

# La gestion des threads et la synchronisation

table des matières

**LA GESTION DES THREADS ET LA SYNCHRONISATION.....1**

TABLE DES MATIÈRES.....2

QUELQUES GÉNÉRALITÉS.....5

*Pourquoi recourir à la programmation concurrente ?.....5*

*Les problèmes posés par la programmation concurrente.....5*

LE MODÈLE D'ACTIVITÉ AVANT JAVA5.....6

*Les concepts de base .....6*

LA CRÉATION DES ACTIVITÉS (1).....7

*L'utilisation de l'interface Runnable et de la classe Thread.....7*

LA CRÉATION DES ACTIVITÉS (2).....8

*La dérivation de la classe Thread.....8*

LA GESTION DES ACTIVITÉS (1).....9

*Les états d'un thread.....9*

*L'ordonnancement dans la JVM .....9*

LA GESTION DES ACTIVITÉS (2).....10

*Les méthodes de la classe Thread .....10*

*Thread utilisateur vs Thread daemon.....10*

LA GESTION DES ACTIVITÉS (3).....11

*Les méthodes de la classe Thread (suite).....11*

*La synchronisation sur la terminaison d'une activité.....11*

LE MODÈLE D'ACTIVITÉ DE JAVA5 (1).....12

*Les modèles d'activité de java5.....12*

LE MODÈLE D'ACTIVITÉ DE JAVA5 (2).....13

*Un exemple simple.....13*

LE MODÈLE D'ACTIVITÉ DE JAVA5 (3).....14

*La tâche et le résultat d'une tâche.....14*

*Le service d'exécution.....14*

*Quelques implémentations.....14*

LA SYNCHRONISATION : GÉNÉRALITÉS.....15

*La synchronisation générale .....15*

LA SPÉCIFICATION DE LA SYNCHRONISATION.....16

*L'expression de la synchronisation générale.....16*

*La spécification des points de synchronisation.....16*

*Un exemple simple.....16*

LA PROGRAMMATION DE LA SPÉCIFICATION DE LA SYNCHRONISATION (1).....17

*La notion de moniteur.....17*

LA PROGRAMMATION DE LA SPÉCIFICATION DE LA SYNCHRONISATION (2).....18

java avancé	chapitre 02
<i>Les méthodes synchronized</i> .....	18
L'INSTRUCTION SYNCHRONIZED.....	19
<i>La syntaxe et l'utilisation</i> .....	19
LA SÉMANTIQUE DES OBJETS.....	20
<i>Le maintien des invariants via synchronized</i> .....	20
LA PROGRAMMATION DE LA SPÉCIFICATION DE LA SYNCHRONISATION (3).....	21
<i>Un exemple (suite)</i> .....	21
LA PROGRAMMATION DE LA SPÉCIFICATION DE LA SYNCHRONISATION (4).....	22
<i>Un exemple à base d'attente passive</i> .....	22
LES VARIABLES "CONDITION".....	23
<i>La définition et l'API</i> .....	23
LA PROGRAMMATION DE LA SPÉCIFICATION DE LA SYNCHRONISATION (5).....	24
<i>Un exemple simple d'utilisation du wait() et notify()</i> .....	24
LA SYNCHRONISATION COOPÉRATIVE (1).....	25
<i>Un exemple simple</i> .....	25
<i>Une solution</i> .....	25
LA SYNCHRONISATION COOPÉRATIVE (2).....	26
<i>Un exemple plus complexe : un pool de threads simple</i> .....	26
LA SYNCHRONISATION COOPÉRATIVE (3).....	27
<i>Un exemple plus complexe : un pool de threads simple (fin)</i> .....	27
LES THREADS ET LES PERFORMANCES.....	28
<i>La création de threads</i> .....	28
<i>La synchronisation des threads</i> .....	28
<i>Les structures de données optimisées</i> .....	28
LE RAFFINEMENT DE LA LOGIQUE DE SYNCHRONISATION (1).....	29
<i>La spécialisation des variables Condition : un exemple simple</i> .....	29
LE RAFFINEMENT DE LA LOGIQUE DE SYNCHRONISATION (2).....	30
<i>La spécialisation des variables Condition : un exemple simple (suite)</i> .....	30
LE PROBLÈME DE L'INTERBLOCAGE SUR L'ACCÈS AUX RESSOURCES (1).....	31
<i>Un exemple : les deadlocks liés à l'utilisation de plusieurs verrous</i> .....	31
<i>solution</i> .....	31
LE PROBLÈME DE L'INTERBLOCAGE SUR L'ACCÈS AUX RESSOURCES (2).....	32
<i>Les conditions nécessaires pour un interblocage</i> .....	32
LE PROBLÈME DE L'INTERBLOCAGE SUR L'ACCÈS AUX RESSOURCES (3).....	33
<i>Les politiques de prévention des interblocages</i> .....	33
<i>Les techniques d'évitement des interblocages</i> .....	33
<i>Les algorithmes de détection-correction des interblocages</i> .....	33
COMPLÉMENTS SUR LA SYNCHRONISATION : LES NOUVEAUTÉS (1).....	34
<i>Les primitives de synchronisation</i> .....	34
COMPLÉMENTS SUR LA SYNCHRONISATION : LES NOUVEAUTÉS (2).....	35
<i>Les primitives de synchronisation</i> .....	35
LES THREADLOCAL.....	36
<i>Les objectifs et le principe</i> .....	36
version du 14/11/2010	page 3

<i>Quelques utilisations</i> .....	36
COMPLÉMENTS SUR LE RAFFINEMENT DE LA LOGIQUE DE SYNCHRONISATION (1).....	37
<i>Un moniteur simple pour la gestion d'un buffer circulaire</i> .....	37
COMPLÉMENTS SUR LE RAFFINEMENT DE LA LOGIQUE DE SYNCHRONISATION (2).....	38
<i>Un moniteur pour la gestion optimisée d'un buffer circulaire</i> .....	38
QUELQUES AUTRES PROBLÈMES LIÉS À LA GESTION DES ACTIVITÉS .....	39
<i>La famine de certaines activités</i> .....	39
<i>Le problème du non-déterminisme</i> .....	39
COMPLÉMENTS SUR LE PROBLÈME DE L'INTERBLOCAGE DANS LES PROBLÈMES DE COMPÉTITION.....	40
<i>L'algorithme du banquier</i> .....	40
COMPLÉMENTS SUR LA CONSTRUCTION DE MONITEURS (1).....	41
<i>La traversée d'un pont</i> .....	41
<i>Les points de synchronisation du problème (spécif d'un Véhicule [G → D])</i> .....	41
COMPLÉMENTS SUR LA CONSTRUCTION DE MONITEURS (2).....	42
<i>Le code du programme test et des véhicules</i> .....	42
COMPLÉMENTS SUR LA CONSTRUCTION DE MONITEURS (3).....	43
<i>Le code du pont (le moniteur)</i> .....	43
COMPLÉMENTS SUR LA CONSTRUCTION DE MONITEURS (4).....	44
<i>Une variante du problème .... (spécif d'un Véhicule [G → D])</i> .....	44
COMPLÉMENTS SUR LES THREADS : LES GROUPES DE THREADS, .....	45
<i>Le principe et les objectifs</i> .....	45
<i>La classe ThreadGroup</i> .....	45

## Quelques généralités

### Pourquoi recourir à la programmation concurrente ?

- simplifier la conception et la programmation d'applications conceptuellement concurrentes : simulation de systèmes réels, supervision-contrôle d'applications
- optimiser les performances par une meilleure utilisation des ressources de calcul et de stockage

### Les problèmes posés par la programmation concurrente

- la création-termination d'une activité
- la programmation du comportement correct d'une activité
  - les contraintes faisant intervenir le temps physique : le problème de l'ordonnancement
  - les contraintes ne faisant intervenir que le temps logique : le problème de la synchronisation
- l'optimisation de performances
  - l'allocation optimale des ressources du système au cours du temps
- la spécification du comportement d'une activité (spécification formelle, semi-formelle)
- la transformation d'une spécification en code exécutable
  - les actions atomiques et les sections critiques
  - les moniteurs

## Le modèle d'activité avant java5

### Les concepts de base

- la tâche : l'interface `java.lang.Runnable`
- une ressource d'exécution : la classe `java.lang.Thread`
- une activité : l'exécution d'une tâche par une ressource d'exécution (i.e. par un thread)
- définition de la tâche = définition de la méthode `run( )` de l'interface `Runnable`
- exécution de la tâche = exécution de la méthode `run( )` de l'instance de `Thread`
- démarrage de l'exécution = invocation de la méthode `start()` de `Thread`

# La création des activités (1)

## L'utilisation de l'interface Runnable et de la classe Thread

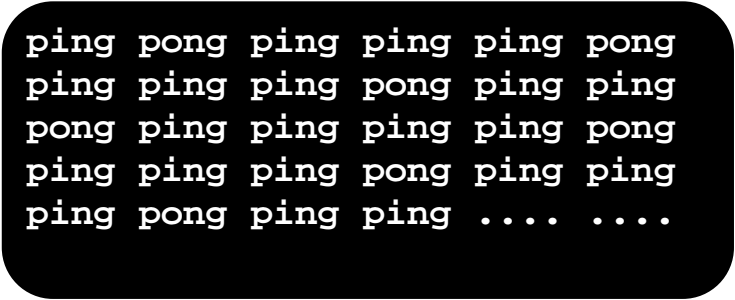
- l'interface **Runnable** ne possède qu'une méthode : **public void run()**
- l'association de l'implémentation de **Runnable** au **Thread** est faite via les constructeurs  
**Thread(Runnable target);**  
**Thread(Runnable target,String name);**  
.....

```
class PingPong implements Runnable {
    private String chaine;
    private int delai;

    public PingPong(String ch,int dl) { chaine=ch; delai=dl; }

    public void run() {
        try {
            for (int i = 0; i < 100; i++) { System.out.print(chaine+" "); Thread.sleep(delai); }
        } catch (InterruptedException e) { return; }
    }
}

class TestThread {
    public static void main(String[] args) {
        PingPong ping = new PingPong("ping",33);
        PingPong pong = new PingPong("pong",100);
        new Thread(ping).start();
        new Thread(pong).start();
    }
}
```



## La création des activités (2)

### La dérivation de la classe Thread

- on peut directement redéfinir la méthode `run()` de `Thread`  
`protected Thread();`  
`protected Thread(String name);`

```
class PingPong extends Thread {
    private String chaine;
    private int delai;

    public PingPong(String ch,int dl) { chaine=ch; delai=dl; }

    public void run() {
        try {
            for (int i = 0; i < 100; i++) { System.out.print(chaine+" "); sleep(delai); }
        } catch (InterruptedException e) {return; }
    }
}

class TestThread {
    public static void main(String[] args) {
        PingPong ping=new PingPong("ping",33);
        PingPong pong=new PingPong("pong",100);
        ping.start();
        pong.start();
    }
}
```

```
public class Thread {
    private Runnable runnable;

    public Thread(Runnable r) { runnable = r; }
    public void start() {
        .....
        run();
        .....
    }
    public void run() {
        if (runnable != null) runnable.run();
    }
    .....
}
```

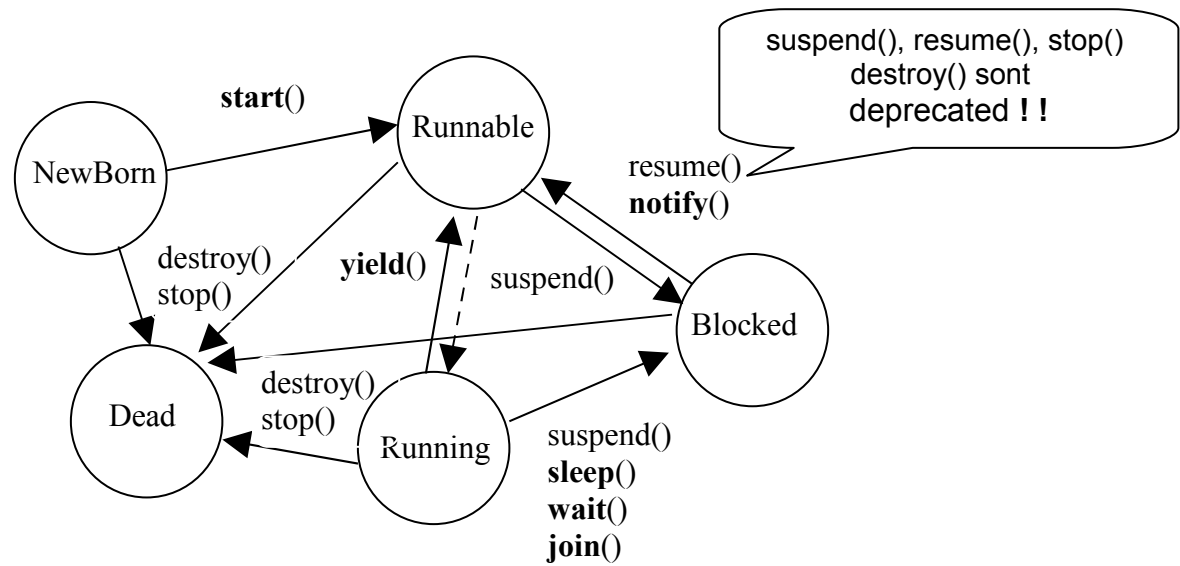
```
ping pong ping ping ping pong
ping ping ping pong ping ping
pong ping ping ping ping pong
ping ping ping pong ping ping
ping pong ping ping .... ....
```



## La gestion des activités (1)

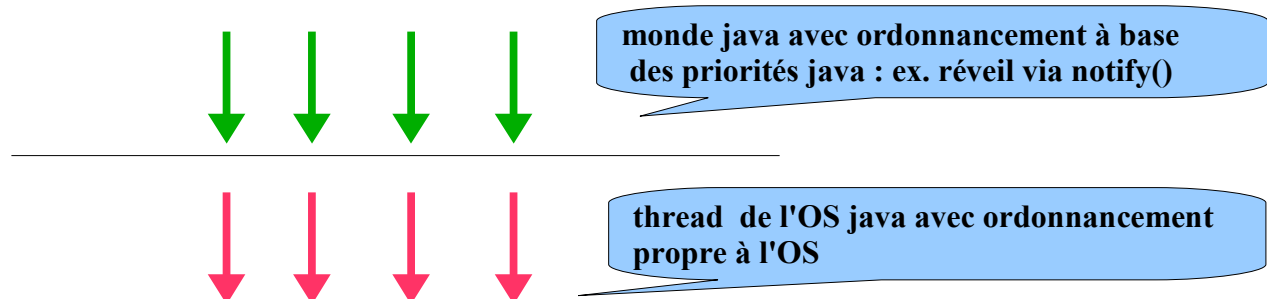
### Les états d'un thread

- modèle à 5 états



### L'ordonnancement dans la JVM

- ordonnancement préemptif à priorité
  - exécution d'un thread dans l'état **Runnable** de plus forte priorité
- priorités comprises entre `MAX_PRIORITY` (10) et `MIN_PRIORITY` (1)
- ordonnancement java = indication de l'urgence de certains threads MAIS ... AUCUNE GARANTIE REELLE !!!



## La gestion des activités (2)

### Les méthodes de la classe Thread

**static void sleep(long millis) throws InterruptedException;**  
**static void sleep(long millis,int nanos) throws InterruptedException;**

**static void yield();** //proposition du processeur pour d'autres threads

**void destroy( ) ;** //destruction sans précaution ! !

**void setPriority(int value);**  
**int getPriority();**

héritage de la priorité du thread créateur

### Thread utilisateur vs Thread daemon

- terminaison des threads daemon à la fin du dernier thread utilisateur
- héritage de l'attribut utilisateur ou daemon

**void setDaemon(boolean) ;** //invoquée avant start() sinon génération de **IllegalThreadStateException**  
**boolean isDaemon( ) ;**

## La gestion des activités (3)

### Les méthodes de la classe Thread (suite)

- la terminaison d'un Thread
  - `final void interrupt()` ; `//marquage d'un flag et déblocage du thread si bloqué`
  - `final boolean isInterrupted()` ; `//teste si le flag est positionné`
  - `final boolean interrupted()` ; `//teste si le flag est positionné avec RAZ`

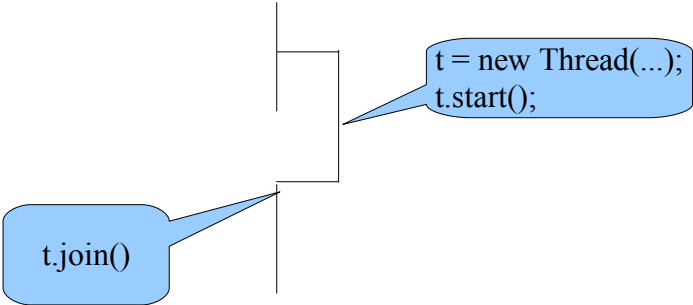
```
.....

public void run() {
    try {
        //code comportant un appel bloquant
    } catch (InterruptedException e) {
        //tâche à réaliser en cas d'interruption
    } finally {
        //tâche à réaliser en fin de tâche (libération des ressources)
    }
    .....
}
```

InterruptedException est une exception testée par le compilateur !!

### La synchronisation sur la terminaison d'une activité

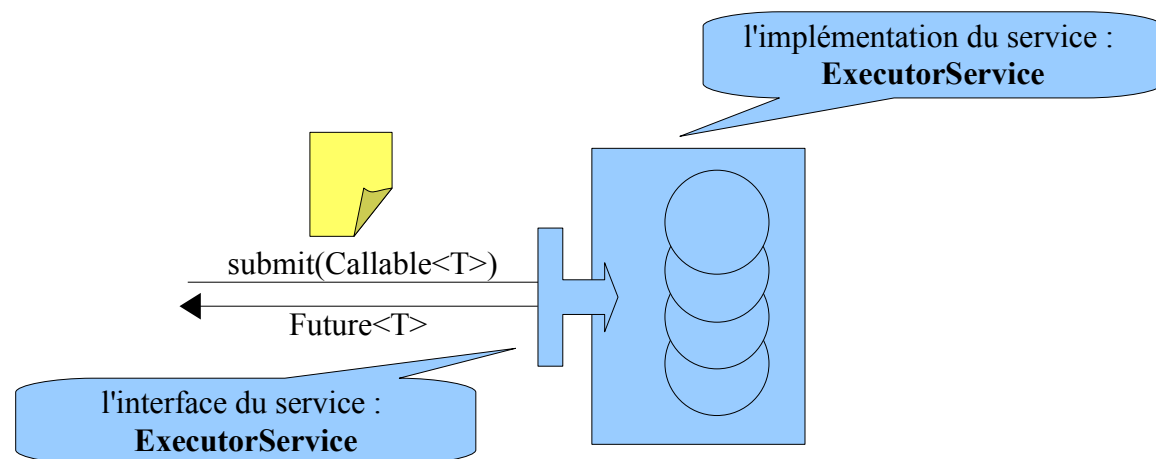
- méthodes de la classe Thread
  - `final void join() throws InterruptedException;`
  - `final void join(long millis) throws InterruptedException;`
  - `final void join(long millis,int nanos) throws InterruptedException;`



## Le modèle d'activité de java5 (1)

### Les modèles d'activité de java5

- le modèle avant java5
  - la tâche (**Runnable**)
  - la ressource d'exécution (**Thread**)
- le modèle depuis Java5
  - la tâche (**Runnable + Callable**)
  - le résultat d'une tâche (**Future**)
  - service d'exécution (**ExecutorService**)
  - ressource d'exécution (**Thread**)



## Le modèle d’activité de java5 (2)

### Un exemple simple

```
public class TestExecutorService {

    public static void main(String[] args) {
        ExecutorService exec = Executors.newFixedThreadPool(5);
        String string0 = "azertyuiopqsdfghjklmwxcvbn";
        Future[] futures = new Future[20];

        for (int i = 0; i < 20; i++) { futures[i] = exec.submit(new SubstringTask(string0, i)); }
        for (int i = 0; i < 20; i++) {
            try { System.out.println("Tache main " + i + " : " + futures[i].get()); }
            catch (Exception e) { e.printStackTrace(); }
        }
    }

    class SubstringTask implements Callable {
        private String string;
        private int nb;

        public SubstringTask(String str, int n) { string = str; nb = n; }

        public Object call() throws Exception {
            Thread.sleep((long) (5000 * Math.random()));
            String sStr = string.substring(0,nb);
            System.out.println("Tache substring " + nb + " : " + sStr);
            return sStr;
        }
    }
}
```

```
Tache substring 1 : a
Tache substring 3 : aze
Tache substring 0 :
Tache main 0 :
Tache main 1 : a
Tache substring 4 : azer
Tache substring 5 : azert
Tache substring 2 : az
Tache main 2 : az
Tache main 3 : aze
Tache main 4 : azer
Tache main 5 : azert
Tache substring 6 : azerty
Tache main 6 : azerty
Tache substring 7 : azertyu
Tache main 7 : azertyu
Tache substring 9 : azertyuio
Tache substring 10 : azertyuiop
Tache substring 11 : azertyuiopq
Tache substring 8 : azertyui
Tache main 8 : azertyui
Tache main 9 : azertyuio
Tache main 10 : azertyuiop
.....
```

## Le modèle d'activité de java5 (3)

### La tâche et le résultat d'une tâche

- Une tâche correspond aux interfaces **Runnable** et **Callable<T>**
- l'interface **Callable<T>**  
**public T call() throws Exception**  
//Elle est semblable à **Runnable**. Elle possède une unique méthode **call()**  
//semblable à **run()** mais retourne un résultat et peut générer des exceptions
- le résultat d'une tâche correspond à l'interface **Future<T>**  
**public T get();** //wrapper destiné à recevoir le résultat (de type T) d'une tâche  
//récupération blocante du résultat encapsulé dans le futur  
**public void cancel(boolean mBC);** //l'argument indique si l'annulation est possible lorsque la tâche a démarré  
**public boolean isCancelled();**  
**public boolean isDone();**

Une exception générée dans **call()** est récupérée lors du **get()**. Elle est encapsulée dans une **ExecutionException**

### Le service d'exécution

- correspond aux interfaces **Executor** et **ExecutorService**  
l'interface **ExecutorService** étend l'interface **Executor**  
**void execute(Runnable run);**  
**Future<T> submit(Callable<T> call);**  
**void shutdown();**  
**boolean isShutdown();**  
.....

Les méthodes **beforeExecute()** et **afterExcute()** sont définies dans la classe **ThreadPoolExecutor**

### Quelques implémentations

- **SingleThreadExecutor**
- **FixedThreadPool**
- **CachedThreadPool**

elles sont créés via des methodes static de la classe **Executors** :

```
ExecutorService newSingleThreadExecutor(...);  
ExecutorService newFixedThreadPool(...);  
ExecutorService newCachedThreadPool(...);  
.....
```

## La synchronisation : généralités

### La synchronisation générale ...

1. contraintes de coopération pour l'atteinte d'un objectif

- **le Rendez Vous**

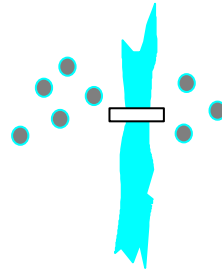
2. contraintes de compétition pour l'accès aux ressources ...

- **la traversée d'un pont**

une rivière et un pont qui l'enjambe ...

des véhicules qui circulent sur les rives et ...

traversent la rivière grâce au pont ...



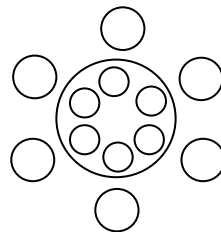
contraintes sur la traversée ...

- **les philosophes**

des philosophes assis autour d'une table dans un restaurant chinois

discutent et mangent

contraintes sur les baguettes ...



# La spécification de la synchronisation

## L’expression de la synchronisation générale

- identification et spécification des tâches des objets
- identification des points de synchronisation

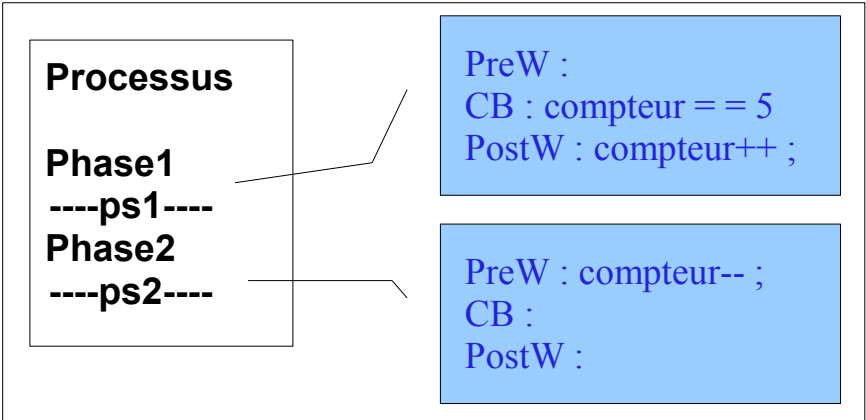
## La spécification des points de synchronisation

Point de synchronisation :  
    <PreW> : mettre à jour les variables d'état (point de synchro... atteint)  
    <CB> : **condition de blocage**  
    <PostW> : mettre à jour les variables d'état (point de synchro... franchi)

expression de <PreW>, <CB> et <PostW>  
à l'aide de données d'états "globales"

## Un exemple simple

- N Processus actifs comprenant 2 phases successives :  
    Phase1, Phase2
- Contrainte : nbre max de processus en phase2 =5
- Variable d’état : compteur (nbre de processus en phase2)





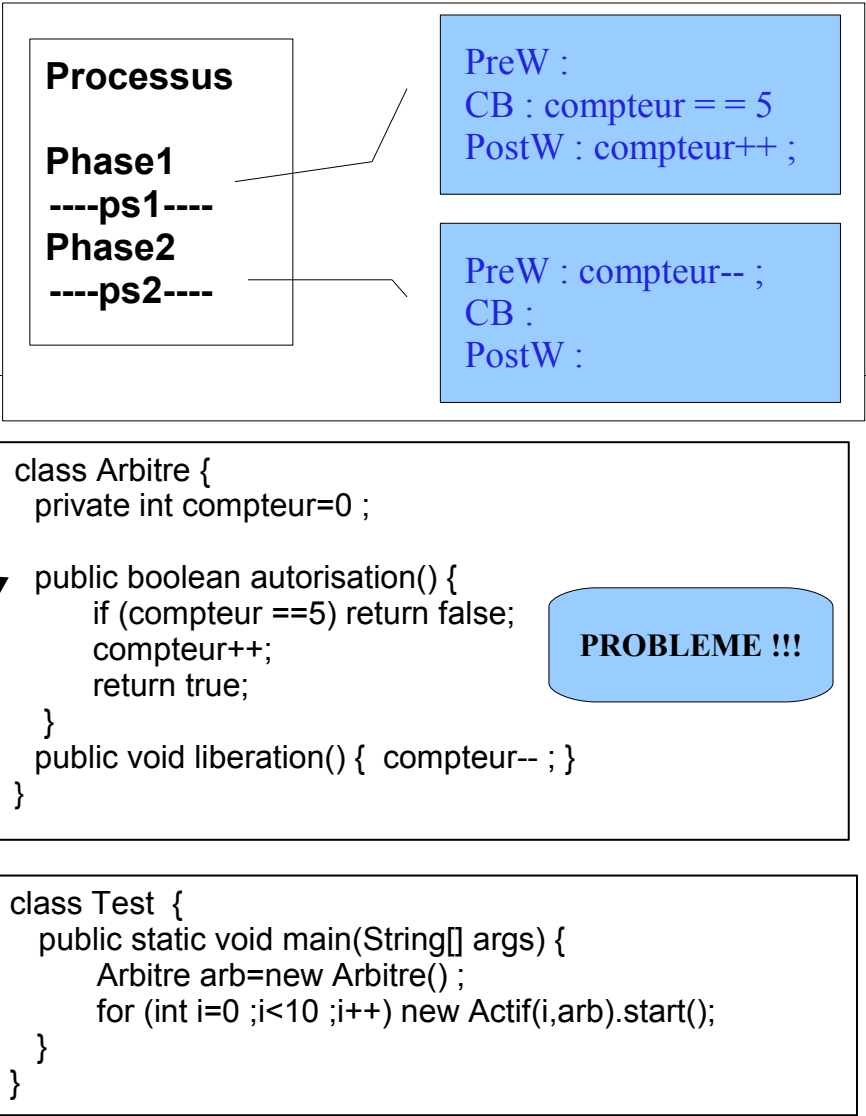
# La programmation de la spécification de la synchronisation (1)

## La notion de moniteur

- regroupement de la synchronisation dans un objet "gestionnaire de synchronisation"
- encapsulation des variables d'état
- méthodes associées aux points de synchronisation

```
class Actif extends Thread {
    private int numero;
    private Arbitre arbitre ;
    Actif(int nm,Arbitre a) { numero=nm; arbitre=a ; }

    public void run( ) {
        try {
            System.out.println(numero+" entre en phase 1");
            sleep((long)(Math.random()*1000));
            while( !arbitre.autorisation()) sleep(200);
            System.out.println(numero +" entre en phase 2");
            sleep((long)(Math.random()*1000));
            arbitre.liberation() ;
            System.out.println(numero +" se termine");
        } catch (InterruptedException e) { return; }
    }
}
```



## La programmation de la spécification de la synchronisation (2)

### Les méthodes synchronized

- le problème à résoudre : rendre une spécification exécutable
- la solution simuler l'instantanéité par l'isolation i.e. la "non-vision" des états intermédiaires
  - **synchronized** associe un verrou à des méthodes ou des blocs d'instructions

```
class Actif extends Thread {
    private int numero;
    private Arbitre arbitre ;
    Actif(int nm,Arbitre a) { numero=nm; arbitre=a ; }

    public void run( ) {
        try {
            System.out.println(numero+" entre en phase 1");
            sleep((long)(Math.random()*1000));
            while( !arbitre.autorisation()) sleep(200);
            System.out.println(numero +" entre en phase 2");
            sleep((long)(Math.random()*1000));
            arbitre.liberation();
            System.out.println(numero +" se termine");
        } catch (InterruptedException e) { return; }
    }
}
```

```
class Arbitre {
    private int compteur=0 ;

    synchronized public boolean autorisation() {
        if (compteur ==5) return false;
        compteur++;
        return true;
    }
    synchronized public void liberation() { compteur-- ; }
}
```

```
class Test {
    public static void main(String[] args) {
        Arbitre arb=new Arbitre() ;
        for (int i=0 ;i<10 ;i++) new Actif(i,arb).start();
    }
}
```

- Un **seul** verrou par **objet** pour toutes les méthodes non statiques
- Un **seul** verrou par **classe** pour toutes les méthodes statiques

# L'instruction synchronized

## La syntaxe et l'utilisation

- synchronisation d'une séquence d'instructions

- syntaxe **synchronized(expr) {**  
    **instruction**  
    **}**

expr vaut un Object

```
class Arbitre {  
    private int compteur=0 ;  
    private Object verrou = new Object();
```

```
    public boolean autorisation() {  
        synchronized(verrou) {  
            if (compteur ==5) return false;  
            compteur++;  
        }  
        return true;  
    }  
}
```

public **synchronized** ... method(...) équivalent à l'utilisation de **synchronized(this) { ... }**

```
    public void liberation() {  
        synchronized(verrou) { compteur--; }  
    }  
}
```

- possibilité de synchroniser l'accès à des tableaux

## La sémantique des objets

### Le maintien des invariants via synchronized

- le non respect de la sémantique des objets dans l'exécution concurrente

```
public class ObjetConstant {
    private int val1, val2 ;

    public ObjetConstant(int init1, int init2) { val1 = init1 ; val2 = init2 ; }
    public synchronized void transfert(int delta) { val1 += delta; val2 -= delta; }
    public synchronized int montant() { return val1 + val2 ; }
}

public class Actif extends Thread {
    private ObjetConstant cObj;

    public Actif(ObjetConstant cO) { cObj = cO; }
    public void run() {
        for (int i = 0; i < 1000; i++) {
            cObj.transfert(5);
            System.out.println("valeur de l'objet constant " + cObj.montant());
        }
    }
}
```

```
class Test {
    public static void main(String[] args) {
        ObjetConstant cO = new ObjetConstant (100, 50);
        for (int i=0 ;i<10 ;i++) {
            tf[i] = new Actif(cO);
            tf[i].start();
        }
    }
}
```

# La programmation de la spécification de la synchronisation (3)

## Un exemple (suite)

```
class Actif extends Thread {
    private int numero;
    private Arbitre arbitre ;
    Actif(int nm,Arbitre a) { numero=nm; arbitre=a ; }

    public void run( ) {
        try {
            System.out.println(numero+" entre en phase 1");
            sleep((long)(Math.random()*1000));
            while( !arbitre.autorisation()) sleep(200);
            System.out.println(numero +" entre en phase 2");
            sleep((long)(Math.random()*1000));
            arbitre.liberation();
            System.out.println(numero +" se termine");
        } catch(InterruptedException e) { return; }
    }
}

class Arbitre {
    private int compteur=0 ;

    synchronized public boolean autorisation() {
        if (compteur ==5) return false;
        compteur++;
        return true;
    }
    synchronized public void liberation() { compteur-- ; }
}
```

```
class Test {
    public static void main(String[] args) {
        Arbitre arb=new Arbitre( ) ;
        Actif[ ] tf=new Actif[10] ;
        for (int i=0 ;i<10 ;i++) {
            tf[i] = new Actif(i,arb); tf[i].start();
        }
    }
}
```

```
synchronized TypeRetour synchro(TypArg0 arg0, ..... ) {
    <mettre à jour les variables d'état (point de synchro... atteint)
    if (condition blocage) return indication blocage;
    mettre à jour les variables d'état (point de synchro... franchi)
    return indication non blocage;
}
```

## La programmation de la spécification de la synchronisation (4)

### Un exemple à base d'attente passive

```
class Actif extends Thread {
    private int numero;
    private Arbitre arbitre ;
    Actif(int nm,Arbitre a) { numero=nm; arbitre=a; }

    public void run( ) {
        try {
            System.out.println(numero +" entre en phase 1");
            sleep((long)(Math.random()*1000));
            arbitre.autorisation();
            System.out.println(numero +" entre en phase 2");
            sleep((long)(Math.random()*1000));
            arbitre.liberation();
        } catch (InterruptedException e) { return; }
    }
}
```

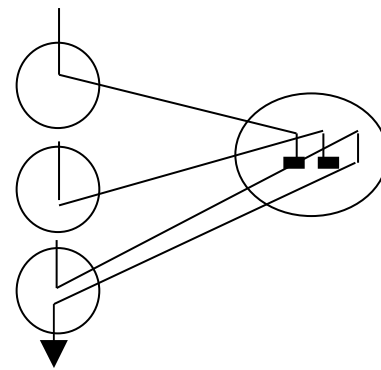
```
class Test {
    public static void main(String[] args) {
        Arbitre arb=new Arbitre( );
        Actif[] tf=new Actif[10];
        for (int i=0 ;i<10 ;i++) {
            tf[i] = new Actif(i,arb);
            tf[i].start();
        }
    }
}
```

Une solution qui ne marche **pas** (blocage de Arbitre)

```
class Arbitre {
    private int compteur=0;

    synchronized public void autorisation() {
        try {
            while (compteur==5) Thread.sleep(200) ;
            compteur++ ;
        } catch (InterruptedException e) { return ; }
    }

    synchronized public void liberation() { compteur-- ; }
}
```

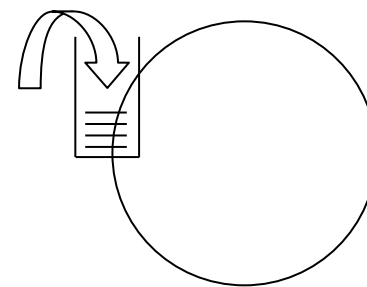


## les variables "Condition"

### La définition et l'API

File de threads bloqués dans l'objet

- un verrou associé à une file de threads
- wait() ⇔ passage à l'état "blocked" + entrée dans la file + libération du verrou
  - final void wait(long timeout) throws InterruptedException;
  - final void wait(long timeout,int nanos) throws InterruptedException;
  - final void wait( ) throws InterruptedException; ≡ wait(0);
- notify() ⇔ sortie de la file avec passage à l'état "runnable" de 1 thread
  - redémarrage après réacquisition du verrou
- notifyAll() ⇔ sortie de la file avec passage à l'état "runnable" de tous les threads
  - final void notify( );
  - final void notifyAll( );



- **ATTENTION : l'accès à la variable condition est NECESSAIREMENT synchronisé**
  - dans une méthode synchronisée
  - dans un bloc synchronized

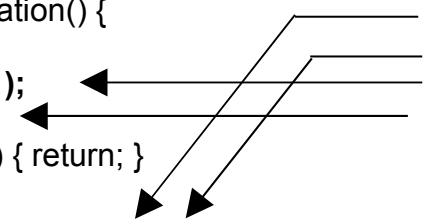
```
synchronized (obj) { ..... obj.wait() ; .....}
```

```
synchronized (obj) { ..... obj.notifyAll( ); ..... }
```

## La programmation de la spécification de la synchronisation (5)

### Un exemple simple d'utilisation du wait( ) et notify( )

```
class Arbitre {  
    private int compteur=0 ;  
  
    synchronized public void autorisation() {  
        try {  
            while (compteur==5) wait( );  
            compteur++ ;  
        } catch (InterruptedException e) { return; }  
    }  
  
    synchronized public void liberation() { compteur-- ; notifyAll( ) ; }  
}
```



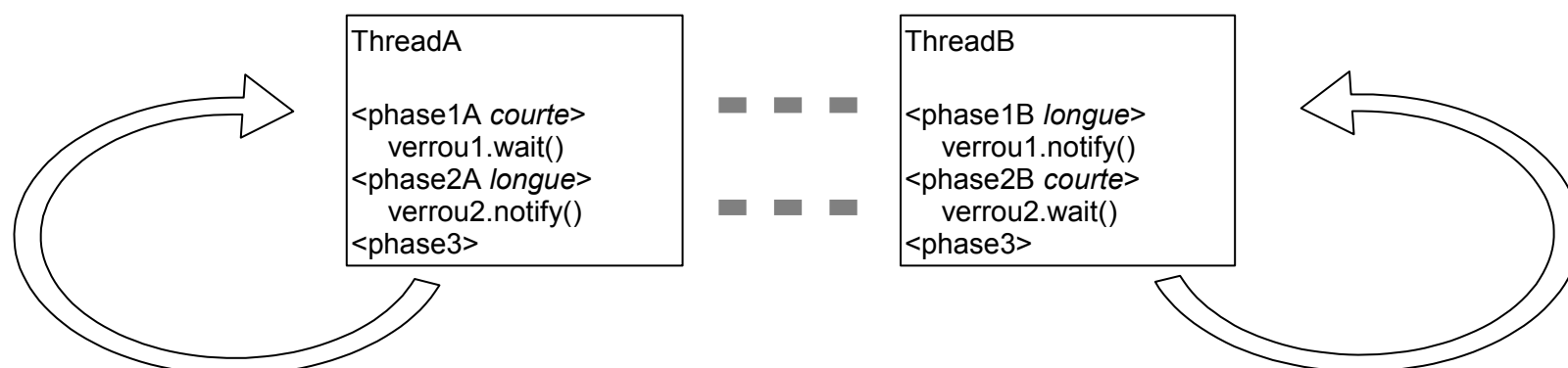
```
synchronized TypeRetour synchro(TypArg0 arg0, ..... ) {  
    <mettre à jour les variables d'état (point de synchro... atteint)  
    si utile notifyAll( ) ou notify( ) ;  
    while (condition blocage) wait( ) ;  
    mettre à jour les variables d'état (point de synchro... franchi)  
    si utile notifyAll( ) ou notify( ) ;  
    return result;  
}
```



## La synchronisation coopérative (1)

### Un exemple simple

- wait, notify et notifyAll (opérations ignorant le passé)
- le risque d'interblocage



### Une solution

- pas d'hypothèse implicite sur l'ordre des blocages/réveils
- utilisation de sémaphores

```
public class Semaphore {  
    private int compteur ;  
  
    public Semaphore(int cmp) { compteur = cmp ; }  
    public synchronized void P(int n) throws InterruptedException { while (compteur < n) wait(); compteur -= n; }  
    public synchronized void V(int n) { compteur+= n; if (compteur > 0) notifyAll(); }  
}
```

## La synchronisation coopérative (2)

### Un exemple plus complexe : un pool de threads simple ...

```
class PThread extends Thread {
    private Runnable runnable;
    private ThreadPoolManager poolManager;

    public PThread( ThreadPoolManager p) {
        poolManager = p;
    }

    public synchronized void load(Runnable r) {
        runnable = r;
    }

    public void run() {
        while(true) {
            synchronized(this) {
                try { wait(); } catch(InterruptedException ex) { .....}
            }

            if (runnable != null) runnable.run();
            poolManager.notifyEnd(this);
        }
    }
}
```

```
class ThreadPoolManager {
    private LinkedList pool;

    public ThreadPoolManager(int size) {
        pool = new LinkedList();
        for (int i=0; i<size; i++) {
            PThread t = new PThread(this);
            pool.add(t);
            t.start();
        }
    }

    public synchronized boolean execute(Runnable r) {
        if (pool.size() == 0) return false;
        PThread t = (PThread) pool.remove();
        t.load(r);
        synchronized (t) { t.notify(); }
        return true;
    }

    public synchronized void notifyEnd(PThread t) {
        pool.add(t);
    }
}
```

## La synchronisation coopérative (3)

### Un exemple plus complexe : un pool de threads simple (fin)

- les solutions correctes :
  - remplacer les **variables Condition** par des sémaphores
  - mettre dans un même bloc synchronized les méthodes notifyEnd() et le blocage du PThread

```
class PThread extends Thread {
    private Runnable runnable;
    private ThreadPoolManager poolManager;

    public PThread( ThreadPoolManager p) { poolManager = p; }

    public synchronized void load(Runnable r) { runnable = r; }

    public void run() {
        synchronized(this) {
            try { wait(); } catch(InterruptedException ex) { .... }
        }

        while(true) {
            if (runnable != null) runnable.run();
            synchronized(this) {
                poolManager.notifyEnd(this);
                try { wait(); } catch(InterruptedException ex) { .... }
            }
        }
    }
}
```

Mettre dans un même bloc synchronized  
les méthodes notifyEnd() et wait

```
class ThreadPoolManager {
    private LinkedList pool;

    public ThreadPoolManager(int size) {
        pool = new LinkedList();
        for (int i=0; i<size; i++) {
            PThread t = new PThread(this);
            pool.add(t);
            t.start();
        }
    }

    public synchronized boolean execute(Runnable r) {
        if (pool.size() == 0) return false;
        PThread t = (PThread) pool.remove();
        t.load(r);
        synchronized (t) { t.notify(); }
        return true;
    }

    public synchronized void notifyEnd(PThread t) {
        pool.add(t);
    }
}
```

# Les threads et les performances

## La création de threads

instruction	durée en ms	équivalent
new Date(index)	90-100	9-10
new ArrayList()	300-350	30-35
new Thread() sans start	7200-7400	720-740
new Thread() avec start	260000-270000	26000-27000

ordre de grandeur : une boucle de 1.000.000 d'itérations sur PC [Dell Inspiron 8100]

→ éviter les threads inutiles et utiliser un pool de threads

## La synchronisation des threads

- surcoût du synchronized négligeable
  - méthodes wait() et notifyAll() performantes
- MAIS .... blocage/redémarrage du thread coûteux

MAIS .... blocage/redémarrage du thread coûteux

instruction	durée en ms	équivalent
Blocage/réveil	11600-11750	1160-1175

→ limiter les risques de blocage inutiles

- réduire le temps passé dans les sections synchronisées (instructions synchronized vs méthodes synchronized)
- raffiner la logique de synchronisation (utiliser des verrous et des variables conditions différenciés)

## Les structures de données optimisées

- les nouvelles classes du package java.util.concurrent :
- ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet, ....., ConcurrentLinkedList

soulève rapidement des problèmes difficiles : interblocage, famine, ...

# Le raffinement de la logique de synchronisation (1)

## La spécialisation des variables Condition : un exemple simple

N Processus actifs comprenant 3 phases successives : Phase0, Phase1, Phase2

Contrainte : nbre max de processus en phase1 =5

Contrainte : nbre max de processus en phase2 =4

Variable d'état: compteur1 (nbre de processus en phase1)

Variable d'état: compteur2 (nbre de processus en phase2)

spécification du problème

Processus<i>

Phase0

• ps1

Phase1

• ps2

Phase2

• ps3

PreW :  
CB : compteur1 == 5  
PostW : compteur1++ ;

PreW : compteur1-- ;  
CB : compteur2 == 4  
PostW : compteur2++ ;

PreW : compteur2-- ;  
CB :  
PostW :

```
class Actif2 extends Thread {
    private int numero;
    private Arbitre arbitre ;

    Actif(int nm,Arbitre a) { numero=nm; arbitre=a ; }

    public void run( ) {
        .....
        arbitre.autorisation1() ;
        .....
        arbitre.passage1Vers2() ;
        .....
        arbitre.liberation2() ;
    }
}
```

## Le raffinement de la logique de synchronisation (2)

### La spécialisation des variables Condition : un exemple simple (suite)

```
class Arbitre {
    private int compteur1= 0;
    private int compteur2 = 0;

    public synchronized void autorisation1() {
        try {
            while (compteur1 == 5) wait();
            compteur1++;
        } catch (InterruptedException e) { return; }
    }

    public synchronized void passage1Vers2() {
        compteur1--; notifyAll();
        try {
            while (compteur2 == 4) wait();
            compteur2++;
        } catch (InterruptedException e) { return; }
    }

    public synchronized void liberation2() {
        compteur2--; notifyAll();
    }
}
```

```
class Arbitre {
    private int compteur1= 0;
    private int compteur2 = 0;
    private Object file1 = new Object();
    private Object file2 = new Object();

    public void autorisation1() {
        try {
            synchronized(file1) {
                while (compteur1 == 5) file1.wait();
                compteur1++;
            }
        } catch (InterruptedException e) { return; }
    }

    public void passage1Vers2() {
        synchronized(file1) { compteur1--; file1.notify(); }
        try {
            synchronized(file2) {
                while (compteur2 == 4) file2.wait();
                compteur2++;
            } catch (InterruptedException e) { return; }
        }
    }

    public void liberation2() {
        synchronized(file2) { compteur2--; file2.notify(); }
    }
}
```

## Le problème de l'interblocage sur l'accès aux ressources (1)

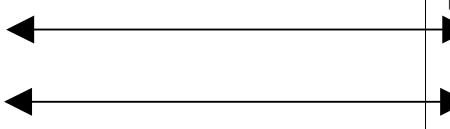
### Un exemple : les deadlocks liés à l'utilisation de plusieurs verrous

- grande sensibilité à l'ordre d'utilisation des primitives de blocage réveil
- problème des moniteurs imbriqués (utilisation de méthodes synchronisées dans une autre méthode synchronisée)

```
class DeadLock {  
    private Resource1 res1;  
    private Resource2 res2;  
    private Object verrou1 = new Object();  
    private Object verrou2 = new Object();
```

```
    public void deadlock1(int val) {  
        synchronized(verrou1) {  
            //utilisation de la ressource1  
            synchronized(verrou2) {  
                //utilisation des ressources 1 et 2  
            }  
        }  
    }  
}
```

```
    public void deadlock2(int val) {  
        synchronized(verrou2) {  
            //utilisation de la ressource2  
            synchronized(verrou1) {  
                //utilisation des ressources 1 et 2  
            }  
        }  
    }  
}
```



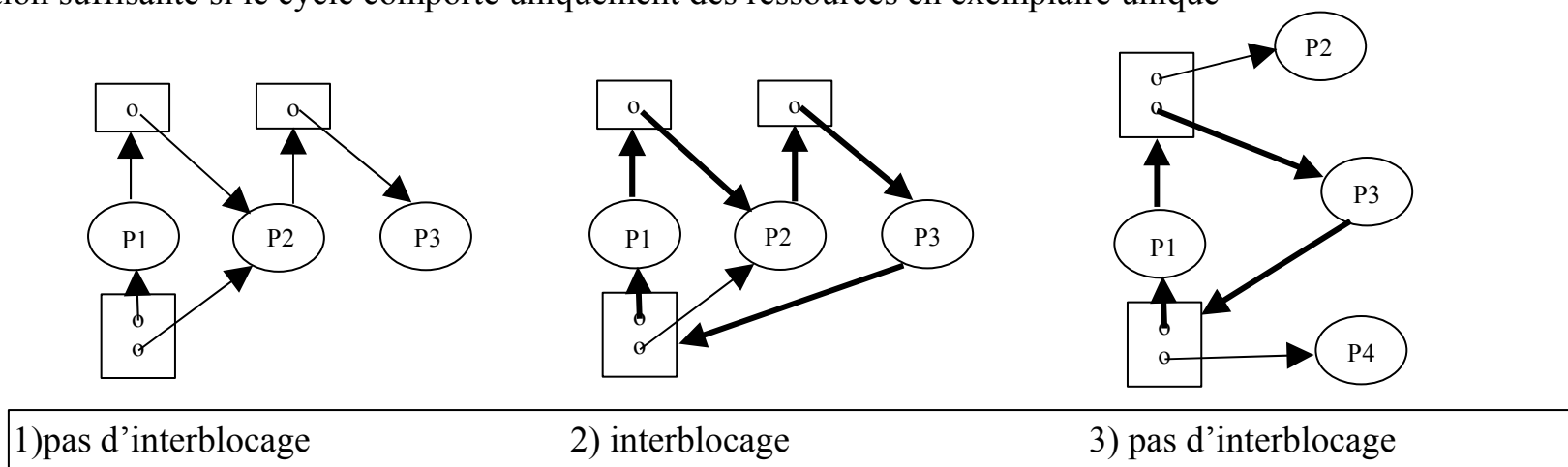
### solution

- voir suite

## le problème de l'interblocage sur l'accès aux ressources (2)

### Les conditions nécessaires pour un interblocage

- exclusion mutuelle (au moins une ressource non partageable)
  - hold and wait (au moins un processus détient une ressource non partageable et est bloqué)
  - non préemption sur l'usage des ressources (seul le processus détenteur d'une ressource non partageable peut la libérer)
  - attente circulaire
- 
- graphe d'allocation de ressources
  - condition nécessaire d'interblocage = cycle dans le graphe d'allocation de ressources
  - condition suffisante si le cycle comporte tous les threads détenteurs de ressources.
  - condition suffisante si le cycle comporte uniquement des ressources en exemplaire unique





## le problème de l'interblocage sur l'accès aux ressources (3)

### Les politiques de prévention des interblocages

- faire en sorte qu'au moins une des conditions ne soit pas réalisée → attention : sous utilisation des ressources
  - interdire l'usage en exclusion mutuelle
  - éviter le hold and wait
    - réclamer (et s'allouer) toutes les ressources simultanément
    - réclamer les ressources  
si non allocation → libérer de toutes les ressources acquises (puis réclamer l'ensemble des ressources)
- adopter une politique de préemption
  - réclamer les ressources  
si non allocation → les autres threads bloqués libèrent les ressources acquises
- empêcher l'attente circulaire
  - associer un numéro aux ressources  
réclamer les ressources par ordre croissant (libération préalable si nécessaire)

### Les techniques d'évitement des interblocages

- algorithmes couteux en performance conduisant à une sous utilisation des ressources (algorithme du banquier)

### Les algorithmes de détection-correction des interblocages

- libération des ressources (techniques de rollback-compensation)
- problème du choix des threads
- algorithmes couteux en performance (lancement périodique)

## Compléments sur la synchronisation : les nouveautés (1)

### Les primitives de synchronisation

- le sémaphore [introduit en java5]
  - compteur associé à une file de Threads
  - opérations ATOMIQUES sur le compteur (acquire(), release(int nb), tryAcquire(), tryAcquire(int nb), ..)
- 3 implémentations de l'interface Lock [introduites en java5]
  - modèle écrivain lecteur (plusieurs lecteurs en parallèle, écrivain en exclusion mutuelle)
  - classes ReadLock, WriteLock, ReentrantLock
  - opérations ATOMIQUES (lock(), unlock(), ..)
- la porte décroissante (CountDownLatch) [introduite en java5]
  - i.e. porte fermée tant que le compteur est  $> 0$
  - opérations **ATOMIQUES** (getCount(), await(), ..)
- la barrière cyclique (CyclicBarrier) [introduite en java5]
  - opérations **ATOMIQUES** (getParties(), await(), getNumberWaiting(), reset(), ..)
- nouvelle barrière (Phaser) [introduite en java7]
- la barrière de bas niveau (Fence) [introduite en java7]
- l'interface Condition [introduite en java5]
  - file de Threads
  - permet d'implémenter des politiques spécifiques
  - opérations ATOMIQUES sur la file

```
public interface Condition {  
    void await() throws InterruptedException;  
    void await(long a,TimeUnit u) throws InterruptedException;  
    void awaitUntil(Date d) throws InterruptedException;  
    void awaitUninterruptibly() throws InterruptedException;  
    void signal();  
    void signalAll();  
}
```

## Compléments sur la synchronisation : les nouveautés (2)

### Les primitives de synchronisation

- les types atomiques
  - **AtomicBoolean, AtomicLong, AtomicInteger, AtomicReference**

```
addAndGet(int nb);  
decrementAndGet();  
incrementAndGet();  
getAndSet(int val);  
get();  
set(int val)  
compareAndSet(int nb, int inc);  
.....
```

## Les ThreadLocal

### Les objectifs et le principe

- une variable **ThreadLocal** stocke une copie distincte de sa valeur pour chaque thread qui l'utilise
- il n'y a pas de support natif en java : implémentation via une classe **ThreadLocal**

```
public class ThreadLocal {  
    public Object get();  
    public void set(Object newValue);  
    public Object initialValue();  
}
```

### Quelques utilisations

- création d'un singleton par **Thread**
- création d'informations propres à un thread (log, debug, synchronisation par exemple)

```
public class DebugLogger {  
    private static class ThreadLocalList extends ThreadLocal {  
        public Object initialValue() { return new ArrayList(); }  
        public List getList() { return (List) super.get(); }  
    }  
    private ThreadLocalList list = new ThreadLocalList();  
    private static String[] stringArray = new String[0];  
    public void clear() { list.getList().clear(); }  
    public void put(String text) { list.getList().add(text); }  
    public String[] get() { return list.getList().toArray(stringArray); }  
}
```

extrait de l'article de B. Goetz :  
*Sometimes it's best not to share*  
*Exploiting ThreadLocal to enhance scalability*

- La classe **InheritableThreadLocal** pour la création d'objets partagés par un thread et ses descendants (utiles pour les objets non mutables).

## Compléments sur le raffinement de la logique de synchronisation (1)

### Un moniteur simple pour la gestion d'un buffer circulaire

- taille du buffer : BUFFER\_SIZE
- variables : putIndex, getIndex

```
class CircularBuffer {  
    private static final int BUFFER_SIZE = 10;  
    private Object[] buffer = new Object[BUFFER_SIZE];  
    private int putNb = 0, int getNb = 0;  
  
    public synchronized void put(Object item) {  
        try {  
            while (putNb - getNb >= BUFFER_SIZE) { wait(); }  
            buffer[putNb % BUFFER_SIZE] = item;  
            putNb++;  
            notifyAll();  
        } catch( InterruptedException ex) { ..... }  
    }  
  
    public synchronized Object get() {  
        try {  
            while (putNb - getNb <= 0) { wait(); }  
            Object _result = buffer[getNb % BUFFER_SIZE];  
            ++getNb;  
            notifyAll();  
            return _result;  
        } catch( InterruptedException ex) { ..... }  
    }  
}
```

solution peu efficace : concurrence d'accès au moniteur ( et les écrivains sont dans la même file que les lecteurs).

## Compléments sur le raffinement de la logique de synchronisation (2)

### Un moniteur pour la gestion optimisée d'un buffer circulaire

```
class CircularBuffer {  
    private static final int BUFFER_SIZE = 10;  
    private Object[] buffer = new Object[BUFFER_SIZE];  
    private int putNb = 0, getNb = 0;  
  
    private Object putVerrou = new Object();  
    private Object getVerrou = new Object();  
  
    private Semaphore putQueue = new Semaphore(BUFFER_SIZE);  
    private Semaphore getQueue = new Semaphore(0);
```

```
    public void put(Object item) throws InterruptedException {  
        putQueue.P(1);  
        synchronized(putVerrou) {  
            buffer[putIndex % BUFFER_SIZE] = item;  
            putIndex++;  
        }  
        getQueue.V(1);  
    }  
}
```

```
    public Object get() throws InterruptedException {  
        getQueue.P(1);  
        synchronized(getVerrou) {  
            item = buffer[getIndex % BUFFER_SIZE];  
            getIndex++;  
        }  
        putQueue.V(1);  
    }  
}
```

## Quelques autres problèmes liés à la gestion des activités

### La famine de certaines activités

- famine d'une activité = blocage non borné d'une activité (attention différent du problème de non respect d'une échéance))
- mauvaise utilisation des priorités, conspiration de threads, mauvaise gestion des ressources
- la résolution des problèmes de famine
  - introduction de politiques FIFO
  - introduction de temporisation, de compteurs permettant le redémarrage des activités bloquées
  - réévaluation dynamique des priorités

### Le problème du non-déterminisme

- non-déterminisme : plusieurs exécutions d'un même programme dans des conditions initiales similaires ne sont pas ordonnancés de la même façon et exhibent des traces « NON TEMPORELLES » différentes.
- problème quant à l'établissement de la correction du comportement dans des conditions données ....
- La transformation d'un THREAD en plusieurs THREADS introduit souvent de l'indéterminisme.
- Une solution générique et efficace : la programmation synchrone
  - modélisation événementielle : associer une tâche à l'occurrence d'un événement
  - discrétisation du temps : notion d'instant
  - Reactive Java et la librairie Junior

Compléments sur le problème de l’interblocage dans les problèmes de compétition

L’algorithme du banquier

- chaque thread doit déclarer pour chaque ressource, le nombre maximum d’exemplaires qu’il va utiliser.
- état « sûr » : le nombre de ressources libres permet à au moins un thread de réclamer le nombre maximum de ressources.
- Refuser une allocation de ressources conduisant à un état « non-sûr »

Exemple : (Peterson, Silbershatz) : 5 threads, 3 ressources A = 10 exemplaires, B = 5 exemplaires, C = 7 exemplaires

	Allocation			Maximum			Supplément			Disponible		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Etat sûr : il existe une suite d'allocation maximale ne conduisant à aucun interblocage : P1, P3, P4, P2, P0

Requête de P4 : A = 3, B = 3, C = 0

	Alloués			Maximum			Supplément			Disponible		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	0	0	2
P1	2	0	0	3	2	2	0	2	0			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	3	3	2	4	3	3	1	0	1			

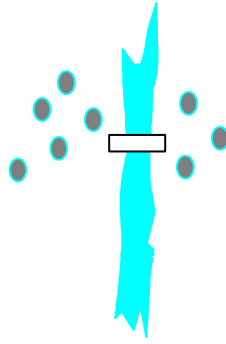
Etat non-sûr : il n'existe aucune suite d'allocation maximale ne conduisant à aucun intrerblocage



## Compléments sur la construction de moniteurs (1)

### La traversée d'un pont

- caractéristiques du pont
  - croisement de véhicule interdit
  - charge maximale (maxWeight)
- caractéristiques des véhicules
  - poids propre (weight)



- variables d'état : nombre de véhicules courant (indication du sens), charge
- 2 points de synchronisation

### Les points de synchronisation du problème (spécif d'un Véhicule [G → D])

```
Point de synchronisation1 :           //à l'entrée du pont
<PreW>
<CB> while (curWeight+weight >maxWeight || nbVehicule[D → G] >0 ) attendre( );
<PostW> curWeight+=weight; nbVehicule[G → D]++;
```

#### TRAVERSEE DU PONT

```
Point de synchronisation2 :           //à la sortie du pont
<PreW> curWeight-=weight; nbVehicule[G → D]--; réveiller(les vehicules bloques) ;
<CB>
<PostW>
```

## Compléments sur la construction de moniteurs (2)

### Le code du programme test et des véhicules ...

```
class Vehicle extends Thread implements PbConstants {
    private int weight;
    private Side location;
    private Bridge bridge ;

    public Vehicle(String nm,int w,Side l,Bridge bd) {
        super(nm);
        weight=w;
        location=l;
        bridge=bd ;
    }

    public Side getLocation() { return location; }
    public void setLocation(Side loc) { location=loc; }
    public int getWeight() { return weight; }

    public void run() {
        for (int i=0;i<5;i++) {
            try {
                sleep((int) (Math.random()*1000));
            } catch (InterruptedException e) { return; }
            bridge.acquireTicket(this); //synchro d'entrée
            try {
                sleep((int) (Math.random()*1000)); //traversée du pont
            } catch (InterruptedException e) { return; }
            bridge.releaseTicket(this); //synchro de sortie
        }
    }
}
```

```
public enum Side { LEFT, RIGHT }
```

```
class TestVehicle {
    private Bridge bridge = new Bridge(450);

    public static void main(String[] args) {
        new Vehicle("v0",250,Side.LEFT, bridge).start();
        new Vehicle("v1",200,Side.RIGHT, bridge).start();
        new Vehicle("v2",200,Side.LEFT, bridge).start();
        new Vehicle("v3",200,Side.LEFT, bridge).start();
        new Vehicle("v4",200,Side .RIGHT, bridge).start();
        new Vehicle("v5",300,Side .RIGHT, bridge).start();
    }
}
```

# Compléments sur la construction de moniteurs (3)

## Le code du pont (le moniteur)

```
class Bridge implements PbConstants {
    private int maxWeight;
    private int nbVehiculeDirect[] = {0,0};
    private int curWeight = 0;

    public Bridge(int mW) { maxWeight=mW; }

    synchronized void acquireTicket(Vehicle v) {
        int loc = v.getLocation().ordinal();
        int wght = v.getWeight();

        try {
            while (nbVehiculeDirect[1-loc] >0 || curWeight+wght>maxWeight) wait();
        } catch (InterruptedException e) { }

        curWeight += wght;
        nbVehiculeDirect[loc]++;
    }

    synchronized void releaseTicket(Vehicle v) {
        int loc = v.getLocation();

        nbVehiculeDirect[loc]--;
        v.setLocation(loc == Side.LEFT ? Side.RIGHT : Side.LEFT);
        curWeight -= v.getWeight();
        notifyAll();
    }
}
```

public enum Side { LEFT, RIGHT }

## Compléments sur la construction de moniteurs (4)

### Une variante du problème .... (spécif d'un Véhicule [G→D])

- généralisation des solutions délicates !!!

```
Point de synchronisation1 :           //à l'entrée du pont
<PreW> nbWaiting[G]++;
<CB> while (curWeight+weight >maxWeight || nbVehicule[D → G] >0 ||
      (nbWaiting[D]>maxWaiting) attendre( );
<PostW> curWeight+=weight; nbVehicule[G → D]++; nbWaiting[G]--;
```

#### TRAVERSEE DU PONT

```
Point de synchronisation2 :           //à la sortie du pont
<PreW> curWeight-=weight; nbVehicule[G → D]--; réveiller(les vehicules bloques) ;
<CB>
<PostW>
```

- attention : interblocage si nbWaiting[G] et nbWaiting[D] tous deux supérieurs à maxWaiting

## Compléments sur les threads : les groupes de threads, ...

### Le principe et les objectifs

- hiérarchie de threads : (arborescence)
- par défaut un thread appartient au groupe du thread qui l'a créé
- sécurité : limiter l'ensemble des threads sur lequel on peut agir
  - par ex. les threads de son groupe ou d'un sous-groupe

### La classe ThreadGroup

```
ThreadGroup(String name) ; //name obligatoirement non nul
ThreadGroup(ThreadGroup parent,String name) ;

final String getName( ) ;
final ThreadGroup getParent( ) ;

final void setDaemon(boolean daemon) ;
final boolean isDaemon( ) ;

final synchronized void setMaxPriority(int maxPri) ;
final int getMaxPriority( ) ;

final boolean parentOf(ThreadGroup g) ; //retourne vrai si g est le groupe parent ou le groupe lui même
final void checkAccess() throws SecurityException ; //exception générée si le thread n'a pas accès au groupe
final synchronized void destroy() throws InterruptedException //exception générée si le groupe ou un sous-groupe contient un thread

synchronized int activeCount();
int enumerate(Thread[] threadsInGroup,boolean recurse);
int enumerate(Thread[] threadsInGroup);
synchronized int activeGroupCount();
```