Intégration des threads temps-réel POSIX sous Java

Nicolas Le Sommer, Frédéric Guidec

RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

Intégration des *threads* temps-réel POSIX sous Java

Nicolas Le Sommer, Frédéric Guidec

Laboratoire VALORIA
Université de Bretagne Sud - Vannes
Nicolas.Lesommer@univ-ubs.fr, Frederic.Guidec@univ-ubs.fr

Résumé

La plupart des machines virtuelles Java (JVM : *Java Virtual Machine*) actuelles diffèrent dans la manière selon laquelle elles supportent la programmation concurrente. Un même programme concurrent exécuté aujourd'hui sur différentes machines virtuelles se comportera souvent différemment avec chacune d'entre elles. L'environnement Java que nous proposons donne au programmeur un contrôle précis de l'ordonnancement des *threads* Java s'exécutant sur une plate-forme Linux. Dans cet environnement les *threads* Java sont mis en œuvre sous la forme de *threads* POSIX. L'ordonnancement de chaque *thread* Java est assuré par le noyau du système d'exploitation, et peut être soumis à n'importe laquelle des trois politiques d'ordonnancement définies par le standard POSIX 1003.1c.

1. Introduction

La popularité du langage Java en tant que support privilégié pour le développement d'applications portables et communicantes ne cesse de croître. Nombre de programmeurs hésitent pourtant encore à utiliser ce langage pour développer des programmes concurrents sur la base de processus légers (*threads*), et ce en particulier si l'ordonnancement de ces *threads* est censé respecter certaines contraintes temporelles. Cette réticence à utiliser Java pour la programmation temps-réel résulte pour une bonne part du manque de précision dans les spécifications du langage et de la machine virtuelle [3] au sujet de la politique d'ordonnancement qu'il convient d'appliquer aux *threads*. En effet, bien qu'un niveau de priorité puisse en principe être attribué à chaque *thread* Java, le document de spécification du langage se contente de préciser « qu'en cas de compétition [...] les *threads* ayant la priorité la plus élevée *devraient en général* être exécutés de préférence aux *threads* ayant un niveau de priorité moindre ».

Il résulte de ce manque de ligne directrice dans les spécifications de Java que, lors de la mise en œuvre d'une nouvelle JVM (*Java Virtual Machine*), les concepteurs ont toute liberté pour implémenter la politique d'ordonnancement qu'ils désirent. Un même programme concurrent exécuté aujourd'hui sur différentes machines virtuelles se comportera souvent différemment avec chacune d'entre elles.

Nous pensons que cette situation est extrêmement dommageable à l'utilisation de Java comme support pour la programmation d'applications concurrentes élaborées, et qu'elle interdit presque totalement le développement de programmes temps-réel. C'est pourquoi nous proposons une plate-forme Java s'appuyant sur les *threads* temps-réel POSIX (alias *pthreads* [5]). Cette plate-forme permet la programmation de *threads* dont le comportement temporel est prévisible car conforme aux exigences du standard POSIX 1003.1c. Elle permet en outre de spécifier explicitement la politique d'ordonnancement qu'on souhaite voir appliquer à chacun des *threads* d'un programme concurrent.

2. Contexte de ce travail

Le travail rapporté dans cet article s'inscrit dans un projet plus vaste, le projet RASC[1] (*Resource-Aware Software Components*), dont l'un des objectifs est le développement d'une plate-forme expérimentale capable d'héberger des composants logiciels tout en supervisant les ressources utilisées par ces composants au cours de leur exécution. Dans cette optique, il est intéressant de pouvoir maîtriser l'ordonnancement de programmes d'application Java s'exécutant en concurrence sur une même plate-forme d'accueil. Ainsi, bien que le développement de programmes temps-réels ne soit pas notre objectif premier, nous nous appuyons sur un standard dédié au temps-réel afin de doter Java de mécanismes d'ordonnancement élaborés.

[1] http ://www.univ-ubs.fr/valoria/Actions/RASC

Notre démarche privilégie la simplicité de mise en œuvre et la portabilité de la plate-forme produite. Cette dernière n'a donc nullement la prétention de répondre aux contraintes définies par le groupe RTJ (*Real-Time for Java Expert Group*), qui a produit pour le compte du NIST (*National Institute of Standards and Technology*) les spécification d'une extension de l'API Java pour le temps réel [7].
Notre plate-forme se distingue également des produits PERC (*Portable Executive for Reliable Control* [6, 4]) et GVM (extension de Java s'appuyant sur le système d'exploitation OSKit [1, 4, 2]), qui s'appuient tous deux sur des extensions du langage Java, et nécessitent de ce fait des modifications conséquentes de la machine virtuelle Java.

3. Caractéristiques des *threads* POSIX 1003.1c

Une mise en œuvre de *threads* temps-réel conforme au standard POSIX 1003.1c est aujourd'hui disponible sur la plupart des systèmes d'exploitation. Une telle mise en œuvre permet d'exploiter simultanément jusqu'à trois politiques d'ordonnancement, dont deux au moins (les politiques `SCHED_FIFO` et `SCHED_RR`) conviennent à la mise en œuvre de programmes temps-réel. Le choix de la troisième politique d'ordonnancement (`SCHED_OTHER`) est laissé à la discrétion de l'équipe chargée de son implémentation. Il s'agit en général d'un ordonnancement en temps partagé capable de supporter les tâches non temps-réel du système.
L'ordonnancement `FIFO` est un ordonnancement simple à base de tranches de temps, avec une politique d'accès de type premier arrivé - premier servi s'appliquant entre *threads* ayant même niveau de priorité. Un *thread* soumis à l'ordonnancement `FIFO` s'exécute jusqu'à ce qu'il soit bloqué, qu'il soit préempté par un *thread* de priorité supérieure, ou qu'il passe la main explicitement. Un *thread* `FIFO` qui a été préempté par un *thread* de priorité supérieure reste en tête de la liste des *threads* ayant même niveau de priorité que lui et reprend son exécution dès que tous les *threads* de priorités supérieures ont terminé leur exécution, ou sont temporairement bloqués. Lorsqu'un *thread* `SCHED_FIFO` devient éligible, il est inséré à la fin des la liste des *threads* ayant même niveau de priorité que lui.
L'ordonnancement *Round-Robin* est une amélioration assez simple de l'ordonnancement `FIFO`. Tout ce qui a été décrit pour l'ordonnancement `SCHED_FIFO` s'applique également à l'ordonnancement `SCHED_RR`, si ce n'est que chaque *thread* ne dispose que d'une tranche temporelle limitée (quantum de temps) pour son exécution. Un *thread* dont la durée d'activité dépasse le quantum de temps autorisé sera préempté par un autre *thread* de priorité équivalente à la sienne, et ne pourra poursuivre son activité qu'une fois que tous les autres *threads* éligibles auront pu s'exécuter (*i.e.*, il est placé à la fin de la liste des *threads* ayant même niveau de priorité que lui). Dans un contexte de programmation temps-réel, l'ordonnancement *Round-Robin* permet de garantir une certaine équité entre *threads* ayant même niveau de priorité.
Dans la plupart des cas la troisième politique d'ordonnancement mise en œuvre dans les distributions de *threads* POSIX est destinée aux traitement des *threads* n'ayant pas de contraintes temporelles spécifiques. Sur les plates-formes Linux actuelles il s'agit d'une politique de temps partagé qui, à la différence des politiques `FIFO` et *Round-Robin*, s'appuie sur un niveau de priorité dynamique affecté à chaque *thread* (alors que l'ordonnancement FIFO ou *Round-Robin* ne tient compte que du niveau de priorité fixé statiquement pour chaque *thread*). La priorité dynamique est calculée en tenant compte de la valeur de « gentillesse » de chaque *thread* (valeur fixée par exemple à l'aide de la commande *nice*). Elle est en outre incrémentée à chaque fois qu'un *thread* éligible par l'ordonnanceur n'est pas choisi par celui-ci. Cette approche assure une progression relativement équitable des *threads* non temps-réel, qui demeurent cependant préemptibles par tout *thread* temps-réel éligible.
Il faut noter que, pour que des *threads* puissent bénéficier d'un ordonnancement réellement prévisible, celui-ci doit être assuré au niveau du noyau du système plutôt que dans l'espace utilisateur. C'est la raison pour laquelle la plupart des systèmes d'exploitation récents (Linux, Solaris, VxWorks, etc.) proposent la mise en œuvre de *threads* sous la forme de tâches gérées par le noyau (*kernel-space threads*). Sur les plates-formes Linux actuelles, les *threads* sont en outre mis en œuvre dans l'espace du noyau du système sous la forme de processus lourds. Il résulte de cette dernière caractéristique que les *threads* Linux sont soumis aux même contraintes et aux mêmes conditions d'ordonnancement que les autres tâches gérées par l'ordonnanceur du noyau.
Il faut également noter que, sous Linux, les ordonnancements temps-réel SCHED_FIFO et SCHED_RR ne sont applicables qu'aux threads bénéficiant des privilèges d'administration (*super-user*).

4. Intégration des *threads* POSIX sous Java

Notre plate-forme à été conçue sur la base de la machine virtuelle Kaffe (version 1.0.6) distribuée par la compagnie *TransVirtual Technology*. D'après la documentation délivrée avec Kaffe, ce produit est censé pouvoir exploiter les *threads* POSIX lorsque le système sous-jacent offre cette possibilité. Cependant, lorsque Kaffe est installé sur une plate-forme Linux de manière à utiliser les *threads* POSIX, les *threads* Java ainsi mis en œuvre n'exploitent en réalité que la politique d'ordonnancement par défaut (`SCHED_OTHER`). Les deux véritables politiques d'ordonnancement temps-réel des *threads* POSIX 1003.1c (`SCHED_FIFO` et `SCHED_RR`) demeurent ainsi inaccessibles au programmeur Java.
À la différence de l'environnement Kaffe standard, notre plate-forme permet de créer et de manipuler des *threads* Java en tant que *threads* POSIX à part entière. La classe `PosixThread` (en héritant de la classe standard `java.lang.Thread`) permet au programmeur de choisir et de modifier à volonté la politique d'ordonnancement souhaitée pour chaque *thread* Java.
L'exemple ci-dessous illustre la création d'un objet Java instanciant la classe `PosixThread`. Dans cet exemple le type d'ordonnancement et le niveau de priorité sont fixés avant le lancement du *thread*.

```
PosixThread pt = new PosixThread() ; // Création de l'objet thread
pt.setSchedulingPolicy(SCHED_RR) ; // Choix de l'ordonnancement
pt.setPriority(8) ; // Choix du niveau de priorité
pt.start() ; // Démarrage du thread
```

5. Résultats expérimentaux

Nous présentons dans cette partie les résultats d'expériences effectuées sous Linux avec la version standard de Kaffe-1.0.6, et discutons les résultats obtenus. Nous montrons également qu'avec notre plate-forme les trois politiques d'ordonnancement du standard POSIX 1003.1c sont bien accessibles au programmeur Java, et donne lieu à des comportements prévisibles de la part de programmes concurrents.

5.1. Implémentations standard de Kaffe

Les figures 1 et 2 montrent l'activité observée au cours de l'exécution d'un programme composé de trois *threads* réalisant tous trois une même série de calculs. Dans cette expérience le *thread* n° 1 et le thread n° 3 se sont vu attribuer une priorité de niveau 5, et le *thread* n° 2 une priorité de niveau 7. En outre l'exécution du *thread* n° 2 était différée afin de commencer 800 millisecondes après celle du *thread* n° 1.

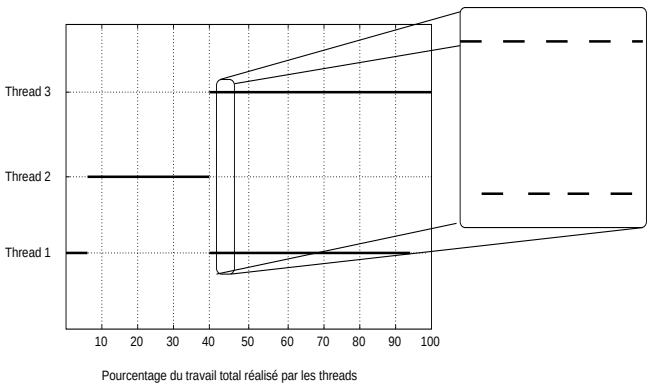

FIG. 1 – Ordonnancement utilisant les « *green threads* »

La figure 1 illustre le comportement observé avec une configuration de Kaffe implémentant des *green threads* (processus légers intégrés au processus de la JVM, cette dernière assurant elle-même l'ordonnancement des *threads* Java). La figure 2 correspond à une autre configuration de Kaffe, dans laquelle les

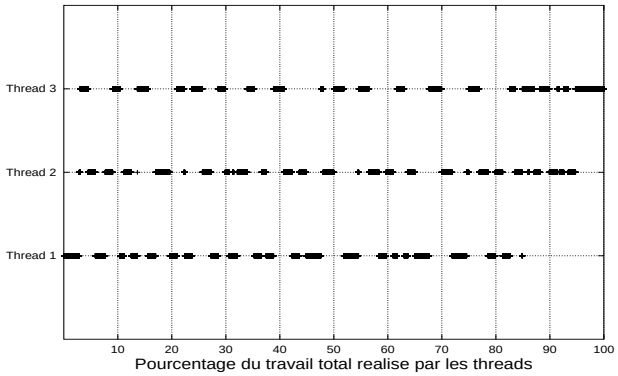

FIG. 2 – Ordonnancement utilisant les *threads* natifs

threads Java sont mis en œuvre sous la forme de *threads* natifs du système (*i.e.*, *threads* POSIX soumis à la politique par défaut `SCHED_OTHER`).

Nous ne discutons ici que les résultats obtenus au cours de tests portant sur la plate-forme Kaffe-1.0.6. Il est toutefois utile de préciser que les mêmes expériences menées avec la plate-forme Java JDK-1.2.2 ont produit des résultats en tous points similaires à ceux rapportés ici.

Dans la figure 1 on constate qu'une fois lancé, le *thread* prioritaire conserve la main jusqu'au terme de son calcul. On observe également un entrelacement de l'activité des *threads* de même priorité. Ce comportement est conforme aux spécifications de la JVM : le *thread* de priorité supérieure est bien exécuté de préférence au *thread* de priorité moindre. Il faut cependant garder à l'esprit que l'ordonnancement observé ici est l'ordonnancement relatif aux *threads* internes à la JVM, qui est elle-même perçu comme un unique processus par le système sous-jacent. Cet ordonnancement assuré par la JVM ne tient bien sûr aucun compte de l'activité des autres tâches présentes dans le système. De son côté, l'ordonnanceur du noyau n'a connaissance ni du nombre de *threads* participant au programme Java, ni de leur priorité respective. Un *thread* Java de priorité maximale peut donc fort bien se trouver interrompu par une autre tâche du système sur décision de l'ordonnanceur du noyau. En conséquence une plate-forme Java implémentant des *green threads* ne saurait supporter l'exécution de programmes Java temps-réel.

Avec la figure 2 on constate aisément que, dans une configuration de Kaffe s'appuyant sur les *threads* natifs du système, le niveau de priorité attribué à chaque *thread* Java n'est absolument pas pris en compte. Du point de vue de l'ordonnanceur, les *threads* Java sont des *threads* POSIX soumis à la politique `SCHED_OTHER` : ce sont des tâches de priorité minimale, les niveaux de priorité supérieurs étant réservés aux *threads* temps-réel. Cette configuration de Kaffe ne convient donc pas non plus à la mise en œuvre de programmes temps-réel. On peut ajouter qu'elle ne répond pas aux spécifications de Java, puisqu'aucune préférence n'est accordée aux *threads* de priorité supérieure.

5.2. Implémentation de Kaffe sur les *threads* POSIX

Avec notre mise en œuvre de Kaffe sur les *threads* POSIX, il est facile de reproduire les deux types de comportement décrits plus haut. Pour obtenir un ordonnancement semblable à celui de la figure 3 il suffit de soumettre le *thread* n° 2 à la politique `SCHED_FIFO`. L'ordonnancement de la figure 2 s'obtient quant à lui en soumettant les deux *threads* du programme à un ordonnancement de type `SCHED_OTHER`.

Mais notre plate-forme permet aussi de produire de véritables schémas d'ordonnancement temps-réel, schémas qui ne pourraient être obtenus avec aucune des configurations standard de Kaffe et de JDK. Ainsi, la figure 3 montre l'activité observée au cours de l'exécution d'un programme semblable à celui utilisé dans les expériences précédentes, excepté que les *threads* n° 1 et n° 3 sont soumis à un ordonnancement de type `SCHED_RR`, et le *thread* n° 2 à un ordonnancement de type `SCHED_FIFO`.

Le comportement obtenu est conforme à celui que l'on peut attendre d'un programme utilisant des *threads* POSIX : une fois lancé, le *thread* prioritaire et d'ordonnancement FIFO s'exécute jusqu'à son terme. Les deux autres *threads* se partagent le reste du temps d'activité de manière équitable par tranches de temps bornées, conformément au principe de la politique *Round-Robin*.

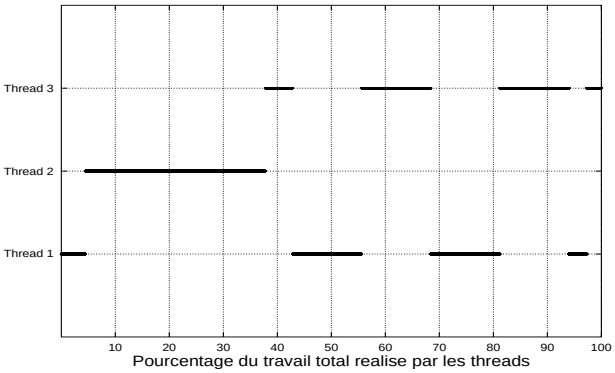

FIG. 3 – Ordonnancement temps-réel

Le problème du ramasse-miettes

À ce stade il importe de souligner que l'ordonnancement illustré par la figure 3 reflète l'activité réelle observée sur l'ensemble du système au cours du test. En effet, avec un système d'exploitation tel que Linux implémentant les *threads* POSIX au niveau du noyau, les *threads* temps-réel (soumis à l'une des politiques `SCHED_FIFO` et `SCHED_RR`) sont prioritaires par rapport à toute autre activité dans le système. Cette propriété permet d'envisager le développement de programmes Java temps-réel, dans lesquels les *threads* Java pourront être soumis à des contraintes temporelles telles que temps d'exécution ou temps de réponse bornés, périodicité, etc.
Pour atteindre cet objectif il faudrait cependant pouvoir éviter que le ramasse-miettes de la JVM se déclenche alors qu'un *thread* temps-réel est lancé dans une opération dont la durée d'exécution est critique. Pour ce faire nous envisageons d'adjoindre à notre plate-forme des mécanismes permettant aux *threads* Java temps-réel d'exercer un contrôle précis sur l'activité du ramasse-miettes.

6. Conclusion

En mettant en œuvre l'outil Kaffe 1.0.6 sur les *threads* POSIX 1003.1c, nous donnons au programmeur la possibilité de manipuler des *threads* Java en tant que véritables *threads* temps-réels susceptibles d'être soumis à un ordonnancement de type FIFO ou *Round-Robin*. Notre plate-forme s'obtient aisément par application d'un simple *patch* logiciel lors de la compilation de la machine virtuelle de Kaffe. Elle n'a jusqu'à présent été testée que sur des systèmes de type Linux, mais devrait pouvoir être portée sans problème sur tout autre système d'exploitation offrant une mise en œuvre des *threads* POSIX 1003.1c au niveau du noyau.

Bibliographie

1. Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the Design of Java Operating Systems. In *Annual Technical Conference*. USENIX, June 2000.
2. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit : A Substrate for Kernel and Language Research. *ACM Symposium on Operating Systems Principles*, October 1997.
3. James Gosling, Bill Joy, and Guy Steel. *The Java Language Specification*. Addison-Wesley, 1996.
4. M. Teresa Higuera-Toledano, Valérie Issarny, Michel Banâtre, Gilbert Cabillic, Jean-Philippe Lesot, and Frédéric Parain. Java Embedded Real-Time Systems : an Overview of Existing Solutions. In *Proceedings of ISORC 2000 – The 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing, Newport Beach (CA), USA*, March 2000.
5. Bradford Nichols and Dick Buttlar. *Pthreads Programming*. O'REILLY, 1998.

6. K. Nilsen. Adding Real-Time Capabilities to Java. *Communication of the ACM*, 41(6) :49–56, June 1998.
7. The Real-Time for Java Expert Group. *The Real-Time Specification for Java*. http ://www.rtj.org.