Système de fichiers scalable pour architectures many-cores à faible empreinte énergétique

Mohamed Lamine Karaoui

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE
UPMC - SORBONNE UNIVERSITÉS

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Mohamed Lamine KARAOUI

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse

Système de fichiers scalable pour architectures many-cores à faible empreinte énergétique

Soutenue le 28 juin 2016
devant le jury composé de :

M. Brice GOGLIN	Rapporteur
M. Vivien QUÉMA	Rapporteur
M. Pierre SENS	Examinateur
M. Fabien CLERMIDY	Examinateur
M. Alain GREINER	Directeur de thèse
M. Franck WAJSBÜRT	Encadrant de thèse

Résumé

Cette thèse porte sur l'étude des problèmes posés par l'implémentation d'un système de fichiers passant à l'échelle, pour un noyau de type UNIX sur une architecture manycore NUMA à cohérence de cache matérielle et à faible empreinte énergétique. Pour cette étude, nous prenons comme référence l'architecture manycore généraliste TSAR et le noyau de type UNIX ALMOS.

L'architecture manycore visée pose trois problèmes pour lesquels nous apportons des réponses après avoir décrit les solutions existantes. L'un de ces problèmes est spécifique à l'architecture TSAR tandis que les deux autres sont généraux.

Le premier problème concerne le support d'une mémoire physique plus grande que la mémoire virtuelle. Ceci est dû à l'espace d'adressage physique étendu de TSAR, lequel est 256 fois plus grand que l'espace d'adressage virtuel. Pour résoudre ce problème, nous avons profondément modifié la structure noyau pour le décomposer en plusieurs instances communicantes. La communication se fait alors principalement par passage de messages.

Le deuxième problème concerne la stratégie de placement des structures du système de fichiers sur les nombreux bancs de mémoire. Pour résoudre ce problème nous avons implémenté une stratégie de distribution uniforme des données sur les différents bancs de mémoire.

Le troisième problème concerne la synchronisation des accès concurrents. Pour résoudre ce problème, nous avons mis au point un mécanisme de synchronisation utilisant plusieurs mécanismes. En particulier, nous avons conçu un mécanisme lock-free efficace pour synchroniser les accès faits par plusieurs lecteurs et un écrivain.

Les résultats expérimentaux montrent que : (1) l'utilisation d'une structure composée de plusieurs instances communicantes ne dégrade pas les performances du noyau et peut même les augmenter ; (2) l'ensemble des solutions utilisées permettent d'avoir des résultats qui passent mieux à l'échelle que le noyau NetBSD ; (3) la stratégie de placement la plus adaptée aux systèmes de fichiers pour les architectures manycore est celle distribuant uniformément les données.

Abstract

In this thesis we study the problems of implementing a UNIX-like scalable file system on a hardware cache coherent NUMA manycore architecture. To this end, we use the TSAR manycore architecture and ALMOS, a UNIX-like operating system.

The TSAR architecture presents, from the operating system point of view, three problems to which we offer a set of solutions. One of these problems is specific to the TSAR architecture while the others are common to existing coherent NUMA manycore.

The first problem concerns the support of a physical memory that is larger than the virtual memory. This is due to the extended physical address space of TSAR, which is 256 times bigger than the virtual address space. To resolve this problem, we modified the structure of the kernel to decompose it into multiple communicating units.

The second problem is the placement strategy to be used on the file system structures. To solve this problem, we implemented a strategy that evenly distributes the data on the different memory banks.

The third problem is the synchronization of concurrent accesses to the file system. Our solution to resolve this problem uses multiple mechanisms. In particular, the solution uses an efficient lock-free mechanism that we designed, which synchronizes the accesses between several readers and a single writer.

Experimental results show that : (1) structuring the kernel into multiple units does not deteriorate the performance and may even improve them ; (2) our set of solutions allow us to give performances that scale better than NetBSD ; (3) the placement strategy which distributes evenly the data is the most adapted for manycore architectures.

Table des matières

Table des figures

CHAPITRE 1

Introduction

Nous assistons ces dernières années à une évolution remarquable dans le monde des processeurs. En effet, la majorité des processeurs sont aujourd'hui multicore, ils contiennent jusqu'à plusieurs dizaines de cœurs, et l'arrivée des processeurs manycore, contenant jusqu'à plusieurs milliers de cœurs, est imminente [13]. Les processeurs manycore considérés dans notre travail sont de type CC-NUMA : *Cache Coherent - Non Uniform Memory Access*. Le terme NUMA signifie que l'architecture mémoire est constituée de plusieurs bancs de mémoire dont les latences d'accès sont variables. Cette variabilité vient du fait que l'architecture est composée de plusieurs sous-systèmes appelés clusters. Chaque cluster est constitué d'un banc de mémoire et d'un certain nombre de cœurs de processeurs. Lorsqu'un cœur accède au banc de mémoire de son cluster, la latence est minimale. Ce type d'accès est appelé local. Par contre, lorsqu'un accès n'est pas local, il est appelé distant. Dans ce cas, la latence et la consommation énergétique varient suivant la distance entre le cluster contenant le cœur et le cluster contenant le banc mémoire. Enfin, le terme CC signifie que la cohérence des caches de premier niveau est assurée par le matériel.

Dans ce cadre, nous nous intéressons au problème du passage à l'échelle des structures de données des systèmes de fichiers pour les systèmes d'exploitation (SE) de type UNIX. Les principales structures considérées sont celles de la couche du *Virtual File System* (VFS). Cette couche a deux rôles principaux. En premier lieu, elle propose une abstraction permettant au noyau d'accéder à plusieurs systèmes de fichiers réels, tels que les systèmes de fichiers FAT, Ext2, UFS ou HFS. En second lieu, elle permet d'accélérer l'accès au système de fichiers en implémentant les caches de données et de métadonnées. Les différents problèmes de ces structures sont étudiés en considérant le noyau de type UNIX **ALMOS** et l'architecture manycore CC-NUMA **TSAR**.

ALMOS est un SE expérimental qui a été spécialement conçu afin d'étudier le passage à l'échelle des noyaux de type UNIX. Le noyau d'ALMOS est monolithique (tout le noyau s'exécute en mode superviseur), il supporte l'API POSIX et il s'exécute en mémoire virtuelle. ALMOS a déjà fait l'objet d'une étude du passage à l'échelle dans la thèse de Ghassan Almaless [5]. Cette étude a permis de montrer le passage à l'échelle de deux sous-systèmes : (1) le sous-système de gestion des processus et (2) celui de la gestion de la mémoire. C'est dans la suite de ces travaux que s'insère notre thèse qui cherche à étudier le passage à l'échelle des structures du

système de fichiers.

L'architecture TSAR est une architecture manycore CC-NUMA à faible empreinte énergétique. Plusieurs choix ont été faits pour réduire l'empreinte énergétique. L'un de ces choix nous impacte particulièrement. TSAR utilise des cœurs de calcul 32 bits, moins gourmands en énergie, mais possède un espace d'adressage physique sur 40 bits. Une telle combinaison implique que la taille de l'espace virtuel (4 Go) est plus beaucoup petite que celle de l'espace physique (1 To). Cette différence de taille nous a conduits à profondément modifier le noyau d'ALMOS afin de le faire fonctionner en adressage physique suivant une architecture multi-instances (aussi appelé *multikernel*), qui consiste à décomposer le noyau en plusieurs instances communicantes.

En plus de la différence de taille entre l'espace virtuel et l'espace physique, les deux autres problèmes posés par l'architecture TSAR sont des problèmes de passage à l'échelle. Le premier concerne la stratégie de placement des données dans une architecture NUMA. Ce problème a été résolu en implémentant une stratégie de distribution uniforme des données sur l'ensemble des bancs de mémoire avec la granularité d'un fichier. Le second problème est celui de la synchronisation des accès concurrents. En effet, l'utilisation de centaines de cœurs nécessite un mécanisme permettent de gérer les conflits d'accès au système de fichiers, lequel est une ressource partagée. Ce problème a été résolu en définissant des mécanismes adaptés à la structure multi-instances du noyau. Ces mécanismes évitent à la fois le surnombre de messages et le risque d'interblocages.

1.1 Contributions

Cette thèse apporte donc trois principales contributions :

1. La première contribution est la définition d'une technique permettant au système d'exploitation ALMOS de supporter l'espace d'adressage physique 40 bits de l'architecture TSAR, tout en utilisant des cœurs 32 bits. Le premier choix de principe est d'adopter une architecture multi-instances, communicant entre elles par RPC, sur un modèle client-serveur. Le second choix de principe est d'exécuter l'ensemble du code noyau directement en adressage physique ;

2. La deuxième contribution est la définition et l'implémentation d'un système de fichiers virtuel distribué, respectant la norme POSIX et passant à l'échelle, pour des architectures manycores clusterisées comportant plusieurs centaines de cœurs.

3. La troisième contribution est la définition d'un algorithme de synchronisation à plusieurs lecteurs et un écrivain passant à l'échelle, et adapté aux structures chaînées telles que les tables de hachage et les listes chaînées, que l'on trouve dans les systèmes de fichiers.

1.2 Plan du manuscrit

Le chapitre 2 définit la problématique de la thèse. La première section présente l'architecture TSAR, en insistant sur les caractéristiques NUMA du sous-système mémoire et l'adressage 40 bits. La deuxième section présente le noyau ALMOS, son architecture et les mécanismes implémentés pour optimiser le placement des tâches et des données. La troisième section analyse en détail les trois problèmes étudiés dans cette thèse.

Le chapitre 3 présente l'état de l'art. La première section présente les mécanismes de synchronisation existants, principalement ceux utilisés dans les systèmes de fichiers. La deuxième section présente les stratégies de placement existantes. La troisième section présente les solutions permettant de supporter un espace d'adressage étendu ainsi que les principaux systèmes multi-instances existants.

Le chapitre 4 présente le principe des solutions proposées. La première partie présente les deux solutions investiguées afin de supporter l'adressage physique étendu, et justifie le choix de la solution utilisant une architecture multi-instances. La deuxième section présente le principe du mécanisme de communication entre instances. La troisième section détaille la stratégie de placement. Enfin, la quatrième section décrit le mécanisme de synchronisation.

Le chapitre 5 décrit l'implémentation des solutions proposées. La première section présente les modifications apportées au noyau ALMOS afin de le découper en plusieurs instances indépendantes. La seconde section présente l'implémentation du mécanisme de communication entre instances. La troisième section présente l'implémentation de la pile du système de fichier : VFS, FAT, DevFS et SysFS. Enfin la quatrième section présente le mécanisme de création à distance de processus.

Le chapitre 6 présente les résultats expérimentaux. La première famille de résultats vise à caractériser l'impact sur les performances du mécanisme de communication par RPC. La seconde famille de résultats vise à comparer les performances de notre version modifiée d'ALMOS et du système d'exploitation UNIX NetBSD, sur un ensemble de benchmarks synthétiques sollicitant fortement le système de fichiers. La troisième famille de résultats analyse plus précisément l'impact sur les performances de la politique de placement des structures de données du système de fichiers.

CHAPITRE 2

Problématique

Notre thèse porte sur les problèmes posés par le passage à l'échelle des structures de données du système de fichiers virtuel (VFS) dans les systèmes d'exploitation pour architectures manycores à mémoire partagée cohérente. Cette étude s'appuie sur l'architecture manycore CC-NUMA TSAR et sur le système d'exploitation ALMOS, lequel est un SE de type UNIX, optimisé pour les architectures manycores CC-NUMA. L'architecture TSAR est conçue pour supporter jusqu'à 1024 cœurs cohérents et un espace d'adressage physique de 1 To. Afin de limiter l'empreinte énergétique, les cœurs utilisés sont de simples cœurs MIPS 32 bits. Ce choix a pour conséquence que l'espace adressable physique est 256 fois plus grand que l'espace adressable par les cœurs.

Les deux premiers problèmes identifiés sont (1) la synchronisation des accès concurrents puisqu'il peut y avoir un grand nombre de threads accédant simultanément au système de fichiers et (2) le placement des structures de données qui doit réduire l'effet de la non-uniformité des latences d'accès à la mémoire (NUMA).

Le troisième problème identifié est lié au fait que l'espace d'adressage du noyau est réduit en raison de l'utilisation de cœurs 32 bits. Cette limitation ne concerne pas seulement le système de fichiers, elle concerne toutes les structures de données du noyau. Comme nous le verrons, ce dernier problème a été le plus difficile à résoudre et a demandé une restructuration profonde du noyau d'ALMOS.

Dans la suite de ce chapitre, nous commençons par décrire l'architecture TSAR, puis nous décrivons le noyau d'ALMOS, et enfin, nous décrivons les trois problèmes auxquels nous apportons des réponses dans cette thèse.

2.1 L'architecture TSAR

L'architecture TSAR est une architecture manycore constituée d'un ensemble de clusters homogènes interconnectés par un réseau intégré sur puce (NoC) possédant une topologie de mesh 2D (grille à deux dimensions). Il y a un cluster à chaque nœud du mesh, et chaque cluster contient 4 cœurs. Pour exemple, la figure 2.1 présente une architecture TSAR à 4x4 clusters. Chaque cluster est identifié par ses coordonnées (x,y) dans le mesh.

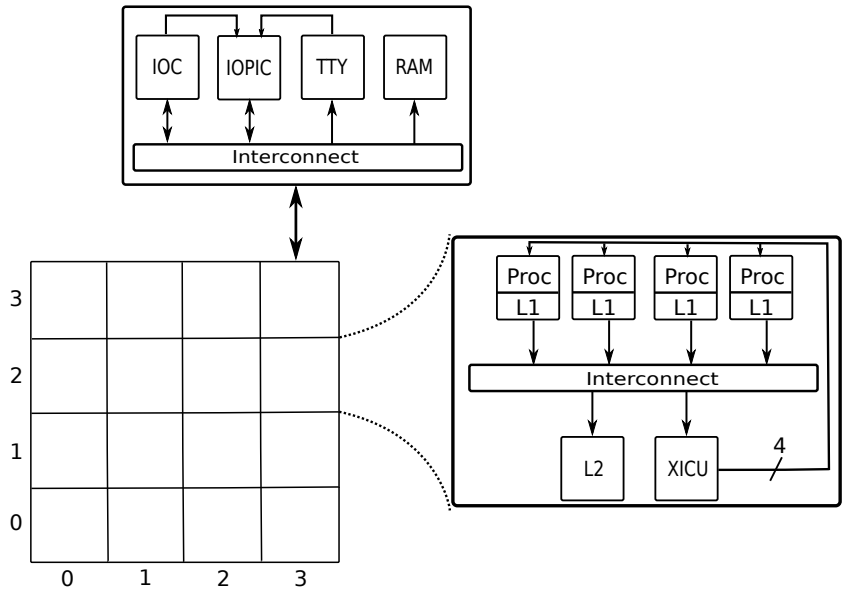


FIGURE 2.1 : Architecture TSAR à 4x4 clusters.

2.1.1 Composition de l'architecture

Composition des clusters

- Chaque cluster de TSAR contient quatre cœurs de processeurs MIPS32. Ces cœurs sont de type RISC 32 bits (*Reduced Instruction Set Computing*). L'utilisation de petits cœurs 32 bits a pour principal objectif de réduire la consommation énergétique. Ce choix est guidé par la règle de Pollack's qui suggère que, pour assurer une consommation énergétique acceptable, les cœurs doivent être simplifiés d'autant plus que leur nombre augmente [13]. En outre, les adresses 32 bits apportent un autre avantage par rapport aux adresses 64 bits : l'empreinte mémoire des exécutables et des données manipulées est plus compacte, puisque les pointeurs font 32 bits au lieu de 64 bits. En conséquence, à taille égale, les caches sont plus efficaces.

  – L'architecture TSAR garantit que les instructions de lectures et d'écritures sont atomiques, tant que la lecture ou l'écriture sont effectuées sur des variables de taille inférieure ou égale à la taille du mot (32 bits). Le terme atomique signifie que lors d'une écriture sur une variable partagée, tout autre thread lisant cette variable voit, soit la nouvelle valeur écrite, soit la valeur précédente et non pas une valeur intermédiaire. Cette propriété est vérifiée sur la majorité des architectures modernes [44].

  – Le couple d'instructions LL/SC défini par le jeu d'instruction du processeur MIPS32 permet d'effectuer des opérations de type *lecture_puis_écriture* atomiques, telle que la lecture d'une donnée à une adresse X, une modification de la valeur lue, puis l'écriture à la même adresse X. Cependant, le couple d'instruction LL/SC n'est pas disponible sur tous les processeurs. L'opération atomique complexe la plus répandue est le CAS (*Compare And Swap*) [22]. Celle-ci modifie la valeur d'une case mémoire

uniquement après avoir vérifié que son contenu est égal à une certaine valeur. Cette instruction n'est pas disponible sur le processeur MIPS32, mais elle peut être émulée grâce au couple d'instructions LL/SC [44] (l'inverse n'étant pas possible). Pour des raisons de portabilité, nous avons choisi de n'utiliser que l'opération CAS pour les différents mécanismes de synchronisation.

- Chaque cœur dispose de deux contrôleurs de caches de premier niveau (L1), un pour les instructions et un pour les données ; deux caches de traduction d'adresses, appelés TLB (*Translation Lookaside Buffer*), un pour les instructions et un pour les données qui contiennent les dernières traductions effectuées par l'unité de gestion de la mémoire, appelée MMU (*Memory Management Unit*) ; et donc enfin, une MMU qui réalise la traduction des adresses virtuelles 32 bits en adresses physiques 40 bits sur la base d'une table de pages (2 tailles de pages : 4 ko et 2 Mo). La cohérence des caches L1 et des TLB est garantie par le matériel.

  La gestion de la mémoire virtuelle paginée est couramment utilisée dans les processeurs modernes. Toutefois, TSAR propose un second mécanisme pour produire des adresses 40 bits qui consiste à simplement à compléter l'adresse 32 bits émise par le cœur avec le contenu d'un registre d'extension sur 8 bits. Ce mécanisme est spécifique à l'architecture TSAR (voir 2.1.3).

- Chaque cluster contient un contrôleur de cache de deuxième niveau (L2), partagé par tous les cœurs de l'architecture. Chaque cache L2 gère son propre segment d'espace physique. Un cache L2 dans un cluster[x,y] peut être accédé par n'importe quel cœur de l'architecture dès lors que l'adresse physique du transfert (lecture ou écriture) appartient au segment mémoire géré par le cache L2 du cluster[x,y] (voir 2.1.2). Dans l'architecture TSAR, les 8 bits de poids fort de l'adresse physique 40 bits définissent les coordonnées [x,y] du cluster auquel appartient cette adresse. Cette propriété permet au SE de contrôler très simplement le placement des données sur les différents clusters, en choisissant les bits de poids fort de l'adresse.

- chaque cluster contient un contrôleur d'interruptions XICU, qui fournit également différents *timers* programmables. Deux types d'interruptions sont gérées par ce composant :

  1. Les interruptions matérielles HWI (*Hardware Interrupt*), lesquelles permettent de router les interruptions internes d'un cluster. Sur l'architecture TSAR, seul le cache L2 utilise ces interruptions afin de signaler à l'OS d'éventuelles erreurs d'écritures en mémoire ;
  2. Les interruptions logicielles, WTI (*Write-Triggered-Interrupt*). Celles-ci sont activées par des écritures effectuées dans des registres spéciaux, appelés boîtes aux lettres. Ce type d'interruptions est généralement utilisé pour implémenter le mécanisme des IPI (*Interruption Inter-Processeurs*) permettant à un cœur d'en interrompre un autre. Il est aussi utilisé pour router les interruptions des périphériques externes à travers l'IOPIC (voir ci-dessous).

Périphériques externes

À l'extérieur de la puce, on trouve principalement les périphériques suivants :

TTY contrôleur des terminaux d'affichage textuel (consoles alphanumériques) ;

IOC contrôleur du disque dur, ou plus généralement du périphérique orienté blocs permettant le stockage des données non volatiles ;

IOPIC contrôleur d'interruption pour l'ensemble des périphériques externes. Ce composant permet de transformer les interruptions de type HWI en une interruption de type WTI, c'est-à-dire une écriture dans l'espace d'adressage physique, vers un des registres boîte aux lettres d'un composant XICU interne ;

Enfin, on trouve aussi une RAM à l'extérieur de la puce. Cette dernière est uniformément partitionnée entre les L2. Par exemple, pour une mémoire de 16 Go et une architecture à 16 clusters, chaque L2 cache un segment mémoire de 1 Go.

2.1.2 Partitionnement de l'espace d'adressage physique

L'architecture considérée dispose d'un espace d'adressage physique de 40 bits. Sur ces 40 bits, seuls les 8 bits de poids fort sont utilisés pour identifier un cluster : les bits 33 à 36 représentent l'abscisse du cluster, tandis que les bits 37 à 40 représentent l'ordonnée. Ce qui permet d'adresser jusqu'à 256 clusters. Quant aux 32 bits de poids faibles, ils sont utilisés pour l'adressage local à un cluster, ce qui fait que chaque cluster dispose d'un espace d'adressage local de 4 Go.

2.1.3 Accès à l'espace physique

Dans l'architecture TSAR, il existe deux mécanismes matériels permettant d'accéder à l'espace d'adressage physique 40 bits, bien que le compilateur du processeur MIPS32 ne génère que des adresses 32 bits.

Accès au travers de la mémoire virtuelle

Le premier mécanisme est celui de la mémoire virtuelle paginée. Les adresses de 32 bits reçues d'un cœur sont traduites en adresses 40 bits. Les informations nécessaires pour effectuer la traduction sont fournies par le SE au travers d'une ou plusieurs table(s) des pages stockées en mémoire. Pour éviter le coût des accès mémoire, les traductions sont cachées dans les TLBs. Enfin, en plus de la cohérence des TLB, les automates matériels gèrent aussi la résolution des MISS TLB.

Accès direct en adressage physique

Le second mécanisme est spécifique à l'architecture TSAR. Il permet aux cœurs d'accéder directement à l'espace physique 40 bits, grâce à deux registres d'extension d'adresse, également situés dans le contrôleur de cache L1. En écrivant dans ces registres, le logiciel

peut explicitement spécifier les 8 bits de poids fort qui seront concaténés à l'adresse 32 bits pour former une adresse physique sur 40 bits. Rappelons que ces 8 bits définissent en pratique le cluster cible. Le premier registre permet de contrôler les adresses d'instructions. Le second permet de contrôler les adresses de données. Ils sont accédés au moyen des instructions `mfc2` et `mtc2`, qui permettent respectivement de lire et d'écrire dans les registres de la MMU.

2.2 Le noyau ALMOS

ALMOS est un noyau de type UNIX développé au sein du LIP6 par Ghassan Almaless [5]. Ce noyau cherche à répondre au problème du passage à l'échelle des systèmes d'exploitation respectant le standard POSIX sur les architectures CC-NUMA manycore et plus particulièrement sur l'architecture TSAR.

2.2.1 Architecture distribuée

Afin de permettre le passage à l'échelle du noyau, ALMOS utilise plusieurs techniques de distribution ou de réplication des données. L'objectif de ces techniques est double : elles permettent d'une part de favoriser les accès mémoire locaux (c'est-à-dire internes à un même cluster) pour minimiser la latence et la consommation électrique ; elles permettent d'autre part de décentraliser les accès afin de réduire les goulots d'étranglement qui peuvent se produire lorsque les accès sont centralisés sur un même cluster.

Cluster manager

Dans le noyau ALMOS, chaque cluster dispose de son propre gestionnaire de ressources, appelé *cluster-manager*. Ce dernier est une structure qui permet de regrouper les ressources relatives à un cluster donné. Elle contient, entre autres :

- un gestionnaire de mémoire qui regroupe les informations nécessaires à l'allocation de la mémoire ;
- N gestionnaires de cœurs, N étant le nombre de cœurs du cluster. Chacune de ces structures regroupe les informations relatives à un cœur telles que l'ordonnanceur.

DQDT

Le *cluster-manager* présente un inconvénient important. En localisant les informations à chaque cluster, la vue globale des ressources du système est perdue. Cette vue est nécessaire pour implémenter des stratégies de placement de tâches ou d'allocation mémoire efficace lorsque l'allocation est distante. Pour pallier cet inconvénient, ALMOS utilise une structure appelée DQDT pour *Distributed Quaternary Decision Tree*. Cette structure est un arbre d'arité quatre (voir figure 2.2). Chaque niveau de l'arbre regroupe les informations relatives à un cluster ou à un groupe de clusters (voir figure 2.3). Chaque feuille de l'arbre correspond à un cluster physique. En remontant dans l'arbre, chaque

nœud correspond à un groupe de 4, 16, 64 ou 256 clusters. Pour améliorer les temps d'accès, les nœuds de l'arbre sont distribués sur les bancs de mémoire, et la mise à jour se fait sans verrou.

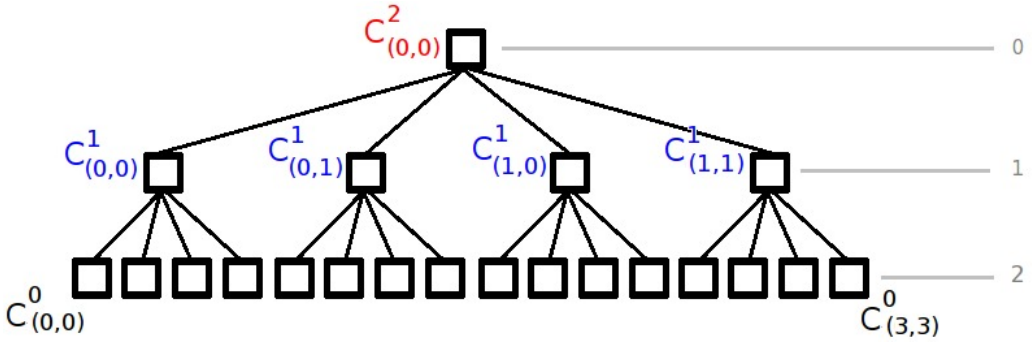


FIGURE 2.2 : L'arbre DQDT. Source : [5]

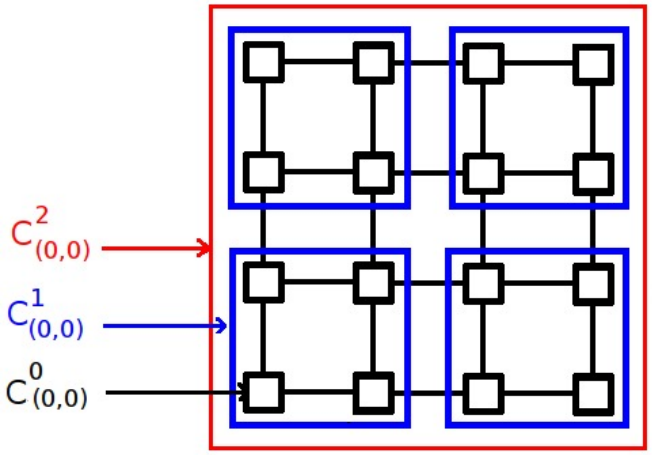


FIGURE 2.3 : Découpage d'une architecture à 16 clusters physiques en clusters logiques. Source : [5]

Gestionnaire d'évènements

Les échanges d'informations entre clusters peuvent se faire de deux manières différentes : soit par accès direct (`read/write`) aux données distantes, soit au travers d'un mécanisme client-serveur. Ce dernier est plus efficace lorsque le nombre d'accès à la mémoire distante est plus grand que le nombre d'accès nécessaires au postage d'une requête au serveur distant. L'efficacité de ce mécanisme vient du fait qu'il minimise les accès distants. En effet, hormis les accès nécessaires pour poster une requête, tous les autres accès sont locaux.

Initialement utilisé uniquement pour les opérations de réveil de tâches distantes, le mécanisme client-serveur a été généralisé sous la forme d'un gestionnaire d'évènements. Ce gestionnaire a été utilisé pour d'autres opérations distantes, telles que créer, dupliquer ou migrer une tâche. Il a aussi été utilisé pour des opérations locales afin de différer leurs exécutions, tel que pour libérer la mémoire d'une tâche terminée (une tâche qui se termine ne peut libérer certaines de ses structures de données, comme la pile, car celles-ci sont utilisées tant que la tâche s'exécute).

Réplica noyau

L'ensemble du code noyau est répliqué dans chaque cluster. Cette réplication est effectuée en s'appuyant sur le mécanisme de la mémoire virtuelle. Au démarrage du système, l'ensemble du code noyau est répliqué dans l'espace physique de chaque cluster. Puis, suivant le placement des processus dans un des clusters, une même adresse virtuelle est associée à différentes adresses physiques (voir la sous-section 2.2.3). Ainsi, chaque processus accède à une copie du code noyau qui est local au cluster où il se trouve.

2.2.2 Processus hybrides

Le mécanisme des réplicas noyau présenté ci-dessus présente une limitation importante lorsqu'un processus est constitué de plusieurs threads, se trouvant dans des clusters différents. Dans ce cas tous les threads utilisent le même code de noyau. En effet, les threads d'un même processus partagent par définition le même espace d'adressage et donc la même table de page. Ainsi, seuls les threads qui se trouvent dans le cluster T contenant la table de pages du processus utilisent le code noyau local. Les autres threads qui se trouvent dans d'autres clusters utilisent le code distant se trouvant dans le cluster T. Dans ALMOS, ce problème a été résolu en introduisant le concept des processus hybrides. Il consiste à permettre à chaque thread d'avoir sa propre table de pages. Ainsi, chaque thread peut utiliser la copie du code noyau local à son cluster.

L'utilisation du réplica noyau local par les threads n'est qu'un exemple des avantages des processus hybrides. Il y a d'autres avantages, décrits ci-dessous :

- La localisation du code utilisateur des threads de processus. Puisque l'ensemble des threads d'un même processus se partage le même code utilisateur, le concept de processus hybride permet d'utiliser un réplica local de ce code (semblable au mécanisme du réplica noyau).
- La distribution des tables des pages. Lorsque la table de pages est partagée par tous les threads d'un même processus, un goulot d'étranglement se crée. Avec les processus hybrides, ce goulot est allégé, puisque chaque thread utilise une table de page différente.
- L'agrandissement de l'espace virtuel des processus. Puisque chaque thread a sa propre table de pages, les régions de l'espace virtuel privées à un thread (telles que la pile d'exécution) peuvent être utilisées pour projeter des contenus différents. Ceci est similaire au mécanisme de réplica du code, sauf que cette fois-ci non seulement la projection est différente, mais le contenu physique projeté est lui aussi différent. Ceci permet d'économiser l'espace virtuel (une seule et même région est utilisée pour mapper les contenus privés des différents threads), mais aussi de permettre l'accès à un espace physique plus grand que l'espace virtuel 32 bits.
- Les processus hybrides permettent une gestion plus fine des pages accédées par les threads. Cela permet d'optimiser les stratégies de placement et de migration des pages utilisées par les threads.

Même si certains inconvénients des processus hybrides existent (tels que la nécessité de mettre en place un mécanisme permettant d'assurer la cohérence des régions virtuelles partagées ou la consommation en espace mémoire pour le stockage des tables de pages), les travaux effectués par Ghassan Almaless durant sa thèse ont montré que les avantages de ce concept sont plus importants que les inconvénients dans le cas des architectures manycores.

2.2.3 Espace virtuel du noyau

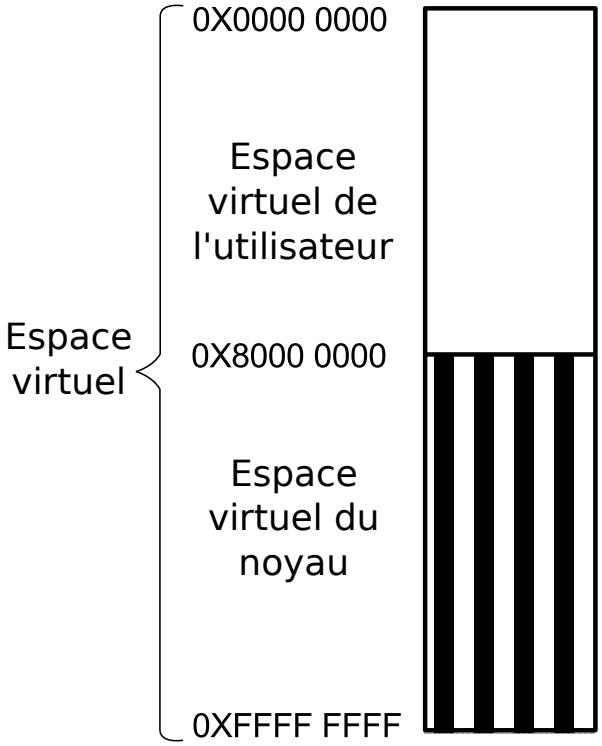


FIGURE 2.4 : Distribution en deux de l'espace virtuel d'un processus dans le noyau ALMOS à 32 bits. La partie hachurée est utilisée par le noyau. L'autre partie est utilisée par l'utilisateur.

Dans ALMOS, comme dans les systèmes monolithiques classiques (tel que NetBSD ou Linux) l'ensemble du code système s'exécute en mode privilégié dans un espace d'adressage qui est le même pour tous les processus. Puisque les cœurs utilisés sont 32 bits, l'espace virtuel accessible par un processus est limité à 4 Go. L'approche classique partage cet espace en deux parties (voir La figure 2.4). La première est réservée au processus utilisateur. La seconde est réservée au noyau. Cette approche est aussi utilisée par ALMOS. Les deux parties de l'espace virtuel sont alors découpées en parts égales de 2 Go (sur d'autres SE, on trouve aussi un découpage à 3 Go pour l'espace utilisateur et 1 Go pour le noyau).

2.2.4 Espace adressable physique

Le premier prototype d'ALMOS a été développé pour une version de l'architecture TSAR utilisant des adresses physiques sur 32 bits. Cette première implémentation était alors suffisante pour démontrer l'efficacité des mécanismes proposés par ALMOS (cluster-managers distribués,

réplicas noyau, processus hybrides, etc.), et a permis d'exécuter des applications parallèles sur des architectures contenant jusqu'à 512 cœurs.

Cette limitation n'est évidemment pas acceptable pour une architecture manycore contenant plusieurs centaines de cœurs, et elle doit absolument être levée pour implémenter un système de fichiers qui passe à l'échelle.

2.3 Les structures du système de fichiers

Dans cette section, nous décrivons, de façon générique, les structures classiques du système de fichiers des SE de type UNIX. Cette description est limitée aux problèmes traités. Ces structures sont présentées par la figure 2.5.

2.3.1 Structures des fichiers ouverts

En premier, on trouve deux structures qui permettent de gérer l'ensemble des fichiers ouverts par un processus. Le terme fichier, au sens des systèmes UNIX, décrit non seulement les fichiers normaux, mais aussi tous types de flux de données, tels que les `sockets` ou les `pipes`.

La première structure, la structure `file`, représente l'état du fichier ouvert, tel que la position de la tête de lecture dans le fichier (le champ `offset`) ou les droits d'accès (en lecture ou en écriture). On retrouve aussi le champ `flops` qui contient un ensemble de fonctions (`read`, `write`, `llseek`, etc) implémentant l'interface d'accès. Cette interface permet - puisque tous les accès aux fichiers ouverts se font au travers eux - d'uniformiser l'accès aux différents types de fichiers (les fichiers, les `sockets`, les `pipes`, etc.).

La deuxième structure est la table des descripteurs de fichier ouverts par le processus, qui contient les pointeurs vers les structures `file`. Un descripteur de fichier est donc un entier entre 0 et N, où N dépend du système d'exploitation (typiquement 1024).

Remarque : Le standard POSIX impose de retourner à l'utilisateur le descripteur le plus petit non encore utilisé lors d'une demande d'allocation (appels système `open`, `dup`, etc.).

Pour être fonctionnelle, la table des descripteurs doit supporter quatre opérations :

1. l'opération **rechercher**
   permet de trouver une structure `file` à partir du descripteur de fichier ;

2. l'opération **insérer**
   permet d'insérer une structure `file` dans une nouvelle entrée du tableau ;

3. l'opération **supprimer**
   supprime une entrée du tableau et libère la structure `file` ;

4. l'opération **dupliquer**
   duplique une entrée du tableau de façon à ce qu'une même structure `file` soit accessible

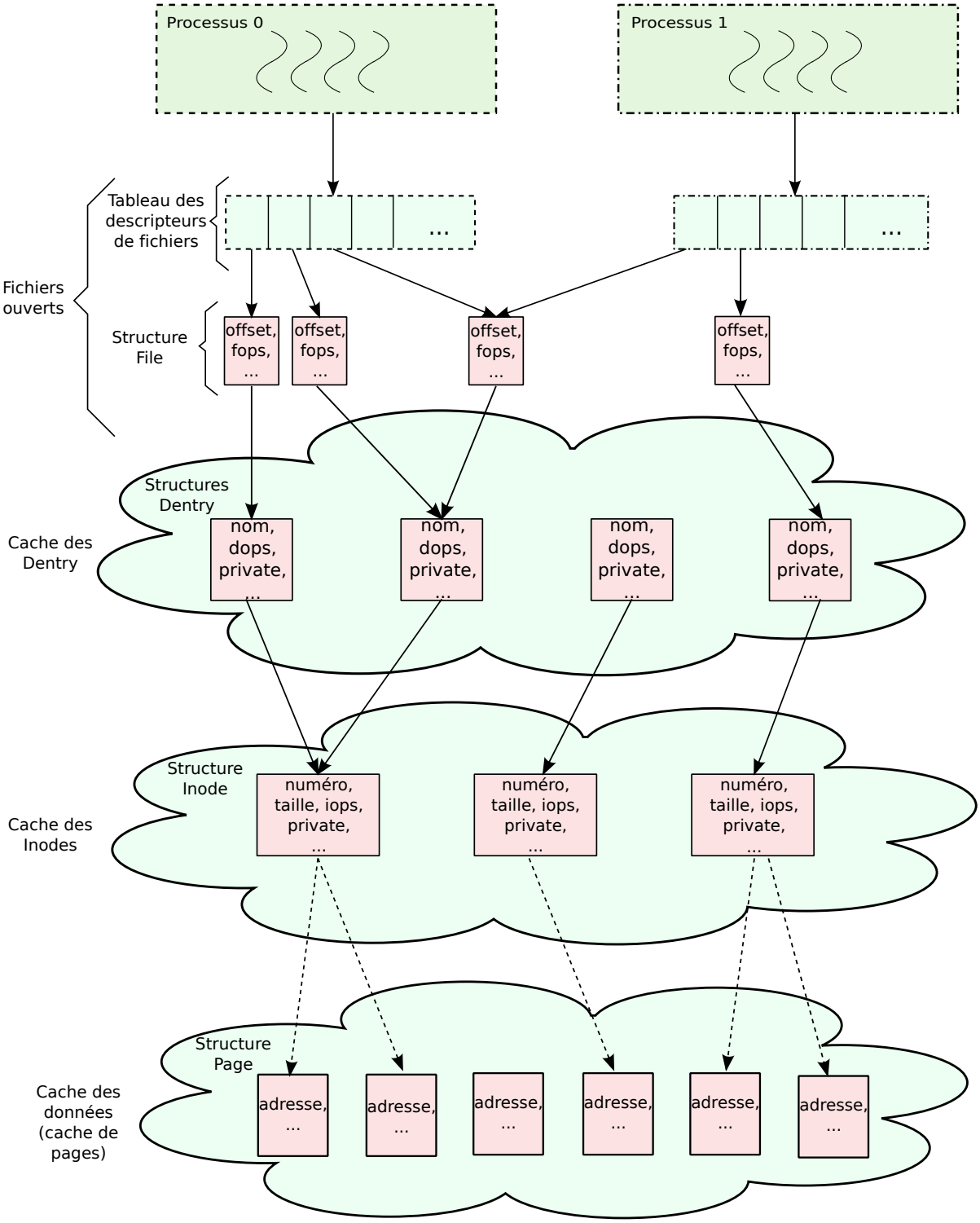


FIGURE 2.5 : Les principales structures de données considérées.

par deux entrées de la table. Cette opération permet d'implémenter les appels système `dup` et `dup2`.

Notons, que la table des descripteurs de fichiers est privée à un processus et partagée entre les threads du processus. Quant à la structure `file`, elle est généralement privée à un processus, mais peut être partagée entre processus suite à une opération `fork` (opération POSIX permettant de créer un processus fils). En effet, lors d'un `fork` le fils hérite des fichiers ouverts par le père. Un compteur de référence, par structure `file`, permet qu'elle ne soit libérée que lorsqu'elle n'est plus utilisée. Les autres structures du système de fichiers sont partagées par tous les processus/threads du système.

2.3.2 Structures des caches de fichiers

En second, on trouve les structures permettant de cacher le contenu du disque : le cache des données et le cache des métadonnées, également appelé cache des *inodes* et des *dentrys*.

Ces caches ont deux rôles. Le premier rôle est d'accélérer les accès au système de fichiers en remplaçant les accès au disque extrêmement lents par des accès mémoire. Le second rôle est de synchroniser l'accès au contenu du système de fichiers, puisque le système de fichiers partagé par tous les processus du système. Par exemple, lorsqu'un fichier est déjà ouvert par un processus A et qu'un autre processus B demande l'ouverture du même fichier, ce dernier doit accéder au fichier qui a été précédemment chargé depuis le disque par le processus A et non pas charger de nouveau une autre copie du fichier.

Structures du cache des métadonnées

Le cache des métadonnées permet de retrouver toutes les informations relatives à un fichier ou à un répertoire, telles que le nom, la taille du fichier, les droits d'accès, l'emplacement sur le disque, et l'emplacement en mémoire (dans le cache des données). Les noms sont stockés dans une structure à part, la structure *dentry*), et le reste des informations dans une autre structure, la structure *inode*. Cette séparation est nécessaire pour supporter la fonctionnalité des systèmes de fichiers d'UNIX qui permet à un même fichier d'avoir plusieurs noms différents (alias, *hard link*).

L'ensemble des structures *dentrys* forme un arbre, puisqu'il reproduit un extrait de l'arborescence du système de fichiers sur le disque. Le nœud racine et les nœuds intermédiaires sont des répertoires, tandis que les feuilles sont des fichiers (ou des répertoires vides).

Un *inode* peut être lié à un ou plusieurs *dentrys* (alias). On dit alors que le *dentry* pointe vers cet *inode*. Un *dentry* a toujours un *dentry* père (répertoire contenant le nom du fichier).

Les principales opérations qui affectent ce cache sont :

1. l'opération de **résolution du chemin d'accès**
   Cette opération est la plus importante du cache de métadonnées. Elle est utilisée par l'ensemble des appels système ayant un chemin d'accès comme argument : `open`, `unlink`,

`rmdir`, `rename`, `creat`, etc. Le chemin d'accès, ou cheminom, est composé de plusieurs noms de répertoires. La résolution de celui-ci consiste à trouver l'*inode* se trouvant en fin du chemin en passant par les différents *dentrys/inodes* de l'arborescence, tout en vérifiant à chaque *inode* traversé les droits d'accès. La résolution du chemin d'accès est toujours effectuée à partir d'un *inode stable* : un nœud qui ne peut pas être supprimé en cours d'utilisation, tel que l'*inode* du répertoire racine du système de fichiers.

2. l'opération de **résolution inverse**
   Cette opération permet à partir d'un *inode/dentry* de retrouver le chemin d'accès absolu. Elle est utilisée par les appels systèmes tel que `getcwd`.

3. l'opération d'**insertion**
   Cette opération permet d'insérer un *dentry* représentant un fichier ou répertoire dans l'arborescence du système de fichiers. L'*inode* et le *dentry* sont aussi insérés dans le cache, s'ils n'étaient pas déjà présents. Cette opération est utilisée par l'opération de résolution de noms pour insérer les *dentrys* absents dans l'arborescence, mais présents sur disque, mais aussi par les appels systèmes permettant de créer un nouveau fichier ou répertoire, tels que `creat` et `mkdir`.

4. l'opération de **suppression**
   Cette opération permet de supprimer un *dentry* de l'arbre. Seules les feuilles de l'arbre peuvent être supprimées, c.-à-d. les fichiers ou les répertoires vides. Cette opération est utilisée lors de l'évincement des entrées du cache, mais aussi par les appels systèmes permettant de supprimer un fichier ou un répertoire, tels que `unlink`/`rmdir`. L'*inode* est aussi supprimé sauf s'il a un autre nom (alias).

   Enfin, notons que le standard POSIX laisse à l'implémentation le choix de permettre ou non la suppression des fichiers ou répertoires en cours d'utilisation.

5. l'opération de **renommage**
   Cette opération permet de renommer un *dentry*. Le renommage change l'emplacement du *dentry* dans l'arbre représentant le système de fichiers, sans changer le lien entre ce *dentry* et l'*inode* vers lequel il pointe. Cette opération est utilisée par l'appel système `rename`.

6. l'opération de **création d'alias**
   Cette opération consiste à créer un *dentry* qui pointe vers un *inode* existant. Elle est utilisée par l'appel système `link`.

Plusieurs implémentations existent pour ce cache des métadonnées. La plus répandue consiste à utiliser deux tables de hachage. Une table pour les *dentrys* et une autre pour les *inodes*. Les *dentrys* sont recherchés à partir du nom du fichier/répertoire et d'un identifiant du parent (telle que l'adresse du *dentry* parent). Les *inodes* sont recherchés à partir du numéro d'*inode*.

En plus des tables de hachage, on retrouve généralement des listes chaînées externes qui permettent de regrouper les nœuds d'un certain type. Pour les *dentrys*, on retrouve une liste

qui permet de collecter tous les *dentrys* inutilisés : ce type de liste est appelée *freelist*. Pour les *inodes*, on a une *freelist* et une liste des *inodes* `dirty`, c'est-à-dire des *inodes* dont le contenu n'est plus à jour avec le disque.

Structures du cache des données

Le cache des données permet de cacher en mémoire le contenu des fichiers ou des répertoires stockés sur le disque dur. La granularité de ce cache étant la page (typiquement 4 Koctets), ce cache est aussi appelé cache de pages.

Une implémentation classique de ce cache consiste à utiliser deux structures. La première est une structure de recherche qui permet d'indexer la seconde, la structure `page`.

La structure `page`, ou plus précisément *descripteur de page*, contient essentiellement l'adresse de base de la page ainsi que l'état des données contenues, tel que l'état de modification de ces données par rapport à la copie sur disque.

Pour la structure de recherche, on trouve deux types d'implémentation. La première est basée sur l'utilisation d'un arbre privé pour chaque *inode* (c.-à-d. pour chaque fichier ou répertoire). Les feuilles de cet arbre pointent sur les structures `pages`, et on y accède avec la valeur de l'*offset* dans le fichier. La seconde implémentation consiste à utiliser une table de hachage partagée par tous les *inodes*, semblable à celles du cache des métadonnées.

2.3.3 Méthodes d'accès aux données des fichiers

Il existe deux méthodes permettant à une application d'accéder au contenu d'un fichier ouvert. La première utilise les opérations d'entrée/sortie de type `read` et `write`, qui réalisent un transfert entre le cache des pages géré par le système et un tampon mémoire dans l'espace utilisateur. La seconde consiste à projeter, dans l'espace virtuel utilisateur, les pages du cache du système. Cette projection se fait au moyen de l'opération `mmap`.

Les opérations d'entrées/sorties peuvent être découpées en trois catégories. En premier, on trouve les opérations `read/write` qui permettent d'accéder aux données du fichier en utilisant l'*offset* courant du fichier. En deuxième, on trouve les opérations `pread/pwrite` qui permettent d'accéder aux données du fichier à partir d'un *offset* donné en argument. En troisième, on trouve les opérations `readv/writev` qui permettent d'accéder à des données non contiguës du fichier.

Quant à l'accès par projection `mmap`, deux modes existent. Dans le mode partagé, l'accès se fait directement dans le cache des pages, et toutes les modifications effectuées sont visibles par tous les threads ou processus ayant ouvert le fichier. Dans le mode privé, l'accès se fait à une copie privée du fichier (à noter que la technique *copy-on-write* peut être utilisée pour retarder la copie). En plus de ces modes, la projection peut se faire avec différentes protections, tel qu'en lecture seule ou en lecture/écriture.

2.4 Problèmes identifiés

L'architecture TSAR introduit trois types de problèmes à résoudre pour assurer le passage à l'échelle des structures de données du système de fichiers : la dimension des structures que le système doit gérer, le placement de ces structures dans les bancs de mémoire et enfin l'accès à ces structures par un grand nombre de threads.

2.4.1 Problème de la capacité d'adressage du noyau

Le premier problème est lié au choix fait par les architectes de TSAR d'utiliser des cœurs 32 bits pour réduire la consommation d'énergie. L'utilisation de ce type de cœurs implique que l'espace d'adressage virtuel du noyau est limité à 1 ou 2 Go (voir 2.2.3).

Or, la capacité du cache des pages est critique, car elle permet non seulement d'améliorer les performances des opérations d'entrées/sorties mais aussi de réduire le goulot d'étranglement lié à l'accès au disque dur. Dans les SE modernes de type UNIX, l'ensemble des données peuvent être classées en deux catégories :

En premier lieu, on trouve les données allouées dynamiquement, telles que les descripteurs de processus, les descripteurs de régions virtuelles, les tables des pages ou le cache des pages du système de fichiers. Ce dernier est celui qui requiert le plus d'espace mémoire. Pour une machine multicœur, ce cache nécessite plusieurs dizaines de Go. Par exemple sur un serveur de notre laboratoire possédant 8 cœurs, et disposant d'une mémoire physique de 128 Go, 108 Go sont utilisés par le cache des pages (chiffre obtenu sur Linux avec la commande `vmstat`). Cette taille des caches devrait être encore plus importante pour des systèmes manycores qui peuvent exécuter plus d'applications en parallèle.

En second lieu, on retrouve les données statiquement allouées, telles que les données globales du noyau, les tables de hachage globales du système ou les descripteurs de pages physiques. Ces derniers peuvent poser un sérieux problème à l'exploitation d'une grande mémoire physique, puisque leur encombrement est proportionnel à la mémoire physique supportée : il faut un descripteur par page physique. Par exemple, en supposant une mémoire de 256 Go, les descripteurs de pages consomment, à eux seuls (sans le contenu des pages qu'elles décrivent), environ 512 Mo d'espace mémoire, ce qui laisse très peu de place pour les autres structures du noyau. Ces descripteurs de pages ne sont pas seulement utilisés par le système de fichiers, mais sont également utilisés par d'autres sous-systèmes. En particulier, le système mémoire les utilise pour implémenter l'allocateur mémoire `Buddy` qui permet, entre autres, de limiter la fragmentation externe de la mémoire.

Il ressort de cette analyse que les structures de données utilisées par le système de fichiers, et en particulier par le cache des pages, requièrent plusieurs dizaines de Go d'espace mémoire, ce qui pose un difficile problème d'adressage, avec un espace virtuel limité à 1 ou 2 Go normalement accessible avec des cœurs 32 bits.

Notons que ce problème a été initialement identifié pour les structures du système de fichiers. Mais il est clair que le problème se pose pour beaucoup d'autres structures de données du noyau.

La solution proposée doit donc être une solution générale, permettant au système d'adresser facilement un espace d'adressage physique de 1 Toctets, non nécessairement contigu, puisque dans TSAR les segments de mémoire physiques distribués dans les clusters ne sont pas adjacents.

Ainsi, la question posée est : comment permettre aux différentes structures du noyau d'exploiter un espace adressable physique non contigu de 1 To, tout en utilisant des processeurs 32 bits possédant un espace adressable virtuel de 4 Go ?

2.4.2 Problème du placement des données

L'aspect NUMA de l'architecture TSAR introduit la question du placement des données du système de fichiers. Un placement idéal serait un placement où tous les accès sont locaux. Toutefois, un tel placement est difficile – pour ne pas dire impossible –, car les structures du système de fichiers sont partagées par tous les threads (c.-à-d. tous les cœurs) du système.

Sachant que les accès ne peuvent être tous locaux, la stratégie de placement devra se concentrer sur un objectif, consistant à minimiser les contentions matérielles au niveau des bancs de mémoire distribués.

La contention survient lorsqu'un grand nombre de cœurs essaient d'accéder simultanément à un même banc de mémoire. Pour qu'il y ait contention, le nombre d'accès simultanés doit être supérieur au nombre de cœurs d'un cluster : en effet, les architectures NUMA sont généralement construites de façon à ce que les cœurs d'un même cluster puissent tous accéder au banc mémoire local sans créer de contention.

Dans cette étude, nous considérons uniquement les stratégies de placement statiques. Ceci signifie que nous ne considérerons pas les solutions utilisant la migration des données. Toutefois, les solutions utilisant la réplication des données restent considérées si aucun mécanisme de cohérence n'est nécessaire (données en lecture seule).

Finalement, la question posée est : peut-on définir une stratégie de placement statique des données partagées du système de fichiers, en particulier le cache des pages, permettant de minimiser les contentions d'accès aux bancs de mémoire ?

2.4.3 Problème des accès concurrents

Le dernier problème est lié à l'aspect manycore de l'architecture TSAR. Le fait d'avoir plusieurs cœurs implique qu'il y a plusieurs threads qui s'exécutent en parallèle. Ainsi, il est nécessaire d'assurer la cohérence des structures partagées du système de fichiers lors des accès concurrents. Ce problème n'est pas nouveau et il existe déjà des solutions, comme l'utilisation de verrous, pour synchroniser les accès. La difficulté est de garantir le passage à l'échelle. Pour que la solution soit adaptée à l'architecture manycore, il faut qu'elle passe à l'échelle à la fois en performance et en quantité de mémoire utilisée.

Ainsi la question posée est : comment assurer la cohérence des structures partagées du système de fichiers sans que les accès concurrents de plusieurs centaines de threads aux

mêmes structures de données deviennent un goulot d'étranglement ?

Détail du problème

Pour ce problème, nous supposons l'existence d'instructions atomiques de lecture et d'écriture de mot ainsi que l'existence d'une instruction atomique de type CAS (voir 2.1.1). Le problème peut être découpé en deux sous problèmes (un découpage similaire a été décrit dans [24]) :

Problème de l'utilisation après libération

Ce problème se produit lorsqu'un thread A obtient une référence (une adresse) à une structure de données partagée, et qu'avant que A n'accède à son contenu, un thread B la supprime et la libère. Ainsi, le thread A accédera à des données qui ne sont plus valides. Le Tableau 2.1 montre un scénario où ce problème peut survenir.

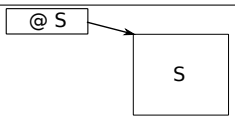
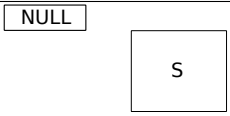
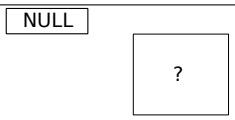
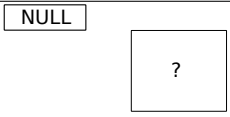
Opérations du thread A	Opérations du thread B	États de la structure et de la référence
Lecture de la référence valeur stockée : @ S		
valeur stockée : @ S	Suppression de la structure : l'adresse est mise à nulle	
valeur stockée : @ S	Libération de la structure	
Accès invalide à la structure S		

TABLE 2.1 : Exemple de scénario du problème de synchronisation : cas où un thread A est en train d'accéder à une structure de données alors que celle-ci a été supprimée et libérée en parallèle par un thread B.

Problème de l'incohérence des données

Ce problème survient lorsque des données sont accédées par un premier thread en écriture alors qu'un second thread y accède en lecture ou en écriture. Dans ce cas, le second thread peut lire des données incomplètes, dans le cas où il accède en lecture. S'il accède en écriture, les données écrites peuvent être incohérentes, car formées de plusieurs écritures distinctes.

On peut noter que la résolution du premier problème est suffisante pour assurer la concurrence d'accès dans le cas où il y a un seul écrivain, tant que les écritures de mot sont atomiques ce qui est le cas sur la plupart des architectures. Dans le cas où l'on a plusieurs écrivains ou lorsque les écritures ne sont pas atomiques, il est nécessaire de résoudre le second problème.

2.5 Conclusion

Ce chapitre a présenté le contexte de l'étude ainsi que la problématique de la thèse. Les trois principales questions auxquelles nous allons essayer de répondre dans cette thèse sont les suivantes :

1. Comment permettre aux différentes structures du noyau d'exploiter un espace adressable physique non contigu de 1 To tout en utilisant des processeurs 32 bits possédant un espace adressable virtuel de 4 Go ?
2. Quelle stratégie de placement des données partagées du système de fichiers permet-elle de minimiser les contentions aux bancs de mémoire distribués dans une architecture CC-NUMA telle que TSAR ?
3. Comment assurer la cohérence des structures partagées du système de fichiers sans que les accès concurrents de plusieurs centaines de threads deviennent un goulot d'étranglement ?

CHAPITRE 3

État de l'art

Dans ce chapitre, nous analysons les solutions utilisées dans les systèmes d'exploitation existants pour résoudre les trois problèmes identifiés dans le chapitre 2 : accès à un espace physique plus grand que l'espace virtuel du processeur, stratégies de placement dans les architectures NUMA, et enfin accès concurrents aux structures de données. Les systèmes analysés peuvent être classés en deux grandes catégories :

Les SE formés par une seule instance de noyau

Ces SE supposent, généralement, que tous les processeurs partagent le même espace d'adressage physique. La mémoire peut être centralisée (dans les architectures SMP), ou physiquement distribuée (dans les architectures NUMA). Cette catégorie inclut les systèmes de type UNIX ou Windows, ainsi que des projets de recherche tels que K42/Tornado [32, 24] (K42 est le successeur de Tornado).

Les principaux systèmes que nous avons analysés sont : Linux, FreeBSD, NetBSD et Solaris. Linux a été particulièrement étudié en raison des nombreux travaux portant sur la scalabilité du système. Dans la suite, nous qualifierons ce premier type de système *mono-instance*.

Les SE formés par plusieurs instances communicantes

Dans ces systèmes, chaque instance du noyau contrôle un espace d'adressage privé, non directement accessible par les autres instances du noyau.

L'existence de plusieurs instances n'est pas visible par les applications utilisateur qui ont l'illusion d'un seul système gérant l'ensemble des ressources de la machine : *single system image* (SSI). La communication entre les différentes instances se fait généralement grâce à des mécanismes de passage de messages ou d'appel de procédures distantes (*RPC* pour *Remote Procedural Call*).

Ce type de système a plusieurs appellations : *Hiearchical clustering* [58], *Multicellular* [20], *Multikernel* [8], *Factored operating system* [62] ou *replicated-kernel* [7]. Pour la suite, nous qualifierons ce second type de système *multi-instances*.

Remarque 1 : cette classification est indépendante du fait que l'architecture du noyau est monolithique, ou construite autour d'un microkernel, puisqu'on retrouve ces deux architectures dans les deux catégories définies ci-dessus.

Remarque 2 : les SE qui définissent une API autre que POSIX ne seront pas mentionnés, tels que ROS [31], Corey [14], Tessalation [39] ou Helios [46].

Le reste de ce chapitre est structuré en trois grandes sections. La première traite du problème de gestion d'un espace physique étendu. C'est dans cette section que l'on décrit les solutions des SE multi-instances. Les deux autres sections présentent les solutions existantes aux problèmes de placement et d'accès concurrents aux structures du système de fichiers. Ces deux dernières sections se concentrent sur les solutions des SE mono-instances.

3.1 Accès à un espace physique plus grand que l'espace virtuel

Nous analysons dans cette section différentes techniques permettant au système d'exploitation de gérer un espace d'adressage physique plus grand que 4 Go dans des architectures construites autour de processeurs 32 bits.

Nous présentons brièvement les techniques utilisées dans les systèmes mono-instances, et nous analysons plus en détail les techniques de communication utilisées par les systèmes multi-instances, puisque c'est une approche multi-instances que nous avons retenue pour le support des adresses 40 bits dans ALMOS.

3.1.1 Processeurs 32 bits supportant un espace d'adressage étendu

L'architecture TSAR n'est pas la seule architecture à posséder un espace d'adressage physique plus grand que l'espace virtuel du processeur 32-bits utilisé. On trouve, dans les processeurs 32-bits d'`Intel` ou d'`AMD`, le mécanisme PAE [54, 2], pour *Physical Address Extension*, qui étend l'espace d'adressage physique à 36 bits. De même, dans certains processeurs ARM [3], le mécanisme LPAE étend l'espace d'adressage physique à 40 bits. Toutefois, pour tous ces processeurs 32 bits, le seul moyen d'accéder à l'espace d'adressage physique reste la MMU, et l'espace virtuel reste limité à 4 Goctets.

3.1.2 Systèmes mono-instances

On trouve dans les systèmes monolithiques commerciaux (Linux, FreeBSD, Windows [51]) différentes techniques permettant d'exploiter un espace d'adressage physique plus grand que 4 Goctets.

Le principe de base consiste à découper la mémoire virtuelle du noyau (1 Go) en deux régions : une région statique et une région dynamique. La région statique permet de projeter de façon permanente les données principales et le code du noyau. La région dynamique permet justement d'accéder aux régions de l'espace physique qui ne sont pas projetées de façon permanente dans l'espace virtuel du noyau.

Pour accéder à une région de l'espace physique non projetée dans l'espace virtuel, le noyau doit d'abord allouer une page virtuelle (une entrée dans la table des pages) pour projeter temporairement la page physique que l'on veut accéder. Une fois la projection effectuée, les données peuvent être temporairement accédées en utilisant l'adresse virtuelle allouée. Enfin, une fois l'accès terminé la page virtuelle est libérée.

Toutefois, ce type de solution n'est utilisé que pour des structures de grande taille, où les accès ne sont pas très fréquents, telles que les données du cache de pages. Les autres structures de données (les descripteurs de pages physiques, les *dentrys*, les *inodes*, les descripteurs de tâches, etc.) restent stockées dans la région statique et sont donc limitées par la taille de l'espace virtuel du noyau. C'est cette limitation qui impose que ces noyaux ne puissent supporter plus de 64 Go de mémoire physique quand ils s'exécutent sur des processeurs 32 bits.

La principale limitation concerne la structure contenant les descripteurs de pages physiques, puisque l'encombrement de cette structure augmente linéairement avec la taille de la mémoire physique supportée (il y a une structure page pour chaque page physique). Par exemple, en considérant que les descripteurs de page ont une taille de 32 octets (taille de la structure dans Linux) et que la mémoire physique est de 64 Go, la quantité de mémoire consommée par ces descripteurs est de 500 Mo, ce qui représente la moitié de l'espace virtuel du noyau (en supposant que l'espace virtuel du noyau est de 1 Go). Cela laisse très peu d'espace pour les autres structures. De plus, ce type de solution à région dynamiquement allouée induit un coût en performance dû à la nécessité de modifier dynamiquement la table des pages.

Une solution a été proposée pour le noyau Linux, mais n'a pas été adoptée. Elle consiste à utiliser deux tables de pages. Une pour l'exécution du processus en mode utilisateur. L'autre pour l'exécution en mode noyau. L'avantage de cette solution est que le noyau et l'utilisateur peuvent accéder chacun à 4 Go d'espace virtuel. Toutefois, cette proposition ne fait que repousser le problème, puisqu'elle permet de passer simplement de 1 Go à 4 Go. En outre, cette solution est très coûteuse en performance, puisqu'elle nécessite de vider le cache TLB à chaque appel système.

On peut noter que le problème est un peu différent dans les systèmes construits autour d'un microkernel. Dans ce cas, le système de fichier s'exécute en espace utilisateur, et dispose donc de 3 Go d'espace virtuel (au lieu de 1 Go), ce qui reste néanmoins insuffisant. La mise en place d'une stratégie dynamique (similaire à celle des noyaux monolithiques) peut s'avérer plus coûteuse, car la manipulation directe de la table des pages n'est pas possible en mode utilisateur : il faut donc envoyer un message au serveur gérant le système mémoire.

Le système K42, qui est basé sur un micro-noyau, ne supporte que les processeurs 64-bits ce qui supprime le problème [32], puisque l'espace virtuel est suffisamment grand pour couvrir l'ensemble de la mémoire physique.

3.1.3 Systèmes multi-instances

Les premiers systèmes d'exploitation multi-instances ont été développés dans les années 90, dans le cadre du développement des premières architectures multiprocesseurs à mémoire partagée à accès non uniforme (CC-NUMA) telles que DASH [36], KSR1 [23], Hector [61], FLASH [34].

Cette structure multi-instances a été reprise à partir de 2009 pour permettre le passage à l'échelle, mais aussi pour supporter des architectures hétérogènes (et pas seulement des architectures CC-NUMA homogène). L'un des premiers travaux effectués dans ce cadre est *Barrelfish* [8]. La principale nouveauté de ce système (par rapport à ceux de la première période) est qu'il interdit la communication directe entre instances. La communication se fait uniquement par passage de messages, afin de supporter aisément les architectures hétérogènes visées. Elle a ensuite été reprise (par un autre groupe de recherche) pour mettre en place un système plus complet basé sur Linux (Popcorn Linux [7]).

On trouve aussi, plus récemment, des travaux portant sur des sous-systèmes particuliers d'une architecture multikernel, tels que [26, 10, 52, 9].

Hurricane

Motivation

Hurricane [56, 58](1993,1995) est le premier SE à utiliser une structure à multiples instances du noyau. La principale motivation était d'améliorer la scalabilité et la localité des accès. La structure du système était alors appelée *Hierarchical Clustering*. Il s'agissait en fait de découper le noyau en plusieurs instances (appelés Clusters), mais aussi de regrouper de façon hiérarchique plusieurs instances entre elles afin de former des îlots d'instances (appelés SuperClusters) partageant certaines informations (telles que le taux d'utilisation des cœurs) pour de meilleures prises de décision. Cette approche est similaire à ce que fait la DQDT du noyau ALMOS, voir section 2.2.1), même si les *Superclusters* n'ont pas été implémentés dans Hurricane.

Hurricane est basé sur un micro-noyau. La figure 3.1 présente une instance du SE avec ses principales composantes. La figure 3.2 présente plusieurs instances sur une même plateforme.

Les premiers systèmes d'exploitation pour architectures CC-NUMA, tels que [6, 19], n'étaient pas réellement scalables. Ils étaient seulement scalables jusqu'à ce que le prochain goulot d'étranglement soit découvert par un autre type d'application. On peut noter que le portage de Linux sur les architectures CC-NUMA actuelles rencontre le même type de difficultés liées à cette politique d'améliorations graduelles [40, 18, 11, 35, 15].

Le choix d'une structure à multiples instances pour assurer la scalabilité du système a été conduit par trois principes fondamentaux (élaborés en utilisant la théorie des files d'attente) : préserver le parallélisme de l'application, borner le coût des services de façon à ce qu'il soit indépendant du nombre de cœur, et préserver la localité de l'application.

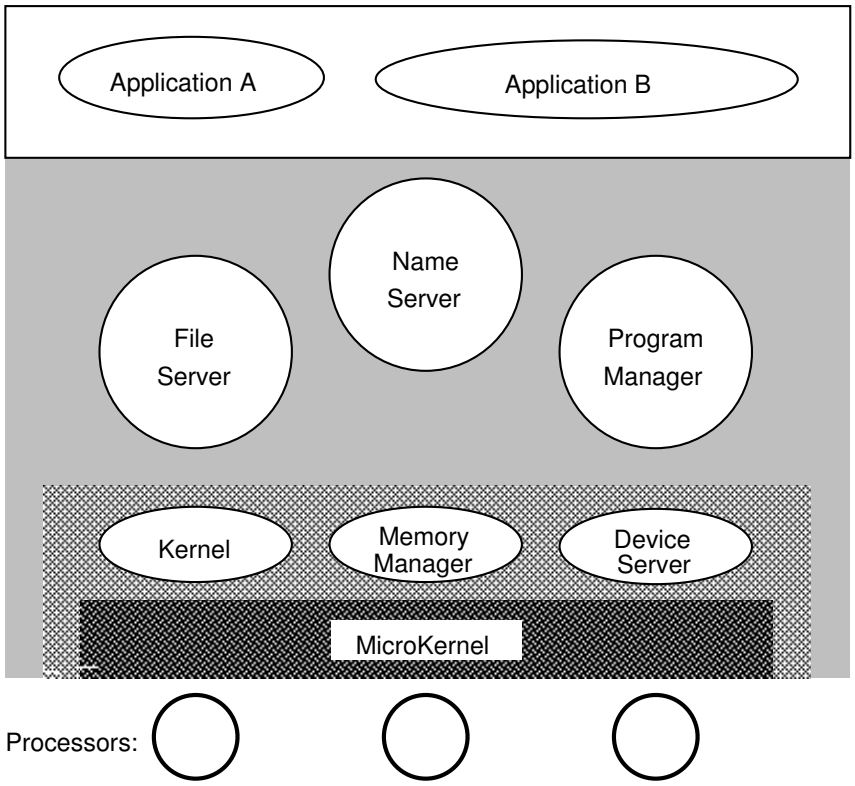


FIGURE 3.1 : Une instance (cluster) du SE Hurricane. Source : [58]

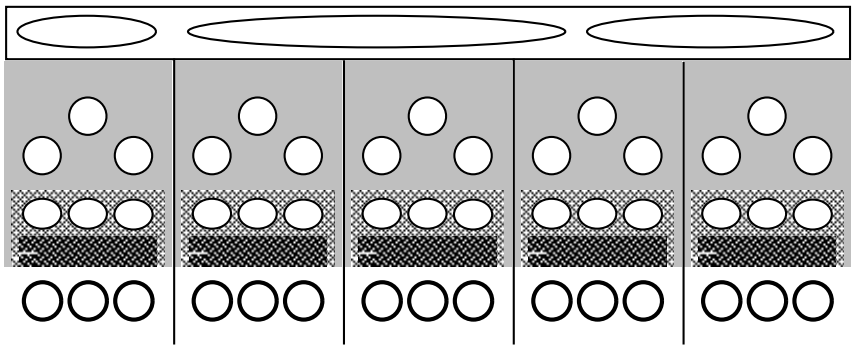


FIGURE 3.2 : Plusieurs instances (clusters) du SE Hurricane.Source : [58]

Architecture matérielle cible

Le SE Hurricane a été défini pour supporter l'architecture Hector [61]. Cette architecture a été conçue 20 ans avant l'architecture TSAR, notre cible, et elle présente donc des différences importantes par rapport à TSAR :

1. Dans l'architecture Hector, contrairement à TSAR, la cohérence des caches et des TLBs n'était pas garantie par le matériel.
2. L'espace d'adressage physique de l'architecture Hector était de 32-bit, alors que celui de TSAR est de 40 bits.
3. Hector utilise un bus en anneau limitant fortement la bande passante entre les clusters, alors que TSAR utilise un NOC scalable.

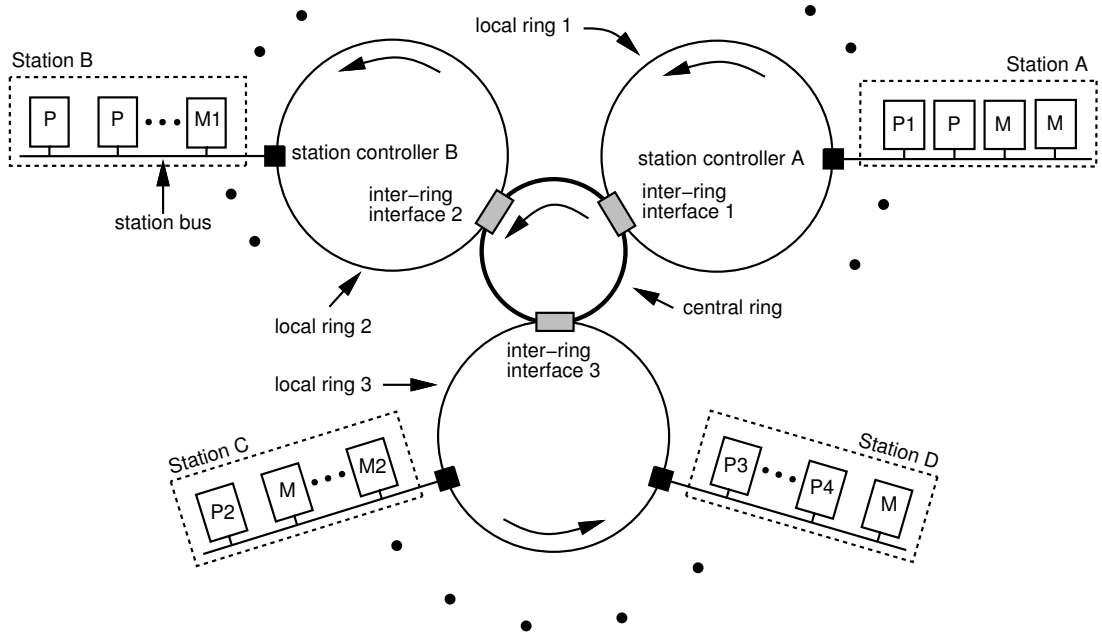


FIGURE 3.3 : Hector : l'architecture de base utilisée pour le développement du SE Hurricane. Source : [58]

Communication entre instances

Dans Hurricane, la principale technique de communication entre clusters repose sur le paradigme client/serveur, puisque Hurricane utilise l'appel de fonction distante (RPC ou Remote Procedure Call). Un appel de fonction distante effectue les étapes suivantes :

1. le cluster appelé (distant) est interrompu par une IPI (Inter Processeur Interrupt). Plus précisément, c'est le cœur distant qui a le même identifiant local que l'appelant qui est interrompu, afin d'équilibrer la charge entre les cœurs d'un même cluster ;
2. la routine de traitement de l'interruption alloue un descripteur (conteneur) dans le cluster appelé, puis copie les données du message depuis le cluster appelant, en utilisant des accès directs en mémoire partagée ;
3. le message est enregistré dans une file d'attente, afin d'être traité ultérieurement, en dehors de la fonction de traitement l'interruption.

Par ailleurs, Hurricane utilise aussi les accès directs à la mémoire distante pour accéder à un petit nombre de structures, telles que les descripteurs des pages physiques. Les auteurs indiquent que ce type de communication doit rester minimal, au risque de voir réapparaître les problèmes de passage à l'échelle caractéristiques des systèmes mono-instance.

Le problème de l'espace adressable physique plus grand que l'espace virtuel du processeur ne se posait pas pour Hurricane, puisqu'il était limité à 32 bits.

Hive

Motivation

Le système d'exploitation Hive [20] vise principalement à confiner les fautes matérielles ou logicielles au cluster fautif du SE. Pour atteindre cet objectif, Hive s'appuie, en plus de techniques logicielles, sur des mécanismes matériels implémentés sur le multiprocesseur CC-NUMA 64-bit FLASH [34]. Hive est basé sur le SE IRIX 5.2 et utilise une instance (ou *cell*) par cluster. Chaque cluster contient un seul cœur.

Communication entre instances

Dans Hive, comme dans Hurricane, les deux principaux mécanismes implémentés pour la communication entre instances sont l'accès direct et les RPC.

Le premier mécanisme utilise les instructions de lecture de mot du processeur afin de lire les données distantes. Toutefois, un certain nombre de protections ont été implémentées pour supporter le confinement des erreurs. C'est d'ailleurs pour cette raison que l'accès direct est exceptionnel.

Le mécanisme des RPC est implémenté en utilisant des composants matériels spécifiques, qui supportent également le protocole de cohérence des caches et le mécanisme des interruptions interprocesseurs. Lorsqu'une RPC est reçue dans une *FIFO* matérielle, celle-ci peut être traitée dans deux différents contextes. Elle peut soit être traitée directement dans le contexte de l'interruption soit dans le contexte d'un thread serveur. Le premier type de traitement est plus rapide (1920 cycles en moyenne), mais interdit d'effectuer certaines opérations, telles que les opérations d'entrées/sorties. Le second type est moins performant avec 6820 cycles au minimum.

Système de fichiers

L'implémentation de Hive, telle que présentée dans l'article, n'est pas complètement aboutie. Le partitionnement du système de fichier entre les instances est partiel. En particulier, l'espace des noms des fichiers n'est pas partitionné entre les instances, il est partagé. Les descripteurs de pages sont partitionnés entre les clusters. Pour accéder à une page distante, une instance créée dynamiquement et localement un descripteur de page. Celui-ci permet de référencer la page distante. Aussi, pour utiliser un fichier distant, des *inodes* (appelé *vnode*) locaux sont créés pour référencer les *inodes* distants.

Extensibilité des caches

L'architecture utilisée étant 64-bits, les auteurs n'ont pas mentionné ce problème.

Évaluations et résultats

Les évaluations ont été effectuées sur un système à quatre clusters ayant chacun un processeur. Le système utilisé et composé de quatre clusters à un cœur. Les résultats ont été comparés au SE IRIX 5.2 à une seule instance.

Deux types de benchmarks ont été utilisés. Le premier type inclut deux applications scientifiques : *Raytrace* et *Ocean* (de la suite Splash-2 [63]). Les résultats obtenus montrent que la structure multi-instances diminue les performances de 1 %, au plus.

Le second type inclut l'application *pmake* (*parallel make*) pour mesurer le temps de compilation en parallèle de 11 fichiers. Ce benchmark effectue plus d'opérations sur le

système de fichiers. Les résultats obtenus montrent une diminution des performances de 11 %, au plus.

Toutefois, ces résultats ne sont pas significatifs vis-à-vis du problème de la scalabilité du système, car ce problème ne se pose que lorsque le nombre de cœurs est important. Ces résultats montrent, au mieux, le surcoût induit par l'utilisation d'une *instance par cœur*.

Barrelfish

Motivation

L'approche multikernel [8] a pour but d'assurer la portabilité (en plus de la scalabilité) du système sur différentes architectures préexistantes. Les auteurs pensent que les architectures futures seront hétérogènes et potentiellement sans espace d'adressage partagé.

Pour justifier leur approche, les auteurs présentent la figure 3.4 qui montre la latence de modification d'une structure de données partagée formée de 1 à 8 lignes de cache. Le nombre de cœurs varie de 1 à 16. La latence de la technique des RPC synchrones (les courbes *MSG1-8*) croit de façon linéaire avec le nombre de copies, et reste (pratiquement) la même, quel que soit le nombre de lignes. La durée du traitement d'un message par le serveur (la courbe *server*) est constante et diminue même avec le nombre de messages (grâce à l'effet des caches matériels). Quant à la latence de la technique d'accès direct en mémoire partagée (les courbes *SHM1-8*), elle augmente avec le nombre de cœurs, mais aussi avec le nombre de lignes modifiées. La technique des RPC l'emporte sur la technique d'accès direct à partir de 4 lignes de cache.

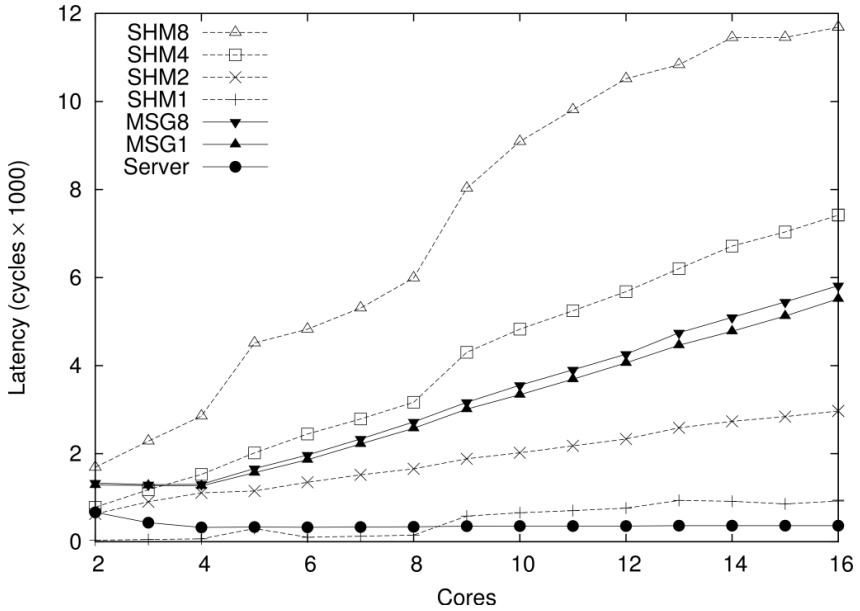


FIGURE 3.4 : Coût de modification d'une structure formée de 1 à 8 lignes de cache en utilisant la mémoire partagée (courbes *SHM1-8*) et en utilisant les RPC (courbes *MSG1-8*). Dans ce dernier cas, les performances du serveur traitant les messages sont représentées par la courbe *server*. Source : [8]

Structure générale

Pour assurer la portabilité du système, l'approche multikernel divise le système en plusieurs instances (une instance par cœur) qui communiquent entre elles uniquement par passage de messages. De plus, l'ensemble des structures du système est indépendant du matériel, excepté pour l'infrastructure de passage de message et l'interface avec le matériel (MMU, périphériques, etc.). Enfin, l'état du système doit être répliqué dans chaque instance, et chaque instance contient les modules suivants :

1. *CPU driver* : seul code qui s'exécute en espace noyau. Ce code est minimal et gère uniquement les opérations locales à un cœur : gestion de la MMU, communication entre processus, ordonnancement des processus, etc. ;
2. *monitor* : processus utilisateur qui assure la coordination entre les différentes instances du SE.
3. *serveurs* : processus utilisateur tels que serveur réseau, serveur d'allocation mémoire, etc.

Notons que l'utilisation d'une instance par cœur facilite l'implémentation, car aucun mécanisme de synchronisation n'est requis au sein d'une instance.

Communication entre instances

La communication entre instances se fait uniquement par passage de messages et directement entre processus utilisateurs. Pour communiquer avec un serveur, un client doit d'abord établir une connexion (*bind* au travers du *monitor*), laquelle permet d'allouer dynamiquement en mémoire partagée une FIFO (logicielle) à un seul écrivain et un seul lecteur. Le serveur utilise l'attente active (scrutation) pour savoir qu'un message est arrivé. Cette attente active a une durée limitée. Une fois la durée expirée, le serveur s'endort et lègue la scrutation de la FIFO au *monitor*, qui se charge de notifier si un message arrive.

Les évaluations ont été effectuées sur des architectures X86 à 64-bits avec différents nombres de processeurs. Le coût de l'envoi d'un message (sans réponse) est compris entre 180 cycles si les cœurs du serveur et du client se partagent le cache L2, et 618 cycles sinon.

Toutefois, ces mesures ne prennent pas en compte le coût d'un éventuel changement de contexte. Ce point est important lorsque plusieurs threads s'exécutent sur le même cœur, ce qui est commun en raison des nombreux serveurs système se trouvant en espace utilisateur.

Surtout, ce mécanisme de communication point à point n'est pas réellement scalable : lorsque le nombre d'instances croît, le nombre de FIFOs croît de façon quadratique, ce qui implique un coût important à la fois en encombrement mémoire et en temps nécessaire pour scruter l'ensemble des FIFOs.

Système de fichiers

Barrelfish ne dispose pas de système de fichiers partagé entre les instances. Il utilise, pour l'instant, un serveur de fichier sur réseau : NFS [53]. Un tel système de fichiers présente l'inconvénient de centraliser les accès, et de constituer un sévère goulot d'étranglement.

Finalement, le problème de l'espace adressable physique plus grand que l'espace virtuel du processeur ne se pose pas pour BarrelFish, puisqu'il ne supporte que des processeurs 64 bits.

Popcorn Linux

Motivation

Le système Popcorn Linux [7] est un autre système multikernel, qui vise les mêmes architectures hétérogènes que BarrelFish, mais les instances du noyau sont basées sur Linux.

Communication entre instances

La communication entre instances se fait uniquement par passage de message. Le canal de communication est une FIFO en mémoire partagée située du côté du serveur. Cette FIFO possède plusieurs écrivains (qui sont les clients) et un seul lecteur (qui est le serveur chargé de traiter les requêtes). La synchronisation entre les écrivains se fait sans prise de verrou (lock-free) en utilisant un mécanisme à *ticket* (voir figure 3.5). L'état de la FIFO est défini par deux variables d'état partagées : *head* et *tail*, qui ne peuvent être qu'incrémentées. La FIFO est pleine lorsque la différence entre *head* et *tail* est supérieure à la taille de la FIFO. Elle est vide s'ils sont égaux.

L'écrivain commence par réserver un ticket en incrémentant atomiquement (opération *fetch_and_add*) la variable *head*. Il scrute la variable *tail* jusqu'à ce que la case à écrire (case numéro : *head*%taille de la FIFO) soit vide. Une fois la donnée écrite, il active un drapeau spécifique à la case qui indique au lecteur que la donnée contenue dans la case est disponible. Le lecteur commence par vérifier que la FIFO n'est pas vide, puis attend que le drapeau indiquant la fin de l'écriture du message à lire soit activé. Une fois que le message est lu, le drapeau est désactivé et la variable *tail* est incrémentée.

L'arrivée d'un message est notifiée au serveur par une IPI envoyée par le client. Afin de limiter le coût des IPIs, celles-ci sont désactivées au premier message reçu, puis réactivées lorsque tous les messages dans la FIFO ont été traités.

Cette implémentation du mécanisme de passage de messages est intéressante. En effet, une telle implémentation permet à la fois d'assurer l'équité du traitement des messages (premier arrivé, premier servi) tout en minimisant les contentions en utilisant un mécanisme de ticket.

Système de fichiers

Popcorn utilise, comme *Barrelfish*, le système de fichiers sur réseau NFS. Toutefois, Popcorn Linux implémente un mécanisme permettant de partager les descripteurs de fichiers. Pour cela le tableau des descripteurs et les structures files sont répliqués lorsqu'une tâche

```
initialize:
    head = tail  = 0;

send:
    ticket = fetch_and_add(head, 1);
    spin until tail reaches (ticket - (RB_SIZE - 1));
    insert item into slot at index (ticket % RB_SIZE);
    set slot.ready;

receive:
    if (head == tail):
        return; // buffer is empty
    spin on tail until slot.ready is set;
    memcpy message to local buffer;
    clear slot.ready;
    tail++;
```

FIGURE 3.5 : Algorithme de synchronisation de l'accès à une FIFO à plusieurs lecteurs et un écrivain. Source : [55]

est migrée sur une autre instance. La réplication se fait à la demande (possible, car il s'agit de threads et non pas de processus), lorsque le processus migré essaye d'y accéder. Pour assurer la cohérence des réplicas, un drapeau à deux modes (exclusif ou partagé) a été ajouté dans les descripteurs de fichiers. Dans le premier mode, aucune cohérence n'est nécessaire. Dans le second mode, la cohérence est assurée en redirigeant toutes les opérations modifiant l'offset vers l'instance maître (instance ayant ouvert le fichier en premier). Le défaut d'une telle solution est qu'il est nécessaire d'envoyer deux messages à chaque opération de lecture ou d'écriture (`read` ou `write`) : une pour modifier l'offset et une seconde pour accéder aux données.

Hare

Motivation

Hare [25, 26] se présente comme un système de fichiers visant explicitement des architectures multiprocesseurs à mémoire partagée, sans cohérence assurée par le matériel. Les promoteurs de ce projet font le pari qu'il ne sera pas possible de maintenir la cohérence par matériel au-delà que quelques dizaines de cœurs, et que les processeurs manycores à espace d'adressage partagé seront donc sans cohérence matérielle. Sur ce type d'architecture, la cohérence des structures de données partagées du système de fichiers doit donc être assurée par logiciel, et le projet *Hare* propose et évalue différentes techniques pour atteindre cet objectif. La machine utilisée pour les expérimentations est un serveur de calcul multi-core comportant 4 processeurs Intel Xeon contenant chacun 10 cœurs (qui est - paradoxalement - une machine à mémoire partagée cohérente). Dans la suite, on appelle cluster l'ensemble des cœurs d'un même processeur multi-core, qui partagent donc le dernier niveau de cache.

Structure générale

Hare est implémenté en tant que bibliothèque logicielle au-dessus d'une seule instance de Linux. Le système fournit, en plus d'un système de fichier cohérent, un service de migration de processus (cette migration se fait au moment de l'appel système `exec`). Le système de fichier est distribué sur plusieurs serveurs, et il existe un serveur par cœur. Les serveurs sont implémentés comme des processus du noyau Linux sous-jacents. Les appels système des processus utilisateurs concernant le système de fichiers sont interceptés, et ont une implémentation spécifique dans la bibliothèque logicielle Hare. Les autres appels système qui ne nécessitent pas d'accéder à des structures de données partagées sont traités de façon standard par Linux. En tant que démonstrateur de concept, le système de fichiers est implémenté directement en mémoire (sans support pour l'accès au disque), et sans la couche de virtualisation (VFS). Ce projet respecte, à quelques exceptions près, le standard POSIX.

Communication client/serveur

Les communications entre un client (un processus utilisateur) et un serveur (un processus Hare) se font, comme dans *Barrelfish*, en espace utilisateur. Le mécanisme est basé sur l'utilisation de FIFOs possédant N écrivains et 1 lecteur. Le serveur utilise l'attente active pour détecter l'arrivée d'un message. Sur la machine utilisée pour les expérimentations, le coût d'un RPC varie entre 706 cycles, lorsque les cœurs communicants sont dans le même cluster, et 2058 cycles sinon.

Système de fichiers

Nous détaillons ci-dessous les principales techniques utilisées par Hare pour implémenter le système de fichiers, puisque la problématique de ce projet est voisine de la nôtre.

répartition de la charge entre les serveurs

Chaque serveur est responsable d'un certain nombre de fichiers ou répertoires, et il est le seul à pouvoir accéder aux métadonnées (*inodes*) associées à ce fichier/répertoire. L'allocation d'un fichier ou d'un répertoire à un serveur est réalisée au moment de la création par un processus utilisateur, et dépend à la fois de la localisation du processus utilisateur et de la localisation du serveur responsable du répertoire père du nouveau fichier. Si le processus utilisateur et le processus serveur sont dans le même cluster, le nouveau fichier est alloué au même serveur que celui responsable du répertoire père. Sinon, il est alloué au serveur situé sur le même cœur que le processus utilisateur.

Cette politique à l'avantage de rapprocher les fichiers appartenant au même répertoire, mais elle a l'inconvénient de ne pas garantir une répartition équilibrée de la charge sur les différents clusters.

Accès aux données des fichiers

L'accès aux données des fichiers distants (c.-à-d. gérés par un serveur s'exécutant sur un autre cœur que le processus client) est effectué en cachant localement les

adresses des pages de données du fichier pour y accéder directement (sans RPC), lors des appels système `read` et `write`.

Cette optimisation permet d'éviter une RPC, mais elle entraîne deux violations de la norme POSIX : en accédant directement au cache des pages, il est impossible de mettre à jour la date de la dernière lecture ou écriture au niveau de l'inode. D'autre part, les écritures ne sont pas atomiques, puisqu'un lecteur peut observer la moitié d'une écriture et que deux écritures peuvent se mélanger.

Cohérence des caches L1

En l'absence de cohérence matérielle, Hare traite en logiciel le problème de cohérence entre les caches L1 des cœurs et la mémoire. Pour traiter l'obsolescence des caches L1, Hare invalide préalablement les données présentes dans le cache L1 lors de l'ouverture du fichier (appel système `open`). Pour traiter le problème de l'obsolescence de la mémoire, Hare force l'écriture en mémoire des données écrites dans le cache L1 lors des appels système `fsync`.

Ce type de solution est utilisé dans les systèmes de fichiers sur réseau : *close-to-open consistency* [47], mais n'est pas compatible avec la norme POSIX.

Résolution du chemin d'accès

Pour identifier à quel serveur est alloué un fichier/répertoire identifié par son chemi-nom, Hare stocke dans chaque répertoire, en plus du nom de fichier/répertoire et du numéro d'*inode* (local au serveur), le numéro du serveur dans lequel l'*inode* se trouve. Ainsi, pour résoudre le chemin d'accès à un fichier, il est nécessaire d'envoyer autant de RPC qu'il y a de composantes (nom de répertoire) dans le chemin d'accès.

Cette technique pose un problème de synchronisation en cas d'accès concurrent entre un processus P qui accède en lecture à un fichier X identifié par son numéro d'*inode*, et un autre processus P' qui détruit le fichier X et crée un autre fichier Y identifié par le même numéro d'*inode*. Si la destruction/création par P' est effectuée entre la lecture du numéro de serveur contenant l'*inode* et l'accès au fichier par P, le processus P accédera au mauvais fichier. Ce problème existe, car seul le numéro d'*inode* est utilisé pour identifier un *inode* dans un serveur donné.

Cache des noms résolus

Enfin, Hare permet aussi de cacher localement les noms de fichiers/répertoires précédemment résolus. La cohérence est alors assurée par le serveur qui détruit ou modifie un fichier. Le serveur doit envoyer un message asynchrone (afin de minimiser le coût de la cohérence) à tous les clients. Ces messages sont stockés dans une file d'attente, qui est consultée par le client avant chaque opération de recherche par nom.

Gestion des structures files

Les structures files peuvent être partagées entre deux processus ayant une relation de paternité. Si la structure file est partagée, il peut donc y avoir des accès concurrents, qui doivent être arbitrés par le serveur responsable du fichier, ce qui impose l'utilisation d'une RPC pour chaque appel système `read` ou `write`. Lorsque la structure file

n'est pas partagée, le processus client peut directement accéder au cache de pages sans passer par le serveur, ce qui viole la norme POSIX comme mentionné plus haut.

Distribution des entrées d'un répertoire

Afin d'éviter la contention lorsque plusieurs fichiers sont accédés en parallèle dans un même répertoire, Hare permet à l'utilisateur de distribuer ce répertoire (en utilisant une fonction de hachage) sur plusieurs serveurs. Cette distribution est activée par un drapeau qui est mis au moment de la création du répertoire. Cette optimisation complexifie fortement les opérations qui lisent l'ensemble du répertoire, tel que l'appel système *readdir*.

3.2 Stratégies de placement dans les architectures NUMA

Les stratégies de placement les plus étudiées portent sur le placement des données utilisateur, tels que [12, 60, 21]. Toutefois, nous nous limiterons, dans cette section, à l'analyse des stratégies de placement des structures de données du système de fichiers.

3.2.1 Systèmes industriels

Le SE Linux différencie, pour le placement, deux types de structures. D'un côté, il y a les structures dont l'allocation est faite au moment de l'initialisation du SE. Elles sont distribuées sur les différents bancs mémoire, il s'agit de la table de hachage des caches de pages, des *freelist* et des *dirtylist*.De l'autre côté, il y a les structures qui sont allouées dynamiquement, pour lesquelles la stratégie de placement par défaut est locale (aussi appelé *first touch*), c'est-à-dire que la mémoire est allouée dans le banc mémoire le plus proche du processeur qui exécute la tâche demandeuse.

Cette stratégie de placement pose deux problèmes. Le premier problème est qu'elle peut générer de la contention dans le cas – fréquent – où un fichier est ouvert par un thread afin d'être utilisé par les autres threads d'une application parallèle. Le second problème est qu'une telle stratégie peut consommer l'ensemble de la mémoire d'un cluster [17], si la taille du fichier est du même ordre de grandeur que la capacité mémoire disponible dans un seul cluster.

Toutefois, Linux met à disposition des développeurs ou de l'administrateur plusieurs moyens pour mettre en place, soit des stratégies de placement ad hoc (en choisissant précisément un banc mémoire), soit la stratégie de placement *interleave*, qui est une distribution systématique sur tous les bancs de mémoire. Cette dernière stratégie est celle qui est d'ailleurs recommandée par les constructeurs de machine NUMA pour équilibrer la charge mémoire sur les différents clusters [27].

Par ailleurs, il existe plusieurs propositions pour répliquer le cache des pages de Linux, telles que [11, 64], et une proposition pour répliquer le cache des *dentrys* [38]. Ce type de stratégie combinée avec un cache central distribué (stratégie *interleave*) peut permettre de bénéficier de la localité, tout en évitant la contention des accès sur le cache central. Toutefois, à l'heure actuelle, aucune de ces propositions n'a été intégrée dans Linux.

Ici encore, Linux est le plus avancé, puisque les autres systèmes commerciaux proposent, au mieux, des stratégies permettant de placer localement les données dynamiquement allouées.

3.2.2 Solutions académiques

Afin de favoriser la localité dans le cas de données partagées entres plusieurs threads, il existe a priori deux stratégies : (1) la stratégie de réplication et (2) la stratégie de migration. Cependant, nous n'avons trouvé aucun système qui utilise la migration.

La stratégie de réplication est utilisée par K42. Elle consiste à répliquer localement les structures placées à distance. Les structures ont alors deux instances : (1) une instance qui représente la structure partagée globale, appelée *Master* et (2) une instance qui représente la portion localement répliquée, appelée *Local*. Cette instance peut être vue comme un cache logiciel, où la réplication est faite à la demande. On retrouve l'utilisation de cette stratégie sur la table de hachage du cache des données appelé *DHashTable*. L'intérêt de cette stratégie est qu'elle permet non seulement de favoriser la localité des accès, mais elle allège aussi la contention du mécanisme de synchronisation et du trafic des accès sur l'instance *Master*, puisque l'instance locale absorbe la majorité des requêtes.

Cependant, cette stratégie de réplication présente, outre le coût en espace mémoire induit par la réplication, le défaut d'utiliser une structure globale non distribuée : la structure Master peut devenir un point de contention lorsque le nombre de cœurs devient important. Les expérimentations effectuées par les auteurs sont limitées à une dizaine de cœurs, alors que nous visons plusieurs centaines de cœurs. Ce type de contention a été mentionné dans [5], où l'auteur décrit le problème de la scalabilité des accès sur une structure qui est globalement partagée (la table des pages) et localement cachée (par les caches TLB) lorsque le nombre de cœurs est supérieur à 256. Ce phénomène a également été observé sur ALMOS.

C'est pour cette raison que nous avons décidé de nous concentrer sur le problème de définition d'une stratégie de placement totalement distribuée visant principalement à éviter la contention, pour garantir le passage à l'échelle (voir la section 4.3).

3.3 Accès concurrent aux structures de données

Dans cette section, nous présentons les principaux mécanismes de synchronisation utilisés dans les systèmes d'exploitation existants. Puis, dans un second temps, nous analysons l'utilisation de ces mécanismes au niveau des structures du système de fichiers.

3.3.1 Mécanismes permettant de limiter le nombre d'accès

Les verrous

Les verrous binaires (aussi appelés *spin-lock*) sont des variables booléennes qui permettent de garantir à un thread l'exclusivité de l'accès à une ressource telle qu'une structure de données

partagée. L'implémentation classique d'un verrou consiste à utiliser une opération de lecture-puis-écriture atomique sur un mot mémoire pour la prise du verrou. De plus, la technique de verrou à ticket permet une allocation équitable et minimise le risque de famine [42].

Lorsque le verrou est déjà pris, deux stratégies d'attente existent. L'attente passive consiste à ce que le thread demandeur s'endorme pour être réveillé plus tard, lorsque le thread propriétaire de la ressource libère le verrou. L'attente active consiste à ce que le thread demandeur scrute le verrou jusqu'à ce qu'il soit libéré, puis réessaye de prendre le verrou. L'attente active est généralement utilisée lorsque le temps d'usage d'une ressource partagée est court et borné. L'attente passive est utilisée lorsque le temps d'usage à une ressource partagée est long, tel que pour l'accès au disque dur.

L'attente active permet d'avoir des temps d'attente plus courts, mais avec un risque de contention important. En effet, lorsque le nombre de threads qui essayent de prendre le verrou devient important, un goulot d'étranglement se crée au niveau du banc mémoire contenant le verrou, avec des latences qui peuvent croître de façon quadratique. Pour atténuer cette contention, plusieurs techniques existent dans la littérature, telles que les verrous à file d'attente MCS [43, 33], les verrous avec *backoff* [4], ou encore les verrous hiérarchiques répartis sur plusieurs bancs de mémoire [57].

Pour conclure, la séquentialisation stricte des accès imposée par la technique de verrou constitue un inconvénient majeur, dans le cas des structures de données du système de fichier, qui peuvent être accédés en parallèle par des centaines, voire des milliers de threads.

C'est la raison pour laquelle d'autres mécanismes sont apparus permettant, en particulier, de paralléliser les accès en lecture.

Les verrous multi-lecteurs

Les verrous multi-lecteurs permettent de paralléliser les accès en lecture tout en ne permettant qu'un seul accès en écriture. Une implémentation classique de ce mécanisme consiste à utiliser deux verrous *r* et *g*, plus un compteur de lecteurs *cr* [50]. Le verrou *g* permet d'assurer l'accès exclusif à la ressource protégée. Le verrou *r* est utilisé uniquement par les lecteurs pour protéger le compteur *cr*.

L'idée de principe est que tous les écrivains doivent prendre le verrou avant d'accéder à la ressource, mais que seul le premier lecteur doit prendre le verrou.

Les étapes permettant de prendre le verrou en lecture sont les suivantes :

- Prise du verrou *r*
- incrémentation de *cr*
- si *cr* est égal à un (ce lecteur est le premier), prise du verrou *g*
- libération du verrou *r*

Les étapes permettant de libérer le verrou en lecture sont les suivantes :

- Prise du verrou *r*

- décrémentation de *cr*
- si la valeur de *cr* égale zéro (dernier lecteur à terminer l'accès), libération du verrou *g*
- libération du verrou *r*

Les écrivains prennent et libèrent le verrou multi-lecteurs en verrouillant/déverrouillant le verrou *g* sans se soucier du compteur *cr* ni du verrou *r*.

Ce type de verrous peut aussi être implémenté avec deux types d'attente (active ou passive), et on trouve également plusieurs variantes pour atténuer la contention. Toutefois, ils présentent le défaut de séquentialiser les accès entre les lecteurs et l'écrivain. Leur avantage reste que les lecteurs peuvent accéder en parallèle aux données dans la section critique.

Les seqlocks

Le mécanisme des *seqlocks* permet aussi à plusieurs lecteurs d'accéder en parallèle à une même structure de donnée partagée [35].

L'idée de base du mécanisme des *seqlocks* consiste à utiliser un compteur initialisé à zéro, lequel est partagé entre les écrivains et les lecteurs.

Pour écrire, un écrivain doit incrémenter atomiquement le compteur afin de rendre sa valeur impaire. Cette incrémentation n'est réalisée que si la valeur du compteur est paire. Autrement, le verrou est déjà pris par un autre écrivain. Lorsque la structure protégée est modifiée, l'écrivain incrémente à nouveau le compteur, rendant sa valeur paire. Pour accéder en lecture, un lecteur doit d'abord lire la valeur du compteur, puis lire les données partagées, puis relire la valeur du compteur. Si les deux valeurs lues sont paires et égales, la lecture s'est bien déroulée. Autrement le lecteur doit réessayer.

Le grand avantage des seqlocks est donc de permettre la parallélisation des accès en lecture sans qu'il y ait de contention lorsqu'il n'y a pas d'écrivains. Le principal inconvénient est que le risque de famine est plus important, puisqu'il ne fournit pas de garantie d'allocation équitable.

Un autre problème de ce mécanisme est qu'il ne peut pas être utilisé sur des structures contenant des pointeurs, telles que les listes chaînées. Ceci est dû au fait que les lecteurs ne sont pas protégés contre les écrivains, qui peuvent à tout moment modifier la structure, ce qui peut entraîner des accès invalides (voir 2.4.3).

Pour l'accès aux structures de type listes chaînées, ce mécanisme doit être combiné avec un mécanisme permettant de garantir l'existence des structures.

3.3.2 Mécanismes permettant de garantir l'existence des structures

Ce type de mécanismes cherchent à résoudre le problème de l'utilisation après libération (voir 2.4.3).

Lorsqu'on utilise des verrous, ce problème n'existe pas, puisque les accès sont mutuellement exclusifs. Par contre, lorsqu'on utilise des mécanismes permettant plusieurs accès parallèles, ce problème devient important.

Ces mécanismes peuvent être classés en deux grandes catégories. En premier lieu, on trouve les mécanismes qui permettent de garantir l'existence des structures sans dépendre d'autres mécanismes, tels que le mécanisme RCU (*Read-Copy-Update*) [41]. En second lieu, on trouve les mécanismes qui imposent des contraintes sur la pérennité du type des structures, tels que le mécanisme CR (compteur de références) [59].

Le mécanisme RCU

Le mécanisme *Read-Copy-Update* (RCU) peut être utilisé de différentes manières. Son utilisation la plus répandue est de permettre un accès en lecture aux structures de types listes chaînées (ou arbres de recherche), sans que les écritures affectent les performances des lecteurs.

Le principe consiste à ce que l'écrivain bloque la libération d'un nœud détaché jusqu'au moment où tous les lecteurs ayant accédé à la structure avant le détachement ont fini leur accès à la structure. Pour cela, tous les lecteurs doivent signaler à l'écrivain le début et la fin de leur accès à la structure.

Une implémentation simple consiste à utiliser deux listes chaînées auxiliaires, chacune associée à un compteur. Ces listes sont utilisées par l'écrivain pour stocker temporairement les nœuds détachés, mais non encore libérés. Les compteurs sont incrémentés atomiquement par les lecteurs lorsqu'ils entrent dans la structure et décrémentés lorsqu'ils en sortent. Seuls une liste et son compteur sont utilisés à un moment donné. Lorsque le nombre de nœuds détachés (mais non libérés) dans la liste en cours atteint un certain seuil, l'écrivain verrouille la liste courante et bascule sur l'autre liste. Toutefois, les lecteurs ayant commencé à utiliser la première liste continuent à l'utiliser. Le compteur correspondant à la première liste décroît donc, et lorsqu'il atteint la valeur zéro les nœuds de la première liste peuvent être effectivement libérés. L'intérêt d'avoir deux listes est de ne pas bloquer l'écrivain.

L'implémentation est généralement plus complexe afin d'éviter la contention sur les incrémentations et décrémentations atomiques des compteurs (on utilise parfois des compteurs par cœur ou par thread).

L'inconvénient majeur de ce mécanisme est la consommation mémoire : si un lecteur est retardé (en raison d'un défaut de page par exemple), alors le nombre de nœuds dans la liste non verrouillée peut devenir important.

Les compteurs de références

Le mécanisme des compteurs de références (CR) a été proposé en 1995 pour permettre des accès sans verrou aux structures de données de type liste chaînée (ou arbres de recherche). Le problème à résoudre est d'éviter l'utilisation d'une donnée stockée à une adresse X par un lecteur, alors que la structure a été modifiée par l'écrivain et que la case mémoire X a été libérée par l'écrivain.

Le principe consiste à attacher à chaque élément de la liste chaînée (ou de l'arbre) un compteur de références. Chaque lecteur doit, avant d'accéder à la structure protégée incrémenter

atomiquement le compteur de références pour indiquer à un éventuel écrivain que cet élément de la structure est en cours d'utilisation, et doit décrémenter atomiquement ce compteur quand il termine son utilisation. L'écrivain doit vérifier, avant de libérer (retirer) un élément de la structure, que le compteur de références est à zéro.

Ce mécanisme est portable sur toute machine possédant une primitive CAS (Compare And Swap) permettant de réaliser des incrémentations et des décrémentations atomiques.

L'avantage de ce mécanisme est de permettre des accès parallèles en lecture, et en écriture. Son principal inconvénient est l'augmentation très importante des temps de parcours de la structure chaînée, puisque tout élément visité implique deux opérations atomiques de type CAS.

Enfin, ce mécanisme requiert la pérennité du type des structures stockées dans certaines zones mémoire, pour garantir que le champ CR de la structure reste toujours valide, même si les valeurs stockées dans les autres champs sont modifiées.

3.3.3 Mécanisme client/serveur

Certains systèmes d'exploitation utilisent des mécanismes de passage de message de type client/serveur pour garantir l'accès exclusif à certaines données partagées : l'idée de base est que les clients postent des requêtes vers le serveur, et que le serveur ne traite qu'une seule requête à la fois, ce qui garantit l'atomicité des accès [28, 24]. Ce mécanisme est en particulier utilisé par le gestionnaire d'événement d'ALMOS (voir la section 2.2.1).

On peut remarquer que, dans le cas où il y a plusieurs clients et une seule file d'attente du côté du serveur, il faut un mécanisme d'enregistrement des requêtes dans la file d'attente, et que cet enregistrement suppose l'existence d'un mécanisme de synchronisation de plus bas niveau entre les écrivains, qui utilise donc un des mécanismes cités plus haut.

3.3.4 Synchronisation utilisée dans les systèmes de fichiers

Dans cette partie, nous décrivons les mécanismes de synchronisation utilisés par les systèmes d'exploitation existants pour chacune des structures du système de fichier.

La table des descripteurs de fichier

Linux utilise le mécanisme CR pour d'accéder aux structures *file*. Ce mécanisme est combiné au mécanisme RCU afin d'assurer l'existence de la structure [40]. Ce mécanisme permet des accès concurrents en lecture sans prise de verrou.

Pour les écritures, Linux utilise un verrou par structure *file* afin de synchroniser les écritures (telles que la modification de l'offset du fichier).

FreeBSD utilise aussi le mécanisme des CR. Toutefois, pour assurer la permanence du type des structures *file*, le noyau interdit la libération de ces structures. Cette solution ne pose pas de problème mémoire, car d'une part, la réutilisation des structures reste possible (ce qui est

interdit est de libérer la mémoire afin qu'elle soit utilisée sous une autre forme), et que, d'autre part, le nombre total de structures *file* utilisable est borné.

Les autres systèmes d'exploitation utilisent un mécanisme de synchronisation à base de verrous pour les deux types d'accès.

Les caches de métadonnées

Pour les accès en lecture au cache des *inodes*, le SE Linux utilise le mécanisme RCU pour garantir l'existence des structures. Les accès en lecture s'effectuent donc sans verrou. Pour le cache des *dentry*s, le SE Linux utilise une solution complexe basée sur trois mécanismes : RCU, seqlock et CR [37] pour permettre les accès concurrents en lecture. L'utilisation des seqlocks est nécessaire pour détecter un scénario subtil : lorsque l'accès en lecture se fait en parallèle à un accès en écriture effectuant l'opération *rename*. Quant au mécanisme des CR, il est utilisé dans les cas où RCU ne peut être utilisé : lorsqu'un thread se met en attente passive lors d'un accès au disque. Les CR sont aussi utilisés pour éviter le risque de famine que peut induire l'utilisation des seqlocks.

Pour les accès en écriture au cache des *dentrys*, Linux utilise un mécanisme de verrous à grain fin : un verrou par entrée de la table de hachage. Quant au cache des *inodes*, c'est un verrou global qui est utilisé.

Par ailleurs, pour améliorer les performances des accès aux listes chaînées des éléments non utilisés des caches de métadonnées (les *freelist*), celles-ci sont partitionnées sur plusieurs nœuds de mémoire, chacune utilisant un verrou privé.

L'implémentation dans le système d'exploitation Linux est l'une des plus avancées, car dans les autres systèmes on trouve l'utilisation de verrous pour les deux types d'accès.

Le cache des pages

Linux utilise le mécanisme RCU combiné avec celui des CR pour les accès en lecture [49]. Le mécanisme RCU est utilisé pour traverser l'arbre de recherche, puisque chaque fichier a son propre cache implémenté par un *radix-tree*. Le mécanisme CR est utilisé pour passer d'une feuille de l'arbre de recherche aux structures page. Pour justifier l'utilisation du mécanisme CR, qui est moins scalable en performance que RCU, l'auteur du mécanisme invoque l'importante consommation en mémoire que peut engendrer le mécanisme RCU.

Pour la synchronisation des accès en écriture, on retrouve l'utilisation d'un verrou par fichier.

Le principal défaut du mécanisme d'accès en lecture est la contention induite par l'utilisation des CR lorsque les structures pages sont accédées en parallèle. Le défaut du mécanisme lors de l'accès en écriture est que la synchronisation est à gros grains. Le mécanisme verrouille tout un ficher même si l'accès porte sur une seule page. Un autre défaut de ce cache est qu'il n'est pas totalement conforme au standard POSIX. Ce dernier spécifie que les écritures doivent apparaître atomiques vis-à-vis des lectures. Dans la solution de Linux, les lecteurs accèdent sans verrou au cache du fichier, sans se synchroniser avec les écrivains. Un tel scénario fait

que les lecteurs peuvent lire des données en cours d'écriture, contredisant par cela la propriété d'atomicité.

Les autres systèmes utilisent une synchronisation à base de verrous avec différents types de structures (table de hachage ou arbre de recherche). Ce type de synchronisation est moins performant que le mécanisme RCU particulièrement en raison des opérations atomiques nécessaires pour prendre le(s) verrou(s).

3.4 Conclusion

Les principales études effectuées sur la scalabilité des systèmes de fichiers se trouvent dans les systèmes d'exploitation commerciaux, en particulier dans le système Linux. Dans ce dernier, on trouve l'utilisation de mécanismes de synchronisation avancés, tels que l'utilisation du mécanisme RCU. Toutefois, la stratégie de placement utilisée par défaut dans ces systèmes n'est pas généraliste. Cette stratégie consiste, par défaut, à effectuer un placement local au demandeur des données du système de fichiers. Cette stratégie peut s'avérer pénalisante, par exemple dans le cas où un fichier est chargé par un premier thread, mais est accédé de façon concurrente par les autres threads. Ces accès sont alors en contention sur le banc mémoire contenant le fichier chargé. Par ailleurs, les solutions permettant d'exploiter un espace physique plus large que l'espace virtuel sont limitées à quelques Giga-octets, alors que notre objectif est de supporter des espaces physiques pouvant aller au Tera-octet.

Les systèmes multi-instances sont très prometteurs pour les architectures manycores, puisque l'approche multi-instances garantit la scalabilité "par construction". Cependant, très peu de travaux ont été effectués sur le système de fichiers, encore moins sur la couche VFS. Les systèmes multi-instances existants utilisent généralement des systèmes de fichiers distribués tels que NFS, qui sont généralement peu efficaces, ou non compatibles avec le standard POSIX. Le système de fichiers le plus prometteur est celui de *Hare*. Ce dernier a été publié pendant notre travail et certaines des stratégies proposées sont semblables aux nôtres. Toutefois, ce système de fichiers ne supporte pas la couche VFS et ne permet pas l'accès au disque. De plus, le projet *Hare* vise des architectures sans cohérence et certaines techniques ne sont pas complètement compatibles avec le standard POSIX.

Cette étude montre qu'il n'existe pas actuellement de systèmes résolvant les différents problèmes considérés dans cette thèse. Cependant, nous utilisons différentes techniques présentées dans ce chapitre pour définir un système répondant à nos objectifs.

CHAPITRE 4

Solutions Proposées

Dans ce chapitre, nous présentons les solutions qui ont été explorées et évaluées pour résoudre les problèmes identifiés dans le chapitre de problématique. Nous commençons par le problème d'adressage de l'espace physique 40 bits. Puis nous présentons et justifions les stratégies que nous proposons pour le placement des structures de données partagées du système de fichiers. Nous décrivons enfin en détail les mécanismes de synchronisation retenus pour gérer la concurrence d'accès aux structures.

Remarque : Certains des mécanismes décrits dans ce chapitre n'ont pas été implémentés, par manque de temps. Ces mécanismes sont indiqués par un texte encadré, semblable à celui-ci.

4.1 Adressage 40 bits

La version actuelle d'ALMOS ne supporte que des adresses physiques sur 32 bits. Il faut donc commencer par introduire dans ALMOS un mécanisme efficace permettant de supporter les adresses physiques 40 bits de l'architecture TSAR, tout en se réservant la possibilité de supporter des adresses physiques plus grandes encore.

4.1.1 Approche Mono-instance

Principe

Pour permettre la gestion des adresses physiques 40 bits par ALMOS, nous avons essayé une première solution, inspirée de celle présente dans les noyaux monolithiques utilisant des processeurs 32 bits. Celle-ci a été implémentée par François Guerret durant son stage d'ingénieur, et elle a abouti aux modifications suivantes dans ALMOS :

1. **Modification de l'encodage des adresses physiques.**
   La représentation interne des adresses physiques est naturellement passée de 32 bits à 64 bits.

2. **Modification de l'espace virtuel du noyau.**
   L'espace virtuel a été réduit à 1 Go, afin d'offrir 3 Go d'espace virtuel à chaque processus utilisateur. Puis, l'espace virtuel de 1 Go réservé au noyau a été explicitement partitionné entre les bancs de mémoire des clusters.
3. **Accès à la mémoire physique non projetée dans l'espace virtuel.**
   Pour accéder aux adresses physiques qui ne sont pas projetées dans l'espace virtuel de 1 Go, ALMOS a utilisé directement l'adressage physique, en désactivant la MMU, et en exploitant les registres d'extension d'adresse, décrits dans le chapitre 2.

Cette méthode a été effectivement utilisée pour les tables de pages construites par le noyau pour les différentes applications. Ces tables de pages ne sont donc plus accessibles dans l'espace virtuel du noyau, mais uniquement en adressage physique. Il était initialement prévu de poursuivre ce travail en stockant directement dans l'espace physique les autres structures volumineuses du noyau telles que les descripteurs de pages physiques ou les structures du système de fichiers. Mais les difficultés de cette approche mono-instance sont apparues, remettant en cause l'approche elle-même.

Difficultés rencontrées

Cette approche mixte, où une partie des données du noyau peut être accédée en adressage virtuel et une autre partie doit être accédée en adressage physique s'est révélée très complexe à implémenter et difficilement généralisable.

1. L'utilisation de deux méthodes d'adressage différentes, et donc de deux types de pointeurs (virtuels et physiques) est une source de difficultés et de complexité considérable pour la programmation du noyau.
2. En supposant une architecture à 256 clusters, l'espace virtuel noyau disponible dans chaque cluster est de 4 Mo (1 Go / 256). Cet espace est juste suffisant pour projeter les objets de base du noyau, qu'ALMOS réplique dans chaque cluster, à savoir le segment de code et le segment des données en lecture seule. Cela signifie que pratiquement toutes les structures de données du noyau allouées dynamiquement doivent être placées en espace physique, et accédées en adressage physique, ce qui implique une re-écriture complète du code du noyau.

Face à ces difficultés, nous nous sommes tournés vers une solution plus systématique — et plus conforme à la philosophie d'ALMOS — consistant à renoncer complètement à l'utilisation de l'adressage virtuel 32 bits pour le noyau, et à faire évoluer ALMOS vers une organisation multi-instances, inspirée des solutions présentées dans le chapitre 3.

4.1.2 Approche Multi-instances

Principe

Dans une architecture de noyau multi-instances, l'écrasante majorité des accès mémoire, réalisés par une instance du noyau, sont locaux. Les accès mémoire se font dans le

banc de mémoire physique appartenant au cluster contenant l'instance du noyau. Dans TSAR, l'espace d'adressage physique est distribué sur tous les clusters de la manière suivante : les 8 bits de poids forts désignent le numéro du cluster et les 32 bits de poids faible désignent l'adresse dans le cluster. Ainsi en désactivant la MMU et en utilisant les registres d'extension d'adresses, présents dans le cache de premier niveau, pour qu'ils désignent le numéro du cluster local, chaque cœur d'un cluster accède directement aux 4 Go d'espace physique du cluster.

Les accès aux bancs de mémoire d'un autre cluster sont nettement plus rares, puisque les communications entre instances du noyau utilisent des appels de fonctions distantes (RPC), sur un modèle client/serveur. Avec cette technique, il n'y a en principe que le mécanisme RPC lui-même qui nécessite de modifier temporairement les registres d'extension d'adresse des cœurs.

Analyse

Cette solution consistant à ne pas utiliser la MMU pour le code du noyau présente l'avantage de minimiser les modifications à apporter au code d'ALMOS existant, et apporte deux avantages supplémentaires : (1) le premier est que chaque application utilisateur peut disposer de 4 Go d'espace d'adressage, puisque le noyau n'utilise plus du tout l'espace virtuel ; (2) le second avantage est que l'accès à la mémoire virtuelle des applications utilisateurs est potentiellement plus performant, car le cache TLB n'est plus partagé avec le noyau.

Un inconvénient de cette solution est qu'il est plus difficile de détecter certains bugs dans le noyau. Typiquement, une écriture faite par le noyau dans le code du noyau lui-même (en phase de mise au point) n'entraîne pas de départ en exception puisque la MMU est désactivée.

Remarque 1 : la structure multi-instances n'affecte que le noyau, les applications utilisateurs ne perçoivent aucune différence par rapport à un noyau formé d'une seule instance et s'exécutant en espace virtuel projetant des pages physiques de n'importe quel banc de mémoire. L'ensemble des applications s'exécutent en mémoire virtuelle avec toutes les protections fournies par ce mécanisme : telles que l'impossibilité d'accéder aux données du noyau, l'impossibilité d'accéder à l'espace virtuel des autres applications utilisateurs, etc.

Remarque 2 : pour optimiser les performances, le noyau peut toujours accéder à certaines données distantes en adressage physique sans passer par le mécanisme RPC. Ce type d'accès est souvent utilisé dans notre implémentation pour copier des blocs de données (généralement des pages complètes) préallouées entre clusters. Il est, par exemple, utilisé pour éviter une double copie des données entre le cache des fichiers et les tampons utilisateurs.

Portabilité

Il est clair que les registres d'extension d'adresse constituent un mécanisme spécifique à l'architecture TSAR, qui limite la portabilité de la solution. Mais la solution multi-instances

retenue n'est pas réellement dépendante de la disponibilité de ce mécanisme, et peut être implémentée en utilisant la mémoire virtuelle paginée disponible sur tous les processeurs modernes.

Une instance du noyau peut très bien s'exécuter dans un espace virtuel qui projette uniquement la mémoire physique locale. Et le mécanisme RPC permettant la communication avec les autres instances peut être implémenté avec un petit nombre de pages virtuelles qui projettent statiquement et durablement les tampons de communication situés dans les autres clusters. L'utilisation de la mémoire virtuelle reste donc une solution viable pour assurer l'implémentation de la solution proposée sur des processeurs 32 bits ne supportant pas l'adressage physique direct.

4.1.3 Conclusion sur l'approche multi-instances

La solution proposée au problème de l'accès à un espace d'adressage nettement plus grand que les 4 Go adressables par un processeur 32 bits consiste donc à renoncer totalement à l'adressage virtuel pour le noyau, et à faire évoluer ALMOS vers une structure multi-instances, avec — au moins — une instance du noyau dans chaque espace physique de 4 Go, c'est-à-dire dans chaque cluster.

Cette solution était déjà fortement suggérée par l'organisation distribuée d'ALMOS qui dispose d'un `cluster manager` par cluster. La principale modification a consisté à systématiser et généraliser l'usage du mécanisme de communication par RPC entre les différentes instances du noyau. Ce mécanisme était déjà proposé par les *events managers* présents dans chaque cluster. En outre, cette solution a le mérite de minimiser les modifications dans le code d'ALMOS qui utilise déjà uniquement des adresses 32 bits.

Nous avons appelé **ALMOS-MK** cette nouvelle architecture totalement distribuée, MK pour Multi-Kernel.

4.2 Communications entre instances du noyau

Le choix de passer à une structure multi-instances affecte évidemment tout le noyau. Dans cette section, nous rappelons les avantages du mécanisme client/serveur pour la communication entre instances du noyau, nous discutons le choix du nombre de cœurs par instance, et nous détaillons le mécanisme des RPC (Remote Procédure Call) que nous avons retenu, qui a pour principal objectif le passage à l'échelle, puisque nous voulons supporter jusqu'à 1024 cœurs.

4.2.1 Mécanisme client/serveur

Les communications entre instances se font par RPC (Remote Procedure Call) et utilisent le passage de message. Les avantages de cette approche sont les suivants :

1. **Simplification des synchronisations.**
   Il n'est plus nécessaire d'utiliser des mécanismes complexes, tels que les mécanismes

sans verrous, excepté pour implémenter le mécanisme des RPC lui-même. En effet, dans une instance, le nombre d'accès concurrents aux structures de données partagées est limité par le nombre de cœurs contenus dans un cluster.

2. **Meilleures performances.**
   Le coût (en latence comme en consommation énergétique) d'un accès à un banc mémoire distant est nettement plus élevé que le coût d'un accès local. Dans une architecture TSAR à 256 clusters (grille 16×16), le rapport peut facilement atteindre un facteur 10. Si le traitement à effectuer dans un cluster est complexe et nécessite donc beaucoup d'accès distants, il est moins coûteux de le sous-traiter à l'instance locale en utilisant une RPC que de le réaliser par des accès directs en mémoire distante. Ceci justifie l'utilisation par ALMOS de l'*events manager*. Cela a également été démontré par le système d'exploitation multi-instances *Barrelfish*.

3. **Meilleure scalabilité.**
   L'utilisation du mécanisme de passage de messages limite — par construction — les accès distants et pousse naturellement à distribuer les structures de données du noyau pour favoriser les accès locaux. Ceci favorise le passage à l'échelle, et permet de limiter le risque de contention sur les bancs de mémoires.

4.2.2 Nombre de cœurs par instance.

Puisque nous souhaitons que les accès mémoire internes à une instance du noyau utilisent des adresses 32 bits, il faut qu'une instance soit limitée à un seul cluster. Ainsi, le nombre de cœurs ne peut dépasser celui d'un cluster, c'est-à-dire pour TSAR, quatre cœurs. En revanche, il est possible d'avoir une instance du noyau par cœur, ce qui signifie donc plusieurs instances du noyau par cluster.

Cette option, à un seul cœur par instance du noyau, a pour avantage de fortement simplifier les mécanismes de synchronisation internes dans un noyau. Avec un seul cœur par instance, il n'y a plus de vrai parallélisme, mais seulement du parallélisme émulé par multiplexage temporel. Si les interruptions sont masquées dans les sections critiques, il n'y a plus de risque d'accès concurrents, et l'on peut se passer de verrous.

En revanche, l'inconvénient de cette option est qu'elle augmente le nombre d'instances du noyau, ce qui augmente l'empreinte mémoire, puisque chaque instance réplique l'état du système. Cette option n'a pas non plus d'intérêt pour le passage à l'échelle du SE puisqu'elle ne réduit pas la contention d'accès à la mémoire dans la mesure où toutes les instances dans un cluster partage le même banc.

Le choix est cependant difficile et aurait nécessité des expérimentations poussées, car il dépend de nombreux facteurs : la quantité de mémoire disponible dans la plateforme, le nombre de cœurs par cluster, et le type des applications visées. Pour limiter le périmètre de l'étude, nous avons choisi la solution d'une instance par cluster, qui a le mérite de minimiser les modifications à apporter dans le noyau d'ALMOS existant et qui nous semble le meilleur compromis.

4.2.3 Canaux de communication entre instances

Type des messages

Les messages permettant d'implémenter le mécanisme RPC ont une taille variable, car le message de commande est constitué d'un identifiant de fonction et d'un nombre variable d'arguments. Par ailleurs, dans la majorité des cas, une RPC requiert une réponse, qui peut elle-même contenir un nombre variable d'arguments.
Une RPC peut être synchrone ou asynchrone. Lorsqu'elle est synchrone le processus émettant la RPC se met en attente jusqu'à la réception d'une réponse. Sinon elle est asynchrone, le processus émettant la RPC peut continuer son exécution sans attendre la réponse.

Partages des canaux

Dans chaque instance du noyau d'ALMOS, tous les services système partagent le même espace d'adressage. Ces différents services peuvent donc sans difficulté partager les mêmes canaux de communications entre instances. Ces canaux sont statiques (ils sont créés au démarrage du noyau) pour éviter le coût de construction des canaux à chaque envoi de message. Chaque cœur peut envoyer plusieurs messages, parce que plusieurs threads peuvent s'exécuter sur un même cœur, ou parce que plusieurs messages asynchrones peuvent être envoyés par un même thread. De plus, n'importe quel cœur peut s'adresser à n'importe quelle instance du noyau (c.-à-d. n'importe quel cluster). Les canaux doivent donc pouvoir stocker plus d'un message afin de qu'ils soient traités par lots (*batch processing*). Les canaux sont donc des FIFOs logicielles, structures contenant plusieurs cases qui sont lues dans l'ordre où elles ont été remplies.

Placement des canaux

Très tôt dans la spécification, il a été décidé de charger le cœur qui envoie la commande (le client), plutôt que le cœur qui traite la commande (le serveur), afin de perturber le moins possible l'exécution des applications sur l'instance fournissant le service. Pour cela, les FIFOs logicielles sont placées dans le banc mémoire du côté du serveur. Ces FIFOs sont implémentées comme des tampons circulaires de taille fixe (contrairement à une implémentation en liste chaînée où la taille est dynamique), car allouer de la mémoire à distance (pour les nœuds de la liste chaînée) nécessiterait un autre message.

Nombre de canaux

Chaque cœur (client) doit pouvoir communiquer directement avec toutes les instances du noyau (serveur). Il faut donc au maximum $C \times N \times N$ canaux de communications, où N est le nombre de clusters, et C le nombre de cœurs par cluster. Mais ce nombre peut être réduit si plusieurs émetteurs se partagent un même canal. Le premier avantage est de minimiser l'encombrement mémoire et de mieux exploiter la mémoire allouée au canal (puisque le partage d'un canal permet aux cœurs actifs d'exploiter l'espace non utilisé par ceux qui sont inactifs). Le second avantage est que l'instance réceptrice n'a pas besoin de surveiller plusieurs canaux pour déterminer si un message a été posté. Nous avons

choisi la solution utilisant un seul canal par cluster (c.-à-d. par instance du noyau), qui est le point d'entrée unique pour tous les clients potentiels. On a donc un total de N FIFOs, possédant chacune $C \times N$ écrivains, et C lecteurs.

Synchronisation sans verrou

Le principal inconvénient de cette solution utilisant des FIFOS accessibles par plusieurs écrivains et plusieurs lecteurs est le risque de contention. Pour la synchronisation entre les écrivains nous avons utilisé le même mécanisme sans verrou que celui utilisé dans Popcorn Linux (décrit en détail dans le chapitre 3), qui réduit fortement le risque de contention. La synchronisation entre les lecteurs est assurée par un mécanisme de *light-lock* permettant de garantir qu'un seul cœur serveur se chargera de traiter la ou les commandes en attente dans la FIFO. Comme un *spin-lock*, un *light-lock* est un drapeau booléen qui garantit l'accès exclusif à une ressource. Le drapeau doit être positionné en utilisant une opération atomique de type Compare-and-Swap. Mais à la différence du *spin-lock*, un *light-lock* n'est pas bloquant : en cas d'échec (drapeau déjà positionné) la fonction de prise de drapeau rend immédiatement la main au demandeur, qui peut donc faire autre chose, ce qui est le comportement souhaité.

Stockage des messages

Le choix d'une FIFO statique (qui a donc des cases de taille fixe) est en contradiction avec la taille des messages, qui ont une longueur variable. Pour résoudre ce problème, trois principales solutions sont possibles. La première consiste à utiliser des cases assez grandes pour contenir n'importe quel message. La seconde consiste à utiliser plusieurs cases mémoire pour stocker les messages de taille importante. La troisième solution consiste à utiliser un tampon mémoire secondaire pour stocker le corps des messages, et de ne stocker dans la FIFO que des *descripteurs de message* de taille fixe, contenant en particulier l'adresse étendue du tampon secondaire. L'inconvénient de la première solution est que beaucoup d'espace mémoire est inexploité dans les cas où les messages envoyés sont plus petits que la taille maximale. De plus, cette solution implique une limite sur la taille des messages. L'inconvénient de la deuxième solution est la difficulté de reconstruire le message à partir de plusieurs cases. Nous avons donc retenu la troisième solution, et le tampon secondaire est alloué du côté de l'émetteur afin d'éviter l'envoi d'un second message pour une allocation distante.

Notification de l'arrivée d'un message

Deux grandes solutions existent pour notifier une instance du noyau de l'arrivée d'un message. La première solution consiste à ce que l'émetteur utilise une IPI (Inter Processeur Interrupt) afin d'interrompre l'un des cœurs de l'instance. Cette méthode impose au client une lecture distante supplémentaire pour déterminer si la FIFO est vide, car l'IPI ne doit pas être envoyée si la FIFO est non-vide. La deuxième solution consiste à ce que les cœurs du serveur scrutent régulièrement l'état de la FIFO, quand ils sont en mode noyau, pour détecter l'arrivée d'un message. Nous avons retenu une solution intermédiaire entre les deux précédentes. Elle consiste à utiliser la scrutation lorsqu'au moins l'un des cœurs

du côté serveur est en mode noyau, et d'utiliser une IPI uniquement lorsque tous les cœurs sont en mode utilisateur. Cette dernière impose au client un accès distant en lecture supplémentaire pour obtenir l'état courant des cœurs du serveur. Elle évite la latence importante de la solution de scrutation lorsqu'aucun cœur du serveur n'est en mode noyau, et évite le coût inutile d'une IPI.

Contexte de traitement d'un message sur le serveur

Un traitement d'un message sur le serveur est bloquant si son traitement exécute une fonction bloquante. Suivant le type du message envoyé, il peut être traité dans deux contextes différents. Si le message est bloquant, il doit être traité dans un thread noyau spécialisé. Si le message n'est pas bloquant, il peut être traité dans le contexte du thread utilisateur interrompu, c'est une optimisation permettant de réduire le temps de traitement.

Dans l'implémentation actuelle, tous les messages sont traités par un thread spécialisé.

Choix du cœur serveur

Afin de diminuer la latence de la réponse à un message, c'est le premier cœur qui détecte (par scrutation) l'arrivée des messages qui traite la commande. L'exclusivité du traitement à ce cœur est garanti par le mécanisme du *light-lock* décrit plus haut qui permet aux autres cœurs de continuer leur exécution normale. Une optimisation consiste à activer ce drapeau lorsqu'au moins un des cœurs est *idle* (sans thread utilisateur à exécuter) ce qui évite aux autres cœurs le coût de la scrutation et du traitement des RPCs.

Traitement par lot

Lorsqu'un cœur a été sélectionné pour accéder en lecture à une FIFO non-vide, il se charge de tous les messages présents dans la FIFO tant que celle-ci n'est pas vide. Ce traitement par lot sur un seul cœur permet de profiter de la localité dans les caches de premier niveau, aussi bien pour les données que pour le code de traitement des RPCs. Toutefois, si les messages sont nombreux, le traitement par lot ne suffit pas, et il faut pouvoir exploiter la puissance de l'ensemble des cœurs de l'instance. Pour cela, il peut être utile de mettre en place des FIFOs auxiliaires à un seul écrivain et un seul lecteur afin que le cœur sélectionné pour vider la FIFO principale puisse rediriger les messages sur les autres cœurs de l'instance. Ce mécanisme de redirection n'est utilisé que si le cœur sélectionné est surchargé. Il peut être amélioré en prenant en compte différents facteurs : le nombre de messages, le type des messages (nécessite l'ajout d'un champ au descripteur de message), la disponibilité des autres cœurs, etc.

Ce mécanisme de redirection n'a pas été implémenté.

Réponse

Avant l'envoi d'un message, le thread client doit allouer un tampon mémoire pour stocker les arguments de la commande. Ce même tampon est utilisé pour stocker les arguments de la réponse. Pour être informé de l'arrivée de la réponse, le client doit scruter périodiquement

un drapeau de ce tampon. Toutefois, dans le cas où un thread plus prioritaire est prêt à s'exécuter, le thread client est mis en attente passive et c'est au noyau (l'ordonnanceur de tâches) de scruter le drapeau et de réveiller le thread client lorsque la réponse arrive.

Interblocages

L'évitement d'interblocage repose sur deux propriétés. La première est que la réponse peut toujours être envoyée (le tampon de réponse est préalloué). La seconde est que toute commande reçue sera traitée. Cette seconde propriété est un défi, à cause des messages bloquants. La solution réside dans le lancement de threads de traitement spécialisés pour les RPCs bloquantes. Ainsi, il y a toujours un thread de traitement, sauf si aucun thread ne peut être créé en raison de la saturation de la mémoire. Dans ce cas, le message est tout de même traité en envoyant une réponse d'erreur indiquant l'indisponibilité d'espace mémoire (le système doit réserver un thread pour l'envoi de réponse d'erreur). Ces deux propriétés permettent uniquement d'éviter les interblocages au niveau de l'infrastructure de passage de messages. Les interblocages liés à des protocoles de plus haut niveau ne sont pas évités. Par exemple, si deux RPC A et B verrouillent chacune une ressource et essayent chacune de prendre la ressource verrouillée par l'autre, il y a interblocage.

4.2.4 Conclusion sur les canaux de communication

Nous avons apporté un soin particulier à définir un mécanisme RPC efficace et passant à l'échelle pour les communications entre instances du noyau, car ce mécanisme est fondamental dans une architecture multi-instances. Ce mécanisme supporte non seulement les messages non bloquants mais aussi les messages bloquants, sans qu'il y ait d'interblocage. Les performances sont évaluées et présentées dans le chapitre 6.

4.3 Stratégies de placement du système de fichiers

Dans une architecture manycore comme TSAR, pouvant contenir 1024 cœurs (256 clusters) partageant le même espace d'adressage, la principale contrainte est d'éviter les contentions, c'est-à-dire les situations où un grand nombre de cœurs cherchent à accéder simultanément au même banc mémoire physique. La thèse de G. Almaless a montré que toute contention de ce type entraîne un écroulement dramatique des performances.

Pour cette raison, le principe général retenu pour le placement des structures de données du système de fichiers (données et métadonnées) consiste à distribuer le plus uniformément possible les données sur l'ensemble des bancs de mémoire de la plateforme. Cette stratégie est appelée *interleave*.

Cette stratégie de distribution uniforme, en l'absence de connaissance sur les besoins réels des applications en termes d'accès aux fichiers, permet de minimiser les risques de contention, mais elle pose un problème de parcours de ces structures, particulièrement dans une architecture multi-instances où les accès aux données distantes reposent sur le mécanisme RPC. En effet, le cache des *dentrys* et des *inodes* par exemple a une structure arborescente.

Si ce cache est distribué sur tous les clusters, le parcours de cet arbre — pour retrouver un *inode* par son cheminom par exemple — va entraîner un nombre important de RPCs. Il faut donc optimiser ces structures, de façon à réduire le nombre de RPCs.

4.3.1 Cache des *inodes*

Pour le cache des *inodes*, la stratégie de placement consiste à distribuer les *inodes* sur l'ensemble des clusters en utilisant une fonction de hachage. Cette fonction prend en entrée l'identifiant unique du fichier et fournit en sortie un numéro de cluster. Dans le cas — rare — où le banc mémoire du cluster ainsi sélectionné est saturé, on peut en principe choisir un autre cluster, en utilisant le gestionnaire mémoire pour localiser un banc mémoire non saturé. Cette stratégie permet d'assurer la création des *inodes* même lorsque certains bancs de mémoire sont saturés, tout en équilibrant la répartition. Ce mécanisme de redistribution pose cependant un problème dans les systèmes de fichiers supportant la création dynamique de liens (*hard link*). La raison est que ce type d'*inode* doit pouvoir être retrouvé à partir de son numéro d'*inode* dans le cas où il est ouvert avec un autre nom. Une solution possible est d'utiliser une table globale permettant de retrouver les *inodes* dont le placement a été modifié. La vérification de cette table ne devrait pas être coûteuse pour les *inodes* dont le placement n'a pas été modifié, puisque de toute façon il faut accéder au disque dur pour rechercher l'*inode*.

> La gestion du cas où un banc mémoire est saturée n'a pas été implémentée. Seul un code d'erreur, indiquant le manque d'espace mémoire, est retourné.

4.3.2 Cache des *dentrys*

Toutes les structures *dentry* associées aux entrées d'un même répertoire sont placées dans le même cluster que la structure *inode* du répertoire auquel elles appartiennent. Cette stratégie de placement nous permet d'utiliser un cache de *dentrys* par *inode*. Surtout, elle permet de confiner certaines opérations, telles que la création ou la suppression d'un fichier, au sein d'une même instance.

La Figure 4.1 présente le placement des *inodes* et *dentrys* du fichier : */home/these/manuscrit* dans le cas d'une architecture à quatre clusters (c.-à-d. quatre instances). Sur cette figure, on trouve l'*inode* racine (ayant le numéro zéro) du système de fichier dans le cluster zéro. Dans ce même cluster, se trouve l'ensemble de ses *dentrys*, dont seules deux sont représentées : *etc* et *home*. Ce dernier *dentry* pointe vers l'*inode* qu'il représente, lequel est dans le cluster deux, ainsi de suite, jusqu'à atteindre l'*inode* 97 du fichier *manuscrit*. Notons que l'adresse de l'*inode* dans la structure *dentry* est une adresse étendue, c'est-à-dire formée de l'identifiant du cluster, plus l'adresse 32 bits locale à ce cluster. Enfin, le champ CG dans les deux structures *dentry* et *inode* est utilisé pour synchroniser les opérations d'accès (voir 4.4.2).

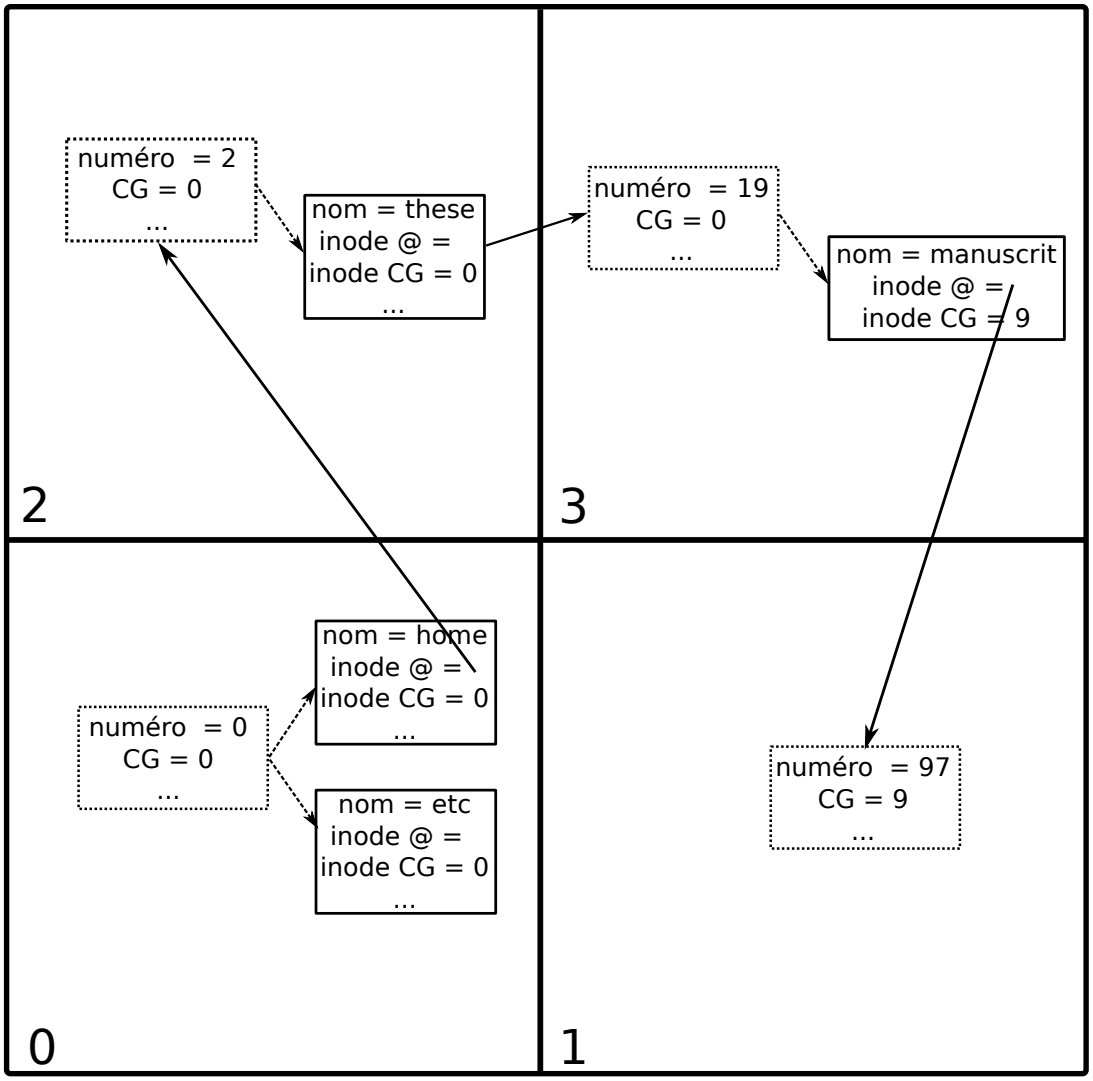


FIGURE 4.1 : Exemple de placement des *inodes* (représentés en traits pointillés) et des *dentrys* (en traits pleins) pour le chemin `/home/these/manuscrit` sur une architecture à quatre clusters.

4.3.3 Cache des pages

Comme dans le SE Linux, il y a un cache de pages par *inode* dans ALMOS-MK. Toutes les pages associées à un fichier sont placées dans le même banc mémoire que celui contenant l'*inode* auquel elles appartiennent. L'organisation détaillée du cache de pages est décrite dans le chapitre 5.

Ce choix permet d'éviter les accès distants lors des opérations d'entrée/sortie, telles que `read` ou `write`, car l'accès aux données du fichier se fait depuis le cluster contenant l'*inode* (voir 4.4.3). En effet, ces opérations doivent accéder à l'*inode* pour la synchronisation des accès concurrents, et pour mettre à jour les champs contenant la date du dernier accès. Ainsi, une seule RPC suffit pour effectuer une opération d'entrée/sortie, car celle-ci s'exécute localement, sans autre accès distant.

Ce choix de placement de toutes les pages d'un même fichier dans un seul cluster entraîne

cependant deux difficultés : d'une part, il peut causer la saturation du banc mémoire contenant l'*inode*. D'autre part, il peut créer un point de contention dans le cas où le fichier est accédé par l'intermédiaire de l'appel système `mmap`, par un grand nombre de threads ou de processus d'une même application. Notons que ce problème de contention ne se pose pas pour les appels système `read` ou `write`, car le mécanisme client/serveur séquentialise les accès.

Pour ce qui concerne la saturation, les pages du cache peuvent être placées dans les clusters voisins dès que le banc mémoire où se trouve l'*inode* commence à saturer.

Pour ce qui est de la contention, ALMOS-MK fournit au programmeur d'application les outils pour l'éviter : si l'appel système `mmap` utilise le mode privé, le gestionnaire mémoire d'ALMOS-MK réplique les pages du fichier dans le cluster du thread demandeur sans porter atteinte à la sémantique POSIX. Cette copie se fait au moment du premier accès distant (*copy-on-access*).

L'implémentation actuelle de cette stratégie de placement ne gère pas la saturation : toutes les pages de données sont placées localement au cluster où se trouve l'*inode*.

4.3.4 Fichiers ouverts

Chaque structure *file* représentant un fichier ouvert par une application est placée dans le même cluster que l'*inode* qu'elle représente. Ce placement permet d'utiliser un seul message pour effectuer les opérations d'entrée/sortie, `read` et `write`, même lorsque la structure *file* est partagée par plusieurs threads s'exécutant sur différents clusters, ce qui est le cas dans beaucoup d'applications parallèles multi-threads. En effet, si la structure *file* est partagée par N threads distribués sur différents clusters, la structure *file* pourra être placée sur le même cluster que l'un des threads, mais si la structure *file* et la structure *inode* ne sont pas sur le même cluster, toute opération d'entrée/sortie effectuée par les (N-1) autres threads nécessitera d'envoyer deux messages : un premier message pour accéder à la structure *file* (pour accéder à l'*offset*) puis un deuxième pour accéder à la structure *inode*.

Nous avons apporté deux optimisations à cette stratégie de base. La première optimisation consiste à répliquer dans le cluster de chaque thread utilisateur les champs en lecture seule de la structure *file*, tels que le mode d'ouverture du fichier, afin de pouvoir effectuer certaines opérations localement (par exemple la vérification des droits d'accès). La seconde optimisation consiste à cacher dans le cluster de chaque thread utilisateur le champ *offset*, dans le cas où la structure *file* n'est partagée avec aucun autre thread, ce qui améliore les performances des opérations `lseek` qui changent la valeur de l'*offset*. Ces champs répliqués ou cachés sont directement insérés dans l'entrée de la table des descripteurs de fichiers ouverts (répliquée dans chaque cluster, comme indiqué ci-dessous). Ce qui fait qu'une entrée de la table contient non seulement un pointeur vers la structure file, mais aussi les champs répliqués. Ce regroupement permet de faciliter la migration des processus, car la copie de toutes les données relatives aux fichiers ouverts est effectuée en une seule opération.

Notons que cette stratégie de placement est similaire à celle du SE Hare (voir chapitre 3).

4.3.5 Tables des descripteurs de fichiers

Pour rappel, la table des descripteurs de fichiers est privée à un processus, et contient les pointeurs vers les structures *file* de tous les fichiers ouverts par le processus, qui sont accessibles par tous les threads du processus. Dans les systèmes monolithiques traditionnels, tous les threads accèdent à la même table des descripteurs. Toutefois, une telle solution ne peut être utilisée dans un système à multiples instances, sous peine de doubler le nombre de messages requis pour les opérations d'entrées/sorties (un message pour l'accès à la table et un second message pour l'accès à la structure *inode*). Surtout, cette solution crée un point de contention au niveau du cluster contenant la table.

Pour éviter ce problème, la table des descripteurs de fichiers est répliquée dans chaque cluster où se trouve au moins un thread du processus. Le mécanisme permettant de garantir la cohérence entre les copies est décrit dans la section 4.4.3.

Puisque ALMOS-MK ne supporte pas encore la migration des threads (seule la migration des processus est supportée), la réplication des tables des descripteurs de fichiers n'a pas été implémentée.

4.3.6 Conclusion sur les stratégies de placement

Les stratégies de placement d'ALMOS-MK suivent deux règles. La principale vise à minimiser les risques de contention. La seconde vise à minimiser le nombre de messages entre instances du noyau lors de l'accès aux ressources distribuées. Ces deux objectifs sont parfois contradictoires, et nous avons recherché le meilleur compromis en donnant une priorité à la première règle.

La stratégie de placement des *inodes* consiste à les distribuer uniformément sur l'ensemble des bancs de mémoire. La stratégie de placement des *dentrys* s'appuie sur celle des *inodes* pour regrouper les *dentrys* ayant le même père dans le même banc mémoire (ce qui permet de réduire considérablement le nombre de messages lorsqu'un *dentry* est recherché au sein d'un *inode* répertoire).

De façon similaire, la stratégie de placement des caches de pages consiste à les placer dans le même cluster que l'*inode* auquel ils appartiennent. Toutefois, ces caches peuvent déborder sur les clusters voisins lorsque la mémoire locale est saturée. Par ailleurs, ALMOS-MK permet aux applications de contrôler (avec le flag "privé" de l'appel système `mmap`) la réplication automatique des pages pour rapprocher les données des threads utilisateurs et réduire la contention.

Les structures de fichiers ouverts sont aussi placées dans le même banc de mémoire que l'*inode*, tout en cachant localement au thread les champs en lecture seule. Le champ *offset* est aussi caché localement lorsque la structure n'est pas partagée entre plusieurs threads.

Enfin, la table des descripteurs, qui est partagée entre tous les threads d'un même processus, est répliquée dans chaque cluster contenant un thread utilisateur.

4.4 Accès concurrents

Pour résoudre le problème général des accès concurrents aux structures partagées de type listes chaînées ou arbres de recherche par un grand nombre de lecteurs, nous avons proposé et évalué un mécanisme de synchronisation appelé GECOS - présenté ci-dessous - qui passe à l'échelle à la fois en latence et en consommation mémoire.

Nous décrivons ci-dessous le mécanisme GECOS dans le contexte général d'une liste chaînée, puis nous présentons sa variante simplifiée utilisée dans le noyau ALMOS-MK pour synchroniser l'accès aux structures partagées du système de fichiers.

Remarque : Notons que pour la suite de ce chapitre, à moins que cela soit explicitement mentionné, tous les verrous utilisés sont à attente active.

4.4.1 Mécanisme GECOS

Le mécanisme GECOS (pour *GEneration Counter Optimistic Synchronisation*) a été défini pour permettre un accès sans verrou par plusieurs lecteurs et un écrivain aux structures de type liste chaînée : structures formées de plusieurs nœuds liés par des pointeurs. S'il y a plusieurs écrivains, ceux-ci doivent être synchronisés par un autre mécanisme, tel que des verrous. Contrairement aux mécanismes CR (lequel ne passe à l'échelle en performance) et RCU (lequel a une consommation mémoire non bornée, voir chapitre 3), ce mécanisme passe à l'échelle à la fois en performances et en consommation mémoire.

Le principe du mécanisme consiste à inclure dans chaque nœud de la structure chaînée un compteur de générations (CG). Ce CG est incrémenté par l'écrivain à chaque opération de suppression d'un nœud. Cette incrémentation est effectuée, après que le nœud a été détaché, et avant qu'il soit libéré, c.-à-d. rendu à l'allocateur de mémoire. Cette incrémentation permet aux lecteurs de détecter les scénarios où un nœud est supprimé de la structure et d'éviter ainsi le problème de l'utilisation après libération.

Problèmes à résoudre

Le mécanisme GECOS doit résoudre deux problèmes qui ne se posent pas avec les mécanismes CR ou RCU, car ces mécanismes interdisent la réutilisation immédiate des nœuds supprimés en retardant leur libération. À l'inverse, GECOS permet à l'écrivain de libérer immédiatement les nœuds supprimés, sans bloquer leur libération.

1. **Accès incomplet.**
   Ce problème survient lorsqu'un thread lecteur (TL) lit le lien (pointeur) vers un nœud et, avant d'accéder à son contenu, un thread écrivain (TE) le supprime puis l'insère à une position différente (voir le Tableau 1). Dans ce cas, le thread TL continuera sa traversée de la liste en manquant une partie de la liste.

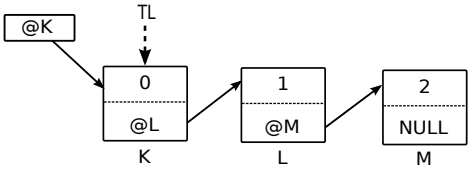
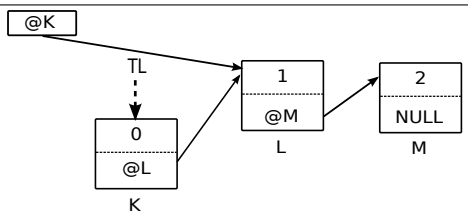
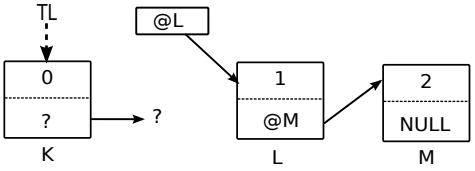
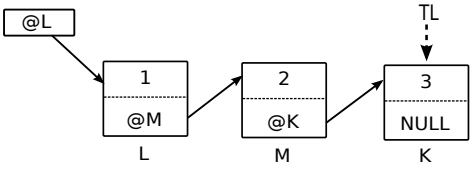
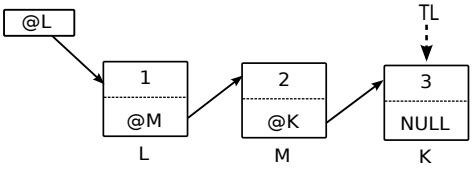
Opérations de TL	Opérations de TE	États de la liste
Traversée de la liste en cours (itération 0)		
	Suppression du nœud K (1/2) : détachement du nœud	
	Suppression du nœud K (2/2) : libération du nœud	
	(Ré-)insertion du nœud K (valeur 3)	
Opération de lecture incomplète		

TABLE 4.1 : Exemple de scénario du problème de l'accès incomplet : cas où le thread TL est en train d'accéder à un nœud alors que celui-ci a été supprimé puis inséré en parallèle par le thread TE

2. **Problème ABA.**
   Ce problème est semblable au précédent, mais plusieurs déplacements (suppression puis insertion) ont été effectués par le thread TE, de sorte que la position relative d'un nœud L par rapport à son prédécesseur, un nœud K, reste inchangée, bien que les deux nœuds aient été déplacés. Ici encore le thread TL continuera sa traversée de la liste en manquant une partie de la liste.

Le second problème est subtil et met en défaut la technique simple consistant à vérifier, à chaque nœud L traversé et après avoir lu son contenu, que le lien de son prédécesseur, le nœud K, n'a pas changé de valeur.

En effet, cette technique simple permet de s'assurer que (1) le nœud L n'a pas été déplacé (ce qui résout le deuxième problème) et que (2) le contenu du nœud L a été lu alors qu'il était

attaché au nœud K (ce qui résout le problème de validité, si l'écrivain met à zéro le lien des nœuds qu'il détache). Mais elle ne permet pas de détecter le troisième problème où la valeur A du lien du nœud K est modifiée à une valeur B puis remodifiée à sa valeur initiale A, d'où l'appellation de problème ABA.

Dans la suite, on désignera par nœud *stable* tout nœud qui n'a pas été supprimé en parallèle de son utilisation. Dans le cas contraire, le nœud est dit *instable*.

Accès en lecture

Nous détaillons ici les étapes que doit effectuer un lecteur respectant le mécanisme GECOS pour traverser un nœud n_1 d'une liste chaînée, à partir d'un lien (pointeur) l_0 :

1. **Lecture du Pointeur** (LPT) :
   Cette étape consiste à lire le pointeur vers le nœud n_1 à partir d'un lien l_0. Ce lien peut être, soit le pointeur contenu dans la tête de la liste, soit celui contenu dans le champ *next* du nœud précédent, que nous nommons n_0.

2. **Lecture du CG** (LCG) :
   Cette étape consiste à stocker la valeur du CG du nœud n_1 dans une variable locale cg_1.

3. **Vérification du Lien** (VLN) :
   Cette étape consiste à vérifier que le lien l_0 pointe toujours vers le nœud n_1.

4. **Vérification de la Stabilité du nœud Précédent** (VSP) :
   Cette étape consiste à vérifier que le nœud dans lequel on a lu le lien l_0 est *stable*. Cette étape n'est pas nécessaire lorsque le lien est lu à partir de la tête de liste, car la tête de liste est toujours stable, étant donné qu'elle est toujours placée à la position 0 de la liste et n'est pas libérable. Par contre, s'il s'agit d'un nœud alors ce nœud a été précédemment traversé : c'est le nœud n_0. Ainsi pour vérifier sa stabilité, il faut comparer son CG à la valeur stockée lors de sa traversée à l'étape précédente : cg_0.

5. **Lecture Des Données** (LDD) :
   Cette étape consiste à lire les données voulues du nœud n_1.

6. **Vérification de la Stabilité du nœud Courant** (VSC) :
   Cette étape consiste à vérifier que le nœud n_1 est stable en comparant son CG à la valeur cg_1 stockée à l'étape LCG.

Les trois premières étapes garantissent que la lecture du CG du nœud n_1 a été effectuée alors que n_1 était placé après le lien l_0 (après la tête de liste ou après le nœud n_0). C'est l'étape VLN qui permet de vérifier que n_1 n'a pas été déplacé pendant l'étape LCG. Cependant cette vérification est nécessaire mais non suffisante en raison du problème ABA (le nœud a pu être supprimé puis réinséré avec un contenu différent). L'étape VSC vérifie que n_1 n'a pas été déplacé depuis l'étape LCG. Sachant que l'étape VSC est aussi nécessaire pour valider l'étape LDD, son exécution est effectuée en dernier. Quant à l'étape VSP, elle assure que les

trois premières étapes ont été effectuées depuis un nœud stable. L'échec de la vérification VSP requiert que l'accès en lecture soit repris depuis le début de la liste. L'échec de la vérification VLN ne requiert pas une telle reprise. Il suffit de reprendre l'accès depuis l'étape LPT courante afin de relire le lien vers le nœud n_1. Ces échecs sont supposés rares, c'est pourquoi le mécanisme est qualifié d'optimiste.

Ce mécanisme, comme le mécanisme CR, requiert la pérennité du type des structures concernées (voir 3.3.2). Pour plus de détails, une étude comparative de ce mécanisme avec les mécanismes tels que RCU et CR est disponible dans [45].

Après avoir défini et publié ce mécanisme, nous avons trouvé dans la littérature un mécanisme très similaire [16], proposé dans le contexte des arbres de recherche binaires. Les auteurs ont pris un chemin différent du nôtre pour aboutir au même résultat. Ils se sont inspirés des mécanismes à mémoire transactionnelle logicielle alors que notre mécanisme s'inspire des mécanismes RCU et seqlock. Cela explique que les auteurs ne citent aucun des travaux auxquels nous nous sommes comparés : RCU, HP ou CR.

4.4.2 Accès concurrents au cache des *inodes* et des *dentrys*

Rappelons que le cache des métadonnées n'est pas répliqué, mais qu'il est distribué sur toutes les instances (c.-à-d. sur tous les clusters, voir 4.3.2). Pour synchroniser les accès à ce cache, plusieurs mécanismes sont utilisés.

Le mécanisme CG-MK

Le passage à une architecture multi-instances a requis une adaptation du mécanisme GECOS. Cette adaptation a pour but de regrouper les accès au sein de chaque instance du noyau, afin de minimiser le nombre de messages entre instances. Par ailleurs, dans une architecture multi-instances, on peut utiliser des verrous sans risque de contention, tant que leur utilisation est locale à une instance du noyau. Le mécanisme GECOS est donc combiné avec l'utilisation de verrous et de compteurs de références. Nous avons appelé cette nouvelle version du mécanisme : **CG-MK**.

Les verrous

Deux verrous sont utilisés. Le premier verrou permet de séquentialiser les modifications effectuées dans le cache des *inodes*, local à une instance. Il existe donc un verrou de ce type par instance.

Le second verrou protège à la fois les accès aux données de l'*inode* (compteur de références, temps du dernier accès, etc) et les accès au cache des *dentrys* appartenant à cet *inode* (s'il s'agit d'un *inode* de type répertoire). Puisque ces données sont par *inode*, il existe un verrou de ce type par *inode*.

Le compteur de références

On trouve un compteur de références (CR) par *inode*. Celui-ci est utilisé pour stabiliser l'*inode* durant l'accès à son contenu. L'utilisation du CR, ici, est différente de celle du

mécanisme sans verrou décrit dans le chapitre 3. Le CR ici, n'est pas manipulé par des opérations atomiques CAS. Les manipulations sont protégées par le verrou de l'*inode*, comme l'ensemble des données de l'*inode*. En plus d'être utilisé pour le parcours de l'arborescence du système de fichiers, le CR est aussi utilisé pour stabiliser de manière prolongée un *inode*. Il est, par exemple, incrémenté lors de chaque ouverture du fichier et décrémenté lors de la fermeture.

Le compteur d'alias

Ce compteur est présent dans chaque *inode*. Il permet de compter le nombre d'alias (de noms) dont dispose un fichier. Celui-ci est aussi protégé par le verrou de l'*inode*.

Les drapeaux `inload`

On utilise un drapeau dans la structure *inode* et un autre dans la structure *dentry*. Ces drapeaux ont la même sémantique. En cas d'activation, ils indiquent que l'*inode* ou le *dentry* sont en cours de chargement ou de suppression. Nous appelons ce type de drapeau : `inload`.

Parmi les six opérations (de **A** à **F**) que nous décrivons dans la suite, seules quatre ont été implémentées : résolution du chemin d'accès (**A**), résolution inverse (**B**), insertion (**C**) et suppression (**D**). Ces opérations sont suffisantes pour avoir un système fonctionnel. Deux opérations n'ont pas encore été implémentées : l'ajout d'alias (**E**) et le renommage (**F**).

Opération A : Résolution du chemin d'accès

Pour rappel, la résolution du chemin d'accès pour retrouver un *inode* à partir du cheminom consiste à parcourir le cache des métadonnées lequel est composé à la fois d'*inodes* et de *dentrys*, . Ce parcours commence à partir d'un *inode* répertoire (généralement le répertoire courant du processus ou la racine du système de fichiers), puis cherche dans le cache de *dentrys* associé à cet *inode* le *dentry* correspondant au premier nom du cheminom. Ce *dentry* permet de localiser l'*inode* suivant. L'opération se répète ainsi jusqu'à ce que l'*inode* du dernier nom du cheminom soit trouvé.

Cette opération de résolution nécessite un mécanisme de synchronisation pour éviter le problème de l'utilisation après libération (voir chapitre 3). Notons que ce problème ne peut pas être traité par un simple compteur de références, puisqu'il est possible qu'entre la lecture de l'adresse de l'*inode* à partir d'un *dentry* se trouvant dans un cluster X et l'accès à l'*inode* dans le cluster Y, le *dentry* et l'*inode* soient supprimés et réutilisés (par un autre thread) pour représenter un autre fichier.

Pour résoudre ce problème, plusieurs mécanismes sont utilisés. On utilise d'un côté les verrous et le CR pour effectuer la recherche d'un *dentry* au sein d'une instance. On utilise le mécanisme CG-MK pour passer d'une instance à une autre. Plus précisément, pour passer d'un *dentry* au sein d'une instance à l'*inode* se trouvant dans une autre instance.

Une implémentation directe du mécanisme GECOS nécessite uniquement de placer un CG dans les nœuds instables, c.-à-d. les *inodes*, puisque les *dentrys* sont stabilisés par un verrou. Toutefois, afin de regrouper les accès au sein d'une même instance, c.-à-d. de minimiser le nombre de messages, le mécanisme CG-MK recopie la valeur du CG dans les *dentrys*. Ainsi, la lecture du pointeur d'*inode* et du CG (étapes LPT et LCG) peut être effectuée depuis le *dentry* sans nécessiter un accès à l'*inode*. Les étapes VLN et VSP ne sont pas nécessaires, car le *dentry* est stabilisé par le verrou du cache.

Une fois que le pointeur et le CG de l'*inode* sont lus, l'accès à l'*inode* peut être effectué dans l'instance où il se trouve (ces informations sont alors envoyées en argument d'une RPC). Il faut ensuite stabiliser l'*inode*. Cela est effectué en vérifiant la valeur du CG (étape VSC) tout en incrémentant le CR. Ces deux opérations sont atomiques, car protégées par le verrou du cache des *inodes* de l'instance. Une fois l'*inode* stabilisé, la recherche du *dentry* peut être effectuée pour résoudre le second nom du cheminom.

La recherche du *dentry* aurait pu être effectuée sans verrou, avec les CG. Toutefois, l'utilisation des compteurs de références et des verrous permet de simplifier le mécanisme.

De plus, puisque les structures accédées appartiennent au cache du système de fichiers, il faut parfois faire un accès au disque afin de charger les structures absentes du cache (*dentry* ou *inode*). Dans ce cas, il est nécessaire de prendre le verrou pour ajouter le *dentry* ou l'*inode* dans le cache des métadonnées. Quant au CR, il est nécessaire afin de pouvoir relâcher le verrou pendant l'accès au disque ou pendant l'accès aux autres instances (afin de charger l'*inode* dans une autre instance, d'accéder aux structures spécifiques au système de fichiers réel se trouvant dans une autre instance, etc). Ainsi, on évite de prendre le verrou pendant de longues périodes, et on évite de potentiels interblocages (voir l'opération d'insertion plus bas).

Exemple

Pour illustrer ce parcours, nous décrivons l'ouverture du fichier `/home/these/manuscrit` (Figure 4.1). Le but de cette opération est de localiser l'*inode* final du fichier « `manuscrit` », puis de créer la structure *file* dans l'instance où se trouve l'*inode*. Le CR de l'*inode* final est incrémenté, car la structure *file* pointe vers cet *inode*.

Pour simplifier l'exemple, toutes les structures sont supposées déjà présentes dans le cache, et ne nécessitent donc pas d'accès au disque.

1. Le thread demandeur commence par envoyer une RPC à l'instance contenant l'*inode* racine du système de fichiers, c.-à-d. le cluster **0**. Cette RPC a comme argument de base l'adresse de l'*inode* racine et le premier nom du cheminom. Cette RPC trouve le *dentry* (la recherche du *dentry* est protégée avec le verrou de l'*inode*) correspondant à « `home` » et renvoie dans la réponse le CG et l'adresse étendue de l'*inode* contenu dans le *dentry*.
2. Une fois ces deux éléments obtenus, une nouvelle RPC est envoyée à l'instance contenant l'*inode* du fichier « `home` », c.-à-d. le cluster **2**. Cette RPC a pour arguments : l'adresse de l'*inode*, la valeur du CG de l'*inode* et le second nom du cheminom.

L'*inode* est d'abord stabilisé en incrémentant le CR, les droits d'accès sont vérifiés, puis le *dentry* correspondant à « `these` » est recherché dans le cache des *dentrys* de l'*inode*.

Les informations contenues dans le *dentry* (adresse étendue et CG de l'*inode*) sont à nouveau envoyées en réponse à la RPC. À la fin de l'exécution de la RPC, le CR de l'*inode* est décrémenté. Dans le cas où la stabilisation échoue (c.-à-d. il y a eu conflit entre cette opération et une opération de suppression), la RPC renvoie une valeur d'erreur indiquant que la résolution doit être reprise depuis le début du chemin d'accès.

3. La même opération est réalisée dans le cluster **3**, lequel contient l'*inode* du répertoire « `these` ».
4. Une dernière RPC est envoyée vers l'instance contenant l'*inode* du fichier « `manuscrit` », c.-à-d. le cluster **1**. La RPC vérifie les droits d'accès après avoir stabilisé l'*inode*, puis alloue et initialise une structure *file*. Puis, la RPC renvoie une réponse contenant en plus des informations de la structure *file* (adresse de la structure et données en lecture-seule répliquées), le code d'erreur indiquant l'absence d'erreur, à moins que la stabilisation ou l'allocation de la structure *file* ait échoué. Enfin, notons que le CR de l'*inode* n'est pas décrémenté, car l'incrémentation du CR, faite au moment de la stabilisation de l'*inode*, est utilisée par la structure *file* pour garder l'*inode* en mémoire. La décrémentation est faite par l'opération fermant le fichier ouvert : `close()`.

Sans le CG, qui est incrémenté à chaque suppression des *inodes*, il serait impossible de savoir si les *inodes* accédés sur les autres instances correspondent bien à celui lu depuis le *dentry*. Ceci est dû au fait qu'entre deux RPCs, un *inode* (pointé par l'adresse envoyée en argument) a très bien pu être supprimé et utilisé pour stocker les données d'un autre fichier.

Par ailleurs, lorsqu'un *dentry* ou un *inode* a le drapeau `inload` activé, l'opération de résolution doit réessayer plus tard. Ce drapeau est utilisé pour signaler que l'*inode* ou le *dentry* sont dans un état instable : en cours de chargement ou de suppression (voir ci-dessous).

Pour que le mécanisme des CG-MK fonctionne correctement, il faut assurer la pérennité des structures, c.-à-d. qu'il faut que les segments mémoire contenant les structures *inode* ou *dentry* ne soient pas réutilisés pour stocker d'autres types de données, car il faut garantir la validité du champ CG. Pour obtenir cette garantie ALMOS-MK réserve certains segments mémoire pour le stockage des structures *inode* et *dentry* .

Risque de famine

Le compteur de génération CG aurait pu être remplacé par un compteur de références CR. En effet, il est possible d'ajouter un CR dans le *dentry* pour signaler à tout thread voulant supprimer l'*inode* du *dentry* que celui-ci est en cours d'utilisation. Cette technique fonctionne puisque la suppression d'un *inode* passe par le *dentry*, mais elle requiert l'envoi d'une RPC supplémentaire afin de décrémenter le CR du *dentry* une fois que l'accès au cluster contenant son *inode* est terminé. C'est pour éviter ce surcoût que cette solution a été écartée.

Néanmoins, cette solution peut être utilisée dans les cas où le mécanisme CG-MK échoue à stabiliser les *inodes* du parcours à plusieurs reprises. Cette solution permet ainsi d'éliminer le risque de famine du mécanisme CG-MK. Cette approche est similaire à la solution de Linux pour parcourir l'arborescence du système de fichiers (voir 3). Il utilise un mécanisme peu coûteux mais avec possibilité de famine (RCU et seqlock) pour effectuer le parcours. Il utilise par contre un CR en cas d'échec répété.

La solution permettant de parcourir l'arborescence en utilisant le mécanisme des CR n'a pas été implémentée.

Opération B : Résolution inverse

Pour rappel, la résolution inverse consiste à obtenir le chemin d'accès à un répertoire à partir de l'*inode* représentant ce répertoire. Celle-ci consiste à traverser l'arborescence du système de fichiers depuis un *inode* intermédiaire vers la racine. Cette opération est très semblable à l'opération précédente.

Elle s'effectue par étapes. Chaque étape consiste à trouver, à partir d'un *inode*, le *dentry* auquel il appartient, puis l'*inode* parent, et ainsi de suite jusqu'à la racine. Si le répertoire de départ n'est pas stable, il est nécessaire de stabiliser l'*inode* de départ avant de commencer l'opération.

Cette opération nécessite un mécanisme de synchronisation pour se protéger du problème de l'utilisation après libération. Toutefois, cette fois-ci, il faut se protéger non pas des opérations de suppression, mais des opérations de renommage. En effet, les structures parentes d'un *dentry* ne peuvent pas être supprimées, mais l'opération de renommage peut changer l'arborescence du système de fichiers en supprimant des *dentrys* (voir l'opération de renommage ci-dessous). En effet, il est possible qu'entre la lecture des informations d'un *dentry* se trouvant dans le cluster Y, à partir d'un *inode* se trouvant dans le cluster X, et l'accès à ce *dentry*, le *dentry* Y soit supprimé suite à une opération de renommage.

Comme pour le problème précèdent, on utilise le mécanisme CG-MK. Toutefois, il n'est pas nécessaire, ici, de le combiner avec le mécanisme des verrous ou des compteurs de références. Ceci est dû au fait que ces deux mécanismes sont utilisés pour prendre en compte le cas où certaines structures ne se trouvent pas en mémoire. Pour la résolution inverse, il ne risque pas d'y avoir d'accès au disque : les structures accédées sont toujours présentes en mémoire. De plus, pour ce mécanisme, il n'est pas nécessaire d'accéder à des structures complexes, telles que celles formant le cache des *dentrys*.

Puisque dans l'implémentation actuelle, l'opération de renommage n'a pas été implémentée, le mécanisme CG-MK n'est pas utilisé pour cette opération.

Exemple

Supposant que le répertoire courant du processus pointe vers */home/these*. Cet exemple suppose un placement des structures tel que décrit dans la Figure 4.1. Notons, toutefois,

que cette figure ne présente pas les pointeurs permettant de parcourir l'arborescence dans le sens inverse. Cet exemple décrit les étapes permettant de retrouver le chemin complet de l'*inode* répertoire numéro **19**, c.-à-d. celui ayant comme cheminom */home/these*.

1. Une RPC est envoyée vers le cluster **3**, lequel contient l'*inode* représentant le répertoire `these`. Cette RPC renvoie en argument les informations du *dentry* `these` : adresse étendue et CG. Ces informations sont extraites de l'*inode* numéro **19**.
2. Une RPC est envoyée vers le cluster deux. Cette RPC renvoie à partir de l'adresse du *dentry* et du CG, les informations du *dentry* suivant.
   Pour cela, elle accède au *dentry* « `these` »pour lire l'adresse (le pointeur) de l'*inode* auquel elle appartient (l'*inode* représentant le répertoire *home*). Aussi, le nom contenu dans le *dentry* est lu. Ces deux lectures sont validées en comparant la valeur du CG du *dentry* à celui envoyé en argument (lu dans l'étape précédente). Cette vérification nous assure que le pointeur vers l'*inode* est valide, sans le risque de déréférencer un pointeur NULL (autrement seule la vérification ci-dessous est nécessaire).
   À partir de cet *inode* sont lues les informations du *dentry* suivant : « `home` ». Ces informations doivent elles aussi être validées en comparant les valeurs des CG.
   Enfin, ces informations et le nom du *dentry* sont envoyées en réponse à la RPC.
3. Le nom du *dentry* est ajouté à un tampon local au demandeur de l'opération, lequel sera envoyé en réponse de l'appel système. Puis une RPC est envoyée vers le cluster zéro. Le reste de cette étape est similaire à l'étape précédente. Avec la différence que les informations lues depuis l'*inode*, lesquelles représentent la racine, pointent vers une adresse NULL, signifiant la fin du parcours inverse.

Opération C : Insertion d'un nœud

L'opération d'insertion consiste à introduire dans le cache des métadonnées un nouvel *inode* et le *dentry* associé. On commence par allouer une structure *dentry*, on initialise le champ indiquant le nom du fichier, on lève le drapeau `inload` puis on l'insère dans le cache des *dentrys* de l'*inode* du répertoire parent. Une fois le *dentry* inséré, les autres champs de la structure sont initialisés à partir des données sur disque (ou en mémoire si le *dentry* appartient à un système de fichier en mémoire, tel que *ramfs*). Ensuite, l'*inode* est construit dans un autre cluster en utilisant une RPC (à moins qu'il n'existe déjà en mémoire : alias de noms).

La construction de l'*inode* se fait en trois étapes. En premier, on alloue une structure *inode*, et on l'initialise avec des valeurs par défaut (compteur de référence à 1, compteur d'alias de nom à 1, taille à 0, etc.). En deuxième, l'*inode*, avec le drapeau `inload` activé, est inséré dans le cache des *inodes*, local à l'instance où il est créé, après avoir pris le verrou du cache. Enfin, le contenu de l'*inode* est chargé depuis le disque (à moins qu'il appartienne à un système de fichiers en mémoire).

Lorsque le *dentry* et l'*inode* sont chargés en mémoire, les drapeaux `inload` sont désactivés.

L'utilisation du drapeau `inload` permet, entre autres, d'éviter la prise de verrous pendant l'accès au disque ou pendant la RPC construisant l'*inode*.

Opération D : Suppression d'un nœud

Cette opération est effectuée en activant d'abord le drapeau `inload` du *dentry*, puis une RPC est envoyée pour supprimer l'*inode*.

L'*inode* n'est supprimé que si le compteur d'alias est à 1. Sinon, seuls les compteurs (compteur de références et compteur d'alias) sont décrémentés sans que l'*inode* soit supprimé. De plus, si l'*inode* est de type répertoire est que le compteur de référence est supérieur à 1, l'*inode* n'est pas supprimable (car en cours d'utilisation). Quant aux *inodes* de type fichiers, ils peuvent être supprimés (sans que leur contenu soit libéré) même s'ils sont en cours d'utilisation (certaines applications requiert la fonctionnalité de supprimer un fichier en cours d'utilisation [25]). Dans ce dernier cas, le contenu du fichier est libéré à la fermeture du fichier.

La suppression d'un *inode* se fait en activant d'abord le flag `inload`, puis en décrémentant les compteurs, puis en supprimant l'*inode* du cache des *inodes*. Le contenu (en mémoire et sur disque) de l'*inode* n'est libéré que si le CR est à 0 (après la décrémentation). Enfin, à chaque fois qu'un *inode* est supprimé du cache, son CG est incrémenté.

Une fois que la RPC est terminée, le *dentry* peut être supprimé du cache des *dentrys* auquel il appartient.

Opération E : Création d'alias

Cette opération est similaire à celle permettant d'insérer un *dentry*. La seule différence est qu'il faut en premier lieu récupérer l'adresse de l'*inode* tout en incrémentant les compteurs de référence et d'alias, puis le *dentry* est créé.

Opération F : Renommage d'un nœud

Cette opération se déroule en trois étapes :

1. Activation du drapeau `inload` dans le *dentry* à supprimer, c.-à-d. le *dentry* contenant l'ancien nom du fichier, tout en récupérant l'adresse de l'*inode* ;
2. Création du *dentry* du nouveau nom de fichier, celui-ci pointe vers l'*inode* dont l'adresse a été récupérée dans la première étape ;
3. Suppression de l'ancien *dentry*, mais avant de libérer le *dentry*, son CG est incrémenté pour que l'opération de résolution inverse puisse détecter que le *dentry* a été renommé. Toutefois, dans le cas où la création du nouveau *dentry* a échoué, il suffit de réactiver le drapeau de l'ancien *dentry*.

L'ordre entre ces étapes est important afin de respecter l'atomicité de l'opération telle qu'elle est requise par le standard POSIX. L'activation du drapeau `inload` à la première étape permet de retarder les opérations des threads qui souhaitent accéder avec l'ancien nom jusqu'à ce que le nouveau nom soit créé. Même si, au niveau système, il y a un moment où aucun des fichiers

n'est accessible (à la seconde étape), l'utilisateur ne peut pas remarquer cela, car sa réponse n'aboutit qu'une fois que le second nom est créé. Ainsi, du point de vue de l'utilisateur il y a toujours un et un seul nom valide.

4.4.3 Accès au cache des pages

Rappelons que les caches des pages (un cache par fichier ou par répertoire) sont distribués sur tous les clusters. Ce cache est situé dans le même cluster que l'*inode* représentant le fichier sauf en cas de saturation de la mémoire.

Le standard POSIX impose que les opérations d'entrées/sorties soient atomiques. Pour cela, toutes ces opérations sont synchronisées entre elles par un verrou multilecteurs à attente passive situé dans la structure *inode*. L'utilisation d'un tel verrou est nécessaire, car l'accès aux données du cache peut accéder au disque. De même que pour les verrous du cache des *dentrys*, celui-ci n'est accessible qu'au sein d'une instance. Le risque de contention est donc parfaitement contrôlé.

Le transfert (en lecture ou en écriture) des données de ce cache vers les tampons utilisateurs se fait directement avec l'adressage physique. Cette copie directe permet d'optimiser le temps d'accès à ce cache. En effet, cette optimisation permet d'éviter de copier à deux reprises les données : une fois par le service des RPC et une seconde fois pour effectuer le transfert.

Pour résumer, les trois étapes effectuées par une RPC pour lire ou écrire les données de ce cache sont :

1. Prise du verrou multilecteurs du cache des données suivant le mode de l'accès ;
2. Copie des données en utilisant directement l'adressage physique ;
3. Relâchement du verrou multilecteurs.

4.4.4 Accès aux descripteurs des fichiers ouverts

Rappelons que la stratégie de placement utilisée pour un fichier ouvert permet d'avoir une seule copie des structures représentant les fichiers ouverts. Cette copie est localisée dans le cluster contenant l'*inode* du fichier ouvert.

La synchronisation des données de cette structure est aussi assurée par un verrou multilecteurs à attente passive. L'utilisation d'un tel verrou permet de prendre en compte le cas où le disque est accédé, suite à un accès au cache de pages, durant la prise de verrou.

4.4.5 Accès aux tables des descripteurs de fichiers

Le standard POSIX spécifie que les descripteurs de fichiers ouverts par un processus peuvent être partagés par tous les threads du processus. De plus, les modifications dans la table des descripteurs de fichiers doivent être atomiques. Pour éviter la contention, ALMOS-MK réplique la table des descripteurs dans tous les clusters contenant au moins un thread du processus

concerné. Lorsque tous les threads se trouvent dans le même cluster, ils se partagent la même copie de la table des descripteurs, et il n'y a pas de problème de cohérence. Mais lorsque les threads sont distribués sur plusieurs clusters, il y a plusieurs copies de la table et il y a donc un problème de cohérence qui doit être traité. La synchronisation entre les copies repose sur la définition d'une *copie centrale*, stockée dans le cluster contenant l'*inode* représentant le fichier concerné. Cette copie centrale est la référence, et contient tous les descripteurs de fichiers ouverts par le processus. Les *copies secondaires*, se comportent comme des caches, et contiennent donc seulement un sous-ensemble de l'ensemble des descripteurs de fichiers ouverts.

> Les principales opérations concernant ces tables sont décrites ci-après. Étant donné que l'implémentation actuelle d'ALMOS-MK ne supporte pas la migration des threads, l'ensemble des opérations décrites ci-dessous n'a pas été implémenté.

Opération A : Insertion ou Duplication

Lorsqu'un thread veut insérer ou dupliquer une entrée dans la table des descripteurs (appel système `open` ou `dup`), il le fait d'abord dans la copie centrale en utilisant une RPC. Si l'insertion passe avec succès, cette même RPC retourne une copie de l'entrée afin de mettre à jour la copie secondaire (à moins que le thread se trouve déjà dans le cluster contenant la copie centrale). Sinon, une erreur est renvoyée au thread demandeur.

Opération B : Suppression

La suppression d'une entrée se fait d'abord dans la copie centrale, puis dans les copies secondaires. Pour cela, la RPC qui met à jour la table centrale doit aussi incrémenter de façon atomique un CG présent dans la structure *file* , qui est placée dans le même cluster que l'*inode* du fichier ouvert. Cette incrémentation atomique du CG définit la date de la fermeture du fichier : les opérations exécutées avant l'incrémentation trouvent le fichier ouvert, tandis que les opérations exécutées après l'incrémentation trouvent le fichier fermé.

Opération C : Recherche

Cette opération permet d'accéder aux informations stockées dans le descripteur de fichier, à partir de l'index désignant une entrée dans la table. La recherche se fait d'abord dans la copie secondaire locale. Deux cas sont possibles :

1. Si l'entrée est vide, une RPC est envoyée au cluster contenant la copie centrale pour vérifier que l'entrée est réellement vide. Si c'est le cas, la recherche retourne un code d'échec. Sinon l'entrée manquante est copiée dans la table locale et on passe au deuxième cas.

2. Si l'entrée n'est pas vide, l'opération ayant justifié la recherche est exécutée normalement (`read`, `write`, `mmap`, etc.). Toutefois, dans le cas où le fichier ouvert est partagé, il est

nécessaire de vérifier que la valeur du CG présente dans l'entrée de la table est la même que celle présente dans la structure *file*. Si cette vérification réussit, l'opération est exécutée normalement, sinon l'entrée n'est plus à jour et la recherche retourne un code d'échec.

4.4.6 Conclusion sur les accès concurrents

La synchronisation des accès concurrents a été guidée par deux règles. La première est de toujours chercher à diminuer le nombre de messages. Cette règle permet d'éviter les latences induites par l'envoi de messages. La seconde est d'éviter de prendre des verrous (à attente active) lors de l'envoi de message. Cette règle permet de diminuer la durée liée à la prise de verrous. Elle permet aussi d'éviter les interblocages.

Ainsi, un mécanisme de synchronisation a été développé pour assurer l'accès aux structures réparties du système de fichiers. Ce mécanisme est basé sur l'utilisation de CG. Il permet de synchroniser les accès avec un minimum de messages et sans prendre de verrous globaux, évitant ainsi le risque de contention.

L'accès au cache des données est synchronisé par un verrou multilecteurs à attente passive. Ce verrou se situe dans le cluster où se trouve l'*inode* du cache. Ce verrou étant restreint à un seul cluster, le risque de contention est aussi limité.

Un verrou multilecteurs à attente passive est aussi utilisé pour synchroniser l'accès aux descripteurs des fichiers ouverts.

Enfin, pour la table des descripteurs de fichiers ouverts, seule structure à répliquer une copie de base, on utilise un mécanisme de synchronisation différent pour les écritures et les lectures. Pour les écritures, elles sont toutes synchronisées par un verrou se trouvant sur le cluster ayant la copie de base. Pour les lectures, l'accès se fait d'abord sur la copie locale à moins que l'entrée recherchée soit vide. Ce dernier cas est le seul où les lecteurs accèdent à la copie de base. Cette stratégie permet d'éviter l'envoi de message à chaque lecture de la table. Elle permet aussi de diminuer la contention au niveau du cluster contenant la copie de base.

4.5 Conclusion

La caractéristique la plus importante de la solution proposée consiste à utiliser une structure multi-instances au niveau du noyau. Ce choix nous a permis de résoudre le problème de la capacité d'adressage limité du noyau. Il possède aussi le double avantage de localiser les accès et de limiter la contention sur les verrous locaux. L'inconvénient potentiel est l'augmentation de la latence des accès distants liée au mécanisme client/serveur. Toutefois, cette pénalité devrait être limitée par l'utilisation directe de l'espace d'adressage physique.

La stratégie de placement est totalement statique : une fois qu'une structure est placée, elle ne peut plus être déplacée. Ainsi et pour éviter tout risque de contention, la stratégie de placement élue consiste à déployer uniformément les données sur l'ensemble des bancs de mémoire. Cette stratégie reste risquée dans le cas où les fichiers ne sont pas partagés,

puisqu'elle peut entraîner un grand nombre d'accès distants, ce qui peut causer une charge importante sur le réseau d'interconnexion (NOC) entre les clusters.

Quant aux mécanismes de synchronisation, ils varient suivant les structures considérées. Pour le cache des métadonnées, on trouve l'utilisation d'un mécanisme hybride combinant deux mécanismes. Le premier est basé sur l'utilisation de verrous, qui permet de synchroniser les accès au sein d'une instance ; le second utilise les compteurs de génération pour passer d'une instance à une autre. Le cache des pages et les structures de fichiers ouverts utilisent un mécanisme à base de verrous à attente passive. Pour la table des descripteurs, un mécanisme a été spécifié pour assurer la cohérence entre les différentes copies.

Les performances des différentes solutions proposées dans ce chapitre sont étudiées dans le chapitre 6. Le chapitre 5 détaille l'implémentation des solutions décrites ci-dessus.

CHAPITRE 5

Implémentation

Les solutions présentées dans le chapitre précédent ont été implémentées dans le noyau ALMOS-MK. Pour des contraintes liées au temps de développement, cette implémentation n'est pas complète : les trois principaux services manquants sont (1) le service création de threads, (2) le service de migration des processus, et (3) le service permettant l'accès au réseau (ce service n'était pas présent dans ALMOS).

Nous présentons donc dans ce chapitre, les quatre étapes qui ont permis de transformer ALMOS en ALMOS-MK. En premier lieu, nous présentons les modifications qui ont permis de démarrer en parallèle plusieurs instances indépendantes du noyau. En deuxième lieu, nous détaillons l'implémentation du mécanisme des RPC. En troisième lieu, nous décrivons l'implémentation des systèmes de fichiers (VFS, FAT32, RAMFS, DEVFS, SYSFS). Enfin, nous présentons le service de création d'un processus sur un cluster distant, qui était nécessaire pour les expérimentations.

5.1 Découpage du noyau en plusieurs instances

Cette étape consiste à démarrer plusieurs instances indépendantes du noyau sur les différents clusters de la plateforme. Aucun mécanisme de communication entre instances n'est nécessaire à ce stade. Chaque instance du noyau gère indépendamment les ressources locales à un cluster : cœurs et banc mémoire. Seuls les périphériques sont partagés entre les différentes instances.

5.1.1 Bootloader

Comme dans la plupart des systèmes d'exploitation, le démarrage d'ALMOS requiert l'exécution d'un code indépendant, appelé *bootloader*, qui effectue deux tâches principales. La première tâche consiste à initialiser la table des pages du noyau. Seule la partie de l'espace virtuel du noyau est initialisée à ce stade. Par la suite, tout nouveau processus hérite de cette partie de la table, et l'autre partie de l'espace virtuel, contenant les segments utilisateur, est initialisée à la demande lors de l'exécution du processus. La seconde tâche du *bootloader* consiste à initialiser

la structure *boot_info*, qui contient la description de l'architecture matérielle cible.

Le *bootloader* d'ALMOS a été modifié pour charger, en parallèle, le code noyau dans la mémoire de tous les clusters. Dans chaque cluster, c'est le cœur d'index 0 qui effectue la réplication. Les noyaux sont placés à l'adresse zéro locale du cluster : le noyau du cluster [0,0] est placé à l'adresse `0x00 0000 0000`, celui du cluster [1,0] à l'adresse `0x10 0000 0000`, etc.

Par ailleurs, la construction de la table des pages a été fortement modifiée, puisqu'une seule entrée de la table est initialisée. Cette entrée permet de projeter à l'identique les adresses de la première page physique du code noyau, c.-à-d. les adresses entre 0 et 4095. Cette page est utilisée pour passer de l'adressage virtuel, utilisé par les applications utilisateurs, à l'adressage physique noyau, ou inversement. En effet, la première page du code noyau contient le code permettant de désactiver ou d'activer la mémoire virtuelle.

Notons que ce *bootloader* peut être utilisé pour démarrer le noyau dans différentes configurations : une instance de noyau par cœur, une instance par cluster, en adressage virtuel, etc.

5.1.2 Modifications dans le noyau

Suppression des accès distants

Tous les accès à des données distantes ont été supprimés. Pour atteindre cet objectif, il a été nécessaire de modifier la DQDT (voir 2.2.1) pour que toutes les demandes de placement de tâches ou d'allocation mémoire renvoient un processeur ou un banc mémoire local au cluster.

Réplication des descripteurs de l'architecture matérielle

Les structures décrivant l'architecture matérielle, c'est-à-dire les périphériques (adresses de base, nombre de canaux, etc.) et les clusters (nombre de cœurs, taille de la mémoire, etc.) sont répliquées.

Changement du mode d'adressage à l'entrée et à la sortie du noyau

Cette modification consiste à changer le mode d'adressage lorsque l'on entre ou lorsque l'on sort du noyau. Chaque processeur passe en adressage physique lorsqu'il entre dans le noyau, et repasse en adressage virtuel juste avant d'accéder à l'espace utilisateur. Ces transitions s'appuient sur une page mémoire possédant la même adresse dans l'espace physique et dans l'espace virtuel ("*identity mapping*", ou mapping identité). Il s'agit de la page zéro du banc mémoire physique de chaque cluster. Cette page est "mappée" dans l'espace virtuel de tout processus avec les protections noyau.

Fonctions permettant l'accès aux bancs de mémoire distants

Ces fonctions permettent à une instance du noyau d'accéder directement aux bancs de mémoire (ou aux périphériques) placés dans un autre cluster que le cluster local. Toutes les adresses physiques sont définies par deux paramètres, qui sont l'adresse 32 bits

interne à un cluster, et un identifiant de cluster. Trois types d'accès sont actuellement supportés :

1. Les fonctions permettant l'accès en lecture ou en écriture à des mots ou à des octets.
2. Deux fonctions implémentant deux opérations atomiques : *COMPARE_AND_SWAP* et *ATOMIC_ADD*. La première opération est une écriture conditionnelle. La seconde opération permet d'effectuer une incrémentation / décrémentation atomique d'une valeur à une certaine donnée.
3. Une fonction *PHYSICAL_MEMCPY* permettant de copier des données entre deux bancs de mémoire physiques.

5.1.3 Accès aux périphériques

Dans la version initiale d'ALMOS, les périphériques étaient accédés au travers de la mémoire virtuelle. Les pilotes de périphériques ont été modifiés pour utiliser les fonctions définies ci-dessus permettant l'accès à des données distantes en adressage physique.

Les périphériques externes sont des ressources partagées entre toutes les instances du noyau (contrairement aux processeurs et à la mémoire, qui sont des ressources privées gérées par chaque instance du noyau), ce qui nécessite de synchroniser les accès. Pour cela, on utilise des verrous à attente active placés dans le cluster zéro. Cette solution est temporaire, car elle peut engendrer de la contention. Une meilleure solution serait d'utiliser le service de passage de messages ou d'utiliser des verrous hiérarchiques distribués.

5.1.4 Interruptions

Pour gérer les interruptions générées par les périphériques externes, nous exploitons le composant matériel IOPIC pour contrôler dynamiquement le routage des interruptions. Les interruptions signalant la terminaison d'une opération d'entrée/sortie sont routées directement vers l'un des cœurs du cluster contenant l'instance du noyau qui a déclenché l'opération d'entrée/sortie.

L'avantage de cette technique est qu'elle permet d'effectuer la gestion d'une interruption par le cœur demandeur du service. De plus, le coût de cette reconfiguration est faible, puisqu'elle ne requiert qu'une seule écriture distante dans un registre de configuration du composant IOPIC.

L'implémentation actuelle est rudimentaire, puisque les boites aux lettres disponibles dans les concentrateurs d'interruptions situés dans chaque cluster (composant matériel XICU 2.1.1) sont statiquement allouées aux différents périphériques, ce qui limite fortement le nombre de périphériques supportés. Cette implémentation peut être facilement améliorée sans dégradation significative des performances, en allouant dynamiquement les boites aux lettres lors du lancement de chaque opération d'entrée/sortie.

5.2 Implémentation du service de passage de messages

Ce service fournit une interface de type RPC (Remote Procedure Call), c'est-à-dire littéralement l'appel d'une fonction distante. Le code du noyau étant simplement répliqué dans tous les clusters, l'adresse locale d'une fonction est la même dans toutes les instances du noyau. De même que l'adresse locale du tampon mémoire utilisé pour transmettre les arguments et des valeurs de retour.

L'interface RPC a été implémentée en utilisant principalement des macros du préprocesseur du langage C. En plus des arguments de la fonction à exécuter, cette interface requiert un premier argument qui spécifie le numéro de l'instance destination. L'envoi d'une RPC n'est effectué que si ce numéro désigne une instance distante. Sinon, si le numéro spécifie l'instance locale, la fonction est directement exécutée sans RPC.

Comme indiqué dans le chapitre précédent, l'envoi d'un message se fait en utilisant une FIFO logicielle située dans le cluster cible, et partagée par tous les émetteurs.

Pour savoir si un cœur est en train d'exécuter du code noyau ou utilisateur, chaque cœur dispose d'un drapeau booléen. Ce drapeau est mis à **1** lors du passage de l'espace utilisateur à l'espace noyau. Il passe à **O** lors du passage de l'espace noyau à l'espace utilisateur. Ainsi, grâce à ce drapeau, un cœur émetteur d'une RPC peut savoir si l'un des cœurs de l'instance destination est en train de s'exécuter en mode utilisateur ou en mode noyau. Si tous les cœurs de l'instance sont en train de s'exécuter en mode utilisateur et que la FIFO indique qu'il s'agit du premier message, une IPI est envoyée à l'un des cœurs de l'instance pour notifier l'arrivée d'un nouveau message. Sinon aucune IPI n'est envoyée, car la notification est implicite : c'est le noyau qui scrute périodiquement la FIFO. Cette scrutation se fait à deux endroits du code noyau : (1) à chaque entrée ou sortie du noyau et (2) à chaque libération d'un verrou. Si plusieurs verrous sont pris de façon imbriquée, c'est le dernier verrou libéré qui effectue la scrutation, car le noyau interdit les changements de contextes tant qu'un verrou est pris.

5.3 Implémentation du système de fichiers

Au dessous de la couche VFS, nous avons implanté trois systèmes de fichiers : Les systèmes FAT, DEvFS et SysFS ont été modifiés pour supporter la structures multi-instances. De plus, pour pouvoir évaluer les performances du noyau indépendamment des performances du contrôleur de disque externe, nous avons implémenté un quatrième système de fichiers en mémoire, appelé RamFS.

5.3.1 VFS

Cache des *inodes* et des dentrys

Le VFS est la couche la plus importante puisqu'elle gère l'ensemble des caches. Le cache des *inodes* est implémenté en utilisant une table de hachage. Il y a un cache des *inodes* par instance. Le cache des *dentrys* est lui aussi implémenté avec une table de hachage.

Toutefois, il y a une table de hachage de *dentry* par *inode*. Ces deux tables ont une taille statique. La table de hachage des *inodes* a une taille égale à une page mémoire, ce qui équivaut à 512 entrées. Celle des *dentrys* contient 64 entrées. Toutefois, il serait sans doute préférable de pouvoir ajuster dynamiquement la taille de ces tables.

L'implémentation de ce cache est suffisamment flexible pour permettre d'autres stratégies de placement que celle proposée dans le chapitre 4. En effet, chaque système de fichiers peut choisir sa propre stratégie de placement. De plus, le VFS propose également une stratégie permettant de répliquer un système de fichiers par instance. Cette réplication ne dispose pas de mécanisme de cohérence. Ainsi, seuls les systèmes de fichiers accédés en lecture seule peuvent l'utiliser. Enfin, le noyau peut être configuré pour utiliser une stratégie de placement locale, similaire à celle qu'on trouve dans les noyaux commerciaux, tels que Linux.

Cache des pages

Le cache des pages est implémenté avec un arbre de recherche (*radix tree*). Il y a un arbre par inode. Cet arbre permet de retrouver à partir d'une position de lecture ou d'écriture (`offset`) la page du cache correspondante. Contrairement à une implémentation standard, ce ne sont pas les adresses des descripteurs de pages qui sont cachées par l'arbre. L'arbre cache les numéros des pages physiques ou *PPN* (*physical page number*) des pages de données. L'intérêt des *PPN* est de pouvoir identifier n'importe qu'elle page mémoire, qu'elle soit locale à l'instance ou non, avec moins de 32 bits (seulement 28 bits).

Dans cette première implémentation, toutes les pages d'un cache sont placées dans le même cluster que l'*inode* auquel le cache appartient.

Autres structures

En plus des structures des caches et des tables de descripteurs, on retrouve des structures permettant de gérer les systèmes de fichiers. Celles-ci sont de deux types.

Le premier type, appelé `fs_type_s`, permet de décrire les systèmes de fichiers supportés par le noyau. L'ensemble des structures de ce type sont regroupées dans un tableau, qui est vérifié à chaque fois qu'un système de fichiers doit être monté (`mount`).

Les principales informations contenues dans cette structure sont l'identifiant du système de fichiers, tel que le nom du système de fichiers ; et les opérations permettant de monter le système de fichiers. L'ensemble de ces informations étant statiques, c'est-à-dire en lecture seule, le partage du tableau entre instances est assuré en répliquant la structure dans toutes les instances.

Le second type, appelé *vfs_context_s*, permet de décrire un système de fichiers monté. Cette structure contient plusieurs informations, telles que la taille de bloc du système de fichiers ou le pointeur vers l'*inode* racine du système de fichiers. L'ensemble des informations étant en lecture seule, cette structure est aussi répliquée dans toutes les instances.

Enfin, notons que, dans l'implémentation actuelle, il n'y a pas d'appels système permettant de monter un système de fichiers. Les systèmes de fichiers sont montés à l'initialisation du système d'exploitation suivant des directives définies lors de la compilation.

5.3.2 Systèmes de fichiers supportés

En plus des structures génériques du VFS, chaque système de fichiers possède ses propres structures de données. Pour vérifier que notre implémentation du VFS supporte un véritable système de fichiers, nous avons porté le système de fichiers FAT32 sur notre VFS.

FAT32

Le système de fichiers FAT32 définit principalement la structure FAT (*File Allocation Table*), qui permet d'allouer et de libérer des blocs du disque. L'ensemble des blocs alloués aux différents fichiers sont représentés par des listes chainées dans la structure FAT. Cette structure peut occuper plusieurs dizaines de Moctets. La gestion de cette structure FAT est sous la responsabilité du système de fichiers FAT32 sans intervention du VFS.

Dans notre implémentation, l'ensemble de cette structure FAT a été placé dans une seule instance. Ainsi, toutes les requêtes d'allocation, de libération ou de recherche sont transférées par RPC à l'instance contenant cette structure. Découper la structure FAT aurait été un travail complexe qui n'aurait pas nécessairement apporté un gain en performance important : les manipulations de cette structure sont rares et souvent nécessitent l'accès au disque. De plus, l'optimisation des performances intrinsèques de ce système de fichiers ne faisait pas partie de nos objectifs.

RAMFS

Pour implémenter le système de fichiers RAMFS, aucune structure supplémentaire n'est nécessaire, puisqu'il utilise directement les caches du VFS comme moyen de stockage. Les inodes, *dentrys* et les pages de données de ce système de fichiers ne peuvent être évincés des caches. Ceci convient à notre objectif qui est de tester la scalabilité du VFS sans prendre en compte les limitations imposées par l'accès au disque externe (voir 6).

DevFS et SysFS

DevFS et SysFS ne sont pas de vrais systèmes de fichiers. Ce sont plus des interfaces pour accéder aux périphériques (dans le cas de DevFS) ou pour accéder aux informations de configuration du noyau (dans le cas de SysFS). Ce type de système de fichiers ne stocke pas de données dans le cache de données. Les opérations d'entrées/sorties permettent alors soit d'accéder à un périphérique, soit de lire ou de modifier une configuration du noyau.

Ces deux systèmes de fichiers ont été répliqués par instance. Cette réplication ne nécessite pas de mécanisme de cohérence, car il n'y a pas de données partagées par les différents réplicas. La seule incohérence qui résulte de cette réplication concerne les métadonnées, telles que la date du dernier accès. Ce type d'incohérence n'est pas compatible avec la norme POSIX.

5.4 Création distante de processus

5.4.1 Aperçu

Dans ALMOS-MK, il faut qu'une instance du noyau d'un cluster (X) puisse créer un processus dans n'importe quel cluster (Y) de l'architecture. Dans le cas où X et Y sont des clusters différents, il s'agit d'une création distante, qui nécessite une RPC.

Dans les systèmes UNIX, la création d'un nouveau processus se fait généralement en deux temps. Dans un premier temps, le processus voulant créer un nouveau processus, se duplique en utilisant, l'appel système `fork()`. Le processus appelant le `fork()` s'appelle alors le père tandis que le processus dupliqué s'appelle le fils. La valeur de la fonction `fork()` retourné au fils est zéro. La valeur retournée au père est l'identifiant du processus fils, appelé PID (Process IDentifier). Ce dernier peut être utilisé par le père ou un autre processus pour, par exemple, arrêter l'exécution du fils.

Dans un second temps, le processus fils exécute un des appels système de la famille `exec()`. Ce dernier a au moins un argument, qui est le cheminom du fichier binaire de l'application à exécuter par le nouveau processus. Les arguments supplémentaires dépendent de l'application lancée. Dans ALMOS-MK, c'est lors de l'exécution de cette opération que le processus est déplacé.

Dans la suite, nous présentons la distribution des identifiants de processus, les principales structures utilisées puis la description des appels système `fork()` et `exec()`.

5.4.2 Identifiants et structures des processus

La plage des identifiants est statiquement distribuée entre les instances du noyau, c.-à-d. que chaque cluster dispose d'un certains nombre de PID qui lui sont propres.

Dans ALMOS-MK, chaque instance a une table qui permet de représenter l'ensemble des processus qui lui appartiennent. Chaque entrée de cette table contient une adresse étendue (numéro de cluster + adresse dans le cluster) qui permet d'accéder au descripteur du processus, quel que soit le cluster auquel il appartient. Un descripteur de processus contient toutes les informations relatives à un processus, telles que l'adresse de la pile, le type du processus, etc.

L'utilisation d'une adresse étendue est nécessaire, car un processus peut être exécuté dans un cluster autre que celui auquel il appartient. C'est le cas dans la phase transitoire existante entre l'exécution de `fork()` et d'`exec()` (voir ci-dessous), mais cet accès distant peut aussi être utile pour le mécanisme d'équilibrage de charge, lors de la migration d'un processus vers un autre cluster contenant des cœurs oisifs (ce mécanisme n'est pas encore implémenté dans ALMOS-MK).

Ainsi, le PID, qui est un entier, est divisé en deux parties. La première partie, les bits de poids fort, contient le numéro du cluster de l'instance propriétaire du processus. La seconde partie, les bits de poids faible, contient un identifiant local qui indexe la table des processus locaux au cluster propriétaire.

Cette structuration des identifiants permet de localiser directement le cluster propriétaire d'un processus à partir de son identifiant, ce qui minimise le nombre de messages entre instances du noyau, et évite la création d'un goulot d'étranglement.

Dans la suite, on désigne par — cluster propriétaire — le cluster auquel appartient le PID donné au processus. Le — cluster hôte — désigne le cluster dans lequel la tâche s'exécute.

5.4.3 Opération `fork()`

Le choix du placement d'un processus se fait au moment de l'appel système `fork()`. La stratégie actuellement implémentée dans ALMOS-MK est un simple round-robin par cluster. Cette stratégie est temporaire en attendant d'implémenter d'autres, plus sophistiquées, telle que la DQDT (cf. 2.2.1). De plus, ALMOS-MK permet à une application utilisateur de spécifier explicitement un cluster cible.

Une fois que le cluster cible est choisi, l'opération `fork()` demande au cluster cible l'allocation d'un identifiant de processus. Cette allocation se fait en envoyant une RPC au cluster cible. Cette RPC contient en argument l'adresse étendue du processus et renvoie un PID. Une fois que le PID est reçu, le processus continue à s'exécuter dans le même cluster que le père jusqu'à l'exécution du `exec()`.

5.4.4 Opération `exec()`

C'est au moment de cette opération que le nouveau processus est déplacé vers le cluster propriétaire. Le fait de retarder ce déplacement jusqu'à cette opération simplifie énormément la tâche, puisqu'il suffit d'envoyer au cluster cible, en utilisant une RPC, les arguments du `exec()` ainsi que certaines informations héritées du père : les variables d'environnement et les descripteurs de fichiers ouverts (voir le chapitre 4).

Une fois que ces informations sont reçues par l'instance du noyau du cluster sélectionné, la création d'un nouveau processus se fait comme dans les systèmes UNIX classiques : initialisation d'une table de pages, lecture du binaire, etc. La principale différence est que, dans le cas d'une création distante, ALMOS-MK alloue les données du processus avant initialisation, car ces données ne sont pas directement accessibles par le cluster propriétaire avant l'exécution du `exec()`.

5.4.5 Scénario détaillé de création distante

Pour ce qui concerne l'opération `fork()`, les étapes sont les suivantes :

1. **Allocation et initialisation d'un descripteur de processus**.
   Cette étape est effectuée dans le cluster propriétaire du processus père.

2. **Allocation d'un PID**.
   Suivant la politique de placement, une RPC est envoyée au cluster cible pour lui demander

d'allouer un PID et d'enregistrer l'adresse du descripteur du processus fils, alloué à l'étape précédente, dans sa table des processus. Ce PID est enregistré dans le descripteur du processus fils.

3. **Duplication des données du père**.
   Les données suivantes sont dupliquées : les données relatives au système de fichiers, telles que la table des descripteurs de fichiers ; les descripteurs de régions virtuelles ; la table des pages (partiellement dupliquée en raison du mécanisme *copy-on-write*) ; duplication des informations de l'unique thread du processus, telles que l'état des registres et le contenu de la pile (si le processus père est multi-threads, seule la structure thread de celui qui exécute l'appel système `fork()` est répliquée, et le processus fils sera donc mono-thread).

4. **Enregistrement du thread principal**
   Le processus père enregistre le thread principal du processus fils dans l'ordonnanceur du cœur appelant l'opération `fork()`.

Une fois l'opération `fork()` terminée, le processus fils continue à s'exécuter dans le cluster propriétaire du processus père jusqu'à l'appel système `exec()`. On fait ici l'hypothèse que le cluster cible sélectionné lors du `fork()` est différent du cluster propriétaire du processus père, et il faut donc maintenant faire migrer le processus fils vers son cluster propriétaire. Les étapes de l'opération `exec()` sont les suivantes :

1. **Préparation des arguments la RPC**.
   Dans cette étape on crée une structure qui intègre l'ensemble des données nécessaires à l'opération `exec()` : PID du processus fils ; PID du processus parent ; les descripteurs de fichiers ; le nom de la fonction à exécuter ; les arguments s'il y en a ; les variables d'environnement du processus.

2. **Exécution de la RPC**.
   Une fois que les arguments sont prêts, la RPC est exécutée. Cette dernière commence par allouer un descripteur de processus ainsi que les structures qui y sont rattachées, telles que la table des pages, puis elle exécute une sous-fonction chargée d'initialiser le processus à partir des variables d'environnement du processus et des arguments (binaire et argument(s) de l'exécutable).

3. **Lancement du processus**.
   Une fois que les structures sont initialisées, le descripteur du processus fils est attaché à l'ordonnanceur du cœur cible. Ce dernier est chargé de lancer, au moment approprié, le processus sur le cœur cible, et le code binaire sera chargé dans la mémoire du cluster cible « à la demande », lors du traitement normal des défauts de page.

4. **Terminaison de l'ancien processus**.
   Si aucune erreur n'est survenue, l'ancien processus, celui créé au moment du *fork()*, est terminé. Ceci signifie que le thread du processus est retiré de l'ordonnanceur et que les données du processus sont libérées.

Ce mécanisme de création distante de processus ne suffit évidemment pas pour faire d'ALMOS-MK un système complet, mais il est suffisant pour tester le système de fichiers présenté dans le chapitre 4. Cette fonctionnalité a été principalement développée par Pierre-Yves Péneau durant son stage de Master [48].

5.5 Conclusion

Dans ce chapitre nous avons décrit l'implémentation, dans le noyau ALMOS-MK, des solutions présentées dans le chapitre 4, et nous avons présenté l'ensemble des mécanismes supportant l'architecture multi-instances.

Elle a été facilitée par le fait que le noyau ALMOS initial possédait déjà une organisation par cluster, puisque chaque cluster disposait d'un *cluster manager* chargé de gérer toutes les ressources d'un même cluster.

Comme cela est détaillé dans le chapitre 6, cette implémentation est suffisante pour analyser les performances et le passage à l'échelle des structures de données distribuées du système de fichiers, mais elle reste très incomplète. On notera en particulier l'absence de service de création de threads à distance. Un tel service est indispensable pour lancer une application parallèle multi-threads sur l'ensemble des cœurs d'une même architecture manycore. Pour l'instant, un processus multi-threads est nécessairement confiné dans un seul cluster.

CHAPITRE 6

Évaluations et résultats

Dans ce chapitre, nous cherchons à évaluer expérimentalement les solutions proposées et décrites dans les chapitres 4 et 5. Nous avons pour cela exécuté, sur un prototype virtuel de l'architecture manycore TSAR, différents benchmarks permettant d'évaluer les performances, et en particulier d'analyser le passage à l'échelle lorsque le nombre de coeurs augmente.

6.1 Questions investiguées

Nous précisons ci-dessous les différentes questions auxquelles les expérimentations décrites cherchent à répondre.

1. La première question porte sur la caractérisation du mécanisme de communication client-serveur entre les différentes instances du noyau. Pour éliminer au maximum les risques de contention, nous avons choisi une approche *multi-instances*, où tous les accès aux ressources distantes se font par RPC (Remote Procedure Call). Ceci peut sembler paradoxal dans une architecture où tous les coeurs partagent le même espace d'adressage physique. Cependant, nous souhaitons évaluer le coût intrinsèque de ce mécanisme de communication par RPC.

2. La seconde question cherche à évaluer l'impact de la structure *multi-instances* du noyau sur les performances des autres services système, comparée à une structure *mono-instance*. Cette architecture distribuée se caractérise aussi par l'exécution du noyau en mode physique. Le principal service qui est étudié est celui de la création distante de processus.

3. La troisième question est de quantifier le passage à l'échelle de l'architecture logicielle proposée pour le système de fichiers, lorsqu'une même ressource partagée (fichier ou répertoire) est accédée en parallèle par plusieurs processus. Le but de cette expérimentation est d'analyser la façon dont le noyau gère les goulots d'étranglement liés au comportement intrinsèque des applications.

4. La quatrième question vise également à analyser le passage à l'échelle, mais cette fois-ci, lorsque des ressources indépendantes (fichiers ou répertoires) sont accédées en parallèle par plusieurs processus. Le but, ici, est de vérifier que les structures de données du noyau, ainsi que les stratégies de placement retenues, ne créent pas de goulot d'étranglement lié au système lui-même.

5. La cinquième question porte sur la stratégie de placement du système de fichiers. La distribution uniforme sur les différents clusters minimise les risques de contention, mais cette politique peut avoir un effet négatif sur les temps d'accès, par rapport à une stratégie de placement au plus près du processus utilisateur. Nous cherchons donc à analyser l'impact de cette stratégie sur les performances, pour différents types d'applications.

6.2 Plate-forme expérimentale

6.2.1 Architecture matérielle

L'ensemble des expérimentations ont été effectuées sur le prototype virtuel de l'architecture TSAR. Ce modèle de simulation écrit en langage SystemC est précis au cycle et au bit près. Il y a 4 coeurs par cluster, mais il est possible de faire varier le nombre de clusters dans la plate-forme. Chaque coeur dispose d'un cache L1 d'instructions de 16 Ko et d'un cache L1 de données de 16 Ko. Les TLB d'instructions et de données disposent chacune de 64 entrées. Enfin, le cache L2 présent dans chaque cluster à une taille 256 Ko. La quantité de mémoire (RAM) utilisée est de 256 Mo par cluster.

Notons que l'implantation matérielle VLSI de cette architecture - réalisée par le laboratoire CEA-LETI - a démontré que les évaluations de performances réalisées sur ce prototype virtuel étaient fiables, et qu'on pouvait espérer une fréquence de fonctionnement proche de 1 GHz.

6.2.2 Systèmes d'exploitation

En plus de ALMOS-MK, nous avons utilisé le système d'exploitation NetBSD, pour comparer les performances d'ALMOS-MK à celles d'un SE commercial existant, en termes de passage à l'échelle, lorsque le nombre de coeurs devient grand. Le portage de NetBSD sur l'architecture TSAR a été réalisé par Manuel Bouyer (Ingénieur de recherche au LIP6).

Pour éviter les limitations imposées par la bande passante du contrôleur de disque, l'ensemble des expérimentations portant sur la scalabilité du système de fichiers utilisent un système de fichiers en mémoire : ALMOS-MK utilise le système de fichier *RamFS*, et NetBSD utilise le système de fichier *mfs*.

6.3 Benchmarks Utilisés

Les benchmarks utilisés peuvent être classés en deux catégories. La première regroupe les benchmarks séquentiels. La seconde regroupe les benchmarks parallèles.

6.3.1 Benchmarks séquentiels

Nous avons utilisé deux benchmarks séquentiels :

Création de processus distants

Ce simple benchmark consiste à créer, depuis le cluster **0**, un nouveau processus sur le cluster $N-1$ de l'architecture, N correspond au nombre de clusters. Il s'exécute en trois étapes :

1. Dès son lancement, le benchmark affiche le temps courant, puis crée un nouveau processus en exécutant les appels système *fork()* et *exec()*. Ce dernier a pour argument le chemin courant de l'exécutable de ce benchmark ainsi que le numéro du processus créé.
2. lorsque le nouveau processus est lancé, le temps courant est affiché et puis le processus s'auto-détruit.

La sortie standard de ces deux processus est redirigée vers un fichier. Le résultat de ce benchmark est obtenu en comparant les deux temps affichés.

Postmark

Ce benchmark permet de simuler le comportement d'un serveur d'e-mails [30]. Ce benchmark s'exécute en trois phases :

1. **Création d'un ensemble de fichiers**.
   Cette phase consiste à créer un certain nombre de fichiers dans un même répertoire.
2. **Exécution des transactions**.
   Quatre types de transactions existent. Le premier type permet de créer un fichier ; le deuxième permet de supprimer un fichier ; le troisième lit les données d'un fichier ; enfin, le quatrième ajoute des données à la fin d'un fichier. Le type de transaction à exécuter ainsi que les fichiers visés, sont choisis aléatoirement.
3. **Suppression des fichiers**.
   Cette phase consiste à supprimer tous les fichiers créés auparavant.

Ce benchmark peut être configuré avec différents paramètres, permettant de définir le nombre de fichiers, la taille des fichiers, et le nombre de transactions effectuées par le processus. Ce benchmark est utilisé uniquement dans sa version parallélisée (voir ci-dessous).

6.3.2 Benchmarks Parallèles

Les benchmarks parallèles utilisés comportent une phase séquentielle, exécutée par un seul processus, et une phase parallèle, exécutée par N processus, ou N est égal au nombre de coeurs de la plate-forme. On mesure uniquement la durée de la phase parallèle.

Parallel read

Ce benchmark consiste à lancer plusieurs processus qui effectuent en parallèle des opérations de lecture sur le même fichier. On détaille ci-dessous les différentes étapes :

1. **Création du fichier partagé**.
   Le processus initial crée un fichier vierge, initialisé à zéro. La taille du fichier est de $128 * N$ octets.
2. **lancement des processus**.
   Le processus initial lance N processus, N étant le nombre de coeurs. Les processus sont explicitement placés sur les coeurs de la plate-forme de manière *round-robin*. Le lancement des processus se fait en exécutant l'appel système *fork()* suivi de *exec()*. Chaque processus a un identifiant unique compris entre 0 et $N - 1$. on note cet identifiant I.
3. **Synchronisation par barrière**.
   Afin de s'assurer que les processus effectuent les opérations de lecture en parallèle, le fichier partagé contient une barrière de synchronisation, implémentée comme un bitmap dans les premiers octets du fichier. Chaque processus doit écrire une valeur non nulle dans l'octet I du fichier, puis entre dans une boucle de scrutation sur les valeurs contenues dans les N premiers octets du fichier. Lorsque tous les octets sont non nuls, la phase parallèle commence.
4. **Exécution en parallèle**.
   Chaque processus exécute à plusieurs reprises la même opération. Cette opération consiste à lire 128 octets du fichier. Chaque processus commence la lecture à l'octet $128 * I$. Ainsi, avant chaque lecture, le benchmark exécute d'abord une opération permettant de positionner l'offset du fichier : appel système *lseek()*.

Parallel Postmark

Ce benchmark est une parallélisation du benchmark séquentiel *Postmark* présenté ci-dessus. Il lance plusieurs processus qui exécutent le benchmark *Postmark*. Chaque processus travaille sur ses propres fichiers, stockés dans un répertoire privé.

Ce benchmark effectue les mêmes étapes que le benchmark précédent. La seule différence est qu'au lieu d'exécuter des opérations de lectures en parallèle, chaque processus exécute l'application *Postmark*.

Parallel RO-Postmark

La principale différence entre ce benchmark et le précédent est que les transactions des processus parallèles sont effectuées sur un même ensemble de fichiers partagés. De plus, on se limite aux transactions de lecture, afin d'éviter les conflits entre processus.

Les fichiers partagés sont créés par le processus initial dans la première étape, en plus du fichier permettant d'implémenter la barrière de synchronisation. Les deux autres étapes se déroulent comme pour les deux benchmarks précédents. Dans la dernière étape, plusieurs applications *Postmark* sont lancées, en se restreignant à des transactions de lecture.

L'intérêt de ce dernier benchmark est de simuler un comportement fréquent dans les applications parallèles : Il s'agit du cas où tous les fichiers sont d'abord chargés par un premier processus puis accédés, en parallèle, par les autres.

6.4 Résultats

6.4.1 Structure multi-instances

Dans cette première partie, nous essayons de mesurer l'impact sur les performances de la structure multi-instances, et de la communication par RPC.

Coût d'une RPC

La première expérimentation évalue le coût moyen d'une RPC, en lançant $(N-1)$ RPC synchrones depuis le cluster **0** vers les $(N-1)$ autres clusters. La RPC exécute une action très simple (incrémentation d'un entier), puisqu'on s'intéresse uniquement au coût des communications.

Puisque les RPC sont synchrones, ces $(N-1)$ RPC sont exécutées de façon séquentielle : la RPC $(i+1)$ ne démarre qu'après réception de la réponse à la RPC (i). L'instance du noyau s'exécutant sur le cluster **0** relève donc deux dates : une première date avant l'envoi de la première RPC. Une seconde date après la réception de la dernière réponse. La durée moyenne d'une RPC est calculée comme la différence entre ces deux dates divisées par $(N-1)$.

La figure 6.1 présente le résultat de cette expérimentation en fonction du nombre de clusters. On peut voir que ce coût ne croît que très faiblement entre 4 et 256 coeurs, et reste compris entre 3200 et 3800 cycles.

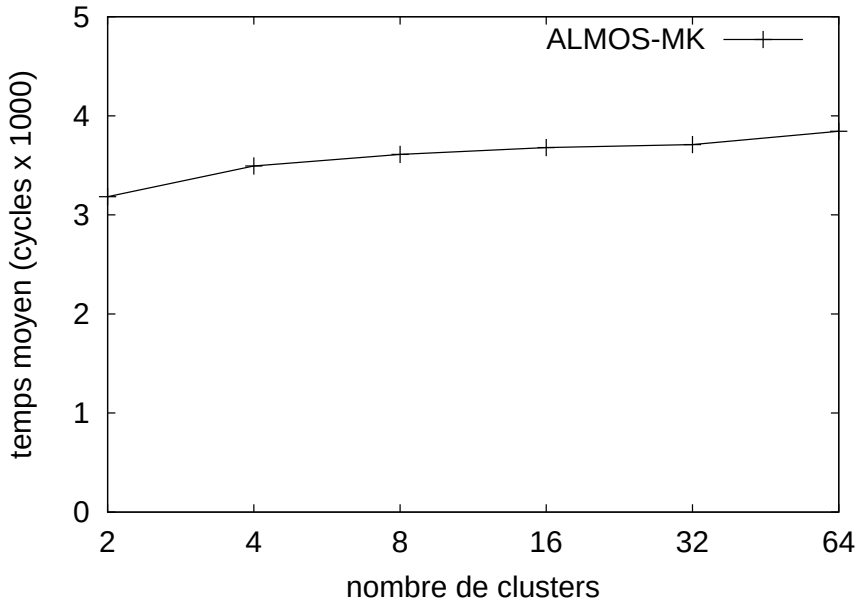


FIGURE 6.1 : Coût moyen des RPC.

Coût de la création distante de processus

La deuxième expérimentation consiste à évaluer le temps de création d'un processus. Plus précisément, le temps nécessaire à l'instance du cluster **0** de créer un processus sur le cluster $(N-1)$, c'est-à-dire le cluster le plus éloigné.

Les mesures sont effectuées à la fois sur ALMOS-MK, et sur la version initiale d'ALMOS avec support de l'adressage à 40-bits, que nous appelons ALMOS-40. ALMOS-40 dispose de deux configurations pour créer une tâche à distance. La première utilise l'accès direct en mémoire partagée, et nous appelons cette configuration ALMOS-40-SH. La seconde utilise le passage de message et le mécanisme des évènements (voir 2.2.1), et nous appelons cette configuration ALMOS-40-EV. Les résultats d'ALMOS-MK sont comparés à ces deux configurations.

La figure 6.2 montre les résultats de cette expérimentation en fonctions du nombre de coeurs. On peut voir que les résultats d'ALMOS-MK varient peu avec le nombre de coeurs, et sont légèrement meilleurs que ceux de ALMOS-40 pour 128 et 256 coeurs.

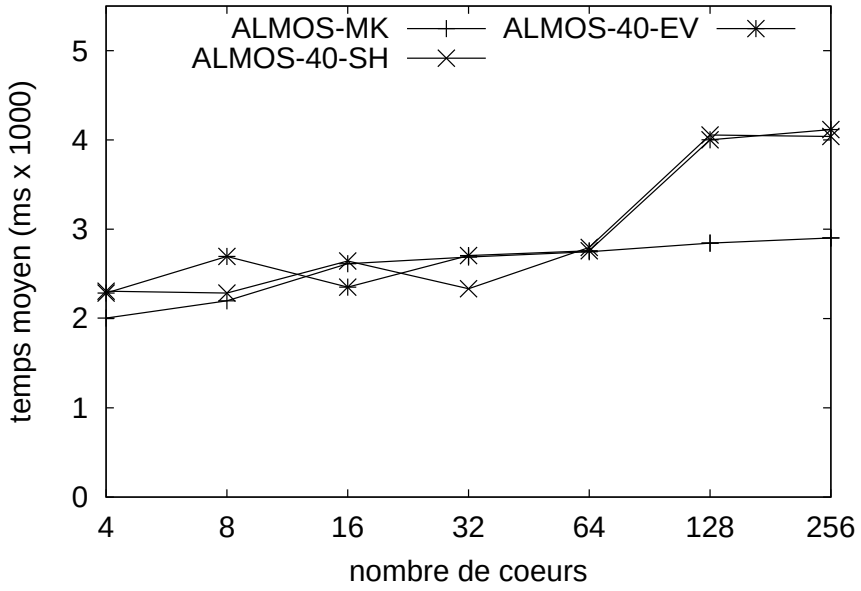


FIGURE 6.2 : Coût de la création distante d'un processus.

En conclusion

Cette dernière expérimentation montre que, même si le coût des RPC est important, les performances d'ALMOS-MK se comparent favorablement à la version initiale d'ALMOS. En effet, les pertes de performance causées par l'utilisation des RPC sont compensées par le fait que tous les accès sont locaux et qu'on a supprimé les coûts liés aux MISS TLB, puisque le noyau n'utilise plus la mémoire virtuelle. En effet, environ $1/3$ des MISS TLB sont évités par ALMOS-MK, tel que le montre la figure 6.3. Sur cette figure, les MISS TLB survenant avec ALMOS-MK sont exclusivement dus aux processus utilisateurs. Avec les autres systèmes, les MISS TLB sont dus à l'utilisateur et au système lorsqu'il est sollicité lors des appels système et périodiquement.

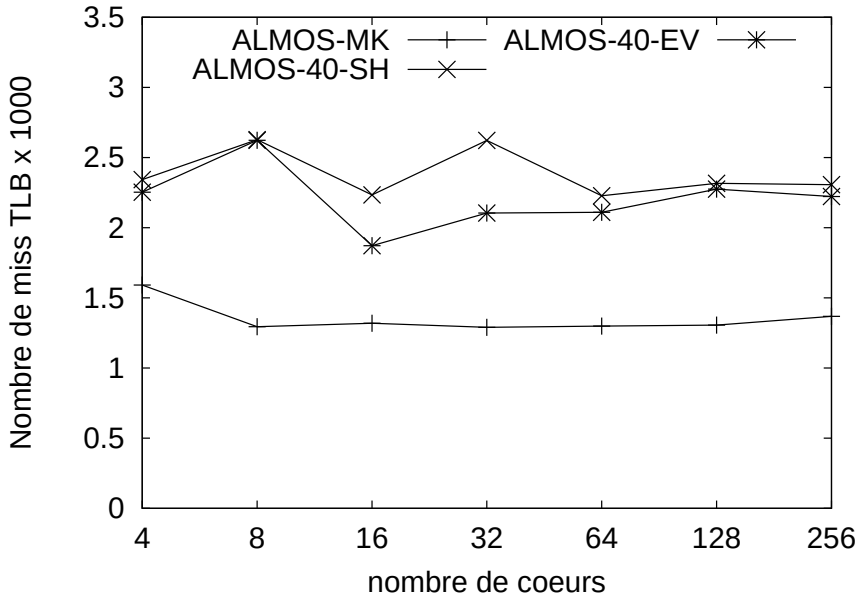


FIGURE 6.3 : Somme des miss TLB des coeurs des clusters 0 et (N-1) lors de la creation distante d'un nouveau processus.

En estimant qu'un accès distant est de l'ordre de 110 cycles (coût moyen d'une incrémentation distante sur ALMOS-40-SH), il suffit d'environ $3800/110 = 34$ accès mémoire distants pour compenser le coût des RPC, et beaucoup de services distants nécessitent nettement plus de 34 accès mémoire.

6.4.2 Scalabilité du système de fichiers

Dans cette seconde partie, nous comparons les performances d'ALMOS-MK avec celle de NetBSD. Deux expérimentations sont effectuées :

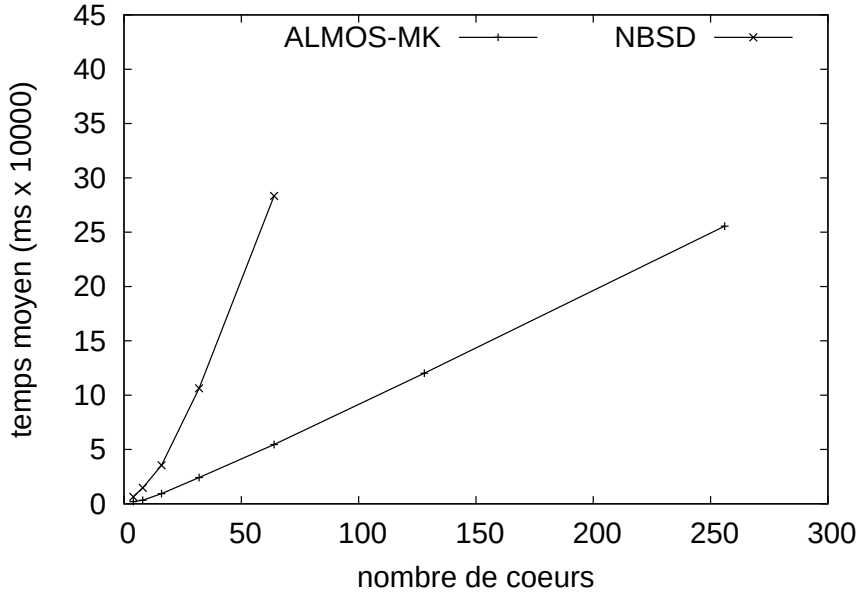


FIGURE 6.4 : Coût de l'accès en parallèle à un même fichier.

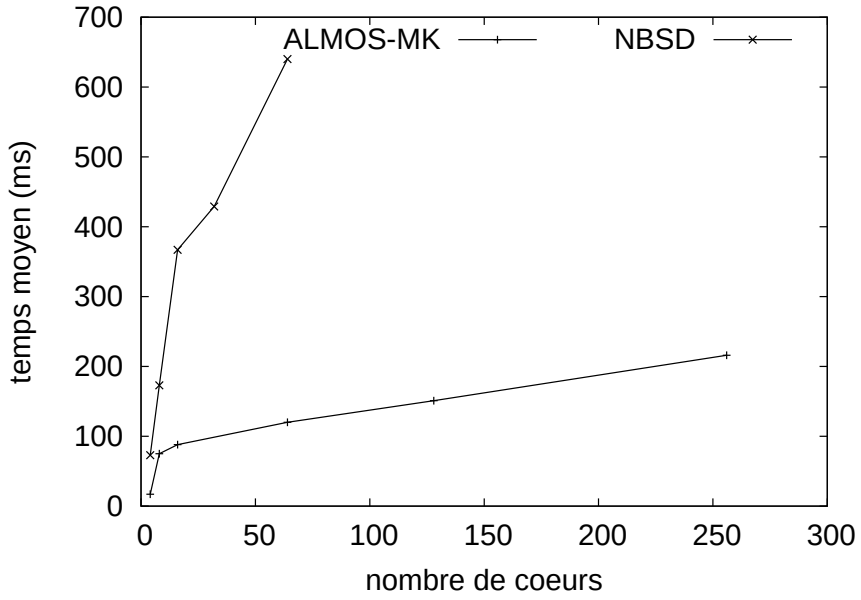


FIGURE 6.5 : Coût de l'accès en parallèle à plusieurs fichiers non partagés.

1. Dans la première expérimentation, on mesure les temps d'exécution lorsqu'un même fichier partagé est accédé en parallèle. Pour cela, on utilise le benchmark *Parallel read*. Les résultats sont présentés par la figure 6.4.

2. Dans la seconde expérimentation, on mesure les temps d'exécution lorsque des fichiers différents son accédés en parallèle. Pour cela, on utilise le benchmark *Parallel Postmark*. Les résultats sont présentés par la figure 6.5.

Les résultats d'ALMOS-MK sont systématiquement meilleurs que ceux de NetBSD. Les temps d'exécution d'AlMOS-MK croissent de façon linéaire avec le nombre de coeurs, alors que l'augmentation est beaucoup plus brutale avec NetBSD, et que la contention devient insupportable pour un nombre de coeurs supérieur à 64 : les benchmarks ne finissent pas dans un temps raisonnable.

La contention dans le noyau NetBSD est due non seulement au système de fichiers (certaines données et verrous sont partagés entre les clusters), mais elle a aussi d'autres causes. Parmi celle-ci, un des points de contention identifié est l'accès au code noyau. En effet dans NetBSD, il n'y a pas de mécanisme permettant de répliquer le code noyau, et le code stocké dans le cluster zéro est partagé par tous les coeurs du système.

6.4.3 Impact du placement des fichiers

Les figures ci-dessous présentent les résultats pour les deux stratégies de placement implémentées dans *ALMOS-MK*. La courbe ALMOS-MK-UNI correspond aux résultats obtenus avec la stratégie de distribution uniforme des fichiers sur tous les clusters. La courbe notée ALMOS-MK-LOC présente les résultats obtenus avec la stratégie de placement locale.

Les deux figures correspondent à deux types de benchmarks différents :

1. Dans la figure 6.6 les processus exécutent le benchmark *Parallel postmark*, et accèdent en parallèle à des fichiers différents.
2. Dans la figure 6.7 les processus exécutent le benchmark *Parallel RO-postmark*, et accèdent en parallèle à des fichiers partagés.

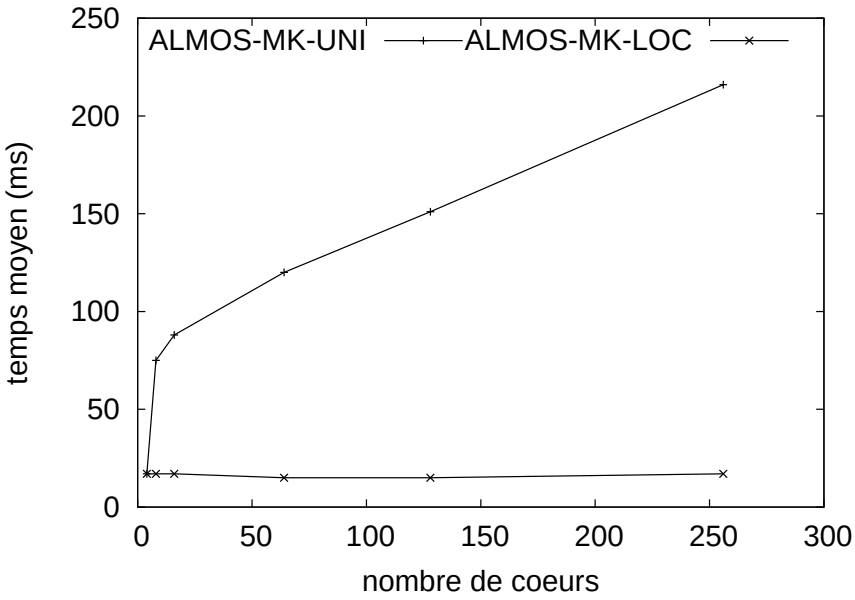


FIGURE 6.6 : Coût de l'accès en parallèle à plusieurs fichiers non partagés.

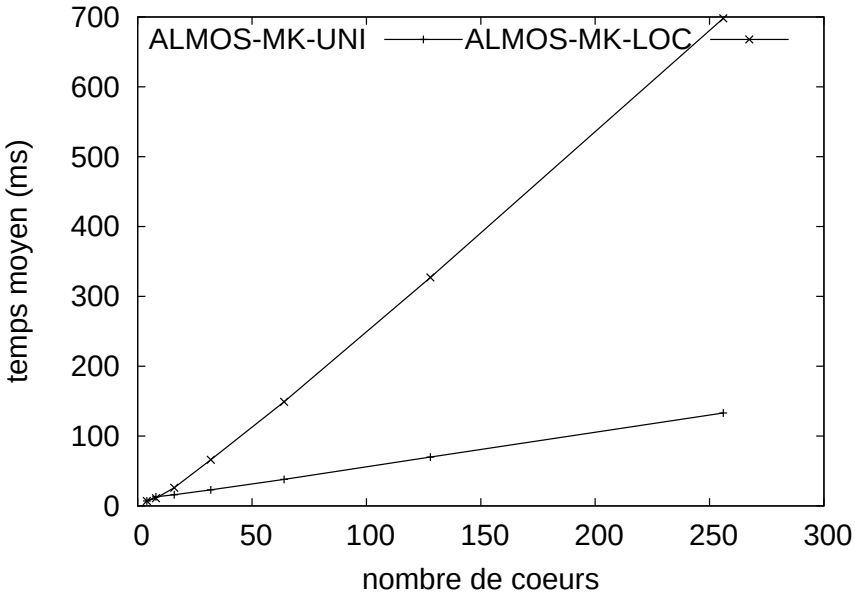


FIGURE 6.7 : Coût de l'accès en parallèle à plusieurs fichiers partagés.

Comme on pouvait s'y attendre, en l'absence de partage, la stratégie de placement locale est préférable, puisqu'elle est parfaitement scalable entre 4 et 256 coeurs.

En revanche, dans le cas d'une application parallèle multi-threads coopérative, où les différents threads accèdent aux mêmes fichiers, la stratégie de placement distribuée l'emporte nettement sur la stratégie locale en termes de scalabilité.

De ces deux dernières expériences, nous concluons que la stratégie de placement distribuée permet, dans le cas général, une meilleure scalabilité et c'est celle qui a été retenue comme stratégie par défaut pour ALMOS-MK.

6.5 Conclusion

Des résultats obtenus dans ce chapitre, nous pouvons tirer trois conclusions.

La première est que le passage à une structure multi-instances ne dégrade pas les performances par rapport à la structure mono-instance précédente. La communication par RPC l'emporte sur la communication par accès direct en mémoire distante dès que le service demandé nécessite plus d'une trentaine d'accès mémoire distants.

La deuxième est que le noyau multi-kernel ALMOS-MK est plus scalable que les noyaux UNIX traditionnels tels que NetBSD, sur les benchmarks qui sollicitent fortement le système de fichiers. En effet, pour les différents benchmarks exécutés, les temps d'exécution sur ALMOS-MK croissent linéairement entre 4 et 256 coeurs, alors que NetBSD ne permet pas de dépasser 64 coeurs.

La troisième est que la stratégie de placement uniforme permet, dans le cas général, d'assurer une meilleure scalabilité que la stratégie de placement locale pour les applications parallèles coopératives accédant à des fichiers partagés.

Notons cependant que les expérimentations menées restent limitées à des benchmarks synthétiques, et que l'utilisation d'applications parallèles réelles est nécessaire pour confirmer les conclusions ci-dessus.

CHAPITRE 7

Conclusion

Dans cette thèse nous avons essayé de répondre à trois questions portant sur le passage à l'échelle des performances du système de fichiers des systèmes d'exploitation pour architectures manycore à espace d'adressage partagé de type CC-NUMA. Nous avons pris comme référence l'architecture manycore TSAR, et comme base d'expérimentation le système d'exploitation de type UNIX ALMOS.

Capacité d'adressage du noyau

La première question concerne l'extensibilité des caches du système de fichiers, et pose plus largement le problème de l'accès à l'espace adressable 40 bits de l'architecture TSAR lorsque l'on utilise des processeurs 32 bits. Notre solution consiste à modifier la structure du noyau ALMOS pour évoluer vers une structure multi-instances, chacune des instances gérant un cluster de l'architecture. De plus, chaque instance s'exécute directement en adressage physique. Les instances communiquent entre elles grâce à un mécanisme de type RPC.

Afin d'évaluer l'impact de cette restructuration sur les performances, nous avons comparé expérimentalement cette nouvelle structure avec la structure mono-instance. L'expérimentation consiste à calculer le temps nécessaire pour faire migrer un processus entre les deux clusters les plus éloignés de l'architecture. Cette expérimentation indique que la structure multi-instances n'affecte pas les performances et peut même les améliorer.

Nous avons montré que le coût en performance du mécanisme RPC ne croit que très faiblement entre 4 et 256 coeurs, et reste compris entre 3200 et 3800 cycles. Même si ce coût peut sembler important, les performances d'ALMOS-MK se comparent favorablement à la version initiale d'ALMOS. En effet, les pertes de performance causées par l'utilisation des RPC sont compensées par le fait que tous les accès sont locaux et que le coût lié à la résolution des miss TLB est supprimé, puisque le noyau n'utilise plus la mémoire virtuelle.

Nous estimons donc que le mécanisme de communication par RPC l'emporte sur un mécanisme d'accès direct en mémoire distante dès que le service distant nécessite plus d'une trentaine d'accès mémoire.

En plus de permettre l'accès à un espace physique étendu, la nouvelle structure favorise la mise en oeuvre de stratégies scalables pour les différents services rendus par le système d'exploitation. En effet, une structure multi-instances fait que le développeur du système est conscient de tous les accès distants, car il doit explicitement définir tous les accès distants. Ceci n'est pas le cas dans une structure mono-instance, où les accès distants sont implicites et peuvent donc passer inaperçus.

Placement des structures du système de fichiers

Le deuxième problème concerne le choix d'une stratégie de placement des données du système de fichiers. Pour ce problème, il existe deux principales solutions. La première consiste à effectuer un placement local, c'est-à-dire dans le banc de mémoire du cluster où s'exécute le processus demandeur. Cette stratégie est aussi appelée *first-touch*. La seconde consiste à distribuer uniformément les données sur l'ensemble des bancs de mémoire, aussi appelée stratégie *interleave*. Malgré le fait que la majorité des systèmes existants utilisent par défaut la première, nous avons choisi la deuxième pour le noyau ALMOS-MK. Ce choix est justifié par le fait que dans un système visant principalement une architecture manycore, il est bien plus important d'éviter la contention que de favoriser la localité des accès.

Ces deux stratégies de placement ont été comparées expérimentalement. Cette comparaison a été effectuée sur deux types de benchmarks. Dans le premier, un ensemble de processus effectuent des opérations sur un sous-ensemble de fichiers non partagés. Dans le second, les processus effectuent des opérations sur un ensemble de fichiers partagés. Les résultats de cette expérimentation indiquent que, dans le cas général, la stratégie de distribution uniforme permet d'obtenir de meilleures performances.

Par ailleurs, on note que l'intérêt de la stratégie de distribution uniforme n'apparaît qu'à partir d'un certain nombre de coeurs. Ceci justifie le choix des systèmes d'exploitation commerciaux d'utiliser une stratégie de placement local. Toutefois, nous pensons que le passage à une architecture manycore requiert un changement de la stratégie par défaut.

Synchronisation des accès concurrents

Le troisième problème concerne la synchronisation des accès concurrents. Notre solution pour ce problème utilise un ensemble de mécanismes : verrous simples, verrous multi-lecteurs, compteurs de références ou compteurs de générations. L'intérêt de cet ensemble de mécanismes est principalement de localiser les prises de verrous à l'intérieur d'une seule instance du noyau. Ceci permet d'éviter les prises de verrous inter-instances, lesquelles peuvent non seulement provoquer de la contention, mais aussi augmenter le risque d'interblocage.

La scalabilité des performances de notre solution a été comparée expérimentalement à celle du noyau UNIX NetBSD. Les résultats obtenus indiquent une bien meilleure scalabilité des performances de ALMOS-MK, par rapport à celles de NetBSD, puisque les temps d'exécution des différents benchmarks augmentent linéairement entre 4 et 256 coeurs sur ALMOS-MK, alors qu'ils augmentent beaucoup plus rapidement avec NetBSD, qui

ne supporte pas plus de 64 coeurs. Ces bons résultats ne sont pas seulement liés aux mécanismes de synchronisation utilisés, mais sont fondamentalement liés à l'approche multi-instances, et au placement distribué qui ont permis d'éliminer pratiquement tous les points de contention dans les accès au système de fichiers.

Perspectives

La refonte du noyau d'ALMOS vers une architecture multi-instances ALMOS-MK nous a demandé un temps que nous avions sous-estimé au départ et nous a empêchés d'aller jusqu'au bout de l'implémentation du système. Toutefois, nous sommes convaincus que la direction multi-instances est la bonne et les chiffres des premières expérimentations renforcent notre conviction.

Un des services manquant à cette implémentation est l'exécution d'un processus multi-threads s'étendant sur plusieurs clusters. Une étude a été entamée au cours de la dernière année de la thèse, mais elle n'a pas pu être intégrée. Ce devra être la première étape à effectuer. Pour implémenter ce service, il faudra le repenser pour séparer les responsabilités par cluster. Pour cela, il faudra répliquer une partie des structures décrivant les processus, telles que les tables de pages. Une réplication de la table des pages a été proposée par Ghassan Almaless avec le concept de processus hybrides. Nous allons devoir repenser la gestion de la cohérence de ces tables dans un contexte client-serveur.

Dissémination des sources et publications

Le code source du noyau ALMOS-MK est disponible sur le site web du projet [1]. Ce code dispose d'une licence *open-source* : *GNU General Public License*.

Dans le cadre de la thèse, nous avons eu l'occasion de faire deux publications. La première pour la conférence française COMPAS-2014 [45]. Cette publication contient une description du mécanisme de synchronisation à base de compteurs de génération dans le cas d'un environnement à mémoire partagée. Cette publication a été suivie par la publication d'un article dans la revue TSI [29].

Bibliographie

[1] ALMOS-MK. `https://www-soc.lip6.fr/trac/almos-mk` [Accessed 17-06-2016].

[2] Advanced Micro Devices. AMD64 Architecture Programmer's Manual, Volume II : System Programming. Publication Number : 24593, Revision : 3.23, May 2013.

[3] Advanced RISC Machine. Principles of ARM® Memory Maps. 2012.

[4] Anant Agarwal and Mathews Cherian. *Adaptive backoff synchronization techniques*, volume 17. ACM, 1989.

[5] Ghassan Almaless. *Conception d'un système d'exploitation pour une architecture many-cores à mémoire partagée cohérente de type cc-NUMA*. PhD thesis, Université Pierre et Marie Curie (UPMC), 2014. Type : Thèse de Doctorat – Soutenue le : 2014-02-27 – Dirigée par : Greiner, Alain – Encadrée par : WAJSBÜRT Franck.

[6] Ramesh Balan and Kurt Gollhardt. A scalable implementation of virtual memory hat layer for shared memory multiprocessor machines. In *USENIX Summer*, 1992.

[7] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn : a replicatedkernel os based on linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel : a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[9] Andrew Baumann, Chris Hawblitzel, Kornilios Kourtis, Tim Harris, and Timothy Roscoe. Cosh : clear os data sharing in an incoherent world. In *Proceedings of the 2014 Conference on Timely Results in Operating Systems (TRIOS), Broomfield, CO*, 2014.

[10] Nathan Z Beckmann, Charles Gruenwald III, Christopher R Johnson, Harshad Kasture, Filippo Sironi, Anant Agarwal, M Frans Kaashoek, and Nickolai Zeldovich. Pika : A network service for multikernel operating systems. 2014.

[11] Martin J Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on numa systems. In *Proceedings of the Linux Symposium*, volume 1, pages 89–102, 2004.

[12] W Bolosky, R Fitzgerald, and M Scott. Simple but effective techniques for numa memory management. In *ACM SIGOPS Operating Systems Review*, volume 23, pages 19–31. ACM, 1989.

[13] Shekhar Borkar. Thousand core chips : a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey : An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[15] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, pages 86–93, 2010.

[16] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010.

[17] Ray Bryant, J Hawkes, J Steiner, J Barnes, and J Higdon. Scaling linux to the extreme. In *Proceedings of the Linux Symposium*, pages 133–148, 2004.

[18] Ray Bryant and John Hawkes. Linux scalability for large numa systems. In *Linux Symposium*, page 76, 2003.

[19] Henry HY Chang and Bryan Rosenburg. Experience porting mach to the rp3 large-scale shared-memory multiprocessor. *Future Generation Computer Systems*, 7(2) :259–267, 1992.

[20] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. *Hive : Fault containment for shared-memory multiprocessors*, volume 29. ACM, 1995.

[21] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management : a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4) :381–394, 2013.

[22] David Dice, Danny Hendler, and Ilya Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Euro-Par 2013 Parallel Processing*, pages 595–606. Springer, 2013.

[23] Steven Frank, Henry Burkhardt III, and James Rothnie. The ksr 1 : bridging the gap between shared memory and mpps. In *Compcon Spring'93, Digest of Papers.*, pages 285–294. IEEE, 1993.

[24] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado : Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, volume 99, pages 87–100, 1999.

[25] Charles Gruenwald III. *Providing a Shared File System in the Hare POSIX Multikernel*. PhD thesis, Massachusetts Institute of Technology, 2014.

[26] Charles Gruenwald III, Filippo Sironi, M Frans Kaashoek, and Nickolai Zeldovich. Hare : a file system for non-cache-coherent multicores. In *Proceedings of the Tenth European Conference on Computer Systems*, page 30. ACM, 2015.

[27] Hewlett-Packard. Red Hat Linux NUMA Support for HP ProLiant Servers. `http://h50146.www5.hp.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/c03261871.pdf`, 5900-3260, Septembre 2013, Rev 2.

[28] Jeffrey M Hsu. The dragonflybsd operating system. *Proceedings USENIX AsiaBSDCon, Taipei, Taiwan*, 2004.

[29] Mohamed Lamine Karaoui, Quentin L. Meunier, Franck Wajsbürt, and Alain Greiner. Gecos : Mécanisme de synchronisation passant à l'échelle à plusieurs lecteurs et un écrivain pour structures chaînées. *Technique et Science Informatiques*, 34 :53–78, May 2015.

[30] Jeffrey Katcher. Postmark : A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

[31] Kevin Klues, Barret Rhoden, Y Zhu, Andrew Waterman, and Eric Brewer. Processes and resource management in a scalable many-core os. *HotPar10, Berkeley, CA*, 2010.

[32] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. K42 : building a complete operating system. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 133–145. ACM, 2006.

[33] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable rea, der-writer lock. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 201–204. IEEE, 1993.

[34] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, et al. The stanford flash multiprocessor. In *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 302–313. IEEE, 1994.

[35] Christoph Lameter. Effective synchronization on linux/numa systems. In *Gelato Conference*, 2005.

[36] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S Lam. The stanford dash multiprocessor. *Computer*, 25(3) :63–79, 1992.

[37] Linux Kernel. Path walking and name lookup locking. https://www.kernel.org/doc/Documentation/filesystems/path-lookup.txt.

[38] Linux Weekly News. Linus Torvalds's dcache patch. http://lwn.net/Articles/510313/.

[39] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanovic, and John Kubiatowicz. Tessellation : Space-time partitioning in a manycore client os. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 10–10, 2009.

[40] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001. Available : http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.pdf [Accessed 17-06-2016].

[41] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[42] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1) :21–65, 1991.

[43] John M Mellor-Crummey and Michael L Scott. Synchronization without contention. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 269–278. ACM, 1991.

[44] Maged M Michael. Hazard pointers : Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6) :491–504, 2004.

[45] Karaoui Mohamed Lamine, Quentin L. Meunier, Franck Wajsbürt, and Alain Greiner. Mécanisme de synchronisation scalable à plusieurs lecteurs et un écrivain. In *Conférence en Parallélisme, Architecture et Systèmes, ComPAS 2014*, Neuchâtel, Switzerland, April 2014.

[46] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios : heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[47] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. Nfs version 3 : Design and implementation. In *USENIX Summer*, pages 137–152. Boston, MA, 1994.

[48] Pierre-Yves PÉNEAU, Franck WAJSBÜRT, Mohamed KARAOUI, and Béatrice BÉRARD. Migration de processus pour un multi-noyau large échelle. 2015.

[49] Nick Piggin. A lockless page cache in linux. In *Proceedings of the Linux Symposium*, volume 2, 2006.

[50] Michel Raynal. *Concurrent Programming : Algorithms, Principles, and Foundations*. Springer, 2013.

[51] Mark E Russinovich, David A Solomon, and Jim Allchin. *Microsoft Windows Internals : Microsoft Windows Server 2003, Windows XP, and Windows 2000*, volume 4. Microsoft Press Redmond, 2005.

[52] Jan Sacha, Henning Schild, Jeff Napper, Noah Evans, and Sape Mullender. Message passing and scheduling in osprey.

[53] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem, 1985.

[54] Shahrokh Shahidzadeh, Bryant E Bigbee, David B Papworth, Frank Binns, and Robert P Colwell. Linear address extension and mapping to physical memory using 4 and 8 byte page table entries in a 32-bit microprocessor, February 19 2002. US Patent 6,349,380.

[55] Benjamin H Shelton. *Popcorn Linux : enabling efficient inter-core communication in a Linux-based multikernel operating system*. PhD thesis, Virginia Polytechnic Institute and State University, 2013.

[56] Ronald C Unrau. *Scalable memory management through hierarchical symmetric multiprocessing*. PhD thesis, Citeseer, 1993.

[57] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a numa multiprocessor operating system kernel. In *IN OSDI SYMPOSIUM*, pages 139–152, 1994.

[58] Ronald C Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering : A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1-2) :105–134, 1995.

[59] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[60] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. *Operating system support for improving data locality on CC-NUMA compute servers*, volume 31. ACM, 1996.

[61] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector-a hierarchically structured shared memory multiprocessor. In *System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on*, volume 1, pages 444–453. IEEE, 1991.

[62] David Wentzlaff and Anant Agarwal. Factored operating systems (fos) : the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2) :76–85, 2009.

[63] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs : Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.

[64] Zhou Yingchao, Meng Dan, and Ma Jie. Dual-layered file cache on cc-numa system. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.