Flots d'exécution multiples et Java

Cédric Motsch, Michel Banâtre, Jean-Paul Routeau, Cyril Ray

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Flots d'exécution multiples et Java

Cédric Motsch, Michel Banâtre, Jean-Paul Routeau, Cyril Ray

N° 5570

Mai 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5570--FR

Flots d'exécution multiples et Java

Cédric Motsch, Michel Banâtre, Jean-Paul Routeau, Cyril Ray*

Thème COM — Systèmes communicants

Projet ACES

Rapport de recherche n° 5570 — Mai 2005 — 31 pages

Résumé : L'objectif de cette étude est d'aborder les problèmes liés à l'exécution de plusieurs flots d'exécution dans un système d'exploitation en Java. En effet, ce dernier doit être capable d'exécuter, de manière concurrente, plusieurs applications *multithreadées* pouvant utiliser différentes API. Cet article établit en premier lieu un parallèle par l'étude des approches et ses mécanismes utilisés dans les systèmes natifs, ainsi que ceux utilisés dans les systèmes Java et machines virtuelles multiapplications existantes. La première partie est consacrée à l'isolation et la protection des applications et du système. La deuxième est consacrée à la construction des threads Java au dessus de structures existantes et aux contraintes de mapping. Enfin, la dernière partie traite de l'ordonnancement des flots et des problèmes liés à la concurrence.

Mots-clés : thread, processus, java, ordonnancement, mapping, isolation

This work is done under a research and development collaboration between Texas Instruments and Inria Rennes.

* prenom.nom@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Multiple execution flows and Java

Abstract: This paper gives an overview of problems related to the execution of multiple execution flows in a Java runtime environment, where the concurrent execution of several multithreaded Java applications is considered. It describes mechanisms used in native operating systems to solve these problems, then tries to study their relevance in Java, also by studying mechanisms used in Java operating systems and java virtual machines which can execute multiple applications per instance. We first study isolation and protection between applications. This report then describes how Java threads can be mapped over structures provided by the underlying system, which may be native or Java. Finally it studies how execution flows can be scheduled and the related concurrency issues.

Key-words: thread, process, java, scheduling, mapping, isolation

Table des matières

1 Introduction

L'objectif de cette étude est de présenter les problèmes liés à la gestion des flots d'exécution dans le cadre d'un système d'exploitation Java.

Java est un langage de programmation orienté objet créé par Sun Microsystems en 1995 [Saulpaugh et Mirho, 1999]. Il dispose de plusieurs atouts comme :

- la portabilité : Le code Java est compilé en un code intermédiaire, le bytecode, indépendant de la plate-forme sur laquelle il s'exécutera, celui-ci étant ensuite interprété par une machine virtuelle. Cette compatibilité binaire permet d'écrire et distribuer des applications sans avoir, en théorie et dans certaines limites, à se soucier de l'environnement d'exécution.
- la sécurité : Java, contrairement à des langages comme C, ne permet pas d'accéder à une adresse mémoire en utilisant un pointeur. Un programme Java utilise des références qui ne peuvent être forgées. De plus, le bytecode est vérifié lors de son chargement par l'environnement d'exécution.

Dans le cadre de l'étude, on s'intéressera particulièrement aux problèmes liés à la gestion des flots en Java, comment ces problèmes ont été traités dans des systèmes natifs, ainsi que les solutions proposées et implémentées dans le cadre de Java.

2 Problématique

Par la suite on désignera par *application* l'instance d'une application, et comme *flot d'exécution* une séquence d'instructions disposant de son contexte d'exécution propre (instruction courante, pile, ...).

On cherche à exécuter plusieurs applications Java de manière concurrente, chaque application pouvant être composée de plusieurs flots d'exécution. Dans le cadre de Java, ces flots sont des "threads", définis dans les APIs (java.lang.Thread). Plusieurs problèmes découlent de cette situation :

Tout d'abord, il faut assurer un minimum de protection et empêcher qu'une application perturbe le fonctionnement d'autres applications ou du système, ce qui implique une politique de protection et d'isolation adaptée.

Ensuite, il faut pouvoir établir la correspondance entre les threads des API, utilisés dans les programmes Java, et les flots d'exécution vus par le système, en prenant en considération à la fois les contraintes des API (fonctionnalités des threads, ordonnancement, ...) et celles du système (coût du changement de contexte, ...).

Enfin, il faut gérer l'exécution des flots par une politique d'ordonnancement adaptée, en définissant également les moyens de synchronisation.

Ces problèmes ne sont pas spécifiques à Java et ont été traités dans d'autres systèmes. Si certaines des solutions apportées sont toujours valables, d'autres peuvent être superflues au regard des possibilités supplémentaires offertes par Java. Différentes réalisations de machines virtuelles Java multi-applications ou de systèmes Java ont également apporté des réponses à ces différents problèmes en, (1) prenant en compte les spécificités du langage, (2) l'enrichissant ou (3) utilisant des mécanismes plus conventionnels.

Les sections suivantes présentent pour chaque problème comment il a été traité dans les systèmes natifs, les différences induites par Java et la façon dont il a éventuellement été traité dans le cadre de Java.

3 Isolation et protection des flots

La nécessité d'isoler les flots est motivée par différentes raisons. La première est la protection entre l'exécution de différents flots. En effet, à partir du moment où les flots ne sont pas interdépendants, par exemple entre deux applications distinctes, il faut pouvoir assurer un minimum de sécurité. On peut résumer les objectifs de celle-ci en trois points [d'Ausbourg, 2001] :

- Garantir la confidentialité des données en empêchant un flot non sûr d'accéder à des informations confidentielles. Ce point est particulièrement important dans un environnement multi-utilisateur.
- Préserver l'intégrité des donnés en empêchant un flot non sûr de les modifier arbitrairement.
- Maintenir la disponibilité de ressources et de services critiques. Un flot ne doit pas empêcher d'autres flots de s'exécuter, par exemple en monopolisant une ressource.

Ces objectifs restent cependant à moduler en fonction de la criticité des flots à exécuter, de l'importance des données à protéger et du surcoût que peut engendrer l'ajout de mécanismes de protection.

La deuxième raison est de faciliter la gestion des ressources. Si une ressource (par exemple une zone mémoire) est spécifique à un flot d'exécution et accessible uniquement par celui-ci, on pourra la libérer à la fin de son exécution.

Afin de répondre à ces besoins, les systèmes classiques utilisent :

- Des structures de données représentant des flots d'exécution ou des unités d'allocation de ressources.
- Des mécanismes de privilèges d'exécution pouvant reposer sur des fonctionnalités matérielles.

3.1 Les processus

Un processus est une abstraction de l'environnement du flot d'exécution. On décline ceux-ci en processus lourds et processus légers en fonction des ressources qui peuvent être communes ou spécifiques.

3.1.1 Processus lourds

Introduits dans vers la fin des années 60, les processus lourds sont les premiers processus à avoir vu le jour [Lampson, 1967].

Le principe d'un processus lourd consiste à isoler toutes les ressources (mémoire virtuelle, entrées/sorties, descripteurs de fichiers, ...), entre chaque structure d'exécution. Cette dernière est ainsi une représentation de l'environnement d'exécution du processus avant le changement de contexte qui devra être restauré lorsque le processus sera à nouveau choisi pour exécution. Les espaces d'adressage entre deux processus lourds étant disjoints, ils doivent utiliser des mécanismes de communication comme les échanges de messages ou les signaux s'ils doivent échanger des informations.

L'interdiction d'accéder directement à des ressources d'un autre processus permet d'éviter que le dysfonctionnement d'un flot d'exécution n'ait pas de conséquences sur l'exécution des autres flots. Cependant, si deux processus doivent échanger fréquemment des informations, le coût des communications peut être pénalisant par rapport à l'accès direct à une valeur partagée. De plus, la création et le changement de contexte entre processus lourds peut être coûteux. En effet, il faut initialiser les structures de données à la création, les permuter à chaque changement de contexte, sans compter les surcoûts dus au matériel comme les défauts de cache. Plusieurs optimisations ont été néanmoins apportées pour diminuer ces coûts. On peut citer par exemple le *Copy-On-Write*, qui partage en lecture une zone mémoire d'un processus fils jusqu'à ce que ce dernier ait besoin d'écrire dans cette zone.

Les processus lourds sont intéressants si l'on exige un degré d'isolation élevé, par exemple entre deux applications.

3.1.2 Processus légers

Le processus lourd, tel qu'il a été décrit dans la section précédente, n'est pas adapté à tous les usages. Par exemple, une application disposant de plusieurs flots d'exécution interdépendants n'a pas besoin du niveau d'isolation fourni par les processus lourd, tout en subissant des surcoûts inutiles lors des communications et des changements de contexte entre ses flots. Les processus légers répondent à ce problème en offrant des structures plus souples qui permettent à des processus de

partager des ressources communes tel que, par exemple, l'adressage mémoire.

Linux est un exemple de système utilisant les processus légers. Il est ainsi possible, lors de la création d'un processus, de définir ce qu'il partagera avec son parent [Bovet et Cesati, 2003]. Notons que Linux offre toutefois la possibilité de créer un processus lourd en choisissant de ne rien partager. Cela permet une utilisation relativement souple en fonction des besoins d'isolation et des performances recherchées. Un autre exemple d'utilisation de processus léger est le micronoyau Mach où les ressources sont regroupées dans une structure *task*, à laquelle on associe un certains nombre de processus appelés *threads*.

3.1.3 Flots privilégiés

Les structures définies jusqu'à présent utilisent principalement une isolation symétrique. Or on peut imaginer une situation où un flot fait confiance à un autre (c'est à dire accepte de dépendre de son bon fonctionnement) sans avoir pour autant la réciproque. Dans ce cas on utilise des niveaux de privilège. Ceux-ci sont souvent restreints à deux : le mode utilisateur et le mode noyau.

L'utilisation de niveaux de privilège reposent sur le principe de permissions, par exemple des autorisations de lecture, écriture ou exécution dans une zone mémoire. Les permissions accordées à un niveau de privilège donné sont un sous-ensemble de celles accordées à un niveau plus privilégié. Lorsqu'un flot a besoin d'utiliser des ressources réservées à un niveau plus privilégié, il doit faire appel à ce dernier en utilisant des points d'entrée définis et contrôlés.

Dans le cas de Linux sur l'architecture IA32 les niveaux de protection sont implémentés matériellement [1] et les *traps* générés par une interruption logicielle que les flots utilisateurs ont la permission d'émettre via un appel système [Bovet et Cesati, 2003]. En dehors de la pile qui est spécifique à chaque niveau d'exécution, il y peu de choses à sauvegarder et à restaurer lors du passage d'un niveau à un autre, ce qui rend le changement de contexte très rapide en comparaison à un changement entre deux processus, même légers. Cette méthode est intéressante également pour les flots d'exécution nécessitant un traitement rapide, comme les traitements d'interruptions, et qui ne peuvent se permettre de subir un changement de contexte complet.

Multics dispose de huit niveaux de privilèges dits *anneaux* [Schroeder et Saltzer, 1972]. Ces niveaux ont pu être implémentés en utilisant la segmentation mémoire (lorsque l'architecture le permet), qui spécifie des droits de lecture, d'écriture et d'exécution en fonction du niveau. Contrairement à Linux, le passage à un anneau privilégié ne s'effectue pas par une interruption, mais par l'appel d'un portail (*gate*), fonction dont l'adresse est spécifiée dans le segment privilégié. Si les paramètres sont valides et que l'appelant est autorisé à appeler le *gate*, l'exécution se poursuit dans

[1] 4 niveaux sont implémentés, mais Linux n'en utilise que 2.

le niveau de privilège du *gate*, avant de revenir au niveau initial à la fin de l'exécution.

Certaines tâches peuvent également s'exécuter uniquement dans un mode privilégié. Ainsi, dans les systèmes d'exploitation monolithiques. Le noyau peut également disposer de tâches de fond. Ces tâches sont généralement mappées dans des processus légers fonctionnant dans l'espace d'adressage du noyau. C'est le cas, par exemple, du processus *'kswapd'* sous Linux qui se charge de la libération de pages mémoire [Bovet et Cesati, 2003].

3.2 Isolation des fbts Java

L'isolation des applications les unes envers les autres peut être envisagée soit pour des raisons de sécurité, soit pour faciliter la gestion des ressources en offrant une abstraction supplémentaire.

Le langage Java offre du point de vue de la sécurité des mécanismes que n'offrent pas des langages comme C. Tout d'abord il est impossible de forger une référence vers un autre objet, contrairement à un pointeur en C. Cela permet un accès plus strict aux références. Ensuite, il dispose de différents niveaux d'accès aux champs et aux méthodes : public (accessible à toutes les classes), protected (accessible uniquement aux classes du package et aux descendant), private (accessible uniquement à l'intérieur de la classe) et l'accès par défaut n'autorisant que les classes du package. Enfin, le bytecode est vérifié avant l'exécution afin, notamment, de s'assurer qu'il n'y a pas de références illicites.

Il reste intéressant de pouvoir isoler les applications. Une première raison est, si des applications partagent le même code, de pouvoir utiliser des variables statiques de valeur différentes entre ces applications [Czajkowski, 2000]. En effet, contrairement aux objets, elles sont directement accessibles aux différents flots. De plus, le dysfonctionnement d'une application ne doit pas, dans la mesure du possible, entraîner celui des autres.

Les sections suivantes présentent les techniques utilisées pour séparer les applications Java dans différentes implémentations :
- de machines virtuelles supportant l'exécution simultanée de plusieurs applications,
- de systèmes d'exploitation écrits en Java.

3.2.1 JSR-121 et les isolates

Le JSR (*Java Specification Request*) 121 est une proposition cherchant à introduire un niveau d'isolation plus fort que celui des threads Java. Pour cela, elle cherche à enrichir l'API en introduisant la notion d'*isolates*. De la même manière que les threads, les isolates sont des objets, dérivés de la classe *java.lang.Isolate* et se comportent comme tels. Un isolate est similaire à un processus

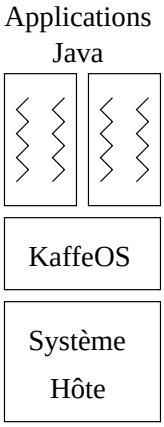

FIG. 1 – Rôle de KaffeOS

lourd dans le sens où l'application s'y exécutant se comporte comme si elle disposait de la machine virtuelle pour elle seule. Ainsi chaque isolate dispose de son propre tas, de ses propres threads, ses propres résolutions de noms et ses propres données statiques.

Les isolates du JSR 121 peuvent communiquer par des canaux de communications unidirectionnels, dérivés d'une classe Link en envoyant des messages, dérivés de la classe *IsolateMessage* [Soper et al., 2002]. La création d'un canal de communication nécessite de spécifier un isolate émetteur et un isolate récepteur. Afin de ne pas se limiter à des communications synchrones, il est possible, grâce à une interface de type *IsolateMessageListener*, d'appeler automatiquement une méthode dès qu'un message est reçu.

3.2.2 KaffeOS ou les processus Java

KaffeOS est une machine virtuelle Java conçue pour exécuter plusieurs applications concurrentes sur une même instance de machine [Back, 2002] (cf figure 1). Elle réintroduit la notion de processus, ainsi que la séparation entre utilisateur et système.

Les processus ont une classe java dédiée, *kaffeos.sys.Process* [2], qui permet de démarrer des nouveaux processus, de les terminer proprement, ainsi que d'accéder à leurs propriétés.). Les références directes entre les objets de différents processus sont interdites. Pour cela, une vérification est

[2] A ne pas confondre avec la classe *java.lang.Process* qui permet de lancer des applications natives sur un nouveau processus du système hôte

effectuée en utilisant les barrières d'écriture (chaque fois d'une nouvelle référence est écrite dans un tas, l'environnement d'exécution est prévenu). Si la nouvelle référence n'est pas autorisée, une exception est levée. Chaque processus dispose également de son propre chargeur de classe et de son propre ramasse-miettes. Le choix d'affecter un ramasse-miette par processus se justifie par le fait que celui-ci tourne avec une priorité supérieure aux autres threads Java et qu'une application appelant trop souvent un ramasse-miettes commun pourrait empêcher les autres applications de s'exécuter.

KaffeOS définit également une partie système, composée de code sûr, qui dispose également de son propre tas, de son propre ramasse-miettes et de son propre chargeur de classes. Les objets du système peuvent avoir des références vers les tas des utilisateurs. Le système intègre les classes définies dans l'API standard, qui sont accessibles aux processus utilisateurs. Lorsqu'un processus a ainsi besoin de charger une classe système, son chargeur de classes délègue le travail au chargeur de classes du système. L'utilisation de classes communes a néanmoins nécessité une réécriture de l'API (par exemple la classe *java.lang.System.out*) afin de prendre en compte le multi-applications.

Les processus de KaffeOS peuvent communiquer par le biais de tas partagés (cf figure 2). Les classes destinées à générer des objets partagés doivent se trouver dans des packages commençant par *shared.**. Avec les classes systèmes, les classes partagées sont les seules à faire partie d'un espace de nommage commun entre les processus. Lorsqu'un processus doit charger une classe partagée, un tas partagé est créé et le chargeur de classe partagé termine le chargement. Afin d'éviter qu'un objet partagé soit un relais indirect pour accéder à une classe du tas d'un autre processus, ce qui aurait des conséquences désastreuses sur la sécurité, les références vers l'objet d'un tas de processus sont interdites depuis un tas partagé. L'affectation d'un objet à un tas partagé ou système dépend de sa classe et son paquetage (*package*) d'appartenance. Les tas partagés n'ont pas de ramasse-miettes propres, le travail étant dévolu au ramasse-miettes du système.

L'isolation est donc assurée à la fois :

1. par les chargeurs de classes pour l'affectation d'un objet à un tas
2. au moyen des barrières d'écriture pour empêcher les références entre les tas

3.2.3 JKernel et les capacités

JKernel est un système écrit en Java s'exécutant au dessus d'une machine virtuelle standard sans modification de cette dernière. Son objectif est d'associer des applications à des domaines au sein d'une même instance de machine virtuelle [von Eicken et al., 2001]. Les références inter-domaines sont interdites et la communication s'effectue par capacités, ces dernières permettent :

– Un contrôle fin des objets sur lesquels les autres domaines peuvent effectuer des opérations. Dans le cas d'une référence directe, toute méthode publique pourrait être appelée.

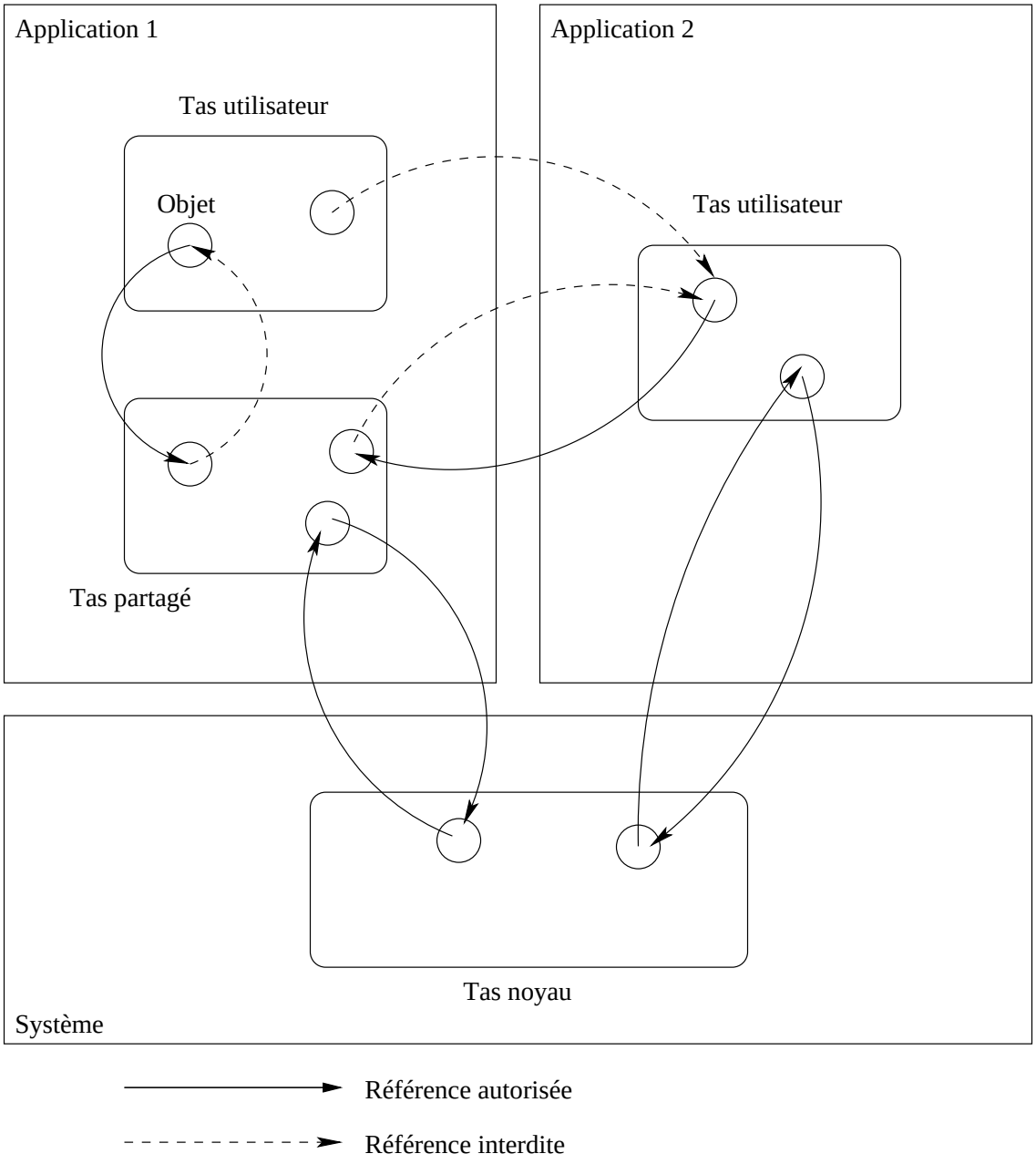

FIG. 2 – références autorisées entre les tas de KaffeOS

– En dehors des capacités, aucun paramètre de fonction n'est passé par référence. Les objets sont copiés en utilisant la sérialisation. Bien que le copiage engendre un surcoût par rapport à une référence directe, il permet de garantir le cloisonnement entre les domaines, en empêchant d'obtenir des références inter-domaines en dehors des capacités.
– Une capacité peut être révoquée par le domaine appelé à tout moment, et en particulier lorsque celui-ci se termine. L'utilisation d'une capacité révoquée génère une exception qui peut être récupérée par le domaine appelant. Ainsi un défaut de fonctionnement d'un domaine n'entraîne pas systématiquement celui des autres.

Les capacités sont implémentées en utilisant le seul langage Java. Une capacité est un objet, dérivé de la classe *Capacity* ayant pour champ privé l'objet manipulé dont la référence est obtenue à la construction. Pour qu'elle puisse être accédée par d'autres domaines, la capacité doit s'enregistrer dans le dépôt du domaine en utilisant un nom de référence, qui servira de repère pour récupérer la capacité. Ainsi au lieu d'avoir une référence directe vers l'objet et appeler ses méthodes, on appelle les méthodes de la capacité qui peuvent, à leur tour, appeler les méthodes de l'objet encapsulé (cf figure 3).

Chaque domaine dispose de son propre espace de nommage et chargeur de classe. Ainsi, JKernel permet d'implémenter un certain degré de protection sans enrichir le langage ou l'API Java.

3.2.4 Autres implémentations

Jx est un système d'exploitation écrit en grande partie en Java, mais compilé en natif, et dont la structure est divisée en domaines [Golm et al., 2002]. Chaque domaine dispose de son propre code, son propre tas et ses threads. Un domaine spécial, appelé domaine 0, est le seul à contenir du code natif et se charge des fonctions de plus bas niveau. Plutôt que de permettre à un autre domaine d'accéder directement à un objet, le principe consiste à fournir un intermédiaire appelé *portail*. Le fonctionnement d'un portail est proche de celui d'une capacité. L'exécution d'une méthode à l'intérieur du portail se fait dans le contexte du domaine appelé, ce qui évite les risques de blocage du domaine appelant.

Luna est une extension de machine virtuelle permettant également d'exécuter plusieurs applications sur une même instance de machine. Elle reprend les principes utilisés dans JKernel en permettant néanmoins un partage limité via un mécanisme de pointeurs distants. Un pointeur distant permet à un domaine d'avoir une référence vers un objet d'un autre domaine, mais cette référence peut être annulée par le domaine propriétaire de l'objet. Malheureusement, cette implémentation se base sur un enrichissement du langage. En effet des bytecodes supplémentaires ont été introduits pour manipuler les pointeurs distants, qui définissent également un nouveau type Java [Hawblitzel, 2000].

MVM (Multitasking Virtual Machine) est une modification de la machine virtuelle de Sun qui supporte le multiapplication. Les applications peuvent partager du code (qui n'est chargé qu'une seule fois en mémoire), mais les champs (variables, méthodes, code) statiques sont spécifiques à

```
/*************************************
Capacité par encapsulation
*************************************/
public Class ObjetInterne
{
void methode(int a) { ... }
}

public class CapaciteObjet
{
  private ObjetInterne objetEncapsule;
  private boolean valide;

  public void CapaciteObjet(ObjetInterne o)
  {
   objetEncapsule = o;
   valide=true;
  }

  public void methode(int a)
  {
   if(valide)
   {
   ObjetEncapsule.methode(a);
   }
   else
   {
   throws new RemoteException()
   }
  }

  public void invalider()
  {
   valide=false;
  }
}
```

FIG. 3 – création de capacité par encapsulation de l'objet

chaque application [Czajkowski, 2000]. Pour faciliter cette séparation, le bytecode est modifié au chargement des classes en divisant chaque classe contenant des champs ou des méthodes statiques en trois classes : la classe de base dont on a ôté les statiques, une classe contenant les champs statiques et une contenant les méthodes statiques. Seule la classe contenant les champs statiques peut être chargée plusieurs fois.

3.2.5 Synthèse

Nous avons vu dans cette partie comment les différentes propositions ont essayé d'isoler les applications Java au sein d'une même instance de machine virtuelle. On peut résumer les différentes techniques d'isolation utilisées de la manière suivante :

- Utilisation des fonctionnalités du langage. L'utilisation de méthodes et de champs privés et de leur encapsulation permet de prévenir des accès directs offrant ainsi un meilleur contrôle.
- Utilisation de chargeurs de classes distincts. Chaque application dispose de son propre espace de nommage et de son chargeur de classes. Le bytecode peut éventuellement être édité au chargement.
- Enrichissement de l'API Java. On définit des classes représentant un niveau d'isolation supérieur (comme les Isolates du JSR121), l'objet correspondant peut être manipulé par le programmeur, soit par la machine virtuelle.
- Modification de la machine virtuelle. On peut ainsi définir plusieurs ramasse-miettes (KaffeOS, Jx), utiliser des barrières d'écriture pour vérifier les références (KaffeOS), ...

4 Construction des threads Java

Java est un langage permettant de programmer des applications multithreadées. Les différentes API Java fournissent des classes de threads (en particulier *java.lang.Thread*), qui définissent les moyens de les manipuler.

Le système sur lequel les applications Java (et leur machine virtuelle) sont exécutées dispose de ses propres structures pour gérer les flots d'exécution. La projection (*mapping*) consiste à déterminer l'utilisation ces structures pour implémenter les threads définis dans les langages et les API.

Dans le cas d'une application Java s'exécutant sur une machine virtuelle classique, on dispose de deux niveaux de mapping. La machine virtuelle pouvant elle-même être une application multithreadée, le premier niveau consiste à établir une correspondance entre les threads de la machine virtuelle et les flots du système. Le second niveau consiste à associer les threads Java et les threads de la machine virtuelle.

Dans cette section, on présentera les politiques de mapping utilisées dans les systèmes classiques, et les contraintes imposées par les threads Java dans le choix de celles-ci.

4.1 Projection des fbts utilisateurs

Le multiflots au sein d'une application peut être géré au niveau système et au niveau utilisateur. S'il n'est géré qu'au niveau utilisateur, lorsqu'un changement de contexte au sein de la même application se produit, le système sous-jacent exécute toujours le même flot : aucun appel système n'est utilisé, et la même structure reste active pour le système. En revanche, si le multiflots est géré au niveau système, la création d'un flot et le changement de contexte entre deux flots passe par un appel système.

On distingue trois principales politiques de mapping. La première, qui consiste en une gestion exclusivement utilisateur, est le *many to one*. Les deux autres politiques, le *one to one* et le *many to many* nécessitent une interaction entre le niveau utilisateur et le niveau système.

4.1.1 Many to one

Le *many to one* est une politique de mapping qui consiste à affecter une seule structure système pour tous les flots de l'application (figure 4). Le système ne voyant qu'un seul flot, la gestion des flots au sein de l'application s'effectue sous l'entière responsabilité du programmeur [Chelf, 2001]. Les premières implémentations de la JVM de Sun sur Solaris utilisaient cette politique : les threads Java sont des *green threads* qui étaient mappés dans un seul processus.

Le principal avantage de cette approche est que le coût du changement de contexte et la création de flots. Ceux-ci, ne passant pas par des appels système, n'ont pas le surcoût dû à ces derniers. De plus, on ne change généralement que le strict nécessaire (on peut par exemple décider de conserver certains registres), ce qui n'est pas toujours le cas lorsqu'on change entre deux processus, même légers, du système. Si le système ne gère que des processus lourds, dont le changement de contexte est extrêmement coûteux, le *many to one* est une approche intéressante pour l'exécution d'applications multithreadées. Enfin, il est possible d'obtenir un meilleur contrôle sur l'exécution des threads, qui n'est pas soumise à l'ordonnanceur du système.

Cette politique a néanmoins un inconvénient majeur. En effet, si l'application effectue un appel système bloquant, tous ses flots seront bloqués puisque le système bloquera la structure unique associée. Le système ne peut pas non plus forcer un changement de contexte entre deux flots de l'application (par exemple après une interruption), faute de connaissance des différents flots de celle-ci.

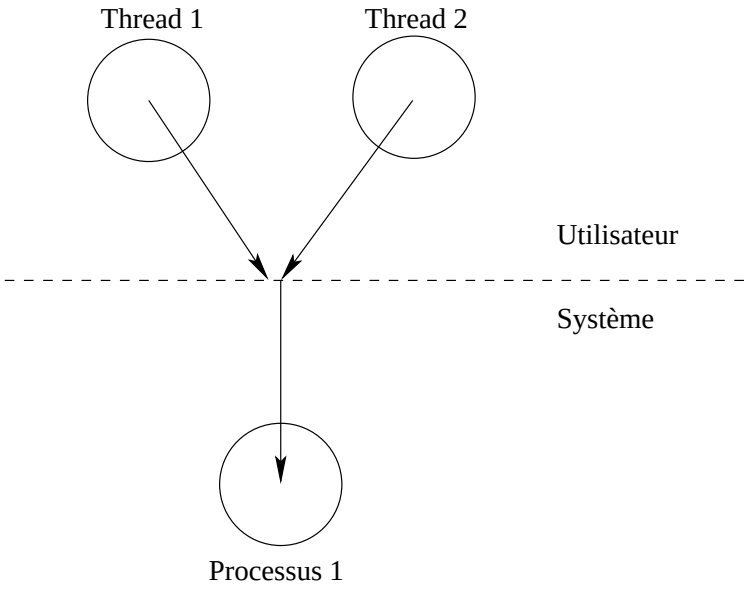

FIG. 4 – many to one

Jalapeño, une machine virtuelle Java écrite en Java, utilise cette politique afin d'avoir un contrôle total sur l'exécution de ses Threads Java. Elle utilise un thread natif (pThread sous AIX) par processeur virtuel. Les threads Java ne peuvent être préemptés qu'à certains endroits dits *sûrs*, par exemple lors de l'appel d'une méthode [Alpern et al., 1999].

4.1.2 One to one

A l'opposé du *many to one*, le *one to one* consiste à affecter une structure système, généralement un processus léger, pour chaque flot utilisateur (figure 5). La création d'un flot conduit alors à la création d'un processus par le système et le changement de contexte est généralement provoqué par l'ordonnanceur.

Cette bijection entre le flot utilisateur et les structures systèmes permet d'obtenir un bon niveau de parallélisme. Ainsi, si un flot effectue un appel système bloquant, il ne bloquera que lui même, et non d'autres flots de l'application. De plus, sur un système multiprocesseurs, plusieurs flots de l'application peuvent être exécutés en parallèle, ce qui n'était pas le cas avec la politique précédente.

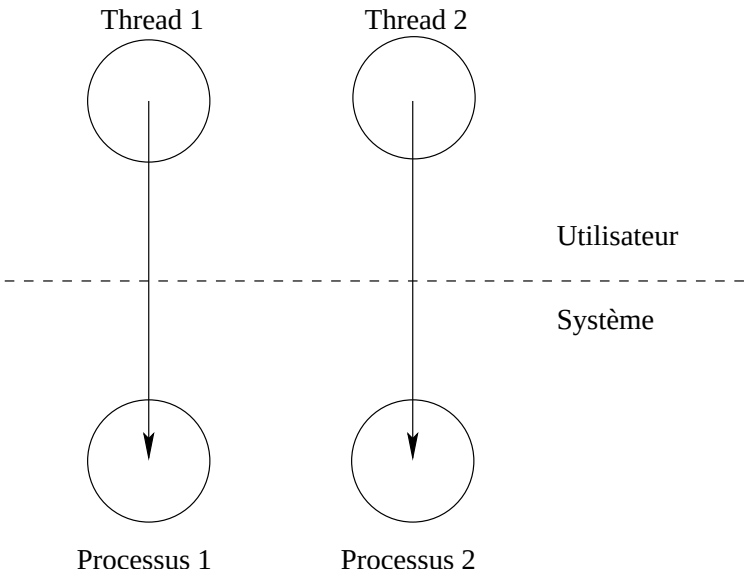

FIG. 5 – one to one

L'inconvénient du *one to one* réside dans l'éventuel surcoût dû aux appels système, notamment lors de la création d'un thread ou d'un changement de contexte. Dans les systèmes où ce coût n'est pas trop pénalisant cette politique facile à mettre en oeuvre est souvent utilisée. L'implémentation actuelle des threads Posix et de la JVM de Sun sur Linux et Solaris utilise ainsi cette politique.

4.1.3 Many to many

Dans le cas de la politique *many to many*, le système affecte à l'application un certain nombre de structures (i.e. processus) et laisse le soin à celle-ci d'y associer ses flots (figure 6). Lorsque le programmeur crée un thread, il peut ainsi décider si celui-ci est affecté à un processus système unique ou s'il peut migrer entre plusieurs processus. De plus tous les processus de l'application sont bloqués, le noyau peut créer des processus supplémentaires pour l'application s'il lui reste des flots pouvant être exécutés.

Le principal intérêt de cette politique est, en plus d'offrir un degré de parallélisme acceptable, de pouvoir réutiliser des structures systèmes existantes lorsqu'un flot utilisateur est créé. Si une application dispose d'un nombre important (de l'ordre de plusieurs centaines) de flots dont la durée vie est relativement courte, le gain de performances peut être appréciable car la création de nouveaux processus peut être coûteuse. Ce modèle permet par ailleurs de contrôler finement le

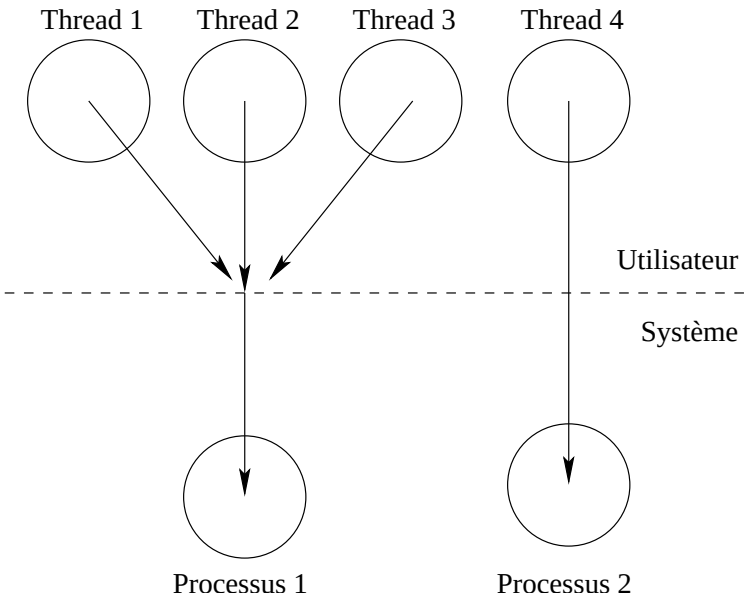

FIG. 6 – many to many

nombre de maximal structures que peut utiliser une application sans pour autant compromettre son exécution. L'implémentation de cette politique reste néanmoins complexe car elle nécessite entre autres que le noyau supporte les activations d'ordonnanceur, c'est à dire être capable d'appeler l'application pour lui signaler qu'un de ses flots est bloqué (ou débloqué) [Anderson et al., 1992].

Le *many to many* a été notamment utilisé pour implémenter les pthreads et la machine virtuelle Java dans les anciennes versions de Solaris (à partir de Solaris 9, le *one to one* est devenu la politique par défaut) [Shah, 2001].

4.2 Les threads Java

Il est possible d'utiliser plusieurs flots en Java par l'intermédiaire de la classe java.lang.Thread. D'un point de vue Java, les threads sont des objets dont la manipulation est soumise aux mêmes contraintes que les autres objets. Un thread peut être créé soit par une nouvelle instance de java.lang.Thread (ou une sous-classe), soit implicitement comme c'est le cas pour le premier thread (qui exécute la méthode *main()*).

Le démarrage et la création d'un code exécuté par un thread java sous-entend l'exécution d'un code qui initie le nouveau flot d'exécution. En fonction de l'initialisation, les points d'entrée sont :
- La méthode principale (*main()*) s'il s'agit du premier thread.
- La méthode *run()* de l'objet (qui doit implémenter l'interface *Runnable* au préalable) passé en paramètre du constructeur de l'objet *Thread*.
- La méthode *run* de l'objet dérivé d'une sous-classe de *Thread*.

D'un point de vue de la machine virtuelle Java, le contexte d'exécution des threads est composé d'une pile divisée en *stack frames*. Les frames sont des zones mémoires comportant une pile et des variables locales. Elles sont créées à chaque appel de méthode. Les objets Java présents en mémoire sont accessibles à tous les threads sous réserve qu'ils puissent se procurer une référence. Les threads de la machine virtuelle ont donc une structure proche des processus légers.

Les threads Java disposent de méthodes et d'attributs permettant d'influencer sur leur exécution et leur ordonnancement, parmi lesquels :
- *yield()*, qui permet au thread de signaler à l'ordonnanceur de manière coopérative qu'il peut élire un autre thread.
- *sleep()*, qui suspend l'exécution du thread pour une durée donnée.
- une priorité pouvant prendre 10 valeurs possibles
- *suspend()*, qui suspend l'exécution du thread jusqu'à ce que *resume()* soit appelé. Ces méthodes ont été désapprouvées à cause des risques d'interbloquage.
- *stop()*, qui arrête définitivement l'exécution d'un thread. Elle a été également désapprouvée.
- *getState()* (J2SE 5.0) permet de connaître l'état du thread. La figure 7 présente les différents états que peut prendre un thread Java (les méthodes désapprouvées ont été exclues).

4.2.1 Threads Java sur un système classique

Le comportement des threads Java peut varier en fonction de l'implémentation de la machine virtuelle. Si celle-ci utilise une politique de *many to one*, où tous les threads Java sont associé à un seul thread natif, elle peut gérer seule leur exécution.

Dans le cas contraire, en particulier si on utilise un mapping en *one to one*, les threads définis dans les bibliothèques natives doivent pouvoir répondre aux contraintes définies par les threads Java associés, en particulier en matière d'ordonnancement. Prenons pour exemple les priorités des threads Java, qui peuvent prendre 10 valeurs possibles : celles-ci ne seront pas forcément mappées en 10 priorités différentes gérées par le système [Pinilla et Gil, 2003] :
- Soit parce que la bibliothèque de threads ne permet pas de définir 10 priorités :
  - L'implémentation de la JVM 1.2 de Sun sous Windows associe les 10 priorités des threads Java en 7 priorités réelles permises par l'API Win32.
  - L'implémentation de la JVM 1.2 de Sun sous Linux n'utilise qu'une priorité car basée sur les threads POSIX et utilisant la politique SCHED_OTHER qui ne définit qu'une seule priorité.
- Soit pour satisfaire aux contraintes d'ordonnancement du système :

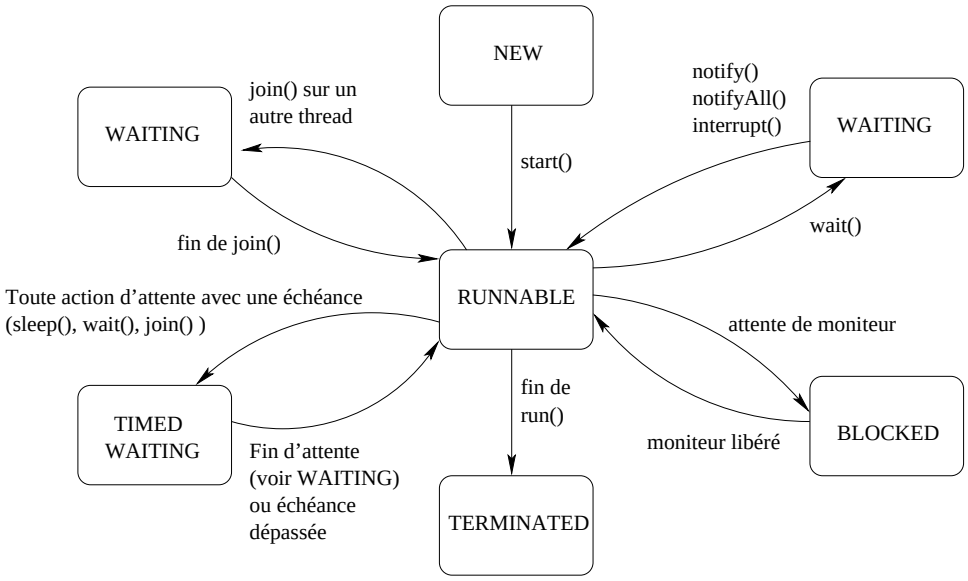

FIG. 7 – Etats des threads Java et transitions

– L'implémentation de la JVM 1.5 (J2SE 5.0) de Sun sous Solaris pourrait utiliser 10 priorités différentes avec les threads Solaris natifs (dans la classe *Timesharing*). Cependant, la priorité par défaut de cette classe étant la plus élevée, les threads Java d'une priorité inférieure à 10 seraient désavantagés par l'ordonnanceur par rapport aux autres applications exécutées. Pour remédier à ce problème tous les threads Java de priorité allant de 5 (défaut) à 10 (maximale) sont associés à une même priorité.

4.2.2 Threads Java dans les systèmes Java

JavaOS [Saulpaugh et Mirho, 1999] est l'un des premiers systèmes d'exploitation écrit à partir de Java. Les seules parties écrites en natif sont le système de boot, le micronoyau, la machine virtuelle et les interfaces (JBI, JavaOS Boot Interface, entre le noyau et le système de boot, et JPI, JavaOS Platform Interface, entre le noyau et la machine virtuelle). Bien que la machine virtuelle s'exécute pas au dessus d'un système d'exploitation complet, les threads Java utilisent néanmoins les threads fournis par le noyau en mapping *one to one*. Les priorités sont entièrement prises en compte (avec la possibilité de définir des priorités supplémentaires pour le système) et les primitives de gestion des threads (création, suspension, reprise, ...) sont directement implémentées en natif.

JxOS [Golm et al., 2002] est un système écrit en grande partie en Java. Il est structuré en domaines qui gèrent leurs propres threads. Un domaine spécial dit *domaine zéro* contient la seule partie écrite en native du système. La création d'un thread dans un domaine s'effectue par l'intermédiaire d'un appel au domaine zéro (en effet, la création de contexte se fait dans la partie native). La pile et les zones de locales sont ensuite gérées par le domaine d'appartenance. Par défaut, les threads d'un domaine ne peuvent être vus par d'autres domaines, y compris le domaine zéro qui ne gère alors que des domaines et non des threads. Cela permet ainsi un ordonnancement à deux niveaux (le domaine zéro ordonnance les domaines et ces derniers ordonnancent leurs threads). Contrairement aux implémentations précédentes, les threads Java ne reposent pas sur une bibliothèque complète de threads natifs. Ils peuvent néanmoins être encapsulés dans des domaines, ce qui suggère également une relation de *mapping*.

Le *mapping* des threads Java au sein d'un système découle d'une double relation. La première est le *mapping* de l'environnement d'exécution au dessus du natif, qui peut être une bibliothèque complète de threads ou juste un support minimal de gestion des contextes. Le deuxième est celui des threads au dessus de l'environnement d'exécution. Les politiques utilisées sont peu différentes des autres systèmes et dépendent principalement des contraintes posées par les threads Java.

5 Ordonnancement

L'ordonnanceur est un flot système chargé de choisir la structure (processus ou thread), de terminer éventuellement la sauvegarde du contexte du flot précédent à activer (une partie ayant été sauvegardée avant l'appel de l'ordonnanceur), puis de procéder à la restauration du contexte de flot associé. Une politique d'ordonnancement consiste à répartir le temps CPU entre les différents flots.

Afin de pouvoir effectuer son choix, l'ordonnanceur doit avoir connaissance des flots à ordonnancer, et par conséquent avoir une référence, directe ou indirecte, vers les structures les représentant. On appellera "tâche" la structure représentant un (ou plusieurs) flot d'exécution, qui sera manipulée par l'ordonnanceur pour effectuer son choix, indépendamment de la composition de cette structure (processus, thread, ...). On dira également qu'une tâche s'exécute sur un processeur si un des flots qu'elle représente ou encapsule est actuellement exécuté.

Définir une politique d'ordonnancement consiste globalement à répondre aux questions suivantes :

- A quel moment appelle-t-on l'ordonnanceur ?
- Lorsque l'ordonnanceur est appelé, quel processus doit-on exécuter ?
- Si l'on dispose de plusieurs processeurs, sur quel processeur les exécuter ?

On étudiera ces trois questions respectivement dans les trois sections ci-dessous.

5.1 Appel de l'ordonnanceur

5.1.1 Evénements

La première étape d'une politique d'ordonnancement est de déterminer les différents événements qui déclencheront l'appel de l'ordonnanceur. On peut établir la liste suivante :

- La tâche active ne peut plus être exécutée. A moins de vouloir laisser le CPU inactif, une autre tâche doit être choisie pour être exécutée.
- La tâche demande volontairement à être préemptée, dans ce cas l'ordonnanceur est appelé et peut choisir une autre tâche à exécuter. C'est sur cette approche qu'est basé le multitâche coopératif.
- Un événement externe, par exemple une interruption, déclenche un changement de contexte et appelle l'ordonnanceur. Dans ce cas, la tâche en cours est préemptée. Le multitâche préemptif est basé sur cette approche.

5.1.2 Coopération ou préemption

Le principal avantage du multitâche coopératif réside dans le fait que les changements de contexte entre tâches sont déclenchés par l'application. Il est plus facile d'écrire des sections critiques et de garantir l'indivisibilité de leur exécution. Une interruption peut toujours survenir, mais une fois le traitement terminé le flot de la tâche en cours reprendra son exécution. Cependant le coopératif n'est pas adapté pour l'exécution concurrente de plusieurs applications, car rien ne garantit qu'une application ne mobilisera pas le processeur au détriment des autres. Pour cette raison, le multitâche coopératif est, de nos jours, rarement utilisé en intégralité.

A l'opposé, le multitâche préemptif permet à d'autres tâches de s'exécuter sans l'accord de la tâche en cours. Dans le cas de tâches d'importances égales, cela permet une distribution équitable du temps processeur (principe du temps partagé), et si une tâche critique devient exécutable, elle peut préempter la tâche exécutée actuellement, la préemption pouvant à priori avoir lieu à tout moment. Il est en revanche difficile d'assurer l'atomicité des opérations effectuées, ce qui rend la conception de sections critiques plus difficile qu'en multitâche coopératif. Pour pallier à cet inconvénient, certains systèmes protègent certaines parties de leur code (qui doit être considéré comme 'fiable', e.g. le code noyau) contre la préemption. Le principal exemple est le noyau Linux jusqu'à sa version 2.4 : lorsqu'une tâche entre en mode noyau, elle ne peut pas être préemptée avant qu'elle revienne en mode utilisateur. Solaris a utilisé une différente approche : une tâche entrant en mode noyau peut être préemptée, mais sa priorité est fortement augmentée, ce qui limite les risques de préemption par d'autres tâches utilisateur [R.Graham, 1995].

L'API de threads Java est assez souple pour reposer sur du multitâche coopératif (via des méthodes comme *yield()*) ou préemptif (via les priorités). L'inconvénient se trouve alors dans la conception de programmes multithreadés, qui doivent pouvoir gérer les deux cas.

5.1.3 Synchronisation et indivisibilité des opérations

A partir du moment où une section de code, qu'il s'agisse d'une application utilisateur ou du noyau, peut être préemptée, des mesures de synchronisation doivent être prises. En effet, si plusieurs flots sont susceptibles d'accéder à la même ressource, il faut pouvoir garantir la cohérence de cette dernière :

- En évitant que deux flots modifient la ressource de manière simultanée, ce qui peut empêcher que toutes les modifications soient prises en compte (écrivains multiples).
- En évitant qu'un flot accède en lecture pendant qu'un autre flot la modifie, ce qui pourrait donner une valeur lue incohérente (lecteurs/écrivains).

Le système doit donc être capable, pour certaines opérations, d'assurer une atomicité, soit complète (en empêchant le flot d'être interrompu ou préempté). A cet effet plusieurs mécanismes ont été créés. Leur principe est de définir des points d'entrée et des points de sortie qui délimitent l'opération critique. Lorsqu'un flot arrive à un point d'entrée contrôlé par une de ces structures, une vérification est effectuée pour savoir s'il peut continuer son exécution. Si le test est positif, le flot peut poursuivre. Dans le cas contraire, soit il doit poursuivre dans une autre branche (on parle alors d'une vérification non bloquante), soit il se bloque sur le point d'entrée. Le franchissement d'un point de sortie déclenche un événement qui peut éventuellement débloquer un point d'entrée pour un autre flot. Dans les systèmes natifs, ces mécanismes d'exclusion reposent sur des mécanismes propres au matériel et à l'architecture comme les instructions atomiques [Hovemeyer et al., 2002] (de type *test and set*) ou la désactivation des interruptions.

Afin d'utiliser à bon escient les mécanismes d'exclusion, il faut savoir à quel niveau et de qui veut-on se protéger :

- Dans le cas de deux threads d'une même application se partageant une ressource interne, des mécanismes d'exclusion de haut niveau peuvent être définis dans la bibliothèque utilisée. Ces derniers, en fonction du mapping utilisé, peuvent reposer sur des mécanismes fournis par le système.
- Dans le cas de deux processus devant accéder à une ressource partagée, le système fournit des primitives d'exclusion adaptées : mutex, sémaphore, ... Les threads bloqués par ces structures sont gérés ensuite par l'ordonnanceur.
- Certains mécanismes fournis ne conduisent pas toujours à l'endormissement d'un thread, par exemple des mécanismes protégeant les structures utilisées par l'ordonnanceur. La plus connue et le verrou pivotant (*spinlock*) qui protège des accès concurrents des autres processeurs.
- Une autre possibilité est d'empêcher provisoirement la préemption. Soit en masquant les interruptions, soit, si ces dernières n'interfèrent pas avec les ressources utilisées, indiquer à l'ordonnanceur qu'il ne doit pas élire de nouveau flot (ce qui permet aux interruptions d'être prises en compte).

Le langage Java définit une structure d'exclusion mutuelle basique appelée le moniteur.

Un moniteur est une zone de code à laquelle un seul flot peut accéder à la fois (on dit alors que le code n'est pas réentrant). Cette zone peut être composée de plusieurs sections disjointes. Si un autre flot essaie d'exécuter le code, il se bloque jusqu'à ce que le flot sorte du moniteur. La déclaration d'un moniteur en Java se fait en utilisant l'attribut *synchronized* autour du code, de la méthode ou de la classe voulue. Au niveau du bytecode l'entrée et la sortie d'un moniteur s'effectue par l'exécution des instructions bytecode *monitorenter* et *monitorexit*. Un moniteur est associé à un objet unique, par conséquent, si deux sections de code dans une même classe sont déclarées *synchronized*, elles sont soumises au même moniteur.

A l'intérieur des moniteurs, Java dispose de mécanismes de synchronisation sur objets via les méthodes *wait()*, *notify()* et *notifyAll()*. Ces méthodes publiques appartenant à la classe java.lang.Object, tous les objets Java en héritent. Ces trois méthodes ne peuvent être appelées qu'à l'intérieur d'un moniteur. Si un thread appelle la méthode *wait()* d'un objet, il se retrouve bloqué, tout en libérant le moniteur, jusqu'à ce qu'un autre thread appelle la méthode *notify()* ou *notifyAll()* sur l'objet. Une fois libéré, le thread doit reprendre le moniteur avant de poursuivre son exécution. La différence entre *notify()* et *notifyAll()* réside dans le fait que la première réveille un thread au hasard et la seconde réveille tous les threads ayant effectué un *wait()* sur l'objet [Lea, 1997]. Pour un même moniteur, il ne peut exister qu'une seule condition d'attente et de réveil (mis à part l'attente temporisée).

Depuis J2SE 5.0, d'autres mécanismes ont été introduits avec le package *java.util.concurrent* [Microsystems, 2004]. On trouve ainsi les sémaphores, les types atomiques ou l'exclusion de lecture-écriture, mais aussi une amélioration des moniteurs qui permet par exemple d'utiliser plusieurs conditions d'attente au sein d'un même moniteur.

La plupart des structures d'exclusion définies par Java sont des structures de haut niveau, devant être implémentées au dessus de structures d'exclusion définies par le système hôte. Ainsi, dans JavaOS, les moniteurs sont directement gérés par le micronoyau natif. Dans le cadre d'un système Java, il faut pouvoir définir des structures d'exclusion et de synchronisation performantes, et ainsi que sur quels mécanismes elles se reposent.

5.2 Choix de la tâche à ordonnancer

La deuxième étape d'une politique d'ordonnancement est l'algorithme lui même. Plutôt que d'établir une liste exhaustive de tous les algorithmes proposés, on se contentera de rappeler ici les tendances principales.

Une tâche est *prête* si elle peut être choisie par l'ordonnanceur pour être exécutée sur un processeur. En effet, elle peut ne pas être exécutable au moment de l'ordonnancement pour différentes raisons, par exemple :

– La tâche peut être bloquée dans l'attente d'une ressource qui n'est pas encore disponible.

– La tâche a été arbitrairement suspendue et attend un signal de réveil.
– La tâche vient d'être créée mais elle n'a pas encore été initialisée.
– La tâche a terminé son exécution mais n'a pas été retirée de la liste.

Lorsqu'il est appelé, l'ordonnanceur doit choisir parmi les tâches prêtes celle qui sera exécutée sur un CPU donné en se basant sur différents critères d'éligibilité.

5.2.1 Utilisation de priorités

La priorité est une valeur, affectée à une tâche, qui représente son éligibilité. L'objectif est de faciliter l'élection de la prochaine tâche en choisissant la tâche prête dont la priorité est la plus élevée (ou la plus faible selon les systèmes, par abus de langage, on considèrera par la suite qu'une priorité élevée signifie une éligibilité élevée).

Lorsqu'au moins deux tâches prêtes sont de la même priorité, l'ordonnanceur doit utiliser une politique complémentaire pour départager. Un algorithme fréquemment utilisé pour ordonnancer des tâches de même priorité est l'ordonnancement en tourniquet (*round-robin*). On attribue à chaque tâche une tranche de temps pendant laquelle elle est exécutée sur le processeur. Si la tâche est toujours prête et n'a pas été préemptée lorsque la tranche de temps est expirée, la tâche suivante de même priorité est alors choisie pour être exécutée.

Une autre manière de procéder est l'ordonnancement FIFO (First In First Out), qui fait exécuter une tâche jusqu'à ce qu'elle se termine, se bloque, libère volontairement le CPU ou soit préemptée par une tâche plus importante. Cette méthode étant coopérative entre tâches de même priorité, elle peut présenter des risques de famine.

5.2.2 Priorités fixes et dynamiques

Lorsqu'une priorité est affectée à une tâche, elle peut, en fonction de la politique d'ordonnancement choisie, varier au cours du temps. En effet, si les priorités sont fixes, il suffit qu'une tâche soit toujours prête pour que les tâches de priorité inférieures ne puissent jamais s'exécuter. L'utilisation de priorités dynamiques pour des tâches non critiques n'ayant pas de contraintes temporelles permet d'éviter les famines.

Afin d'illustrer ces propos, étudions le cas de deux noyaux monolithiques : Linux 2.4 et Solaris 2.3, et de deux micronoyaux : L4 et Mach 3.

Dans le cas de Linux et Solaris, le noyau utilise simultanément plusieurs politiques d'ordonnancement. Ainsi, Linux dans sa version 2.4 dispose de trois politiques d'ordonnancement, qui correspondent aux trois politiques définies par la norme POSIX : priorités fixes départagées par

round-robin ou FIFO, et temps partagé (avec priorités initiales). La dernière catégorie, qui permet d'assurer que tout processus disposera de temps CPU à moyen terme, est celle par défaut des applications utilisateurs. Les deux premières, plus prioritaires, sont réservées à des tâches ayant des contraintes temporelles souples, et pour des raisons de sécurité nécessitent des privilèges superutilisateurs pour être utilisées [Bovet et Cesati, 2003]. Les tâches ordonnancées en temps partagé ont des priorités recalculées à chaque ordonnancement. Une fois qu'elles ont épuisé leur quota de temps, leur priorité devient nulle. Lorsqu'il ne reste plus de tâches de priorité non nulle, les priorités sont alors réinitialisées. On peut ainsi garantir que toute tâche prête sera élue à moyen terme, sous réserve qu'aucune tâche critique à priorité fixe ne monopolise le processeur.

Solaris 2.3 dispose de trois politiques similaires, par ordre croissant de priorité : *Timesharing* (temps partagé), *System* (priorités fixes départagées en FIFO) et *Real-Time* (priorités fixes départagées en round-robin, cette catégorie peut néanmoins être désactivée). La classe *System* est réservée aux tâches du noyau et aux tâches utilisateur se trouvant en mode noyau (ainsi ces dernières ne peuvent pas être préemptées par une autre tâche utilisateur tant qu'elles sont en mode noyau) [R.Graham, 1995]. La politique d'ordonnancement *Timesharing* de Solaris 2.3 est un exemple d'utilisation de priorités dynamiques. Lorsqu'une tâche consomme sa tranche de temps, sa priorité diminue. Si en revanche une tâche sort d'un état non prêt, sa priorité se voit augmenter. Ainsi, une tâche consommatrice de temps CPU se retrouvera avec une priorité faible, ce qui permet aux autres tâches de s'exécuter.

Le micronoyau L4 n'utilise en revanche qu'une politique simple de priorités fixes avec départage en *round-robin* [Potts, 1999]. La valeur de la priorité ainsi que la tranche de temps affectée à chaque tâche sont définis à l'extérieur du noyau, ce qui permet de construire différentes politiques d'ordonnancement à l'extérieur du noyau. Mach 3.0 utilise une politique similaire, mais modifie, dans un intervalle restreint, les priorités des tâches en fonction de leur utilisation du processeur [Black, 1990].

Dans le cadre de Java, l'utilisation de priorités fixes ou dynamiques pour ordonnancer les tâches est soumise aux mêmes contraintes. L'API standard des threads Java ne définit pas de politique d'ordonnancement (contrairement à POSIX). En revanche, la spécification RTSJ (Java Temps Réel) permet de spécifier des paramètres d'ordonnancement. Ainsi, si un thread est déclaré comme temps réel, sa priorité (définie dans les paramètres) est traitée comme fixe par l'ordonnanceur sauf exceptions comme les héritage de priorités à l'intérieur d'exclusions mutuelles [Bollella et al., 2000].

5.2.3 Affectation des priorités

L'affection d'une priorité ou d'une classe de priorité à une tâche dépend de différents facteurs. Parmi ceux-ci, notons :

- La nature de la tâche : les tâches dédiées au système peuvent avoir une éligibilité supérieure à des tâches d'application utilisateur (si celles-ci n'ont pas de contraintes temporelles). Par exemple, dans Solaris, une plage de priorités est dédiée aux tâches système. Les tâches utilisateur non temps réel ont obligatoirement une priorité inférieure [R.Graham, 1995]. Cette idée a également été reprise dans JavaOS où les threads du système peuvent avoir une priorité supérieure à la priorité maximale définie par Java.
- Le niveau d'interactivité de la tâche : dans les systèmes interactifs, on peut privilégier les tâches dépendantes des entrées/sorties (qui sont en général bloquées avant d'avoir consommé le quantum de temps imparti) sur les tâches de fond. Le but est de donner une meilleure impression de réactivité pour l'utilisateur.
- Les décisions du programmeur d'application : lorsque le programmeur code une application multithreadée, il peut, si le langage ou la bibliothèque de threads le lui permet, affecter une priorité différente à ses threads. Ces priorités peuvent éventuellement être représentées par des véritables priorités systèmes, en fonction de l'implémentation du langage ou de la bibliothèque.
- Les décisions arbitraires de l'utilisateur, par exemple la commande *nice()* sous les systèmes de type Unix, permet lancer un processus avec une priorité différente de la priorité par défaut.
- Les contraintes temporelles de la tâche, si celle si est temps réel, comme l'échéance, la périodicité et le temps d'exécution prévu.
- L'utilisation de structures d'exclusion mutuelle peut influencer dynamiquement la priorité d'une tâche. Un problème fréquemment traité est celui de l'inversion de priorité où une tâche C ne peut s'exécuter car elle est en attente qu'une autre tâche A moins prioritaire libère une ressource. Cette même tâche A ayant été préemptée par une tâche B plus prioritaire, mais moins que C. L'une des solutions proposées est l'héritage de priorité, où la tâche détenant l'objet de l'exclusion mutuelle se voit hériter de la priorité de la tâche la plus prioritaire en attente du même objet.

5.3 Choix du processeur

Dans le cas d'un système fonctionnant sur un multiprocesseur symétrique (SMP) à mémoire partagée, la politique d'ordonnancement consiste également à choisir le CPU sur lequel s'exécute la tâche. Deux paramètres sont à prendre en compte :

- L'affinité processeur d'une tâche. Il peut ainsi être intéressant d'affecter au même processeur deux tâches partageant l'adressage mémoire (deux processus légers d'une même application). En effet, le cache du processeur peut être réutilisé et le changement de contexte sera plus rapide, car la mémoire virtuelle reste inchangée.
- L'équilibrage de charge entre les processeurs. Si on cherche à exploiter au maximum les processeurs, il faut éviter d'avoir un processeur inactif pendant qu'un autre processeur dispose de plusieurs tâches en attente.

Ces deux problèmes sont résolus de manière différente selon qu'on dispose d'une liste globale de tâches commune à tous les processeurs ou d'une liste locale par processeur. Dans ce paragraphe, on prendra pour exemple Linux 2.4 (qui dispose d'une liste globale), Linux 2.6 (qui dispose de listes

locales).

La question de l'équilibrage de charge est surtout pertinente dans le cas d'une liste locale par processeur. Ainsi, dans Linux 2.4, la liste des tâches prêtes est protégée par un verrou pivotant et est consultée par chaque processeur qui a besoin d'élire une nouvelle tâche. Dans la version 2.6, chaque processeur vérifie à intervalles réguliers les listes des autres processeurs afin de déterminer si un équilibrage doit avoir lieu. La vérification se fait toutes les 200 millisecondes si la liste des tâches du processeurs n'est pas vide et toutes les millisecondes dans le cas contraire [Lindsley, 2004].

Le problème de l'affinité processeur se pose principalement dans le cas où une liste globale de tâches est gérée et dans la phase d'équilibrage de charge afin de déterminer s'il est opportun de migrer une tâche d'un processeur à un autre. Linux 2.4 fournit ainsi un bonus de priorité aux tâches (à priorité dynamiques) qui s'étaient exécutées précédemment sur le même processeur. Il fournit également un bonus pour les tâches partageant l'espace d'adressage de la tâche exécutée avant l'appel de l'ordonnanceur [Bovet et Cesati, 2003].

Le choix d'utiliser une liste globale ou une liste globale dépend de plusieurs facteurs. Une liste globale permet d'assurer qu'à tout instant, les taches de plus hautes priorité sont exécutées. En utilisant des listes locales, il faut vérifier les listes de chaque processeur. Solaris 2.3 effectue cette opération pour les tâches de la classe *Real-Time* (les plus prioritaires). De plus, si le nombre moyen de tâches prêtes est faible (de l'ordre du nombre de processeurs), du temps peut être perdu à réaliser des rééquilibrages de charge et des migrations de tâches. En revanche, les accès concurrents aux listes de tâches prêtes sont moins nombreux lors de l'utilisation de listes locales.

6 Conclusion

Nous avons vu au cours de cette étude les problèmes soulevés par la gestion du multiflots dans la conception d'un système, en particulier les différentes relations possibles et la manière de les traiter, ainsi que la façon de répartir leur exécution. Nous avons également regardé les contraintes et les possibilités offertes par le langage Java dans cet objectif, et les différentes propositions et implémentations réalisées.

Le premier problème est de définir le multi-applications au sein d'une même instance de machine virtuelle Java. Alors que les systèmes conventionnels disposent de processus lourd et légers permettant de traiter différents cas, et de construire les threads définis par les langages de manière adéquate, Java ne dispose que de threads similaires aux processus légers. Plusieurs implémentations ou propositions ont tenté d'introduire un niveau d'isolation proche du processus lourd, soit en enrichissant le langage Java (comme les *Isolates* du JSR 121 ou les *Processes* de KaffeOS), soit de manière transparente. La principale caractéristique est la séparation du tas (regroupant les objets) entre les applications. Certaines implémentations ont choisi également de séparer les ramasse-miettes, le

code, les statiques, ou les chargeurs de classes. Les communications inter-applications peuvent se faire soit par partage direct d'objets (KaffeOS), soit par capacités (JxOS), soit par messages (JSR 121), en fonction des performances et des besoins d'isolation recherchés.

La construction des threads des APIs Java au dessus d'un système, Java ou natif, est un autre problème. Ceux-ci imposent des contraintes et sont implémentés de manières différentes selon le système sous-jacent. En plus des règles de projection utilisées dans les systèmes natifs, il faut prendre en compte une projection à deux niveaux dans le cadre d'une machine virtuelle Java exécutée sur un système natif. Dans le cas d'un système Java, la construction des threads dépend de ce que fournit la partie native. Dans JavaOS, les threads utilisent ceux fournis par le micronoyau sous-jacent alors que dans Jx, la partie native ne sert qu'à la création et au changement de contexte. Même dans ce cas, les threads gérés par le système Java ne sont pas obligatoirement ceux utilisés dans les applications utilisateurs. On retrouve alors les problèmes vus dans les systèmes natifs.

Le dernier problème traité est l'ordonnancement. La politique d'ordonnancement même des threads Java est soumise aux mêmes contraintes que celles des systèmes d'exploitation conventionnels. Il faut tout d'abord déterminer les événements susceptibles d'activer l'ordonnanceur et d'effectuer un changement de contexte entre les threads. Ensuite une politique d'élection des threads doit être établie en accord avec les contraintes de ces derniers et, dans le cas d'un système multiprocesseur, politique de répartition adaptée. En revanche, certaines opérations comme le changement de contexte, qui nécessite, par exemple, un accès aux registres et à la pile ne sont pas réalisables en Java. Il reste également à définir les structures pour ordonner les threads Java en fonction de leurs caractéristiques (état, priorité effective, type d'attente, ...) afin d'obtenir des performances correctes lors de l'élection d'un nouveau thread, tout en maintenant la cohérence de ces structures. On retrouve également des contraintes similaires aux autres systèmes, auxquels s'ajoutent celles définies par les APIs Java comme les différents états des threads.

Références

[Alpern et al., 1999] Alpern, B., Cocchi, A., Lieber, D., Mergen, M., et Sarkar, V. (1999). Jalapeño - a compiler-supported java virtual machine for servers. *Workshop on Compiler Support for Software System (WCSSS 99)*.

[Anderson et al., 1992] Anderson, T. E., Bershad, B. N., Lazowska, E. D., et Levy, H. M. (1992). Scheduler activations : Effective kernel support for the user-level management of parallelism. Technical report.

[Back, 2002] Back, G. (2002). *Isolation, resource management and sharing in the KaffeOS Java runtime system*. Thèse de doctorat, The University of Utah, Salt Lake City, Utah, USA. 175 pages.

[Black, 1990] Black, D. L. (1990). *Scheduling and Resource Management Techniques for Multiprocessors*. Thèse de doctorat, Canergie Mellon University. 122 pages.

[Bollella et al., 2000] Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M., et Belliardi, R. (2000). *Real-Time Specification for Java*.

[Bovet et Cesati, 2003] Bovet, D. P. et Cesati, M. (2003). *Understanding the Linux Kernel*. O'Reilly.

[Chelf, 2001] Chelf, B. (2001). The fibers of threads. *Linux Magazine*.

[Czajkowski, 2000] Czajkowski, G. (2000). Application isolation in the java virtual machine. *Conference on Object Oriented Programming Systems Languages and Applications*.

[d'Ausbourg, 2001] d'Ausbourg, B. (2001). *Contribution à la définition de systèmes de confiance et de systèmes utilisables*. Habilitation à diriger des recherches, Université Paul Sabatier.

[Golm et al., 2002] Golm, M., Felser, M., Wawersich, C., et Kleinöder, J. (2002). The jx operating system. *USENIX 2002 Annual Conference*.

[Hawblitzel, 2000] Hawblitzel, C. K. (2000). *Adding Operating System Structure to Language-based Protection*. Thèse de doctorat, Cornell University. 140 pages.

[Hovemeyer et al., 2002] Hovemeyer, D., Pugh, W., et Spacco, J. (2002). Atomic instructions in java. *Proceedings of the 16th European Conference on Object-Oriented Programming*.

[Lampson, 1967] Lampson, B. W. (1967). A scheduling philosophy for multi-processing systems. *ACM Symposium on Operating Systems Principles*.

[Lea, 1997] Lea, D. (1997). *Concurrent Programming in Java*. Addison-Wesley.

[Lindsley, 2004] Lindsley, R. (2004). Kernel korner : What's new in the 2.6 scheduler. *Linux Journal*.

[Microsystems, 2004] Microsystems, S. (2004). *J2SE 5.0 Documentation*.

[Pinilla et Gil, 2003] Pinilla, R. et Gil, M. (2003). Jvm : platform independent vs. performance dependent. *SIGOPS Oper. Syst. Rev.*

[Potts, 1999] Potts, D. P. (1999). *L4 on Uni- and Multiprocessor Alpha*. Thèse de doctorat, The University of New South Wales. 78 pages.

[R.Graham, 1995] R.Graham, J. (1995). *Solaris 2.x, Internals and Architecture*. McGraw Hill Inc.

[Saulpaugh et Mirho, 1999] Saulpaugh, T. et Mirho, C. (1999). *Inside the JavaOS Operating System*. The Java Series. Addison-Wesley.

[Schroeder et Saltzer, 1972] Schroeder, M. D. et Saltzer, J. H. (1972). A hardware architecture for implementing protection rings. *Communications of the ACM*.

[Shah, 2001] Shah, J. (2001). Alternate thread library – new in the solaris 8 operating environment. *Solaris, Technical Articles and Tips*.

[Soper et al., 2002] Soper, P., Donald, P., Houldsworth, R., Lea, D., Kuck, N., Mehta, M., Sabin, M., Tullmann, P., Unitis, D., Webster, M., et Yogaratnam, K. (2002). *JSR-121 Public Review Draft*. Sun Microsystems.

[von Eicken et al., 2001] von Eicken, T., Chang, C.-C., Czajkowski, G., Hawblitzel, C., Hu, D., et Spoonhower, D. (2001). J-kernel : a capability-based operating system for java. *Secure Internet programming : security issues for mobile and distributed objects*.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399