la somma di 20 Euro. Il processo termina quando il negozio è chiuso e non ci sono più clienti nel locale.

   c) Processo **ChiudiNegozio**: attende 15 secondi, poi chiude il negozio, indipendentemente dal numero di clienti ancora presenti (i clienti possono uscire anche quando il negozio è chiuso).

   I processi comunicano le attività svolte visualizzando appositi messaggi sullo schermo. Il programma, dopo aver atteso la fine dei tre processi, stampa il ricavo giornaliero del negozio.

```
Negozio aperto.
Un nuovo cliente entra in negozio. I clienti sono ora 1.
Un nuovo cliente entra in negozio. I clienti sono ora 2.
Un nuovo cliente entra in negozio. I clienti sono ora 3.
Un cliente paga ed esce. Ora i clienti sono 2 e il ricavo è di 20 Euro.
Un nuovo cliente entra in negozio. I clienti sono ora 3.
Un nuovo cliente entra in negozio. I clienti sono ora 4.
Un nuovo cliente entra in negozio. I clienti sono ora 5.
Negozio affollato, impossibile entrare. I clienti sono ora 5.
Un cliente paga ed esce. Ora i clienti sono 4 e il ricavo è di 40 Euro.
Un nuovo cliente entra in negozio. I clienti sono ora 5.
Negozio affollato, impossibile entrare. I clienti sono ora 5.
Negozio chiuso.
Un cliente paga ed esce. Ora i clienti sono 4 e il ricavo è di 60 Euro.
Un cliente paga ed esce. Ora i clienti sono 3 e il ricavo è di 80 Euro.
Un cliente paga ed esce. Ora i clienti sono 2 e il ricavo è di 100 Euro.
Un cliente paga ed esce. Ora i clienti sono 1 e il ricavo è di 120 Euro.
Un cliente paga ed esce. Ora i clienti sono 0 e il ricavo è di 140 Euro.
Ricavo giornaliero: 140 Euro
Premere un tasto per continuare . . .
```

7. [Soluzione **TreVettori**] – Dati tre vettori di numeri interi VA, VB, VC, ciascuno di lunghezza 20, si considerino le seguenti attività da eseguire concorrentemente:
   - Attività 1: memorizzazione di valori casuali positivi minori di 100 nel vettore VA (seed 2018) e stampa su schermo dei contenuti dei tre vettori.
   - Attività 2: memorizzazione di valori casuali positivi minori di 50 nel vettore VB (seed 2019) e stampa su schermo del più grande valore contenuto nei tre vettori.
   - Attività 3: memorizzazione di valori casuali positivi minori di 75 nel vettore VC (seed 2020) e stampa su schermo del conteggio di tutti i numeri dispari contenuti nei tre vettori.
8. [Soluzione **Multipli5ProdCons**] – Scrivere un'applicazione console che, utilizzando il modello del produttore/consumatore, generi 720 numeri interi casuali minori di 1000 e li memorizzi in un apposito buffer. Successivamente, l'applicazione estrae i numeri dal buffer e stampa su monitor solo quelli che sono multipli di 25. Si supponga che l'inserimento di un numero nel buffer richieda 6 ms e l'estrazione dal buffer 4 ms.
   Sviluppare l'applicazione in tre varianti:
   - [Progetto **Multipli5BS**]: un produttore, un consumatore e buffer singolo;
   - [Progetto **Multipli5BC**]: un produttore, un consumatore e buffer circolare a 6 celle;
   - [Progetto **Multipli5NM**]: 4 produttori, 3 consumatori e buffer circolare a 8 celle (suggerimento: usare task di tipo *long running*);

   Per ogni variante è richiesta la stampa di: tempo totale di elaborazione; tempo impiegato dai soli processi produttori; tempo impiegato dai soli processi consumatori.
9. [Soluzione **TemperatureCasa**] – Una casa moderna è dotata di quattro sensori in grado di rilevare la temperatura dei locali in gradi Celsius. I sensori sono dotati di NR registratori per la memorizzazione delle temperature in appositi buffer di memoria e di NV visori per la visualizzazione di alcune statistiche sulle temperature rilevate. Si vuole sviluppare un'applicazione concorrente che simuli il funzionamento di registratori e visori (ciascun componente è riprodotto da un thread distinto).

   Il thread "registratore" è caratterizzato da un codice intero univoco **idReg** e ripete a ciclo continuo le seguenti attività:

   a) attende un tempo casuale compreso tra 1500 e 2000 ms;
   b) modifica le temperature presenti nel buffer secondo il seguente algoritmo:
      - se *idReg* è pari, determina un numero casuale A compreso tra 0.5 e 1.3 °C e aumenta tutte le temperature nel buffer di un valore pari a A;
      - se *idReg* è dispari, determina un numero casuale D compreso tra 0.6 e 1.1 °C e diminuisce tutte le temperature nel buffer di un valore pari a D;
   c) stampa un messaggio contenente *idReg* e il valore (A oppure D) usato nell'aggiornamento dei dati. Si suppone che l'attività di aggiornamento richieda complessivamente 750 ms.

   Il thread "visore" è caratterizzato da un codice intero univoco **idVis** e ripete a ciclo continuo le seguenti

attività:

a) attende un tempo casuale compreso tra 250 e 1000 ms;

b) elabora e stampa su schermo alcune statistiche secondo il seguente algoritmo:

   - se *idVis* è pari, determina la media artimetica delle temperature presenti nel buffer;
   - se *idVis* è dispari, determina le temperature minima e massima;

c) stampa un messaggio contenente *idVis* e la statistica prodotta (media oppure min/max) . Si suppone che l'elaborazione delle statistiche richieda complessivamente 450 ms.

Sviluppare un'applicazione console concorrente che, dopo aver inizializzato il buffer delle temperature con i valori (16.5, 18.4, 20.2, 18.1), risolva il problema secondo i seguenti modelli:

- [Progetto **TemperatureCasa1**] – Lettori/scrittori con precedenza ai lettori, eseguendo l'applicazione inizialmente con NR = 2 registratori e NV = 2 visori. Successivamente, eseguire l'applicazione con NV = 16 e verificare l'eventuale presenza di *starvation* dei registratori.
- [Progetto **TemperatureCasa2**] – Lettori/scrittori con gestione equa di lettori e scrittori, eseguendo l'applicazione con NR = 2 registratori e NV = 16 visori. Si può ancora osservare il fenomeno dello *starvation*?

```
Registratore n. 2: le temperature sono aumentate di 0,5 °C
Visore n. 1: temperatura min = 16,6 °C; temperatura max = 20,3 °C
Visore n. 2: temperatura media = 18,4 °C
Registratore n. 1: le temperature sono diminuite di 1,0 °C
Visore n. 2: temperatura media = 17,4 °C
Visore n. 1: temperatura min = 15,6 °C; temperatura max = 19,3 °C
Registratore n. 2: le temperature sono aumentate di 0,7 °C
Visore n. 1: temperatura min = 16,3 °C; temperatura max = 20,0 °C
Visore n. 2: temperatura media = 18,1 °C
Registratore n. 1: le temperature sono diminuite di 0,6 °C
Visore n. 1: temperatura min = 15,7 °C; temperatura max = 19,4 °C
Visore n. 2: temperatura media = 17,5 °C
Registratore n. 2: le temperature sono aumentate di 0,7 °C
Visore n. 2: temperatura media = 18,2 °C
Visore n. 1: temperatura min = 16,4 °C; temperatura max = 20,1 °C
Registratore n. 1: le temperature sono diminuite di 0,8 °C
Visore n. 1: temperatura min = 15,6 °C; temperatura max = 19,3 °C
Visore n. 2: temperatura media = 17,4 °C
```

Applicazioni proposte

I. [Soluzione **CorrettoreWeb**] – Una scuola partecipa a una gara di informatica in cui gli studenti, suddivisi per classe (ogni classe partecipa a una gara diversa), compilano un questionario composto da un certo numero di domande. A ogni domanda corrispondono quattro possibili risposte, contrassegnate dalle lettere A, B, C, D, di cui una sola è corretta.

Il questionario è somministrato in modalità online: lo studente usa un'apposita piattaforma software per rispondere a ciascuna domanda, digitando la lettera corrispondente alla risposta ritenuta corretta oppure il carattere '.' (punto) nel caso in cui preferisca non rispondere. Alla fine della compilazione le risposte sono memorizzate localmente (cioè nello stesso PC usato dallo studente).

Conclusa la gara, si procede alla vautazione delle risposte calcolando il punteggio totale di ogni studente secondo le seguenti regole: a ogni risposta corretta si assegnano 4 punti allo studente; a ogni risposta errata si toglie un punto; nessun punto è assegnato in caso di risposta mancante.

Per semplificare la gestione del questionario, la piattaforma dispone di un'applicazione web in grado di offrire due servizi:

(a) *Acquisizione dell'elenco dei partecipanti*: permette di ottenere, per una data classe, le seguenti informazioni:

- Nome della classe
- Numero totale di quesiti presenti nel questionario
- Stringa delle soluzioni ai quesiti
- Numero di punti assegnati in caso di risposta corretta, errata o mancante
- Numero di alunni partecipanti
- Dati degli alunni (vettore di record, ciascun record include: matricola, cognome e nome)

Il servizio è ottenibile specificando il seguente URL:

```
https://www.netlab.fauser.edu/s/correttore/elenco.php?classe=1A
```

contenente alla fine il nome della classe desiderata (in questo caso 1A). L'elenco è inviato in formato JSON. Il sistema riconosce attualmente tre classi: 1A, 1B, 1C (i dati degli studenti sono fittizi).

(b) *Acquisizione delle risposte dello studente*: questo servizio si collega al PC dello studente (individuato sulla rete a partire dalla matricola dell'alunno), attende il completamento del questionario e restituisce la stringa contenente le lettere corrispondenti alle risposte indicate. Le risposte sono prelevabili attraverso l'URL:

```
https://www.netlab.fauser.edu/s/correttore/elaborato.php?matricola=20215
```

in cui la matricola è specificata alla fine della stringa (in questo caso si acquisiscono le risposte dell'alunno di matricola 20215).

Si vuole scrivere una nuova applicazione che, utilizzando i due servizi precedentemente descritti, effettui la correzione automatica dei questionari riportando il punteggio totale di ciascun alunno. In particolare, l'applicazione deve:

- richiedere il nome della classe da correggere, quindi acquisire l'elenco contenente i dati degli

alunni, le soluzioni e i punteggi previsti utilizzando il primo servizio;
- a partire delle matricole ottenute, contattare - uno a uno - tutti i PC utilizzati dalla classe, recuperare le risposte dell'alunno utilizzando il secondo servizio, elaborare il punteggio finale e visualizzarlo sullo schermo insieme al tempo impiegato per scaricare e correggere il questionario;
- (facoltativo) determinare e visualizzare la classifica finale della classe, ordinata in modo decrescente in base al punteggio ottenuto.

L'applicazione deve essere sviluppata in due varianti:

(1) progetto **CorrettoreS**: l'applicazione risolve il problema con un algoritmo sequenziale;

(2) progetto **CorrettoreP**: l'applicazione, di tipo concorrente, risolve il problema esprimendo il massimo grado di parallelismo (si suggerisce di aumentare il numero massimo di connessioni TCP simultanee supportate inserendo all'inizio del programma l'istruzione `ServicePointManager.DefaultConnectionLimit = 100;`).

È richiesto inoltre il calcolo dello speedup.

Esempio di output:

```
CORRETTORE PARALLELO

Classe da correggere: 1A
Correzione di 3 elaborati in corso. Attendere prego...
Elaborato n.  3 Punteggio: 19  Tempo [s]: 7,87
Elaborato n.  2 Punteggio: 75  Tempo [s]: 7,89
Elaborato n.  1 Punteggio: 36  Tempo [s]: 7,95
Correzione completata.

                 CLASSIFICA FINALE

Pos. Matricola  Cognome      Nome         Punteggio
-------------------------------------------------
 1.    20002    Bardi        Alessandro       54
 2.    20001    Antoni       Stefano          34
 3.    20003    Boes         Antonio          21

Tempo impiegato [s]: 8,27
```