PIGOURIER Vincent

ANNEE SPECIALE 99/00

RAPPORT DE PROJET :

LES THREADS JAVA

Responsable : Serge Rouveyrol

I -INTRODUCTION A L'UTILISATION DES THREADS 3
1 - Géneralités et propriétés des threads 3
2 - La classe Thread 3
3 - Analogie et différences entre processus et processus léger 3
4 - Utilité des threads 4
II - L'API DES THREADS JAVA 6
1 - créer et démarrer un thread 6
2 - arrêt d'un thread 7
3 - Mise en sommeil d'un thread 8
4 - cycle de vie d'un thread, méthode isAlive() et join() 8
5 - Obtention d'informations sur les processus légers 8
III - SYNCHRONISATION DES THREADS JAVA 9
1 - les moniteurs java 9
2 - Méthodes wait(), notify(), notifyAll() 11
Cas de conflit d'accès concurrent lié au mécanisme d'attente et de notification 11
wait et sleep 11
3 - Illustration de la programmation concurrente et de la synchronisation des threads java : le problème des philosophes 12
Exposé du problème des philosophes 12
Solution avec des processus et des sémaphores 13
Solution avec les threads java 13
Commentaire sur l'implémentation des philosophes avec les threads java 16
IV - ORDONNANCEMENT DES THREADS 18
1 - panorama de l'ordonnancement 18
exemple d'ordonnancement avec des threads de priorités différentes 18
exemple d'ordonnancement avec des priorités égales: 20
Inversion de priorité et héritage de priorité 21
Ordonnancement du tourniquet 21
Modèle de gestion des threads 22
2 - Méthodes de l'API java concernant les priorités et l'ordonnancement des threads 22
3 - Modèle green-thread 23
Ordonnancement dans le modèle green-thread 23
4 - Threads natifs sous windows 23
Ordonnancement des threads natifs de windows 24
Conclusion sur l'ordonnancement des threads java sur plate-forme windows 24
5- Threads natifs sous Solaris 25
Ordonnancement des threads natifs solaris 25
Les LWPs de la machine virtuelle 26
V - EXEMPLE D'UTILISATION DES THREADS DANS LA PROGRAMMATION RESAU (EXEMPLE CLIENT/SERVEUR) 27
1 - Serveur 27
2 - Client 32

I -INTRODUCTION A L'UTILISATION DES THREADS

1 - Géneralités et propriétés des threads

Les threads java permettent d'effectuer des traitements dissociés les uns des autres au sein d'un même programme. Un thread où processus léger peut être vu comme un représentant d'un fil d'exécution à l'intérieur d'un programme. On entend par fil d'exécution, une section de code qui s'exécute indépendamment des autres fils de d'exécution d'un même programme.

Les deux propriétés fondamentales des threads sont :

- ? Ils s'exécutent dans le même espace d'adressage en mémoire. En Particulier dans le cadre d'un programme java, un objet créé par un thread est utilisable par les autres threads du programme.

- ? Ils s'exécutent de manière concurrente. En d'autres termes, ils sont en concurrence entre eux pour l'accès au processeur.

2 - La classe Thread

Lorsque l'on parle de processus léger, il y a trois notions à distinguer :

1. le fil d'exécution, c'est à dire la suite d'instruction du programme qui correspondent au processus léger

2. le contrôleur de ce processus

3. l'objet représentant les instructions du code à exécuter

En java le contrôleur des processus légers est un objet de la classe Thread. L'objet Thread n'est pas le processus léger mais ce qui permet de stocker les informations propres à ce processus léger. De plus ses méthodes permettent d'influencer son comportement comme nous le verrons par la suite.

3 - Analogie et différences entre processus et processus léger

On peut faire une analogie entre un processus léger et un processus. En effet un processus léger possède comme un processus des structures qui permettent de stocker son pointeur de pile, son PC, et les différents registres du CPU. Chaque processus léger possède également sa propre pile.

Cependant un processus est une entité du noyau du système d'exploitation. En tant que tel, son comportement ne peut être modifié que par des appels système. De même que la consultation d'informations propres à ce processus ne peut être réalisée que par appel à des primitives du

noyau. De plus un processus dispose d'un espace mémoire virtuel, et ses données ne sont pas accessibles aux autres processus.
Par opposition un thread java est une entité niveau utilisateur. La structure qui stocke les registres et les autres informations du processus léger est dans l'espace utilisateur et peut être accédée par des appels aux fonctions de la librairie des threads, qui sont de simples fonctions de niveau utilisateur.
Une autre différence essentielle est que les processus d'un même programme, comme nous l'avons dit, possèdent le même espace mémoire et font un partage du code. Précisons les données que les processus partagent et ne partagent pas.

- les variables locales des méthodes que les processus légers exécutent sont séparées pour chaque thread et sont complètement privées. Si deux threads exécutent la même méthode, ils reçoivent chacun un exemplaire séparé des variables locales.

- par contre les objets et leurs variables d'instance peuvent être partagés entre les différents threads.

4 - Utilité des threads

Les threads sont omniprésents dans les programmes java et il est presque impossible d'écrire des programmes java sans créer des threads. Beaucoup d'API java mettent déjà en œuvre des threads si bien qu'on utilise des threads parfois même sans le savoir. Voici les cas les plus courants de l'utilisation des threads:

- *Réalisations de tâches indépendantes :* un programme java est souvent utilisé pour réaliser des tâches indépendantes les unes des autres. Par exemple, un applet java peut réaliser deux animations indépendantes sur une page web, de même qu'un serveur de calcul peut effectuer des tâches distinctes pour des clients différents. Les threads java offrent au programmeur la possibilité de mettre en œuvre des programmes multitaches et c'est là leur utilité intrinsèque.

- *Entrées/sorties non bloquantes :* En java on reçoit des entrées utilisateurs en exécutant la méthode read() et en spécifiant le terminal de l'utilisateur (System.in). Quand le système exécute la méthode read(), le système se bloque en attente, tant que l'utilisateur n'a pas tapé au moins un caractère. Ce type d'entrée/sortie est bloquante.

  Souvent ce genre de comportement n'est pas celui que l'on souhaite. L'utilisation des threads permet de rendre les entrées/sorties asynchrones.

- *Alarmes et compteurs de temps* : les compteurs de temps sont implémentés en java à l'aide d'un thread séparé qui se met en sommeil pendant un laps de temps donné et notifie à son réveil les autres threads de l'expiration de ce laps de temps.

- *Algorithmes parallèlisables* : sur des machines virtuelles java qui sont capables d'utiliser simultanément plusieurs processeurs, les threads permettent de paralléliser l'exécution d'un programme qui comporte des séquences parallélisables. Lorsque les boucles qui sont paraléllisables nécessitent beaucoup de ressources CPU, l'usage de

trhreads s'exécutant en parallèle sur plusieurs processeurs permet des gains de temps considérables.

II - L'API DES THREADS JAVA

1 - créer et démarrer un thread

Au lancement d'un programme, la machine virtuelle possède un unique thread qui s'exécute. C'est le processus léger initial. Pour en créer un nouveau, on commence par créer un nouveau contrôleur (objet de la classe Thread ou d'une de ses sous classe).Lorsque le contrôleur est créé, le thread se trouve dans l'état initial, et n'exécute aucun code.

Pour faire passer le thread de l'état initial, à l'état prêt, il faut appeler la méthode start() de son contrôleur. Cet appel, débute l'exécution du processus léger dans la machine virtuelle et celle ci appelle la méthode run() sur l'objet représentant le code à exécuter. Le processus léger entre alors en concurrence pour l'accès au processeur. L'état prêt correspond à un processus léger qui a été démarré et qui peut éventuellement bénéficier du processeur. Le thread qui s'exécute réellement sur le processeur est appelé le *thread actif courant*.

L'objet représentant le code à exécuter doit être d'une classe qui implémente l'interface *Runnable*. Celle ci déclare une seule méthode, la méthode run(). L'interface Runnable peut être implementée par toutes classes dont les instances sont destinées à être exécutées par un thread. Les contrôleurs de la classe Thread en font évidemment partie et implémentent cette interface.

En résumé, lorsqu'un contrôleur est créé, le processus léger qu'il contrôle est démarré par l'appel à sa méthode start() qui crée effectivement le processus léger dans la machine virtuelle, laquelle appelle la méthode run() de l'objet code pour débuter l'exécution de la suite d'instructions définie dans celle ci.

Ils existe deux manières de spécifier la méthode run() à exécuter sur l'objet code:

? la première par héritage, consiste à masquer la méthode run() dans une sous classe de la classe Thread, qui par défaut est la méthode vide. Le contrôleur d'un processus léger est alors un objet de cette sous classe . Par ce procédé, le contrôleur du fil d'exécution et le code à exécuter sont encapsulés dans un même objet

Exemple :

```
class Mythread1 extends Thread {
        // redéfinition de la méthode run
        public void run() {
                System.out.println("hello monde");
                }
}

public class ExtendThread {
        Mythread1 thread1;

        public static void main(String args[]) {
                thread1 = new Mythread1(); // création d'un contrôleur
                thread1.start ; // démarrage du processus léger
        }
```

}

? la seconde consiste à implanter la méthode run() de l'interface Runnable dans une nouvelle classe représentant le code à exécuter. Le contrôleur du processus léger est alors obtenu en passant un objet de cette nouvelle classe en argument du constructeur de la classe Thread. Cette solution est un peu plus complexe à mettre en œuvre mais elle possède deux avantages. D'une part, la classe implémentant l'interface Runnable peut hériter d'une autre classe que la classe Thread, et d'autre part si un même objet implémantant l'interface Runnable est passé en argument à plusieurs constructeurs de la classe Thread , les champs de cet objet sont naturellement partagés par les processus légers. Cette caractéristique est difficile à mettre en œuvre avec la première méthode.

Exemple :

```
    // objet qui contrôle le code à exécuter par le rocessus léger
class Mythread2 impléments Runnable {
        // redéfinition de la méthode run
        public void run() {
                System.out.println("hello monde");
                }
}

public class ImplementRunnable {
        Mythread2  mythread2;

        public static void main(String args[]) {
                Thread thread2 = new Thread(mythread2); // création d'un contrôleur
                thread2.start ; // démarrage du processus léger
        }
}
```

2 - arrêt d'un thread

Un processus léger se termine lorsqu'il retourne de sa méthode run(). Le processus léger n'est plus présent dans la machine virtuelle. En revanche, son contrôleur est toujours accessible. Il ne permet pas de reprendre l'exécution du code contrôlé mais il permet de connaître l'état du processus léger.

Il existe dans l'API java une méthode *void stop()* qui permet d'arrêter brutalement un thread. Cependant cette méthode est d'usage dangereux car si le thread exécute une section critique lorsqu'il est stoppé, il peut laisser des données dans un état incohérent. La manière recommandée pour contrôler l'arrêt d'un thread est de faire usage d'une variable de condition qui permet de faire retourner le thread de sa méthode run().

Exemple :

```
class Mythread1 extends Thread {
        // redéfinition de la méthode run
        public void run() {
                while (ShouldRun){
                        System.out.println("hello monde");
                        }
                }
```

```
        }
```

3 - Mise en sommeil d'un thread

La méthode *static void sleep(int milliseconds)* de la classe Thread permet de mettre le thread courant actif en pause pendant le laps de temps spécifié. Le thread courant passe de l'état prêt à l'état bloqué.

Exemple du compteur de temps:

```
class Timer extends Thread {
        boolean shouldRun = true;

        public void run() {
                while( shouldRun ){
                        try {
                                System.out.printl("debout !!");
                                sleep(100);
                        }
                        catch(Exception e) {}
                }
        }
}
```

Ce processus léger écrit donc toutes les 100 ms "debout !!". Le mécanisme d'un tel thread est souvent utilisé pour effectuer des tâches à intervalles réguliers.

4 - cycle de vie d'un thread, méthode isAlive() et join()

Pour savoir si un thread est toujours présent dans la machine virtuelle, on peut via son contrôleur appeler la méthode *boolean isActive()*. Celle ci retourne true si le processus léger n'a pas terminé sa méthode run, sinon elle retourne false (processus en terminaison).

La méthode join() appelée dans la méthode run() d'un processus léger, via le contrôleur d'autres threads permet de bloquer ce processus léger jusqu'à la mort des autres threads (threads joints).

5 - Obtention d'informations sur les processus légers

La méthode *static Thread currentThread()* permet d'obtenir la référence du contrôleur du processus léger actif courant.

La méthode *static int enumerate(Thread[] treadArray)* obtient tous les objets contrôleurs d'un programme s'exécutant dans la machine virtuelle, et les place dans un le tableau threadArray. La valeur retournée est celle du nombre d'éléments de ce tableau.

La méthode *static activeCount()* permet de connaître le nombre d'objets contrôleurs présent dans la JVM.

III - SYNCHRONISATION DES THREADS JAVA

Lorsque les threads d'un programme manipulent des données partagées, les accès à ces ressources doivent être contrôlés pour garantir leur intégrité et éviter les risques de conflit lors d'accès simultanés à ces ressources.

1 - les moniteurs java

Java utilise les moniteurs pour contrôler les accès aux sections critiques d'un programme. Un moniteur est un verrou qui permet l'accès en exclusion mutuelle aux sections critiques du code. Lorsqu'un thread rentre dans la section critique il acquiert le moniteur et si d'autres threads tentent de rentrer dans cette section critique, ils se mettent en attente jusqu'à ce que le détenteur du moniteur le libère.

L'acquisition et la libération du moniteur n'est pas à la charge du programmeur java. Chaque objet java et chaque classe java possède un moniteur. Chaque fois qu'un objet est instancié, un moniteur unique est alloué pour cet objet. Ce moniteur est instancié si l'on en fait usage.
Pour acquérir un moniteur d'un objet,un thread doit exécuter une de ses méthodes déclarées **synchronized**. En déclarant une méthode synchronized, on garantit qu'un thread qui exécute celle-ci a acquis le moniteur. Ainsi tous les autres threads qui tentent d'exécuter la méthode sont bloqués et forcés d'attendre que le thread possédant le verrou ressorte de la méthode. La libération du moniteur se fait lorsque la méthode déclarée synchronized retourne.

Il existe deux types de moniteurs, les moniteurs d'objet et les moniteurs de classe:

- *Les moniteurs d'objet:* un moniteur d'objet unique est associé à chaque objet qui possède une ou plusieurs méthodes synchronisées. La déclaration de telles méthodes ce fait ainsi : *public* ***synchronized*** *void maMethode(){}.*
- *les moniteurs de classe:* ce type de moniteur est commun à tous les objets d'une même classe. Ce type de moniteur est instancié lorsque la classe possède une méthode de classe déclarée synchronisée de la manière suivante :

  public ***static synchronized*** *void maMethode(){}.*

Le code qui suit permet d'illustrer l'utilisation des moniteurs de classe et les moniteurs d'objets. La classe deuxThreads utilise l'interface Runnable. Elle permet de créer deux threads, thread_a et thread_b. Ces deux threads font appel à un objet de la classe Critique qui possède une méthode synchronized (méthode a) et une méthode static synchronized (méthode b). La méthode a fait donc appel à un moniteur d'objet propre à chaque objet et la méthode b fait appel à un moniteur de classe commun à tous les objets de la classe Critique.

```
Class Critique{

public synchronized void method_a(){
        System.out.println(Thread.currentThread().getName() + "est dans la methode a");
        try{
            Thread.sleep((int) Math.round(Math.random() * 5000));}
        catch( InterrupedException e){}
        System.out.println(Thread.currentThread().getName() + "sort de la methode a");
```

```
        }

public static synchronized void method_b(){
        System.out.println(Thread.currentThread().getName() + "est dans la methode b");
        try{
            Thread.sleep((int) Math.round(Math.random() * 5000));}
        catch( InterrupedException e){}
        System.out.println(Thread.currentThread().getName() + "sort de la methode b");
        }
}

public class deuxThreads implements runnable{
Critique critique;
public deuxThreads( String name){
        Thread thread_a, thread_b;
        critique = new Critique();
        thread_a  = new Thread(this, name + " : thread_a);
        thread_b  = new Thread(this, name + " : thread_b);
        thread_a.start();
        thread_b.start();
        }

public void run(){
        critique.methode_a;
        critique.methode_b;
        }

public staic void main(){
        deuxThread1 = new deuxThreads ("objet1");
        deuxthread2 = new deuxThreads("objet2");
        }
}
```

Le résultat de ce programme est :

```
objet1 : thread_a  entre dans la methode a
objet2 : thread_a entre dans la methode a
objet1 : thread_a sort de la methode a
objet1 : thread_a entre dans la méthode b
objet1 : thread_b entre dans la methode a
objet2 : thread_a sort de la methode_a
objet2 : thread_b entre dans la méthode a
objet1 : thread_a sort de la methode b
objet2 : thread_a entre dans la methode b
```

On constate que la méthode_a() est exécutée par les deux objets deuxThread en même temps. en effet chacun possède un objet Critique et ils acquièrent en entrant dans la méthode a un moniteur distinct propre à leur objet Critique.
Par contre la méthode_b() ne peut être exécuter en parallèle par les threads de deuxThread1 et de deuxThread2. En effet lorsque un thread entre dans cette méthode il acquiert le moniteur de classe commun à tous les objets de la classe Critique, ce qui bloque les autres threads tentant d'entrer dans cette méthode.

2 - Méthodes wait(), notify(), notifyAll()

Java ne supporte pas les variables de conditions propres aux moniteurs. Cependant le fonctionnement classique des moniteurs peut être émulé en faisant appel aux méthodes wait() et notify() de la classe Object.
Pour appeler ces deux méthodes, un thread doit obligatoirement disposer du moniteur de l'objet. Aussi il est impératif que l'appel à ces deux méthodes se fasse dans une méthode ou un bloc déclaré synchronized !

- **wait() :** le thread qui appelle la méthode wait() libère le moniteur dont il avait la possession et se bloque en attendant d'être notifié. Lorsqu'il est signalé, le thread reprend son exécution à l'instruction suivant le wait(), après avoir réacquit le moniteur.
- **notify() :** la méthode notify() permet de signaler son réveil à un thread qui avait appelé wait() sur le même objet. Il se peut que le thread notifié se bloque sur le moniteur si le thread qui a effectué le notify() le possède encore. Le thread notifié reprendra réellement son exécution lorsque celui qui l'a réveillé libérera le moniteur et qu'il en obtiendra la possession.

Il est à noter que si plusieurs threads sont bloqués sur un moniteur, lors d'un appel de notify() le thread qui sera débloqué et qui obtiendra le moniteur est déterminé par la machine virtuelle et l'on a pas le contrôle de ce choix.
Une alternative à ce problème est d'utiliser la méthode notifyAll() qui permet de notifier tous les threads bloqués sur le moniteur.

Cas de conflit d'accès concurrent liés au mécanisme d'attente et de notification

En général on utilise la méthode wait() dans un thread parce qu'une condition n'est pas vérifiée (généralement on teste une variable puis on appelle la méthode wait). Quand un autre thread valide la condition, il appelle la méthode notify(). Il peut se produire un conflit d'accès concurrent lorsque:

1 - le premier thread teste la condition et en conclut qu'il doit se mettre en attente
2 - le deuxième thread valide la condition
3 - le deuxième appelle notify() ; celle-ci n'est pas entendue puisque le premier thread n'est pas encore en attente
4 - le premier thread appelle la méthode wait().

Pour éviter ce genre de problème, il faut utiliser une condition qui se trouve dans une variable d'instance de l'objet verrouillé. Ainsi comme le thread ne libère le moniteur qu'après avoir exécuté le wait(), le second ne pourra obtenir le moniteur et effectuer notify(), qu'une fois le premier thread mis en attente.

wait et sleep

Dans la classe objet il existe une méthode surchargée de la méthode wait(). Elle reçoit en paramètre un timeout spécifié en milliseconde. Après ce délai, si le thread en attente n'a

toujours pas été notifié, il se réveille et reprend son exécution comme il le ferait s'il avait été notifié.

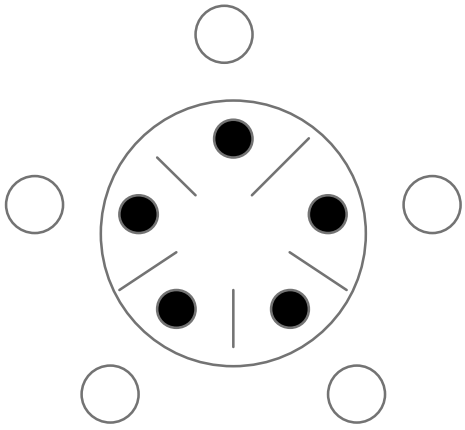
Cette méthode, utilisée sans notification, fonctionne donc comme le ferait un sleep. A la différence près qu'elle opère une libération et une acquisition de verrou. On peut donc faire appel à cette méthode pour mettre un thread en sommeil pour un laps de temps déterminé et libérer le moniteur pendant cet intervalle de temps.

3 - Illustration de la programmation concurrente et de la synchronisation des threads java : le problème des philosophes

Ce problème classique de programmation concurrente, se résout par l'usage de sémaphores. En java, les mécanismes de synchronisation, d'attente et de notification des threads permettent une implémentation originale de ce problème, sans faire appel aux sémaphores. Cet exemple permet d'illustrer les mécanismes présentés ci-dessus.

Exposé du problème des philosophes

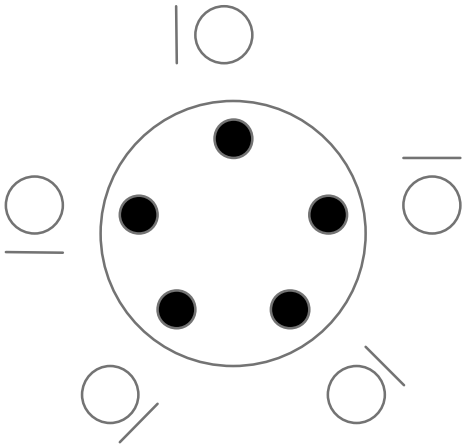
Cinq philosophes sont assis autours d'une table circulaire à l'ombre d'un grand chêne. Leur statut de philosophe leurs impose bien sûr de penser (ils sont payés pour ca après tout...), mais ils n'en demeurent pas moins des hommes, et ils doivent comme tout un chacun s'alimenter. Cependant suite à des restrictions budgétaires de la caisse des philosophes anonymes, on n'a disposé que cinq fourchettes sur la table. Comme les philosophes sont des hommes du monde, ils se refusent de manger tant qu'ils ne disposent pas de deux fourchettes.



Les philosophes ont donc trois états possibles : soit ils pensent, soit ils sont affamés et attendent de disposer des deux fourchettes, soit ils mangent (quand ils ont à disposition les deux fourchettes).

Ce problème permet d'illustrer les mécanismes des threads java qui permettent d'allouer des ressources partagées aux threads java sans pénurie de temps CPU ni interblocage.

La pénurie de temps CPU correspond à la situation où un thread n'est jamais actif, c'est à dire qu'il reste en attente de la ressource sans jamais y accéder. Dans notre cas cela correspond au décès d'un des philosophes par manque de nourriture !
L'interblocage peut se produire lorsque les philosophes prennent chacun la fourchette à leur droite et se mettent en attente de la fourchette gauche (détenue par leur voisin de gauche, bloqué en attendant la fourchette à leur gauche) comme l'illustre la figure ci-dessous.



Solution avec des processus et des sémaphores

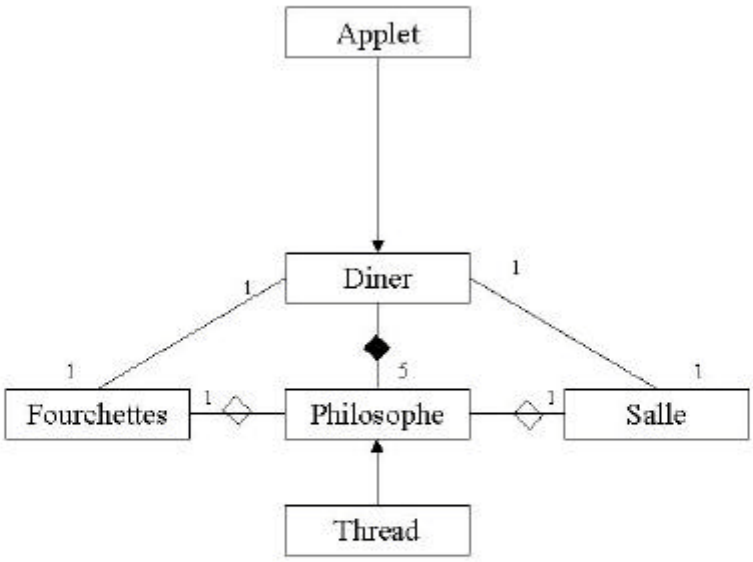
On limite à quatre le nombre de philosophes qui essayent de manger en même temps. Ceci garantit qu'il y ait toujours un philosophe parmi les quatre qui peut manger tandis que les trois autres détiennent une fourchette. Pour ce faire on utilise un sémaphore dont le compteur est initialisé à 4. Ceci permet d'une part de garantir qu'il n'y a pas d'interblocage et que tous les philosophes auront accès aux deux fourchettes (principe du tourniquet).
Le code d'un processus philosophe i est donc :

```
while(1){
        think( );
        wait( table );
        wiait( fourchette[ i ] );
        wait(fourchette[ ( i + 1 ) mod 5]);
        eat( );
        signal( fourchette[ ( i + 1 ) mod 5 ] );
        signal( fourchette[ i ] );
        signal( table );
        }
```

où table est un sémaphore initialisé à 4 et où fourchette est un tableau de 5 sémaphores initialisés à 1.

Solution avec les threads java

Le diagramme UML simplifié du programme est le suivant. Par souci de clarté la partie graphique de l'application n'est pas représentée.



Les processus philosophes sont remplacés par des threads java. L'objet Salle émule le sémaphore table et l'objet Fourchettes émule le tableau de sémaphores fourchette.

? La classe Philosophe est implémentée de la manière suivante :

```
Class Philosophe extends Thread {
        private int id ;  // identificateur
        private Diner d ;
        private Salle s;
        private Fourchettes f;

        // Constructeur
        public Philosophe( int id , Fourchettes f , Salle s, Diner d ) {
                this.id = id ;
                this.f = f ;
                this.s = s ;
                this.d = d ;
         }

        // code exécuter par le thread
        public void run() {
                while( true ) {
```

```
                    try {
                        sleep( d.tempsPense() ) ; }
                    catch( InterruptedException e) {}
                    s.entrer( id ) ;
                    f.prendre( id ) ;
                    try {
                        sleep( d.tempsMange() ) ; }
                    catch( InterruptedException e) {}
                    f.poser ( id ) ;
                    s.quitter( id ) ;
                    }
            }
}
```

? La classe Fourchettes est implémentée ainsi :

```
class Fourchettes {
        // tableau des fourchettes libres pour chaque philosophes
        private int num_fourchettes[] ;
        // nombre de philosophes mangeant
        private int philo_mangeant = 0 ;
        private boolean active[] ;

        // constructeur

        public Fourchettes(){
                num_fourchettes = new int [5] ;
                active = new bollean [5] ;
                for ( int i = 0 ; i < 5 ; i ++ ) {
                        num_fourchette[ i ] = 2 ; // au début toutes les fourchettes sont libres
                        active[ i ] = false ; // et aucun philosophe mange
                        }
                }

        // méthode prendre : un philosophe affamé doit disposer de deux fourchettes pour
        // pouvoir manger. Si une ou les deux fourchettes sont déjà prise le thread philosophe
        // se met en attente.

        public synchronized void prendre ( int id ) {
                while ( num_fourchette[i] != 2) {
                        try {
                            wait(); }
                        catch( InterruptedException e ) {}
                }
                // lorsque les fourchettes sont libres
                num_fourchettes[( i + 1 ) % 5 ] -- ;
                num_fourchettes[( i + 4 ) % 5 ] -- ;
                System.out.println("philosophe " + id + " eating ");
                philosophe_mangeant ++ ;
                active[ i ] = true ; }

        // methode poser : lorsqu'un philosophe a fini de manger, il repose les fourchettes
        // puis notifie tous les autres threads en attente de fourchettes

        public synchronized void poser ( int id ) {
                philo_mangeant -- ;
                active[ id ] = false ;
                System.out.println( " philosophe " + id " thinking" );
                num_fourchettes[( i + 1 ) % 5] ++;
```

```
                num_fourchettes[( i + 4 ) % 5] ++;
                notifyAll() ;
                }
        }
```

? Enfin la classe Salle est la suivante :

```
        class Salle {
                private int place_libre = 4 ;
                private int philo_en_attente = 0 ;

                // un philosophe avant de manger doit "entrer dans la salle" : si il y a déjà 4
philosophes
                // qui  tentent de manger, celui ci doit attendre que l'un des quatre autre "quitte la
        salle"

                public synchronized void entrer ( int id ) {
                        if ( place_libre <= 0 || philo_en_attente > 0 ) {
                                philo_en_attente ++ ;
                                try {
                                     wait() ;}
                                catch( InterruptedException e) {
                                philo_en_attente --;
                                }
                        place_libre -- ;
                }

                // lorsque un philosophe a fini de manger il "quitte la salle" et notifie un des threads
        qui
                // attend pour entrer

                public void synchronized quitter( int id ) {
                        place_libre ++ ;
                        notify() ;
                }
        }
```

Commentaire sur l'implémentation des philosophes avec les threads java

Examinons tout d'abord l'objet de la classe Salle. Cet objet possède deux méthodes déclarées synchronized.
La première, la méthode entrer() permet d'une part de s'assurer qu'il y ait au plus quatre philosophes qui essayent de manger en même temps. Comme nous l'avons dit cela permet d'éviter un interbloquage. Chaque philosophe, lorsqu'il décide de manger appelle cette méthode. Il acquiert le moniteur de l'objet Salle ce qui permet de régler l'accès concurrent au variables d'instances place_libre et philosophe_en_attente. Si place_libre est nul ou est négatif (ce qui signifie que quatre philosophes tentent déjà de manger), le thread qui possède le moniteur se bloque en appellant la méthode wait() de l'objet Salle après avoir incrémenté philosophe_en_attente. Ensuite il libère le moniteur.
La condition portant sur le nombre de philosophes en attente philosophes_en_attente est utile dans le cas où il y a quatre threads en train d'essayer de manger et un thread bloqué sur le wait() de la méthode entrer(). Supposons que cette condition ne soit pas imposée, et que l'un des quatre threads "présents dans la salle", appelle la méthode quitter(). La variable de

condition place_libre est incrémentée et passe de 0 à 1. Si juste après que le thread sortant ressorte de la méthode quitter(), u il la méthode entrer() et saisit le moniteur de l'objet Salle. Lorsqu'il teste la condition place_libre <= 0 , comme place_libre est égal à 1 il en déduit qu'il peut entrer dans la Salle. Ainsi il passe devant le thread bloqué qui attendait qu'une place se libère pour pouvoir tenter de manger. Cette situation critique peut, si elle se reproduit indéfiniment, aboutir au dépérissement d'un des threads (pénurie de temps CPU). Aussi il est plus sûr de rajouter cette variable de condition pour éviter ce phénomène.
La seconde méthode quitter() permet de libérer une place dans la salle (place_libre est incrémentée) et permet de réveiller l'un des éventuels threads bloqués sur la méthode wait() de la méthode entrer(). Notons que le thread que sera effectivement notifié est déterminé par la machine virtuelle et que l'on a pas de contrôle sur ce phénomène. Cela est lié à l'implémentation de cette méthode dans la machine virtuelle.

Examinons à présent les méthode de l'objet de la classe Fourchettes. La méthode prendre() permet de bloquer le philosophe i s'il n'a pas deux fourchettes libres en face de lui. Lorsque il est débloqué (i.e. les fourchettes devant lui sont libres), il reprend son exécution après avoir repris le moniteur. Il décrémente num_fourchette[(i+1) % 5] et num_fourchette[(i+4)%5], car les fourchettes qu'il prend ne sont plus libres pour ses voisins de gauche et de droite.
Lorsque le philosophe i a fini de manger il exécute la méthode poser() après avoir réacquit le moniteur. Il incrémente num_fourchette[(i+1) % 5] et num_fourchette[(i+4)%5] (car les fourchettes qu'il utilisait sont à présent libres) et fait un notifyAll(), ce qui réveille tous les philosophes bloqués sur le wait() de la méthode prendre(). Tous après avoir acquis chacun leur tour le moniteur,ils testeront s'ils ont deux fourchettes à disposition. Celui qui validera cette condition pourra manger (i.e. continuer son exécution et se mettre en sommeil pendant le temps de sa réflexion), les autres se bloqueront à nouveau sur le wait().

IV - ORDONNANCEMENT DES THREADS

Lorsque le nombre de threads est plus important que le nombre de processeurs, comme un processeur ne peut exécuter qu'un seul thread à la fois, les threads d'un programme ne sont pas tous actifs à un instant donné. L'ordonnancement a pour but de déterminer quel thread d'un programme va s'exécuter à un instant donné.
L'ordonnancement des threads java est une question difficile. En effet la spécification java n'exige pas un mode particulier d'ordonnancement pour les implémentations de la machine virtuelle. D'autre part certaines méthodes utilisées pour intervenir sur l'ordonnancement ont été placées en désuétude à partir de java2. Par conséquence on ne peut pas garantir absolument un ordre d'exécution des threads valables pour toutes les machines virtuelles.

1 - panorama de l'ordonnancement

Pour chaque thread de la machine virtuelle, il existe quatre états distincts:

- ? *initial :* un objet thread se trouve dans l'état initial entre le moment de l'appel de son constructeur et l'appel de sa méthode start.
- ? *prêt:* état par défaut d'un thread. Un thread est prêt lorsqu'il n'est pas dans un autre état.
- ? bloqué: un thread bloqué est un thread qui ne peut pas s'exécuter car il est en attente de l'arrivée spécifique d'un événement
- ? en cours de terminaison : un thread est en cours de terminaison lorsqu'il sort de sa méthode run() ou bien lorsque sa méthode stop() est appelée.

Il est fréquent que plusieurs threads soient simultanément à l'état prêt. Lorsque cela se produit un thread est élu pour devenir le thread actif courant. Les autres restent à l'état prêt et attendent une éventualité de pouvoir s'exécuter (i.e. devenir le thread courant actif).
La technologie Java implémente un ordonnanceur de type préemptif basé sur la priorité. Chaque thread reçoit une priorité qui ne peut être modifiée que par le programmeur. Le thread actif courant doit être le thread de plus haute priorité parmi les threads prêts. De plus lorsqu'un thread de plus haute priorité devient prêt, le thread de priorité inférieure qui était actif est interrompu par la JVM afin que le thread de plus haute priorité devienne le thread actif courant.

exemple d'ordonnancement avec des threads de priorités différentes

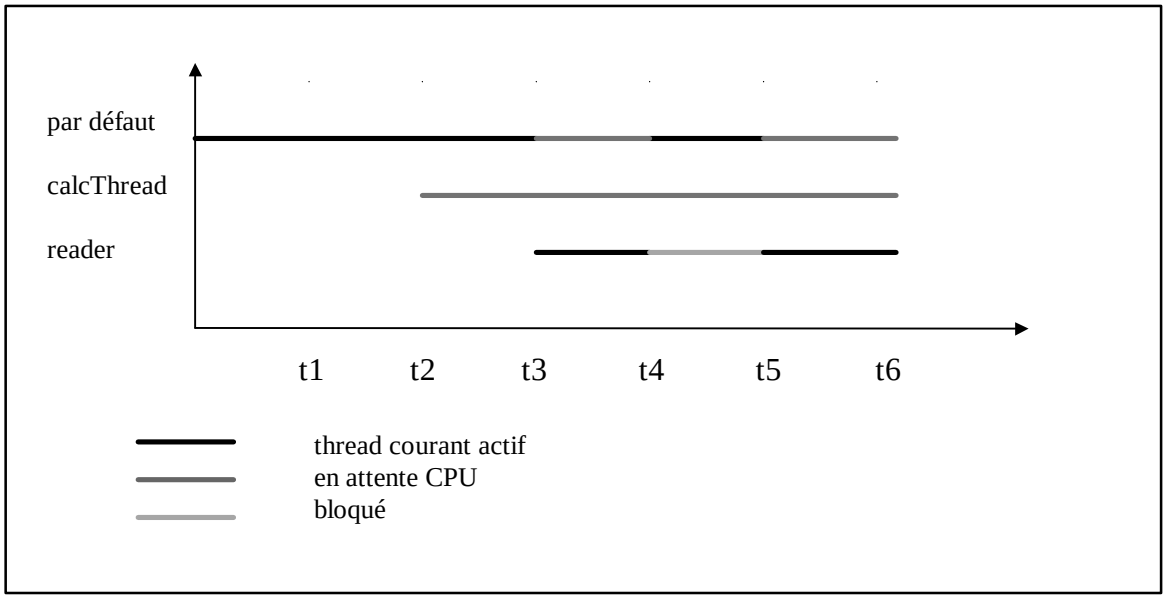
pour illustré les concepts utilisé par l'ordonnancement étudions l'exemple de code ci-dessous :

```
public class SchedulingExample implements Runnable{
        public static void main (String[] args){
                Thread calcThread = new Thread(this);
                calcThread.setPriority(4);
                calcThread.start();

                AsyncReaderSocket reader;
                reader = new AsynReaderSocket(new Socket(host,port);
                reader.setPriority(6);
```

```
                        reader.start();

                        DoDefault();
                        }
        public void run(){
                doCalc();
        }
}
```

Ce programme comporte trois threads. Le thread par défaut exécute la méthode main() : après avoir créé les autres threads il execute la méthode doDefault(). Le deuxième est le thread de calcul qui exécute la méthode doCalc(). Le troisième instancie la classe AsyncReaderSocket et lit sur un socket.

La figure ci-dessous montre la transition des threads par les différents états :



A l'instant t1 le thread par défaut exécute la méthode main(). Il a une priorité de 5 et il est le seul thread présent dans la machine virtuelle. C'est donc le thread actif courant.

A l'instant t2, il crée le thread calcThread, lui attribut une priorité de 4 et appelle sa méthode start(). Il y a donc deux threads prêts dans la JVM. Le thread par défaut reste le thread actif courant car il a une priorité plus forte et le calThread est en attente du processeur.

Le thread par défaut continu son exécution. Il crée le thread reader, lui attribut une priorité de 6 et appelle sa méthode start(). Le thread reader est alors prêt. Comme il a une priorité plus forte, il devient le thread actif courant au dépend du thread courant qui passe en attente du CPU.

Le thread reader lit des données sur un socket. Si il n'y a pas de données sur le socket il se met en attente en invoquant la méthode wait() et passe dans l'état bloqué (instant t4). Le thread par défaut reprend donc le CPU et reste le thread actif courant tant que le thread reader n'a pas de données à lire sur le socket. Lorsque cela se produit le thread reader redevient le thread courant actif (instant t5). Et ainsi de suite.

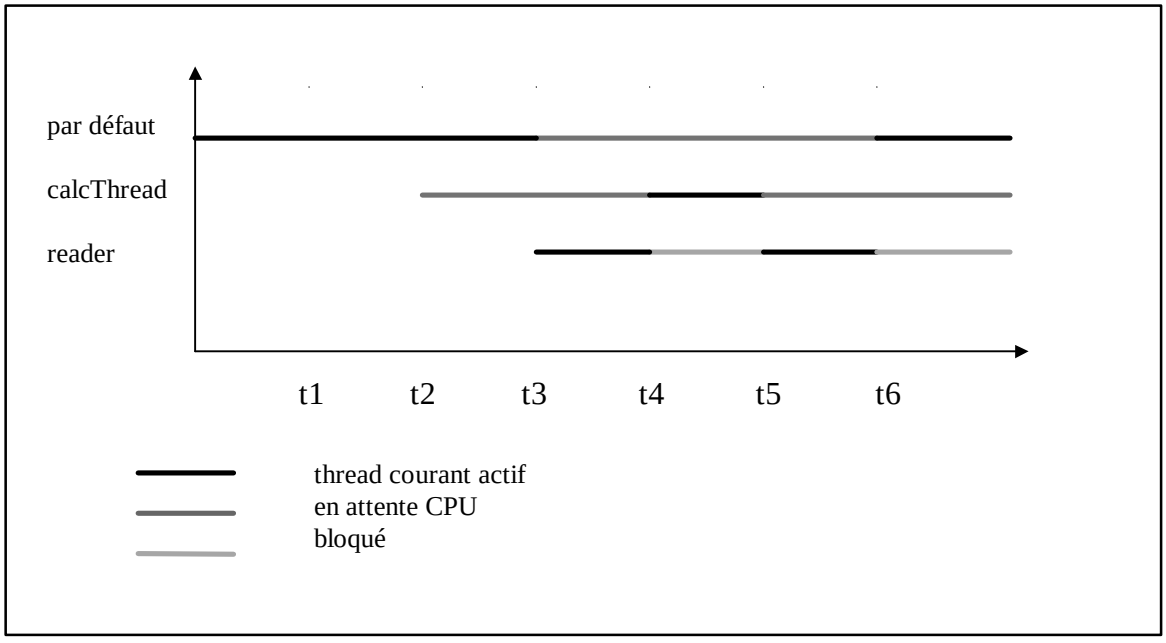
Remarquons que le thred calcThread ne prend jamais la main et se trouve dans une pénurie de temps CPU.

exemple d'ordonnancement avec des priorités égales:

Dans la plupart des programmes java, plusieurs threads ont des priorités égales. Nous allons nous placer à un niveau conceptuel pour décrire ce qui se passe dans la JVM, afin d'étudier de manière générale l'ordonnancement des threads et non de fournir un plan de l'implémentation spécifique d'une JVM particulière.

On peut imaginer que la machine virtuelle conserve la trace de tous les threads d'un programme au moyen de listes chaînées. Chaque thread se trouve dans une liste chaînée correspondant à son état. Il y a onze priorités donc on peut donc concevoir quatorze listes chaînées : une pour les threads dans l'état initial, une pour les threads bloqués, une pour les thread en cours de terminaison et une pour chaque niveau de priorité. Une liste correspondant à un niveau de priorité particulier contient que des threads de cette priorité à l'état prêt.

Reprenons l'exemple précèdent en donnant au thread calcThread la même priorité que le thread par défaut. La figure ci-dessous illustre le partage du CPU entre les trois threads:



Le thread par défaut et le thread calcThread ont la même priorité, l'ordonnancement se fait donc de manière différente lorsque le thread reader se bloque. Dans cet exemple nous nous intéressons à trois listes internes: la liste des threads de priorité 5, la liste des threads de priorité 6 et la liste des threads bloqués. Au démarrage de thread calcThread (t2) ces listes sont ainsi:

Prioroté 5 : thread par défaut -> calThread -> null
Priorité 6 : null
Bloque : null;

Comme le thread par défaut et en tête de liste des threads de plus haute priorité, il devient le thread actif courant et la liste de priorite 5 devient

Priorité 5 : calcThread -> thread par défaut -> null

A l'instant t3 le thread par défaut démarre le thread reader qui a un droit de préemption sur lui. Les listes internes ont maintenant l'aspect suivant:

Priorite 5 : calcThread -> thread par défaut -> null
Priorité 6 : reader -> null
Bloque : null

A l'instant t4 le thread reader se bloque en attente de données. La machine virtuelle cherche la liste de plus forte priorité non vide, soit celle de la priorité 5. Le premier thread de cette liste devient le thread actif courant. Ainsi le thread calcThread devient actif courant et se trouve déplacé en fin de liste:

Priorité 5 : thread par défaut -> calcThread -> null
Priorité 6 -> null
Bloqué : reader -> null

Et ainsi de suite. Chaque fois que le thread reader se bloque le thead par défaut et le calcThread changent de position dans la liste et deviennent en alternance le thread courant actif.

Dans cet exemple, nous avons posé comme principe qu'un thread qui devient actif courant est placé en fin de liste, ce qui permet une alternance des thread de même priorité. Bien que cela de loin l'implémentation la plus courante pour les machines virtuelles, cela n'est pas une exigence : il existe des système d'exploitation temps réel dans lesquels les threads interrompus ne sont pas réorganisés entre eux !

Inversion de priorité et héritage de priorité

L'inversion de priorité se produit lorsqu'un thread attend un verrou occupé par un thread de priorité inférieure. La priorité effective du thread de plus forte priorité est temporairement égale à celle du thread de plus faible priorité.

Cette situation anormale est souvent résolu par l'héritage de priorité. Avec ce procédé, si un thread détient un verrou demandé par un autre thread de priorité plus forte, sa propre priorité est relevée temporairement.; sa nouvelle priorité est celle du thread en attente du verrou.

L'objectif de l'héritage de priorité est de permettre au thread de plus forte priorité de devenir actif le plus tôt possible. L'héritage de priorité est une caractéristique courante des JVM mais elle est non obligatoire.

Ordonnancement du tourniquet

Le cas où les threads de même priorité ont droit de préemption les uns sur les autres et font un partage du temps est appelé ordonnancement du tourniquet. C'est un des aspects les plus

contestés l'ordonnancement des threads. Dans la spécification du langage java rien n'oblige à ce type d'ordonnancement. Beaucoup d'implémentation de la machine virtuelle y font appel à cause de leur lien avec le système d'exploitation . Mais beaucoup d'autres ne l'emploient pas notamment sur les plates-formes non-windows.

Cela introduit un niveau de comportement non déterministe. Sur une plate forme dotée de l'ordonnancement du tourniquet les threads de même priorité se passent périodiquement la main. Ce processus suit toujours le même principe : le thread qui devient actif est déplacé en queue de la file de priorité et un compteur de temps interne se déclenche périodiquement pour interrompre le thread actif courant et rendre actif le thread en tête de file.

Sur une plate forme ne possédant pas ce type d'ordonnancement, un thread actif courant n'est interrompu que lorsque un thread de plus forte priorité devient actif ou lorsque il se bloque.

Modèle de gestion des threads

L'absence ou la présence de l'héritage de priorité, ou bien de l'ordonnancement en tourniquet sont des exemples des différences qui peuvent exister entre différentes machines virtuelles. Ces différences existent car la spécification est peu prolixe pour l'ordonnancement et par conséquence les différentes implémentations ont tiré parti des caractéristiques des plates-formes hôtes.

On distingue deux types d'implémentations différentes :

- ? Le modèle green-thread : dans ce modèle l'ordonnancement est entièrement pris en charge par la machine virtuelle java. C'est le modèle qui s'écarte le moins de l'ordonnancement basé sur les priorités.
- ? le modèle des threads natifs : dans ce modèle c'est le système d'exploitation qui effectue l'ordonnancement des threads.

2 - Méthodes de l'API java concernant les priorités et l'ordonnancement des threads

La classe thread possède trois variables de classe permettant de définir l'étendue des priorités que le programmeur peut affecter aux threads.

- ? Thread.MIN_PRIORITY correspond à la priorité minimale.
- ? Thread.MAX_PRIORITY correspond à la priorité maximale
- ? Thread.NORM_PRIORITY correspond à la priorité par défaut dans l'interpréteur java

Chaque thread possède une valeur de priorité située sur cette échelle entre MIN_PRIORITY (valant 1) et MAX_PRIORITY (valant 10). Cependant tous les threads ne peuvent pas prendre n'importe quelle valeur de priorité. En effet ils appartiennent à un groupe de threads et chaque groupe possède une priorité maximale que les thread du groupe ne peuvent dépasser.

Deux méthodes permettent de manipuler les priorités :

- *void setPriority(int priority)* assigne une priorité au thread donné. Une exception est levée si la priorité affectée est en dehors de l'échelle de priorité (de 1 à 10).
- *int getPriority()* retourne le priorité d'un thread donné.

Enfin la méthode *yield()* permet de d'intervenir sur le thread actif courant. A l'appel de celle ci le thread courant rend la main, ce qui permet à la machine virtuelle de rendre actif un thread de même priorité. C'est une méthode de classe et elle n'agit que sur le thread courant.

3 - Modèle green-thread

Dans le modèle green-thread, c'est la machine virtuelle qui gère tous les détails de l'API des threads. Du point de vue du système, il n'y qu'un seul processus et qu'un seul thread.
Chaque thread est une abstraction interne à la machine virtuelle. Celle ci doit garder dans un objet Thread toutes les informations concernant le processus léger (pile, compteur ordinal ...). Quand la machine virtuelle choisit d'exécuter un nouveau thread, elle sauvegarde toutes ces informations du thread courant et les remplaces par celles du thread élu. Mais pour le système d'exploitation, la machine virtuelle exécute du code arbitraire . Le fait que ce code puisse émuler plusieurs threads distincts n'est pas connu en dehors de la machine virtuelle.

Ordonnancement dans le modèle green-thread

Les threads dans le modèle green-thread sont ordonnancés selon le principe de la plus forte priorité.
D'autre par il n'y a pas de partage de temps dans la plupart des implémentation.
L'ordonnancement est entièrement a la charge de la machine virtuelle et généralement elle change le thread courant uniquement lorsque un thread se bloque ou qu'un thread de plus forte priorité devient prêt.

4 - Threads natifs sous windows

Dans le modèle des threads natifs utilsés sous windows 95 et windows NT, le système d'exploitation a entièrement connaissance des différents threads utilisés par la machine virtuelle et il y a correspondance un-à-un entre les threads java et des threads du système d'exploitation. Par conséquence l'ordonnancement des threads java est soumis à l'ordonnancement sous-jacent des threads du système d'exploitation.
Ce modèle est habituellement simple pour l'ordonnancement car le système ne fait pas de distinction entre thread et processus pour l'ordonnancement. Chaque thread est ordonnancé à même titre qu'un processus..
Le système d'exploitation ayant connaissance des threads, les threads natifs de windows sont plutôt lourds et si le nombre de ces derniers est important, la réactivité du système peut être ralentie.

Ordonnancement des threads natifs de windows

Dans le modèle des threads natifs windows, le système effectue l'ordonnancement des threads java comme il le fait pour un processus. Cela signifie que les threads sont ordonnancés suivant un mécanisme préemptif basé sur les priorités. Cependant une certaine complexité se trouve ajoutée au modèle générique décrit précédemment**.**

Tout d'abord, windows ne reconnaît que sept niveaux de priorités. Il en résulte que les onze niveaux de priorités programmables des threads java doivent correspondre au sept niveaux du sytème. Voici la table des correspondances :

priorité java	**priorité windows**
0	THREAD_PRIORITY_IDLE
1	THREAD_PRIORITY_LOWEST
2	THREAD_PRIORITY_LOWEST
3	THREAD_PRIORITY_BELOW_NORMAL
4	THREAD_PRIORITY_BELOW_NORMAL
5	THREAD_PRIORITY_NORMAL
6	THREAD_PRIORITY_ABOVE_NORMAL
7	THREAD_PRIORITY_ABOVE_NORMAL
8	THREAD_PRIORITY_HIGHEST
9	THREAD_PRIORITY_HIGHEST
10	THREAD_PRIORITY_TIME_CRITICAL

La deuxième complication provient du fait que la priorité programmable n'est qu'une partie de l'information utilisée par le système pour déterminer la priorité absolue d'un thread java. D'autre aspects peuvent intervenir dans la détermination de la priorité d'un thread java:

- ? les threads natifs windows sont sujet à l'héritage de priorité
- ? la priorité effective d'un thread est fonction de la priorité programmée (de l'inverse plus exactement) moins une valeur indiquant si un thread a déjà été actif récemment. Cette valeur se trouve constamment modifiée : plus le temps passe plus elle devient proche de zéro. Cela introduit un partage du temps entre les thread de même priorité et aboutit à un système d'ordonnancement en tourniquet.
- ? Un thread qui n'a pas été actif depuis longtemps bénéficie d'une prime temporaire de priorité. La valeur de cette prime de priorité diminue dans le temps à mesure que le thread acquiert une éventualité de devenir actif. Cela protège les threads d'une pénurie absolue tout en respectant une préférence pour les threads de plus haute priorité.

Conclusion sur l'ordonnancement des threads java sur plate-forme windows

En définitive il est très difficile d'assurer explicitement l'ordre d'exécution des threads sur les plates-formes windows. Cependant, comme le système d'exploitation garantit qu'il n'y a pas de pénurie de temps CPU et que les threads de priorités égales font le partage du temps, cela ne pose généralement pas de problème.

5- Threads natifs sous Solaris

Les threads natifs solaris utilisent un modèle de programmation à deux niveaux . d'une part il y a les threads de niveau utilisateur qui ne sont pas connus du système d'exploitation et qui sont ordonnancés de la même manière que dans le modèle green-thread. D'autre part il y a les threads de niveau système, connus sous le nom de processus legers (lightweight processes où LPWs) qui sont ordonnancés de manière très semblable au threads natifs windows. L'interaction entre ces deux niveaux donne une grande souplesse aux threads natifs solaris.

Ordonnancement des threads natifs solaris

L'ordonnancement des threads natifs solaris est relativement complexe mais il fait appel aux mécanismes présentés en début de chapitre.
Du point de vue d'un LWP particulier, l'ordonnancement applique le modèle green-thread. A tout moment un LWP exécute un thread de plus haute priorité. Un LWP ne fournit pas de partage de temps pour les threads prêts. Tout comme dans le modèle green-thread , un fois que le LWP a sélectionné un thread actif, le thread reste actifs jusqu'à ce qu'il se bloque ou qu'un thread de plus forte priorité devienne actif. Il existe une correspondance un-à-un entre les threads que peut exécuter un LWP et les threads java. Par conséquence pour une machine virtuelle java réduite à un LWP, l'ordonnancement se passe exactement comme pour le modèle green-thread.

Les LWP ont une priorité propre qui n'est pas connue du programmeur java et n'est pas influencé par la priorité des threads que le LWP exécute. Par conséquence les LWP de la machine virtuelle ont fondamentalement la même priorité et cette priorité est sujette à des ajustements périodiques selon l'activité du LWP. En d'autres termes les LWPs font le partage du temps.

Supposons une machine virtuelle ayant reçu deux LWPs du système et qui exécute deux threads de priorité 5. Quand le premier LWP commence, il prend de façon arbitraire un des deux threads et l'exécute. Lorsque sa tranche de temps s'achève le second LWP prend la main et exécute le thread restant.. Lorsque sa tranche de temps se termine le premier LWP redevient actif et continu l'exécution du premier thread et ainsi de suite. Les deux threads reçoivent une tranche de temps parce que les LWPs en reçoivent une du système.

Supposons à présent qu'il y ait trois threads de priorité égale et seulement deux LWPs. Les deux premiers threads s'exécutent en partage de temps et le troisième ne devient pas actif du tout tant l'un des deux autre ne s'est pas bloqué ou achevé.

Si il y a deux threads de priorité 4 et 5 et toujours deux LWPs ces threads vont s'exécuter en partage du temps. Le premier LWP sélectionne le thread de priorité 5 et l'exécute pendant la tranche de temps qui lui est impartie. Lorsque celle ci s'achève le second LWP sélectionne le thread de priorité 4 et l'exécute. En règle générale est que n LWPs d'une machine virtuelle exécutent en partage de temps les n threads de plus hautes priorités même s'ils ne sont pas de priorités égales.

Les LWPs de la machine virtuelle

Le nombre de LWPs de la machine virtuelle n'est pas fixe et varie selon les règles suivantes:

? La machine virtuelle commence avec un LWP. A mesure qu'elle crée des threads ceux-ci sont exécutés par ce LWP. Ce LWP en effectue l'ordonnancement en respectant la priorité et en ne faisant pas le partage du temps.

? Quand un thread fait un appel système, il devient temporairement lié au LWP (i.e. ce thread ne peut pas subir de droit de préemption).

? Si l'appel système retourne immédiatement le thread n'est plus lié et peut à nouveau subir le droit de préemption. En d'autre terme le LWP exécutera le thread de plus haute priorité.

? Si le thread par contre bloque le LWP auquel il est lié bloque également. Si il y'a d'autres threads prêt dans la machine virtuelle alors le système va attribué un Lwp pour les exécuter.

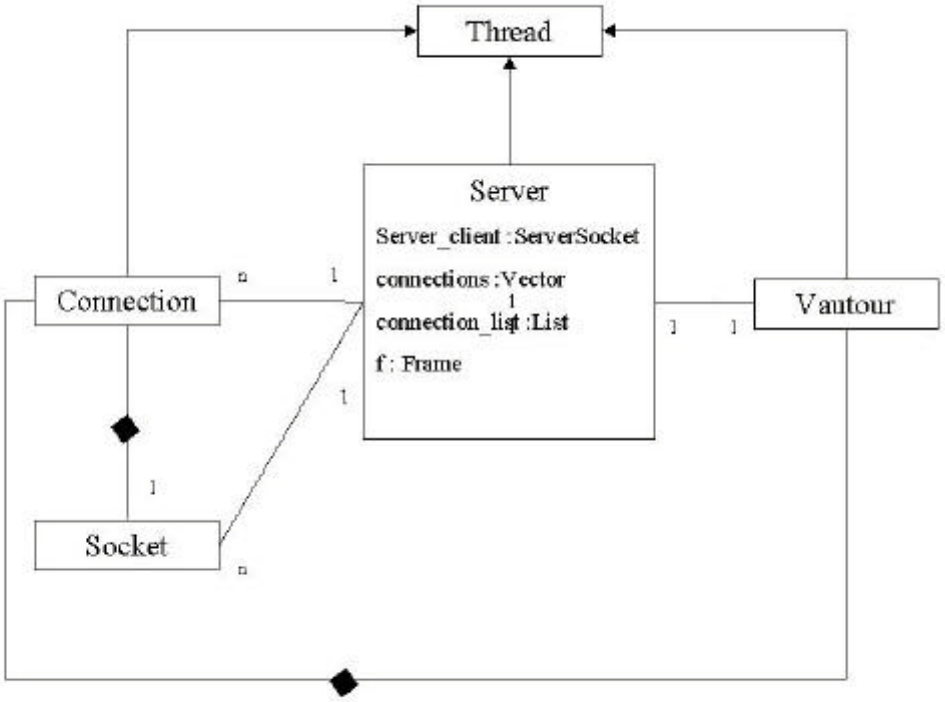
La machine virtuelle ne crée jamais de LWP; cette création est gérée par le système d'exploitation. Le nombre de LWPs de la machine virtuelle est égal au nombre de threads java faisant des appels système bloqués simultanément.

V - EXEMPLE D'UTILISATION DES THREADS DANS LA PROGRAMMATION RESAU (EXEMPLE CLIENT/SERVEUR)

Cet exemple de client-serveur est très simple. L'application serveur ouvre une connexion à chaque client qui en fait la demande, affiche dans un frame la liste des connexions. Le service offert par cette connexion est de renvoyer en majuscule tous les messages que lui envoie un client.

1 - Serveur

Diagramme UML et fonctionnement



Le programme serveur exécute plusieurs tâches indépendantes et fait appel à plusieurs threads dédiés à chacune de ces tâches:

Le premier thread, contrôlé par l'objet Serveur, effectue une boucle infinie dont le cycle est le suivant :

1. attente d'une connexion (methode accept() de la variable d'instance ServerSocket)
2. création d'une instance de la classe Connection à partir du socket renvoyé par accept()
3. tente d'acquérir le verrou de l'objet connections

4. après obtention du verrou, ajout de la nouvelle connexion dans la variable d'instance connections (classe Vecteur)
5. mise à jour de la variable d'instance connection_list (classe List) qui permet l'affichage des connections.

A chaque connexion un nouveau thread, contrôlé par un objet Connection est créé. Cette connexion réalise une boucle infinie dont le cycle est le suivant :

1. attente bloquante d'un message utilisateur.
2. réalisation du service

Après une demande de déconnexion, le processus sort de la boucle infinie, et notifie le thread vautour. Le thread vautour réalise lui aussi une boucle infinie dont voici le cycle :

1. se boque en effectuant un wait(5000).
2. s'il a été notifié ou si les 5 secondes sont écoulées, tente de prendre le moniteur de l'objet connections du Serveur
3. lorsqu'il possède le verrou, il parcourt tous les éléments de ce vecteur. Les éléments de ce vecteur sont les contrôleurs des threads Connection. Si un de ces threads est terminé, le vautour l'enlève du vecteur, et de la connection_list.

code

? Classe Serveur.java

```
import java.net.*;
import java.io.*;
import java.awt.*;
import java.util.Vector;

public class Server extends Thread{

  public final static int DEFAULT_PORT = 6789;
  protected int port;
  protected ServerSocket listen_socket;
  protected ThreadGroup threadgroup;
  protected List connection_list;
  protected Vector connections;
  protected Vautour vautour;

    // message d'erreur quand une exception est levee

    public static void fail(Exception e, String msg){
     System.err.println(msg + " : " + e);
     System.exit(1);
    }

    // cree un ServerSocket pour ecouter les connexions et
    // demarre le thread

    public Server(int port){
```

```
    // creation du server en lui donnant un nom
       super("Server");
           if (port == 0) port = DEFAULT_PORT;
           this.port = port;
           try{ listen_socket = new ServerSocket(port); }
           catch(IOException e) { fail(e,"Exception creating server socket");}
    // cretaion d'un groupe de Threads pour les connextions
           threadgroup = new ThreadGroup("Server Connections");
    // creation d'une fenetre affichant les connexions
           Frame f = new Frame("etat server");
           connection_list = new List();
           f.setResizable (true) ;
           f.setLayout (new BorderLayout ()) ;
           f.add("Center",connection_list);
           f.show();
    // initialise un vecteur pour stocker les connextions
           connections = new Vector();
    // cree le thread vautour
           vautour = new Vautour(this);
    // se demare lui meme
           this.start();
   }

   //methode run du thread Server: boucle infinie ecoute et accepte des
   //connections des clients. Pour chaque connection il cree un objet
   //Connection pour gere la communication a travers la nouvelle socket
   //Quant une nouvelle connection est cree on l'ajoute dans le vecteur
   //et on l'affiche dans la list.
   //Un verrou est mis en place sur l'Objet Vector. le vautour faisant de
   //meme, on peut ainsi gerer le partage de cet objet

   public void run(){
           try{
           while(true){
              Socket client_socket = listen_socket.accept();
                     Connection c = new Connection(client_socket, threadgroup,
3,vautour);
                     // gestion des acces concurrents
                     synchronized(connections){
                        connections.addElement(c);
                        connection_list.addItem(c.toString());
                     }
              }
           }
           catch(IOException e){fail(e,"Exeption while listening for connections");
           }
   }

   // Demarre le serveur sur un port optionel
public static void main(String[] args){
   int port = 0;
   if(args.length == 1) {
           try{
              port = Integer.parseInt(args[0]);}
      catch(NumberFormatException e) {port = 0;}
   }
   new Server(port);
  }
}
```

? classe Connection.java

```
import java.net.*;
import java.io.*;
import java.util.*;

// thread gerant toute communication avec un client
// previent le vautour quand une copnnection tombe

class Connection extends Thread{
    static int connection_number = 0;
    protected Socket client;
    protected Vautour vautour;
    protected DataInputStream in;
    protected PrintStream out;

 // initialise les flots et demarre le thread
    public Connection(Socket client_socket, ThreadGroup threadgroup, int priority, Vautour vautour){
            // donne un groupe, un nom et une priorite au thread
            super(threadgroup,"Connection-"+ connection_number ++);
        this.setPriority(priority);
            // sauve les autres parametres
            client = client_socket;
            this.vautour = vautour;
            // cree les flots
            try{
                in = new DataInputStream(client.getInputStream());
                out = new PrintStream(client.getOutputStream());
            }
            catch(IOException e){
                try{ client.close(); } catch(IOException e2){}
                System.err.println("Exception getting socketStream: " +e);
                return;
            }
            // demarre le thread
            this.start();
    }
    // fournit le service
    public void run(){
            String line;
            // message de bienvenu
            out.println("welcome a tous");
            try{
                for(;;){
                        line = in.readLine();
                        if(line == null) break;
                        out.println(line.toUpperCase());
                }
            }
            catch(IOException e) {}
            // quand on a finit on s'assure de fermer le socket et on previent
            // le vautour
            finally{
                try {client.close() ;} catch(IOException e2){}
                synchronized(vautour){ vautour.notify();}
            }
    }

    // methode retournant la representation sous forme de chaine de
    // l'objet connection
```

```
    public String toString(){
        return this.getName() +" connected to" + client.getInetAddress().getHostName() + " : " +
client.getPort();
    }
}
```

? classe Vautour.java

```
import java.util.*;
import java.awt.*;
import java.net.*;


// le vautour attend qu'un thread soit mourrant et nettoie la liste
// des threads et la liste graphique

class Vautour extends Thread{
    protected Server server;

    protected Vautour(Server s){
     super(s.threadgroup,"connection Vulture");
     this.server = s;
     this.start();
    }

public synchronized void run(){

for(;;){
    try{ this.wait(5000);} catch(InterruptedException e){}
    // evite les acces simultanes
    synchronized(server.connections){
        // boucles sur les connection
        for(int i=0; i< server.connections.size();i++){
            Connection c;
            c= (Connection)server.connections.elementAt(i);
            if(!c.isAlive()){
                server.connections.removeElementAt(i);
                server.connection_list.delItem(i);
                i--;
            }
        }
     }
    }
  }
}
```

2 - Client

Un client demande une connexion au server, puis lance deux threads. Un est chargé d'envoyer les messages rentrés sur l'entrée standards au serveur, et l'autre est chargé de récupérer les messages renvoyés par le serveur et de les afficher. L'usage de ces deux threads permet des lectures et écritures asynchrones.

code

? Client.java

```
import java.io.*;
import java.net.*;

public class Client{
   public static final int DEFAULT_PORT = 6789;
   Socket socket;
   Thread reader,writer;

   public Client(String host, int port){
         try{
            socket = new Socket(host,port);
            reader = new Reader(this);
            writer = new Writer(this);
            reader.setPriority(6);
            writer.setPriority(5);
            reader.start();
            writer.start();
         }
         catch(IOException e){System.err.println(e);}
   }

   public static void usage(){
         System.out.println("Usage : java Client <hostname> [<port>]");
         System.exit(0);
     }

   public static void main(String[] args){
         int port = DEFAULT_PORT;
         Socket s = null;
         if ((args.length !=1) &&(args.length !=2)) usage();
         if (args.length == 1) port = DEFAULT_PORT;
         else{
            try{port = Integer.parseInt(args[1]);}
            catch(NumberFormatException e){usage();}
         }
      new Client(args[0],port);
   }
}
```

? Reader.java

```
import java.io.*;
import java.net.*;

class Reader extends Thread{
  Client client;

    public Reader(Client c){
        super("Client Reader");
        this.client = c;
    }

    public void run(){
        DataInputStream in = null;
        String line;
        try {
            in = new DataInputStream(client.socket.getInputStream());
            while (true){
                line = in.readLine();
                if(line == null){
                    System.out.println("server closed connection");
                    break;
                }
                System.out.println(line);
            }
        }
        catch(IOException e) {System.out.println("Reader : "+e);}
        finally {
            try{ if(in != null) in.close();}
            catch(IOException e) {}
        }
        System.exit(0);
    }
}
```

? Writer.java

```
import java.io.*;
import java.net.*;

class Writer extends Thread{
    Client client;

    public Writer(Client c){
        super("Client Writer");
        client = c;
    }

    public void run(){
        DataInputStream in = null;
        PrintStream out = null;

        try{
            String line;
            in = new DataInputStream(System.in);
            out = new PrintStream(client.socket.getOutputStream());
            while(true){
                line = in.readLine();
                if (line == null) break;
```

```
                out.println(line);
            }
         }
            catch(IOException e){ System.err.println("Writer: "+ e);}
            finally{ if(out != null) out.close();}
            System.exit(0);
        }
}
```