

M9 - UF2

# Processos i fils en C#



Jordi Cortés Comellas  
[jcortes@lasalle.cat](mailto:jcortes@lasalle.cat)

## **Sobre el document:**

UF2 – Programació de procesos i serveis v.2.0

	Professor	Contacte	Versió
<b>Creació del document</b>	Roger Torrell Domènech	rtorrell@lasalle.cat	<b>v 1.0</b>
<b>Adaptació</b>	Jordi Cortés	jcortes@lasalle.cat	<b>v.2.0</b>

## **Índex:**

- 1. Resultats d'aprenentatge, criteris d'avaluació i continguts**
- 2. Concepte de procés i fil**
- 3. Pros i contres d'utilitzar la programació en paral·lel**
- 4. L'ús de fils en C#**
- 5. Passant informació als fils**
- 6. Bloquejos i secció crítica d'un fil**
- 7. Ús dels fils per a la millora del rendiment d'un programa**
- 8. Ús de fils per a interfícies d'usuari més responsives**
- 9. Sincronització bàsica de fils utilitzant semàfors**
- 10. Treballant amb processos**

## 1. Resultats d'aprenentatge, criteris d'avaluació i continguts:

Tot seguit s'exposen els resultats d'aprenentatge, criteris d'avaluació i continguts que defineix la Generalitat de Catalunya per aquesta UF.

### Resultat d'aprenentatge 1:

***Desenvolupa aplicacions compostes per diversos processos reconeixent i aplicant principis de programació paral·lela.***

#### Criteris d'avaluació pel resultat d'aprenentatge 1:

- Reconeix les característiques de la programació concurrent i els seus àmbits d'aplicació.
- Identifica les diferències entre programació paral·lela i programació distribuïda, els seus avantatges i inconvenients.
- Analitza les característiques dels processos i de la seva execució pel sistema operatiu.
- Caracteritza els fils d'execució i descriu la seva relació amb els processos.
- Utilitza classes per programar aplicacions que creïn subprocessos.
- Utilitza mecanismes per sincronitzar i obtenir el valor retornat per als subprocessos iniciats.
- Desenvolupa aplicacions que gestionin i utilitzin processos per a l'execució de diverses tasques en paral·lel.
- Depura i documenta les aplicacions desenvolupades.

Continguts pel resultat d'aprenentatge 1:

- Caracterització de la programació concurrent, paral·lela i distribuïda
- Identificació de les diferències entre els paradigmes de programació paral·lela i distribuïda.
- Identificació dels estats d'un procés.
- Executables. Processos. Serveis.
- Caracterització dels fils i relació amb els processos.
- Programació d'aplicacions multiprocés
- Sincronització i comunicació entre processos
- Gestió de processos i desenvolupament d'aplicacions amb finalitat de computació paral·lela
- Depuració i documentació d'aplicacions

***Resultat d'aprenentatge 2: Desenvolupa aplicacions compostes per diversos fils d'execució analitzant i aplicant llibreries específiques del llenguatge de programació.***

criteris d'avaluació per resultat d'aprenentatge 2:

- Identifica situacions en què sigui útil la utilització de diversos fils en un programa.
- Reconeix els mecanismes per a crear, iniciar i finalitzar fils.
- Programa aplicacions que implementin diversos fils.
- Identifica els possibles estats d'execució d'un fil i programa aplicacions que els gestionin.
- Utilitza mecanismes per compartir informació entre diversos fils d'un mateix procés.
- Desenvolupa programes formats per diversos fils sincronitzats mitjançant tècniques específiques.
- Estableix i controla la prioritat de cadascun dels fils d'execució.
- Depura i documenta els programes desenvolupats.

Continguts pel resultat d'aprenentatge 2:

- Elements relacionats amb la programació de fils. Llibreries i classes.
- Gestió de fils.
- Programació d'aplicacions multifil.
- Estats d'un fil. Canvis d'estat.
- Compartició d'informació entre fils i gestió de recursos compartits pels fils.

- Programes multifil, que permetin la sincronització entre ells.
- Gestió de fils per part del sistema operatiu. Planificació i accés a la seva prioritat.
- Depuració i documentació d'aplicacions

## 2. Concepte de procés i de fil:

### *Concepte de procés i programa:*

En terme informàtics, un procés, és una unitat bàsica d'execució, és a dir, un conjunt d'instruccions interpretables per un ordinador i que es poden executar de manera independent.

Cal tenir clara la diferència entre programa i procés. Un programa fa referència a un conjunt d'instruccions emmagatzemades en un fitxer o varis, però que no està en execució, sinó disponible per quan un usuari o un procés en requereixin la seva execució. Es tracta, doncs, d'un **element estàtic** en el nostre sistema.

Un procés, per contra, és un conjunt d'instruccions que s'estan executant en la CPU de l'ordinador. Un procés és un **element dinàmic** que pot variar en funció de les dades inicials que processa i dels recursos del sistema que hi hagi disponibles.

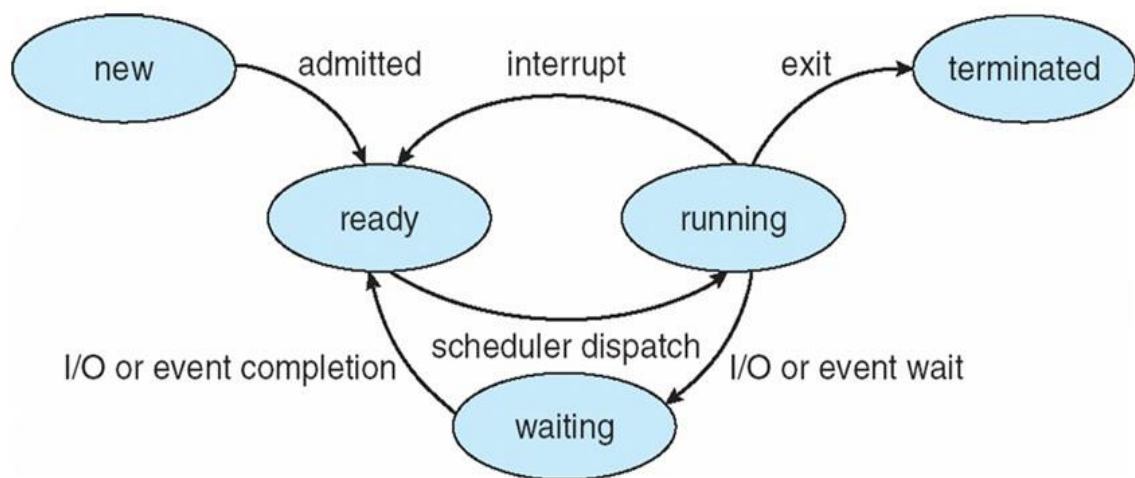
Els processos es generen a partir de programes. Quan executem un programa, el que fa el sistema operatiu, és copiar una instància d'aquest programa a la memòria RAM i crear un nou procés amb les instruccions que conté el programa. Aquest nou procés entrarà en competència amb la resta de processos del sistema per tal d'accedir a la CPU i executar-se fins que acabi.

Com és obvi en un ordinador hi ha molts processos executant-se alhora, és a dir, en **paral·lel**. Que els processos s'executin alhora no vol dir que s'executin tots al mateix temps, perquè per això necessitaríem una CPU per cada procés, cosa que és impossible. El que fa el sistema operatiu és decidir en tot moment quin procés dels que s'estan executant pot accedir a la CPU, la resta de processos es mantenen a l'espera que el sistema operatiu els hi doni permís per executar-se. Per evitar que un procés col·lapsi la CPU i bloquegi la resta de processos, el sistema operatiu assigna un temps màxim d'execució a cada procés i així pot anar avançant en l'execució del conjunt de processos.



El sistema operatiu intenta gestionar de la manera més eficient possible l'assignació de la CPU a un procés. Així, per exemple, si un procés s'atura a l'espera d'algun recurs extern (per exemple entrada de dades externa), el sistema operatiu el traurà de la CPU per deixar passar un altre procés. Quan el procés inicial tingui la dada que necessitava podrà tornar a entrar a la CPU quan el sistema operatiu ho permeti.

En el següent esquema es mostra el diagrama d'estats en que es pot trobar un procés dins del sistema operatiu:



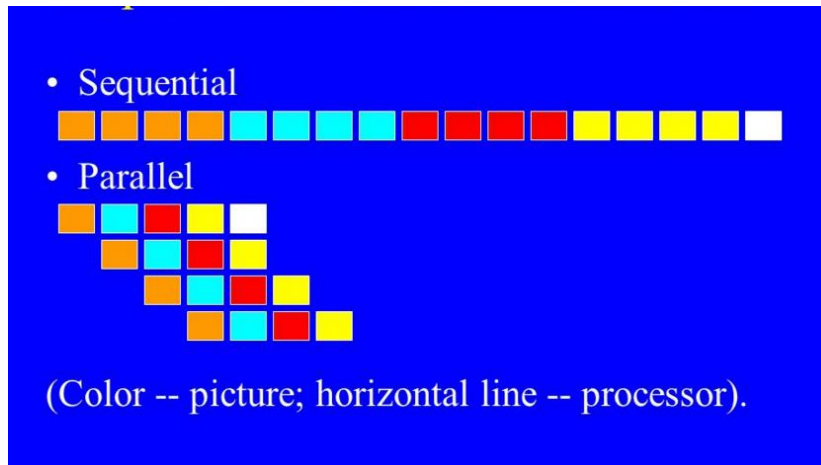
Quan es crea un nou (*new*) procés passa a l'estat de preparat (*ready*). En aquest punt s'espera que el S.O li doni permís per entrar a la CPU (*running*). De la CPU en pot sortir per dos motius, perquè acaba el seu temps de CPU (*interrupt*), aleshores passarà a l'estat de preparat o bé perquè espera un recurs extern (*I/O or event wait*) que passarà a l'estat d'espera (*waiting*). Quan el recurs extern arribi, passarà de l'estat d'espera a l'estat de preparat (*I/O or event completion*). Quan el procés acabi, passa a l'estat d'acabat (*terminated*).

### **Paral·lelisme per a millorar l'eficiència dels sistemes informàtics:**

En informàtica sovint ens trobem amb el clàssic problema en que diversos processos pretenen accedir a un recurs al mateix temps. En el cas anterior ho hem vist per accedir a la CPU, però el mateix problema el podem tenir per accedir a una base de dades o a qualsevol altre recurs.

Si intentéssim solucionar aquest problema de manera seqüencial, és a dir, gestionar les peticions d'accés a aquest recurs una darrera l'altre, ens podríem

trobar que el sistema es torna molt poc eficient. Mireu el següent gràfic que ho explica de manera gràfica:



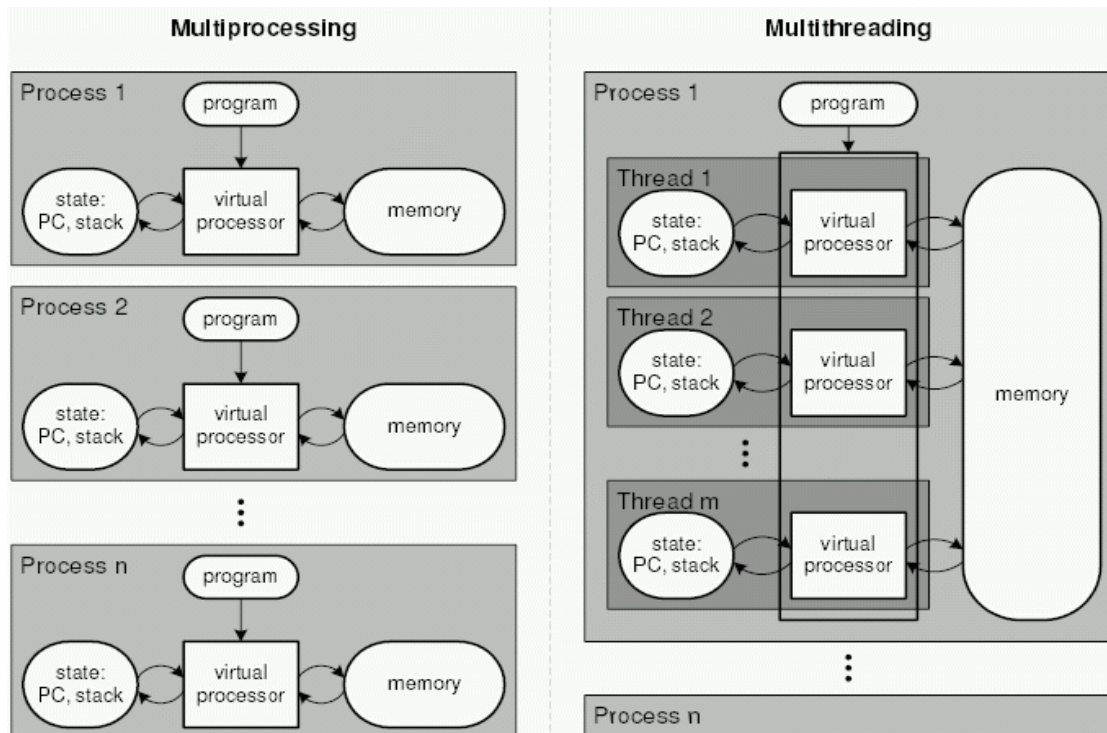
### Diferències entre processos i fils:

Hem dit que els processos són element independents entre ells que es poden executar a la CPU. En un ordinador hi poden haver diversos processos executant-se en paral·lel.

Els fils, són parts d'un mateix procés que es poden executar també en paral·lel.

La principal diferència entre un procés i un fil és que els processos són independents entre ells, cada procés té la seva pila d'execució i el seu espai de memòria. Qualsevol incidència en un procés no té perquè afectar el desenvolupament de la resta de processos.

En canvi, els fils tenen una certa dependència ja que depenen del mateix procés, així cada fil té la seva pila d'execució, però comparteixen espai de memòria. Una incidència en un fil d'una aplicació pot afectar a la resta de fils.



### 3. Pros i contres d'utilitzar la programació en paral·lel

#### Avantatges d'utilitzar la programació en paral·lel:

- **Increment la potència de càlcul:** Avui en dia molts ordinadors tenen diversos CPU. Si som capaços de programar una aplicació que permet executar diverses parts del codi en paral·lel, el sistema operatiu aprofitarà que disposem de diverses CPU i executarà cada part paral·lela en una CPU diferent, així guanyem en potència de càlcul.

Això també té a veure amb la programació distribuïda. Aquest tipus de programació dissenya aplicacions que es poden executar per parts simultàniament. Un cop s'han obtingut aquestes parts, cada part pot ser executada remotament per un ordinador diferent i enviar els resultats a l'aplicació mare, d'aquesta manera també es guanya capacitat de càlcul.

- **Millora de l'eficiència:** També cada vegada més els programes no s'executen de manera aïllada, sinó que depenen de recursos externs (connexions a base de dades, accés a fitxers de dades, etc.). Aquestes operacions d'accés a recursos externs poden suposar esperes que fan que l'aplicació funcioni de manera pot eficient. Si som capaços d'aprofitar el temps que una aplicació està esperant un recurs extern per a fer d'altres operacions, augmentem el rendiment i eficiència de l'aplicació.
- **Interfícies d'usuari (UI) més responsives:** La gràcia d'una bona interfície d'usuari és que sempre estigui activa per l'usuari. Si cada vegada que hem de realitzar una operació lenta (per exemple, la connexió amb una base de dades), la interfície d'usuari es quedés "penjada", el programa no seria massa usable. Per això s'utilitza la programació en paral·lel, per tal d'executar en segon pla les operacions lentes i deixant la UI sempre lliure de tasques "pesades".
- **Aplicacions client – servidor:** El servidor l'haurem de programar de manera que pugui atendre les peticions dels clients de manera simultània, per tant haurem d'utilitzar alguna tècnica de programació en paral·lel.

De la mateixa manera que hi ha avantatges cal dir que la programació en paral·lel també presenta dificultats o reptes:

- **Comunicació i sincronització entre fils.** Sovint ens podem trobar que diverses tasques que volem fer en paral·lel depèn d'altres tasques. Com podem saber en quin punt acaba una tasca A per començar-ne una B? Com es passen dades d'un fil a un altre?
- **Integritat de les dades.** El fet de treballar en paral·lel vol dir que diversos fils poden accedir a una mateixa dada, sense tenir, d'entrada el control del moment en que s'accedeix a aquesta dada. Això pot portar a problemes d'integritat de la informació, és a dir, a trobar-nos que aquella dada que pensàvem que tenia una valor A, té un valor B perquè un altre fil l'ha canviat.
- **Pèrdua de control del flux del programa.** Quan realitzem una aplicació amb fils que s'executen en paral·lel és impossible saber quines instruccions s'executaran abans i després. El control del flux del programa ja no el tenim els programadors sinó que el té el sistema operatiu, que decideix en quin moment dóna accés a la CPU a un fil o a un altre. Això dificulta la programació i, sobretot, la depuració del codi, a part que pot ser font d'errors imprevistos ja que cada nova execució pot realitzar les instruccions en un ordre diferent.

En definitiva, en el moment de plantejar-nos de realitzar un programa utilitzant la programació en paral·lel hem de valorar si les avantatges que aconseguirem justifiquen l'ús d'aquestes tècniques ja que també hem de ser conscients que el programa guanyarà en complexitat i en dificultat de manteniment.

## 4. L'ús de fils en C#

Per tal d'utilitzar les tècniques de programació concurrent o paral·lela .Net i C# ofereixen una gran quantitat d'objectes, probablement els més utilitzats són:

- **Thread.** Que implementa un fil de la manera més “pura”, creant tot el necessari per executar una part del codi de manera paral·lela.
- **Task.** Que implementa els fils, però creant una estructura una mica més àgil que la classe *Thread*.

En general, pel propòsit d'aquesta UF podem utilitzar-les ambdues sense cap problema. De moment utilitzarem la classe *Thread*.

### Com es creen els fils?

Recordem que conceptualment un fil és un tros de codi que pot executar-se en paral·lel a la resta del codi de l'aplicació, per tant, el primer que necessitem per crear un fil és aquest “tros de codi” que es vol executar en paral·lel. Aquest tros de codi és un mètode que definim amb aquest propòsit i que passarem a la classe *Thread*.

Observeu el següent exemple:

```
class ProvaFils
{
    //El codi que volem executar en paral·lel el posem en un mètode que li
    //passarem al Thread.
    static void Go()
    {
        for (int i = 0; i < 5; i++)
            Console.WriteLine(i);
    }

    static void Main()
    {
        //Creem un nou fil (T), amb el mètode Go com a ThreadStart
        Thread T = new Thread(Go);
        T.Start();

        Console.ReadLine();
    }
}
```

Fixeu-vos que l'inici el fil requereix de dues accions. Per una banda indiquem quin mètode executarem en paral·lel: `Thread T = new Thread(Go);` i després iniciem el *Thread*: `T.Start();`

En el moment que fem *Start*, ja tenim el codi del mètode *Go* executant-se en paral·lel. Si executeu aquest codi no veureu res que faci pensar que s'ha

executat en paral·lel, anem a afegir algunes coses que ens permetran veure millor l'efecte de treballar en paral·lel.

Un mètode de la classe *Thread* que ens serà molt útil per treballar amb fils és el mètode *Sleep*. Aquest mètode permet aturar l'aplicació durant el nombre de milisegons que li indiquem. Afegim un *Sleep* en l'exercici anterior per tal de fer que el programa vagi més lent.

```
class ProvaFils
{
    //El codi que volem executar en paral·lel el posem en un mètode que li
    //passarem al Thread.
    static void Go()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.Write(i);
            Thread.Sleep(500);
        }
    }

    static void Main()
    {
        //Creem un nou fil (T), amb el mètode Go com a ThreadStart
        Thread T = new Thread(Go);
        T.Start();

        Console.ReadLine();
    }
}
```

Ara veurem el mateix que abans, però la impressió del números es farà més lentament. Seguim, però sense veure els efectes de treballar en paral·lel. Per veure-ho anem a crear un segon fil i així si que podrem veure com s'executen en paral·lel.

Farem un fil que s'encarregui d'imprimir lletres "a" i un altre fil que s'encarregui d'imprimir lletres "b". Ho executarem de manera seqüencial per veure'n el resultat i també de manera paral·lela.

```
class ProvaFils
{
    static void escriure_a()
    {
        for (int i = 0; i < 500; i++)
        {
            Console.Write("a");
            Thread.Sleep(10);
        }
    }

    static void escriure_b()
    {
        for (int i = 0; i < 500; i++)
        {
            Console.Write("b");
            Thread.Sleep(10);
        }
    }

    static void Main()
    {
        //Primer ho provem en seqüencial
        escriure_a();
        escriure_b();

        //Després ho provem en paral·lel
        Thread T_a = new Thread(escriure_a);
        Thread T_b = new Thread(escriure_b);
        T_a.Start();
        T_b.Start();

        Console.ReadLine();
    }
}
```

Executant aquest codi es pot veure com l'execució seqüencial escriu primer les "a" i després les "b", mentre que l'execució en paral·lel s'escriuen les "a" i les "b" de manera simultània.

Si us hi fixeu també podreu veure que l'execució seqüencial és considerablement més lenta que la paral·lela. Això és així perquè en l'execució seqüencial, cada vegada que es fa el `Thread.Sleep(10)`; aquest es fa sencer, és a dir, el programa es queda 10 milisegons sense fer res. En canvi en l'execució en paral·lel quan un dels fils fa el `Thread.Sleep(10)`;, el sistema operatiu detecta que el fil està a l'espera i aprofita per executar l'altre fil. D'aquesta manera no es perden els 10 milisegons sencers sinó que s'aprofita la pausa per executar d'altres fils.



### **Atenció, el Main també és un fil:**

Si us hi fixeu al final de tot del programa posem un `Console.ReadLine();`.

Això ho fem perquè el programa principal no s'acabi quan arribi al final. Perquè és important això? Doncs bé molt senzill, el programa principal és l'encarregat d'iniciar els dos fils. Com que aquests s'executen en paral·lel el programa principal segueix la seva execució, com un fil més. És a dir, després de fer els dos `Start()` en total tenim tres fils en funcionament: els dos que hem creat i el fil principal (*Main*).

El fil principal, després de fer els `Start()` ja no té res més a fer, per tant s'acabaria. Com que és el fil que ha creat els altres dos, per seguretat això provocaria que els dos fils també acabessin i el programa pràcticament no faria res. Per això posem el `Console.ReadLine();` per evitar que el fil principal s'acabi just després de fer els `Start`.

### **Sincronització bàsica entre fils. Mètode *Join*:**

Com ja s'ha dit anteriorment un dels temes més complexos de treballar amb fils és la sincronització entre ells que ens permeti tenir un mínim control de com s'està executant l'aplicació.

Hi ha moltes classes que permeten treballar la sincronització de fils, per exemple els semàfors. Nosaltres, de moment, veurem la més bàsica, el mètode ***Join()***.

El mètode *Join* permet al fil que n'ha creat un altre, esperar-se fins que el fil fill ha finalitzat la seva execució.

Vegem-ho amb un exemple, imaginem que en l'exemple anterior el programa principal volgués escriure un missatge per pantalla indicant que `escriure_a` i `escriure_b` han acabat, ho podríem provar de la següent manera:

```
class ProvaFils
{
    static void escriure_a()
    {
        for (int i = 0; i < 500; i++)
        {
            Console.Write("a");
            Thread.Sleep(10);
        }
    }

    static void escriure_b()
    {
        for (int i = 0; i < 500; i++)
        {
            Console.Write("b");
            Thread.Sleep(10);
        }
    }

    static void Main()
    {
        Thread T_a = new Thread(escriure_a);
        Thread T_b = new Thread(escriure_b);
        T_a.Start();
        T_b.Start();

        //Això hauria de sortir al final del programa però fet així, surt al
principi!!
        Console.WriteLine("Escriure a i escriure b ja han acabat");

        Console.ReadLine();
    }
}
```

Si ho provem així, el que passa és que el missatge, en lloc de sortir al final del programa quan realment han acabat escriure\_a i escriure\_b, surt al principi!!. El motiu ja l'hem explicat abans, el fil principal (*Main*), segueix la seva execució un cop s'han iniciat els fils i, per tant, el primer que fa és mostrar el missatge.

El que hem de fer és indicar-li en el *Main* que s'espera a que els dos fils hagin acabat i això ho farem amb el *Join*, de la següent manera:

```
class ProvaFils
{
    static void escriure_a()
    {
        for (int i = 0; i < 500; i++)
        {
            Console.Write("a");
            Thread.Sleep(10);
        }
    }

    static void escriure_b()
    {
```

```
        for (int i = 0; i < 500; i++)
        {
            Console.Write("b");
            Thread.Sleep(10);
        }
    }

    static void Main()
    {
        Thread T_a = new Thread(escriure_a);
        Thread T_b = new Thread(escriure_b);
        T_a.Start();
        T_b.Start();
        T_a.Join(); //espera que acabi T_a
        T_b.Join(); //espera que acabi T_b

        Console.WriteLine("Escriure a i escriure b ja han acabat");
        Console.ReadLine();
    }
}
```

Gràficament, el que fa el *Join* ho podríem representar així:



## 5. Passant informació als fils:

Qualsevol programador que es miri el codi que hem fet servir en l'exemple anterior pot veure que estem escrivint massa codi:

```
static void escriure_a()
{
    for (int i = 0; i < 500; i++)
    {
        Console.Write("a");
        Thread.Sleep(10);
    }
}

static void escriure_b()
{
    for (int i = 0; i < 500; i++)
    {
        Console.Write("b");
        Thread.Sleep(10);
    }
}
```

Aquests dos mètodes fan exactament el mateix amb la única diferència que un escriu "a" i l'altre escriu "b". No podríem fer això amb un sol mètode que rebí com a paràmetre la lletra a escriure?

```
static void escriure_lletra(string lletra)
{
    for (int i = 0; i < 500; i++)
    {
        Console.Write(lletra);
        Thread.Sleep(10);
    }
}
```

Si intentem aplicar aquest nou mètode `escriure_lletra` a l'exemple anterior ens trobarem amb un problema, com ho fem per passar paràmetres a un fil?

Tenim dues maneres de fer-ho:

- Passant la informació directament en el moment de fer `I'Start()`
- Utilitzant les *lambda expressions*.

Ambdues tenen les seves avantatges i inconvenients. Anem a veure les dues tècniques.

### Pas de paràmetres directament:

Per tal d'explicar el pas de paràmetres directament, utilitzarem el mètode anterior `Escriure_lletra`

```
static void escriure_lletra(string lletra)
{
    for (int i = 0; i < 500; i++)
    {
        Console.WriteLine(letra);
        Thread.Sleep(10);
    }
}
```

Aquest serà el mètode que passarem a la classe *Thread*, per tal que s'executi en un fil. Com es pot observar el mètode té dos paràmetres per tant, també li haurem de passar aquesta informació.

La classe *Thread* permet passar un paràmetre en el moment en que es fa *Start* de la següent manera

```
static void Main(string[] args)
{
    Thread T = new Thread(escriure_lletra);
    T.Start("a");
}

static void escriure_lletra(object lletra)
{
    string lletra_parsejada = (string)lletra;

    for (int i = 0; i < 500; i++)
    {
        Console.WriteLine(letra_parsejada);
        Thread.Sleep(10);
    }
}
```

El paràmetre, en aquest cas la lletra "a", el passem en el moment de fer *Start*. Fixeu-vos que si passem el paràmetre en el moment de fer *Start*, li podríem passar qualsevol informació, en aquest cas és un *String*, però si el mètode que executa el fil fos un altre, potser hauríem de passar un enter o qualsevol altre dada. Per aquest motiu, definim el tipus del paràmetre com a *object*, per tal d'indicar que pot ser qualsevol tipus.

```
static void escriure_lletra(object lletra)
```

Un cop rebut el paràmetre ja el podem parsejar per tal de transformar-lo al tipus que realment necessitem. `string lletra_parsejada = (string)lletra;`

Una limitació que té el pas de paràmetres directament és que només podem passar un sol paràmetre de tipus *object*. La pregunta, aleshores, és com

podríem utilitzar un mètode per executar en un fil si aquest té més d'un paràmetre? Vegem-ho amb un exemple:

Seguirem amb el mètode `Escriure_lletra` d'abans, però li afegirem un paràmetre més, que serà el nombre de vegades a escriure una determinada lletra. Així el codi resultant del mètode `Escriure_lletra` serà el següent:

```
static void escriure_lletra(string lletra, int num)
{
    for (int i = 0; i < num; i++)
    {
        Console.WriteLine(letra);
        Thread.Sleep(10);
    }
}
```

Si utilitzem la mateixa tècnica que abans, però de la següent manera, ens trobarem amb un error de compilació en el moment de creació del fil:

```
static void Main(string[] args)
{
    Thread T = new Thread(escriure_lletra);
    T.Start("a", 500);
}

static void escriure_lletra(string lletra, int num)
{
    for (int i = 0; i < num; i++)
    {
        Console.WriteLine(letra);
        Thread.Sleep(10);
    }
}
```

Com hem dit abans, el pas de paràmetres directament al fil només es pot fer a partir d'un sol paràmetre de tipus *object*. Per tal de passar-li només un paràmetre al fil, si en necessitem més d'un el que farem serà crear una estructura de dades (una classe, un *array*, etc.) que contindrà tots els paràmetres, és a dir, una mena de “capsa”, que contindrà tota la informació que necessita el fil. Ara ja tenim una sol paràmetre a passar, aquesta “capsa”.

Quan la passem al fil i aquest la rebí, el que farà el fil és obrir la capsa per obtenir els diferents paràmetres que requereix per funcionar. Vegem-ho amb un exemple.

```
class Parametres
{
    public int num { get; set; }
    public string lletra { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        //Creem la capsa de paràmetres
        Parametres P = new Parametres();
        P.lletra = "a";
        P.num = 500;

        Thread T = new Thread(escriure_lletra);
        T.Start(P); //Al fil li passem la capsa sencera
    }

    //Com sempre si passem informació al fil ho fem a través d'un paràmetre
    //de tipus object
    static void escriure_lletra(object P)
    {
        //Obrim la capsa on hi ha la informació que necessitem
        Parametres myP = (Parametres)P;

        for (int i = 0; i < myP.num; i++)
        {
            Console.Write(myP.lletra);
            Thread.Sleep(10);
        }
    }
}
```

### **Pas de paràmetres utilitzant *lambda expressions*:**

Si ens volem estalviar tot aquest codi extra per a fer una cosa tant senzilla com passar paràmetres a un mètode, podem utilitzar un altre mètode per a passar informació al un fil: utilitzant les ***lambda expressions***. Atenció, però, quan utilitzem aquest sistema ja que, com veurem més endavant, no és segur al 100% i si no sabem exactament com estem treballant podem tenir problemes.

Les *lambda expression* permeten cridar un mètode de manera anònima. S'utilitzen en moltes situacions i també les podrem utilitzar per a cridar un mètode delegat d'un *Thread*. En el cas anterior ho faríem de la següent manera:

```
static void Main(string[] args)
{
    Thread T = new Thread(()=>escriure_lletra("a",500)); //Això és una
    expressió lamda
```

```
        T.Start(P);
    }

    static void escriure_lletra(string lletra, int num)
    {
        for (int i = 0; i < num; i++)
        {
            Console.Write(lletra);
            Thread.Sleep(10);
        }
    }
}
```

Com podem veure, la crida al mètode delegat del *Thread* és molt més fàcil amb les *lambda expressions*, només cal posar la capçalera de l'expressió *lambda*, és a dir: `()=>` i tot seguit la crida al mètode delegat tal i com ho faríem normalment: `()=>escriure_lletra("a",500)`

### Limitacions en la utilització de les lambda expressions

L'ús de les lambda expressions és més senzill que el pas de paràmetres directament a *Thread*, però té un problema, imaginem el següent programa que crea 5 fils mitjançant un bucle. Veiem els dos exemples, un utilitzant el pas de paràmetres directament i l'altre utilitzant les lambda expressions:

```
static void Main(string[] args)
{
    //Creació de fils utilitzant les lambda expressions
    for (int i = 0; i < 5; i++)
    {
        Thread T = new Thread(() => Escriu_Lambda(i));
        T.Start();
    }

    //Creació de fils utilitzant el pas de paràmetres directe
    for (int j = 0; j < 5; j++)
    {
        Thread T = new Thread(Escriu_Param);
        T.Start(j);
    }
}

static void Escriu_Lambda(int valor)
{
    Console.WriteLine("{0}", valor.ToString());
}

static void Escriu_Param(object valor)
{
    Console.WriteLine("{0}", valor.ToString());
}
```

El programa anterior, teòricament fa el mateix, tant en el cas de la creació dels fils amb *lambda expressions* com en el cas de la creació de fils amb pas de paràmetres directament. Ara bé, quan ho executem, veiem que el programa



funciona correctament amb pas de paràmetres directes i no va bé amb les expressions *lambda*.

Resultats amb pas de paràmetres directes:

012345 (que és correcte)

Resultats amb *lambda expression*:

33355 (o alguna cosa similar, ja que a cada execució pot treure resultats diferents)

### **Quin és el problema?**

Les *expressions lambda* no passen realment la informació directament al mètode delegat, sinó que només li passen la referència de memòria on es troba aquesta informació. Si només creem un fil, això no suposa cap problema, però si creem varis fils de manera consecutiva les *expressions lambda* poden no funcionar correctament.

L'explicació és que un cop creat el primer fil, quan s'intenta crear el segon es passa la referència de la mateixa variable (la *i*) i, per tant, el primer fil ja no té la referència del seu paràmetre real, sinó del nou valor que té la variable *i*.

Dit en altres paraules, només podrem utilitzar les *expressions lambda* si estem completament segurs que els valors de les variables que passem com a paràmetres no varien entre el moment que creem l'*expressió lambda* i el moment en que fem l'*Start()* del *Thread*.

## 6. Bloquejos i secció crítica:

La programació en paral·lel, com ja s'ha comentat anteriorment, si no es fa bé pot suposar una font de problemes de difícil compressió i depuració. Un dels problemes més comuns que ens podem trobar és la pèrdua d'integritat de les dades. Això vol que, un programa que en teoria sempre hauria de donar el mateix resultat amb les mateixes dades d'entrada, dona resultats diferents i, de vegades, incoherents.

### Vegem-ne un exemple:

Consisteix en dos fils que intentaran realitzar una tasca concreta, només un dels dos ho pot fer, el primer que arribi farà la tasca i escriurà per pantalla "Done", l'altre ja no podrà fer la tasca.

```
class Program
{
    static bool done = false;

    static void Main(string[] args)
    {
        Thread Fil1 = new Thread(Go);
        Thread Fil2 = new Thread(Go);
        Fil1.Start();
        Fil2.Start();
        Console.ReadLine();
    }
    static void Go()
    {
        if (!done)
        {
            Thread.Sleep(10);
            //Si el programa canvia just aquí del fil 1 al fil 2 no funciona
correctament.
            done = true;
            Console.WriteLine("Done");
        }
    }
}
```

Si executem aquest programa comentant la línia on es fa el *Thread.Sleep(10)* rarament ens trobarem que va malament i el podríem donar per bo. Però hi ha un punt del programa, on si es produís un canvi d'execució del fil 1 al fil 2, el programa deixaria de funcionar.

Si després de fer el `if (!done)`

i entrar dins de l'execució de l'*if*, el programa canvia del fil 1 al fil 2, veurem com es produeix un error i és que els dos fils realitzen la tasca i escriuen *Done*. Per tal de forçar que es produeixi un canvi del fil 1 al fil 2 en aquest punt hem posat el *Thread.Sleep (10)*, que forçarà el canvi i, efectivament, veurem com l'execució del programa no va bé.

Això s'explica perquè, un cop un dels dos fils entri en l'*if* ha d'executar tota la tasca sencera, és a dir, tot el codi inclòs dins de l'*if*.

En programació en paral·lel com que perdem el control del flux d'execució de les instruccions del codi, pot ser que necessitem definir zones del codi que s'han d'executar de manera continuada, és a dir, que dins d'aquestes seccions de codi no es pot produir un canvi d'un fil a un altre. Aquestes seccions s'anomenen seccions crítiques.

La identificació d'una secció crítica pot dependre de moltes circumstàncies d'ella programació, però per norma general **crearem seccions crítiques per:**

- Accedir a un **recurs compartit**
- Per accedir a una **variable a nivell de classe**

Per tal de crear seccions crítiques utilitzem bloquejos, que s'implementen amb la instrucció ***lock***.

Veiem com quedaria l'exemple anterior creant la secció crítica pertinent:

```
class Program
{
    static bool done = false;
    static readonly object locker = new object();

    static void Main(string[] args)
    {
        Thread Fil1 = new Thread(Go);
        Thread Fil2 = new Thread(Go);
        Fil1.Start();
        Fil2.Start();
        Console.ReadLine();
    }
    static void Go()
    {
        lock (locker)
        {
            if (!done)
            {
                //Aquest codi és la secció crítica. S'executarà de manera
consecutiva, sense cap interrupció
                Thread.Sleep(10);
                //Si el programa canvia just aquí del fil 1 al fil 2 no
funciona correctament.
                done = true;
                Console.WriteLine("Done");
            }
        }
    }
}
```

Com es pot veure, per utilitzar el *lock*, necessitem crear una variable de tipus *object*, que no té cap altra funció que permetre treballar al *lock*.

```
static readonly object locker = new object();
```

Serà aquesta variable *locker*, que cridarà la classe *lock*, per tal de fer el bloqueig d'una part del codi.

**Recordeu, la norma d'or de la programació en paral·lel: protegiu sempre amb un *lock* (bloqueig) qualsevol acció de modificació d'una variable a nivell de classe que sigui, també accessible per altres fils.**

## 7. Ús dels fils per a la millora del rendiment d'un programa

Un dels propòsits més comuns en l'ús dels fils és per tal de millorar el rendiment d'un programa. L'ús dels fils ens permeten millorar el rendiment en dues situacions concretes:

- 1) **Si el programa realitza operacions que tenen un temps d'espera, per exemple connexions a una base de dades remota.**

Per exemple, imaginem que un programa ha de recollir dades de 100 sensors i la recepció de les dades triga 10 segons per cada sensor. Si ho fem en seqüencial, la recepció de les dades de tots els sensors pot trigar uns 1000 segons. En canvi, si fem la operació de recepció en paral·lel, els temps d'espera no es sumaran sinó que es faran al mateix temps. En total el temps d'espera de les dades pot ser d'uns 10 segons.

- 2) **Si disposem d'una computadora amb més d'una unitat de processament (CPU), ja que en aquest cas cada CPU pot executar una part del programa en paral·lel.**

Per exemple, si realitzem un programa que fa 1 minut d'operacions de càlcul, si podem separar aquests càlculs en 4 parts i la nostra té 4 processadors o més, podem dividir el temps total de càlcul (1 minut) en quatre parts, una per cada processador. El programa pot trigar, doncs, una quarta part.

Una de les classes que ofereix .Net per a implementar operacions en paral·lel és precisament la classe **Parallel**, que permet realitzar operacions en paral·lel sense la necessitat de crear els fils, sinó que és el propi .Net qui s'encarrega de fer l'execució en paral·lel de manera transparent a l'usuari.

Però per explicar-ho anem a veure un exemple. Simularem un programa que s'ha de connectar a 10 sensors de manera remota, mitjançant connexions lentes. Simularem aquesta connexió lenta amb el següent mètode:

```
static void connexio()
{
    Thread.Sleep(1000); //simulem un temps de retard en la connexió
    Console.WriteLine("Sensor connectat");
}
```

Per connectar-nos als sensors ho farem amb el següent codi (de moment fet de manera seqüencial):

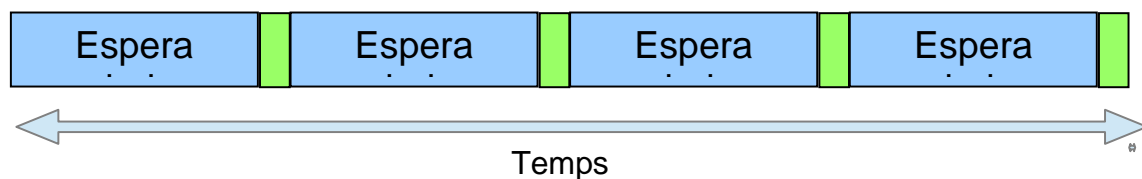
```
for (int i = 0; i < 10; i++)
{
    connexio();
}
```

Si executem aquest programa ens trobarem que triga uns 10 segons:

```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i < 10; i++)
        {
            connexio();
        }
        Console.ReadLine();
    }

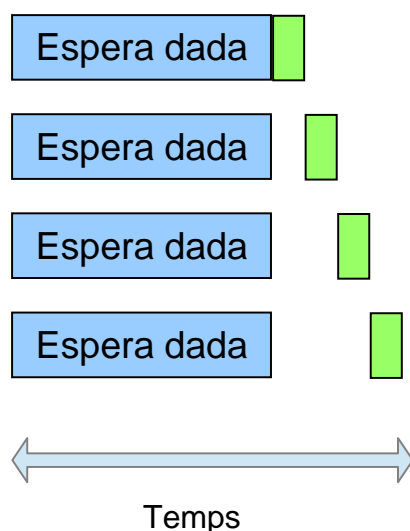
    static void connexio()
    {
        Thread.Sleep(1000); //simulem un temps de retard en la connexió
        Console.WriteLine("Sensor connectat");
    }
}
```

Gràficament podem representar l'execució d'aquest programa de la següent manera:



En verd representem el temps real d'execució d'alguna instrucció i en blau el temps d'espera, com es pot veure el programa passa la major part del temps parat esperant dades

En canvi si realitzem el mateix programa en paral·lel en diferents fils d'execució veurem que el programa trigarà poc més d'un segon. Gràficament es pot representar de la següent manera:



Per tal de programar els fils en paral·lel podríem crear un fil per cada connexió, en total 10 fils, i que cada fil s'encarregués d'una connexió diferent. De fet, és això mateix el que farem, però enlloc d'utilitzar la classe *Thread* per a la creació dels fils ho farem amb *Parallel*.

La classe *Parallel* ens permet passar operacions repetitives seqüencials (bucles) a operacions repetitives en paral·lel, que seran molt més òptimes des del punt de vista del temps. Veiem l'exemple anterior programa amb la classe *Parallel*:

```
class Program
{
    static void Main(string[] args)
    {
        //El bucle el fem en paral·lel, totes les operacions de connexió al
mateix temps
        Parallel.For(0, 10, delegate(int i)
        {
            connexio();
        });
        Console.ReadLine();
    }

    static void connexio()
    {
        Thread.Sleep(1000); //simulem un temps de retard en la connexió
        Console.WriteLine("Sensor connectat");
    }
}
```

El que fa la classe *Parallel.For* és definir, en primer lloc, el primer i últim valor sobre els que cal iterar, en el nostre exemple del 0 (inclusiu) al 10 (exclusiu). I tot seguit especifica un delegat, que és el tros de codi que s'ha d'executar en paral·lel.

Si necessitéssim disposar d'un comptador de les iteracions es pot fer servir el paràmetre i que passem al delegat, vegem-ho en un exemple:

```
class Program
{
    static void Main(string[] args)
    {
        //El bucle el fem en paral·lel, totes les operacions de connexió al
mateix temps
        Parallel.For(0, 10, delegate(int i)
        {
            connexio(i);
        });
        Console.ReadLine();
    }

    static void connexio(int i)
    {
        Thread.Sleep(1000); //simulem un temps de retard en la connexió
        Console.WriteLine("Sensor {0} connectat",i);
    }
}
```

Si executem aquest programa veiem que l'ordre d'execució dels fils no és seqüencial i ja és això el que volem, ja que s'executen en paral·lel i això vol dir que l'ordre del bucle ja no és important.

En general podrem utilitzar un *Parallel.For* enlloc d'un *For* i, per tant, millorar el rendiment del programa sempre que:

**Les operacions que faci el bucle *For* no siguin dependents les unes de les altres.**

**Si una operació del bucle depèn de l'operació anterior o si l'ordre de les operacions és important, no es podria utilitzar el *Parallel.For*.**

**Cal tenir en compte, també, que si accedim a alguna variable a nivell de classe (un atribut de la classe), o bé una variable fora del *Parallel.For*, caldrà fer-ho amb un bloqueig, tal i com s'ha explicat en capítols anteriors.**



En el cas que la iteració no la realitzem sobre un valor enter, sinó que sigui una col·lecció de valors qualsevol, també podem utilitzar l'opció d'un ***Parallel.Foreach*** de la següent manera:

```
namespace Exemple_ParallelForEach
{
    class CustomClass
    {
        public string AtributA { get; set; }
        public int AtributB { get; set; }

        public CustomClass(string atributA, int atributB)
        {
            AtributA = atributA;
            AtributB = atributB;
        }
    }
    class Program
    {
        static List<CustomClass> colleccioCustomClass = new List<CustomClass>();

        static void Main(string[] args)
        {
            Inicialitza();

            Parallel.ForEach(colleccioCustomClass, delegate (CustomClass C)
            {
                Console.WriteLine(C.AtributA);
                Console.WriteLine(C.AtributB);
            });
            Console.ReadLine();
        }

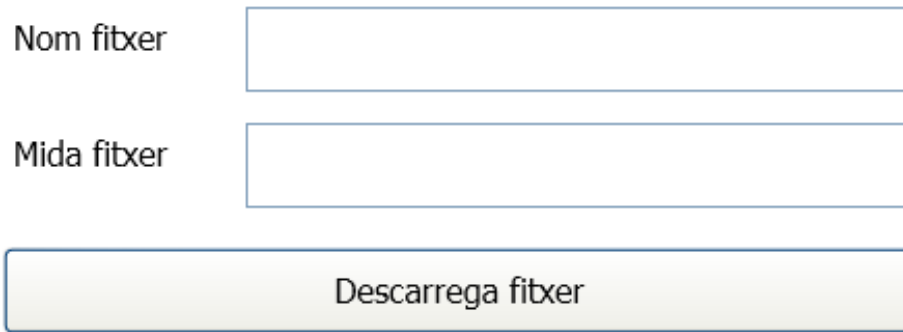
        static void Inicialitza()
        {
            for (int i=0; i<10; i++)
            {
                CustomClass myCustomClass = new CustomClass(i.ToString(), i);
                colleccioCustomClass.Add(myCustomClass);
            }
        }
    }
}
```

## 8. Ús dels fils per a fer interfícies d'usuari més responsives:

Un altre problema típic que podem afrontar amb l'ús de fils és referent a les interfícies d'usuari.

Plantegem el problema amb un exemple. Ara ho farem a partir d'un exemple de programa amb interfícies gràfica que hem desenvolupat amb *WPF*.

Imaginem que tenim una aplicació que en un moment donat permet a l'usuari descarregar un fitxer. Per programar aquesta prova hem desenvolupat la interfície d'usuari següent:

The user interface consists of two text input fields and one button. The first field is labeled "Nom fitxer" and the second is labeled "Mida fitxer". Below these fields is a button labeled "Descarrega fitxer".

Nom fitxer	<input type="text"/>
Mida fitxer	<input type="text"/>
<input type="button" value="Descarrega fitxer"/>	

En el moment de fer click en el botó de Descarrega, el programa iniciarà la descàrrega del fitxer indicat en el *TextBox* "Nom fitxer". Per fer la prova, no farem una descàrrega real, sinó que simularem la descàrrega amb el següent codi:

```
private void ButtonDescarrega_Click(object sender, RoutedEventArgs e)
{
    int mida = Convert.ToInt32(TextBoxMida.Text);
    string nom = TextBoxNom.Text;
    descarrega_fitxer (mida, nom);
}

private void descarrega_fitxer(int mida, string nom)
{
    for (int i = 1; i <= mida; i++)
    {
        descarrega_1MB();
    }
}

private void descarrega_1MB()
{
    Thread.Sleep(1000);
}
```

El mètode descarrega\_1MB només atura el programa un segon, simulant una descàrrega lenta, així, si indiquem que el fitxer té 20 MB la programa trigarà 20MB a finalitzar.

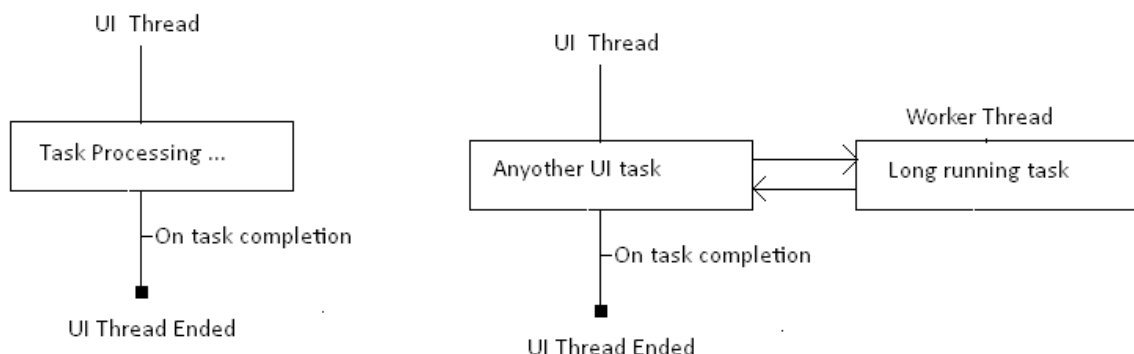
Si executem el programa d'aquesta manera el resultat serà que, durant el temps que s'estigui executant la tasca lenta, en aquest cas el descarrega\_fitxer, la interfície d'usuari es queda bloquejada, no es pot accedir a cap dels seus controls.

Òbviament això és una limitació molt important per una aplicació ja que desitjarem sempre tenir interfícies d'usuari plenament responsives, en que l'usuari sempre pugui interactuar.

### L'ús dels fils per millora una interfície d'usuari:

Per tal de millora una interfícies d'usuari utilitzant fils, el que farem serà que, cada vegada que tinguem una tasca lenta a executar, no ho farem en el fil principal que és on s'executa la UI, sinó que ho farem en un fil en segon pla, que només s'encarregarà de processar la tasca lenta en qüestió.

Gràficament es podria interpretar de la següent manera:



En aquest esquema la UI queda bloquejada

En aquest esquema la UI no queda bloquejada ja que la tasca llarga es fa en segon pla

### *El problema de treballar amb UI i fils*

Normalment, si llencem una tasca lenta en segon pla voldrem saber com avança aquesta tasca, i és molt normal que la tasca en segon pla hagi d'actualitzar algun dels controls de la interfície, per exemple una barra de progrés.

En la nostra aplicació, doncs, tenim un fil principal que és on s'executa la UI i un fil en segon pla que és on s'executa la tasca lenta. Quan aquest segon fil

secundari intenti accedir a un control de la UI ens trobarem que salta un error similar a això:

***Cross-thread operation not valid: Control 'textBox1' accessed from a thread other than the thread it was created on***

La limitació que tenim quan treballem amb *Threads* i UI és que, en principi, només pot accedir directament als controls de la UI el fil on s'executa la UI, per tant qualsevol altra fil que intenti accedir directament a la UI tindrà aquest error.

Per tal de solucionar aquest problema i permetre que un fil secundari pugui accedir als controls de la UI tenim dues possibilitats:

- Utilitzar el mètode ***Invoke*** des del fil secundari
- Utilitzar la classe ***BackgroundWorker***, especialment dissenyada per a executar tasques pesants en segon pla.

### ***Accés a la UI mitjançant Invoke***

El mètode `Dispatcher.Invoke` és la manera més genèrica per a poder accedir a la UI des de un fil. En la següent línia podeu veure un exemple de com es faria.

```
barra.Dispatcher.Invoke(()=> TextBox.Text = "Ara sí que puc escriure a la UI");
```

El mètode `Dispatcher.Invoke` permet executar un mètode delegat en el subprocés associat al control que executa el `Dispatcher`. Dit en altre paraules, li podem passar al fil on estar la UI el mètode que cal executar vinculat a un control de la UI.

Com a mètode delegat hem de passar alguna *Action* o un *Delegate*; és a dir, un conjunt de línies de codi que cal executar. Una manera de fer-ho és utilitzant les *lambda expression* com es mostra en l'exemple anterior i passant directament el codi de modificació del control de la UI.

Un altra manera de passar el codi a executar a l'*Invoke* seria mitjançant un mètode delegat, de la següent manera:

```
public void EscriuUI()
{
    //Codi corresponent a la UI
    myLabel.Content = "hola";
}

void MetodeEnFilSecundari()
{
    Iniciats.Dispatcher.Invoke(new Action(EscriuUI));
}
```

Veient el codi anterior es planteja ràpidament un dubte, com podríem passar paràmetres a al delegat `EscriuUI`? Per fer això ho haurem de fer utilitzant un altre mètode similar a *Invoke*, que és el *BeginInvoke*. I ho faríem de la següent manera:

```
//Definim un delegat
private delegate void EscriuParametresUIDelegate(string text);

//Definim el mètode que passarem al delegat
public void EscriuUIParametres(string text)
{
    //Codi corresponent a la UI
    Iniciats.Content = text;
}

void MetodeEnFilSecundari
{
    //Definim els paràmetres a passar al delegat
    string[] args = new string[]{"hola"};

    //Cridem BeginInvoke amb el mètode delegat i els seus paràmetres
    Iniciats.Dispatcher.BeginInvoke(new
    EscriuParametresUIDelegate(EscriuUIParametres), args);
}
```

### ***BackgroundWorker per a tasques lentes en segon pla***

*(això no s'ha explicat a classe, ho deixem com a coneixement extra)*

Com hem dit, la classe *BackgroundWorker* (BGW) permet l'execució de tasques en un segon pla, per evitar bloquejar temporalment la UI.

Anem a explorar al detall com funciona aquesta classe. Una gran part de l'explicació següent es troba en el següent enllaç:

<http://www.codeproject.com/Articles/99143/BackgroundWorker-Class-Sample-for-Beginners>

L'objectiu de la classe BGW és poder executar una tasca en segon pla i també informar a la UI sobre el progrés d'aquesta tasca.

El primer que cal tenir en compte quan utilitzem el BGW és que aquest funciona a partir d'events. Hi ha tres events principals que permetran gestionar el BGW:

**DoWork:** aquest event s'activarà quan s'executi, des de la UI, l'ordre d'inici del BGW. El codi associat a aquest event serà la tasca en si a executar en segon pla i no podrà accedir a la UI.

**ProgressChanged:** aquest event s'activarà quan, des del BGW s'especifiqui un canvi en el progrés de la tasca. Aquest event serveix per tal de poder accedir als controls de la ja que el codi associat a aquest event si que podrà accedir als controls de la UI.

**RunWorkerCompleted:** aquest event s'activarà quan el BGW acabi. El codi associat a aquest event permetrà saber per quin motiu ha finalitzat el BGW (cancel·lat, finalitzat correctament o error). També podrà accedir a la UI.

### ***Com associar una tasca en segon pla al BGW***

El primer que farem serà associar a l'event *DoWork*, la tasca a executar en segon pla. Vegem-ho en un exemple:

La nostra tasca feixuga és la descàrrega d'un fitxer, segons el codi que hem vist anteriorment.

```
private void descarrega_fitxer(int mida, string nom)
{
    for (int i = 1; i <= mida; i++)
    {
        descarrega_1MB();
    }
}
```

```
}  
  
private void descarrega_1MB()  
{  
    Thread.Sleep(1000);  
}
```

Per associar aquest codi a l'event *DoWork* ho farem de la següent manera:

Primer creem una nova instància de la classe *BGW*:

```
BackgroundWorker MyBGW = new BackgroundWorker();
```

I associem l'event *DoWork* al codi (delegat) que s'executarà en segon pla. Per definir un delegat de l'event *DoWork* ho haurem de fer segons la definició d'aquest delegat, per tant caldrà modificar breument el mètode *descarrega\_fitxer*:

```
void descarrega_fitxer(object sender, DoWorkEventArgs e)
```

Com podeu el delegat *DoWork* ha de tenir dos paràmetres, el sender és l'objecte que ha generat l'event i el *DoWorkEventArgs*, conté tota la informació necessària per poder processar l'event.

El mètode delegat *descarrega\_fitxer* quedarà de la següent forma:

```
//Definim l'event DoWorkEventHandler, que és el que s'encarrega de fer la  
tasca en segon pla, en el nostre cas la tasca és el descarrega_fitxer  
private void descarrega_fitxer(object sender, DoWorkEventArgs e)  
{  
    for (int i = 1; i <= mida; i++)  
    {  
        descarrega_1MB();  
    }  
}
```

Fixeu-vos que el mètode delegat *descarrega\_fitxer*, per tal de poder funcionar ha de conèixer la mida del fitxer. La pregunta que ens hem de fer ara, doncs, és: com podem passar dades de partida a un *BGW*? Aquesta pregunta la respondrem més endavant.

Ara hem d'associar el mètode delegat de l'event *DoWork* amb el propi event *DoWork*. Això es farà de la següent manera:

```
MyBGW.DoWork += new DoWorkEventHandler(descarrega_fitxer);
```

Ja tenim definida la tasca que volem executar en segon pla, ara només cal iniciar-la. Nosaltres l'iniciarem en el moment que l'usuari faci click en el botó de "Descarrega fitxer", en aquest punt el que farem serà cridar al mètode del BGW *RunWorkerAsync()*, que activarà l'event *DoWork* i, per tant, s'executarà el codi que li hem associat anteriorment.

```
MyBGW.RunWorkerAsync();
```

Ens queda pendent saber com li passem al BGW informació en el moment de partida. En el nostre exemple ens interessaria passar-li la mida del fitxer. Això ho farem en el moment en que es crida el *RunWorkerAsync*

```
//Iniciem la tasca en segon pla passant-li com a paràmetre la informació de la  
mida del fitxer a descarregar  
MyBGW.RunWorkerAsync(Convert.ToInt32(TextBoxMida.Text));
```

En el mètode delegat de l'event *DoWork* recuperarem la informació que li passem mitjançant la propietat *Argument* dels *DoWorkEventArgs*, quedaria així:

```
int mida = (int)e.Argument;
```

En resum, el codi que tenim fins al moment és el següent:

```
//Definim un objecte BGW a nivell de classe perquè sigui visible a tota  
la classe  
private BackgroundWorker MyBGW = new BackgroundWorker();  
  
public MainWindow()  
{  
    InitializeComponent();  
}  
  
//En el moment en que l'usuari indica que vol iniciar la descarrega el  
que fem és crear el BGW, que treballarà en segon pla  
private void ButtonDescarrega_Click(object sender, RoutedEventArgs e)  
{  
    //Al BGW li hem d'indicar algunes coses perquè pugui funcionar  
  
    //Indiquem quina tasca s'ha de fer en segon pla, això es fa creant un  
event de tipus DoWorkEventHandler, que s'activarà en el moment que iniciem le BGW  
    MyBGW.DoWork += new DoWorkEventHandler(descarrega_fitxer);  
  
    //Iniciem la tasca en segon pla passant-li com a paràmetre la  
informació del fitxer a descarregar  
    MyBGW.RunWorkerAsync(Convert.ToInt32(TextBoxMida.Text));  
}  
  
//Definim l'event DoWorkEventHandler, que és el que s'encarrega de fer la  
tasca en segon pla, en el nostre cas la tasca és el descarrega_fitxer  
private void descarrega_fitxer(object sender, DoWorkEventArgs e)  
{  
    //Per tal de recuperar la informació del fitxer ho podem fer en els  
arguments de l'event, concretament a e.Arguments.  
    int mida = (int)e.Argument;  
  
    for (int i = 1; i <= mida; i++)  
    {  
        descarrega_1MB();  
    }  
}
```



```
private void descarrega_1MB()
{
    Thread.Sleep(1000);
}
```

Si executeu el programa tal i com el tenim ara veureu que la interfície d'usuari no es bloqueja com abans, però que, tampoc es veu cap mena d'informació de com avança la descàrrega del fitxer, ni tant sols, de quan acaba. Això és el que veurem tot seguit.

### ***Com informar del progrés de la tasca des del BGW:***

Tot el que fa referència a informar del progrés de la tasca en segon pla ho farem mitjançant l'event *ProgressChanged*.

Igual que abans hem de definir un delegat per l'event *ProgressChanged*. La definició d'un delegat per aquest event es farà de la següent manera:

```
void MyBGW_ProgressChanged(object sender, ProgressChangedEventArgs e)
```

Com en el cas anterior, el paràmetre *sender* fa referència a l'objecte que genera l'event i el paràmetre *ProgressChangedEventArgs* conté tota la informació necessària per a reportar sobre el progrés de la tasca. En concret ens serà molt útil la propietat *ProgressPercentage*, que conté un enter indicant el percentatge de tasca processada.

Val la pena dir, també, que mitjançant la propietat *UserState* podrem passar a l'event *ProgressChanged* qualsevol altra informació que sigui necessària.

El mètode delegat de l'event *ProgressChanged* quedaria de la següent manera:

```
//Definim l'event de ProgressChanged, que és l'encarregat d'interactuar
amb la UI i mostrar la informació a l'usuari
void MyBGW_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    //A l'argument e.ProgressPercentage hi trobem el percentatge
completat de la tasca.
    int a = e.ProgressPercentage;
    missatge.Content = "Percentatge de descàrrega: " +
e.ProgressPercentage.ToString() + "%";
}
```

Com es pot veure, en l'event *ProgressChanged* si que podem accedir als controls de la UI.

Igual que abans li hem d'associar a l'event *ProgressChanged*, el codi del delegat anterior. Això ho farem de la següent manera:

```
MyBGW.ProgressChanged += new ProgressChangedEventHandler(MyBGW_ProgressChanged);
```

També caldrà activar la propietat `MyBGW.WorkerReportsProgress = true;`

per tal que el BGW pugui utilitzar l'event de *ProgressChanged*.

Per tal d'informar sobre el progrés ho farem des de l'event *DoWork* que és realment qui està realitzant la tasca en segon pla i sap en quin punt es troba. Per fer això utilitzarem el mètode *ReportProgress* del BGW on li passarem un enter indicant el percentatge

```
MyBGW.ReportProgress(percentatge);
```

Per últim ens queda decidir que farà el BGW quan aquest acabi. Farem servir un sistema similar que en els events anteriors, però ara pel *RunWorkerCompleted*.

El primer a fer és definir el mètode delegat per aquest event, ho definirem així:

```
//Definim l'event de RunWorkerCompleted que s'executarà quan acabi la tasca en  
segon pla.  
//En aquest cas l'únic que fem és indicar que s'ha finalitzat la descàrrega  
correctamet o si s'ha cancel·lat  
void MyBGW_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)  
{  
    if (e.Cancelled)  
    {  
        missatge.Content = "La descàrrega s'ha cancel·lat";  
    }  
    else  
    {  
        missatge.Content = "La descàrrega ha finalitzat";  
    }  
}
```

En el cas de l'event *RunWorkerCompleted* disposem de tres propietats dins dels arguments (*RunWorkerCompletedEventArgs*) que ens posen resultar interessants per decidir que fem un cop hagi acabat el BGW:

- *e.Cancelled*, indica que el BGW ha finalitzat per la cancel·lació per part de l'usuari
- *e.Error*, indica que el BGW ha finalitzat amb algun error
- *e.Result*, conté el resultat del BGW si és que aquest hagués de retornar alguna cosa.

Igual que abans hem d'associar el mètode delegat amb l'event *RunWorkerCompletedEventArgs*

```
MyBGW.RunWorkerCompleted += new
```

```
RunWorkerCompletedEventHandler(MyBGW_RunWorkerCompleted);
```

Ens queda encara una cosa a fer i és permetre que l'usuari pugui cancel·lar la descàrrega del fitxer mitjançant el botó "Cancel·lar". Per tal que l'usuari pugui cancel·lar un BGW cal fer dues coses

- 1) Activar la propietat `WorkerSupportsCancellation` del BGW
- 2) Cridar el mètode `CancelAsync()`

Quan es crida el mètode `CancelAsync` el BGW no s'acabarà tot seguit. Això podria ser perillós ja que no sabem en quin punt es troba l'evolució de la tasca en segon pla i es podria cancel·lar en un moment crític que provoqués pèrdua de dades o altres incidents. El que fa `CancelAsync` és activar la propietat `CancellationPending` del BGW. Serà l'event `DoWork` qui s'encarregarà d'anar comprovant si hi ha una cancel·lació pendent, i si així es detecta es cancel·larà definitivament el BGW posant la propietat `Cancel` a `true`. Vegem-ho en un exemple (aquest codi aniria a l'event `DoWork`):

```
        if (MyBGW.CancellationPending == false)
        {
            descarrega_1MB();
        }
        else
        {
            e.Cancel = true; //En aquest punt indiquem que sortim per una
cancel·lació
            return; //En aquest punt cancel·lem realment el BGW
        }
```

Amb tot el vist anteriorment, el codi de l'exemple que hem estat programant quedaria de la següent manera:

```
public partial class MainWindow : Window
{
    //Definim un objecte BGW a nivell de classe perquè sigui visible a tota
la classe
    private BackgroundWorker MyBGW = new BackgroundWorker();

    public MainWindow()
    {
        InitializeComponent();
    }

    private void ButtonDescarrega_Click(object sender, RoutedEventArgs e)
    {
        MyBGW.DoWork += new DoWorkEventHandler(descarrega_fitxer);
        MyBGW.ProgressChanged += new
ProgressChangedEventHandler(MyBGW_ProgressChanged);
    }
}
```

```
MyBGW.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(MyBGW_RunWorkerCompleted);
MyBGW.WorkerSupportsCancellation = true;
MyBGW.WorkerReportsProgress = true;

int mida = Convert.ToInt32(TextBoxMida.Text);

//Iniciem la tasca en segon pla passant-li com a paràmetre la
informació del fitxer a descarregar
MyBGW.RunWorkerAsync(mida);
}

private void descarrega_fitxer(object sender, DoWorkEventArgs e)
{
    int mida = (int)e.Argument;

    for (int i = 1; i <= F.mida; i++)
    {
        if (MyBGW.CancellationPending == false)
        {
            descarrega_1MB();
            MyBGW.ReportProgress((i * 100) / F.mida);
        }
        else
        {
            e.Cancel = true;
            MyBGW.ReportProgress(0);
        }
    }
}

private void descarrega_1MB()
{
    Thread.Sleep(1000);
}

void MyBGW_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    int a = e.ProgressPercentage;
    missatge.Content = "Percentatge de descàrrega: " +
e.ProgressPercentage.ToString() + "%";
}

void MyBGW_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs
e)
{
    if (e.Cancelled)
    {
        missatge.Content = "La descàrrega s'ha cancel·lat";
    }
    else
    {
        missatge.Content = "La descàrrega ha finalitzat";
    }
}

private void ButtonCancel_Click(object sender, RoutedEventArgs e)
{
    MyBGW.CancelAsync();
}
```

```
}  
}
```

## 9. Sincronització bàsica de fils mitjançant semàfors

Un aspecte que cal conèixer alhora de treballar amb fils és com realitzar una sincronització bàsica quan tenim dos o més fils funcionant alhora. Recordeu que quan estem executant fils, aquests s'executen sense cap ordre previsible i, per tant, no podem tenir un control sobre el fluxe del programa.

Si cada fil fa tasques independents això no suposa cap problema, però si els fils executen tasques que depenen les unes de les altres haurem de posar algun mecanisme de sincronització.

Anteriorment ja hem vist el mètode *Join*, que permet fer una sincronització bàsica de fils i ara veurem la sincronització amb **semàfors**.

En programació de fils un semàfor és un objecte que permet aturar i reiniciar un fil en funció d'uns requisits que li indiquem al semàfor.

En el cas de C# l'objecte semàfor el podem trobar en les classes [Semaphore](#) i [SemaphoreSlim](#). Normalment utilitzarem el [SemaphoreSlim](#) ja que és una estructura molt més lleugera i perfectament útil quan els temps d'espera dels semàfors no seran molt elevats.

Per tal de declarar un [SemaphoreSlim](#) ho farem de la següent manera:

```
SemaphoreSlim Sem = new SemaphoreSlim(1);
```

En aquest cas definim un nou semàfor de nom *Sem*. El valor que passem en el moment de crear la instància del semàfor representa el nombre màxim de sol·licituds que es poden atendre.

Si posem un 1, vol dir que només deixarem passar un fil i on en podrà entrar cap altre fins que el fil que ha entrat surti.

Per tal d'utilitzar el semàfor existeixen dues operacions bàsiques

- **Wait.** El fil realitza una petició al semàfor per a poder continuar amb l'execució del codi. Si encara hi ha sol·licituds disponibles el fil podrà

continuar, si les sol·licituds ja s'han esgotat el fil passarà a l'estat «d'espera»

- **Release.** El fil surt del semàfor, per tant allibera una sol·licitud de tal manera que un altre fil que estava en espera en el semàfor, podrà continuar la seva execució.

Vegem el funcionament del semàfor amb un exemple.

Imaginem que volem fer un programa per controlar l'accés a un local. Simularem que cada fil representa un client que vol accedir al local. El local tindrà un aforament màxim de 5 clients, és a dir, que podrà atendre 5 sol·licituds alhora.

Anem a fer el programa, primer, sense semàfors per veure'n el comportament.

```
static void Main(string[] args)
{
    Random rnd = new Random();

    for (int i=0; i<=20;i++)
    {
        //Creació de 20 fils (clients)
        DataClient dataClient = new
        DataClient(rnd.Next(1000,10000), "Client"+i.ToString());

        Thread TClient = new Thread(Client);
        TClient.Start(dataClient);
    }
    Console.ReadLine();
}

static void Client (object dataClient)
{
    DataClient myDataClient = (DataClient)dataClient;

    Console.WriteLine("El client {0} vol entrar",
        myDataClient.NomClient);

    Console.WriteLine("El client {0} està dins",
        myDataClient.NomClient);

    Thread.Sleep(myDataClient.TempsDins);
    Console.WriteLine("El client {0} surt", myDataClient.NomClient);
}
```

Si executem aquest programa veiem que els clients accedeixen al local sense control i que tots entren en el moment que volen.

Ara anem a utilitzar els semàfors per tal d'afegir una sincronització bàsica que permeti que només 5 clients estiguin alhora al local.

Per això necessitem un semàfor de 5 sol·licituds.

Ho programarem de la següent manera:

```
//Creació del semàfor amb 5 sol·licituds
static SemaphoreSlim Porter = new SemaphoreSlim(3);

static void Main(string[] args)
{
    Random rnd = new Random();

    for (int i=0; i<=20;i++)
    {
        //Creació de 20 clients
        DataClient dataClient = new
DataClient(rnd.Next(1000,10000),"Client"+i.ToString());

        Thread TClient = new Thread(Client);
        TClient.Start(dataClient);
    }
    Console.ReadLine();
}

static void Client (object dataClient)
{
    DataClient myDataClient = (DataClient)dataClient;

    Console.WriteLine("El client {0} vol entrar",
myDataClient.NomClient);
    //El fil demana sol·licitud (si ja estan totes preses s'esperarà aquí)
    Porter.Wait();
    Console.WriteLine("El client {0} està dins",
myDataClient.NomClient);

    Thread.Sleep(myDataClient.TempsDins);
    Console.WriteLine("El client {0} surt", myDataClient.NomClient);
    //Quan el client surt, allibera una sol·licitud i si hi ha fils a
l'espera n'entrarà un
    Porter.Release();
}
```



## 10. Treballant amb processos

Fins ara hem vist com treballar amb fils amb C#. Hem de recordar que quan treballem en fils dins d'una mateixa aplicació, estem treballant amb un sol procés, un sol espai de memòria i diversos fils que s'executen en paral·lel.

L'ús de fils pot ser interessant per a programes que requereixin algun tipus d'optimització, ja sigui perquè fan moltes operacions d'accés remot a recursos o bé perquè tenen una gran necessitat de càlcul.

Ara bé, en d'altres ocasions, utilitzar els fils per a realitzar tasques en paral·lel no és la millor opció.

Per exemple, imaginem una aplicació client – servidor, on el servidor a d'atendre les peticions de cada client de manera simultània, és a dir, en paral·lel. Una estratègia que podríem pensar per a programar el servidor és crear una aplicació i per cada nou client que es connectés al sistema crearíem un nou fil.

Que passaria, però, si un dels fils tingués un problema i es quedés penjat. Això provocaria que tota l'aplicació es quedaria "penjada", ja que, de fet, estem treballant amb un únic procés.

Per això, molt sovint el que fem en un sistema on volem treballar amb part del codi completament independents entre elles, com per exemple les aplicacions client – servidor, és utilitzar els processos.

Seguin amb l'exemple anterior, per cada client que es connectés al servidor el que faríem seria crear un nou procés encarregat d'atendre la petició d'un client en concret. Tindríem tants processos com clients connectats. D'aquesta manera, si un dels processos tingués un problema i es "pengés", la resta de processos no es veurien afectats i, per tant, només un client es veuria afectat.

### Com crear processos en C#

Per tal de crear processos en C# disposem de la classe *Process*, que es troba en el *namespace System.Diagnostics*.

Anem a veure amb un exemple, com crear un procés. El primer que necessitem és utilitzar un programa ja existent, que serà el que associarem al procés. Recordem que tot procés executa un determinat programa, per tant, si volem crear un procés necessitem un programa ja creat (un .exe).

Agafarem d'exemple el programa que vam fer a l'inici del curs, que s'encarregava d'escriure una lletra un nombre concret de vegades:

```
static void escriure_lletra(string lletra, int num)
{
    for (int i = 0; i < num; i++)
    {
        Console.Write(lletra);
        Thread.Sleep(10);
    }
}
```

Com podeu veure, aquest mètode rep dos paràmetres, lletra i mida. Si volem que aquest mètode s'executi en un procés independent, haurem de tenir alguna manera de poder passar informació a un procés. Ho expliquem tot seguit:

### Com passar informació a un procés.

Si us fixeu, quan creem un programa en .Net, el mètode Main té una paràmetre Args[].

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

Aquest paràmetre permet passar arguments a un programa, quan aquest és executat (procés). Normalment els arguments es passen separats per espais. Per exemple, si del programa anterior d'escriure una lletra haguéssim acabat generant un escriure\_lletra.exe, per a passar-li els dos paràmetre que necessita fariem el següent a la línia de comandes.

escriure\_lletra.exe a 500

El codi del programa, passant els arguments d'entrada per línia de comandes quedaria així:

```
//Per executar aquest programa haurem de passar els arguments en el següent
format "lletra num", per exemple "a 500"
```

```
class Program
{
    static void Main(string[] args)
    {
        string lletra = args[0];
        int num = Convert.ToInt32(args[1]);

        escriure_lletra(lletra, num);
    }

    static void escriure_lletra(string lletra, int num)
    {
        for (int i = 0; i < num; i++)
        {
            Console.Write(lletra);
            Thread.Sleep(10);
        }
    }
}
```

Molt bé, ara ja tenim creat el programa que volem executar en un procés independent. El que toca ara és crear un procés en si. Per això crearem un altre projecte on es crearà i s'executarà el procés escriure lletra.

```
Process P = new Process();
```

Un cop creat el procés li haurem d'especificar la informació necessària per poder treballar. Aquesta informació s'especificarà en l'atribut *StartInfo* de la classe *Process*. Bàsicament serà:

**FileName**, que és la ruta del fitxer .exe a executar en el procés.

**Arguments**, que són els paràmetres d'entrada del procés.

Ho farem de la següent manera:

```
P.StartInfo.FileName = @"c:/ruta/del/fitxer/exe/escriure_lletra.exe";  
P.StartInfo.Arguments = "a 500";
```

I per últim només ens queda indicar al procés que s'iniciï:

```
P.Start();
```

Tot el programa quedaria de la següent manera:

Programa escriure lletra:

```
class Program
{
    static void Main(string[] args)
    {
        string lletra = args[0];
        int num = Convert.ToInt32(args[1]);

        escriure_lletra(lletra, num);
    }

    static void escriure_lletra(string lletra, int num)
    {
        for (int i = 0; i < num; i++)
        {
            Console.Write(lletra);
            Thread.Sleep(10);
        }
    }
}
```

Programa creador del procés:

```
class Program
{
    static void Main(string[] args)
    {
        Process P = new Process();

        P.StartInfo.FileName = @"c:/ruta/del/fitxer/exe/escriure_lletra.exe";
        P.StartInfo.Arguments = "a 500";

        P.Start();
    }
}
```