Programació de serveis i processos

TEMA 01 – PART II

Programació de processos i fils

1. Programació multiprocés i paral·lela
   1.3. Programació multiprocés

multiprocés i paral·lela

1.3 Gestió de processos i desenvolupament d'aplicacions amb finalitat de computació paral·lela

Un del objectius del sistema operatiu és **proporcionar als processos els recursos** necessaris per la seua **execució**. Ha d'aconseguir una **política** de **planificació** que determine quin procés farà ús del processador. El sistema operatiu és, per tant, l'encarregat de decidir **quin procés** cal **executar** i **durant quant** de **temps** cal fer-ho. Per sort, els **processos** contemplen moltes estones de **repòs** en què no els cal utilitzar el processador, per exemple esperant dades. Si en iniciar l'espera el procés avisa al sistema operatiu, aquest podrà **reemplaçar** el **procés actiu per** un **altre** en disposició d'utilitzar el processador. En **acabar** el temps de repòs, els processos hauran d'avisar de nou al sistema per tal que **planifique** una futura **execució** quan el processador quede lliure. Quan finalment els **processos finalitzen** l'execució, el sistema operatiu **alliberarà** els **recursos** utilitzats per a l'execució i deixarà lliure el processador. El temps des que un procés inicia l'execució fins que la finalitza s'anomena **cicle de vida del procés**. Per tal que el sistema puga gestionar les necessitats dels processos durant tot el seu cicle de vida, els identificarà etiquetant-los amb un **estat** que n'indique les necessitats de processament.

1.3.1. Estats d'un procés

Tots els sistemes operatius disposen d'un **planificador de processos** encarregat de repartir l'ús del processador de la forma més eficient possible i assegurant que tots els processos s'executen en algun moment. Per realitzar la planificació, el sistema operatiu es basarà en l'**estat dels processos** per saber quins **necessitaran** l'ús del **processador**. Els processos en disposició de ser executats s'organitzaran en una **cua esperant** el seu **torn**.

Depenent del sistema operatiu i de si s'està executant en un **procés** o un **fil**, la quantitat d'estats pot variar. El diagrama d'estats de la figura.10 mostra els **estats més habituals**.

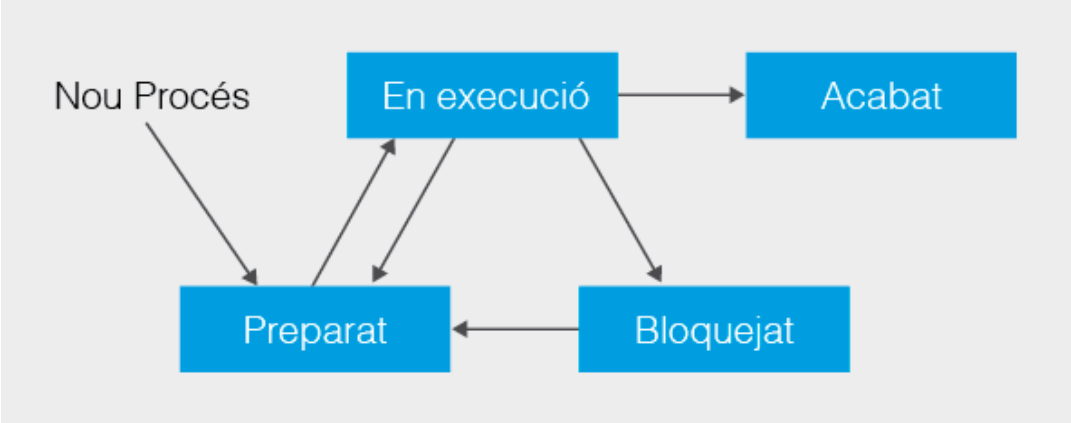

Figura 10. Estats d'un procés

El primer estat que ens trobem és **nou**, és a dir, quan un procés és creat. Una vegada creat, passa a l'estat de **preparat**. En aquest moment el procés està preparat per fer ús del processador, està **competint** pel **recurs** del processador. El **planificador** de processos del sistema operatiu és el que **decideix quan** entra el procés a **executar**-se. Quan el procés s'està executant, el seu estat s'anomena **en execució**. Una altra vegada, és el **planificador** l'encarregat de decidir quan **abandona** el **processador**.

■ Assignar un **temps d'execució fix** a **cada procés** i una vegada acabat aquest temps canviar el procés de l'estat d'execució a l'estat de **preparat** esperant de ser assignat de nou al processador, pot ser una **política del planificador** de processos per assignar els diferents **torns d'execució**.

■ El **planificador de processos** segueix una **política** que indica quan un procés ha de canviar d'estat, quan ha de deixar lliure o entrar a fer ús del processador.

Quan un procés abandone el processador, perquè el planificador de processos així ho haurà decidit, canviarà a l'estat de **preparat** i es quedarà **esperant** que el **planificador** li assigne el torn per aconseguir de nou el processador i **tornar** a l'estat d'**execució**.

Des de l'estat **d'execució**, un procés pot passar a l'estat de **bloquejat** o **en espera**. En aquest estat, el procés estarà a l'**espera d'un esdeveniment**, com pot ser una operació d'**entrada/eixida**, o l'espera de la finalització d'un **altre procés**, etc. Quan **l'esdeveniment** esperat **succeïsca**, el procés **tornarà** a l'estat de **preparat**, guardant el torn amb la resta de processos preparats.

L'últim estat és **acabat**. És un estat al qual s'hi arriba una vegada el procés ha **finalitzat** tota la seua **execució** i estarà a punt per tal que el sistema n'**allibere** quan puga els **recursos associats**.

Les **transicions** de l'estat d'un procés poden ser **provocades** per algun dels motius següents:

- D'**execució a bloquejat**: el procés realitza alguna operació d'entrada i eixida o el procés ha d'esperar que un altre procés modifique alguna dada o allibere algun recurs que necessita.
- D'**execució a preparat**: el sistema operatiu decideix traure un procés del processador perquè ha estat un temps excessiu fent-ne ús i el passa a l'estat de preparat. Assigna a un altre procés l'accés al processador.
- De **preparat a execució**: és el planificador de processos, depenent de la política que practique, l'encarregat de fer aquest canvi.
- De **bloquejat a preparat**: el recurs o la dada per la qual havia estat bloquejat està disponible o ha acabat l'operació d'entrada i eixida.
- D'**execució a acabat**: el procés finalitza les seues operacions o bé un altre procés o el sistema operatiu el fan acabar.

Cada vegada que un procés canvia a l'estat d'execució, s'anomena **canvi de context**, perquè el sistema operatiu ha d'emmagatzemar els resultats parcials aconseguits durant l'execució del procés eixint i ha de **carregar** els **resultats** parcials de

l'última execució del procés entrant en els **registres del processador**, assegurant **l'accessibilitat** a les **variables** i al **codi** que toque seguir executant.

1.3.2. Planificació de processos

> ■ La **planificació de processos** és un conjunt de **protocols** o polítiques que decideixen en quin **ordre s'executaran** els **processos** al processador. Aquestes polítiques segueixen alguns **criteris** de prioritat segons la importància del procés, el temps d'utilització del processador, el temps de resposta del procés, etc.

La funció de **planificació** (scheduler en anglès) del nucli del sistema operatiu, és l'encarregada de decidir quin procés entra al processador, pot dividir-se en tres **nivells**:

- **Planificació a llarg termini**: en el **moment** que es **crea un procés** es decideixen alguns criteris de planificació, com ara la unitat de **temps màxim** que un procés podrà romandre en execució (que s'anomena **quantum**) o la **prioritat** inicial que se li assignarà a cada procés de forma dinàmica per la utilització del processador.

- **Planificació a curt termini**: cada vegada que un **procés abandona** el **processador** cal prendre la decisió de quin serà el **nou procés** que **entrarà** a fer-ne ús. En aquest cas les polítiques intenten **minimitzar** el temps necessari per aconseguir un **canvi de context**. Els processos recentment executats solen tindre prioritat però s'ha d'assegurar que mai s'excloga permanentment cap procés de l'execució.

- **Planificació a mig termini**: existeixen altres parts del sistema operatiu que també són importants a la planificació de processos. En el cas del SWAP de memòria, si es trau un procés de la memòria per problemes d'espai fa que no puga ser planificable de forma immediata. El planificador a mig termini serà l'encarregat de decidir **quan** cal **portar** a la **memòria principal** les **dades** dels **processos**

emmagatzemats fora i quines caldrà **traslladar de** la **memòria** principal **a** l'espai **d'intercanvi**.

La figura.11 mostra gràficament els nivells de planificació de processos.

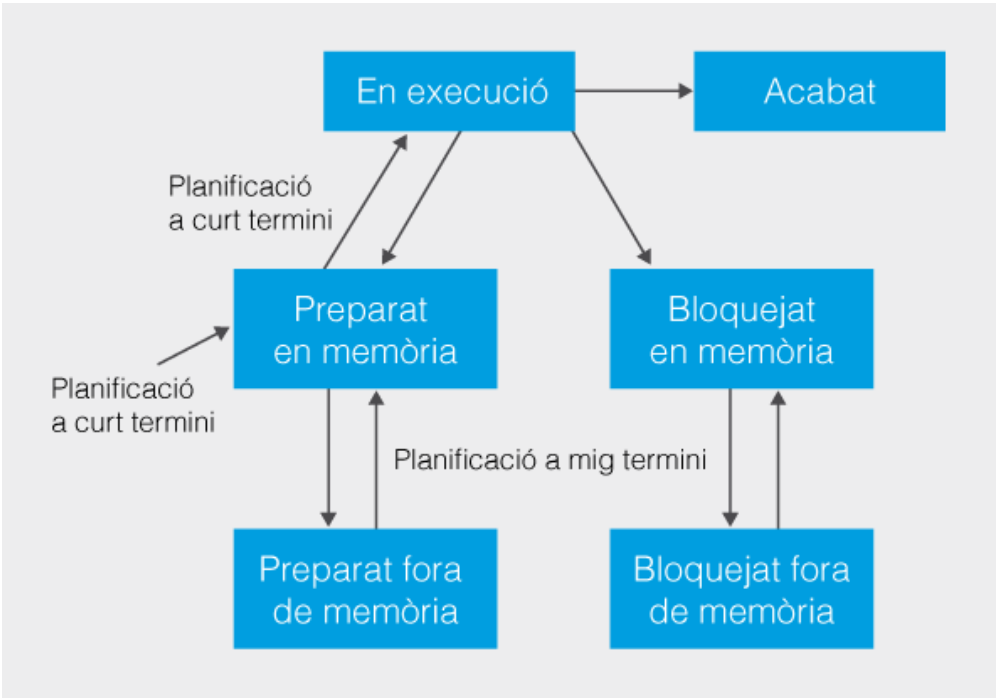

Figura 11. Planificació de processos

Els **sistemes multiprocessadors** presenten a més **dos components de planificació** específics.

- El component de **planificació temporal** en el qual es troba definida la **política de planificació que actua sobre cada processador** de forma individual, com si fóra un sistema monoprocessador. El planificador temporal decideix quan de temps cal dedicar a processar cada procés, quan cal fer canviar, per quin motiu, etc.

- El component de **planificació espacial**, en el qual es defineix **com es reparteixen els processos entre els processadors**. És a dir, organitza quin processador executa quin procés.

Aquesta planificació en els multiprocessadors utilitza alguns criteris per aplicar les polítiques d'assignació. El planificador va memoritzant **quins** són el **processos executats sobre cada processador** i **decideix** si serà adequat que torne a **executar-se sobre** el **mateix**. Una política **d'afinitat al processador** pot provocar una **sobrecàrrega** en alguns processadors i fer que **l'execució no** siga massa **eficient**. Per contra una política de **repartiment de càrrega** evita aquesta suposada infrautilització d'alguns processadors, però **augmenta** les **sobrecàrregues** a la **memòria compartida**.

1.3.3. Programació paral·lela transparent

Quan busquem resultats ràpidament o molta potència de càlcul podem pensar en augmentar el número de processadors de la nostra computadora i executar una aplicació de forma paral·lela als diferents processadors. Però l'**execució** en **paral·lel no** és **sempre possible**. En primer lloc perquè hi ha **aplicacions** que no es poden paral·lelitzar, sinó que la seua **execució** ha de ser **seqüencial**. I en segon lloc, per la **dificultat** per desenvolupar **entorns** de programació a **sistemes paral·lels**, ja que es requereix un esforç important pels programadors.

Però, i si en lloc d'aplicar la concurrència explícita, en la qual el programador aporta a l'algoritme les pautes del paral·lelisme, s'aplica una **programació** de **concurrència implícita**, en la qual és el **sistema operatiu** el que **decideix** quins **processos** es **poden dividir** i **quins no**, aplicant certes regles genèriques i comuns a la majoria de problemes? El **programador** només hauria d'**escollir** les **regles**.

S'han generalitzat els sistemes informàtics multinuclis a servidors, ordinadors de sobretaula i també a sistemes mòbils, smartphones, tauletes, etc. Tots aquests dispositius ajuden a potenciar la programació concurrent i paral·lela. Però la **programació paral·lela** és complexa, cal **sincronitzar processos** i **controlar dades compartides**, cosa que afegeix complexitat a la tasca pel programador.

La plataforma Java a Java SE 6 introdueix un conjunt de paquets que proporcionen suport a la programació concurrent i Java SE 7 millora encara més el suport a la

programació en paral·lel.

Java proporciona suport a la programació concurrent amb **llibreries de baix nivell** a través de la classe `java.lang.Thread` i de la interfície `java.lang.Runnable`. Hi ha problemes, però, que poden programar-se de forma paral·lela seguint patrons semblants. L'ús de `Thread` i `Runnable` comporta un **esforç addicional** del **programador**, obligant-lo a afegir **proves** extres per **verificar** el **comportament** correcte del codi, evitant **lectures** i **escriptures errònies**, **sincronitzant** les execucions dels processos i evitant els problemes derivats dels **bloquejos**.

Una aproximació relativament senzilla a alguns d'aquests problemes que poden resoldre's de forma paral·lela amb implantacions típiques que presenten un cert nivell de patronatge són els **marcs predefinits** de programació paral·lela del llenguatge a **Java**: `Executor` i `fork-join`. Són llibreries que **encapsulen** bona part de la **funcionalitat** i amb ella de la complexitat addicional que la programació de més baix nivell (Threads) presenta.

Ús de patrons per resoldre problemes similars

- En enginyeria, anomenem **patrons** a aquelles **solucions genèriques** que ens permeten resoldre diversos **problemes similars** canviant xicotetes coses. L'enginyeria informàtica utilitza aquest concepte per referir-se a **biblioteques complexes** que permeten un alt grau de configuració malgrat que per això siga necessari escriure aquelles parts específiques que adaptaran el patró al problema que s'intente resoldre.

Java utilitza aquest concepte en moltes de les seues biblioteques definint una estructura base a través d'**interfícies** i **classes genèriques**. El programador pot **reutilitzar** aquesta estructura i el **codi** ja **implementat**, usant-la en diferents situacions i adaptant-la amb codi específic per aquelles definicions abstractes i classes que queden indefinides en la implementació de la biblioteca.

La biblioteca **Executor** és una d'aquestes que implementa diversos patrons amb l'objectiu de solucionar de la forma més transparent possible bona part de les situacions en les quals es necessita programació paral·lela. Cada patró s'adapta a diferents situacions per la qual cosa cal saber **distingir** les **situacions** i **aplicar correctament** el **patró** que calga.

Executor

Java ens dóna eines per la simplificació de la programació paral·lela. La idea és **dividir** un **problema** en **problemes més xicotets** i cadascun d'aquests **subproblemes** enviar-los a **executar per separat**. Per tant, d'una tasca complicada en tenim diverses més simples. Per poder **executar** en **paral·lel** totes aquestes **subtasques** i així augmentar el rendiment, es **creen** diferents **subprocessos** o **fils**, però aquesta creació és **totalment transparent pel programador**, és la màquina virtual de Java l'encarregada de gestionar la creació de fils per executar en paral·lel les subtasques.

Executor és una millora sobre la creació de fils, ja que s'abstreu de la creació i la gestió dels subprocessos, De fet només cal **indicar** la **tasca** o **tasques** a realitzar de forma **paral·lela** (instanciant objectes de tipus **Runnable** o **Callable**), **escollir** el **gestor** adequat i deixar que aquest s'encarregue de tot.

A grans trets, podem dir que disposem de tres **tipus de gestors**, un de **genèric** (**ThreadPoolExecutor**), un capaç de seguir **pautes temporals** d'execució (**ScheduledThreadPoolExecutor**) i una darrer que per la seua especificitat i complexitat l'estudiarem a banda. És l'anomenat marc **Fork-Join**.

ThreadPoolExecutor, és la **classe** base que permet implementar el **gestor genèric**. Els gestors genèrics són adequats per solucionar problemes que es puguen resoldre executant **tasques independents entre si**. Imaginem que volem saber la suma d'una llista gran de nombres. Podem optar per sumar seqüencialment la llista o bé per dividir-la en dos o tres trossos i sumar cada tros per separat. Al final només caldrà

afegir el valor de cada tros al resultat. La suma de cada tros és totalment independent i per tant pot realitzar-se amb independència de la resta. És a dir, només caldria crear tantes tasques com trossos a sumar i passar-les al gestor perquè les execute.

`ThreadPoolExecutor`, **no crea un fil per cada tasca** (instància de **`Callable`** o **`Runnable`**). De fet, és habitual assignar-li **més tasques que fils**. Les tasques pendents romanen en una **cua** de **tasques** i un nombre de **fils** determinat s'encarrega d'anar-les **executant**. Quants fils cal crear? Segur que aquesta és un decisió difícil i dependrà de cada cas. Per això JAVA ens ofereix 3 **formes** ràpides de **configurar** les **instàncies** de **`ThreadPoolExecutor`** utilitzant la classe **`Executors`**. Es tracta d'una classe que aglutina un conjunt d'utilitats en forma de **mètodes estàtics**. Els tres tipus s'obtenen a partir dels mètodes estàtics següents:

- **`newCachedThreadPool()`**: crea un pool que va **creant fils a mesura que** els **necessita** i **reutilitza** els **fils inactius**.

- **`newFixedThreadPool(int numThreads)`**: crea un **pool** amb un **número** de **fils fix**. L'indicat en el paràmetre. Les tasques s'emmagatzemen en una cua de tasques i es van associant als fils a mida que aquest queden lliures.

- **`newSingleThreadExecutor()`**: crea un **pool** amb **un sol fil** que serà el que processe totes les tasques.

Qualsevol d'aquestes configuracions obtindrà una instància de **`ThreadPoolExecutor`** que es comportarà com indiquen els comentaris. Per poder afegir noves tasques, iniciar el processament paral·lel i mantenir-lo actiu fins acabar haver processat totes les tasques, la classe **`ThreadPoolExecutor`** disposa de mètodes com:

- **`execute`** per anar **introduint una tasca** (instància de **`Runnable`** o **`Callable`**) **cada vegada**.
- **`invokeAll`** per afegir, **tot d'una vegada**, una **llista** de diverses **tasques**.
- **`getPoolSize()`** per obtenir el **nombre de fils** en execució.
- **`getCompleteTaskCount()`** per saber el nombre de **tasques finalitzades**.

ScheduledThreadPoolExecutor és un **cas específic** de **pool**. Es tracta d'un gestor de fils, amb una política d'**execució** associada a una **seqüència temporal**. És a dir **executa** les **tasques programades cada cert temps**. Pot ser d'**una sola vegada** o de **forma repetitiva**. Per exemple si volem consultar el nostre correu cada 5 minuts, mentre realitzem altres feines, podem programar un fil independent amb aquesta tasca usant **`ScheduledThreadPoolExecutor`**. La tasca podria comprovar el correu i avisar només en cas que hagen arribat nous. D'aquesta manera podem concentrar-nos en la nostra feina deixant que el fil periòdic treballe per nosaltres. Podem obtenir **instàncies** de **`ScheduledThreadPoolExecutor`** usant una utilitat de la classe **`Executor`**, el mètode **`static newScheduledThreadPool(int numThreads)`**.

La classe **`ScheduledThreadPoolExecutor`** disposa dels **mètodes** següents per **gestionar** el **retard** i **cadència**:

- **`schedule (Callable task, long delay, TimeUnit timeunit)`**: crea i executa una tasca (**`task`**) que és **invocada després** d'un **temps** concret (**`delay`**) expressat en les unitats indicades (**`timeunit`**).
- **`schedule (Runnable task, long delay, TimeUnit timeunit)`**: crea i executa una acció (**`task`**), **una sola vegada**, després que haja **transcorregut** un **temps** concret (**`delay`**) expressat en les unitat indicades (**`timeunit`**).
- **`scheduleAtFixedRate (Runnable task, long initialDelay, long period, TimeUnit timeunit)`**: **crea** i **executa** una **tasca** (**`task`**) de **forma periòdica**. La primera vegada s'invocarà després d'un retard inicial (**`initialDelay`**) i posteriorment cada període de temps determinat (**`period`**). El temps es mesura en les unitats indicades (**`timeunit`**).
- **`scheduleWithFixedDelay (Runnable task, long initialDelay, long delay, TimeUnit timeunit)`**: **crea** i **executa** una **tasca** (**`task`**) de **forma periòdica**. La primera vegada s'invocarà després d'un retard inicial (**`initialDelay`**) i posteriorment amb el retard (delay) entre l'acabament d'una tasca i el començament de la següent. El temps es mesura en les unitats indicades (**`timeunit`**).

El mètodes que reben tasques de tipus **Callable** **retornaran** el **resultat obtingut** durant el **càlcul**.

*Exemple amb **ThreadPoolExecutor***

Anem ara a veure un exemple per calcular deu multiplicacions aleatòries d'enters usant un Executor amb un màxim de 3 fils. Per a la instanciació del gestor caldrà:

```
ThreadPoolExecutor executor =
(ThreadPoolExecutor) Executors.newFixedThreadPool(3);
```

Això significa que si hi ha més de 3 tasques per executar, **únicament n'enviarà 3 a executar** i la resta es quedaran **bloquejades** fins que un **fil acabe** el seu processament.

La classe interior **Multiplicacio** implementa la interfície **Callable**. Es podria fer servir el mètode **run** i implementar **Runnable**. La diferència és que **Callable** retorna valors del resultat en forma de qualsevol objecte. La **concurrència** es realitza dins del mètode **call()**.

Es crea una **llista de deu multiplicacions** a l'atzar. I després **crida** al **mètode** d'executor **invokeAll()**. Aquest rep una col·lecció de tasques de tipus **Callable**, les **executa** i en **retorna** el **resultat** dins d'un **objecte** **Future**. La classe **java.util.concurrent.Future** és una interfície dissenyada per donar suport a les classes que mantinguen **dades calculades** de forma **asíncrona**. Les instàncies de **Future** obliguen a utilitzar els mètodes **get** per accedir als resultats. Això assegura que el **valor real** només serà **accessible** quan els **càlculs hagen acabat**. Mentre duren els càlculs la invocació del mètode **get** implicarà un bloqueig del fil que l'estiga invocant.

Els objectes **Future** es generen com a conseqüència d'executar el mètode **call** de les instàncies **Callable**. Per cada **Callable** invocada es genera un objecte **Future** que romandrà bloquejat fins que la instància **Callable** associada finalitze.

Exemple MultiplicaLlista:

```
package multiplicallista;

import java.util.*;
import java.util.concurrent.*;

public class MultiplicaLlista {

static class Multiplicacio implements Callable<Integer> {
 private int operador1;
 private int operador2;
 public Multiplicacio(int operador1, int operador2) {
   this.operador1 = operador1;
   this.operador2 = operador2;
 }
 @Override
 public Integer call() throws Exception {
     return operador1 * operador2;
 }
}
       public    static    void    main(String[]    args)    throws
InterruptedException,    ExecutionException {
      ThreadPoolExecutor executor =
          (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
                       List<Multiplicacio>    llistaTasques=    new
ArrayList<Multiplicacio>();
      for (int i = 0; i < 10; i++) {
          Multiplicacio calcula =
                    new Multiplicacio((int)(Math.random()*10),
                                      (int)(Math.random()*10));
          llistaTasques.add(calcula);
```

```
      }
     List <Future<Integer>> llistaResultats;
     llistaResultats = executor.invokeAll(llistaTasques);

     executor.shutdown();
     for (int i = 0; i < llistaResultats.size(); i++) {
        Future<Integer> resultat = llistaResultats.get(i);
        try {
              System.out.println("Resultat tasca "+i+ " és:" +
resultat.get());
        } catch (InterruptedException | ExecutionException e) {
           }
     }
  }
}
```

ThreadPoolExecutor disposa també del mètode **invokeAny(tasques)** que rep una col·lecció d'instàncies de **Callable** que anirà **executant fins que una** d'elles **finalitze sense error**. En aquest moment retornarà el resultat obtingut per la tasca finalitzada en un objecte **Future**.

Un altre mètode de **ThreadPoolExecutor** que cal conèixer és **shutdown()**, que **impedeix executar noves tasques** però **deixa** que els **fils** que estan en **execució** puguen **acabar**.

Exemple amb ScheduledThreadPoolExecutor

En el següent exemple utilitzarem l'executor **ScheduledThreadPoolExecutor** per a programar **tasques** que **s'executen periòdicament**, després d'un **retard** programat.

Primer creem un pool de dos fils del **ScheduledThreadPoolExecutor** i utilitzem

scheduleWithFixedDelay, que ens permet **programar** i **executar l'inici** de la **tasca** (**initialDelay**) i **programar** les **tasques successives** després de la primera finalització (**long period**).

Executa el mètode **run** d'**ExecutaFil**. La primera vegada al cap de 2 segons i posteriorment cada 3 segons.

Per últim, el mètode **awaitTermination** es queda a l'espera abans del **shutdown** (aturada), fins que **totes** les **tasques** hagen **acabat**, es produïsca el **timeout** (temps d'espera) o els fils **siguen finalitzats abruptament**, el que **passe primer**.

```
package tascaprogramada;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class TascaProgramada {

   public static void main(final String... args) throws
     InterruptedException, ExecutionException {
       //mostrem hora actual abans d'execució
      Calendar calendari = new GregorianCalendar();
                                System.out.println("Inici:     "+
calendari.get(Calendar.HOUR_OF_DAY)+
          ":" + calendari.get(Calendar.MINUTE) + ":" +
          calendari.get(Calendar.SECOND));
        // Crea un pool de 2 fils
      final ScheduledExecutorService schExService =
```

```
          Executors.newScheduledThreadPool(2);
        // Crea objecte Runnable
      final Runnable ob = new TascaProgramada().new ExecutaFil();
          // Programa Fil, s'inicia als 2 segons i després es va
executant cada 3 segons
                  schExService.scheduleWithFixedDelay(ob,  2,  3,
TimeUnit.SECONDS);
        // Espera per acabar 10 segons
        schExService.awaitTermination(10, TimeUnit.SECONDS);
        // shutdown
        schExService.shutdownNow();
        System.out.println("Completat");
    }
    // Fil Runnable
    class ExecutaFil implements Runnable {
        @Override
        public void run() {
            Calendar calendari = new GregorianCalendar();
           System.out.println("Hora execució tasca: "+
               calendari.get(Calendar.HOUR_OF_DAY) + ":" +
               calendari.get(Calendar.MINUTE) + ":" +
               calendari.get(Calendar.SECOND));
            System.out.println("Tasca en execució");
            System.out.println("Execució acabada");
        }
    }
}
```

Quan executem el codi anterior obtindrem per la pantalla uns resultats semblants als següents:

```
Inici: 11:29:41
```

```
Hora execució tasca: 11:29:43
Tasca en execució
Execució acabada
Hora execució tasca: 11:29:46
Tasca en execució
Execució acabada
Hora execució tasca: 11:29:49
Tasca en execució
Execució acabada
Completat
```

Fixeu-vos com la **primera execució** es produeix **després de 2 segons**, a continuació la **tasca** s'executa **cada 3 segons**.

Fork-Join

El marc **`Fork-Join`** és una **evolució** dels patrons **`Executor`**. De fet ací trobem també un **gestor de fils** (la classe **`ForkJoinPool`**), però un poquet **més sofisticat**. **Processa** les tasques amb l'**algoritme** per **robatori** (work-stealing). Això vol dir que el gestor del pool **busca fils poc actius** i hi **intercanvia tasques** en **espera**. A més les **tasques** poden **crear noves tasques** que s'incorporaran a la **cua** de **tasques pendents** per a ser executades. L'**eficiència** aconseguida és **molt alta**, ja que s'aprofita al màxim el processament paral·lel. Per aquest motiu **`Fork_Join`** és una alternativa **ideal** per algoritmes que puguen resoldre's de forma **recursiva** ja que s'aconseguirà la màxima eficiència intercanviant aquelles tasques que es troben esperant resultats per d'altres que permeten avançar en els càlculs.

La classe **`ForkJoinTask`** és una **classe abstracta** per les tasques que s'executen a **`ForkJoinPool`** i conté els mètodes **`fork`** i **`join`**.

- El mètode **`fork()`** **situa** la **tasca** invocada a la **cua d'execucions** en qualsevol moment per tal que siga planificada. Això permet a una tasca crear-ne de noves i deixar-les a punt per ser executades quan el gestor ho considere.

- El mètode **`join()`** **pararà l'execució** del **fil invocador** a l'**espera** que la **tasca invocada finalitze** l'**execució** i retorne si fóra el cas els resultats. El **bloqueig** del fil posarà en alerta el gestor **`ForkJoinPool`** que podrà **intercanviar** la **tasca** parada per una altra que estiga **en espera**.

A la figura.12 es mostra com cooperen les tasques a través dels mètodes **`fork()`** i **`join()`**.

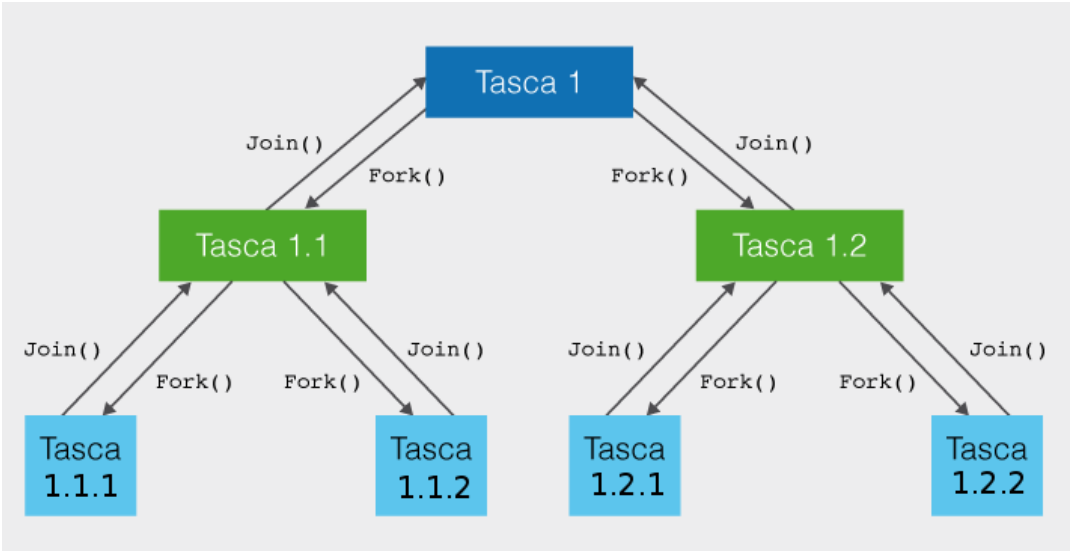

Figura 12. Mètodes fork-join

`ForkJoinTask` està implementada per dues classes que l'especialitzen, la classe **`RecursiveTask`** i la classe **`RecursiveAction`**. La primera és adequada per a tasques que necessiten **calcular** i **retornar** un **valor**. La segona en canvi representa **procediments** o **accions** que **no necessiten retornar** cap **resultat**.

Quan utilitzem **`RecursiveAction`** resulta molt pràctic **utilitzar** el mètode **`invokeAll`**, en comptes de cridar, per cada tasca generada els mètodes **`fork`** i **`join`**. Concretament el mètode **`invokeAll`** fa un invocació de **`fork`** per cada una de les tasques rebudes com a paràmetre i seguidament s'espera a la finalització de cada fil amb una crida al mètode **`join`** de les tasques engegades. **`invokeAll`** no retorna cap resultat, per tant no resulta molt útil d'utilitzar amb objectes de tipus **`RecursiveTask`**.

Algoritmes recursius i algoritmes iteratius

Els algoritmes **recursius** tenen la peculiaritat de poder-se **resoldre** realitzant diverses vegades una **mateixa operació** però **usant valors diferents**. Cada execució s'encarrega de realitzar un **càlcul parcial** fent servir les **dades** d'una o més **execucions anteriors** de la mateixa operació. El **darrer resultat parcial** calculat coincidirà amb la **solució final**. Per tal que el càlcul recursiu siga factible és **necessari** que **existisca** un **solució trivial** per algun dels valors a operar. Per exemple l'algoritme que calcula el valor factorial d'un número és recursiu perquè podem dir que el valor factorial d'un enter positiu és: n! = n* (n-1)! excepte quan n val 1 (el cas trivial) ja que sabem que 1! = 1.

Usant un llenguatge de programació podem expressar aquest algoritme recursiu fent:

```
long factorial(long n){
    if(n==1){
       return n;
    }else{
       return n*factorial(n-1);
    }
}
```

Cal recordar que els **algoritmes recursius consumeixen molta memòria** i que en un processament seqüencial, poden resultar poc eficients si el seu càlcul requereix moltes crides recursives. Però **qualsevol algoritme recursiu pot transformar-se** sempre en **iteratiu**. En un processament seqüencial, els **algoritmes iteratius són més eficients que els recursius**. Ara bé, és possible aprofitar el **processament paral·lel** i les característiques recursives d'alguns algoritmes per aconseguir **executar** en **paral·lel** les **crides recursives** de manera que incrementem l'eficiència final. Malauradament, l'**increment d'eficiència només** serà efectiu **si** el **càlcul** a realitzar **té prou entitat**.

Per donar l'entitat idònia als càlculs i agilitzar l'eficiència, es **combina** el **càlcul**

iteratiu amb el **recursiu**. És a dir, es defineix un **llindar** o condició a partir de la qual seria **preferible** resoldre el problema **iterativament**. Mentre no s'arribe al llindar el càlcul es derivarà a la **versió recursiva** executada en paral·lel, utilitzant el marc **Fork-Join**. Quan arribem al llindar, en canvi, realitzarem el càlcul seqüencialment. L'**elecció** del **llindar no és trivial** perquè d'ella dependrà en bona part disposar d'un algoritme eficient o no.

Així doncs, l'esquema d'execució de les tasques a implementar a les instàncies de **`ForkJoinTask`** seria similar al codi següent:

```
if (el que queda per resoldre ix més a compte resoldre-ho directament){
//llindar
   • execució seqüencial
}else{
   • divisió recursiva de la tasca.
   • crida/planificació de l'execució paral·lela de les tasques creades.
   • recollida dels resultats i combinació adequada per obtenir el calcul desitjat.
}
```

Veiem el que acabem d'explicar amb un **exemple**. **Buscarem** dins d'un **array** d'enters el **número més gran**. La recursió es basarà en dividir la col·lecció d'enters en dos i demanar per cada tros l'obtenció del valor màxim i el càlcul seqüencial en un algoritme de cerca iteratiu.

Usarem **`RecursiveTask`** perquè ens **cal retornar** el **valor** màxim. Els **valors de partida els passarem sempre a través del constructor de la tasca** i els emmagatzemarem com atributs per tenir-los disponibles en el mètode **`compute`** que és l'equivalent específic de les tasques **`ForkJoinTask`** al **`run`** o **`call`** ja vistos en parlar de l'**`Executor`**.

```
package maximtask;
```

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class MaximTask extends RecursiveTask<Short> {
    private static final int LLINDAR=10000000;
    private short[] arr ;
    private int inici, fi;

    public MaximTask(short[] arr, int inici, int fi) {
            this.arr = arr;
            this.inici = inici;
            this.fi = fi;
    }

    private short getMaxSeq(){
        short max = arr[inici];
        for (int i = inici+1; i < fi; i++) {
            if (arr[i] > max) {
                    max = arr[i];
            }
        }
        return max;
    }

    private short getMaxReq(){
        MaximTask task1;
        MaximTask task2;
        int mig = (inici+fi)/2+1;
        task1 = new MaximTask(arr, inici, mig);
        task1.fork();
        task2 = new MaximTask(arr, mig, fi);
```

```
        task2.fork();
        return (short) Math.max(task1.join(), task2.join());
    }

    @Override
    protected Short compute() {
        if(fi - inici <= LLINDAR){
            return getMaxSeq();
        }else{
            return getMaxReq();
        }
    }

    public static void main(String[] args) {

        short[] dades = createArray(100000000);
        System.out.println("Inici càlcul");
        ForkJoinPool pool = new ForkJoinPool();
        int inici=0;
        int  fi= dades.length;
        MaximTask tasca = new MaximTask(dades, inici, fi);

        long time = System.currentTimeMillis();
        // crida la tasca i espera que es completen
        int result1 = (int) pool.invoke(tasca);
        // màxim
        int result=  tasca.join();
        System.out.println("Temps utilitzat:"+
          (System.currentTimeMillis()-time));
        System.out.println ("Màxim es " + result);
    }
```

```
    private static short [] createArray(int size){
        short[] ret = new short[size];
        for(int i=0; i<size; i++){
            ret[i] = (short) (1000 * Math.random());
            if(i==((short)(size*0.9))){
                ret[i]=1005;
            }
        }
        return ret;
    }
}
```

Proveu **diferents llindars** per veure que l'actual és el més eficient.