Noyau de concurrence par moniteur pour Java ou C#

pour une autre sémantique plus fiable et plus performante

Claude Kaiser, Jean-François Pradat-Peyre

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
{kaiser, peyre}[at]cnam[dot]fr
http://quasar.cnam.fr/

29 avril 2006

Résumé. Avec l'extension de Java et C# et de leur emploi pour programmer des applications embarquées mobiles ou temps réel puis pour introduire des processus concurrents dans ces applications, se développe l'idée d'écrire aussi les systèmes d'exploitation et les plates-formes dans ces langages. Nous comparons deux politiques possibles pour gérer le blocage et le réveil des "threads" qui partagent un objet sous contrôle de la structure de "moniteur". Cela renvoie à deux sémantiques différentes pour le moniteur. L'une de celles-ci entraîne des interblocages pour des algorithmes qui, pour d'autres implantations, sont fiables ou à des commutations de contexte superflues et coûteuses. L'autre sémantique n'a pas ces défauts. Il se trouve que Java et C# ont choisi la plus mauvaise de ces politiques. Nous explicitons cela sur des exemples. Nous montrons aussi la limitation de performance introduite par le choix de Java d'une file d'attente unique, implicite, par objet monitoré. Enfin nous utilisons une mesure de complexité des algorithmes concurrents pour comparer des implantations avec ces deux sémantiques. Nous proposons aux développeurs de systèmes d'exploitation des éléments pour améliorer la qualité du contrôle de concurrence en reconsidérant les choix d'implantation du moniteur.

Abstract. With the development of embedded and mobile systems, Java and C#, which are widely used for application programs, are also considered for implementing systems kernel or application platforms. It is the aim of this paper to exemplify some subtle deadlocks and some inefficiencies that may result from the process queuing and awaking policy which has been chosen for implementing the monitor concept in these languages. Several examples are given, two of them showing unforeseen deadlocks, another one showing the limitation of the implicit unique queue of Java. We compare some solutions with a complexity measure for concurrent programs. Last we propose some iplementation changes for an operating system providing more robust Java or C# monitor implementations.

1. Introduction

Avec l'extension de Java et C# et de leur emploi pour programmer des applications embarquées mobiles ou temps réel puis pour introduire des processus concurrents dans ces applications, se développe l'idée d'écrire aussi les systèmes d'exploitation et les plates-formes dans ces langages ou dans des sous-ensembles spécialisés de ces langages (Java Card, RTJava). Notre article vise à analyser quelques aspects du contrôle de concurrence tel qu'il est réalisé aujourd'hui dans ces langages et qui mériteraient d'être améliorés. Nous proposons aux développeurs de tels systèmes des éléments pour reconsidérer les choix d'implantation. Notre analyse porte sur Java et est valable aussi pour C#.

1.1. Contrôle de concurrence avec le mécanisme du moniteur

Pour les langages qui ont choisi le moniteur [Hoare 1974] comme mécanisme de concurrence, se pose le classique problème du choix de la sémantique associée au blocage, au réveil et à l'activation des processus (appelés « threads » en Java et en C#) utilisant un moniteur. Plusieurs sémantiques sont possibles et certaines ont été étudiées et expérimentées. On en trouve une étude comparative dans [Buhr 1995]. Depuis, plusieurs nouveaux langages dont Ada [Burns 1995], Java [Lea 1997] et C# [ ], ont choisi le moniteur. De même depuis Pilot réalisé avec le langage Mesa [Lampson 1980], plusieurs systèmes ont utilisé le moniteur.

Si on analyse les diverses implantations du moniteur, on y trouve un socle commun garantissant une exclusion mutuelle d'accès à des variables partagées pendant une section critique de code (définie par « synchronized » en Java, par « protected » en Ada, par « monitor.Enter» et « monitor.Quit » en C#) . Dans la terminologie des langages à objets, un objet partagé dont l'accès est contrôlé par un moniteur est accessible uniquement par des méthodes qui s'exécutent en exclusion mutuelle (encore que Java et C# autorisent aussi de ne mettre qu'un bloc d'instructions en exclusion mutuelle). Ce contrôle est assuré par un verrou d'accès à cet objet monitoré.

Les implantations du moniteur diffèrent par la sémantique choisie pour bloquer, réveiller et activer les processus qui l'utilisent, tout en restant compatible avec cette exclusion mutuelle primitive. Cinq aspects permettent de préciser la sémantique.

1) Le style de programmation utilisé pour exprimer les conditions de synchronisation et les actions dérivées. Il peut être explicite avec les instructions classiques, « wait » et « signal » proposées par Hoare (sous la forme « wait, notify, notifyAll » en Java, «Wait, Pulse, PulseAll » en C#). Il peut être implicite à la suite de Kessels [Kessels 1977] avec une expression booléenne formant barrière ou avec des gardes contrôlant l'exécution d'une méthode (appelée « entry » en Ada).

2) Le nombre de blocages ou de réveils possibles dans une méthode. Java et C# ne donnent pas de limite. Ada ne bloque qu'au début et ne signale qu'à la fin d'une méthode.

3) Le nombre de files d'attente associées à l'objet monitoré. Hoare a proposé un type condition permettant autant de files d'attente que de conditions, Ada a gardé cette liberté. Java et C# n'associent qu'une seule file, implicite, par objet monitoré. Java J2SE/JDK 1.5 permet de construire un moniteur avec un verrou et des conditions, en utilisant la classe concurrent.locks.condition, mais ce n'est plus une construction primitive du langage.

4) La politique lorsque le réveil d'un processus conduit à avoir deux candidats pour l'accès exclusif à l'objet. La plupart des implantations de systèmes, de même que les langages Java et C#, ont choisi de laisser le moniteur au processus élu (on économise une commutation de contexte). C'est la politique « signal and continue ».

5) Le sort des processus réveillés. La plupart des implantations (Mesa, Java, C#) les mettent dans la file des processus prêts, ce qui permet de banaliser leur traitement, mais cela permet à un nouveau processus d'obtenir le moniteur avant un processus réveillé et, partant, oblige tout processus réveillé à réévaluer la condition de blocage. Le choix de Ada donne priorité aux réveillés sur tout autre processus, suivant en cela la politique du schéma des sémaphores privés [Dijkstra 1968]. Une optimisation peut être associée à cette politique de réveil prioritaire en demandant au processus signalant et s'apprêtant à libérer le moniteur d'exécuter, avant de le quitter, les méthodes appelées par les processus réveillés et de leur retourner les résultats ; ceci permet de ne pas commuter les processus réveillés pour leur donner accès l'un après l'autre à l'objet monitoré, de supprimer l'indéterminisme des commutations, et d'évaluer le temps maximal pris par ces exécutions (WCET, « worst case execution time »).

Cette optimisation prolonge celle déjà faite en Ada 83 pour des rendez-vous entre tâches clientes et une tâche serveur cyclique lorsque cette dernière était utilisée pour simuler un moniteur (optimisation de Habermann et Nassi).

Dans la suite de cet article on appellera réveil banalisé la sémantique choisie pour Java et C# et réveil prioritaire la sémantique choisie pour Ada.

1.2. Choix sémantiques pour le mécanisme de moniteur dans Java et C#

Les choix faits pour Java et C# sont très voisins. La programmation des attentes et réveils des processus est explicite ; on peut en mettre autant qu'on veut. Il n'y a qu'une seule file d'attente, implicite, par objet monitoré. La politique appliquée au réveil est « signal and continue ». Les processus réveillés sont envoyés dans la file des processus prêts et se retrouveront banalisés lors de la compétition pour accéder aux objets monitorés ; le verrou d'un objet monitoré est visible par tous les processus prêts, ce qui ne garantit pas aux processus réveillés de l'obtenir en priorité. Il faut donc en tenir compte dans le code de chaque processus (l'attente doit être programmée dans une boucle contrôlée par la condition de blocage et tout processus réveillé doit commencer par réexaminer cette condition). Cette acquisition du verrou ne facilite donc pas les optimisations qui résultent d'un accès favorisé au moniteur par les processus réveillés. Enfin la sémantique de réveil banalisé doit être prise en compte lorsqu'on examine globalement un algorithme concurrent pour le valider.

Java J2SE/JDK 1.5 qui, permet de construire un moniteur avec un verrou et des conditions, garde les mêmes choix sémantiques ; il en est de même pour C#, qui en est proche.

On va examiner les conséquences de ces choix sémantiques sur la fiabilité des algorithmes, sur le nombre de commutations de processus et donc sur l'efficacité des exécutions, lorsque l'on se place dans le contexte de Java, avec une seule condition implicite et une seule file d'attente par objet monitoré. On prolongera notre réflexion lorsque l'on peut simuler un moniteur avec plusieurs conditions. On indiquera enfin un outil d'analyse de la complexité des algorithmes systèmes.

2. Comparaison des politiques de réveil banalisé et de réveil prioritaire

2.1. Fiabilité de l'évitement de l'interblocage ou de la famine.

Des algorithmes reconnus par ailleurs comme fiables ne le sont plus à cause de la politique actuelle de Java et C#, alors qu'ils le redeviennent si l'on adopte la politique de réveil prioritaire. Nous présentons ici deux exemples en Java, dont les programmes sont donnés en annexe. Ces implantations sont réalisées en utilisant un seul objet monitoré, donc une seule file d'attente implicite.

Notre premier exemple concerne une solution du paradigme du dîner des philosophes [Dijkstra 1971], celle où les baguettes sont allouées globalement. On sait que cette solution est fiable mais non équitable. Une solution équitable a été donnée [Courtois 1977] en enregistrant le nom de chaque philosophe dont la demande a été refusée et en ajoutant comme contrainte qu'un philosophe ne peut être servi si son voisin de gauche a eu un refus. La sémantique du moniteur de Java laisse s'installer une chaîne circulaire de contraintes, donc un interblocage. La sémantique de réveil prioritaire n'autorise pas cette chaîne circulaire et ne conduit pas à interblocage. C'est l'exemple en annexe1. Ce paradigme est une épreuve sévère pour le langage Java. Nous en avons proposé une autre solution [Kaiser 2006], où se manifeste encore le même défaut de fiabilité lors de son implantation en Java.

Notre deuxième exemple est l'utilisation d'une agora de rendez-vous avant une communication entre pairs (« peer to peer cooperation») [Kaiser 2003]. Chaque pair se fait

connaître en appelant l'agora et y attend l'arrivée d'un second pair. Quand deux pairs sont au rendez-vous dans l'agora, chacun reçoit alors le nom de son partenaire et quitte l'agora pour laisser celle-ci à d'autres appelants. Les pairs appariés communiquent alors directement. La sémantique du moniteur de Java n'empêche pas d'autres pairs de perturber le rendez-vous et l'on peut avoir des appariements incorrects qui conduisent à interblocage quand les pairs appariés s'appellent pour communiquer. La sémantique de réveil prioritaire donne un résultat correct. Voir l'annexe 2.

Rendre ces algorithmes fiables malgré la sémantique de réveil banalisé suppose que l'on sache trouver une programmation défensive, ce qui n'est pas toujours évident et demande souvent beaucoup de tatonnements.

2.2. Nombre de commutations de contextes des processus.

Le deuxième exemple, celui du rendez-vous entre pairs a été implanté avec les deux sémantiques (on a corrigé le programme faux pour le rendre fiable par une programmation défensive) et simulé en Ada (Java ne permet pas de simuler simplement la sémantique de réveil prioritaire, alors que la sémantique de Java peut se simuler en Ada). La simulation lance N processus et chacun fait R demandes de pair. On a pu compter les nombres de fois où un pair est mis dans la file d'attente de l'objet monitoré, ce qui donne une idée du nombre de commutations de processus et du coût d'exécution qui en découle. De par l'algorithme, un processus sur deux doit attendre son pair, ce qui donne un seuil minimum. Ce minimum n'est pas dépassé par la politique de réveil prioritaire, alors que la politique de réveil banalisé le nombre de commutations explose (proportionnellement au carré du total de requêtes).

nombre de processus	requêtes par processus	seuil minimum	réveil prioritaire	réveil banalisé
3	6	9	9	13
6	6	18	18	58
9	6	27	27	121
18	6	54	54	486
36	6	108	108	2 084

Tableau 1. Comparaisons du nombre de commutations de contexte

Cet exemple, présenté Tableau 1, montre non seulement le coût mais aussi l'indéterminisme résultant du choix actuel de Java. En effet le nombre de commutations de contexte varie aussi selon la longueur et la gestion de la file des processus prêts, donc selon la charge.

2.3. Proposition pour une implantation plus fiable et plus efficace

La modification la plus simple consiste à garder une seule file implicite par objet monitoré mais à la gérer avec la sémantique de réveil prioritaire. Cela permet de fiabiliser la concurrence dans Java en réduisant le nombre de cas où la programmation défensive est nécessaire. Cette sémantique implique la présence d'une file des processus bloqués, d'une file prioritaire des processus réveillés et d'une file de processus prêts.

C'est l'amélioration la plus simple et la plus utile, qui présente plusieurs avantages

1. Compatibilité ascendante des algorithmes : les algorithmes corrects le demeurent.
2. Meilleure fiabilité du code : on a davantage d'algorithmes fiables, sans code défensif.
3. Meilleures performances : il y a moins de commutations de processus.
4. Plus de déterminisme : la variabilité entre les exécutions possibles est moindre.

La solution actuelle qui consiste à transférer les processus réveillés dans la file d'attente des processus prêts est facile à implanter et c'est sans doute une des raisons de son choix. La solution avec réveil prioritaire n'est pas difficile à implanter non plus. Elle repose sur des

techniques bien connues et qui, en outre, ont été développées pour le langage Ada depuis sa révision de 1995.

La technique bien connue sous le nom « passing the baton » [Andrews 1991] permet de passer l'exclusion mutuelle d'un processus à un autre de la file des processus réveillés, sans rendre le verrou accessible à tous (ni surtout aux processus de la file des processus prêts).

Pour faire la même optimisation qu'en Ada, c'est un peu plus compliqué car Ada a un style d'écriture qui la facilite : il n'y a ni blocage ni réveil dans le code d'une méthode et les gardes sont écrites en avant du code d'une méthode, ce qui permet leur évaluation avant et après l'exécution de la méthode. L'optimisation pratiquée est comparable à une exécution de procédure à distance. Au lieu d'activer un processus réveillé, on demande au processus qui va quitter le moniteur de réaliser à distance chaque appel de méthode que souhaite faire chaque processus réveillé, de leur passer le résultat de l'appel puis de les mettre alors seulement dans la file des processus prêts pour continuer l'exécution des instructions qui suivent ce retour de résultat. Il faut donc pouvoir récupérer dans la pile d'un processus bloqué le nom de la méthode appelée, les paramètres effectifs de l'appel d'une méthode par un processus bloqué, les noms des paramètres résultats, et l'adresse de retour de cette méthode dans le code de ce processus. En Java, une méthode peut avoir des « wait() » à n'importe quel endroit du code et l'exécution à distance doit pouvoir la relancer après un de ces « wait() ». Il faut aussi avoir accès au compteur ordinal du processus réveillé pour connaître cette adresse. Si l'exécution s'arrête sur un nouveau « wait() », il faut l'intercepter pour que ce soit le processus réveillé et non le processus réveilleur qui soit bloqué et renvoyé vers la file des processus bloqués. De même tout signal par « notify() » (resp. « notifyAll() ») doit contribuer à faire passer un (resp. tous) processus de la file des processus bloqués dans la file des processus réveillés.

3. Comparaison de programmation avec une seule ou plusieurs conditions

3.1. Utilisation d'un objet monitoré avec une condition implicite

Notre troisième exemple concerne l'allocation de ressources à partir d'un stock de ressources banalisées. Implanter une politique de service avec priorité ou sans famine n'est pas simple avec une seule file d'attente implicite. Il faut ajouter des structures de données et faire défiler un après l'autre les processus bloqués pour pouvoir retenir le bon. Pour notre exemple, nous avons choisi une solution très simple qui permet de limiter, sinon de supprimer, la famine quand les files sont gérées à l'ancienneté. Lorsqu'une demande ne peut être honorée, le demandeur est mis en attente et toute autre requête est refusée tant que ce demandeur privilégié n'est pas servi. Ce cas, aussi simple, est inefficace lorsqu'il est programmé avec une seule file d'attente, car celle-ci doit contenir le client privilégié et tous les autres clients demandeurs. À chaque retour significatif de ressource, il faut les réveiller tous pour n'en retenir qu'un au mieux. La programmation est donnée en annexe 3.

3.2. Utilisation d'un objet monitoré avec une deux conditions explicites

Une file d'attente unique et implicite par objet monitoré limite le pouvoir d'expression de la concurrence. Les concepteurs de Java l'ont bien compris et, dans Java J2SE/JDK 1.5, ils ont introduit une classe concurrent.locks.condition qui permet d'utiliser un moniteur avec plusieurs conditions d'attente, chacune étant associée à une file d'attente, comme dans la proposition initiale de Hoare. Mais ce moniteur n'est pas une structure primitive et il faut le construire avec les composants de sa structure :

- exclusion mutuelle par déclaration et utilisation d'un verrou encadrant le code critique
  mutex.lock() ; …… ….. ; mutex.unlock() ;
- déclaration des objets condition par condition = lock.newCondition() ;

- utilisation par condition.await() ; ou par condition.signal() ;

L'annexe 4 présente une implantation de l'allocation de ressources utilisant deux objets condition (et deux files d'attente). Elle donne une solution plus efficace, car elle permet de ne réveiller qu'un seul processus, celui qui est privilégié.

Cette construction ad hoc n'impose pas de structuration de haut niveau comme le permet la déclaration d'un moniteur, et il faut une discipline de programmation (comme pour les sémaphores ou les événements) pour bien l'encapsuler dans un objet que l'on veut monitorer. Par ailleurs laisser coexister plusieurs mécanismes de synchronisation (et donc des sémantiques diverses) n'est pas un bon choix d'ingénierie du logiciel, car cela entraîne des confusions ou des erreurs quand on mélange les mécanismes ou quand on prend l'un pour l'autre. Des exemples passés de systèmes qui ont eu pour cette raison de grosses difficultés de mise au point sont là pour en témoigner. Et même averti, un programmeur peut se tromper. En reprenant l'exemple d'allocation de ressource pour lui donner, dans l'annexe 4, une implantation avec deux files d'attente, nous avions oublié d'enlever « synchronized », ce qui donnait un interblocage avec « lock ». Et pourtant notre code était bien valide lorsque programmé dans un autre langage (en l'occurrence Ada) ! Nous avons mis quelque temps à trouver cette erreur évidente.

3.3. Nombre de commutations de contextes des processus.

Ces deux solutions d'allocation de ressources ont été implantées avec les deux sémantiques et simulées en Ada (Java ne permet pas de simuler simplement la sémantique de réveil prioritaire, alors que la sémantique de Java peut se simuler en Ada). La simulation lance N processus et chacun fait R demandes de ressources. Dans la première solution avec une file unique et implicite, on a compté les nombres de fois où un processus est mis dans la file d'attente implicite de l'objet monitoré, ce qui donne une idée du nombre important de commutations de processus (de 10 à 25 fois par requêtes dans notre simulation) et du coût d'exécution qui en découle. dans la seconde solution avec deux files, une requête n'est mise qu'une fois au plus en attente dans chaque file. Voir Tableau 2 .

Nombre de processus	Nombre de requêtes	Attentes file implicite	Attentes file explicite des ressources	Attentes file explicite du privilège
50	500	5 400	480	479
100	1 000	18 000	980	979
200	2 000	50 000	1 980	1 979

Tableau 2. Comparaison des attentes pour un objet monitoré avec une ou deux conditions

3.4. Proposition pour une programmation plus expressive et donc plus fiable

Nous pensons que le pouvoir d'expression serait meilleur si Java disposait d'une structure de base unique pour construire un objet monitoré avec la possibilité de déclarer plusieurs variables condition et plusieurs files d'attente. Lorsqu'on peut leur programmer des comportements spécifiques, il est plus facile de comprendre et de contrôler le comportement des processus concurrents que lorqu'on doit les faire tous passer par une file d'attente fourre-tout. D'autre part disposer du même moyen d'expression quel que soit le nombre de conditions de synchronisation serait plus simple, et, partant, plus fiable. Notre proposition d'améliorisation du moniteur de base s'appliquerait ici aussi où il faudrait également adopter la politique du réveil prioritaire. L'effort à faire pour cette proposition nous paraît beaucoup plus simple que celui qui a été fait pour ajouter l'ensemble des mécanismes de concurrence de bas niveau dans Java J2SE/JDK 1.5.

4. Complexité du code et variabilité des comportements des processus.

Certains comme Knuth attachent une importance esthétique au style de programmation : "*The process of preparing programs can be an aesthetic experience much like composing poetry or music*"[Knuth 1969]. D'autres pensent qu'un style simple et clair permet de mieux comprendre, prouver et mettre au point des programmes, et vont jusqu'à proposer de construire conjointement un programme concurrent et sa preuve [Kaiser 1997].

Pour ce faire, nous disposons d'un outil d'analyse de comportement de programmes concurrents, Quasar [Evangelista 2003], qui nous permet de vérifier certaines propriétés comme l'absence d'interblocage. Il donne aussi deux mesures de complexité des programmes. La première est une indication de la complexité du code : elle se déduit de la taille du réseau de Petri coloré généré par Quasar pour modéliser le programme à analyser. L'autre est une indication de la variabilité des comportements : cette variabilité traduit l'indéterminisme des exécutions et le grand nombre d'exécutions différentes possibles (histoires de l'algorithme) ; elle se déduit de la taille du graphe des marquages du réseau modélisateur. Le tableau 3 présente les mesures faites pour la solution du paradigme des philosophes présentée ici.

Programme	PN coloré #places	PN coloré #transitions	graphe des marquages # noeuds	graphe des marquages #arcs
Allocation fiable non équitable	116	89	4 745	5 073
Allocation avec réveil banalisé	131	104	68 455	72 153
Allocation avec réveil prioritaire	137	109	50 574	51 923

Tableau 3. Mesures de complexité obtenues avec Quasar

On constate sans surprise que, pour une solution fiable et équitable, la complexité du code est comparable dans les deux politiques mais que la politique du réveil banalisé, choisie par Java, a une variabilité un tiers supérieure à la solution avec réveil prioritaire.

5. Remarque sur le style de programmation et son influence

Cette remarque porte sur les structures de contrôle de concurrence, que l'on exprime les structures de programmation avec des langages « littéraires » dérivant d'Algol 60 (Algol 68, PL360, PL10070, Pascal, Ada) ou avec des langages « hiéroglyphiques » (APL, C, C#, Java). Au niveau des structures de contrôle de la concurrence précisément, on peut recommander d'utiliser une forme d'expression avec des commandes gardées pour définir une condition de blocage ou de franchissement d'une barrière et pour contrôler l'exécution d'une méthode pure (qui ne comprend pas d'action bloquante). On peut en plus se donner la possibilité d'enchaîner, dans un objet monitoré, de telles suites :

(< gardes -> méthode pure > ;)*

(en Ada on utilise pour cela une instruction requeue) dont l'exécution est complétée par l'évaluation des gardes en début et fin de méthode pure, avec comme conséquence un réveil implicite possible entre chaque étape de la chaîne. On obtient ainsi un niveau d'expression de plus haut niveau, plus déclaratif, sans instructions ou méthodes « wait() » et « notify() » ou « notifyAll() », et un fonctionnement des méthodes d'un objet monitoré plus proche de celui d'un automate avec files d'attentes. C'est plus facile à comprendre, à exprimer et à valider. Cela existe en Ada et a fait ses preuves.

6. Conclusion

Notre expérience en programmation de systèmes et en enseignement de la concurrence avec divers outils [ACCOV 2005] nous permet d'affirmer que, sauf exemples particuliers, l'utilisation généralisée du moniteur avec la programmation explicite de la synchronisation par « wait » et signal » n'élève finalement pas plus le niveau de programmation que lorsqu'on

utilise des sémaphores confinés à l'intérieur d'un objet. La programmation implicite, telle qu'elle apparaît avec les objets protégés de Ada, permet, elle, un gain de structuration de l'expression de la concurrence et apporte une expression de la concurrence qui est réellement de haut niveau dans tous les cas. Les concepteurs de Java et de C# n'ont pas fait le meilleur choix, qui pour nous est celui des objets protégés Ada. Et en choisissant le moniteur avec programmation explicite de la synchronisation, ils ont choisi la politique de réveil qui ramène encore plus la programmation concurrente vers le bas niveau. Et les choix de Java J2SE/JDK 1.5. semblent confirmer cette volonté de donner des outils de ce niveau.

On a montré dans cet article que la politique donnant la priorité aux processus réveillés, qu'on a appelé la politique de réveil prioritaire, possède une compatibilité ascendante avec la politique de réveil banalisé, actuellement utilisée pour Java et C#. Des algorithmes qui conduisaient à interblocage avec celle-ci retrouvent un comportement fiable avec le réveil prioritaire. Les algorithmes fiables avec la politique de réveil banalisé le demeurent avec la politique de réveil prioritaire. L'efficacité des algorithmes est meilleure parce que le nombre de commutation de contexte est moindre. La complexité du code et des exécutions est souvent moindre. Alors pourquoi s'en priver et ne pas implanter cette politique dans un noyau de système écrit en Java ou C#. D'autant que le manuel de Java ou C# ne l'interdit pas. (Selon la documentation Java : « The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object ». Ceci n'interdit pas d'utiliser Java avec un système attribuant des priorités variables aux processus et donnant momentanément une plus forte priorité à un processus réveillé). En attendant le moment où une programmation implicite s'inspirant des objets protégés de Ada (avec le requeue et les familles d'entrées, qui apportent une grande puissance d'expression) sera adoptée pour Java ou C# sous la pression du besoin de fiabilité des applications, on pourrait implanter en Java un moniteur avec plusieurs conditions de synchronisation explicites, ce qui permettrait déjà une expression unique, plus simple et plus fiable, des algorithmes. Ici encore le gain serait appréciable pour ces langages et permettrait mieux leur emploi dans les systèmes d'exploitation, en particulier embarqués ou temps réel.

Les programmes avec lesquels ont été faits les essais et mesures présentées dans cet article sont disponibles sur le site Quasar [Quasar 2006] à

http://quasar.cnam.fr/files/concurrency_papers.html

Bibliographie

[ACCOV 2005] http://deptinfo.cnam.fr/Enseignement/CycleSpecialisation/ACCOV/

[Ada 1999] Ada 95 Reference Manual, International Standard ANSI/ISO/IEC-8652 :1995, January 1999.

[Andrews 1991] G. Andrews. *Concurrent Programming: Principles and Practice*. 637 pages. Benjamin/Cummings, 1991

[Buhr 1995] P. Buhr, M. Fortier, M. Coffin. Monitor Classification, *ACM Computing Survey*, 27,1, pp. 63-107. 1995.

[Burns 1995] A. Burns and A. Wellings. *Concurrency in Ada*, 390 pages. Cambridge University Press, 1995.

[Courtois 1977] P. Courtois and J. Georges. On starvation prevention. *RAIRO Informatique/Computer Science*. Volume 11, 2 . pp. 127-141, 1977.

[Dijkstra 1968] E.W. Dijkstra. The Structure of the "THE" Multiprogramming System. *Communications of the ACM*, 11,5. pp. 341-346, May 1968.

[Dijkstra 1971] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, number 1, pp. 115-138, 1971.

[Evangelista 2003] S. Evangelista, C. Kaiser, J.F. Pradat-Peyre, P. Rousseau. Quasar : a new tool for analyzing concurrent programs. *International Conference on Reliable Software Technologies, Ada-Europe'03*, LNCS vol. 2655, pp. 166-181, Springer-Verlag 2003. Toulouse, France, June 2003.

[Hoare 1974] C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*. 17,10. pp. 549-557. October 1974.

[Kaiser 1997] C. Kaiser and J.F. Pradat-Peyre. Comparing the reliability provided by tasks or protected objects for implementing a resource allocation service : a case study. *Tri-Ada'97 Conference*, Saint-Louis, USA. pp. 51-65. ACM publication, 1997.
[Kaiser 2003] C. Kaiser and J.F. Pradat-Peyre. Chameneos, a Concurrency Game for Java, Ada and Others. 8 pages, *ACS/IEEE Int. Conf. AICCSA'03*, Tunis, July 2003. IEEE CS Press.
[Kaiser 2006] C. Kaiser, J.F. Pradat-Peyre, S. Evangelista, P. Rousseau. Comparing Java, C# and Ada Monitors queuing policies: a case study and its Ada refinement. 15 pages. To be published in *Ada Letters*, ACM Press 2006
[Kessels 1977] J. Kessels. An alternative to event queues for synchronization in monitors. *Communications of the ACM*, 20,7. pp. 500-503, July 1977.
[Knuth 1969] D. Knuth. The Art of Computer Programming, Volume 1. Fundamental Algorithms. 634 pages. Addison-Wesley 1969.
[Lampson 1980] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23,2. pp. 105-117, February 1980.
[Lea 1997] D. Lea. *Concurrent Programming in Java*. 340 pages. Addison Wesley 1997.
[Quasar 2006]. http://quasar.cnam.fr/

Annexes

Annexe 1. Dîner des philosophes avec allocation globale (non fiable en Java actuel)

L'intérêt de cette solution est qu'elle permet une exécution fiable et équitable qui apporte un maximum de concurrence. Des essais avec une charge simulée et 5 philosophes ont permis de comparer les nombres de fois où deux philosophes mangent en même temps. Pour l'allocation globale originale [Dijkstra 1971], qui est fiable mais non équitable, cette simulation permet d'atteindre un taux de 80%, pour d'autres allocations fiable et équitable grâce à une allocation progressive [Kaiser 1977] on n'a plus que 40% alors que la présente solution (corrigée pour être fiable) obtient un taux de 60%.

```
public final class Chop {

    private int N ;
    private boolean available [ ] ;
    private boolean requestor [ ] ; // starving philosopher

    Chop (int N) {
        this.N = N ;
        this.available = new boolean[N] ;
        this.requestor = new boolean[N] ;
        for (int i =0 ; i < N ; i++) {
            available[i] = true ; // non allocated stick
            requestor[i] = false; // presently non starving
        }
    }
    // méthode appelée pour obtenir les ressources
    public synchronized void get_LR (int me) {
        while ( !available[me] || !available[(me + 1)% N] || requestor[(me + N - 1)% N] ) {
    try { requestor[me] = true ; wait() ; } catch (InterruptedException e) {}
        }
        available[me] = false ; // left stick allocated
        available[(me + 1)% N] = false ; // right stick allocated also
        requestor[me] = false ;
    }

    // méthode appelée pour restituer les ressources
    public synchronized void release (int me) {
        available[me] = true ; available[(me + 1)% N] = true ;
        notifyAll() ;
    }
}
// l'utilisation de requestor permet d'indiquer une attente et d'empêcher qu'elle se poursuive indéfiniment
// mais elle induit un interblocage par un empêchement circulaire dû à requestor
```

Pour obtenir un programme fiable en Java avec N = 5, il faut empêcher un philosophe dont les deux voisins ont faim de requérir les baguettes qu'il vient de libérer. En effet pour N = 5, cette situation n'est possible que si le philosophe mangeait. Donc on empêche un philosophe qui mangeait de reprendre immédiatement les deux baguettes qu'il vient de rendre. On donne priorité à l'un de ses voisins qui attendent.

La méthode partner doit alors commencer par le code défensif suivant :

```
while (requestor[(me + 1)% N] && requestor[(me + N - 1)% N] ) {
    try { wait() ; } catch (InterruptedException e) {}
}
```

Annexe 2. Rendez-vous entre pairs (non fiable en Java actuel)

```
public class Rendez_Vous {
    private ThreadId APartner, BPartner;        // names of the first and second requesting thread
    private boolean FirstCall = true;           // false when SecondCall

    // la méthode partner(x) appelée par x fournit à x le nom d'un pair avec qui faire un échange pair à pair
    public synchronized ThreadId partner(ThreadId x){
        ThreadId result;
        // the following is the code solving the specifications of the problem
        if (FirstCall){
            APartner = x; FirstCall = false;    // now the caller must wait the end of the second request
            while ( !FirstCall ){
                try{wait();} catch(InterruptedException e){}
            }
            result = Bpartner ; // notifyAll(); will be necessary in the defensive code
        }
        else{
            BPartner = x; result = APartner; FirstCall = true; MustWait = true;
            notify(); // // notifyAll(); will be necessary in the defensive code
        }
        return result;
    }
}
```

Ce programme est faux à cause de la sémantique Java de réveil banalisé. Cette sémantique n'interdit pas à d'autres pairs de perturber le rendez-vous et l'on peut avoir des appariements incorrects, par exemple : partenaire(A) = D, partenaire(B) = A, partenaire(C) = D, partenaire(D) = C, qui conduisent à interblocage quand les pairs s'appellent pour communiquer. La sémantique de réveil prioritaire donne un résultat correct.

Pour obtenir un programme correct en Java avec la sémantique de réveil banalisé il faut ajouter du code défensif pour empêcher la perturbation par un troisième appelant. Il faut ajouter un booléen supplémentaire

```
private boolean MustWait = false;           // used for defensive code
```

Ce booléen doit être mis à vrai quand le second pair s'est fait connaître et que le premier pair, qui est en attente, n'est pas encore revenu dans l'objet monitoré pour lire son nom. Une fois cette lecture faite, le booléen MustWait est remis à faux.

La méthode partner doit alors commencer par le code défensif suivant :

```
// the following loop is a necessary defense, forbidding access by a third partner
while (MustWait){
    try{wait();} catch(InterruptedException e){}
}
```

Annexe 3. Allocation de ressources avec une file d'attente

```
public final class Resource {

    private int Max ; // Number of resources
    private int stock ; // Current number of non allocated resources
    private boolean requested = false;

    Resource (int Max) {
        this.Max = Max ;
        this.stock = Max;
    }

    // méthod called when "number" ressources are needed
    public synchronized void request(int number) {
        while ( requested ){
            try { wait() ; } catch (InterruptedException e) {}
        }
        requested = true;
        stock -= number; // new value of stock, may be negative
        while (stock < 0) {
            try { wait() ; } catch (InterruptedException e) {}
        }
        requested = false;
    }

    // méthod called when the "number" ressources are no longer used
    public synchronized void release (int number) {
        stock += number ;
        if (stock >= 0) notifyAll() ;
    }
}

// The first requesting thread is served first
// either it receives its requested number at first call
// or once signalled with all  other waiting requests when once enough resources are returned
// it will be the sole allowed to proceed when calling request anew
// other signalled callers request is denied and they are queued again
```

Annexe 4. Allocation de ressources avec deux files d'attente

```
import java.util.concurrent.locks.Condition ;
import java.util.concurrent.locks.Lock ;
import java.util.concurrent.locks.ReentrantLock ;

public final class Resource {

    private int Max ; // Number of resources
    private int stock = Max ; // Current number of non allocated resources
    private boolean requested = false;
    final Lock lock = new ReentrantLock();
    final Condition Participation = lock.newCondition() ; // waiting queue for clients
    final Condition Grant = lock.newCondition() ; // waiting queue for the selected client

    Resource (int Max) {
        this.Max = Max ;
        this.stock = Max;
    }

    // méthod called when "number" ressources are needed
    public void request(int number) {
        lock.lock();                    // mutual exclusion entrance
        try {
            while ( requested ) Participation.await();
            requested = true;
            stock -= number; // new value of stock, may be negative
            while (stock < 0) Grant.await() ;
            requested = false;
            Participation.signal(); // for a possibly blocked request
        } catch (InterruptedException e) {
        } finally {
            lock.unlock();              // mutual exclusion leave
        }
    }

    // méthod called when the "number" ressources are no longer used
    public void release (int number) {
        lock.lock();                    // mutual exclusion entrance
        stock += number ;
        if (stock >= 0 ) Grant.signal() ; // requested is not necessary
        lock.unlock();                  // mutual exclusion leave
    }

}
// The first requesting thread is served first
// either it receives its requested number at first call or it waits if there is not enough resources available
// once enough resources returned; it is the sole to be signalled for ending its request call
// other callers requests are ignored and blocked until the priviledged client is granted its full request
```