MUTEK : UN NOYAU MULTI-TÂCHES/MULTI-PROCESSEURS SMP POUR SYSTÈMES EMBARQUÉS

Pascal GOMEZ Frédéric PÉTROT Denis HOMMAIS

Département ASIM, laboratoire LIP6, Université Pierre et Marie CURIE
4, place Jussieu, 75252 Paris Cedex 05
Pascal.Gomez@lip6.fr Frederic.Petrot@lip6.fr

RÉSUMÉ : Le micro-noyau logiciel Mutek est un système d'exploitation multi-tâches et multi-processeurs symétriques minimaliste. Il est destiné aux applications embarquées et supporte un sous ensemble des fonctions *thread* POSIX. Chaque processeur a une vision équivalente de l'espace mémoire et un rôle équivalent dans l'exécution des *threads*. Il n'y a ni *shell*, ni gestion des entrées/sorties, ni de système de fichiers. Le noyau est en revanche très léger et gère efficacement les communications avec des coprocesseurs.

I – INTRODUCTION

Dans le cadre de la conception conjointe matérielle-logicielle, il est possible, pour certaine classes de problèmes, de décrire les applications comme des graphes de tâches communicantes, plus particulièrement sous forme de réseau de Kahn [1]. Dans ce formalisme, les tâches communiquent entre elles par l'intermédiaire de piles *fifo*. Cette méthode de conception vise des applications embarquées sur puces, l'architecture cible étant un bus (PI-Bus) autour duquel sont connectés un ou plusieurs microprocesseurs du même type par l'intermédiaire de leurs caches instructions et données, une RAM et des coprocesseurs. Les processeurs ont alors une vision identique de l'espace mémoire. L'utilisation d'un Bus permet d'assurer la cohérence mémoire par le matériel.

Afin d'accélérer le processus de conception, la spécification fonctionnelle de l'application embarquée doit pouvoir être compilée et exécutée aussi bien sur une station de travail (Solaris, linux,...) que sur la plate-forme embarquée. Le formalisme des réseaux de Kahn nous assurant alors que le comportement fonctionnel est identique dans les deux cas. L'utilisation des *pthread* pour réaliser le graphe de tâche permet d'assurer ce portage simplement. L'utilisateur-concepteur de systèmes n'utilise que des primitives de lecture/écriture dans des *fifo* construites au dessus des fonctions POSIX. Il est donc nécessaire de disposer d'une implantation des *thread* POSIX dans le noyau de l'application embarquée.

II – STRUCTURE DU NOYAU

Le micro-noyau est composé de trois couches, la couche de plus haut niveau étant la couche POSIX. La couche intermédiaire, que nous ne détaillerons pas, est composée de certaines fonctions de la librairie C standard. Les fonctions dépendantes de l'architecture sont regroupées dans la couche de plus bas niveau.

II–1 – Couche de bas niveau

La couche de plus bas niveau est l'ensemble des fonctions dépendantes de l'architecture : type de processeurs, type d'interconnexion , accès aux sémaphores physiques, accélérateurs matériels. Les processeurs utilisés pour le développement et l'expérimentation sont des MIPS R3000, le bus est un PI-Bus. Les fonctions de bas niveau sont écrites en assembleur pour le processeur cible. Elles ne représentent qu'une faible portion de l'OS, ce sont essentiellement :

- La commutation de contexte ;
- Le gestionnaire d'interruption, qui effectue une sauvegarde et une restitution de contexte ;
- Le masquage des interruptions ;
- Le code d'initialisation des processeurs ;
- La prise de sémaphore physique (verrou).

Le fonctionnement en mode système permet d'éviter les appels systèmes dans tous ces cas et d'avoir un gain significatif en performance.

La prise d'un sémaphore physique est une opération atomique, il est donc nécessaire de disposer d'un mécanisme matériel de *test and set*. Nous disposons d'un ensemble fini de sémaphores, mais le nombre d'objets qui les manipulent peut être supérieur. Plusieurs objets peuvent donc se partager un même sémaphore physique. Afin d'éliminer les attentes indéterminées, la prise d'un sémaphore physique par un *thread* doit être faite avec certaines précautions : Un *thread* ne peut prendre un sémaphore physique que pendant une période de temps bornée et de préférence la plus courte possible. Ceci implique en particulier qu'un *thread* ne peut pas commuter s'il n'a pas relâché tous les sémaphores physiques qu'il a verrouillé. Pour limiter la durée des blocages, le verrou ne doit être pris que lorsqu'on en a réellement besoin.

II–2 – Couche POSIX

La couche POSIX s'appuie d'une part sur la couche de plus bas niveau et d'autre part sur quelques fonctions de la librairie C standard, notamment *malloc*. C'est, avec les fonctions de la librairie C standard, l'interface utilisateur du système d'exploitation. Seule une partie des fonctions POSIX a été implantée dans le noyau : les fonctions de création et de destruction des *threads*, les fonctions concernant les *mutex* et les conditions, et les fonctions concernant les sémaphores POSIX. Les *threads* sont des programmes séquentiels indépendants qui s'exécutent de manière concurrente sur un ou plusieurs processeurs, ils diffèrent des processus UNIX par le partage de l'espace d'adressage avec les autres *threads*. Chaque *thread* possède sa propre pile d'exécution. Les fonctions POSIX assurent un fonctionnement multi-thread, avec la possibilité de synchroniser les *threads* entre eux, par des mécanismes de sémaphores ou d'exclusion mutuelle.

III – IMPLANTATION

III–1 – Ordonnanceur

L'implantation d'un noyau multi-processeurs symétrique (SMP) implique le partage de l'ordonnanceur par tous les processeurs. L'ordonnanceur peut être accédé soit par les *thread*, par l'intermédiaire de certaines fonctions POSIX, soit lors d'une interruption. Nous utilisons un verrou unique pour le partage de l'ordonnanceur ; à un instant donné, un seul *thread* peut posséder ce verrou. Un autre *thread* ne peut y accéder que lorsque le *thread* qui possède le verrou le relâche. Les accès à cette ressource critique doivent être le plus court possible et bornés dans le temps. L'ordonnanceur ne peut partager ce verrou physique avec aucun autre objet car il y aurait risque d'inter-blocage. En pratique, ce verrou n'est pris que pour déplacer un *thread* d'une liste à l'autre.

Les manipulations rapides de l'ordonnanceur sont assurées par l'utilisation intensive des listes. Les listes correspondantes aux cinq états possibles des *threads*, RUN, RUNNABLE, WAIT, JOIN, ZOMBIE, sont pointées par l'ordonnanceur. Pour l'état RUN, il existe autant de listes que de processeurs, chacune de ces listes ne pouvant contenir qu'un seul *thread*. Quatre listes sont utilisées pour les quatre autres états. Le noyau n'étant pas, pour l'instant, temps réel, nous utilisons une politique d'ordonnancement de type *fifo*. Cette stratégie simple permet d'éviter le parcours des listes.

Il peut arriver qu'un processeur n'ait plus de *threads* a exécuter. Le processeur se met alors en attente soit d'une interruption matérielle, soit d'une décision de l'ordonnanceur, cette décision étant prise sur un autre processeur.

III–2 – Autres structures

D'autres listes de *thread* sont présentes dans le noyau, elles se superposent aux listes de l'ordonnanceur.

La structure *mutex* pointe sur le *thread* qui possède le *mutex* et sur la liste des *threads* qui sont en attente de ce *mutex*. Un *mutex* peut être dans deux états : *LOCK*, le *mutex* est pris, ou *UNLOCK*, le mutex est libre. Deux cas peuvent se présenter lors de la prise d'un mutex par un *thread*. Si le *mutex* est libre, le *thread* est ajouté a la liste *owner* du *mutex*. La liste *owner* ne contient qu'un seul *thread*. Si le *mutex* est déjà pris, le *thread* est placé dans la liste WAIT de l'ordonnanceur et aussi placé au début de la liste *wait* du *mutex*, avec les autres *threads* qui attendent pour posséder le *mutex*. Le *thread* est alors endormi réellement par une commutation de contexte. Cette situation est représentée fig1, fig2 et fig3. Lorsqu'un *thread* relâche un *mutex*, celui-ci est immédiatement donné au premier de la liste des *threads* en attente du *mutex*. Ce *thread* est placé dans la liste des RUNNABLE de l'ordonnanceur. Cette stratégie d'election peut être redéfinie.

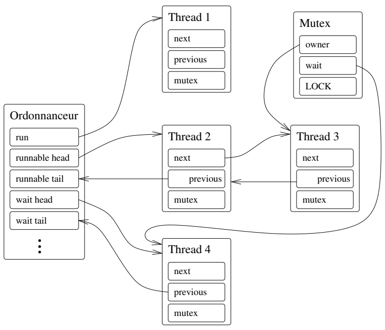

FIG. 1 – Le mutex est pris par le *thread* 3 qui est prêt à être exécuté. Le *thread* 4 est en attente du mutex. Le *thread* 1 est en cours d'exécution sur le processeur.

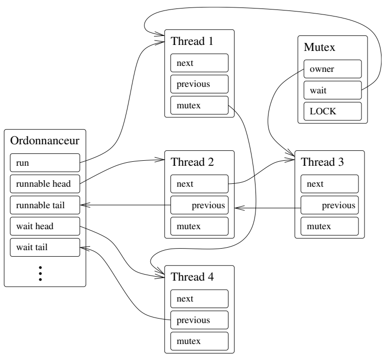

FIG. 2 – Le *thread* 1 veut prendre le mutex, il est ajouté à la liste des *thread* en attente du mutex.

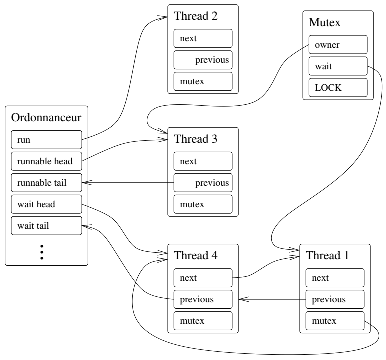

FIG. 3 – Le *thread* 1 est bloqué et est mis dans la liste WAIT. Le *thread* 2, qui était prêt, prend sa place et est exécuté par le processeur.

Cet exemple montre que toutes les manipulations de listes se font avec une complexité en O(1). Ceci est due, dans certain cas, à l'utilisation d'une politique d'ordonnancement simple de type fifo. Une politique d'election plus complexe, avec des priorités fixes par exemple, amènerai à une complexité en O(n) si les structures sont conservées dans l'état. Chaque *mutex* est associé à un sémaphore physique qui doit être pris par le *thread* qui veut modifier les listes pointées par le *mutex*.

Lors d'une commutation, le contexte d'exécution d'un *thread* est sauvé dans la structure *thread*. Cette structure contient aussi la pile d'exécution, des pointeurs de listes ainsi que les éléments requis par la norme POSIX.

Les sémaphores POSIX sont implémentés de façon assez similaire aux *mutex*, la principale différence réside dans la fait qu'un *thread* peut imbriquer plusieurs prise de sémaphores.

III–3 – Interruptions

Une interruption est susceptible d'amener une modification des listes de l'ordonnanceur ou de provoquer une commutation de contexte, il faut donc s'assurer qu'aucune interruption n'arrive lors des manipulations de liste. Pour cela, les interruptions sont masquées pendant toutes ces opérations avec deux contraintes : il faut qu'à chaque masquage soit associé un démasquage et que le temps qui sépare un masquage et un démasquage soit borné et le plus court possible. Les interruptions sont masquées aussi pendant la commutation de contexte, ce qui implique que masquage et démasquage soit réalisés par des *threads* différents. Dans le cas d'une commutation par interruption, la commutation de contexte est effectuée dans le gestionnaire d'interruption. Le retour de l'interruption peut donc avoir lieu sur un processeur différent de celui qui l'a reçu. Ce mécanisme est rendu possible par l'utilisation du processeur en mode système et par le fait que les masques d'interruptions sont identiques pour tous les processeurs. Lors de l'interruption, le contexte qui contient notamment le masque d'interruption, doit obligatoirement être sauvé dans la pile du *thread* en cours d'exécution.

IV – PERFORMANCES

La simulation de systèmes précise au cycle permet de mesurer les performances du noyau pour diverse architectures. Les temps d'exécution de certaines opération, en nombre de cycles d'horloge du système, sont donnés en Tab.1. Ces mesures ont été effectués sur plusieurs systèmes comportant de 1 à 6 MIPS r3000 connectés par un PI-Bus, chaque processeur possédant des caches instructions et données. Tous les processeurs ont la même horloge. Les opérations mesurées sont :

– La commutation de contexte : la valeur mesurée ici représente le temps d'élection d'un *thread* plus la commutation de contexte ;
– La prise et la libération d'un *mutex* sans blocage. Dans ce cas le *mutex* n'était pas déjà pris par un autre *thread* ;
– La prise d'un *mutex* avec blocage. Le *mutex* était déjà pris. Le temps de commutation et la durée de blocage du *thread* n'est pas comptabilisée ;
– La libération d'un *mutex* avec blocage. Le *mutex* était pris par un *thread* et un seul autre était en attente. Il n'y a pas de valeur pour un processeur car, dans ce cas, la probabilité d'un tel événement est très faible.
– La création et la terminaison d'un *thread*.
– La prise et la libération d'un sémaphore POSIX. Le temps de commutation et la durée de blocage du *thread* n'est pas comptabilisée ;
– le gestionnaire d'interruption. Ces valeurs ont été mesurés sur quelques itérations seulement et sont présentées à titre indicatif. Tous les processeurs étaient tous interrompus simultanément ;

opérations	nombre de processeurs			
	1	2	4	6
commutation	172	187	263	351
prise d'un *mutex* (sans blocage)	36	56	61	74
libération d'un *mutex* (sans blocage)	30	30	31	34
prise d'un *mutex* (avec blocage)	117	123	258	366
libération d'un *mutex* (avec blocage)	-	191	198	218
création d'un *thread* (1)	667	738	823	1085
terminaison d'un *thread* (1)	98	117	142	230
prise d'un sémaphore POSIX	36	48	74	76
libération d'un sémaphore POSIX	36	50	78	130
gestionnaire d'interruptions (2)	200	430	1100	1900

TAB. 1 – mesure de performance de quelques fonctions du noyau. Les mesures ont été effectuées sur 1000 itérations de chacune des fonctions, sauf pour (1), les mesures ont été effectuées sur une centaine d'itération, et pour (2), les mesures ont été effectués que sur quelques itérations des fonctions, en interrompant tous les processeurs simultanément.

V – CONCLUSION

Mutek est un noyau logiciel particulièrement compact : 25 ko si toutes les fonctions POSIX sont utilisées, et performant du point de vue de la charge des processeurs comparé à [2]. Le code lié à l'architecture du système est assez restreint, ce qui laisse espérer un portage rapide vers d'autres architectures. Nous espérons notamment pouvoir utiliser ce noyau sur un système multi-processeurs utilisant un réseau commuté pour l'interconnexion, la difficulté étant d'assurer la cohérence mémoire. Les deux axes qui nous intéressent désormais sont la possibilité d'avoir un ensemble de processeurs hétérogène et le développement d'un noyau temps réel. Un système avec un ensemble de processeurs hétérogène nécessite de spécifier la nature du protocole de communication entre les processeurs. Les contraintes temps réel des tâches doivent être gérées par le noyau, il est nécessaire pour cela d'implémenter des politiques d'ordonnancement par priorités et de prendre en compte la périodicité des tâches.

RÉFÉRENCES

[1] *The semantics of a simple language for parallel programming*, G.Kahn, In information processing 74, pages 993-998. North Holland,1977.

[2] *Experience with a Prototype of the POSIX "minimal Realtime System Profile"*, T.P.Baker, Franck Mueller, Viresh Rustagi, Proceedings 11th IEEE Workshop on Real-Time Operating Systems and Software. RTOSS'94.