

## Cinquième partie

# Programmation multiactivité

# Contenu de cette partie

Préparation aux TP : présentation des outils de programmation concurrente autour de la plateforme Java

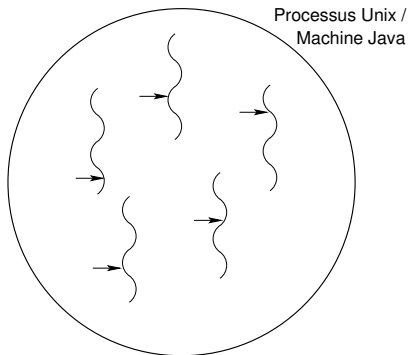
- notion d'activité
- présentation de la plateforme
- classe Thread
- objets de synchronisation : moniteurs. . .
- régulation des activités : pools d'activités, appels asynchrones, fork/join. . .
- outils de synchronisation de bas niveau
- autres environnements et modèles : Posix, OpenMP. . .

# Plan

- 1 Généralités
- 2 Threads Java
  - Manipulation des activités
  - Données localisées
- 3 Synchronisation Java
  - Moniteur Java
  - Autres objets de synchronisation
- 4 Parallélisme régulé
- 5 Autres approches
  - Synchronisation – java d'origine
  - Autres langages



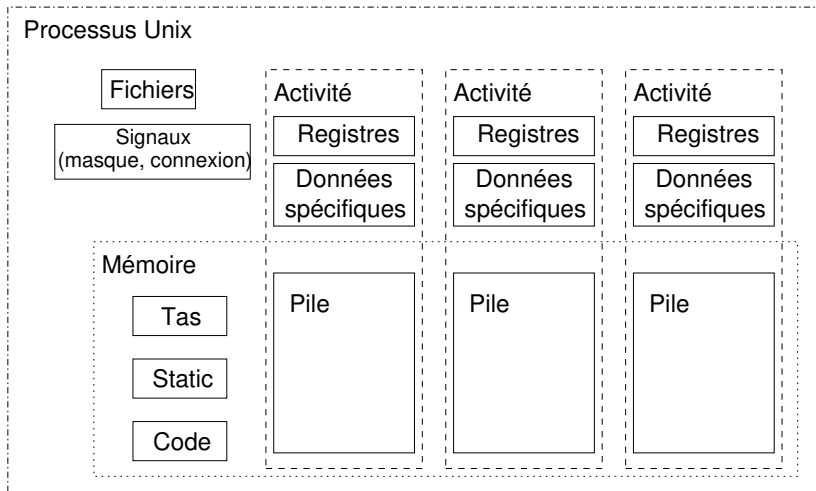
# Processus multiactivité



1 espace d'adressage, plusieurs flots de contrôle.

⇒ plusieurs **activités** (ou processus légers) au sein d'un même processus UNIX / d'une même machine virtuelle Java.

# Processus multiactivité



# Processus lourds vs activités

- *Processus lourds* : représentent l'exécution d'une application, du point de vue du système
  - **unité d'allocation de ressources**
    - isolation des espaces d'adressage
    - commutation coûteuse
- *Activités* (threads, processus légers...) :
  - **unité d'exécution** : décomposition d'un traitement en sous-traitements parallèles, pour tirer profit de la puissance de calcul disponible, ou simplifier la conception
  - les ressources (mémoire, fichiers...) du processus lourd exécutant un traitement sont partagées entre les activités réalisant ce traitement
  - une bibliothèque **applicative** gère le partage entre activités du temps processeur alloué au processus lourd
    - commutation plus efficace.

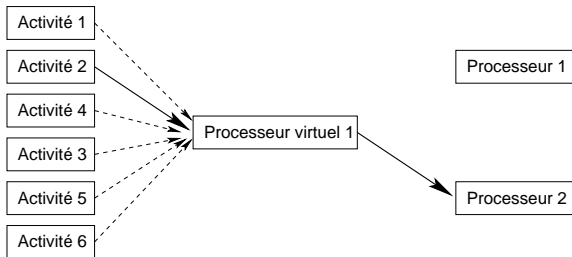
# Processeurs virtuels

Entre le processeur physique et les activités, il existe généralement une entité interne au noyau, appelé *kernel process* ou *processeur virtuel*.

Cette entité est *généralement* l'unité de blocage : un appel système bloquant (`read...`) bloque le processeur virtuel qui l'exécutait.

- ➊ Many-to-one : 1 seul processeur virtuel par processus
- ➋ Many-to-many : 1 processeur virtuel par activité
- ➌ Many-to-few : quelques processeurs virtuels par processus

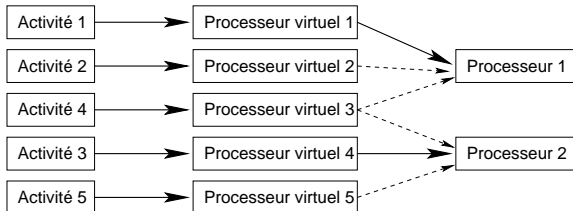
# Many-to-one



- + commutation entre activités efficace
- + implantation simple et portable
- pas de bénéfice si plusieurs processeurs
- blocage du processus (donc de toutes les activités) en cas d'appel système bloquant, ou implantation complexe

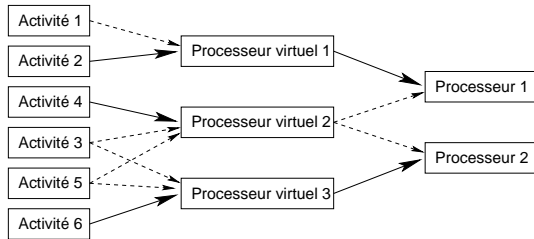


# Many-to-many



- + vrai parallélisme si plusieurs processeurs physiques
- + pas de blocage des autres activités en cas d'appel bloquant
- commutation moins efficace (dans le noyau)
- ressources consommées élevées

# Many-to-few



- + vrai parallélisme si plusieurs processeurs physiques
- + meilleur temps de commutation
- + meilleur rapport ressources/nombre d'activités
- + pas de blocage des autres activités en cas d'appel bloquant
- complexe, particulièrement si création automatique de nouveaux processeurs virtuels
- faible contrôle des entités noyau

# Plan

- 1 Généralités
- 2 Threads Java
  - Manipulation des activités
  - Données localisées
- 3 Synchronisation Java
  - Moniteur Java
  - Autres objets de synchronisation
- 4 Parallélisme régulé
- 5 Autres approches
  - Synchronisation – java d'origine
  - Autres langages

# Conception d'applications parallèles en Java

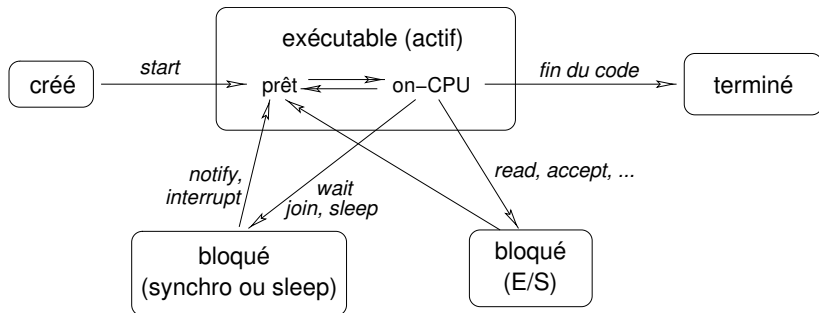
Java permet de manipuler

- les processus lourds : classes `java.lang.ProcessBuilder` et `java.lang.Process`
- les activités : classe `java.lang.Thread`

Le degré de parallélisme des applications Java peut être

- contrôlé directement (manipulation des threads)
- ou régulé
  - explicitement : interface `java.util.concurrent.Executor`
  - implicitement : programmation asynchrone/fonctionnelle

# Cycle de vie d'une activité



# Création d'une activité – interface Runnable

## Code d'une activité

```
class MonActivité implements Runnable {  
    public void run() { /* code de l'activité */ }  
}
```

## Création d'une activité

```
Runnable a = new MonActivité(...);  
Thread t = new Thread(a); // activité créée  
t.start(); // activité démarrée  
...  
t.join(); // attente de la terminaison
```

```
Thread t = new Thread(() -> { /* code de l'activité */ });  
t.start();
```

## Création d'activités – exemple

```
class Compteur implements Runnable {  
    private int max;  
    private int step;  
    public Compteur(int max, int step) {  
        this.max = max; this.step = step;  
    }  
    public void run() {  
        for (int i = 0; i < max; i += step)  
            System.out.println (i);  
    }  
}
```

```
public class DemoThread {  
    public static void main (String [] a) {  
        Compteur c2 = new Compteur(10, 2);  
        Compteur c3 = new Compteur(15, 3);  
        new Thread(c2).start();  
        new Thread(c3).start();  
    }  
}
```

# Création d'une activité – héritage de Thread

Héritage de la classe Thread et redéfinition de la méthode run :

## Définition d'une activité

```
class MonActivité extends Thread {  
    public void run() { /* code de l'activité */ }  
}
```

## Utilisation

```
MonActivité t = new MonActivité(); // activité créée  
t.start(); // activité démarrée  
...  
t.join(); // attente de la terminaison
```

Déconseillé : risque d'erreur de redéfinition de Thread.run.



# Quelques méthodes

Classe Thread :

```
static Thread currentThread()
```

obtenir l'activité appelante

```
static void sleep(long ms) throws InterruptedException
```

suspend l'exécution de l'activité appelante pendant la durée indiquée (ou jusqu'à ce que l'activité soit interrompue)

```
void join() throws InterruptedException
```

suspend l'exécution de l'activité appelante jusqu'à la terminaison de l'activité sur laquelle join() est appliquée (ou jusqu'à ce que l'activité appelante soit interrompue)

# Interruption

Mécanisme minimal permettant d'interrompre une activité.

La méthode `interrupt()` appliquée à une activité provoque :

*soit* la levée de l'exception `InterruptedException` si l'activité est bloquée sur une opération de synchronisation (`Thread.join`, `Thread.sleep`, `Object.wait...`)

*soit* le positionnement d'un indicateur `interrupted`, testable :

`boolean isInterrupted()` qui renvoie la valeur de l'indicateur de l'activité sur laquelle cette méthode est appliquée ;

`static boolean interrupted()` qui renvoie et *efface* la valeur de l'indicateur de l'activité appelante.

Pas d'interruption des entrées-sorties bloquantes  $\Rightarrow$  peu utile.

## Données localisées / spécifiques

Un **même** objet localisé (instance de `InheritableThreadLocal` ou `ThreadLocal`) possède une **valeur spécifique** dans chaque activité.

```
class MyValue extends ThreadLocal {  
    // surcharger éventuellement initialValue  
}  
  
class Common {  
    static MyValue val = new MyValue();  
}  
  
// thread t1  
o = new Integer(1);  
Common.val.set(o);  
x = Common.val.get();  
  
||  
  
// thread t2  
o = "machin";  
Common.val.set(o);  
x = Common.val.get();
```

Utilisation  $\approx$  variable globale propre à chaque activité : identité de l'activité, priorité, date de création, requête traitée. . .

# Plan

- 1 Généralités
- 2 Threads Java
  - Manipulation des activités
  - Données localisées
- 3 Synchronisation Java**
  - Moniteur Java
  - Autres objets de synchronisation
- 4 Parallélisme régulé
- 5 Autres approches
  - Synchronisation – java d'origine
  - Autres langages

# Objets de synchronisation

Le paquetage `java.util.concurrent` fournit

- une réalisation des moniteurs
- divers autres objets de synchronisation
  - barrière
  - compteur
  - sémaphore
  - ...
- le contrôle du degré de parallélisme : `Thread`, `Executor`
- des structures de données autorisant/facilitant les accès concurrents
  - accès atomiques : `ConcurrentHashMap...`
  - accès non bloquants : `ConcurrentLinkedQueue`

# Moniteur Java

## Principe des moniteurs

- 1 verrou assurant l'exclusion mutuelle
- plusieurs variables conditions associées à ce verrou
- attente/signalement de ces variables conditions
- = un moniteur
- pas de priorité au signalé et pas de file des signalés

# Moniteur Java - un producteur/consommateur (1)

```
import java.util.concurrent.locks.*;
class ProdCon {
    Lock verrou = new ReentrantLock();
    Condition pasPlein = verrou.newCondition();
    Condition pasVide = verrou.newCondition();
    Object[] items = new Object[100];
    int depot, retrait, nbElems;

    public void deposer(Object x) throws InterruptedException {
        verrou.lock();
        while (nbElems == items.length)
            pasPlein.await();
        items[depot] = x;
        depot = (depot + 1) % items.length;
        nbElems++;
        pasVide.signal();
        verrou.unlock();
    }
    ...
}
```

## Moniteur Java - un producteur/consommateur (2)

```
...  
public Object retirer() throws InterruptedException {  
    verrou.lock();  
    while (nbElems == 0)  
        pasVide.await();  
    Object x = items[retrait];  
    retrait = (retrait + 1) % items.length;  
    nbElems--;  
    pasPlein.signal();  
    verrou.unlock();  
    return x;  
}  
}
```



# Producteurs/consommateurs

Paquetage `java.util.concurrent`

## BlockingQueue

`BlockingQueue` = producteurs/consommateurs (interface)

`LinkedBlockingQueue` = prod./cons. à tampon non borné

`ArrayBlockingQueue` = prod./cons. à tampon borné

```
BlockingQueue bq = new ArrayBlockingQueue(4); // capacité  
bq.put(m);    // dépôt (bloquant) d'un objet en queue  
x = bq.take(); // obtention (bloquante) de l'objet en tête
```

# Barrière

```
java.util.concurrent.CyclicBarrier
```

Rendez-vous bloquant entre  $N$  activités : passage bloquant tant que les  $N$  activités n'ont pas demandé à franchir la barrière ; passage autorisé pour toutes quand la  $N$ -ième arrive.

```
CyclicBarrier barriere = new CyclicBarrier(3);
for (int i = 0; i < 8; i++) {
    Thread t = new Thread(
        () -> { barriere.await();
                System.out.println(" Passé !");
            });
    t.start();
}
```

Généralisation : la classe Phaser permet un rendez-vous (bloquant ou non) pour un *groupe variable* d'activités.



# Compteurs, Verrous L/R

## `java.util.concurrent.CountDownLatch`

`init(N)` valeur initiale du compteur

`await()` bloque si strictement positif, rien sinon.

`countDown()` décrémente (si strictement positif).

Lorsque le compteur devient nul, toutes les activités bloquées sont débloquées.

## `interface java.util.concurrent.locks.ReadWriteLock`

Verrous pouvant être acquis en mode

- exclusif (`writeLock().lock()`),
- partagé avec les autres non exclusifs (`readLock().lock()`)

→ schéma lecteurs/rédacteurs.

Implantation : `ReentrantReadWriteLock` (avec/sans équité)



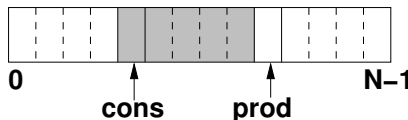
# Sémaphores

## Définition

Un sémaphore encapsule un entier, avec une contrainte de **positivité**, et deux opérations **atomiques** d'incrémentation et de décrémentation.

- contrainte de positivité : l'entier est toujours positif ou nul.
- opération **down** ou **acquire** : décrémente le compteur s'il est strictement positif ; si le compteur est nul, l'activité appelante est bloquée en attendant de pouvoir le décrémenter.
- opération **up** ou **release** : incrémente le compteur.

# Producteurs/consommateurs avec sémaphore



```
init () { mutex := new Semaphore(1);  
  vide := new Semaphore(N); // compte le nb de cases vides  
  plein := new Semaphore(0); // compte le nb de cases occupées  
}
```

```
déposer(Object v) {  
  vide.acquire();  
  // précond : pas tout plein  
  mutex.acquire(); // excl. mutuelle  
  tampon[prod] = v;  
  prod = prod + 1 % N;  
  mutex.release();  
  // postcond : pas tout vide  
  plein.release();  
}
```

```
Object retirer() {  
  plein.acquire();  
  // précond : pas tout vide  
  mutex.acquire(); // excl. mutuelle  
  Object res = tampon[cons];  
  cons = cons + 1 % N;  
  mutex.release();  
  // postcond : pas tout plein  
  vide.release();  
  return res;  
}
```

# Atomicité à grain fin

Outils pour réaliser la coordination par l'accès à des données partagées, plutôt que par suspension/réveil (attente/signal d'événement)

- le paquetage `java.util.concurrent.atomic` fournit des classes qui permettent des accès atomiques cohérents,
- et des opérations de mise à jour conditionnelle du type `TestAndSet`.
- Les lectures et écritures des références déclarées `volatile` sont atomiques et cohérentes.

⇒ synchronisation non bloquante

## Danger

Concevoir et valider de tels algorithmes est très ardu. Ceci a motivé la définition d'objets de synchronisation et de patrons (producteurs/consommateurs...)

# Plan

- 1 Généralités
- 2 Threads Java
  - Manipulation des activités
  - Données localisées
- 3 Synchronisation Java
  - Moniteur Java
  - Autres objets de synchronisation
- 4 Parallélisme régulé
- 5 Autres approches
  - Synchronisation – java d'origine
  - Autres langages

# Services de régulation du parallélisme : exécuteurs

## Idée

Séparer la création et la vie des activités des autres aspects (fonctionnels, synchronisation...)

→ définition d'un service de gestion des activités (exécuteur), régulant/adaptant le nombre d'activités effectivement actives, en fonction de la charge courante et du nombre de CPU disponibles :

- trop d'activités → consommation de ressources inutile
- pas assez d'activités → capacité de calcul sous-utilisée



# Interfaces d'exécuteurs

- Interface `java.util.concurrent.Executor` :  
`void execute(Runnable r),`
  - fonctionnellement équivalente à `(new Thread(r)).start()`
  - mais `r` ne sera pas forcément exécuté immédiatement / par une nouvelle activité.
- Interface `java.util.concurrent.ExecutorService` :  
`Future<T> submit(Callable<T> task)`  
soumission d'une tâche rendant un résultat, récupérable ultérieurement, de manière asynchrone.
- L'interface `ScheduledExecutorService` est un `ExecutorService`, avec la possibilité de spécifier un calendrier (départs, périodicité...) pour les tâches exécutées.

# Utilisation d'un Executor (sans lambda)

```
import java.util.concurrent.*;
public class ExecutorExampleOld {
    public static void main(String[] a) throws Exception {
        final int NB = 10;
        ExecutorService exec = Executors.newCachedThreadPool();
        Future<?>[] res = new Future<?>[NB];
        for (int i = 0; i < NB; i++) { // lancement des travaux
            int j = i;
            exec.execute(new Runnable() {
                public void run() {
                    System.out.println("hello" + j);
                }
            });
            res[i] = exec.submit(new Callable<Integer>() {
                public Integer call() { return 3 * j; }
            });
        }
        // récupération des résultats
        for (int i = 0; i < NB; i++) {
            System.out.println(res[i].get());
        }
    }
}
```

# Utilisation d'un Executor (avec lambda)

```
import java.util.concurrent.*;
public class ExecutorExample {
    public static void main(String[] a) throws Exception {
        final int NB = 10;
        ExecutorService exec = Executors.newCachedThreadPool();
        Future<?>[] res = new Future<?>[NB];

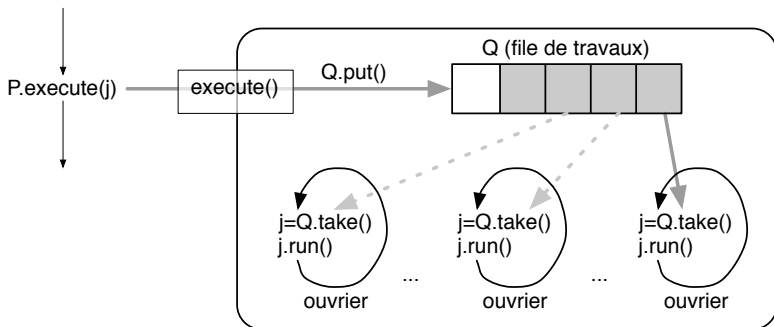
        // lancement des travaux
        for (int i = 0; i < NB; i++) {
            int j = i;
            exec.execute(() -> { System.out.println("hello" + j); });
            res[i] = exec.submit(() -> { return 3 * j; });
        }

        // récupération des résultats
        for (int i = 0; i < NB; i++) {
            System.out.println(res[i].get());
        }
    }
}
```

# Pool de Threads

Schéma de base pour la plupart des implémentations d'exécuteurs

- Une file d'attente de travaux à effectuer
- Un ensemble (fixe ou dynamique) d'activités (ouvriers)
- Une politique de distribution des travaux aux activités (réalisée par un protocole ou par une activité)



Pool P [sans politique de distribution particulière (file partagée)]

# Implantation minimale d'un pool de threads

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

public class NaiveThreadPool2 implements Executor {
    private BlockingQueue<Runnable> queue;

    public NaiveThreadPool2(int nthr) {
        queue = new LinkedBlockingQueue<Runnable>();
        for (int i=0; i<nthr; i++)
            (new Thread(new Worker())).start();
    }

    public void execute(Runnable job) { queue.put(job); }

    private class Worker implements Runnable {
        public void run() {
            while (true) {
                Runnable job = queue.take(); // bloque si besoin
                job.run();
            }
        }
    }
}
```

# Exécuteurs prédéfinis

`java.util.concurrent.Executors` est une fabrique pour des stratégies d'exécution :

- Nombre fixe d'activités : `newSingleThreadExecutor()`,  
`newFixedThreadPool(int nThreads)`
- Nombre d'activités adaptable : `newCachedThreadPool()`
  - Quand il n'y a plus d'activité disponible et qu'un travail est déposé, création d'une nouvelle activité
  - Quand la queue est vide et qu'un délai suffisant (p.ex. 1 min) s'est écoulé, terminaison d'une activité inoccupée
- Parallélisme massif avec vol de jobs :  
`newWorkStealingPool(int parallelism)`

`java.util.concurrent.ThreadPoolExecutor` permet de contrôler les paramètres de la stratégie d'exécution : politique de la file (FIFO, priorités...), file bornée ou non, nombre minimal / maximal de threads...



# Évaluation asynchrone

- Evaluation paresseuse : l'appel effectif d'une fonction peut être différée (éventuellement exécutée en parallèle avec l'appelant)
- `submit(...)` fournit à l'appelant une référence à la valeur `future` du résultat.
- L'appelant ne se bloque que quand il veut utiliser le résultat de l'appel, si l'évaluation n'est pas terminée.
  - appel de la méthode `get()` sur le Future

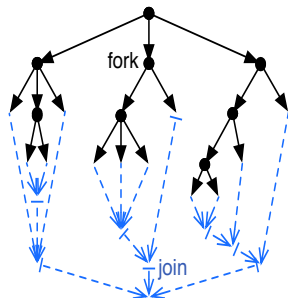
```
class FonctionAsynchrone implements Callable<TypeRetour> {  
    public TypeRetour call() { ... return v; }  
}
```

```
ExecutorService executor = Executors.newCachedThreadPool();  
Callable<TypeRetour> fonc = new FonctionAsynchrone();  
Future<TypeRetour> appel = executor.submit(fonc);  
...  
TypeRetour ret = appel.get(); // éventuellement bloquant
```

# Schéma diviser pour régner (fork/join, map/reduce)

## Schéma de base

```
Résultat résoudre(Problème pb) {  
  si (pb est assez petit) {  
    résoudre directement pb  
  } sinon {  
    décomposer le problème en parties indépendantes  
    fork : créer des (sous-)tâches  
           pour résoudre chaque partie  
    join : attendre la réalisation de ces (sous-)tâches  
    fusionner les résultats partiels  
    retourner le résultat  
  }  
}
```



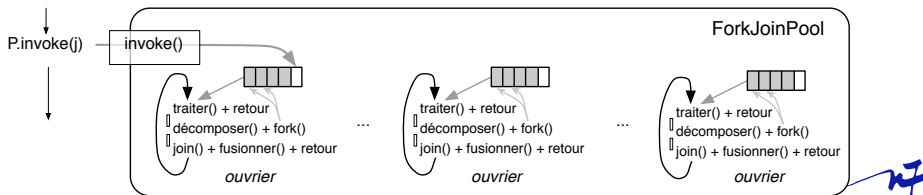


# Exécuteur pour le schéma fork/join (1/3)

Difficulté de la stratégie diviser pour régner :  
schéma exponentiel + coût de la création d'activités

## Classe ForkJoinPool

- Ensemble prédéterminé (pool) d'activités,  
**chacune** équipée d'une file d'attente de travaux à traiter.
- Les activités gérées sont des instances de **ForkJoinTask**  
(méthodes **fork()** et **join()**)



## Exécuteur pour le schéma fork/join (2/3)

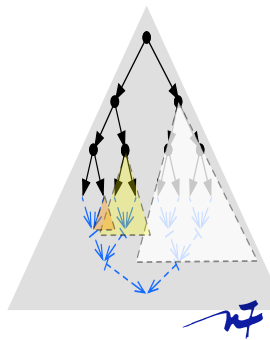
Activité d'un ouvrier du ForkJoinPool :

- Un ouvrier traite la tâche placée en **tête** de **sa** file
- Un ouvrier appelant `fork()` ajoute les travaux créés en **tête** de **sa** propre file

→

Chaque ouvrier traite un arbre de tâches qu'il

- **parcourt** et traite **en profondeur** d'abord → économie d'espace
- **construit** progressivement **en largeur**, au fur et à mesure de son parcours : lorsqu'un ouvrier descend d'un niveau, les frères de la tâche à traiter sont créés, et placés en tête de la file d'attente





# Plan

- 1 Généralités
- 2 Threads Java
  - Manipulation des activités
  - Données localisées
- 3 Synchronisation Java
  - Moniteur Java
  - Autres objets de synchronisation
- 4 Parallélisme régulé
- 5 Autres approches
  - Synchronisation – java d'origine
  - Autres langages

# Synchronisation (Java ancien)

## Obsolète

La protection par exclusion mutuelle (`synchronized`) sert encore, mais éviter la synchronisation sur objet et préférer les véritables moniteurs introduits dans Java 5.

## Principe

- exclusion mutuelle
- attente/signalement sur un objet
- équivalent à un moniteur avec **une seule** variable condition

# Exclusion mutuelle

Tout objet Java est équipé d'un verrou d'exclusion mutuelle.

## Code synchronisé

```
synchronized (unObj) {  
    // Exclusion mutuelle vis-à-vis des autres  
    // blocs synchronized(cet objet)  
}
```

## Méthode synchronisée

```
synchronized T uneMethode(...) { ... }
```

Équivalent à :

```
T uneMethode(...) { synchronized (this) { ... } }
```

(exclusion d'accès à l'objet sur lequel on applique la méthode, pas à la méthode elle-même)



# Exclusion mutuelle

Chaque classe possède aussi un verrou exclusif qui s'applique aux méthodes de classe (méthodes statiques) :

```
class X {  
    static synchronized T foo() { ... }  
    static synchronized T' bar() { ... }  
}
```

`synchronized` assure l'exécution en exclusion mutuelle pour toutes les méthodes **statiques synchronisées** de la classe X.  
Ce verrou ne concerne pas l'exécution des méthodes d'objets.

# Synchronisation par objet

Méthodes `wait` et `notify[All]` applicables à tout objet, pour lequel l'activité a obtenu l'accès exclusif.

`unObj.wait()` libère l'accès exclusif à l'objet et bloque l'activité appelante en attente d'un réveil via une opération `unObj.notify`

`unObj.notify()` réveille une unique activité bloquée sur l'objet, et la met en attente de l'obtention de l'accès exclusif (si aucune activité n'est bloquée, l'appel ne fait rien) ;

`unObj.notifyAll()` réveille toutes les activités bloquées sur l'objet, qui se mettent toutes en attente de l'accès exclusif.



## Synchronisation basique – exemple

```
class StationVeloToulouse {  
    private int nbVelos = 0;  
  
    public void prendre() throws InterruptedException {  
        synchronized(this) {  
            while (this.nbVelos == 0) {  
                this.wait();  
            }  
            this.nbVelos--;  
        }  
    }  
  
    public void rendre() {  
        // assume : toujours de la place  
        synchronized(this) {  
            this.nbVelos++;  
            this.notify();  
        }  
    }  
}
```

## Synchronisation basique – exemple

```
class BarriereBasique {  
    private final int N;  
    private int nb = 0;  
    private boolean ouverte = false;  
    public BarriereBasique(int N) { this.N = N; }  
  
    public void franchir() throws InterruptedException {  
        synchronized(this) {  
            this.nb++;  
            this.ouverte = (this.nb >= N);  
            while (! this.ouverte)  
                this.wait();  
            this.nb--;  
            this.notifyAll();  
        }  
    }  
  
    public synchronized void fermer() {  
        if (this.nb == 0)  
            this.ouverte = false;  
    }  
}
```

# Difficultés

- prises multiples de verrous :

```
synchronized(o1) { synchronized(o2) { o1.wait(); } }
```

o1 est libéré par wait, mais pas o2

- une seule notification possible pour une exclusion mutuelle donnée → résolution difficile des problèmes de synchronisation

## Pas des moniteurs de Hoare !

- programmation difficile
- affecter un objet de blocage distinct à chaque requête et gérer soit-même les files d'attente
- pas de priorité au signalé, pas d'ordonnancement sur les déblocages

# Plan

- 1 Généralités
- 2 Threads Java
  - Manipulation des activités
  - Données localisées
- 3 Synchronisation Java
  - Moniteur Java
  - Autres objets de synchronisation
- 4 Parallélisme régulé
- 5 Autres approches
  - Synchronisation – java d'origine
  - Autres langages

# PThreads : activités en C, C++

## Librairie multiactivité pour le C

- manipulation d'activités (création, terminaison...) :  
`pthread_create`, `pthread_exit`
- synchronisation : verrous, variables condition :
  - `pthread_mutex_init`, `pthread_mutex_lock`,  
`pthread_mutex_unlock`
  - `pthread_cond_init`, `pthread_cond_wait`,  
`pthread_cond_signal`, , `pthread_cond_broadcast`
- primitives annexes : données spécifiques à chaque activité,  
politique d'ordonnancement...

# Windows API (C, C++)

Plus de 150 (?) fonctions, dont :

- création d'activité : `CreateThread`
- exclusion mutuelle : `InitializeCriticalSection`,  
`EnterCriticalSection`, `LeaveCriticalSection`
- synchronisation basique : `WaitForSingleObject`,  
`WaitForMultipleObjects`, `SetEvent`
- synchronisation « évoluée » : `SleepConditionVariableCS`,  
`WakeConditionVariable`

Note : l'API Posix Threads est aussi supportée (ouf).

# .NET (C#)

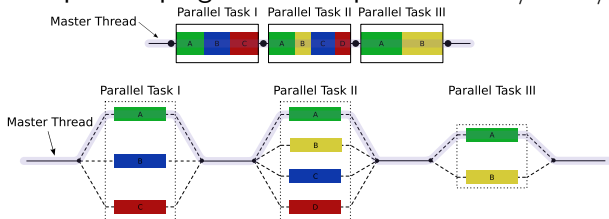
Très similaire à Java ancien :

- Création d'activité :  
`t = new System.Threading.Thread(méthode);`
- Démarrage : `t.Start();`
- Attente de terminaison : `t.Join();`
- Exclusion mutuelle : `lock(objet) { ... }`  
(mot clef du langage)
- Synchronisation élémentaire :  
`System.Threading.Monitor.Wait(objet);`  
`System.Threading.Monitor.Pulse(objet); (= notify)`
- Sémaphore :  
`s = new System.Threading.Semaphore(nbinit,nbmax);`  
`s.Release(); s.WaitOne();`



# OpenMP

- API pour la programmation parallèle en C/C++/Fortran



- Annotations dans le code, interprétées par le compilateur

## Boucle parallèle

```
int i, a[N];  
#pragma omp parallel for  
for (i = 0; i < N; i++)  
    a[i] = 2 * i;
```



# OpenMP avantages/inconvénients

- + simple
- + amélioration progressive du code
- + une seule version séquentielle / parallèle
- + peu de modifications sur le code séquentiel d'origine
- exclusivement multiprocesseur à mémoire partagée
- compilateur dédié
- peu de primitives de synchronisation (atomicité uniquement)
- gros travail sur du code mal conçu
- introduction de bugs en parallélisant du code non parallélisable

# Intel Threading Building Blocks

- Bibliothèque pour C++
- Structures de contrôles optimisées `parallel_for...`
- Structures de données optimisées `concurrent_queue...`
- Peu de primitives de synchronisation (exclusion mutuelle, verrou lecteurs/rédacteurs)
- Implantation spécialisée par modèle de processeur
- Partage de tâches par « vol de travail »
- Inconvénient : portabilité (compilateur + matériel)

# Conclusion

- Cycle de vie des activités
  - explicite
  - implicite par gestionnaire du parallélisme
- Synchronisation
  - Outil générique : moniteur (= exclusion mutuelle + conditions)
  - Objets spécifiques : producteur/consommateur, compteurs...