HAL
archives-ouvertes.fr

Linux Activations : un support système performant pour les applications de calcul multithreads

Vincent Danjean

▶ **To cite this version:**

Vincent Danjean. Linux Activations : un support système performant pour les applications de calcul multithreads. [Research Report] LIP RR-2000-14, Laboratoire de l'informatique du parallélisme. 2000, 2+9p. hal-02101929

HAL Id: hal-02101929
https://hal-lara.archives-ouvertes.fr/hal-02101929

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme

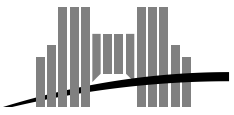
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

LinuxActivations : un support système performant pour les applications de calcul multithreads

Vincent Danjean

March 17, 2000

Research Report N° 2000-14



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

LinuxActivations : un support système performant pour les applications de calcul multithreads

Vincent Danjean

March 17, 2000

Abstract

In this paper, we present LinuxActivation, an efficient system support for user level thread scheduling implemented within Linux. This work is an extension to the "Scheduler Activations" model (proposed by Anderson *and al.* [1]) that better meets the needs of high performance applications. The aim is to allow a user level thread scheduler to handle correctly blocking system calls. After dealing with the limitations of original model, we show how we can efficiently handle all system calls with good performance if we suppose a dedicated machine. We describe the implementation of the mechanisms involved in this new approach within the Linux operating system and the modifications made to a user level thread scheduler to take profits of these mechanisms. The performance numbers we obtain validate our approach.

Keywords: Multithreading, Hybrid scheduling, Operating System, Linux.

Résumé

Cet article présente LinuxActivations, un support système efficace pour l'ordonnancement des processus légers (*threads*) de niveau utilisateur implanté dans Linux. Ces travaux sont une variante des "*Scheduler Activations*" (introduites par Anderson *et al.* [1]) dans le contexte du calcul parallèle. L'objectif est de permettre à un ordonnanceur de *threads* de niveau utilisateur de fonctionner correctement en présence d'appels systèmes bloquants. Après avoir examiné les limitations du modèle originel, nous montrons comment il est possible de couvrir l'intégralité des appels systèmes tout en conservant de bonnes performances si l'on suppose l'exécution sur une machine "dédiée". Nous décrivons l'implantation des mécanismes découlant de cette nouvelle proposition dans le système d'exploitation Linux ainsi que les modifications apportées à un ordonnanceur de threads existant pour tirer parti de ces mécanismes. Les performances obtenues valident notre approche.

Mots-clés: Multithreading, Ordonnancement mixte, Système d'exploitation, Linux.

LinuxActivations : un support système performant pour les applications de calcul multithreads

Vincent Danjean*

17 Mars 2000

Résumé

Cet article présente LinuxActivations, un support système efficace pour l'ordonnancement des processus légers (*threads*) de niveau utilisateur implanté dans Linux. Ces travaux sont une variante des "*Scheduler Activations*" (introduites par Anderson *et al.* [1]) dans le contexte du calcul parallèle. L'objectif est de permettre à un ordonnanceur de *threads* de niveau utilisateur de fonctionner correctement en présence d'appels systèmes bloquants. Après avoir examiné les limitations du modèle original, nous montrons comment il est possible de couvrir l'intégralité des appels systèmes tout en conservant de bonnes performances si l'on suppose l'exécution sur une machine "dédiée". Nous décrivons l'implantation des mécanismes découlant de cette nouvelle proposition dans le système d'exploitation Linux ainsi que les modifications apportées à un ordonnanceur de threads existant pour tirer parti de ces mécanismes. Les performances obtenues valident notre approche.

Mots Clés : *Multithreading, Ordonnancement mixte, Système d'exploitation, Linux.*

Table des matières

*LIP, ENS Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France. Contact: `Vincent.Danjean@ens-lyon.fr`.

This work has been partially supported by the NSF/INRIA C*IT Cooperative Research Grant, the CNRS ARP Research Program and the ReMaP Project, INRIA Rhône-Alpes.

1 Introduction

Le multithreading connaît actuellement un regain d'intérêt dans le domaine du calcul parallèle, et ce pour plusieurs raisons. En premier lieu, la capacité naturelle des processus légers à partager des ressources telles que la mémoire physique en fait évidemment un support privilégié pour l'exploitation des machines multiprocesseurs à mémoire commune (*e.g.*, Sun Enterprise, SGI Origin2000). Cependant, même sur les machines monoprocesseurs, les *threads* demeurent attractifs notamment en raison du faible coût lié à leur gestion. La création, destruction ou synchronisation sont des opérations dont l'efficacité, en ce qui concerne les processus légers, dépasse d'un ordre de grandeur celle relative aux processus classiques. Ainsi, bon nombre de supports d'exécution utilisent les processus légers pour automatiser le recouvrement des communications par des calculs sur des architectures distribuées telles que des grappes de stations ou encore des machines parallèles telles que l'IBM SP2 [4, 5].

En pratique, il faut malheureusement modérer ce bilan car les processus légers se répartissent en deux catégories, chacune n'offrant pas les mêmes propriétés (et par conséquent pas les mêmes avantages) aux applications. Les processus légers *de niveau utilisateur* sont ordonnancés complètement en espace utilisateur (au sein d'un processus) et le noyau du système d'exploitation les ignore totalement. Leur gros avantage réside dans leur excellente efficacité car leur gestion ne nécessite aucune intervention particulière du système. En contrepartie, ces processus légers ne permettent pas un parallélisme réel intra-application car ils ne sont pas ordonnancés individuellement sur les processeurs. Pire, si un processus léger effectue un appel bloquant au système, c'est tout le processus englobant qui se bloque, entraînant donc le blocage forcé des autres processus légers qu'il contient. Les processus légers *de niveau noyau*, quant à eux, ne souffrent évidemment pas de cet inconvénient mais leur gestion est plus coûteuse car certaines opérations nécessitent des appels systèmes (*e.g.*, création, changement de contexte).

Afin de concilier les avantages des deux approches, Anderson *et al.* ont proposé une extension aux systèmes d'exploitation classiques offrant un support pour l'ordonnancement de processus légers au niveau utilisateur : les *Scheduler Activations*. L'idée est d'utiliser un ordonnancement "mixte" à deux niveaux (processus légers + activations) doublé d'une coopération entre l'ordonnanceur du système et l'ordonnanceur des processus légers de manière à prendre en compte les appels bloquants correctement. En tentant une première implantation sous Linux, nous avons toutefois constaté que la mise en œuvre de ce modèle dans un système d'exploitation existant posait plusieurs problèmes importants. En particulier, la simplicité apparente du modèle cache en fait un inconvénient sérieux : il est presqu'impossible de l'étendre à l'ensemble des appels systèmes sous peine de devoir réécrire tout le noyau. D'autre part, les mécanismes impliqués infligent un surcoût non négligeable dû à la prévention d'un certain nombre de situations improbables dans le contexte du calcul parallèle.

Notre contribution est donc de proposer une variante du modèle des *Scheduler Activations* bien mieux adaptée aux applications de calcul intensif sur machines parallèles. L'idée est qu'en introduisant l'hypothèse que la machine est dédiée (exploitation mono-utilisateur), ou du moins faiblement chargée, il est possible de proposer bon nombre de simplifications et d'optimisations substantielles à la fois au niveau du noyau et au niveau de l'ordonnanceur de processus léger. Cet article décrit notre approche et son implantation dans le système d'exploitation Linux. Les performances obtenues sont comparées un ordonnanceur de niveau utilisateur "pur" ainsi qu'avec LinuxThreads [6].

2 Scheduler Activations : un support système pour l'ordonnancement des processus légers

Les processus légers de niveau utilisateur sont incapables d'exploiter réellement une architecture multiprocesseur. Une solution possible consiste à réaliser l'ordonnancement des processus légers par-dessus un ensemble fini de *threads noyaux* : c'est ce que l'on appelle un ordonnancement mixte. Cependant, cela ne résoud pas totalement le problème des appels bloquants qui peuvent, s'il sont nombreux, empêcher certains threads de s'exécuter alors qu'ils sont prêts. En fait, le problème est qu'il n'y a aucun retour d'information lorsque des évènements d'ordonnancement (blocage, préemption, ...) surviennent dans les threads noyaux, ce qui empêche l'ordonnanceur utilisateur d'en tenir compte. Les "Scheduler Activations" apportent cette fonctionnalité, ce qui permet un fonctionnement plus intelligent et efficace de l'ordonnanceur utilisateur.

2.1 Principe général

Dans ce modèle, présenté en 1991 par Anderson *et al.*[1], le noyau met à la disposition de l'application un certain nombre d'activations. Celles-ci peuvent être vues comme des threads noyaux. Ce sont les seuls flots d'exécution connus par le système. Le nombre d'activations disponibles pour une application est compris entre zéro (l'application n'est pas ordonnancée actuellement par le système) et le nombre de processeurs du système. Ce nombre est constant au fil du temps sauf si le noyau décide volontairement de diminuer ou d'augmenter le nombre d'activations de l'application au profit d'une autre application. Cette situation survient notamment sur les systèmes multi-utilisateurs à temps partagé.

Le point crucial de ce modèle réside dans le fait que le noyau prévient l'application de chacun des évènements d'ordonnancement la concernant. Pour ce faire, il utilise un mécanisme symétrique aux appels systèmes : les *upcalls*. Ce mécanisme est assez semblable à celui des signaux : le noyau exécute une fonction située dans l'espace utilisateur de l'application. Cette fonction et ses paramètres vont permettre à l'application de prendre connaissance des évènements survenus dans le noyau. Ainsi, l'ordonnanceur utilisateur sera prévenu lorsque le noyau augmente ou diminue le nombre d'activations disponibles, lorsqu'une activation se bloque dans le noyau, ou lorsqu'une activation bloquée est réveillée. Dans le cas particulier de l'évènement "*activation bloquée*", l'upcall fournit l'état de l'activation supprimée, ce qui permet de continuer le code que cette activation exécutait dans une autre activation.

Comme les seuls flots d'exécution connus par le système sont les activations, cela signifie que les upcalls doivent s'exécuter au sein d'activations. Ainsi, pour signaler un blocage par exemple, le noyau doit interrompre une autre activation pour effectuer l'upcall. Le mécanisme est illustré dans la figure 1. Au temps T_1, deux upcalls `new` sont faits pour prévenir l'application des deux activations disponibles. Quand le thread de l'activation A se bloque au temps T_2, une nouvelle activation est créée (pour occuper le processeur désormais libre), un upcall `new` est effectué pour le signaler à l'application. De même, un upcall `block` prévient l'application du blocage du thread. Enfin, à la fin de l'appel bloquant, au temps T_3, un upcall `unblock` est fait par le noyau pour signaler à l'application la fin de l'appel système bloquant.

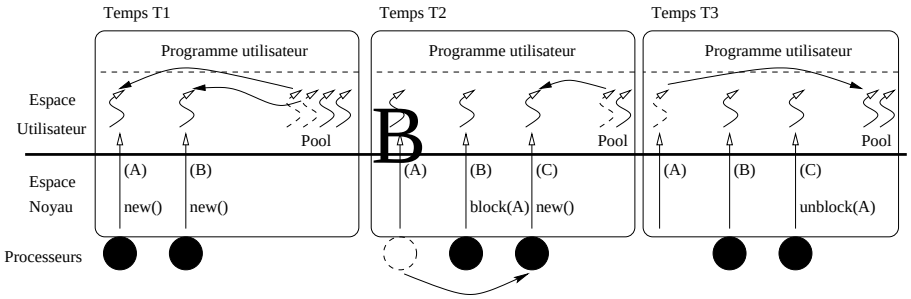


FIG. 1: Déroulement d'un appel système bloquant avec les activations

2.2 Gestion "multi-utilisateurs"

Afin de garantir l'exécution "normale" d'une application en présence d'autres utilisateurs sur une machine, le noyau doit informer ladite application lorsqu'il préempte une activation lui appartenant. Une telle préemption peut se produire dans plusieurs situations. Tout d'abord, il peut s'agir de la perte temporaire d'un processeur au profit d'un processus extérieur (de plus forte priorité par exemple). Ensuite, il peut s'agir de la volonté de l'application d'utiliser une activation de moins (pas assez de threads utilisateurs pour occuper tous les processeurs). Enfin, il peut s'agir de la signalisation de la fin d'un appel bloquant ne pouvant s'effectuer à l'aide d'une nouvelle activation si leur nombre maximal est atteint.

Toutes ces notifications du noyau à l'application sont importantes et ont leur utilité. En outre, il faut noter que, même lorsque la perte d'un processeur n'est que temporaire (*e.g.*, préemption causée par la gestion en temps partagé), l'application doit être prévenue. En effet, le principal aspect ici concerne les verrous utilisateurs. Si le noyau préempte une activation exécutant un thread détenant un verrou, cela peut bloquer d'autres activations tentant de s'approprier le même verrou. En prévenant l'ordonnanceur utilisateur de la préemption, celui-ci peut s'apercevoir du problème et donc décider de remplacer un thread dans une autre activation par ce thread-là.

3 LinuxActivations : une alternative performante pour grappes dédiées

Le modèle des activations tel qu'il a été présenté et exposé ici pose des contraintes lourdes sur l'implantation du mécanisme des activations et leur intégration avec un ordonnanceur de threads. Ces contraintes sont principalement dues à la prise en compte des aspects liés à la multiprogrammation en temps partagé de la machine.

Les upcalls signalant la préemption d'une activation par l'ordonnanceur du système ont principalement pour but la gestion efficace des verrous. Cependant, on s'aperçoit très vite de la complexité de cette tâche. En effet, à tout moment, le nombre d'activations disponibles pour une application peut se réduire à "une". Cela signifie que si l'ordonnanceur veut exécuter les threads détenant des verrous, il faut alors soit hiérarchiser tous les verrous (dans le cas où deux threads ont des verrous, il faut pouvoir choisir lequel exécuter en priorité), soit ne traiter qu'un seul verrou pour toute l'application.

Un autre problème dû à la préemption est le fait que les activations, lorsqu'elles exécutent un upcall, ne sont pas initialement "prêtes à être préemptées". En effet, elles doivent d'abord s'approprier

un thread à exécuter, faute de quoi l'état de l'activation fourni lors des upcalls de préemption serait inconsistant. Cela impose alors un appel système à la fin de chaque upcall pour prévenir le noyau que l'activation est dans un état correct (exécute un thread utilisateur). Cela rend beaucoup plus coûteux les upcalls et complexifie grandement l'exécution d'upcalls en parallèle.

Enfin, il faut noter qu'un upcall de préemption peut survenir à n'importe quel moment et qu'il doit toujours fournir à l'ordonnanceur utilisateur l'état du thread préempté. Cependant, si l'on était au milieu d'un changement de contexte de threads, l'ordonnanceur utilisateur doit absolument détecter et gérer correctement ce cas. Tout cela rend le changement de contexte beaucoup plus coûteux qu'à l'ordinaire et diminue les performances de ce modèle.

3.1 Un nouveau modèle pour une implantation efficace

Pour remédier à ces problèmes, nous proposons une variante des Scheduler Activations permettant à la fois la prise en compte de tous les appels systèmes ainsi qu'une exécution bien plus efficace. Dans ce nouveau modèle, nous conservons les mécanismes fondamentaux décrits précédemment. Les activations sont toujours les seuls flots d'exécution connus par le noyau ; c'est ce dernier qui met les activations à disposition de l'application ; les communications du noyau vers l'application se font toujours par upcall. Toutefois, quelques modifications sont apportées au modèle originel.

Dans le modèle proposé par Anderson, une activation se bloquant au sein d'un appel systèm est immédiatement supprimée du noyau pendant qu'une nouvelle activation est crée, ce qui permet conserver un nombre constant (ou au moins borné par le nombre de processeurs) d'activations s'exécutant simultanément. Ceci impose toutefois une profonde intégration du mécanisme des activations dans le noyau car ce dernier doit être capable, à chaque fois qu'un appel système initialement bloquant est satisfait, de retrouver l'application qui attendait cet événement et de générer l'upcall donnant l'état du thread après complétion de l'appel système. Il apparaît donc difficile de prendre en compte la totalité des appels systèmes bloquants, car il faut intégrer les mécanismes d'upcall directement aux endroits où le noyau peut recevoir un événement signifiant la fin d'un appel système bloquant !

Dans notre modèle, nous n'avons pas retenu cette solution : nous ne détruisons pas les activations lorsqu'elles se bloquent, mais seulement après la fin de l'appel bloquant. Cela a plusieurs conséquences. La première est que le nombre d'activations utilisées par une application n'est plus borné a priori. Il y a toujours moins du nombre de processeurs d'activations "actives", mais il peut y avoir un nombre quelconque d'activations bloquées dans le noyau. L'intérêt est que tous les appels systèmes peuvent être traités à l'aide d'une simple modification localisée dans le noyau : Il suffit d'insérer un test dans l'ordonnanceur système à l'endroit où les activations changent d'état pour gérer automatiquement toutes les situations pouvant amener le noyau à générer un upcall.

Un autre point important de notre modèle est le fait que nous destinons notre système aux grappes de calcul parallèle. Dans ce contexte, nous supposons que notre processus sera le seul sur la machine. Cela signifie que nous n'avons plus à tenir compte de la préemption. Si le noyau nous enlève d'un processeur, nous pouvons supposer que cette préemption sera de durée négligeable puisque nous sommes le seul processus. La préemption n'intervient pas sur une machine mono-processeur : si une activation est préemptée sur une machine mono-processeur, il n'y a pas d'autre activation en fonctionnement pour signaler cette préemption jusqu'à ce que cette activation soit de nouveau ordonnancée, donc cette préemption est ignorée. Le fait que nous ignorions les préemptions ne pourra apporter des conséquences négatives par rapport au modèle précédent que dans le cas de machines multi-processeurs. Mais comme nous allons le montrer dans la section suivante, cette

hypothèse simplificatrice ouvre la porte à de très nombreuses optimisations et conduit en définitive à une augmentation substantielle des performances.

3.2 Articulation Activations/Threads

Lorsque le noyau crée une activation, il a besoin de deux piles : une pile pour son fonctionnement en mode noyau et une pile pour son fonctionnement en mode utilisateur. En régime continu, la pile du mode utilisateur sera la pile du thread utilisateur exécuté. Cependant, lors de l'upcall `new`, le noyau doit disposer d'une pile utilisateur temporaire. Afin de permettre des upcalls en parallèle sur des machines multi-processeurs, nous mettons donc à la disposition du noyau une pile utilisateur par processeur. L'absence de préemption est une propriété très importante ici : on a l'assurance que l'activation continue normalement son exécution tant qu'elle n'effectue pas d'appel système bloquant. Ainsi, l'application peut choisir un thread (avec sa pile) qui sera exécuté par cette activation avec la certitude que le système ne réclame pas entre temps la pile utilisateur utilisée lors de l'upcall. Gérer la préemption aurait nécessité que l'application prévienne le noyau du moment où il peut à nouveau utiliser la pile utilisateur.

Lorsqu'un thread utilisateur exécute un appel bloquant, le noyau s'aperçoit du changement d'état de l'activation sous-jacente. Il crée alors une nouvelle activation et exécute un upcall `new` avec la pile utilisateur correspondant au processeur de l'activation bloquée (Figure 2, situation A). Lorsque l'appel bloquant se termine, l'activation est arrêtée juste avant son retour en mode utilisateur. Le noyau choisit alors une activation en exécution. Il sauvegarde l'état des deux threads des deux activations, chacun sur leur pile respective (situation B). Puis il utilise la pile du thread débloqué pour faire l'upcall `unblock` (situation C). L'application peut alors sauver son état et marquer le thread comme prêt à être exécuté. Par la suite, un appel système sera effectué pour que le noyau retourne exécuter le thread interrompu (en relisant son état sur sa pile).

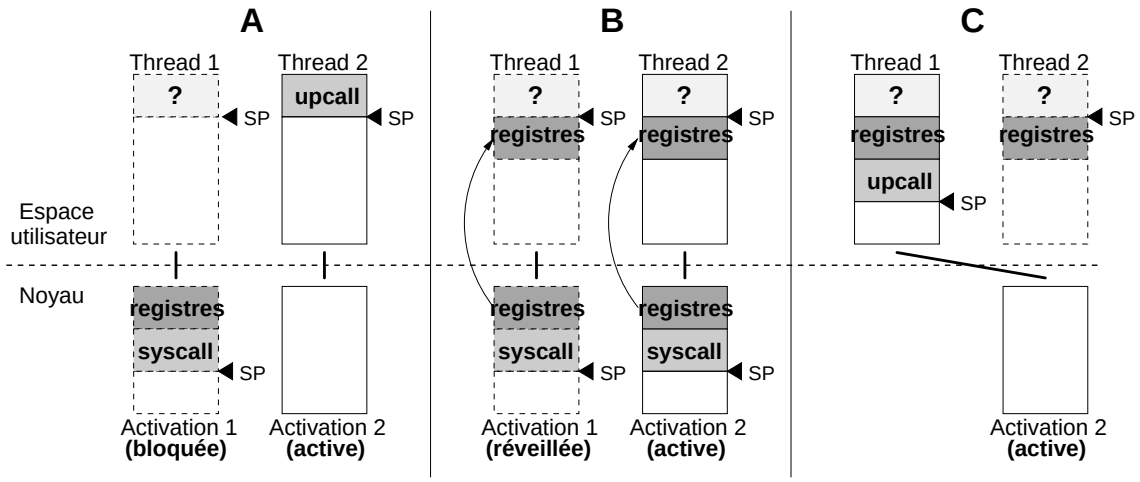


FIG. 2: Scénario typique du déroulement d'un appel bloquant et de son réveil. *SP* désigne le pointeur de pile courant. Il y en a un pour la pile de niveau utilisateur (thread) et un pour la pile dans le noyau (activation).

3.3 Discussion

Nous dressons ici le bilan des optimisations effectuées grâce aux différences introduites par rapport au modèle originel des Scheduler Activations.

Tout d'abord, ne pas chercher à borner a priori le nombre d'activation utilisé par le noyau pour une application nous permet, comme on l'a vu, à la fois de traiter tous les appels systèmes bloquants sans exception, et d'avoir peu de modifications à apporter au noyau. En pratique, il est possible de limiter ce nombre à une constante raisonnable de manière à éviter la saturation du système.

Ensuite, le choix de ne pas signaler les préemptions du noyau a de nombreuses conséquences. Les changements de contexte utilisateurs sont beaucoup plus légers puisque l'on n'a pas à se prémunir contre les préemptions intempestives. De même, les upcalls sont moins coûteux puisqu'il n'est pas nécessaire de prévenir le noyau lorsqu'il peut à nouveau se servir de la pile utilisateur utilisée pour l'upcall.

Il faut cependant remarquer que notre modèle ne propose plus de mécanismes pour assurer la continuité des threads possédant des verrous en cas de préemption. Toutefois, ce problème n'apparaît que sur une machine multi-processeur utilisée par de nombreux processus, ce qui n'est pas le cas des grappes de calcul dédiées. En effet, il n'y a que dans ce cas qu'une activation ayant un verrou peut être préemptée par le noyau (pour donner la main à un autre processus) alors qu'une autre activation continue d'être exécutée et tente de prendre le verrou. Cette activation devra alors faire un peu d'attente active, le temps que le noyau redonne la main à l'activation préemptée qui possédait le verrou.

4 Implantation et performances

Nous avons choisi d'implanter notre modèle dans le système d'exploitation Linux car ses sources sont disponibles librement et la documentation abondante sur Internet. Le mécanisme d'upcall est complètement générique, mais nécessite évidemment une bibliothèque de threads utilisateurs pour en tirer parti. Nous avons choisi de modifier la bibliothèque de threads Marcel, développée dans le cadre de PM^2[7] (*Parallel Multithreaded Machine*). Notons qu'une première implantation a été faite où l'on tenait compte de la préemption. Les détails et performances de cette implantation peuvent être trouvés dans [3, 2].

Grâce au modèle choisi, nous n'avons dû apporter que peu de modifications au noyau Linux (la version du noyau modifiée est la 2.2.14, mais les modifications sont aisément transposables aux autres versions). Ces modifications concernent principalement les fonctions `schedule()` (pour gérer l'état des activations : actives, bloquées, réveillées), `do_fork()` et `do_exit()` (pour la création et destruction des activations). Un peu de code est également nécessaire pour effectuer les upcalls, si besoin est, lors des transitions des processus du mode noyau vers le mode utilisateur. Les modifications à apporter dans Marcel sont également très localisées. Elles consistent principalement à initialiser l'utilisation des activations au départ, exécuter les upcalls du noyau et protéger le verrou interne utilisé par Marcel.

Nous avons testé notre implantation avec différents programmes. Il faut d'abord remarquer que les programmes fonctionnant avec les autres versions de Marcel fonctionnent encore, sans aucune modification. Désormais, les threads Marcel peuvent faire des appels systèmes bloquants (lecture sur socket, utilisation des IPCs, etc.) sans aucun problème. Les performances présentées dans le tableau 1 montrent le comportement de notre implantation dans deux cas extrêmes. Le

TAB. 1: Performance of MARCEL using activations compared with other thread libraries (on a single processor machine)

Library	Synchronisation	Test E/S
MARCEL/user-level	0.308ms	1959.620ms
MARCEL/hybrid	0.435ms	0.761ms
MARCEL/activations	0.355ms	1.735ms
LINUXTHREAD	13.319ms	0.773ms

premier (*Synchronisation*) utilise de façon intensive les primitives de synchronisation entre threads en calculant une somme par la méthode *diviser pour régner* (le programme génère un arbre de threads). Le second programme (*Test E/S*) contient deux threads qui font alternativement une lecture et une écriture sur un unique tube unix. Il génère ainsi une suite d'appels systèmes bloquants.

Les bibliothèques testées sont MARCEL en version threads purement utilisateurs, MARCEL hybride (threads utilisateurs ordonnancés par quelques threads noyaux), MARCEL ACTIVATION (notre proposition décrite dans cette article), et LINUXTHREAD (la bibliothèque standard de threads noyaux sous Linux).

On peut constater que notre implantation est très proche des performances d'une bibliothèque de threads purement utilisateurs pour les opérations de synchronisation, et bien meilleure que la bibliothèque de threads noyaux LINUXTHREAD. La gestion des appels bloquants provoque toujours un surcoût par rapport à LINUXTHREAD, ce qui normal : notre version introduit un upcall supplémentaire à chaque appel bloquant.

5 Conclusion et travaux futurs

Nous avons présenté LinuxActivations, un support système efficace pour l'ordonnancement des processus légers (*threads*) de niveau utilisateur implanté dans Linux. L'objectif était de permettre à un ordonnanceur de *threads* de niveau utilisateur de fonctionner correctement en présence d'appels systèmes bloquants. Ces travaux sont une variante des "*Scheduler Activations*" bien mieux adaptée aux applications de calcul parallèle s'exécutant sur machines dédiées.

Après avoir mis en évidence l'impact de certaines caractéristiques du modèle original sur les performances, nous montrons comment il est possible de couvrir l'intégralité des appels systèmes tout en conservant de bonnes performances en supposant l'exécution sur une machine "dédiée". Nous avons décrit une implantation dans le système d'exploitation Linux et mesuré les performances de quelques applications synthétiques. Les performances obtenues confirment la validité de notre approche.

Nous travaillons actuellement à l'amélioration de l'intégration de ce mécanisme dans la bibliothèque MARCEL, en particulier pour autoriser une exécution en temps partagé au niveau utilisateur.

Références

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations : Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1) :53–79, February 1992.

[2] Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4rd Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, Lect. Notes in Comp. Science, Cancun, Mexico, May 2000. Held in conjunction with IPPS/SPDP 2000. IEEE TCPP and ACM SIGARCH, Springer-Verlag. To appear.

[3] Vincent Danjean, Raymond Namyst, and Robert Russell. Linux kernel activations to support multithreading. In *Proc. 18th IASTED International Conference on Applied Informatics (AI 2000)*, Innsbruck, Austria, February 2000. IASTED. To appear.

[4] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal on Parallel and Distributed Computing*, (37) :70–82, 1996.

[5] I. Ginzburg. *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications*. Thèse de doctorat, Institut National Polytechnique de Grenoble, LMC, Sep 1997.

[6] Xavier Leroy. The LinuxThreads library. http ://pauillac.inria.fr/~xleroy/linuxthreads.

[7] R. Namyst and J-F. Méhaut. PM2 : Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95 (PARallel COmputing)*, pages 279–285. Elsevier Science Publishers, Sep 1995.