Programmazione Parallela e Asincrona

Programmazione Parallela e Asincrona

Fino a questo momento abbiamo utilizzato solamente la programmazione SINCRONA, ovvero:

- Le istruzioni sono eseguite una alla volta, in sequenza
- Ogni istruzione può essere eseguita solo quando la precedente ha terminato l'esecuzione
- Una istruzione che richiede molto tempo ritarda l'esecuzione di tutte le istruzioni successive

Ci sono situazioni in cui c'è la necessità/l'utilità di eseguire più istruzioni contemporaneamente,

sfruttando le architetture parallele dei pc moderni.

Casi d'uso della programmazione parallela

- Rendere più veloci algoritmi che possono essere eseguiti in parallelo:
  - Render di immagini
  - Calcoli matematici
- Rendere più responsive le applicazioni
  - Dividendo il codice che si occupa della UI con quello che esegue operazioni lunghe, lasciando sempre operativa la UI

.NET e C# supportano tutti e tre i paradigmi: Sincroni, Parallelo e Asincrono

Architetture parallele

Ogni elaboratore è caratterizzato da quante istruzioni può eseguire in parallelo. E' un parametro che dipende da:

- Numero di CPU installate
- Numero di Core in ogni CPU
- Numero di Thread gestiti da ogni core

Agli estremi abbiamo quindi:

- Architetture con singola CPU, singolo Core e singolo Thread
- Architetture multi CPU, multi Core, multi thread

NB: Esiste anche la programmazione parallela che utilizza più elaboratori (non ne parleremo)

Applicazioni Parallele

- Ogni applicazione è rappresentata da un Processo. Ogni volta che avviamo un programma il sistema operativo crea un processo.
- Ad ogni processo è assegnata un'area di memoria ed è completamente isolato da tutti gli altri processi.
    - La comunicazione può avvenire solo con modalità "Iter Process"
- Ogni processo contiene uno o più Thread (quello sempre presente è detto Principale o Main)
- I thread condividono la memoria e possono usarla per comunicare / sincronizzarsi
- I thread sono eseguiti in parallelo e in modo indipendente l'uno dall'altro (salvo le sincronizzazioni imposte via codice)

Applicazioni Parallele

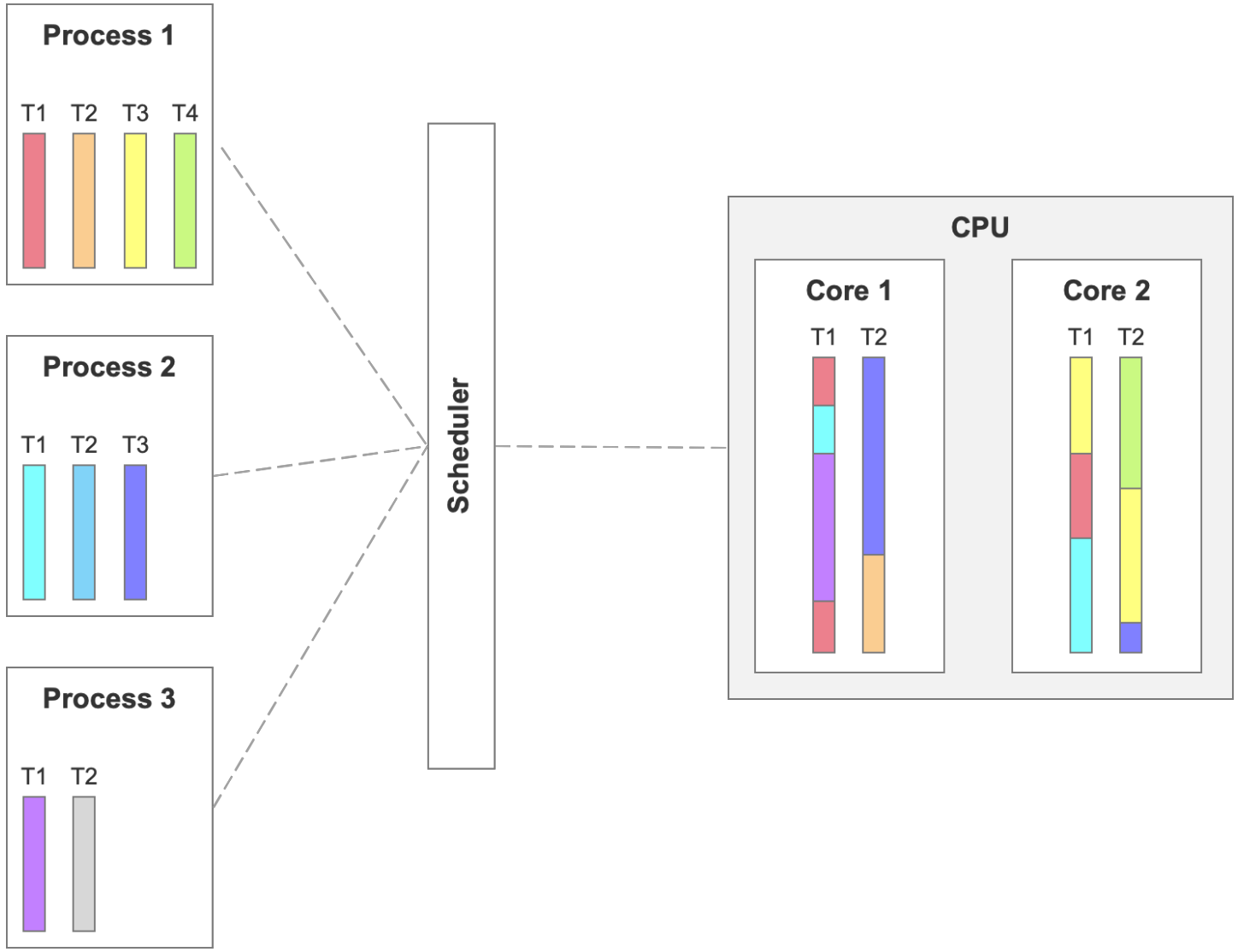
- I singoli thread sono composti da istruzioni SINCRONE

Pertanto possiamo definire con il nome di THREAD l'unità minima di esecuzione di una applicazione che viene direttamente allocata alla CPU

Schedulazione

- Normalmente i thread applicativi sono molti di più dei thread eseguibilli contemporaneamente.
- E' il sistema operativo che si occupa di allocare la CPU, decidendo quale thread eseguire, in quale core. Decide anche quando interrompere l'esecuzione di un thread e quando riprenderla.
- **Non è possibile determinare pienamente l'ordine di schedulazione**
- Nella schedulazione vengono tenuti in considerazione gli eventuali vincoli imposti dal codice.
- Dal punto di vista dell'applicazione lo schedulatore è trasparente. Tutto va come se il numero di core disponibili sia sempre sufficiente.

Schedulazione



Creazione di un Thread in C#

- I Thread in C# sono gestiti dall'omonima classe
- Vengono creati fornendo una porzione di codice da eseguire
- Dopo la creazione devono essere avviati

```
//Creazione di un Thread senza parametri

Thread t1 = new Thread(NomeMetodo);

t1.Start()

public void NomeMetodo() { ... }
```

Creazione di un Thread in C#

```
//Creazione di un Thread con passaggio di parametri

Thread t1 = new Thread(NomeMetodo);

t1.Start(parametro)

public void NomeMetodo(object p) { ... }


//Tramite lambda

Thread t2 = new Thread((p) => { NomeMetodo((int)p); });

t2.Start(parametro);

public void NomeMetodo(int p) { ... }
```

Thread: Memoria condivisa

- Tutti i thread possono accedere alla memoria dell'applicazione

```
private static int valore = 0;

static void Main(string[] args)

{

      Thread t1 = new Thread(NomeMetodo);

      t1.Start();

}

public static void NomeMetodo()

{

      Console.WriteLine(valore);

}
```

Sincronizzazione dei Thread

- Esistono due modi per sincronizzare l'esecuzione di thread:
  - Tramite la memoria condivisa, utilizzando variabili, lock e semafori
  - Tramite il metodo Join
- Il lock garantisce che solo un thread alla volta possa eseguire il codice all'interno del blocco lock

```
lock(o)
{
      somma +=i;
}
```

Sincronizzazione dei Thread

- I semafori sono analoghi al lock ma permettono un controllo più fino, dando la possibilità di definire il numero di accessi simultanei che possono avvenire su una risorsa. Le classi che implementano i semafori sono Semaphore e SemaphoreSlim
- Il metodo Join fa sì che il chiamate attenda il completamento del Thread. E' una attesa attiva, ovvero il Thread rimane in esecuzione, senza fare nulla ma occupando la CPU.

```
Thread t1 = new Thread(NomeMetodo);
t1.Start();
t1.Join();
```

Thread: Valore di ritorno

- Non c'è un modo specifico per restituire un valore al termine dell'esecuzione di un Thread.
- L'unica possibilità è quella di utilizzare una variabile in memoria condivisa

Lo scenario finale potrebbe quindi essere:

1. Il thread principale avvia un thread secondario
2. Il thread principale attende il termine dell'esecuzione (Join)
3. Il thread secondario scrive il risultato dell'elaborazione in una variabile
4. Il thread principale legge il risultato dalla variabile

Task in C#

- .Net e C# mettono a disposizione la classe Task di più alto livello rispetto alla classe Thread.
- Si basa sugli stessi concetti della classe Thread ma li integra per un utilizzo più agevole:
  - Migliore gestione del passaggio dei parametri
  - Migliore gestione del valore di ritorno
  - Possibilità di sincronizzazione maggiori
  - Possibilità di implementare facilmente il paradigma asincrono
- Utilizza una gestione ottimizzata dei Thread sottostanti. Il framework crea un certo numero di Thread (ThreadPool) e schedula l'esecuzione dei task su di essi, con un miglioramento in termini di prestazioni. Non c'è quindi una corrispondenza 1:1 tra il numero di task e di thread

Task in C#

L'esecuzione di una porzione di codice in un Task separato avviene utilizzando il metodo Run della classe statica Task.

```
Task<T> t = Task.Run( () => metodoDaEseguire() );
```

Il generics `Task<T>` permette di specificare il tipo del valore di ritorno restituito dal task

```
Task<int> t = Task.Run( () => metodoCheRitornaUnInt() );
```

```
public int metodoCheRitornaUnInt() { }
```

Task in C#

Il passaggio dei parametri può avvenire normalmente senza necessità di passare da oggetti di classe object e relativo cast. E' possibile avere più parametri

```
Task<int> t = Task.Run( () => metodoDaEseguire(10,"p") );

public int metodoDaEseguire(int p1, string p2) { }
```

I task, analogamente ai thread, condividono la memoria e possono utilizzarla per passarsi valori. Tuttavia la migliore gestione dei parametri ne sconsiglia l'utilizzo

Sincronizzazione di Task in C#

L'attesa del completamento avviene con il metodo Wait()

```
t.Wait()
```

L'attesa del completamento di un set di Task avviene con il metodo WaitAll(), invocato su un array di Task

```
List<Task<int>> tasks = new List<Task<int>>();

Task.WaitAll(tasks.ToArray());
```

E' anche possibile attendere il completamento di un solo task della collezione:

```
Task.WaitAny(tasks.ToArray());
```

I metodi `Wait*` effettuano **attesa attiva**, ovvero bloccano il chiamante occupando risorse.

Valori di ritorno

Al valore restituito dal Task si accede semplicemente tramite la proprietà Result.

```
Task<T> t = Task.Run( () => metodoDaEseguire() );

t.Wait();

T risultato = t.Result
```

E' possibile omettere la chiamata a Wait ottenendo lo stesso risultato. La lettura della proprietà Result infatti implica l'attesa della terminazione del Task

```
Task<T> t = Task.Run( () => metodoDaEseguire() );

T risultato = t.Result
```

Modello asincrono

Il modello di programmazione asincrona punta ad evitare l'attesa attiva, permettendo comunque la sincronizzazione tra task.

C# mette a disposizione il metodo ContinueWith con cui concatenare l'esecuzione di task:

```
Task<T> t = Task.Run( () => metodoDaEseguire() );

t.ContinueWith(prevTask => secondoMetodoDaEseguire(prevTask.Result)
);
```

Attesa Attiva

L'attesa attiva comporta problemi in due specifiche situazioni:

- Applicazioni con Interfaccia utente (p.e. WPF): Un solo thread è dedicato alla gestione dalla UI (UI-thread). In caso di attesa attiva l'intera interfaccia utente risulterà bloccata
- Applicazioni che fanno uso di thread pool (p.e. API): In caso di attesa attiva il task occupa il thread per un tempo maggiore, senza eseguire elaborazioni. Il tasso di saturazione del thread pool risulterà pertanto maggiore

Modello asincrono

Il caso d'uso, molto frequente, in cui si avvia una attività di cui se ne vuole recuperare il risultato può essere coperto in tre modi:

- Codice Sincrono

```
T risultato = metodoDaEseguire(); secondoMetodoDaEseguire(risultato);
```

- Codice Parallelo (attesa attiva)

```
Task<T> t = Task.Run( () => metodoDaEseguire() );

T risultato = t.Result; secondoMetodoDaEseguire(risultato);
```

- Codice Asincrono

```
Task<T> t = Task.Run( () => metodoDaEseguire() ).ContinueWith(prevTask
=> secondoMetodoDaEseguire(prevTask.Result) );
```

Modello asincrono

I tre approcci hanno caratteristiche e prestazioni differenti:

- L'approccio Sincrono è il più lento. Nulla può essere eseguito mentre il metodo oneroso è in esecuzione
- L'approccio Parallelo permette di eseguire più operazioni in parallelo ma si blocca e genera attesa attiva quando cerca di accedere alla proprietà Result
- L'approccio Asincrono è il più performante in quanto non genera mai attesa attiva. E' però il più complicato e scomodo da scrivere (e da leggere)

NB: Il fatto che del codice sia eseguibile in parallelo non implica necessariamente che lo sia. Dipende dallo scheduler.

Modello asincrono – await e async

A partire dalla versione 5.0, C# ha reso disponibili sintassi semplificati per lo scenario asincrono. Il codice risultante diventa molto simile al codice parallelo e al codice sincrono.

La versione parallela:

```
Task<T> t = Task.Run( () => metodoDaEseguire() );

T risultato = t.Result
```

Può essere riscritta in forma asincrona con la sola aggiunta della parola chiave await (e la scomparsa della proprietà Result):

```
Task<T> t = Task.Run( () => metodoDaEseguire() );

T risultato = await t;
```

Modello asincrono – await e async

Il tutto può essere scritto in un'unica riga:

```
T risultato = await Task.Run( () => metodoDaEseguire() );
```

Creando un metodo che restituisce un Task è anche possibile ricondursi ad una sintassi quasi identica a quella utilizzata nel modello sincrono:

```
Task<T> metodoDaEseguireAsync()
{  return Task.Run( () => metodoDaEseguire() );  }
```

```
T risultato = await metodoDaEseguireAsync();
```

Modello asincrono – await e async

Il metodo al cui interno si utilizza await deve **necessariamente** avere il valore di ritorno Task o Task<T> se il metodo originario restituiva un oggetto di classe T.

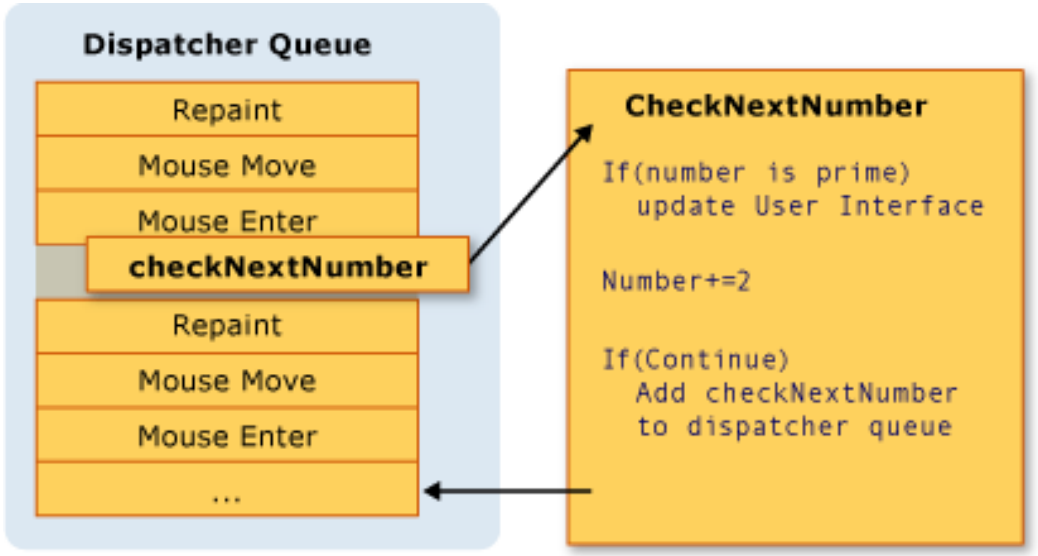
Deve inoltre prevedere la parola chiave async

```
public async Task<int> MetodoCheUsaAsync()
{
      T risultato = await metodoDaEseguireAsync();
      return risultato;
}
```

WPF e programmazione asincrona

Le applicazioni WPF utilizzano più di un thread ma solamente uno, detto UI thread, è responsabile dell'interazione con l'interfaccia utente e dell'esecuzione del codice ad esso associato.

Lo UI Thread mantiene una coda di eventi da gestire. Un oggetto chiamato Dispatcher si occupa di prelevare un evento alla volta e di eseguire il codice associato (code behind / view model)



WPF e programmazione asincrona

Le singole operazioni non possono essere interrotte. Se una di queste è particolarmente lunga l'applicazione risulta bloccata, in quanto nessun'altra attività può essere portata avanti.

E' pertanto necessario "restituire" il controllo al dispatcher al più presto:

- Limitando la quantità di codice associata ad ogni evento
- Utilizzando la programmazione asincrona.

L'utilizzo della programmazione asincrona permette di restituire il controllo al dispatcher che può pertanto eseguire altre attività di UI.

Programmazione asincrona nelle librerie .NET

Microsoft spinge fortemente sulla programmazione asincrona ed ha implementato versioni asincrone di tutti i metodi che possono avere una esecuzione "lunga". Ai metodi sincroni vengono affiancati i metodi asincroni con suffisso Async.

Troviamo metodi asincroni a fronte di:

- Operazioni di rete (p.e. download di un file)
- Operazioni su database
- Operazioni su File