Escola Pia
Santa Anna-Mataró

A1. Processos i fils

UF 2. Programació de processos i fils
MP09-Programació processos i serveis
CFGS Desenvolupament d'Aplicacions Multiplataforma
Carlos Alonso Martínez
carlos.martinez@escolapia.cat
v29122017

Concepte procés

- Un programa és una seqüència d'instruccions que implementen un determinant algoritme.
- Quan s'executa un programa:
  - El SO carrega el programa a memòria (codi, espai per les variables i el stack)
  - Es comença a executar el codi

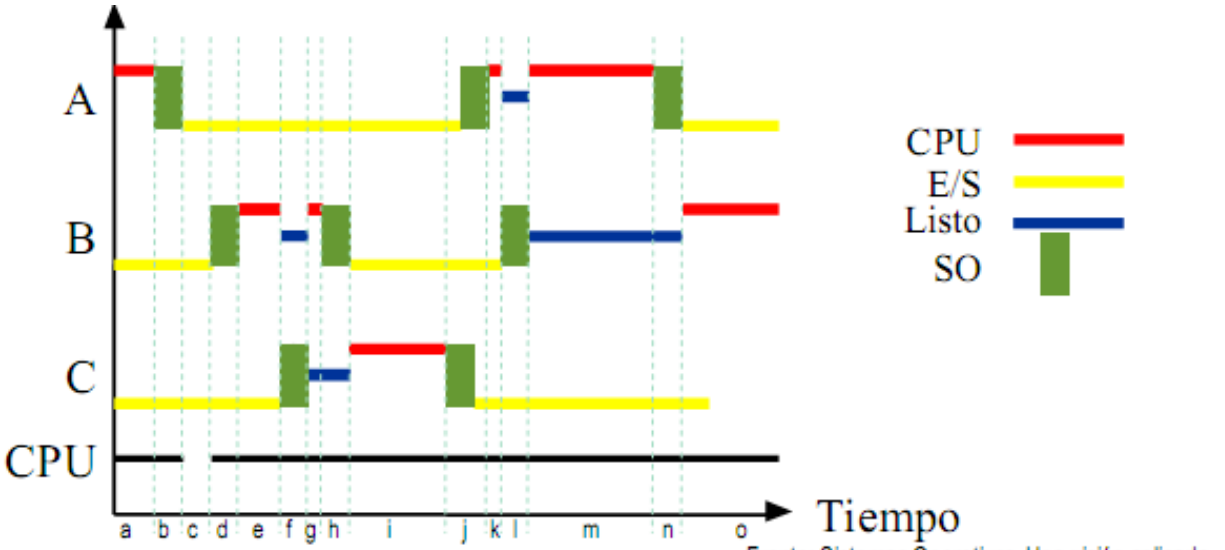
PROCÉS

Execució d'un procés

- L'execució d'un procés bàsicament implica dues accions:
  - Execucions CPU: accessos a memòria i càlculs.
  - Operacions E/S: discos, teclats, pantalla...
- Les operacions d'E/S als sistemes operatius actuals les realitza el SO, per tant el procés crida al SO -> syscall.

Multiprogramació

- Tenim diversos processos carregats a memòria (programes en execució).
- Paral·lelisme CPU i E/S gràcies a la DMA.



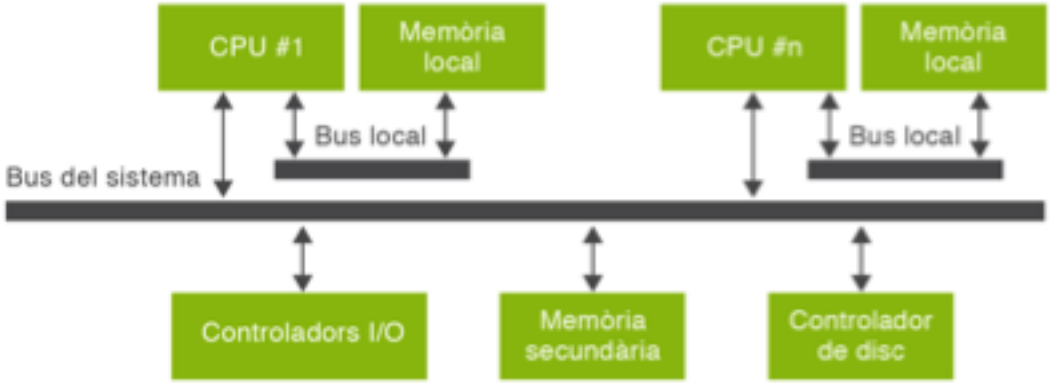
Fuente: Sistemas Operativos. Una visión aplicada.

Multitasca

- La multiprogramació només treu de la CPU un procés quan aquest necessita fer una operació d'E/S.
- **Multitasca**: els processos es van alternant per crear la sensació de simultaneïtat i respondre ràpidament a l'usuari (interactivitat).

Multiprocessador

- L'ordinador disposa de diverses CPU i a través d'un bus comparteixen memòria, recursos i perifèrics.
- Concurrència física: s'executen processos simultàniament (tants com processadors).



Multiprocessador

- Per què?
  - Tecnològicament la llei de Moore que ens assegurava doblar les prestacions de la CPU cada 24 mesos té un límit.
  - La cursa a nivell d'integració està arribant al seu límit físic.
  - Solucions:
    - Augmentar nuclis
    - Paral·lelisme

Programació concurrent

- Model de programació que permet l'execució simultània de diverses accions.

Programació paral·lela

- Els programa paral·lel està format per diversos processos que s'executen simultàniament.
- Un mateix processador pot estar executant més d'un procés del programa (multitasca).
- Concurrència real: execució més ràpida.
- Problemàtica amb la coherència de les dades amb accessos múltiples.

Java i multiprocés

- En Java és possible crear processos fills amb:
  - *Process ProcessBuilder.start()*
  - *Process Runtime.exec(String[] cmdarray, String[] envp, File dir)*
- Per defecte en Java els processos fills no moren en destruir el procés pare.

Exemple Java

```
1   import java.io.IOException;
2   import java.util.Arrays;
3   public class RunProcess {
4   public static void main(String[] args) throws IOException {
5    if (args.length <= 0) {
6     System.err.println("Se necesita un programa a ejecutar");
7     System.exit(-1);
8     }
9    ProcessBuilder pb = new ProcessBuilder(args);
10   try {
11    Process process = pb.start();
12    int retorno = process.waitFor();
13    System.out.println("La ejecución de " +
14    Arrays.toString(args) + " devuelve " + retorno);
15   }catch(IOException ex){
16    System.err.println("Excepción de E/S!!");
17    System.exit(-1);
18  }catch(InterruptedException ex){
19    System.err.println("El proceso hijo finalizó
```

.NET i processos

- Classe *System.Diagnostics.Process* permet crear i monitoritzar processos.
- Llençar processos:
  - *System.Diagnostics.Process.Start("iexplore.exe");*
- .NET té una gestió de programació multiprocés força limitada.
  - Aposta pels threads -> Tasks

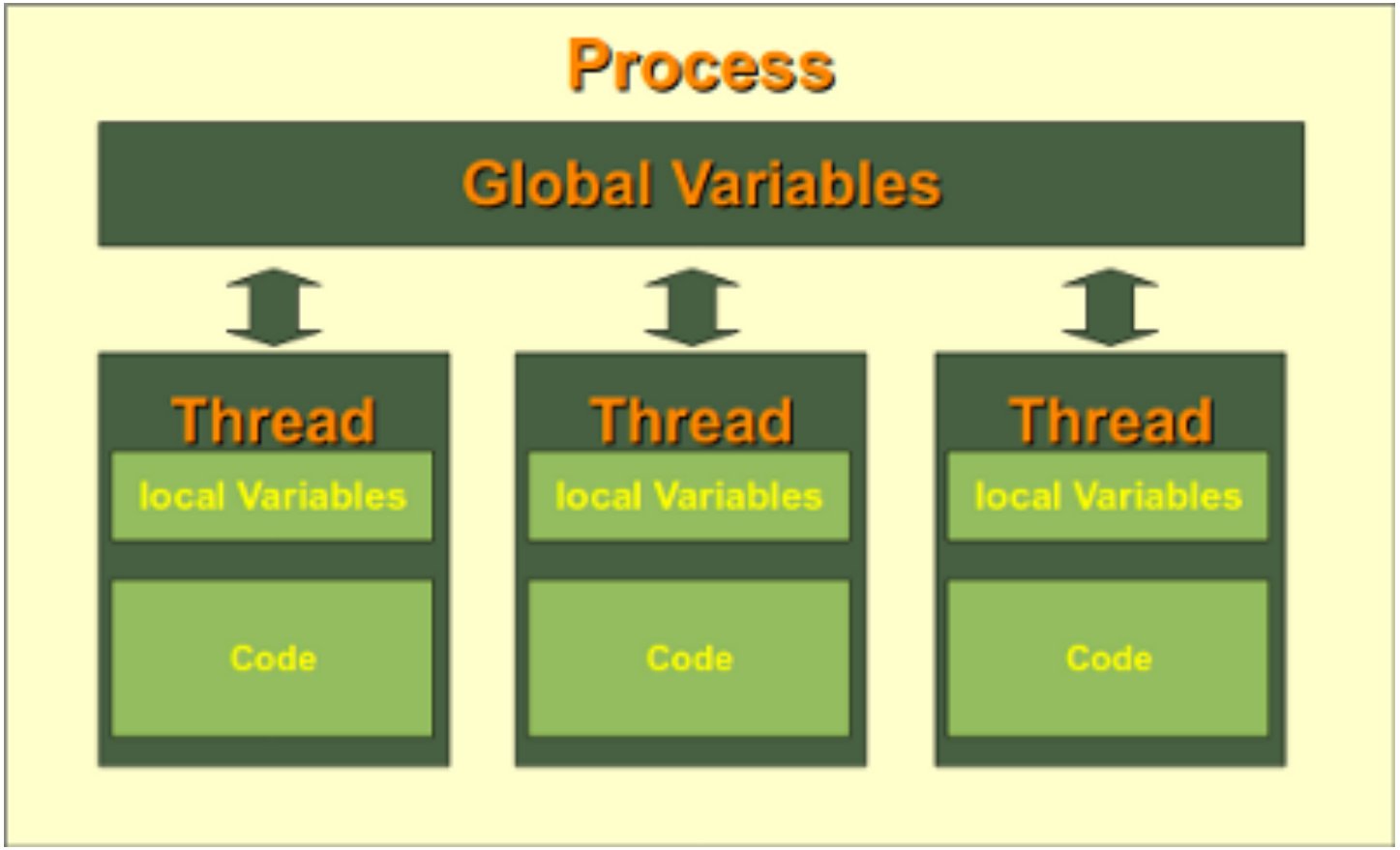
Exemple .NET

```
22    // ProcessObj.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
23
24    // This ensures that you get the output from the DOS application
25    ProcessObj.StartInfo.RedirectStandardOutput = true;
26
27    // Start the process
28    ProcessObj.Start();
29
30    // Wait that the process exits
31    ProcessObj.WaitForExit();
32
33    // Now read the output of the DOS application
34    string Result = ProcessObj.StandardOutput.ReadToEnd();
35
36    //Close Process
37    ProcessObj.Close();
38
```

Fil (thread)

- És l'entitat que dins un procés s'encarrega d'executar el codi.
- Tots els threads que conté un procés comparteixen els recursos i la memòria virtual.
- Cada thread manté un controlador d'excepcions, una prioritat, emmagatzematge local (stack) i identificador de thread únic.
- Com a mínim un procés té un thread (thread principal o main thread).

Fil (thread)


Process
Global Variables
Thread
local Variables
Code
Thread
local Variables
Code
Thread
local Variables
Code

Avantatges ús threads

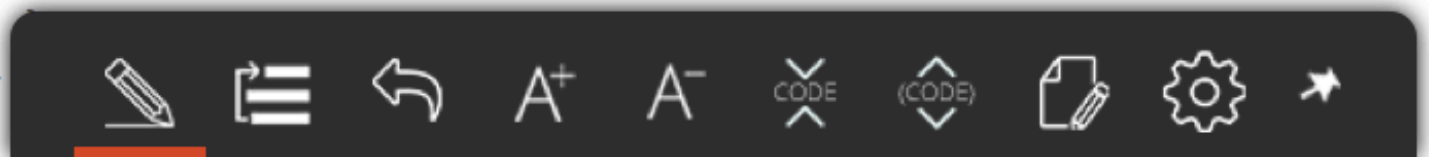
- Commutació de threads més ràpida.
- Crear i terminar threads és més ràpid.
- Baixa el nombre de commutació de processos.
- Simplifica la programació:
  - Quan un algoritme té vàries línies d'execució
- Comunicació eficient entre threads:
  - Comparteixen memòria

Quan utilitzar threads

- Un ús habitual del multithreading és per realitzar tasques en segon pla que consumeixen temps.
- Aplicacions servidor que han d'atendre diversos clients (xat) també són molt adequades per realitzar multithreading.
- Hi ha característiques de .NET com Remoting Server, Web Services i ASP.NET que implícitament funcionen amb multithreading.

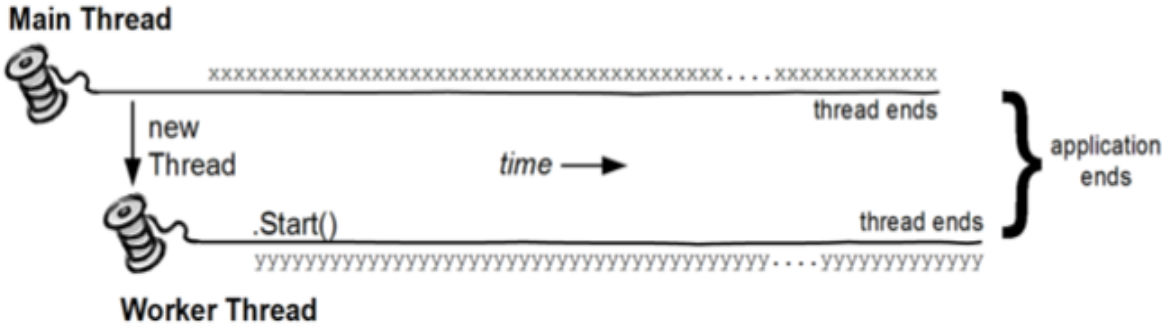
Exemple senzill

```
1   using System;
2   using System.Threading;
3   class ThreadTest
4   {
5       static void Main()
6       {
7           // Crear thread usant operador lambda permet passar
8           // paràmetres al mètode de forma molt senzilla
9           Thread t = new Thread (()=>Write("y"));
10           t.Start();              // Iniciem thread
11          // Al mateix temps al principal executem això
12          for (int i = 0; i < 1000; i++) Console.Write ("x");
13      }
14      static void Write(string lletra)
15      {
16          for (int i = 0; i < 1000; i++) Console.Write (lletra);
17
18  }
19
```



Multithreading

- Què obtenim com a resultat?
- Procés:



- Un cop iniciat un thread està actiu (la propietat IsAlive com a true) fins que finalitza).
- Un Thread finalitzat no es pot reiniciar.

Multithreading

- Els diferents threads seran gestionats pel programador de threads.
- El programador realitza time-slicing assignant temps de CPU entre els threads actius.
- Si disposem de diversos nuclis o processadors la càrrega es va repartint entre els processadors disponibles i per cada CPU es realitza time-slicing.

Variables locals

- Les variables locals són **independents** per cada thread (recordar que es creen a la pila).

```
3     // Crida a Go() en un nou thread
4     new Thread (Go).Start();
5      // Crida a Go() al  thread principal
6     Go();
7     }
8     static void Go()
9     {
10     // Declara i utilitza la variable local 'cycles'
11     for (int cycles = 0; cycles < 5; cycles++)
12      Console.Write ('?');
13    }
14
```

Si executeu aquest codi comprovareu que s'escriuen 10 '?' per pantalla.

Establint prioritats als threads

- Es poden establir prioritats entre els diferents threads
  - ThreadPriority
    - Valors possibles: Lowest, BelowNormal, Normal, AboveNormal, Highest
- La prioritat del thread sempre ve limitada per la prioritat del procés.

Sincronització bàsica

- La sincronització permet com coordinar les accions dels threads i com gestionar l'accés a les dades comunes.
- Veurem les opcions més bàsiques de sincronització (a la bibliografia en teniu informació molt més àmplia).

Dades compartides

- Per compartir dades entre Threads les opcions més senzilles són:
  - Si referencien la mateixa instància d'un objecte.
  - Utilitzant camps estàtics.
- Bàsicament aquestes dues situacions equivalen en concepte a utilitzar variables globals en C.

Dades compartides: Exemple 1

```
1   class ThreadTest
2   {
3     bool done;
4     static void Main()
5     {
6         ThreadTest tt = new ThreadTest();   // Es crea la instància
7         new Thread (tt.Go).Start();
8         tt.Go();
9     }
10    // Go ara és un mètode d'instància
11    void Go()
12    {
13      if (!done)
14      {
15        done = true;
16        Console.WriteLine ("Done");
17      }
18    }
19  }
20
```

Tant el thread principal com l'altre comparteixen el camp *done* perquè criden al mètode Go() a la mateixa instància tt. Per tant, "Done" s'escriu un cop (en teoria).

Dades compartides: Exemple 2

```
1   class ThreadTest
2   {
3    static bool done;
4   // Els camps estàtics són compartits entre threads
5
6    static void Main()
7    {
8        new Thread (Go).Start();
9        Go();
10   }
11   static void Go()
12   {
13       if (!done)
14       {
15         done = true;
16         Console.WriteLine ("Done");
17       }
18   }
19   }
20
```

El camp *done* s'ha definit com estàtic, per tant sense necessitat d'instanciar es pot accedir a ell. Implica que tots dos threads accedeixen a la mateixa posició de memòria.
El resultat aquí també (almenys en teoria) és escriure només un Done.

Race condition

- El resultat normalment serà escriure un sol *Done*, però tot plegat, no sembla gaire determinista.
- Per acabar-ho d'empitjorar si canviem l'ordre de les instruccions del mètode, passarà tot just el contrari, serà més probable escriure dos *Done*, enlloc d'un.

```
static void Go()
 {
            if (!done)  {
            Console.WriteLine ("Done");
            done = true;     }
}
```

Proveu a canviar el mètode i veure com canvia radicalment el resultat.

Secció crítica i bloqueig

- Per evitar aquestes inconsistències, es defineix la secció crítica i es bloqueja l'accés.
- D'aquesta manera, un thread pot accedir aquella secció només si l'altre no hi està accedint.

Secció crítica i bloqueig

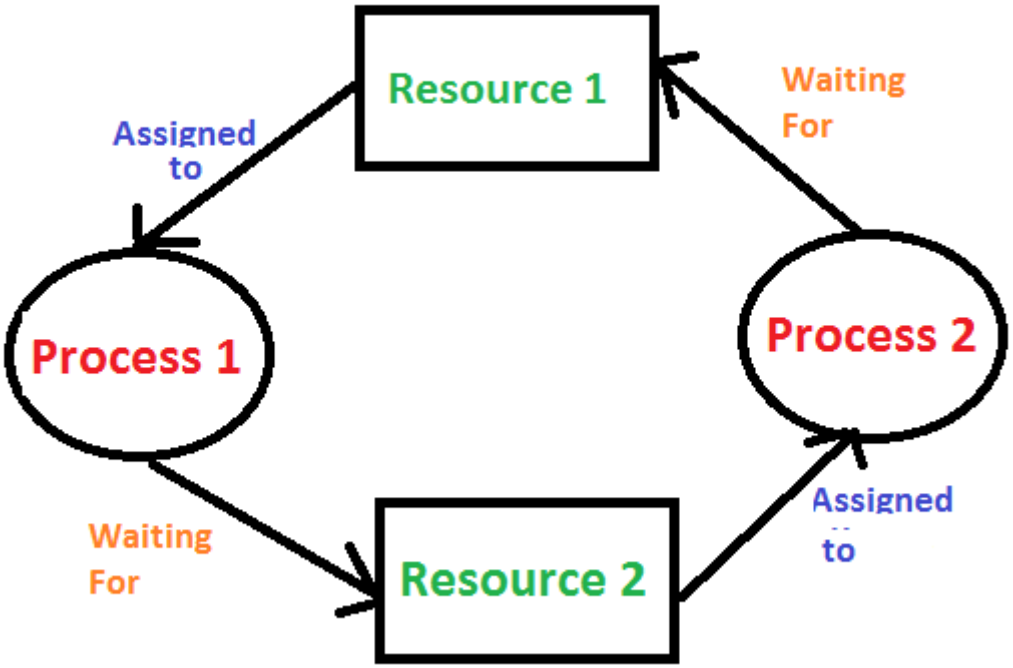
```
class ThreadSafe
{
    static bool done;  static object locker = new object();
    static void Main()
    {
          new Thread (Go).Start();
          Go();
    }
   static void Go()
   {
       lock (locker)
       {
          if (!done)
            {
               Console.WriteLine ("Done"); done = true;
             }
       }
   }
}
```

Secció crítica

Deadlocks

- Situació on dos fils bloquegen l'execució a l'entrar en dues seccions critiques.
- El CLR no detecta ni per tant pot resoldre els deadlocks que es produeixin entre threads.

Deadlock


Resource 1
Assigned to
Waiting For
Process 1
Process 2
Waiting For
Resource 2
Assigned to

Exclusió mútua (mutex)

- Mutex és similar al lock però pot treballar a través de múltiples processos.
- És força més lent (uns 50 cops) que el lock.
- Un ús molt habitual de mutex és assegura que només es pot executar una sola instància d'un programa.

Pausar un thread

- Un thread es pot pausar per tal que dormi un lapse de temps.
  - Thread.Sleep() permet parar un determinar thread un lapse de temps
    - Si no s'especifica res el temps són ms
    - Si es vol usar altre unitat temporal:
      - Thread.Sleep(TimeSpan.FromHours(1));
      - Thread.Sleep(TimeSpan.FromSeconds(30));

Join

- Podem forçar que un thread (inclós el principal, esperi a que finalitzi un altre).

```
...
Thread t = new Thread (Go);
t.Start();
t.Join();
...
```

Aquest fil d'execució quedarà blocked fins que finalitzi el thread t.
Evidentment un fil blocked no consumeix CPU

Semàfors

- Una altra forma de sincronitzar és utilitzar semàfors.
- La diferència entre un semàfor i un lock, és que el lock l'ha d'alliberar el thread que l'ha provocat, mentre que el semàfor qualsevol thread pot "posar-lo en verd".
- Els semàfors es poden utilitzar per limitar la concurrència de threads per executar un part de codi.
- A partir de .NET 4.0 hi ha una implementació més lleugera de semàfor amb menys latència, tot i que no serveix per senyalització entre processos.

Exemple de semàfor

```
1   static SemaphoreSlim sem = new SemaphoreSlim(3);
2      static void Main()
3      {
4        for (int i =0; i < 5; i++)
5          new Thread (Enter).Start(i);
6                }
7       static void Enter( object id)
8      {
9         Console.WriteLine(id + "vol entrar");
10         sem.Wait();
11         Console.WriteLine(id + "ha entrat");
12         Thread.Sleep(1000*(int) id);
13         Console.WriteLine(id + "marxa");
14        sem.Release();
15      }
16
17
```

Executeu aquest codi i observeu quin resultat s'obté a la consola.

En codi real el Thread.Sleep podria ser una operació a disc. Evitar molts accessos al disc, evita una ralentització de l'execució

Cas particulars .NET

- A .NET n'hi ha dues formes de crear threads:
- Crear threads directament a nivell de SO:
  - Es poden controlar totalment però la seva creació és costosa (crear l'stack, etc.)
- Usar ThreadPool (conjunt threads gestionat pel CLR)
  - Perdem part del control (només tenim .Join) però al ser gestionats pel CLR i no pel SO són més eficients.
- Veurem que la tendència actual és usar l'abstracció Task enlloc de definir els threads.

Fi