Conception et réalisation d'une mémoire partagée répartie

Jay Han

To cite this version:

Jay Han. Conception et réalisation d'une mémoire partagée répartie. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 1996. Français. NNT : . tel-00004993

HAL Id: tel-00004993

https://theses.hal.science/tel-00004993

Submitted on 23 Feb 2004

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

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

THÈSE

présentée par

Jay HAN

Pour obtenir le grade de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 mars 1992)

Spécialité : INFORMATIQUE

Conception et réalisation d'une mémoire partagée répartie

Date de soutenance : le 25 novembre 1996

Composition du jury

M. Guy Mazaré	Président
Mme. Christine Collet	Examinateurs
M. Pascal Dechamboux	
M. Xavier Rousset de Pina	Directeur
M. André Schiper	Rapporteurs
M. Jean Seguin	

Remerciements

Cette thèse s'est déroulée sous la direction de Xavier Rousset de Pina. Sans ses conseils et ses suggestions parfois véhémentes, ce travail n'aurait pas pu aboutir.

Je tiens à remercier les membres du jury, M. Mazaré, Mme. Collet, Pascal, ainsi que les rapporteurs, MM. Schiper et Seguin, pour l'attention qu'ils ont bien voulu accorder à l'évaluation de mon travail.

Le Centre National d'Études en Télécommunications et l'Université Joseph Fourier m'ont soutenu financièrement durant la préparation de cette thèse. Qu'ils en soient ici remerciés.

Enfin, last but not least, je tiens à exprimer mes sentiments chaleureux envers mes compagnons de travail. Adriana, Agnès, Alain, André, Andrzej, Béatrice, Chérif, Christian, Christian, Christine, Claudia, Cécile, Dan, Daniel, Daniel, Dominique, Elizabeth, Emmanuel, Fabienne, Fernando, Frédéric, Irène, Jacques, Jacques, Jean-Philippe, Jean-Yves, Joëlle, Leïla, Loay, Luc, Manuel, Marc, Mauricio, Michael, Michel, Miguel, Nabil, Olivier, Olivier, Philippe, Rémy, Roland, Rushed, Sacha, Serge, Slim, Stéphane, Thierry, Vincent, Vincent, Vladimir, Youssef, merci.

Bobo, Momo et Catherine, coucou !

Introduction

On a généralement coutume d'opposer deux grands modèles de multiprogrammation : le modèle client/serveur et le modèle de mémoire partagée. Les avantages du modèle client/serveur sont principalement la protection et la modularité. La protection est assurée par le fait que le serveur ne communique qu'à travers une interface prédéfinie et aussi sécurisée que nécessaire. Réciproquement, dès lors qu'un client se conforme à cette interface, il peut accéder au serveur. Ce modèle n'est pas suffisant pour certaines applications. En particulier, l'échange de données structurées nécessite une interface complexe dont la mise en œuvre est lourde. Par ailleurs, le modèle n'est pas adapté pour exprimer des modes de coopération qui dépassent la simple relation client/serveur.

Le modèle de mémoire partagée possède des caractéristiques inverses. Il permet un échange immédiat de données de structure complexe, sans spécifier s'il doit exister un client et un serveur. C'est un modèle qui, intuitivement, se rapproche le plus de la simple programmation mono-processus. Mais tout processus ayant accès à la mémoire partagée peut la manipuler à sa guise, ce qui signifie que l'on perd la protection et la modularité du modèle client/serveur. En effet, il est difficile de préciser à un grain fin les droits d'accès des processus aux données, et les processus doivent s'accorder sur le format exact des données en mémoire partagée. De plus, les processus doivent synchroniser leurs accès aux données pour préserver leur cohérence. Cela ne signifie pas que la mémoire partagée n'a pas d'utilité pratique, mais que, dans la conception d'une application, il faut trouver le bon compromis entre les échanges via des messages et les échanges via la mémoire partagée.

Lorsque l'on étend ces notions de multiprogrammation aux systèmes répartis, on constate que le modèle client/serveur est plus facilement adaptable. Un système réparti est constitué d'un ensemble de machines reliées par un réseau, dont l'interface d'accès fondamental est l'envoi de messages. Il est donc simple de transposer le modèle client/serveur sur ce mécanisme de base. Le cas du RPC [8] montre combien cette adéquation est forte : ce mécanisme de communication, d'abord inventé dans le cadre des systèmes répartis pour en simplifier la programmation, a ensuite été réintégré dans le cadre des systèmes parallèles (LRPC [6]) et enfin dans les systèmes à micro-noyau (RPC Chorus [70]).

Qu'en est-il de la mémoire partagée ? Dans un environnement réparti, une mémoire partagée répartie (MPR) nécessite une infrastructure autrement plus lourde qu'en multiprogrammation sur une machine unique. Lorsqu'un processus sur une machine écrit une

donnée en MPR, il faut qu'à un instant ultérieur cette écriture soit répercutée à l'intention d'un processus s'exécutant sur une autre machine (on dit alors que la mémoire est *cohérente*). Savoir identifier les modifications des données et déterminer à quel moment les répercuter nécessite soit de faire des hypothèses sur le comportement des applications, soit d'obtenir de l'aide de celles-ci. Dans ce dernier cas, il faut que les applications soient modifiées en conséquence, par exemple en imposant une interface de notification. Pour éviter cela, les réalisations existantes ont généralement préféré adopter le premier choix, en garantissant un certain type de comportement cohérent au prix d'une certaine inefficacité.

Parallèlement à la recherche en systèmes répartis, la recherche en architectures multiprocesseurs à bus de message a aussi fourni de beaux résultats en algorithmique distribuée. Ce type d'architecture parallèle comporte plusieurs processeurs et plusieurs modules de mémoire reliés entre eux par un système de communication à message, et non par un bus d'accès mémoire, ce qui le rend similaire à un système réparti. Cependant, alors que ces architectures parallèles sont dotées d'un système de communication spécialisé et rapide, un système réparti doit se contenter d'une connectique du type d'un réseau local, dont la bande passante et la latence sont bien moins performantes. L'objectif est alors de réduire au minimum le nombre et la taille des messages qu'il est nécessaire d'échanger pour préserver la cohérence de la mémoire. Pour cela, divers systèmes faisant appel à la coopération de l'application ont été inventés : annotation de variables (Munin [9]), cohérence relâchée (Midway [7]), etc.

Une difficulté supplémentaire survient lorsque l'on veut rendre persistante et uniforme la mémoire répartie. L'utilité d'une mémoire persistante a été largement démontrée depuis MULTICS [20]. Une mémoire partagée, répartie et persistante répond à des besoins très variés, allant des applications interactives du type éditeur de document jusqu'aux bases de données réparties. Ajoutons à cela la notion d'uniformité de la désignation : une donnée se trouve toujours à la même adresse virtuelle et l'unique identifiant d'une donnée est justement son adresse, la relation adresse virtuelle $\rightarrow$ site de stockage étant prise en charge par le système. Cette définition de l'uniformité va plus loin que celle de MULTICS, par exemple, où l'architecture segmentée impose un adressage à deux niveaux, par segment et par déplacement. Mais une mémoire entièrement uniforme est limitée par l'espace virtuel disponible. Les besoins des applications modernes telles que la CAO (Boeing) et l'indexation de documents (Alta Vista [24]) nécessitent de plus en plus des quantités de données qui dépassent la capacité d'adressage des processeurs 32 bits.

À ces deux problèmes, la taille de l'espace d'adressage et la vitesse du support de communication, l'évolution technologique récente apporte des réponses. Les processeurs les plus récents ont un espace d'adressage agrandi, supérieur aux 32 bits de la génération précédente, atteignant 64 bits. Un espace virtuel de 64 bits est énorme : on a calculé [14] qu'un tel espace, consommé à la cadence de 1 Go par seconde, ne serait épuisé qu'au bout de 500 ans. Même utilisé par un ensemble de machines, il est raisonnable de penser qu'il sera assez grand pour adresser la totalité des données persistantes utilisées par cet ensemble durant toute la vie du système, et ce sans jamais réutiliser d'adresse.

Ensuite, de nouveaux supports de réseau local rapide font leur apparition : Ethernet-100, FDDI ou ATM promettent des bandes passantes de 100 à 500 mégabits par seconde, avec aussi une diminution importante de la latence. De plus, ces réseaux rapides vont de pair avec des interfaces matérielles intelligentes qui sont capables de recevoir un message et le traiter à la volée, sans nécessiter de traitement d'interruption ou de changement de contexte. Bien qu'il ne dispense pas d'être économe en communication, ce progrès brise la tendance antérieure qui allait vers un déséquilibre entre la vitesse du processeur, qui ne cessait de croître, et celle du réseau de connexion, qui stagnait au même niveau.

Ces solutions techniques font de la MPR un modèle de plus en plus attrayant, et l'on peut espérer faire disparaître l'idée largement répandue que l'utilisation d'une MPR implique une perte de performance. Cette idée a été un frein important dans l'introduction des MPR dans les milieux industriels, et est due aux médiocres performances des premiers prototypes très peu optimisés. Comme le souligne Carter [10], la quasi-absence de systèmes commerciaux entretient l'ignorance de solutions nouvelles et efficaces. Une exception notable est TreadMarks [2], un produit commercialisé pour des tâches de production réelle et soutenu par une entreprise qui en assure le support technique. Bien qu'efficace, TreadMarks est un système relativement limité, qui vise principalement les applications de calcul intensif. Mais cet exemple montre que pour avoir une large audience, une MPR doit s'intégrer aux environnements existants.

Une MPR ne prétend pas apporter d'améliorations de performance importantes par rapport au modèle de programmation par échange de messages. Son avantage principal est qu'elle constitue un modèle de programmation beaucoup plus simple à utiliser, et particulièrement adapté au partage de données de structures complexes. Paralléliser une application est une tâche complexe, qui est grandement facilitée par la présence d'un modèle uniforme d'accès aux données, en l'occurrence l'accès direct à la mémoire virtuelle.

Ces considérations font de la réalisation des MPR un sujet d'étude encore très ouvert. En particulier, une MPR persistante et uniforme sur un vaste espace d'adressage virtuel, bien intégrée à un système existant, en l'occurrence AIX [55], compatible avec les environnements standards, et adaptable à une large palette de besoins applicatifs allant des langages à objets aux bases de données, représente un projet digne d'intérêt.

* * *

Les sujets d'étude que l'on peut trouver autour d'une telle MPR sont nombreux. Par exemple, l'intégration d'un système de stockage résistant aux pannes [42] ou la définition d'un mécanisme de protection inter-application [73] sont des sujets importants et intéressants. Le service de mémoire de base doit comporter certaines fonctions essentielles, généralement de trois ordres :

- l'allocation et la localisation des données partagées,
- la synchronisation et la mise en cohérence de ces données, et

- la mise à disposition de ces données à travers l'interface de mémoire virtuelle du système hôte.

Le second point a fait l'objet de nombreuses études antérieures, parmi lesquelles on peut citer des projets comme Munin [9] ou Midway [7]. La plupart de ces projets présentent de nouveaux modèles de cohérence de mémoire répartie ou des implémentations particulières d'un modèle existant. Notre objectif sur ce point est de laisser à chaque le choix du modèle qui est le plus adapté à ses besoins. Pour ce faire, nous avons défini une interface à deux niveaux, constituée d'une couche spécifique et d'une couche générique. La couche spécifique est composée de modules, chacun fournissant une interface de synchronisation vers l'application et gérant un protocole de cohérence utilisant la couche générique. Cette dernière comporte les fonctions indispensables à tous les protocoles : elle factorise les tâches communes et rébarbatives (communication, routage de message, recopie de mémoire, etc.). Arias gère la multiplicité de modules spécifiques, au gré des besoins des applications qui s'exécutent.

La couche générique, qui fait l'objet d'une autre thèse [65], repose sur la notion de zone. Par définition, la *zone* est simplement un ensemble contigu d'octets, sans limitation de taille, mais qui correspond à une unité d'accès et de cohérence. La gestion de la cohérence et de la synchronisation d'une zone dépend du modèle et du protocole qui lui sont associés. L'enjeu de la couche générique est de mémoriser cette association, c'est-à-dire de localiser, pour une zone donnée, le site qui est chargé de sa gestion, et que l'on appelle son *site maître* (similaire au site propriétaire d'Ivy [51]).

Idéalement, la MPR est entièrement constituée de zones, qui sont la seule entité manipulée. Cependant, l'étendue de l'espace d'adressage nous a conduit à effectuer des regroupements. De fait, nous avons défini une entité de grain intermédiaire, appelée segment. Une telle entité de grain intermédiaire est très utile pour l'architecture d'un service de mémoire. On peut noter que beaucoup de systèmes de MPR ont adopté des notions proches, comme les grappes de Guide [33] ou les ≪bunches≫ de Larchant [29].

Le chemin d'accès typique à la mémoire Arias est de passer par un module spécifique de synchronisation et de cohérence, qui met en œuvre un modèle de synchronisation et un protocole de cohérence. Ce module spécifique s'adresse à la couche générique de gestion des zones pour transférer les données d'une part, et d'autre part pour assurer l'acheminement des messages au site maître d'une zone donnée. Cette dernière opération nécessite l'identification de la zone et la recherche des informations qui lui correspondent.

Cette opération, que nous appelons *localisation de zone*, est découpée en deux étapes. La première consiste à retrouver le segment auquel appartient la zone. Parmi les métadonnées associées à un segment, se trouve son *descripteur*, qui lui-même contient les informations concernant l'ensemble des zones appartenant à ce segment. Une fois le segment identifié, la seconde étape consiste à parcourir son descripteur pour y trouver les informations concernant la zone précise demandée.

Ce découpage en deux étapes autour de l'entité segment peut sembler artificielle, mais il correspond à plusieurs contraintes. D'abord, le segment est l'unité d'allocation vue par

les applications. Ensuite, le segment intervient comme entité de grain élémentaire pour les services optionnels tels que la protection et la résistance aux pannes. Par exemple, le segment est l'unité de protection, c'est-à-dire qu'un droit d'accès est accordé soit à un segment entier, soit à aucune partie de celui-ci. Ainsi, le segment est une unité de gestion de la mémoire qui sert d'unité de manipulation à tous les services intégrés dans Arias.

Parmi les deux étapes de localisation, à savoir adresse $\rightarrow$ segment puis segment $\rightarrow$ zone, la question qui préoccupe cette thèse est la première. À partir d'une adresse, il s'agit d'identifier le segment auquel elle appartient, c'est-à-dire connaître l'adresse de début du segment et sa taille, et aussi de savoir où se trouve son descripteur.

La combinaison de deux caractéristiques, à savoir la grande taille de l'espace virtuel et la distribution clairsemée des segments dans cet espace, rend la recherche difficile. Car cette recherche doit être rapide, plus spécifiquement elle doit minimiser le nombre de messages réseau requis. De plus, la création de segment doit être efficace, car nous voulons minimiser le surcoût de la mémoire persistante relativement à la mémoire volatile.

Ce dernier point, qui n'a pas encore été soulevé, a eu un impact important sur la conception du service. En effet, la réalisation de mémoires persistantes s'est souvent appuyée sur des systèmes de stockage secondaire [71, 57]. Par conséquent, l'allocation de mémoire dans ce type de systèmes nécessite l'allocation d'un espace correspondant sur le support de mémoire secondaire, ce qui est généralement coûteux. Notre objectif est de minimiser ce surcoût, de manière à banaliser l'usage de la mémoire persistante. En particulier, la gestion de la persistance ne repose pas sur l'utilisation d'une mémoire secondaire lente[1]. En d'autres termes, on encourage le programmeur à allouer systématiquement toutes ses données dans l'espace persistant, ce qui garantit que ses données seront partageables si le besoin s'en fait sentir a posteriori.

Outre la localisation de zone, l'étape de localisation de segment sert aussi au moment du couplage de segment. En effet, Arias a besoin de coopérer avec le système de mémoire virtuelle du système hôte pour effectivement rendre la MPR accessibles aux applications. Pour cela, on utilise les mécanismes d'adressage offerts par AIX, qui nécessitent que l'on puisse fournir l'identification d'un segment à partir d'une adresse virtuelle quelconque lui appartenant.

En résumé, la mission du module qui nous intéresse est :

- d'allouer des segments dans l'espace virtuel,
- d'identifier le segment à partir d'une adresse virtuelle quelconque, et
- d'en localiser le descripteur.

* * *

[1] Sauf dans le cas des segments dits permanents, pour lesquels on offre des garanties supplémentaires au prix d'un surcoût en performance et d'une interface de journalisation [42].

Il est étonnant de constater que très peu de travaux ont été publiés sur le sujet de la localisation et de l'allocation dans un espace d'adressage réparti vaste et clairsemé. De fait, les projets actuels se heurtent tôt ou tard à ce problème, mais très peu de résultats ont été publiés [27, 12, 13]. Des idées originales, telles que les GPT (*Guarded Page Tables* [53]), adoptées par le projet Mungi [54], ne résolvent pas tous les problèmes que nous rencontrons avec des solutions classiques à base de tables de pages linéaires ou de tables de hachage.

Le module que nous devons concevoir doit en effet :

- avoir un faible coût en mémoire, c'est-à-dire avoir une structure compacte,
- effectuer une localisation et une allocation rapides,
- permettre la migration des méta-données au gré des exigences de localité, et
- laisser la possibilité d'une reconfiguration de réseau (c-à-d. permettre l'ajout ou le retrait d'une machine).

Les études récentes sur les réseaux rapides [83] font apparaître qu'un échange de messages à travers le réseau coûte encore cher par rapport à un traitement strictement local. Cependant, il faut contrebalancer ce résultat par le fait qu'un traitement local, s'il nécessite une grande quantité de mémoire, pourra faire appel au mécanisme de pagination, qui coûte généralement plus cher encore qu'un message. C'est la raison pour laquelle nous privilégierons généralement des structures de données compactes, au détriment, éventuellement, de la vitesse de recherche dans ces structures, l'ensemble devant rester rapide relativement aux vitesses de transmission ou d'accès disque.

Par ailleurs, il est important pour éviter les surcoûts inutiles de pouvoir détecter le plus tôt possible les cas favorables dont la solution est locale. Pour augmenter les cas favorables, on peut être amené à déplacer les méta-données pour les rapprocher du site où elles sont le plus utilisées. C'est ce que nous appelons la migration du segment, qui consiste à déplacer son descripteur. Le coût de cette opération doit rester aussi bas que possible, et il faut pouvoir déterminer le meilleur moment pour l'effectuer, ce qui est un problème typique d'optimisation en ligne que nous résolvons à l'aide d'algorithmes heuristiques classiques [28].

La question primordiale est le placement des informations de localisation. Ces informations doivent en effet se trouver sur les sites qui en ont le plus besoin. Or, tout segment doit rester localisable depuis chacun des sites, ce qui implique une structure répartie où le point de départ de la recherche est clairement déterminé à tout instant. Il est problématique de concilier les exigences de mobilité et d'accessibilité.

La solution que nous avons adoptée combine la rapidité des tables de hachage et la densité d'information de l'algorithme d'allocation dichotomique. Pour résumer, elle consiste essentiellement à structurer la mémoire globale en deux niveaux de grain, et à stocker les méta-données relatives à chaque niveau dans des structures adaptées à ce grain particulier. Le niveau à gros grain est constitué de partitions de taille fixe. Chaque

partition est découpée à son tour en blocs de taille fixe, et l'indexation des blocs se fait par une technique de hachage. Les segments sont alloués et rangés dans les blocs par un algorithme dérivé de l'allocation dichotomique, qui permet une allocation rapide et une structure compacte, au prix d'un taux d'utilisation non optimal de l'espace virtuel.

Par-dessus cette mécanique de localisation à deux niveaux, nous intégrons dans l'algorithme de recherche des *accélérateurs* de localisation qui permettent d'améliorer la localité de l'algorithme dans de nombreux cas favorables.

* * *

Notre réalisation comprend de nombreux paramètres et options qu'il n'a pas été possible de figer par une analyse qualitative. Pour passer à une analyse quantitative, nous avons eu recours à l'expérimentation. Nous devions aussi montrer que notre mécanisme était réellement efficace, ou du moins qu'il se comportait de la manière prévue.

Les réglages à déterminer sont généralement de trois ordres :

- choix de paramètres quantitatifs (taille des tables, nombre de bits de hachage, etc.)
- choix algorithmiques lorsque ces choix ne peuvent pas être clairement justifiés par une analyse qualitative,
- choix de protocoles de synchronisation et de mise à jour des structures réparties (en particulier des accélérateurs).

L'objectif de cette étude n'est pas nécessairement de réaliser la mise en œuvre la plus rapide dans l'absolu. Il s'agit plutôt de dégager les mécanismes réellement utilisables parmi ceux qui sont envisagés et de repérer les comportements non intuitifs. De fait, nous devons soigneusement faire la part des choses entre les surcoûts dus à la plate-forme choisie, en éliminant par exemple le temps d'invocation des appels systèmes ou le temps de transit à travers les pilotes de STREAMS [4]. De même nous ne nous attacherons pas outre mesure aux délais imposés par les communications, mais compterons plutôt le nombre, la taille et le type des messages. Ces précautions sont nécessaires pour faire apparaître les bénéfices ou déficiences spécifiques à tel ou tel algorithme ou telle ou telle structure.

Notre plan d'expérimentation comprend deux types d'expériences :

- des micro-tests pour faire jouer les mécanismes de base, et
- des scénarios d'utilisation pour percevoir le comportement global.

Le premier type de tests permet de déterminer où se trouvent les compromis à faire, et quels sont les extrêmes. Le second type vient ensuite déterminer dans quel sens ces compromis doivent s'orienter pour s'adapter à des situations particulières. Notre arrière-pensée est de pouvoir codifier aussi rigoureusement que possible les gammes de paramétrage les mieux adaptées à tel ou tel type d'applications.

Plan de la thèse

La suite de cette thèse se divise en quatre chapitres et une conclusion.

Dans le premier chapitre, nous étudions d'abord l'évolution des modèles de multiprogrammation, depuis le modèle client/serveur jusqu'au modèle de mémoire partagée répartie (MPR). Cette revue historique sert de toile de fond à l'étude des MPR réalisées antérieurement, et permet de dégager l'origine de leurs points forts et de leurs défauts. À partir des leçons apprises de ces réalisations antérieures, nous argumentons en faveur d'une nouvelle MPR, à la lueur des avancées techniques accomplies récemment. Puis, nous en arrivons aux objectifs particuliers de notre MPR, Arias, en définissant plus formellement les notions qui y interviennent. Enfin, nous énonçons la problématique de cette thèse, puis esquissons l'architecture générale d'Arias en mettant aussi en lumière les contraintes que nous nous imposons.

Le second chapitre commence par décrire les interfaces entre les divers modules d'Arias et le module de gestion de segments, qui est notre préoccupation principale. Nous détaillons en particulier les interfaces avec la couche générique de cohérence et avec le module de pagination de mémoire virtuelle, dont nous décrivons aussi le lien avec le système de protection inter-application. Nous faisons ainsi le point sur l'ensemble des méta-données qui accompagne chaque segment, et la manière dont on veut pouvoir y accéder. Pour l'organisation de ces méta-données, nous étudions successivement plusieurs possibilités et identifions leurs lacunes. Nous décrivons finalement la structure que nous avons conçue et les raisons pour lesquelles cette structure est adaptée à notre cahier des charges.

Le troisième chapitre décrit la mise en œuvre technique de cette conception. Les algorithmes et les structures de données utilisés pour la localisation et l'allocation sont décrits et justifiés par des analyses qualitatives. Nous faisons appel aux contraintes et aux modes d'accès présentés pour éliminer ou préférer certaines solutions à d'autres. Nous détaillons les notions que nous avons déjà introduites : partitions, tables de hachage, allocation dichotomique, accélérateurs. Nous terminons par une image globale de cette architecture et énonçons les choix qui restent à faire et les paramètres qui restent à fixer.

Les choix et les paramètres font l'objet du quatrième chapitre. En suivant le plan d'expérimentation esquissé plus haut, nous détaillons le protocole de tests. En pratique, nous n'avons pas eu accès à une machine à 64 bits d'adressage (le processeur PowerPC 620, prévu pour la fin de l'année 1996, n'est pas encore disponible sur le marché). Nous avons donc procédé à des tests réels sur machines 32 bits, et nous avons simulé le comportement du système dans le cas des 64 bits. Cela nous a permis de paramétrer le système dans les deux cas, sachant bien entendu que les choix diffèrent sensiblement d'un cas à l'autre.

En conclusion, nous évaluons la portée de cette étude par rapport au système Arias global et par rapport aux MPR en général. Nous faisons le point sur les qualités et les défauts que l'on peut trouver à Arias, et les leçons que nous pouvons tirer sur la manière de réaliser une MPR ayant les caractéristiques que nous visions.

Chapitre I

La mémoire partagée répartie

Dans ce chapitre, nous étudions d'abord l'évolution des modèles de multiprogrammation, depuis le modèle client/serveur jusqu'au modèle de mémoire partagée répartie (MPR). Cette revue historique nous sert de toile de fond pour faire un tour d'horizon des MPR réalisées antérieurement, et étudier leurs points forts et de leurs défauts. À partir des leçons tirées de ces réalisations antérieures et à la lueur des avancées techniques récentes, nous argumentons les raisons en faveur d'une nouvelle MPR. Puis, nous en arrivons aux objectifs particuliers de notre MPR, Arias, en définissant plus formellement les notions qui y interviennent. Enfin, nous énonçons la problématique de cette thèse, puis esquissons l'architecture générale d'Arias en mettant aussi en lumière les contraintes que nous nous imposons.

I-1 Du client/serveur à la MPR

Le succès du modèle client/serveur est dû à des caractéristiques qui le rendent indispensable dans de nombreuses applications : la protection et la modularité. De plus, les interfaces évoluées telles que le RPC et l'appel d'objets l'ont rendu à la fois plus facile à utiliser et plus souple d'emploi. Étudions les avantages qu'apporte ce modèle, et gardons en mémoire qu'une MPR, si elle devait prendre la place d'une interface client/serveur, devra fournir l'équivalent ou au moins une approximation de ces avantages.

I-1.1 Le modèle client/serveur

Les origines du modèle remontent aux premières expériences d'interconnexion en informatique. Il consiste à définir une interface constituée d'un protocole d'échange de messages qui permette à un processus client de formuler des requêtes d'information ou de traitement vers un serveur. Ce dernier possède les données et exécute le traitement à la demande du client.

Ce modèle a trouvé une synthèse élégante avec celui des processus séquentiels communicants [37], qui décrit les interactions des processus dans un environnement de

multiprogrammation. En effet, les notions de séquentialité sont bien définies lorsque le mode de communication des processus est simplement modélisé par un système d'échanges de messages. De plus, le modèle confère une structure modulaire à la programmation. Ainsi des systèmes d'exploitation comme V [17] ou THE [26] utilisent un modèle client/serveur pour l'accès aux différents services.

I-1.2 L'appel procédural à distance

Le modèle client/serveur s'est trouvé consolidé par la formalisation du RPC (*Remote Procedure Call*) par Nelson et Birrell [8]. Une interface RPC entoure l'interface d'échange de messages protocolaires avec une interface d'appel de procédure. Le RPC rend la programmation d'un système client/serveur beaucoup plus aisée, car tous les aspects d'empaquetage des paramètres et de transfert de données sont pris en charge par le compilateur d'interface. Cependant, un système de RPC ne sait pas gérer des paramètres de type pointeur ou le transfert de tableaux, et de manière générale le passage de structures complexes. En effet, tous les échanges de données se font par valeur.

Le RPC simplifie aussi les aspects de protection du modèle client/serveur. En effet, la protection du serveur nécessite de superposer un protocole d'authentification au-dessus du protocole d'échange d'information proprement dit. L'interface RPC, résout ces aspects de manière transparente au programmeur. De plus, le RPC s'accommode très bien de l'hétérogénéité : même si l'appelant et l'appelé sont des machines de types différents, un système de RPC utilisant un format commun permet à ces machines de communiquer avec la même interface d'appel procédural.

Le système de RPC peut aussi prendre en charge la localisation du processus appelé. Le même appel procédural peut donc être destiné à un processus local (c.-à.-d. résidant sur la même machine que l'appelant) ou distant. Cette facilité de nommage rend transparente la localisation des points d'entrée. Cela permet par exemple d'utiliser les RPC de manière strictement locale pour les appels inter-processus. Le modèle de RPC est tellement bien adapté à cette tâche que des variantes strictement locales, telles que le LRPC [6] ont été développées.

Les systèmes de communication ont connu un point de maturité avec l'apparition de PVM [32], qui est une interface très simple d'échanges de messages et d'exécution à distance. La force de l'interface PVM est d'avoir été portée sur la plupart des architectures parallèles et distribuées existantes, ce qui en fait un standard de fait dans la programmation parallèle. Un programme qui utilise PVM fonctionne de la même manière sur super-ordinateur parallèle comme le T3E de Cray [19] ou sur un ensemble de stations de travail sous Unix relié par un réseau Ethernet.

I-1.3 Le modèle de MPR

Le modèle de MPR s'est dégagé progressivement à partir des travaux en architectures parallèles, telles que la machine DASH [48], ou C.mmp [84] puis Cm* [39, 63], et plus

récemment KSR [75]. Sur ce type de machines, les *n* modules processeurs sont reliés aux *m* modules mémoire à travers un réseau d'interconnexion, selon diverses architectures, dont quelques-unes sont illustrées sur la figure I-1.3. Ces travaux ont permis de classifier les méthodes de mise en cohérence de la mémoire, c'est-à-dire la manière dont un système peut faire croire aux processus qu'ils utilisent une mémoire physiquement partagée. Cela se fait en définissant des types de cohérence comme la cohérence séquentielle [46] qui précise les garanties offertes par le systèmes quant à la mise à jour des mémoires à travers l'interconnexion.

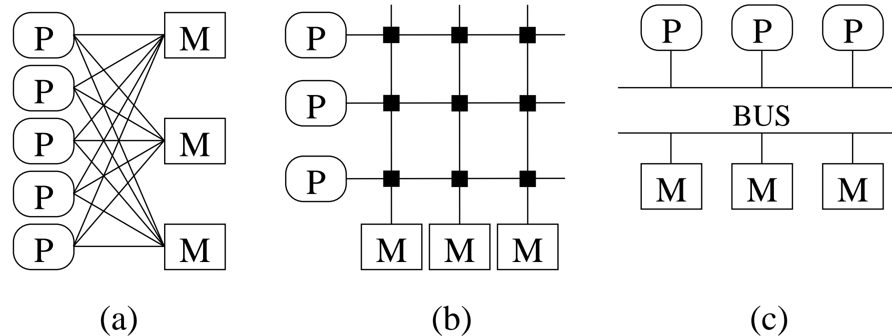

Figure I-1 : Exemples d'architectures multiprocesseurs. Les boîtes «M» sont des modules mémoire, «P» des processeurs. (a) connexion complète (b) switch (c) bus

La thèse de doctorat de Kai Li [50], soutenue en 1986 à l'université Yale, est la transposition de ces techniques à un ensemble de machines reliées par un réseau local. À la différence des machines parallèles citées ci-dessus, cette architecture comprend *n* machines complètes (processeur + mémoire) reliées par un réseau relativement lent et globalement partagé (cf. figure I-2. L'objectif du système Ivy est de rendre accessible la mémoire d'une machine A à partir d'une machine B comme si cette mémoire faisait partie de la machine B.

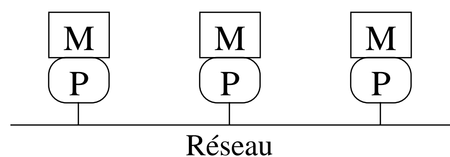

Figure I-2 : Architecture d'un réseau local de stations

Cette abstraction de mémoire partagée répartie s'est avérée extrêmement puissante, comme en témoignent les nombreuses utilisations que l'on en a faites. En particulier, de plus en plus de systèmes à objets répartis s'appuient sur des mécanismes de MPR.

I-1.4 Les modèles à objets

Le modèle de programmation par objets a ceci d'intéressant qu'il participe des deux modèles (client/serveur et mémoire partagée). En effet, un objet est, à la base, une structure de données, et l'accès à ces données revient à partager une petite portion de mémoire. D'un autre côté, le modèle d'objets impose aux accès de toujours passer par une interface composée de méthodes, qui sont des points d'appel procédural.

En calquant la technique du RPC sur l'appel de méthode, on parvient à créer un système d'objets distribués tel qu'Emerald [40] ou Guide [33], qui sont des *machines virtuelles à objets distribués*. L'intérêt d'un tel système, par rapport au modèle client/serveur traditionnel, est de permettre une grande liberté de placement. Dans Emerald, des directives de placement permet au programmeur de spécifier par exemple qu'un objet appelé par un autre objet doit d'abord migrer vers le site de l'objet appelant avant d'exécuter la méthode appelée. Ce contrôle fin du placement relatif des objets nécessite que le programmeur connaisse bien le fonctionnement de son application pour qu'il puisse en tirer le maximum de bénéfices. Mais alors, à partir de quelques annotations dans le programme, le système se charge d'effectuer tous les transferts nécessaires de manière transparente. Guide 1 [33] est un autre système et langage à objets où l'on privilégie le transfert d'exécution, ce qui rapproche son modèle d'exécution du modèle client/serveur.

D'un autre côté, comme Emerald a évolué vers Amber [11] et Guide 1 vers Guide 2 [31], les systèmes à objets distribués peuvent tirer grand profit de mécanismes de MPR. En effet, en utilisant une mémoire partagée entre différentes machines contenant les mêmes objets, et en utilisant les points d'entrée des méthodes pour effectuer la synchronisation d'accès à cette mémoire, on obtient un système capable de gérer la localité par transfert de données. Les propriétés de localité s'en trouvent modifiées, et l'on peut faire la distribution dynamique des ressources de manière plus souple qu'avec le seul transfert d'exécution.

I-2 L'évolution des MPR

Dans cette section, nous passons en revue un certain nombre de travaux antérieurs ou actuels dans le domaine des mémoires partagées réparties. Nous décrivons brièvement les caractéristiques principales de ces systèmes et soulignons leur apport. Nous analysons aussi leurs forces et leurs faiblesses et nous évaluons dans quelle mesure ils contribuent aux objectifs que nous nous sommes fixés.

Les projets les plus récents se concentrent sur les SASOS (*Single Address Space Operating System*), où le système d'exploitation est entièrement plongé dans un espace d'adressage global réparti et uniforme. Un autre sujet à la mode est celui des systèmes à objets persistants où le système d'exploitation lui-même est entièrement composé d'objets contenus dans une MPR globale.

I-2.1 L'ancêtre : MULTICS

MULTICS [20] est un des projets les plus novateurs dans le domaine des systèmes d'exploitation. Une des contributions majeures de MULTICS concerne l'accès à la mémoire. Les points importants de son interface mémoire sont :

- la segmentation,
- la persistance,
- l'uniformité,
- la protection d'accès,
- le chargement à la demande.

La segmentation de la mémoire, dictée par l'architecture matérielle, permet de mettre en œuvre la persistance à travers un mécanisme d'équivalence entre segments et fichiers. La mémoire est alors dite *uniforme*, car l'image d'un segment en mémoire et sur disque est la même quel que soit le processus qui y accède[2]. De plus, les fichiers sont soumis à un système de protection en anneaux qui permet une bonne séparation entre le système et les programmes des usagers. Enfin, le chargement des pages mémoires se fait à la demande, de même que le déchargement des pages vers le disque se fait de manière transparente et en fonction des besoins.

Ces caractéristiques ont fait de MULTICS un vrai système multi-usager[3], où de nombreux utilisateurs pouvaient travailler simultanément tout en se protégeant les uns des autres et en conservant l'intégrité du système. Les idées qui ont été intégrées dans MULTICS dès 1965 ne cessent d'être redécouvertes avec chaque génération de chercheurs en informatique.

I-2.2 La MPR de pages

L'idée originelle de la mémoire partagée répartie peut être attribuée à Kai Li et sa thèse de doctorat de 1986 [50]. Dans cette thèse et dans des publications correspondantes, Li présente son système prototype Ivy.

Le système Ivy repose sur des pages. La page mémoire, telle que définie par l'architecture matérielle, est l'unité de partage et de distribution. À chaque instant, chaque page partagée est gérée par un unique site *propriétaire*, qui seul à le droit d'accès en écriture. Tant qu'aucune écriture n'a eu lieu, d'autres machines peuvent obtenir des copies de la page pour la lecture seule. Dès qu'une écriture est effectuée sur la page, les copies en lecture sont invalidées, réduisant le nombre de copies existante à une seule, celle

[2]Notre propre définition de l'uniformité, que nous explicitons en I-4.2-a, est légèrement différente.

[3]Un des leitmotiv de MULTICS était justement d'offrir un accès simplifié à des utilisateurs non experts, augurant ainsi le début de la fin du règne des opérateurs.

du propriétaire de la page. Une fois l'écriture effectuée, les lecteurs peuvent à nouveau demander des copies fraîches pour la lecture.

La technique de placement dynamique [52] consiste à transférer la propriété d'une page d'un site à un autre en fonction des processus qui écrivent sur la page. Par exemple, si la page est gérée à un instant donné par le site A, et qu'un site B tente une écriture, le système de mémoire du site B doit d'abord obtenir du site A la propriété de la page, ce qui a pour effet :

- d'invalider immédiatement la copie sur A ainsi que toutes les autres copies,
- de transférer le contenu de la page vers B, et
- d'investir B de la propriété de la page.

La détection des accès en lecture ou en écriture repose sur le support matériel de mémoire virtuelle. Une analyse théorique permet à Li d'affirmer que ce protocole de placement dynamique est le plus performant dans la majorité des cas.

On constate que, du point de vue des applications, le comportement de cette mémoire répartie est équivalent à celui d'une mémoire physiquement partagée : on dit que cette mémoire répartie possède la *cohérence séquentielle*. Un programme qui s'exécute correctement sur une mémoire physiquement partagée s'exécutera de la même manière sur une mémoire répartie dotée de la cohérence séquentielle. Cette caractéristique peut avoir une grande importance pour les milieux industriels où le coût de portage des applications est un souci omniprésent.

Ce modèle de cohérence, que l'on appelle aussi la *cohérence forte*, pose cependant plusieurs problèmes. D'abord, avec des tailles de pages relativement importantes (quelques kilo-octets), il y a risque de *faux partage*. On dit qu'il y a faux partage sur une page si deux processus distincts accèdent simultanément chacun à une partie différente de la page. Comme l'unité de partage est la page, seul un processus peut accéder à la page à un instant donné, ce qui limite le degré de parallélisme et augmente le nombre de changements de contexte dus à l'attribution alternative des droits d'accès. Dans le cas d'une page partagée entre deux machines distinctes, cette page doit effectuer un va-et-vient entre les deux sites pour satisfaire les accès. Si les données traitées par ces deux processus résidaient sur des pages différentes, ces processus auraient pu s'exécuter simultanément sans l'intervention des opérations de mise en cohérence.

Le problème de faux partage peut être évité par différentes méthodes. Une solution consiste à remplacer les messages d'invalidation par des mises à jour. Lorsqu'une écriture a lieu, seules les modifications sont envoyées aux copies. Ces dernières restent alors à jour après l'écriture, et l'on a minimisé la quantité de données transférées. Cette technique de mise à jour différentielle (*diffing*), en conjonction avec une mise à jour différée, est utilisée par TreadMarks [41] pour optimiser le protocole de mise en cohérence des pages partagées.

Ce genre de technique implique une politique de cohérence optimiste et suppose que l'application ne s'adonne pas à une compétition d'accès (*data race*). En d'autres termes,

le systèmes est capable de fusionner un ensemble de mises à jour sur la même page à condition que ces mises à jour ne chevauchent pas. Une application qui se synchronise correctement ses accès à un grain plus fin que la page est assurée de ne pas entraîner de compétition d'accès ; dans ce cas le système peut gérer les mises à jour correctement.

I-2.3 Les modèles de cohérence

Le problème du faux partage est une manifestation du problème plus général de la mise en cohérence des données partagées. Les systèmes comme Ivy et TreadMarks offrent aux application l'illusion d'une mémoire séquentiellement cohérente. Or, si l'on exige de l'application qu'elle délimite exactement la plage de mémoire sur laquelle elle travaille à un instant donné, le système peut accepter des incohérences temporaires sur les parties de la mémoire que l'application n'utilise pas à cet instant [57]. Cela permet de différer et restreindre les mises à jour, de les accumuler, ou de les transmettre de manière asynchrone aux moments les plus justes. Il existe ainsi une panoplie de modèles de comportement de la mémoire que l'on regroupe sous le terme de *cohérence relâchée*.

L'utilisation d'un modèle de cohérence relâchée nécessite que l'application contienne des annotations ou des déclarations supplémentaires qui signalent les accès à des parties de mémoire. Une idée naturelle est d'associer ces annotation à une sémantique de synchronisation sur des données partagées. En effet, il se trouve que des processus parallèles ont naturellement besoin de synchroniser leurs accès aux données partagées. L'idée des systèmes comme Munin ou Midway est de profiter de ces points de synchronisation pour effectuer en même temps la mise en cohérence.

Munin [9] est un système de MPR s'appuyant sur la présence d'annotations dans l'application qui indiquent le protocole de cohérence à utiliser pour chacune des variables partagées. Il nécessite un pré-compilateur qui se charge d'interpréter ces annotations et d'insérer des opérations de mise en cohérence et de synchronisation au moment des accès. Munin offre de nombreuses options pour permettre au programmeur de choisir le type de cohérence avec beaucoup de finesse.

Midway [7] va plus loin que Munin dans le sens qu'il comprend non seulement un compilateur spécialisé, mais aussi un outil d'aide à l'annotation des programmes. Un programmeur n'appréhende pas toujours le comportement précis de toutes les variables partagées de son application. Les outils de Midway l'aident alors à mettre au point ces annotations, en partant par défaut d'un modèle simple de cohérence forte, qui garantit la correction de l'exécution au détriment des performances, et en offrant une palette d'autres modèles pour optimiser les parties les plus critiques où les optimisations ont un impact important.

I-2.4 Les systèmes à objets

On a vu l'intérêt qu'il y a à définir un grain de partage plus fin que la page, et qui corresponde à une structure sémantique applicative. La programmation par objets offre

justement une abstraction de ce type. Un objet possède une structure interne et une interface d'accès clairement spécifiée. Chaque méthode de l'interface d'un objet ne peut directement manipuler que les données contenues dans cet objet. En sérialisant les accès aux méthodes de l'interface, on en garantit le fonctionnement correct.

Par exemple, l'évolution d'Emerald, une machines virtuelles à objets distribués vers Amber [11], une MPR d'objets, constitue à la fois une extension et une restriction. L'extension provient de l'utilisation d'une mémoire partagée répartie, qui rend transparent le nommage des objets (les noms d'objets d'Emerald était des identifiants particuliers ; ceux d'Amber sont des adresses). La restriction est qu'Amber fournit ce support à un unique espace d'adressage non persistant, distribué parmi un ensemble de stations, c'est-à-dire sans protection. Amber permet à une application de s'étendre sur plusieurs machines, mais ne permet pas à plusieurs applications de coopérer à travers des objets persistants.

L'objectif des systèmes de MPR d'objets est de rendre dynamique et transparente la localisation des objets de manière à avoir plus de liberté pour optimiser le placement des objets. En pratique, ces systèmes nécessitent des techniques astucieuses pour atteindre cet objectif. Par exemple, le transfert d'un objet d'un site à un autre nécessite de reconstruire tout un environnement d'exécution approprié sur le site cible, en particulier la pile d'exécution.

Par ailleurs, une MPR d'objets implique aussi que l'on confère une sémantique aux données de la MPR, et nécessite que la MPR connaisse cette sémantique, ce qui diminue d'autant sa généralité. De plus, on crée un lien fort entre la MPR et le langage de programmation utilisé. Cette tendance vers l'utilisation du modèle de MPR dans les systèmes à objets montre malgré tout que la MPR est un support bien adapté aux objets distribués et persistants.

I-2.5 Les SASOS et la protection inter-applications

De même qu'il existe des systèmes d'exploitation entièrement constitués d'objets (par exemple Clouds [21]), on a envisagé des systèmes dans lesquels tous les processus partagent le même espace d'adressage : les *SASOS* (*Single Address Space Operating System*) [64]. Cette structure a l'avantage de rendre directement adressable l'ensemble des données présentes dans une machine, ce qui simplifie grandement le partage des informations. Toute donnée a un unique identifiant, son adresse virtuelle, et ce pour toutes les machines participant au système.

Pilot [68] est un système d'exploitation pour ordinateur personnel. Il a été implémenté et vendu sur les ordinateurs Apple II. Son objectif était de fournir aux ordinateurs de cette taille un système capable de multiprogrammation (multi-tâche). Pour cela, sachant que la plupart des micro-ordinateurs n'avaient alors pas de mémoire virtuelle, il a fallu se contenter d'un espace d'adressage unique, sans même de protection matérielle. La seule protection de Pilot était fournie pas un langage dédié, Mesa, avec lequel tout est programmé, y compris certains sous-systèmes. C'est le compilateur qui se chargeait de vérifier les références et de s'assurer que les applications n'interféraient pas entre elles.

Plus récemment, Opal [15] est un projet de SASOS réparti. La mémoire globale d'Opal est répartie, et l'ensemble des machines constituant le système possèdent toutes le même espace d'adressage. En composant la persistance sur une telle mémoire globale, on obtient une mémoire globale uniforme (*single-level store*). La localisation et le stockage sont alors totalement transparents.

Le point crucial d'un tel système est la protection : une donnée adressable n'est pas nécessairement accessible si le processus qui tente d'y accéder n'a pas les droits requis. La protection dans Opal repose sur le support matériel de mémoire virtuelle paginée. L'unité de protection est le *segment*, et un processus s'exécute dans un *domaine* qui définit son contexte d'adressage et ses droits d'accès. En fait, les domaines référencent les segments par des *capacités* [13], qui servent à la fois à la désignation et à la protection. La manipulation des capacités doit passer par des appels systèmes spécifiques, puisqu'elles sont stockées au sein du noyau du système, protégées de toutes les applications.

Le système Mungi [36], possède une structure similaire à Opal, à la différence que les capacités sont en fait des mots de passe. Une capacité est simplement un jeton qu'une application doit présenter au système lorsqu'elle requiert l'accès à un segment. Cette mécanique de capacité par mot de passe (*password capability* [79]) a l'avantage de simplifier l'échange des données. Une application peut donner à une autre application l'accès à un segment simplement en lui fournissant le mot de passe. Inversement, les désavantages sont d'une part que le système ne maîtrise pas le transfert abusif des capacités, et d'autre part que l'application doit utiliser une interface particulière pour s'ouvrir l'accès à un segment : le couplage d'un segment dans l'espace accessible d'un processus n'est pas automatique.

En combinant un grand espace d'adressage et un système à objets, on a abouti récemment à des systèmes d'une grande maturité et d'une grande sophistication. On peut dénombrer quelques systèmes représentatifs de ce mouvement de consolidation des techniques accumulées au fil des expériences et des projets de recherche.

- Angel [60] propose de fusionner trois concepts :
  - un système à objets,
  - un micro-noyau, et
  - un espace d'adressage global unique.
- Larchant [29] est une MPR à objets persistants qui combine l'élimination des miettes et la mise en cohérence. La persistance dans Larchant est dite *par référence*, c'est-à-dire que certains objets particuliers sont déclarés comme *racines de persistance*, et tous les objets référencés par les racines sont alors «vivants». Tout objet qui n'est pas accessible depuis les racines est une miette.

  Larchant possède un protocole de cohérence intégrant le ramasse-miettes d'objets. En fait, la thèse de Larchant est précisément que ces opérations sont intimement liées.

Un grand espace de mémoire virtuelle persistante et répartie est un support de programmation distribuée qui possède de nombreux avantages. Cependant, un système d'exploitation entièrement contenu dans un tel espace n'est pas nécessairement la bonne solution, et en tous cas pose de nombreux problèmes tant théoriques que pratiques [12]. En effet, certaines structures de données comme la pile d'exécution ou les structures internes du système ont intérêt à être séparées et protégées de l'espace global [59]. De plus, malgré certaines propositions d'évolution architecturale comme le *Protection Lookaside Buffer* [45], la réalisation d'un système de protection fin et efficace reste problématique.

I-2.6 Synthèse

Comme le souligne le manifeste du projet Mungi [71], une MPR complète et utilisable nécessite beaucoup plus que la simple persistance et la distribution. Une MPR persistante et uniforme comporte des composantes dans plusieurs domaines :

- la protection inter-application,
- la résistance aux pannes,
- une interface applicative simple et transparente, accompagnée d'outils d'aide à la programmation,
- un système de mise en cohérence efficace,
- la capacité à s'adapter à des besoins différents.

I-3 Pourquoi une nouvelle MPR?

L'énumération des projets antérieurs ci-dessus pourrait faire penser que tout, ou presque, a déjà été fait en matière de MPR. Il existe cependant plusieurs raisons pour motiver la réalisation d'une nouvelle MPR.

I-3.1 L'adressage uniforme

Il y a quelques années, les MPR ont trouvé un regain d'intérêt avec l'apparition des processeurs à grands espaces d'adressage. Ces nouveaux processeurs offrent un espace d'adressage agrandi, sur 42 à 64 bits. La motivation initiale était que certaines applications particulièrement exigeantes nécessitaient à elles seules plus que les 4 Go de mémoire auxquels les processeurs 32 bits pouvaient accéder. Un exemple frappant est la base de données du service de localisation sur le WWW appelé *Alta Vista* [24]. Ce serveur contient une base de données de 6 Go entièrement stocké en mémoire physique.

Il s'avère que le doublement de la largeur d'adressage provoque la quadrature de l'espace correspondant. 64 bits permettent de distinguer 2^{64} adresses, soit environ 8.10^{18} (8 exa-) octets. Une telle taille permet de ne jamais réutiliser l'espace virtuel lorsqu'un objet est détruit. En effet, un calcul simple [14] montre que même en «consommant» les adresses virtuelles au taux confortable d'un giga-octet par seconde, cet espace ne sera épuisé qu'au bout de 500 ans. Il est possible qu'alors un adressage encore agrandi, sur 128 bits par exemple, aura mis fin à cette boulimie.

De nombreux projets exploitant un espace d'adressage étendu ont déjà vu le jour, parmi lesquels Opal de l'université de Washington aux USA [15], Mungi de l'université des Nouvelle Galles du Sud en Australie [36], ou encore Angel de la City University en Angleterre [60]. L'avantage de cette politique de consommation est qu'un objet, s'il existe, garde toujours la même adresse, donc le même identifiant, tout au long de son existence. Cela élimine les problèmes de translation d'adresses au moment de l'utilisation de la mémoire (*swizzling* [81, 80]).

Mais ces projets, pour pionniers qu'ils soient, sont très radicaux. Il s'agit de SASOS dans lesquels la totalité du système est plongé dans un espace uniforme global et unique. Cela implique des modifications importantes dans la structure des applications, et impose des difficultés de taille [59] dans la réalisation de sous-systèmes comme l'éditeur de liens et le chargeur de programmes. En élargissant autant l'ampleur du projet, l'aspect purement MPR se trouve noyé au milieu des autres sujets de réflexion tels que la distribution du traitement ou le support linguistique. Les besoins immédiats des applications actuelles ne nécessitent pas de systèmes révolutionnaires. Comme il existe encore de nombreux problèmes non résolus liés à la réalisation d'une MPR, il nous semble important de commencer par étudier et «apprivoiser» les techniques de MPR de base avant d'envisager des projets plus ambitieux.

I-3.2 Les réseaux rapides

Une des limitations importantes des systèmes répartis par rapport aux architectures parallèles est la performance du réseau de communication. Les techniques récentes comme l'ATM [58] ou le FDDI [62] offrent des bandes passantes supérieures de plus d'un ordre de grandeur par rapport au traditionnel Ethernet. Même s'ils ne peuvent pas concurrencer les systèmes de communication inter-processeur comme le HIPPI [18], ces réseaux rapides brisent l'évolution du rapport réseau-processeur qui ne cessait de se dégrader.

Outre le médium de transport matériel, les interfaces de connexion connaissent aussi un renouveau. Grâce à la définition de standards d'interfaçage, on voit apparaître des cartes de communication «intelligentes» capables d'effectuer des traitements complexes sur les messages réseau. Il est ainsi possible de déporter une partie du traitement protocolaire sur la carte, libérant ainsi le processeur et le bus mémoire.

Il est intéressant de tenir compte de ces nouvelles technologies dans la réalisation d'une nouvelle MPR. En particulier, bien qu'il existe quelques travaux visant à exploiter ponctuellement les niveaux de performances offerts par le matériel récent (par exemple sur

la gestion des pages physiques [28]), nous avons encore peu de connaissances pratiques sur leur impact sur une MPR complète.

I-3.3 L'économie d'échelle

L'organisation socio-technique autour des moyens informatiques est aussi en rapide évolution. À l'inverse d'une structure centralisée traditionnelle [22], les moyens informatiques actuels sont de plus en plus axés sur de grands nombres de stations de travail puissantes et sur la répartition physique des données. Le rapport performance/prix croît plus vite pour les petits systèmes que pour les super-ordinateurs.

La difficulté reste l'interconnexion. Il est largement acquis qu'un système composé de n unités capables de m opérations par seconde n'est pas équivalent à un système d'une seule unité capable de $n \times m$ opérations par seconde. Si certaines applications, de par leur structure profonde, peuvent effectivement s'exécuter n fois plus vite sur n machines réparties que sur une seule, le rapport d'accélération vitesse/nombre de machines est le plus souvent sous-linéaire.

Cela dit, cette constatation reste vraie pour les super-ordinateurs parallèles. Le point important est que, pour une valeur donnée de puissance théorique, il est très nettement moins coûteux de s'équiper d'une grande quantité de matériel grand public que d'une seule unité de matériel très spécialisé [3]. De plus, un système composé d'équipements standards est plus facile à faire évoluer, à agrandir, et à entretenir. Le seul problème de ce type d'investissement est le support logiciel pour la programmation parallèle. La MPR contribue à résoudre ce problème.

I-3.4 La maturation de la MPR

Comme on a pu le constater à plusieurs reprises, les techniques de mise en œuvre de la MPR arrivent à maturation. Il importe désormais de produire des MPR qui, tout en étant innovantes, sont susceptibles d'apporter une réelle solution de développement industriel. À mesure que la technologie devient fiable, de plus en plus de projets en grandeur réelle s'intéressent à la MPR. Dans le projet Sirac [23], nous avons pu nouer de nombreux contacts industriels, et avons constaté l'attitude générale d'attente d'un produit de qualité et de fiabilité commerciales. Les milieux industriels connaissent déjà les avantages du modèle, mais il faut encore les convaincre que ce modèle est utilisable en pratique avec les logiciels existants.

I-4 Objectifs d'Arias

Nous avons étudié un certain nombre de projets ayant trait à la MPR, et souligné leurs originalités et leurs défauts. Nous avons ce faisant introduit les notions essentielles pour définir les objectifs d'Arias en général.

I-4.1 Objectifs

Les objectifs de la mémoire partagée répartie Arias sont multiples :

- implémenter une MPR persistante et uniforme fondée sur l'allocation de mémoire indifférenciée,
- éviter de conférer une quelconque sémantique aux données contenues dans la MPR,
- rendre transparents la localisation et le nommage des données en mémoire,
- permettre un contrôle d'accès à grain fin en laissant aux applications le choix du modèle de synchronisation et du protocole de cohérence,
- proposer facultativement un modèle de protection inter-application,
- offrir facultativement certaines garanties en cas de pannes avec l'aide de l'application,
- intégrer le tout à l'intérieur d'un système existant en minimisant l'impact sur celui-ci.

Nous allons détailler ces objectifs en définissant précisément les termes employés. Nous présentons ensuite l'architecture générale du système[4] et circonscrivons la portée de la présente thèse parmi les modules qui le composent.

I-4.2 Définitions

Pour aller plus loin dans la description d'Arias, il est nécessaire de définir avec exactitude les notions que nous faisons intervenir.

I-4.2-a Espace d'adressage partagé uniforme

Une mémoire partagée à adressage uniforme signifie pour nous que, quelle que soit la machine sur laquelle se produit l'accès à une donnée, son adresse désigne de manière unique la donnée en question. En d'autres termes, une adresse x désigne sans équivoque la même donnée à tout instant sur toutes les machines participant à la MPR.

Ce principe d'adressage uniforme implique que l'espace d'adressage offert aux applications soit constitué de deux régions : une région correspondant à l'espace d'adressage partagé et une région correspondant à l'espace d'adressage privé, local à l'application. Pour qui connaît le principe d'utilisation de la mémoire partagée sous Unix, cela revient à avoir une région partagée allouée par un appel à `shmat` à l'intérieur de l'espace

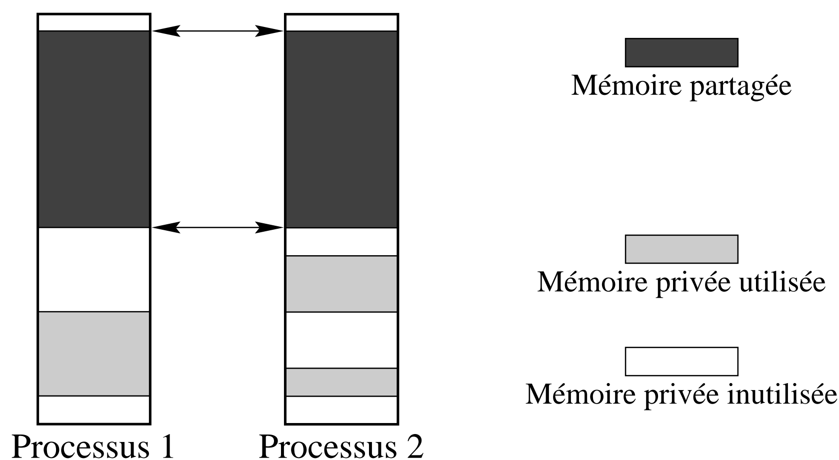

Figure I-3 : Le partage d'une région de l'espace d'adressage entre deux applications

d'adressage autrement totalement réservé à l'application. Schématiquement, la figure I-3 illustre l'espace d'adressage de deux processus avec une région partagée en gris foncé.

La caractéristique particulière de la MPR à adressage uniforme est que la région partagée se trouve toujours à la même adresse du point de vue de n'importe quel processus. Ce principe d'adressage uniforme possède des avantages importants [64]. Comme le nom global d'une donnée partagée est entièrement défini par son adresse, cela permet aux applications de manipuler les données partagées et les données privées de la même manière, sous la forme de simples adresses mémoire, sans nécessiter des identificateurs complexes. L'écriture des applications partagées s'en trouve grandement simplifiée.

De plus, le système de désignation est entièrement indépendant de la localisation : même si à un instant une donnée requise ne se trouve pas sur la machine d'où provient l'accès, et quelle que soit la machine où elle se trouve à ce moment, l'application n'a pas à se préoccuper de localiser la donnée ou d'en demander explicitement une copie locale.

I-4.2-b La mémoire persistante

Une des caractéristiques concomitantes au principe de partage global est que les données résidant dans l'espace partagé sont *persistantes*, c'est-à-dire qu'elles doivent survivre à la terminaison du processus qui les a créées. En effet, à partir du moment où une donnée a été instanciée dans l'espace global, n'importe quelle application sur n'importe quelle machine y a à priori accès, et peut en garder un pointeur. Même après la terminaison du processus créateur, la donnée peut être nécessaire à l'exécution d'un autre processus ; il faut donc la préserver.

La totalité de la MPR étant persistante, cela implique à son tour que cette MPR doit être

[4] Ceci a fait l'objet d'une présentation à ERSADS [34].

gérée à partir d'un niveau protégé et privilégié, où la terminaison normale ou accidentelle d'un processus ne peut perturber le fonctionnement de la MPR.

Précisons ici que nous distinguons nettement la persistance de la résistance aux pannes. La persistance est une propriété abstraite, alors que la résistance aux pannes est une propriété pragmatique, liée aux contingences pratiques. Une conséquence particulière de cette distinction est que l'allocation de mémoire persistante ne nécessite pas de réservation d'espace sur stockage secondaire. On a souvent posé l'analogie entre les données persistantes et les données stockés dans un système de fichiers. Un système de fichiers est certes une forme de mémoire persistante, puisqu'un fichier persiste après la terminaison du processus qui l'a créé. Mais un système de fichiers n'est pas la seule forme de mémoire persistante.

I-4.2-c Un grand espace d'adressage

Une MPR persistante et globale contient l'ensemble des données partagées entre plusieurs processus au cours de toute l'existence du système réparti. Cela représente une grande quantité de données, qui dépasse facilement la limite des 4 Go d'un adressage sur 32 bits.

Tous les grands constructeurs proposent désormais des architectures matérielles construits autour d'un adressage sur 64 bits. C'est le cas de l'Alpha de Digital [76], le premier de cette catégorie, ainsi que le R4000 de MIPS [35], le PowerPC 620 d'IBM et Motorola [56], le PA-RISC 8000 d'Hewlett-Packard [47], ou encore l'UltraSparc de Sun [78]. Nous pouvons alors faire la supposition qu'un seul espace d'adressage sur 64 bits peut contenir toutes les données d'un système réparti au cours de son existence.

I-4.2-d Partitions

Bien que la gestion de la MPR soit globale, il faut éviter que cette gestion ne provoque un goulot d'étranglement sur une machine particulière. Pour s'assurer de la répartition équitable des tâches de gestion entre les machines, nous devons répartir la gestion la MPR sur l'ensemble des machines. Pour cela, l'espace global est découpé en *partitions* et assignons la gestion d'un certain nombre de partitions à chaque machine.

Le partitionnement de la mémoire Arias ne concerne que les *méta-données*, les données de gestion pour la localisation des plages de mémoire. Il n'implique pas que les données elles-mêmes sont assignées à une machine particulières. Le placement et la réplication des données est entièrement à la charge des protocoles de partage et synchronisation, dont nous parlons plus loin.

I-4.3 Sémantique des manipulations

Idéalement à ce stade, on pourrait envisager de concevoir une MPR totalement uniforme et sans structure. Cependant, pour faciliter la gestion de la mémoire, nous avons introduit un niveau de segmentation des données. Dans la nomenclature Arias, un *segment*

est simplement une région contiguë d'espace mémoire qui constitue l'unité d'allocation. On requiert de l'application qu'elle demande l'allocation explicite d'un segment avant de pouvoir l'utiliser.

La fonction de création de segment entraîne plusieurs actions :

- l'allocation d'une région vierge de la MPR de la taille requise pour y loger le segment,
- la diffusion de cette allocation vers toutes les machines participant à la MPR pour signifier la réservation de cette région,
- la mise en place des structures de gestion du segment, en particulier vis-à-vis du système de pagination.

Une fois le segment créé, n'importe quel processus sur n'importe quelle machine peut y accéder (à condition qu'il en possède le droit, comme nous verrons plus loin, section I-4.4). Le segment, devenu persistant, a une durée de vie potentiellement illimitée. L'accès à un segment déjà créé ne nécessite pas d'action particulière de la part de l'application.

Une autre conséquence de l'adressage uniforme apparaît ici : le système sous-jacent est incapable de distinguer un pointeur d'une donnée. S'il le pouvait, il pourrait savoir quand un segment n'est plus référencé, auquel cas il pourrait le détruire. L'adressage uniforme interdit cette forme de ramasse-miettes.

Il est donc nécessaire de demander l'aide des applications ; elles seules peuvent savoir à quel moment un segment donné est définitivement inutile. Ainsi, la destruction d'un segment est explicitement requise (nous verrons qu'il existe un moyen de détecter l'obsolescence d'un segment à travers le système de protection, section II-8.4).

I-4.4 Protection inter-application

Comme nous l'avons vu, le segment est l'unité d'allocation (et de destruction) de la MPR. Il s'avère que l'introduction de cette unité de manipulation intermédiaire est propice à de nombreuses utilisations. En particulier, le segment est aussi l'unité manipulée par le système de protection.

La conception du système de protection d'Arias doit satisfaire à plusieurs contraintes.

- Il constitue un module optionnel. Un système entièrement dépourvu de protection peut être souhaitable lorsque l'on désire maximiser les performances et que la protection est assurée par un mécanisme extérieur à Arias.
- Il fournit un mécanisme sans imposer de structuration. En effet, les anneaux de protection de MULTICS [72] ou les droits d'accès d'Unix imposent les relations de confiance entre les processus. Il est difficile d'y faire coopérer deux applications mutuellement méfiantes.

- Il est indépendant de l'algorithmique. L'environnement de protection est défini par l'utilisateur (ou l'administrateur), et non par le programmeur. Le code même des programmes est indépendant de la protection ; de fait, le même programme doit pouvoir fonctionner aussi dans un environnement protégé que dans un environnement où la protection a été désactivée.

En imposant ces contraintes, notre objectif est de rendre la protection indépendante du reste d'Arias. Une application A donnée doit pouvoir s'exécuter et échanger des données avec une autre application B dans des environnements variés :

- en l'absence de protection,
- par échange de données à travers un segment protégé et partagé,
- par appel procédural dans les deux sens.

I-4.5 Résistance aux pannes

Un système distribué est essentiellement plus vulnérable aux pannes qu'un système mono-processeur. D'une part, la présence de plusieurs processeurs augmente d'autant la probabilité d'une panne de l'un d'entre eux. Ensuite, l'interconnexion des processeurs passe par un réseau dont la caractéristique habituelle est d'être asynchrone (les délais de transmission ne sont pas bornés). De plus, notre MPR est persistante, donc les données qu'elle contient ont une durée de vie potentiellement illimitée. Il faut pouvoir préserver cette propriété même en cas de panne : certaines applications désirent une garantie sur la permanence de leurs données, au prix d'un abaissement des performances ou/et d'une plus grande difficulté de programmation.

Par conséquent, Arias intègre un service facultatif de stockage fiable et de journalisation [42] pour servir les besoins de telles applications. À travers ce service, une application peut demander de maintenir l'*image permanente* d'un segment, et de mettre à jour cette image à travers un système de journalisation. Le module de stockage permanent est largement indépendant du système de gestion des segments. Le point de rattachement se situe à la localisation des images permanentes. En effet, le couplage d'un segment permanent nécessite une gestion de page plus complexe, du fait de l'existence de l'image permanente. Cela dit, la conception du module de stockage le rend largement indépendant, à la manière du module de protection. Dans le cadre de cette thèse, nous n'irons pas plus loin dans sa description.

I-4.6 Synchronisation et cohérence

La mise en cohérence des données réparties est un problème crucial pour une MPR. Notre objectif à ce sujet est non pas de fournir un protocole unique, mais d'offrir un support pour le développement de protocoles multiples et variés, adaptés aux besoins des

applications. La décision la plus importante a été d'associer étroitement la cohérence et la synchronisation. De nombreux travaux [7, 61] ont montré que dans des MPR à cohérence relâchée, les points de synchronisation peuvent servir de points de mise en cohérence moyennant un faible surcoût.

Généralement, un segment est constitué d'une collection d'objets de petite taille, de structures de données dont le grain se situe en-dessous de celui d'une page [16]. Pour maximiser le parallélisme des applications, il est nécessaire de pouvoir effectuer des synchronisations d'accès à un grain fin. Nous définissons donc une entité appelée la *zone*, qui est une suite contiguë d'octets sans limite de taille ni de frontière, si ce n'est qu'elle doit être entièrement contenue dans un segment. La zone est à la fois l'unité d'accès, de synchronisation et de mise en cohérence de la mémoire.

Les objets de grande taille peuvent se décomposer en un ensemble de pages contiguës encadré par deux petites zones (cf. figure I-4).

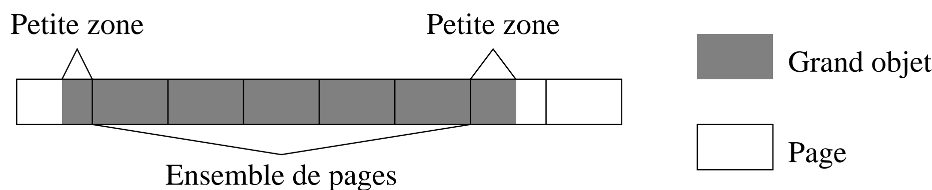

Figure I-4 : Composition d'un objet de grande taille. La partie sombre représent l'objet, les rectangles les pages.

Normalement, une application ne gère pas directement la synchronisation ou la mise en cohérence. Elle fait appel à une bibliothèque de fonctions correspondant à un modèle de synchronisation (par exemple, l'exclusion mutuelle). Cette bibliothèque est reliée à un *module spécifique de cohérence* qui se charge de mettre en œuvre le protocole de cohérence attaché au modèle de synchronisation (par exemple un protocole à copie unique). En-dessous de ce module se trouve une interface générique de manipulation de zones appelée CGC (*Couche Générique de Cohérence* [67]), dont le fonctionnement est décrit en II-1.2. Brièvement, la CGC se charge de la gestion des zones et de ses éventuelles répliques, ainsi que de la correspondance entre les zones et les sites chargés d'arbitrer les accès.

Une des originalités de cette architecture (cf. section I-6) est qu'elle permet la coexistence de plusieurs modules spécifiques au sein du système, chaque application choisissant le module le plus approprié a ses besoins.

I-4.7 Répartition et migration

Une des tâches d'un module spécifique de cohérence est de gérer le placement des méta-données. Par exemple dans Ivy [51], chaque page est gérée par un propriétaire qui en attribue les droits d'accès selon un modèle à plusieurs lecteurs et écrivain unique. Kai

Li a montré en l'occurrence qu'il était intéressant de transférer la propriété d'une page en même temps que le droit d'écriture. Arias n'impose pas un modèle particulier de placement ; c'est le module spécifique en charge de la zone qui détermine le placement des méta-données par rapport à l'utilisation des données et en fonction de critères qui sont propres au module.

I-5 La problématique de la gestion de segments

Nous avons vu les grandes caractéristiques d'Arias et la définition des segments comme entité intermédiaire de manipulation de mémoire. La gestion des segments est le point central de notre étude. Cette section décrit plus en détail le cahier des charges du module de gestion de segments, et les difficultés qui ont motivé les choix algorithmiques que nous avons faits.

I-5.1 Interface du module de gestion de segments

L'interface précise du module de gestion de segments avec le reste d'Arias est décrite en détail en section II-1. Nous en donnons ici un petit aperçu pour expliciter l'enjeu du module.

I-5.1-a La création et la destruction de segments

La création et la destruction de segments sont explicites (I-4.3). Pour cela, l'application dispose de deux appels correspondant à ces opérations. À noter que ces opérations nécessitent une coopération avec le CGC via des appels de fonction vers ce module.

I-5.1-b La gestion des zones

Lorsqu'une application accède à une donnée partagée, elle fait normalement appel à l'interface d'utilisation d'un module spécifique. Ce dernier fait alors appel à la CGC pour la manipulation de la zone correspondant à la donnée partagée. La CGC, à son tour, a besoin de trouver le descripteur du segment contenant la zone à gérer. Cette dernière opération constitue le fond de commerce du module de gestion de segments.

I-5.1-c Le couplage dans un espace d'adressage

Le système de mémoire virtuelle d'AIX permet la manipulation des objets mémoire du grain des segments. Il offre une interface de paginateur externe adéquat, sur laquelle repose le module de pagination Arias. Une difficulté surgit dans ce module au moment du traitement d'un défaut de page. Le système AIX fournit l'adresse fautive ainsi que l'identité du processus demandeur. Partant de ces informations, le paginateur doit être

en mesure d'identifier le segment correspondant pour procéder à la vérification des droits d'accès. Il s'agit donc de retrouver, à partir d'une adresse quelconque, l'identité du segment qui contient cette adresse[5]. C'est là qu'intervient le module de localisation de segment.

I-5.2 La localisation dans un grand espace d'adressage

L'utilisation d'un grand espace d'adressage encourage la fragmentation de l'espace virtuel, c'est-à-dire que les données réellement «vivantes» sont éparpillées à travers tout cet espace. En effet, nous avons adopté la politique qui consiste à ne jamais réutiliser une adresse virtuelle. Au fur et à mesure que des segments sont créés et détruits, il se crée des «trous» de plus en plus vastes dans l'espace d'adresses qu'on ne pourra jamais combler.

La fragmentation de l'espace virtuel pose un problème de gestion dans un grand espace d'adressage. Cet espace est tel qu'il est impossible de le représenter à l'aide de tables de pages habituelles, qui sont linéaires. Même les tables de pages hiérarchiques [27] sont peu efficaces en termes de place mémoire. Les tables de pages gardées [53], plus prometteuses, ne sont pas adaptées non plus. Ces techniques sont étudiées et critiquées en détail aux sections II-3.1 et II-3.2.

Il est important de maintenir les méta-données, c'est-à-dire les structures de données internes pour la gestion de l'espace utilisateur, aussi petites que possible. En effet, ces méta-données devront être transférées à travers le réseau au moment des changements dans la répartition de la mémoire (migration, création de segments, etc.). De plus, les méta-données doivent être fixées en mémoire physique pour éviter des défauts de page en cascade. Cependant, la compacité des structures de données nécessitant une technique d'indexation trop complexe nuit généralement à la vitesse de recherche.

La recherche doit être rapide, car tout accès à la CGC nécessite une identification de segment, de même que toute instanciation de page. Ces opérations fréquentes doivent donc être optimisées. Pour accélérer la recherche, nous maintenons un cache de localisation contenant des *accélérateurs de localisation* qui raccourcissent la recherche. Mais à son tour, la gestion et la mise à jour des accélérateurs nécessitent des algorithmes et éventuellement des messages supplémentaires, qui pour la plupart peuvent néanmoins se faire de manière asynchrone. Il existe un grand nombre de compromis et paramètres à ajuster pour obtenir un fonctionnement optimal de ce système.

I-5.3 L'allocation et la localisation

Une des originalités d'Arias, comme nous l'avons vu en I-4.2-b, est la possibilité de créer des segments persistants sans faire appel au stockage secondaire. Cela encourage le programmeur à utiliser systématiquement la mémoire persistante, car le surcoût de

[5] Il faut aussi être capable de déterminer, le cas échéant, que cette adresse n'appartient à aucun segment Arias.

l'allocation en mémoire globale par rapport à l'allocation en mémoire privée est faible. L'allocation doit être rapide, et pour cela elle ne doit faire appel ni au stockage secondaire, ni au réseau.

Il est clair que la vitesse d'allocation dépend des structures de données utilisées pour la localisation. Ces deux fonctions, l'allocation et la localisation, sont très subtilement liées. En effet, un mécanisme de localisation judicieux peut simplifier l'algorithme d'allocation, et un algorithme d'allocation astucieux permet ultérieurement une localisation rapide. Le mécanisme d'allocation que nous utilisons permet à la fois une allocation et une recherche rapides, et produit des structures de données compactes.

I-6 Architecture

Cette section décrit succinctement l'architecture générale d'Arias et l'interconnexion de ses modules. Cette description définit le module de gestion de segments et délimite la portée de ce rapport. Elle énumère les interfaces entre ce module et le reste d'Arias ; la description de ces interfaces ouvre le chapitre suivant.

Pour atteindre le double objectif de la conformité aux standards Unix et de la modularité, nous avons choisi d'implanter la quasi-totalité du système Arias dans l'environnement STREAMS. Nous avons choisi AIX 4.1 comme système de base, sur une plate-forme composée de stations Escala et Estrella fabriquées par Bull, munies de processeurs PowerPC 601 et 604 et reliées par un réseau local en Ethernet 10Base-T.

Nous avons composé les avantages des STREAMS et d'AIX. STREAMS fournit un moyen standard et relativement simple d'intégrer des modules dans l'environnement d'exécution privilégié. De plus, il encourage et aide naturellement la modularité : les services de protection et de journalisation sont des modules facultatifs qu'il est possible d'omettre complètement.

Le système AIX est une version d'Unix sophistiquée offrant en particulier une interface de programmation pour les systèmes de fichiers externes, ainsi qu'une structure segmentée de la mémoire virtuelle qui permet le vrai partage. Nous utilisons ces particularités pour assurer l'interfaçage entre la gestion des segments et le couplage à la demande. En outre, AIX possède naturellement des activités (*threads*) au niveau du noyau, de même que la possibilité de créer des processus noyau, s'exécutant entièrement dans l'environnement privilégié. Ces caractéristiques ont grandement simplifié la programmation.

L'architecture générale d'Arias est illustrée en figure I-5. Cette figure montre les différents modules et les interfaces systèmes qui interviennent dans leur fonctionnement.

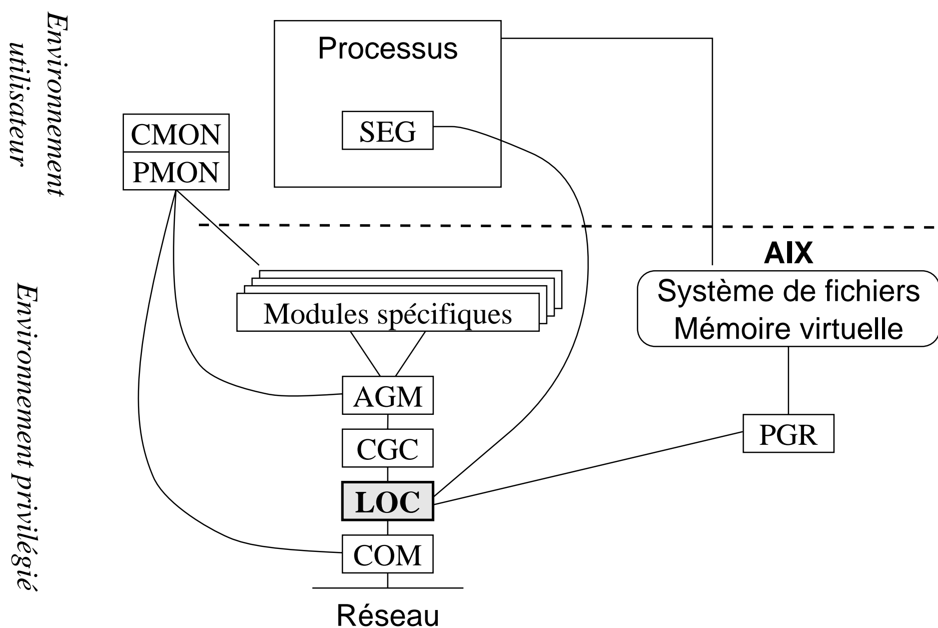

Figure I-5 : Architecture générale

I-6.1 La pile STREAMS

I-6.1-a COM : le pilote de STREAMS

Le module COM sert de base à la construction de la pile de modules et sert en même temps à gérer le protocole de communication interne à Arias. Ce pilote spécifie un format standard pour l'ensemble des messages qui sont véhiculés d'un module à un autre, distant ou pas. Le pilote se charge alors d'acheminer ces messages, à travers le réseau si nécessaire.

I-6.1-b AGM : le multiplexeur de modules spécifiques

Pour ce faire, le pilote, en coopération avec le multiplexeur de canaux STREAMS (module AGM), qui délivre des estampilles de messages, définit des règles de transfert de messages. Le multiplexeur permet la connexion simultanée de plusieurs modules spécifiques

I-6.1-c CGC : la gestion de zones

Le module CGC, que l'on a vu à la section I-4.6, communique avec les modules spécifiques à travers le multiplexeur. Il s'appuie sur LOC pour la localisation des zones et l'identification des sites primaires de segments.

I-6.1-d LOC : la gestion de segments

Le module de gestion de segments (LOC) est le module qui fait l'objet de la présente thèse. Ce module se charge à la fois de l'allocation et de la localisation des segments à travers tout le système.

I-6.2 PGR : le paginateur

Le module de pagination, qui n'entre pas dans l'environnement STREAMS comme les autres modules, est décrit en détail à la section III-3. En quelques mots, il se constitue d'un traitant d'exception pour les défauts de segments, d'un module de système de fichiers externe, ainsi que d'un ensemble de *threads* noyaux distribués à travers les processus représentant les applications Arias, et chargés du couplage des segments. Lorsqu'un processus provoque un défaut de page, c'est ce module qui est chargé d'assurer le couplage du segment dans l'espace d'adressage du processus fautif, ainsi que d'instancier les pages physiques sous-jacents.

I-6.3 SEG : l'interface applicative

L'interface applicative (décrite ci-dessous) de gestion de segments se présente sous la forme d'une bibliothèque de fonctions liée dans le processus qui utilise Arias.

I-6.4 Les moniteurs

Un moniteur (CMON) supervise en permanence l'état de connexion du réseau, alors qu'un autre (PMON) s'occupe de la gestion (chargement/déchargement dynamique) des multiples modules spécifiques.

I-7 Transition

Dans ce chapitre, nous avons revu quelques notions fondamentales sur la MPR et son évolution pour aboutir à la définition d'une nouvelle MPR, Arias, en justifiant les raisons qui en motivent la réalisation. Nous avons ensuite détaillé les caractéristiques générales d'Arias, et en particulier son système de mise en cohérence des données partagées et son objectif de placement dynamique agressif. Nous avons introduit deux grains de données, les zones et les segments, et avons défini le principe de gestion de ces derniers ainsi que les contraintes sur leur placements et leur déplacements. Nous avons terminé en esquissant l'architecture générale d'Arias et en situant l'objet de cette thèse, le module de gestion de segments, dans le projet global. Nous nous concentrons désormais sur la réalisation de ce module.

Chapitre II

Conception de la gestion de segments

... this manual ... doesn't actually tell the truth. When certains concepts ... are introduced informally, general rules will be stated ... the later chapters contain more reliable information than the earlier ones do. The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.

— Don Knuth, *The TEXbook* (1984)

Dans ce chapitre, nous décrivons la conception du service de gestion de segments. Nous commençons par expliciter les interfaces entre ce service et les autres modules d'Arias, puis nous précisons le problème de l'identification des segments. Nous analysons ensuite diverses solutions classiques au problème de localisation dans un grand espace. Les insuffisances de ces solutions justifient la structuration de la fonction de localisation en trois niveaux. Cette structure permet en outre un algorithme d'allocation distante qui apporte de nombreux avantages. Après un aperçu général qui fait la synthèse des sections précédentes, nous traitons deux questions annexes : l'interface d'administration et l'interaction avec les modules de protection et de couplage.

II-1 Interfaces

Cette section traite des deux interfaces principales :

- celle qui permet aux applications de créer et détruire des segments, et
- celle qui offre le service de localisation au module de cohérence.

II-1.1 Interface applicative

L'interface applicative qui nous intéresse se compose essentiellement des appels explicites de création et de destruction de segments (cf. section I-4.3), ainsi qu'un appel de

demande d'identification. Ces trois appels sont intégrés dans la bibliothèque de programmation que les applications doivent incorporer pour utiliser Arias.

II-1.1-a Les adresses virtuelles

Nous avons choisi de ne jamais réutiliser les adresses virtuelles. Imaginons en effet qu'un segment ait été détruit, et qu'une partie de la plage d'adresses correspondante ait été réattribuée à un nouveau segment. Le risque existe alors qu'une applications mal programmée tente d'accéder à l'ancien segment en utilisant une adresse qui a été réaffectée au nouveau segment. L'action correcte serait de signaler à l'application un accès périmé, mais le système n'a pas de moyen de détecter cette péremption, et donnera l'accès au nouveau segment. Si les adresses virtuelles ne sont jamais réutilisées, ce même accès périmé peut être détecté comme tel et le système peut réagir correctement.

II-1.1-b SEG_Create (taille, partition)

À l'instar de la fonction `malloc` de la bibliothèque standard du langage C, cette fonction prend en paramètre **taille** un nombre entier et se charge de créer un segment de cette taille, retournant son adresse de début si la création est possible, ou une valeur nulle sinon. La taille du segment est limitée à un maximum dicté par le système sous-jacent et par d'autres contraintes de mise en œuvre qui sont explicitées plus loin. Inversement, il est interdit de demander la création d'un segment de taille nulle, car cela n'aurait pas de sens. Les applications sont tenues de contrôler que la création s'est bien déroulée en vérifiant que la valeur retournée n'est pas nulle.

Le paramètre **partition** est facultatif. Il spécifie la préférence de l'application pour une partition donnée. Nous détaillons plus loin le rôle des partitions (II-4.3) et l'impact de ce paramètre sur l'algorithme d'allocation (II-5).

II-1.1-c SEG_Destroy (adresse)

La destruction d'un segment est commandée par l'appel à cette fonction, dont l'utilisation est similaire à celle de `free`. L'**adresse** spécifiée n'a pas nécessairement besoin d'être l'adresse de début du segment.

Comme on l'a déjà noté en section I-4.4 et comme il est précisé en section III-3, la destruction d'un segment est conditionnée par des droits particuliers[6]. Les applications peuvent contrôler la destruction effective en examinant le code d'erreur renvoyé par l'appel.

[6] Cela dans l'hypothèse où le système de protection est activé dans Arias ; la présence du module de protection est facultative, pour ne pas désavantager les applications qui n'en ont pas besoin (I-4.4).

II-1.1-d SEG_Find (adresse, début, taille)

À partir d'une **adresse** virtuelle, cet appel identifie le segment auquel elle correspond, et renvoie son adresse de **début** et sa **taille** à l'application. Cet appel donne aux applications un moyen fiable d'identifier l'étendue des segments et d'éviter des dépassements[7].

II-1.2 Interface CGC — LOC

L'interface entre LOC et CGC se compose de l'identification du segment et de la localisation de son descripteur d'une part, et d'autre part de la création d'un nouveau segment et du descripteur associé. À cause du lien très intime entre ces deux modules, l'interface est constituée, outre de messages STREAMS, d'un petit nombre de fonctions directement exportées d'un module à l'autre.

II-1.2-a Structure de la CGC

Le fonctionnement de l'interface avec la CGC est assez complexe, car il existe une interaction forte entre les fonctionnalités de ce module et celles du localisateur. Essentiellement, cette interaction a pour objectif la localisation des méta-données associées à un segment. Ces méta-données sont regroupées dans le descripteur de segment, dont la structure interne est définie par la CGC (cf. section I-4.6) et inconnue du gestionnaire de segments. Ce dernier se contente de gérer l'existence et la localisation des descripteurs.

Pour arbitrer l'attribution des droits d'accès aux zones, la CGC gère pour chacune d'elles un unique *site maître* responsable de la sérialisation et éventuellement de la mise en cohérence des copies distantes. Une zone a un unique site maître à tout moment, mais le site maître d'une zone peut décider de transférer la maîtrise à un autre site, à sa discrétion.

L'enjeu est alors, pour une zone donnée, d'en trouver le site maître. Cela est fait par l'intermédiaire du *descripteur de segment*. Cette structure référence l'ensemble des zones contenues dans un segment et les sites maîtres correspondants. Le descripteur lui-même est en fait répliqué dans des *caches*, dont la cohérence est à son tour assurée par le *site primaire* du segment. La situation d'une zone dans un segment et la relation entre le site maître de la zone, le descripteur de segment et le site primaire du segment sont résumées dans la figure II-1.

Une des hypothèses de base concernant la cohérence de la mémoire est qu'elle est toujours couplée à un modèle de synchronisation d'accès [66]. Cette hypothèse est généralisable à tous les modèles de cohérence relâchée. L'architecture du système de contrôle d'accès et de mise en cohérence a été conçue pour factoriser dans la CGC les fonctions universellement utiles pour tout type de cohérence. La CGC est un support de programmation des *modules spécifiques de cohérence*, qui fournissent un modèle de

[7] Une application particulière a besoin de cette fonction : le chargeur de programme, qui édite les liens en cours d'exécution.

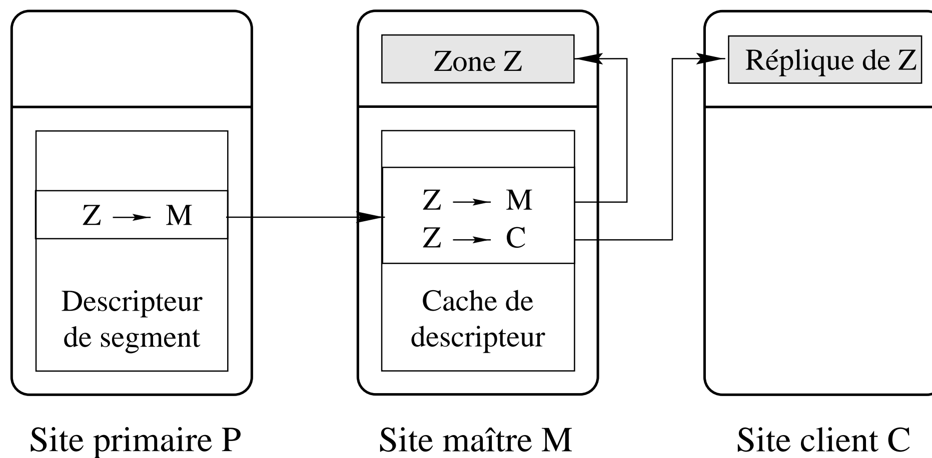

Figure II-1 : Relation entre une zone, son site maître, le segment et son site primaire

synchronisation et un protocole de mise en cohérence associé. La programmation d'un module spécifique est débarrassé des problèmes de localisation et peut se concentrer sur le protocole lui-même. De plus, il est possible d'avoir plusieurs modules spécifiques simultanément dans le même système, utilisant le même support CGC. Les applications choisissent d'utiliser l'une des interfaces proposées par les modules spécifiques selon leurs besoins et selon le type d'accès aux données.

II-1.2-b Le descripteur de segment

Le descripteur de segment existe en un exemplaire unique sur le site appelé *site primaire* du segment. Il contient le détail des zones utilisées dans ce segment ainsi que la localisation de ces zones éparpillées sur différents sites. La CGC fait appel au descripteur pour gérer l'association entre les zones et les sites utilisateurs, et surtout pour déterminer le *site maître* responsable du protocole de synchronisation et de mise en cohérence relatif à la zone.

Un segment est dit *local* à un site si ce dernier est son site primaire.

II-1.2-c Le cache de descripteur

Pour optimiser son fonctionnement, la CGC maintient un *cache de descripteur* sur tous les sites non primaires qui utilisent des zones d'un segment. La cohérence du cache est assurée par la CGC mais la connaissance de son existence réside dans le module de localisation. Le cache de descripteur sur un site donné contient toutes les informations relatives aux zones utilisées, c'est-à-dire les zones pour lesquels ce site détient des droits d'accès ou pour lesquels ce site est le site maître.

La création d'un nouveau segment entraîne celle du descripteur correspondant sur le site dont la création est originaire, ce site devenant automatiquement le site primaire.

Lorsqu'une demande de localisation provient d'un autre site, le module de localisation est chargé de provoquer la création du cache propre à ce site utilisateur, et d'en notifier la CGC. Par la suite, toutes les demandes de localisation émanant de ce site sont satisfaites par le cache. Si ce dernier ne contient pas les informations pertinentes à une zone en particulier, c'est la CGC qui se charge de les obtenir du site primaire, dont l'identité est stockée dans le cache lui-même au moment de sa création.

II-1.2-d Localisation de zone

Dans la situation en «régime stationnaire», l'opération de localisation de zone est matérialisée par une requête d'acheminement de message provenant de la CGC en direction du module de localisation. Cette requête peut prendre plusieurs formes.

- La requête doit rester locale : la réponse contient les références du cache local s'il existe (ou du descripteur du segment s'il réside localement), ou un code d'échec dans le cas contraire. En aucun cas la recherche ne doit émettre de requête en-dehors du site.
- La requête doit aller au site primaire : si le site présent est le site primaire, ce message est redirigé localement. Dans le cas contraire, la requête est acheminée au site primaire. De plus, s'il n'existe pas encore, un cache local est créé pour le segment concerné.

II-1.2-e Migration de segment

Pour favoriser la localité des méta-données par rapport à l'utilisation de la mémoire répartie, il peut être judicieux sous certaines conditions de transporter la totalité du descripteur d'un site à un autre, c'est-à-dire de changer de site primaire. Nous appelons cette opération la *migration de segment*.

Le point crucial d'une migration est celui où l'ancien site primaire devient un site client et où le nouveau site primaire prend en charge la totalité du descripteur. La migration d'un segment nécessite en fait plusieurs étapes.

1. Gel du descripteur : les modifications sont interdites.
2. Transfert du contenu du descripteur.
3. Fusion entre le descripteur et le cache du nouveau site primaire.
4. Transformation du descripteur en cache sur l'ancien site primaire.
5. Mise à jour des tables de localisation.
6. Dégel du descripteur.

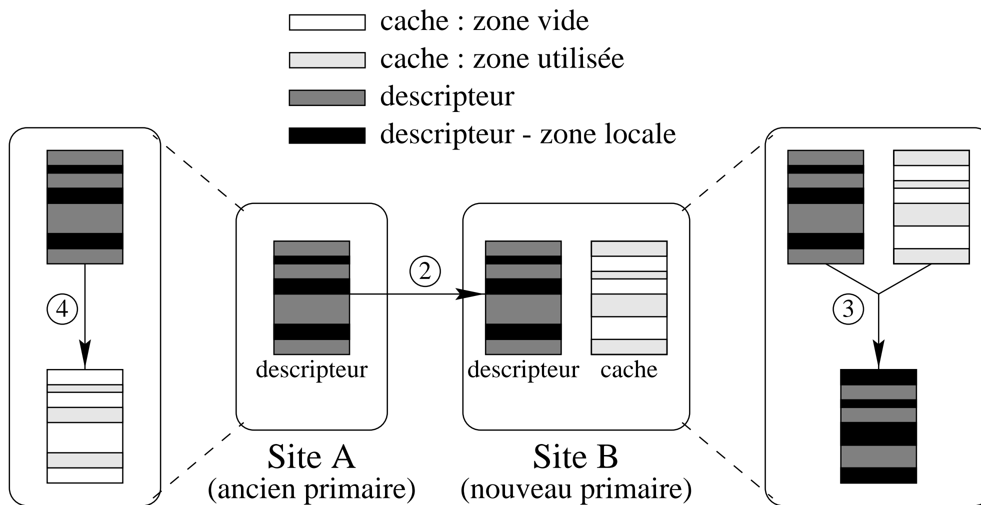

Figure II-2 : Migration de segment. Étape 2 : transfert ; étape 3 : fusion sur B ; étape 4 : transformation sur A.

La figure II-2 montre les étapes 2 à 4 de la migration d'un descripteur de segment. Le site A est l'ancien site primaire, le site B le nouveau. Ce dernier possède avant la migration un cache de descripteur où sont matérialisés en gris clair les zones référencées. Sur le site A, le descripteur (gris foncé) contient des zones en noir qui représentent les zones maîtrisées par A. Après le transfert, le site B fusionne son cache et la copie du descripteur pour former le nouveau descripteur. Sur le site A, l'ancien descripteur est transformé en cache en ne gardant que les zones maîtrisées par A.

Durant le déroulement de ces opérations, il est impossible de modifier le descripteur. Les consultations restent cependant permises sauf pendant les étapes de fusion et de transformation (étapes 3 et 4).

Le placement du descripteur de segment est déterminé par Arias. Idéalement, le site primaire du segment doit être celui sur lequel il y a le plus d'activité sur ce segment. À l'opposé d'une zone, un segment est un objet de grande taille, nécessitant une grande quantité de méta-données. La migration d'un segment (c'est-à-dire le changement de site primaire) implique donc un transfert coûteux. L'initiative de la migration est donc une décision délicate, d'autant plus qu'il est impossible d'anticiper les besoins de l'application.

Les seuls critères impératifs de migration sont :

- lorsqu'un segment n'est plus utilisé par aucune application sur le site primaire, ou
- par demande explicite de l'administrateur système (II-7).

II-1.2-f Destruction de segment

Comme esquissé un peu plus haut, la création d'un nouveau segment provoque des interactions entre la CGC et le module de localisation. Des interactions similaires ont lieu au moment de la destruction. Le localisateur doit en effet coordonner la destruction de toutes les structures associées au segment.

1. Notification du site primaire et marquage du descripteur ; toute tentative de modification du descripteur échoue.
2. Destruction des structures associées aux droits d'accès aux zones en utilisation (dans les modules spécifiques appropriés).
3. Destruction du cache local.
4. Marquage des structures de localisation ; le segment devient inaccessible.
5. Destruction du descripteur sur le site primaire.

II-2 Identification des segments : la fiche

Pour satisfaire aux besoins des interfaces décrites ci-dessus, chaque segment est identifié par une petite structure de données appelée sa *fiche*, qui regroupe essentiellement trois informations :

- l'adresse de début du segment,
- sa taille,
- l'adresse du descripteur de segment.

La fonction de localisation consiste simplement à retrouver cette structure pour un segment spécifié par une adresse virtuelle. Un des problèmes majeurs est que cette adresse virtuelle peut se trouver n'importe où à l'intérieur du segment qu'il désigne, puisqu'il n'existe pas de fonction simple qui permet de trouver l'adresse de début du segment à partir de l'adresse fournie.

II-3 La localisation dans un grand espace

Chaque accès synchronisé, chaque couplage, nécessite une localisation préalable. Cette dernière est une fonction très fréquemment appelée, et doit donc être aussi efficace que possible. Plusieurs difficultés se combinent pour rendre la tâche délicate.

- L'espace d'adressage est vaste ; il faut décoder une adresse plus grande.

- Les objets sont éparpillés dans cet espace ; les structures linéaires sont inadéquates.
- La mémoire est répartie ; les données demandées peuvent être distantes.
- Les méta-données sont mobiles ; elles peuvent aussi être distantes.

Il est crucial d'éviter les deux grandes sources de retard : les requêtes distantes, à cause de la latence du réseau de communication, et les accès à la mémoire secondaire, à cause de la latence des accès disques.

Le second écueil est facilement évité en forçant les méta-données à résider en mémoire primaire (la RAM physique). Mais cette solution introduit une contrainte supplémentaire : pour éviter de consommer des quantités déraisonnables de mémoire primaire, les méta-données doivent rester compactes.

En ce qui concerne les communications par le réseau, il est possible, en mémorisant sur chaque site les informations de localisation, de limiter les requêtes distantes au minimum. Il est important de trouver un bon équilibre entre le nombre de requêtes, pour réduire la latence, et la quantité de données véhiculées au cours de chaque requête, pour réduire l'utilisation de la bande passante. Cet équilibre peut parfois être déterminé par une analyse qualitative, ce qui est fait au cours du chapitre III. Dans d'autres cas, seule une expérimentation permet de choisir un bon compromis, ce qui fait l'objet du chapitre IV.

Cette section passe en revue deux des solutions proposées dans la littérature pour la localisation de pages dans un grand espace d'adressage. Ces solutions, quoique possédant des avantages certains, ont des inconvénients qui les rendent inadéquates pour Arias.

II-3.1 Tables de pages hiérarchiques

La structure des tables de pages hiérarchiques, mise en œuvre dans les systèmes Unix modernes tels que AIX sur PowerPC [55], suppose que les pages effectivement utilisées sont distribuées dans l'espace d'adressage de manière globalement éparse mais localement dense. Cette hypothèse est concomitante à l'emploi de segments qui sont des plages d'adresses s'étalant sur un grand nombre de pages. La figure II-3 donne une idée de la manière dont les pages d'un segment (pages grises) sont référencées par deux niveaux de tables de pages linéaires. Dans ce cas de figure, le segment est entièrement référençable à partir d'une entrée dans la table de niveau 3. Il est typique de trouver à ce niveau non plus des tables linéaires mais des tables de hachage. Cela donne une structure générale illustrée sur la figure II-4.

Une optimisation supplémentaire non négligeable est l'utilisation des tables intermédiaires de taille égale à une page. Cela permet de maximiser le taux d'utilisation de ces tables et de simplifier la gestion de la mémoire dédiée aux méta-données. La traversée de ces tables devient très rapide. De plus, l'utilisation d'une taille fixe pour les tables intermédiaires rend abordable une solution où ces tables elles-mêmes sont

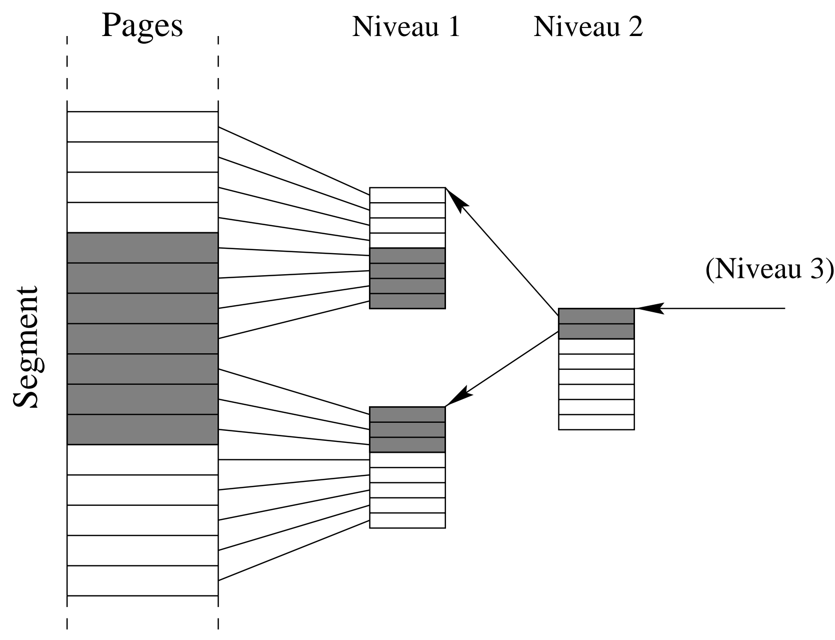

Figure II-3 : Table de pages référençant un segment

stockées en mémoire virtuelle banalisée (≪*swappable*≫), ce qui réduit de beaucoup le coût en mémoire physique et rend acceptable un taux d'utilisation plus faible.

Par ailleurs dans AIX, les tables de niveau 1 sont organisées en tables inverses, c'est-à-dire qu'il s'agit d'une liste des pages utilisées, et non une table directement indexée par le numéro de page. Une telle liste est beaucoup plus dense, dans ce sens que seules les pages réellement existantes utilisent une entrée dans la liste. Cependant, l'opération de recherche consiste à parcourir les entrées de la liste en séquence jusqu'à trouver la page recherchée, et la liste doit donc être courte. En fait, sur les PowerPC, les 8 entrées de la liste sont examinées en parallèle par un décodeur spécialisé du processeur.

La technique des tables de pages hiérarchique est assez bien adaptée pour la recherche de pages dans une mémoire organisée en segments de grande taille (épars à grande échelle mais dense à moyenne échelle). Mais cela ne correspond pas à la structure de la mémoire partagée d'Arias. Cette dernière est en effet composée d'un grand nombre de segments de tailles très variables. L'effet d'éparpillement est plus radical.

De plus, le module de localisation a besoin de référencer non pas des pages, mais des segments de taille variable[8]. Partant d'une adresse quelconque à l'intérieur d'un segment, la fonction de hachage doit donner le même résultat quelle que soit la page du segment dans laquelle se trouve cette adresse. À défaut de cela, un segment peut être représenté par plusieurs entrée dans la table de hachage, ce qui complique les mises à jour. Et même dans ce cas, il n'est pas possible de garantir une borne au nombre

[8]Deux segments ne peuvent pas utiliser la même page ; autrement dit les segments sont alignés sur les pages.

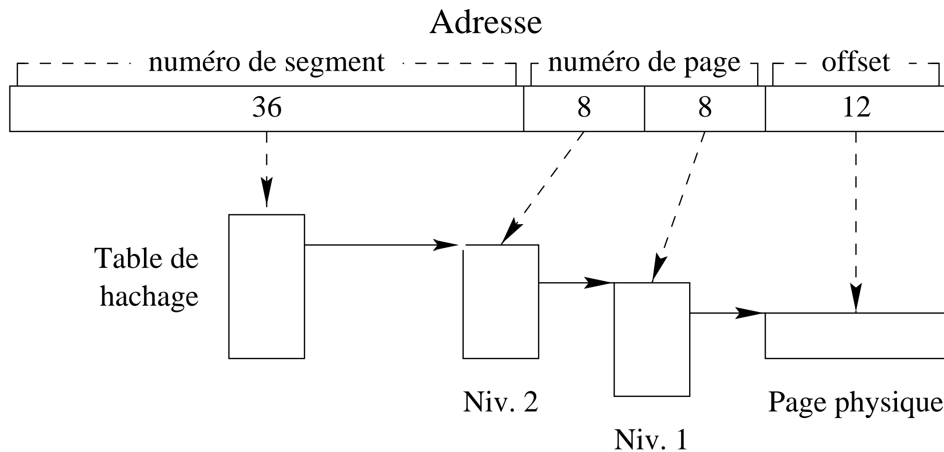

Figure II-4 : Structure générale d'une table de pages hiérarchiques

d'entrées utilisées pour chaque segment.

OSF/1 sur Alpha [76] permet d'utiliser en même temps plusieurs tailles de pages sur des régions distinctes de l'espace d'adressage. On pourrait penser que cette technique est utilisable pour résoudre le problème des segments de tailles différentes. Ce n'est pas le cas. D'abord, la solution d'OSF/1 s'appuie sur le fait que les pages ont des tailles qui sont des puissances de 2, alors que les segments peuvent difficilement être soumis à la même contrainte. Ensuite, la taille des pages est un paramètre qui s'applique à toute une région de l'espace d'adressage. On ne peut avoir de panachage fin des tailles de pages. Pour reprendre l'illustration de la figure II-3, la taille des pages est codée dans la table de niveau 3, et s'applique à toutes les pages référencées par cette table. Une autre table du même niveau 3 peut utiliser une taille de pages différentes, mais ces deux tables référencent des régions disjointes de l'espace d'adressage.

II-3.2 Tables de pages gardées

Proposée par Jochen Liedtke [53] et adoptée par le projet Mungi [36], la technique des tables de pages gardées se propose de résoudre à la fois le problème des tailles de pages et leur dispersion. Cette technique consiste à décoder les adresse virtuelles par petits blocs, en utilisant des ≪gardes≫ pour pouvoir éventuellement décoder des bits supplémentaires à chaque étape. On progresse ainsi en partant des bits de poids fort et en traversant les niveaux successifs de tables linéaires de petite taille (on n'imposent pas, contrairement aux tables de pages hiérarchiques, qu'elles soient de taille fixe égale à celle d'une page physique).

Partons du décodage normal de la séquence `111011` par trois niveaux de tables à quatre entrées. La figure II-5 illustre les trois niveaux de tables (*a* à *c*) et les bits décodés à chaque étape. Dans cet exemple, nous imaginons en plus que chaque table n'a qu'une seule entrée utilisée. De ce fait, les tables intermédiaires sont redondantes : il n'y a pas

lieu de les consulter, puisqu'elles ne contiennent qu'un seul chemin de décodage.

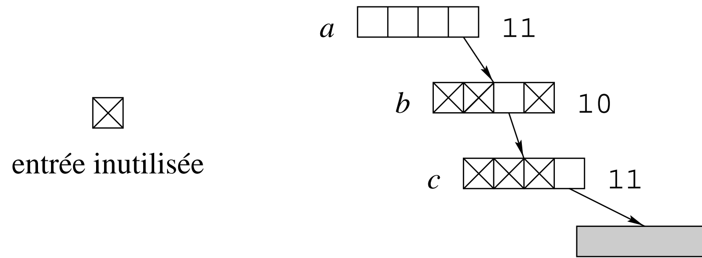

Figure II-5 : Exemple de décodage avec des tables de pages linéaires

La figure II-6 illustre le processus de décodage de ces mêmes six bits d'adresse (`111011`) en une seule étape au lieu de trois. La garde est possible parce que les tables intermédiaires ne contiennent qu'une seule entrée active. Après avoir décodé le premier `11`, la garde indique que les quatre bits suivants sont inévitablement `1011`. L'entrée pointe directement vers la donnée finale, comme si l'on avait traversé le chemin en pointillés.

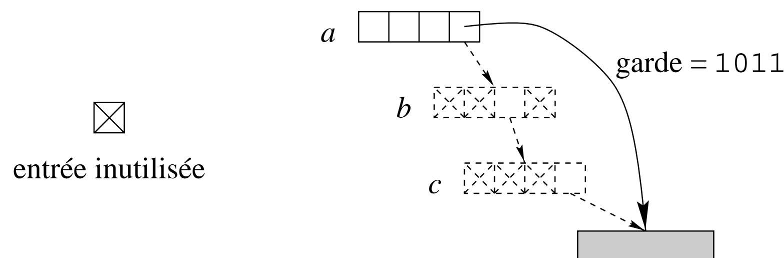

Figure II-6 : Décodage avec des tables de pages gardées

Liedtke démontre que l'utilisation des gardes dans une hiérarchie de tables linéaires permet de borner à la fois la taille des structures et le temps de traversée. Cependant, ces théorèmes sont en pratique peu intéressants, car ils concernent les cas les plus défavorables et les bornes supérieures sont élevées. Cela ne prouve donc pas grand-chose sur les cas courants.

Une démonstration supplémentaire de l'efficacité des pages de tables gardées est faite dans son utilisation dans le cadre du projet Mungi. En utilisant à fond les caractéristiques du processeur Mips R4400, Liedtke et Elphinstone [54] ont montré qu'il est possible d'implémenter l'algorithme de décodage d'adresse en très peu d'instructions. Mais cette implémentation est destinée à la recherche de pages (à nouveau de taille fixe) et se propose comme solution de remplacement pour les tables de pages hiérarchiques. Or, nous savons que le cahier des charges des tables de pages hiérarchiques ne nous convient pas.

II-4 Conception du système en trois niveaux

En suivant les préceptes de l'épigraphe de ce chapitre [44], l'exposé de cette section suit une structure par couches successives. Nous commençons par présenter la technique de décodage d'adresse en trois étapes, puis nous affinons cette solution en intégrant les accélérateurs de localisation.

II-4.1 Tables de hachage hiérarchiques

La technique de hachage d'adresse [43], nous intéresse pour plusieurs raisons.

- La recherche est rapide : le temps de recherche est indépendant du nombre absolu d'éléments[9].
- Son faible coût en mémoire par rapport à des solutions linéaires.
- Son bon comportement face à la fragmentation : le profil de distribution des clés n'a pas d'influence directe sur le temps de recherche.

Cela nous pousse à l'utiliser au moins à certains niveaux où ces avantages sont importants vis-à-vis de ses inconvénients, qui sont :

- ses éléments terminaux qui doivent être de taille fixe,
- le surcoût du rehachage et du traitement des collisions.

Nous utilisons donc une technique de hachage pour décoder un certain nombre de bits d'adresse de poids forts. Lorsque l'on utilise une table de hachage, il est important de trouver un bon compromis entre le risque de collision et la taille de la table. La solution la plus sophistiquée consiste à utiliser une table dont la taille varie dynamiquement en fonction de son taux de remplissage. Lorsque ce taux dépasse un certain seuil, la table est agrandie ; lorsqu'il passe en-dessous d'un autre seuil, elle est rétrécie. Pour intéressante qu'elle soit pour l'optimisation du taux de remplissage, il faut noter qu'un changement de taille de la table entraîne un rehachage complet de tous les éléments référencés. L'opération de rehachage est coûteuse non seulement en temps mais aussi en mémoire : il est nécessaire au cours du rehachage de garder temporairement deux tables en mémoire.

Pour éviter le rehachage complet, nous utilisons une hiérarchie de tables. Le premier niveau est de taille fixe, relativement petite, et utilise une fonction f. Du fait de sa petite taille, de nombreuses collisions sont attendues. Les collisions sont alors gérées par un second niveau de tables de hachage, dont la fonction de hachage g est conçue pour s'éloigner le plus possible de f, pour éviter que les collisions au premier niveau entraîne systématiquement des collisions au second niveau, ce qui annulerait tout le

[9] Mais il dépend du taux de remplissage.

bénéfice de la table secondaire. Les inévitables collisions au second niveau sont gérées à l'intérieur même de la table, en utilisant un itérateur. La structure globale est illustrée sur la figure II-7.

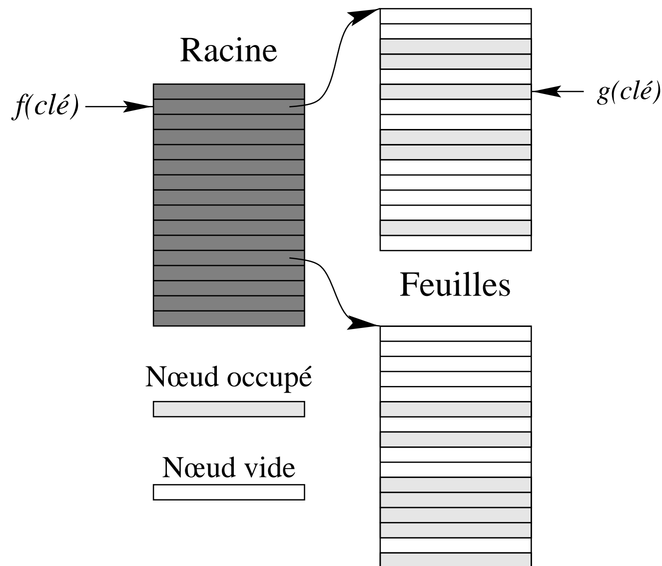

Figure II-7 : Tables de hachage hiérarchiques à deux niveaux, avec fonctions de hachage f et g

Avec cette hiérarchie, nous minimisons un des désavantages du hachage, qui est le coût du rehachage. Il reste encore un problème. La clé que nous utilisons pour calculer les fonctions f et g est une adresse virtuelle quelconque du segment recherché. Si nous voulons avoir une seule entrée par segment, cela signifie que les fonctions f et g doivent fournir toujours la même valeur quelle que soit l'adresse clé à l'intérieur d'un même segment, et ce quelle que soit la taille du segment.

Nous résolvons ce problème en introduisant un niveau d'organisation intermédiaire, le bloc, que nous décrivons ci-dessous.

II-4.2 Blocs

Notre solution repose sur l'utilisation d'une entité de taille fixe, que nous appelons le *bloc*. Un bloc représente une plage de mémoire relativement vaste, car il doit être au moins aussi grand que la taille maximale des segments. Désignons par n le nombre de bits d'adresse qui correspond à cette plage (par exemple, $n = 28$ pour une plage de 256 Mo). Il reste alors $64 - n$ bits à faire décoder par les tables de hachage. Une valeur de n plus petite donne plus d'importance à ces tables, qui sont justement capables de décoder un grand nombre de bits en un nombre fixe d'opérations.

II-4.2-a Buddy-system

À l'intérieur d'un bloc, nous utilisons un technique de *buddy-system* [1, §12.1] pour référencer les segments. La variante la plus simple de cette technique est le système binaire, illustré sur la figure II-8, qui met en regard l'occupation de l'espace d'adressage et la structure de données qui référence cette occupation.

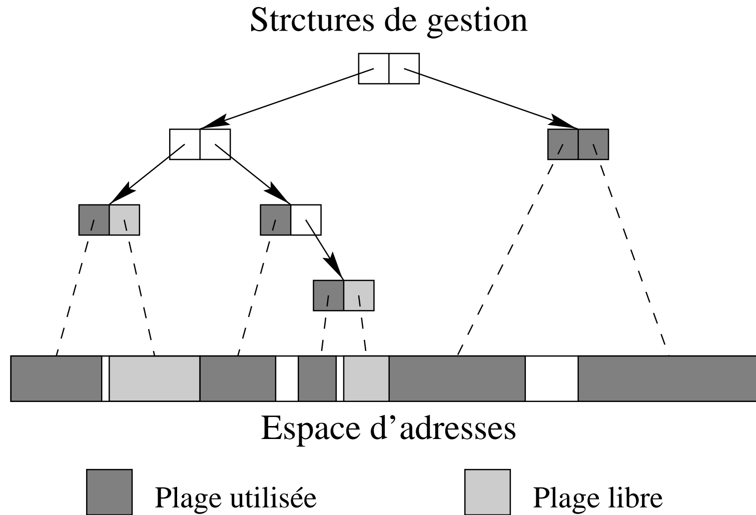

Figure II-8 : Buddy-system binaire

L'avantage du système binaire est l'extrême compacité des données de référence. Mais son inconvénient principal est le faible taux d'utilisation de l'espace d'adressage. En effet, tous les objets doivent être alignés sur une adresse multiple de la puissance de 2 supérieure. En d'autres termes, si la taille de l'objet est $2^n + p$ (avec $p < 2^n$), alors il doit commencer sur une adresse a qui soit un multiple de $2^{(n+1)}$, et un espace de $2^n - p$ est gâché. Sur la figure II-9, par exemple, l'objet représenté ne peut avoir de compagnon dans la tranche d'adresses de $2^{(n+1)}$ octets qu'il occupe. Il faudrait pour cela que son compagnon soit à l'adresse $a + 2^n$, or cette adresse est occupée par lui-même.

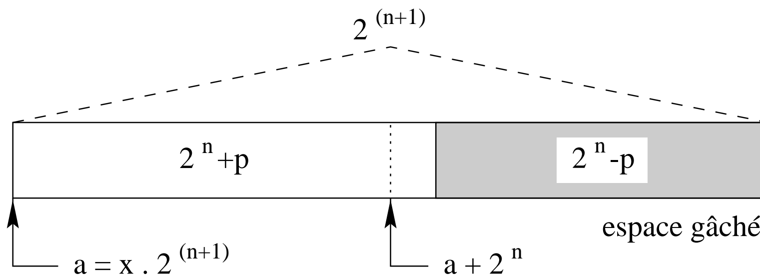

Figure II-9 : Gâchis d'espace d'adressage dans un buddy-system binaire

II-4.2-b Buddy-system non binaire

Il est possible d'éviter ce gâchis dans une certaine mesure en utilisant un buddy-system non binaire. La figure II-10 illustre un système de base 16. Chaque niveau contient 16 entrées, qui peuvent elles-mêmes pointer vers une nouvelle table de 16 entrées. Un élément de cette table peut occuper de 1 à 15 entrées[10]. L'élément «X» par exemple comprend une plage de 9 entrées, c'est-à-dire que le segment correspondant a une taille qui est 9 fois la taille de base de ce niveau de table.

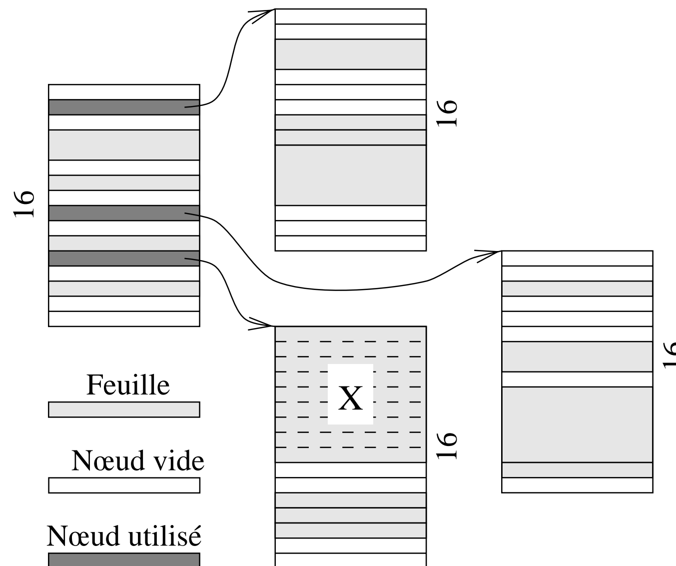

Figure II-10 : Buddy-system de base 16

Cette structure permet de mieux remplir l'espace d'adressage. En reprenant l'exemple du gâchis illustré plus haut, prenons une table alignée sur une adresse multiple de 2^n et un segment dont la taille est de $\frac{11}{16}2^n = 11.2^{(n-4)}$ (cf. figure II-11). L'espace restant de taille $5.2^{(n-4)}$ reste utilisable, puisqu'il est représenté par 5 entrées libres dans la tables. Dans un buddy-system binaire, cet espace aurait été gâché.

L'allocation dans un tel système est un peu plus complexe que pour le système binaire. En effet, comme l'exemple du segment de la figure II-11, un segment peut occuper plusieurs cases d'une même table (sept dans l'exemple). Cela signifie que les onze cases doivent référencer le même segment. Cette fois-ci, nous gâchons de l'espace dans les structures de référence pour mieux utiliser l'espace d'adresses.

[10] Un unique élément occupant les 16 entrées n'aurait pas de sens : cet élément aurait dû être représenté par une entrée au niveau immédiatement supérieur.

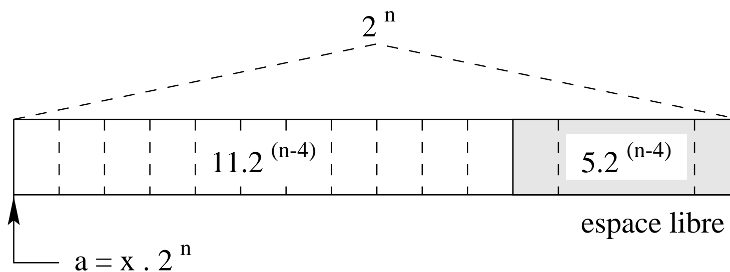

Figure II-11 : Une table de 16 entrées dont 11 occupées et 5 libres.

II-4.2-c Réduction du bloc

Une particularité d'Arias est le fait qu'il ne réutilise jamais une adresse. Si un segment est alloué à une adresse, il occupe cette adresse jusqu'à sa destruction, et à ce moment l'espace qu'il occupait reste à jamais inutilisé. Une conséquence est qu'une fois le bloc rempli (c'est-à-dire que l'on ne peut plus allouer de nouveaux segments dedans en utilisant une stratégie d'allocation donnée), les tables de références des segments ne peuvent que se rétrécir à mesure que les segments qu'il contient sont détruits. Or, un buddy-system binaire se prête bien à la destruction de références : quand un objet et son compagnon sont détruits, l'espace qu'ils occupaient ensemble est détruit en tant qu'objet de la taille supérieure (figure II-12). Cette opération est aussi possible pour un buddy-system non binaire.

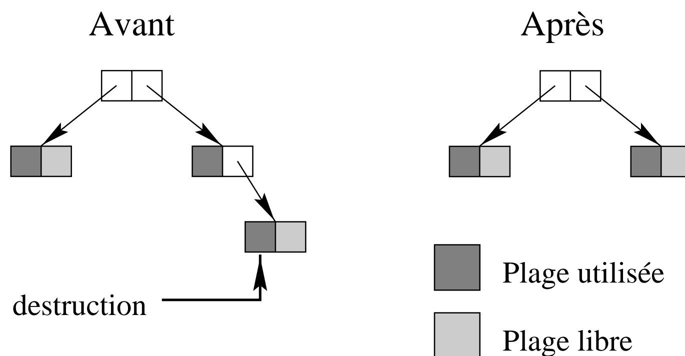

Figure II-12 : Destruction d'un élément dans un buddy-system binaire

Dans le cas du buddy-system non binaire, il est possible de rendre la table de références encore plus compacte en utilisant des indicateurs apparentés aux gardes des tables de pages gardées (section II-3.2). La figure II-13 montre un exemple de système de base 16 où une entrée dans une table de niveau n référence directement un segment de taille inférieure à $2^{(n-4)}$. La table de niveau $n-4$ (représentée en pontillés) est superflue car elle ne contient qu'une seule entrée active, la numéro 5. En codant ce numéro dans l'entrée de la table de niveau n, on peut «court-circuiter» la table intermédiaire.

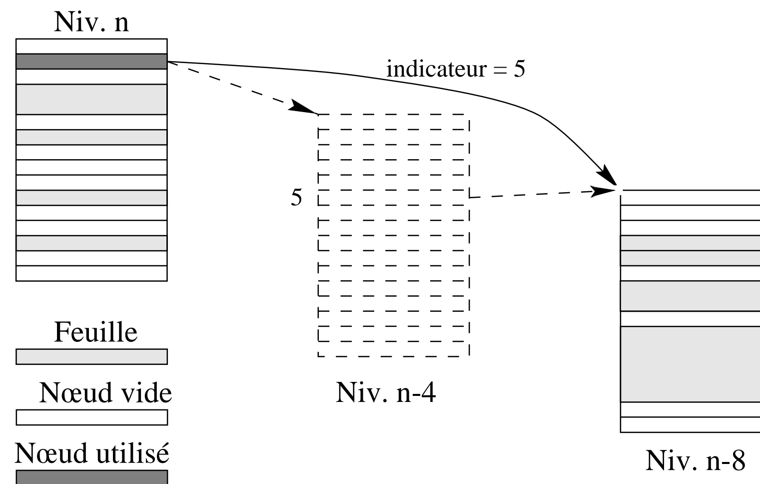

Figure II-13 : Compactage d'un buddy-system de base 16

II-4.3 Partitionnement

Le système composé d'une hiérarchie de tables de hachage et d'un ensemble de blocs de buddy-system de base 16 résout le problème de la localisation de segments sur un système centralisé. Il est tout à fait possible de réaliser un système centralisé où les données réparties sont référencées par un unique ≪serveur≫ qui posséderait la connaissance de tous les segments existants. Mais notre ambition est de répartir les méta-données autant que les données elles-mêmes.

Pour cela, outre la mobilité des segments, nous introduisons un niveau supplémentaire de découpage de l'espace d'adressage global, que nous appelons la *partition*. Une partition représente un grand pan de l'espace global et est attribuée à une machine donnée. La vocation du partitionnement est de répartir la responsabilité des connaissances de localisation sur l'ensemble des machines participant au réseau. Plutôt que de charger un unique ≪serveur≫ de centraliser la connaissance sur la localisation de tous les segments de l'espace global, chaque machine a la responsabilité d'agir en tant que ≪serveur≫ pour les segments contenus dans la partition qui lui attribuée.

Formellement, une partition est une plage d'adresses de taille fixée dans le système. Chaque partition possède un unique *site localisateur*, qui est la machine responsable de la gestion de la localisation des segments contenus dans cette partition. En fait, une machine donnée peut agir comme site localisateur pour plusieurs partitions. La répartition des attributions dépend des capacités des machines (capacité en mémoire, en puissance de traitement et éventuellement en accès réseau). La figure II-14 montre un exemple de partitionnement sur trois machines. Notez qu'une partie seulement de l'espace global est attribué ; certaines partitions n'ont pas de site localisateur. Cela signifie que ces partitions, dites *inactives*, ne sont pas encore utilisables, mais peuvent le devenir dans le futur en leur attribuant un site localisateur (cf. interface d'administration de partition, section II-7).

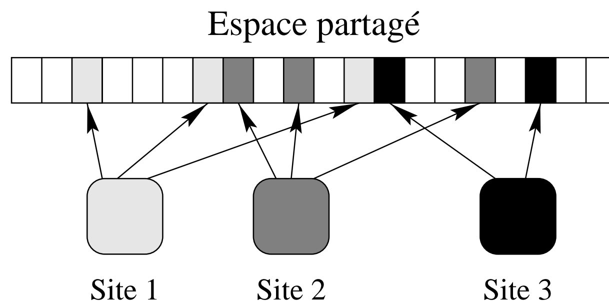

Figure II-14 : Exemple de partitionnement

Les correspondances entre partitions et sites localisateurs sont stockées dans la *table de partition*, qui est une simple table linéaire indexée par quelques bits de poids fort de l'adresse. Le nombre de bits décodés par cette table dépend du nombre de partitions que l'on désire avoir sur un réseau, ce qui à son tour dépend de la géométrie du réseau.

Pour donner un ordre d'idée, imaginons un réseau constitué d'une dizaine de machines, dont une particulièrement puissante, qui sert de serveur de fichiers et d'impression. Certaines applications allouent leurs segments de données dans des partitions gérées par ce serveur, et le service d'impression alloue des segments temporaires dessus. Cela entraîne une plus forte ≪consommation≫ de mémoire virtuelle sur le serveur que sur les autres machines. Par ailleurs, ce réseau est appelé à être étendu par l'adjonction d'une dizaine de nouvelles machines. Ces nouvelles machines arriveront vers le milieu de la durée de vie du système. Au total, on peut estimer que le serveur ≪consomme≫ dix partitions durant l'unité de temps où les autres machines ≪consomment≫ chacune une partition, et les nouvelles machines auront ≪consommé≫ à la fin de la durée de vie du système moitié moins de partitions que les machines d'origine. Si l'on prévoit au départ 50 partitions, la répartition en fin de vie aura la structure de la figure II-15. Pour se donner une marge de manœuvre, on peut multiplier ce chiffre par 4 ou 5, ce qui a pour effet de multiplier tous les nombres d'autant.

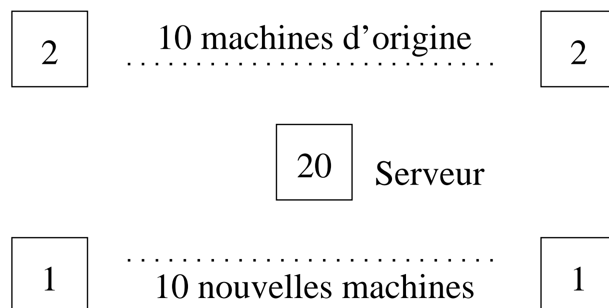

Figure II-15 : Exemple de partition

La table de partition doit être répliquée à l'identique sur toutes les machines, ce qui

nécessite un protocole strict lors des accès en écriture. Mais c'est une structure de données très rarement modifiée. Les ajouts et les modifications n'interviennent a priori que lorsqu'une nouvelle partition jusque-là inactive devient nécessaire. Un protocole d'accès atomique est donc justifiable.

II-4.4 Résumé

Pour résumer l'utilisation des structures décrites ci-dessus, l'algorithme de localisation de segment peut être décomposé comme suit.

1. Consultation de la table de partition $\longrightarrow$ site localisateur.
2. Envoi d'une requête au site localisateur.
3. Le site localisateur :
   (a) consulte les tables de hachage $\longrightarrow$ identité du bloc ;
   (b) traverse le buddy-system décrivant l'occupation du bloc $\longrightarrow$ identité du segment ;
   (c) trouve l'identité du site primaire du segment. C'est la réponse à la requête.

Cet algorithme décrit en fait ce qui se produit au moment où un site doit localiser un segment qu'il n'avait jamais rencontré jusque là. En particulier, si le site courant est déjà le site primaire du segment, il n'a aucun moyen, d'après cet algorithme, de s'en apercevoir sans le demander au site localisateur. De plus, les caches de descripteurs, dont nous avons introduit l'existence au niveau de la CGC (section II-1.2), ne sont pas pris en compte par cet algorithme. Voyons comment nous remédions à cela.

II-4.5 Accélérateurs de localisation

Le rôle des accélérateurs de localisation est justement de fournir l'identité d'un segment avant qu'il soit nécessaire de contacter un site localisateur. Un accélérateur est créé sur un site et intégré à ses structures de localisation pour tout segment

- dont il est le site primaire (le descripteur est sur ce site) ou
- dont il possède un cache de descripteur (la localisation du site primaire s'y trouve).

Un accélérateur est assimilable à une fiche (cf. II-2) pour un segment qui ne fait pas partie des partitions dont le site est localisateur. Ces accélérateurs sont organisés de la même manière que les autres fiches. En fait, c'est la même structure qui contient toutes les données de localisation sur un site ; cette structure est appelée le *fichier du site*.

Dans une fiche (accélérateur ou pas), la localisation du descripteur d'un segment est matérialisée par l'une des trois informations :

- l'adresse du descripteur de segment, s'il est local, ou
- l'adresse du cache local du descripteur, s'il existe, ou encore
- l'identité du site primaire en dernier recours.

Dans le premier cas, le site courant est le site primaire du segment recherché. Les deux autres cas correspondent à un accélérateur (désormais nous ne ferons plus de distinction entre fiche pour segment local et accélérateur). Dans le second cas, le site courant possède encore un cache, ce qui signifie que le segment est en cours d'utilisation, et qu'il est possible qu'un processus ait utilisé un mode de synchronisation sur des zones appartenant au segment. Dans le dernier cas, le segment n'est pas en utilisation courante, mais a été utilisé récemment, et l'accélérateur est encore référencé.

Avec la présence des accélérateurs, l'algorithme de localisation se déroule comme suit :

1. Consulter en premier la table des segments locaux.
2. En cas d'échec, reprendre l'algorithme comme dans la section précédente en repartant de la première étape (consultation de la table de partition).

II-5 L'allocation

L'un des points forts que nous voulons mettre en valeur dans Arias est la possibilité de créer des segments persistants avec un surcoût minimal par rapport à la création d'un segment en mémoire privée non persistante. L'un des problèmes d'autres systèmes persistants est que la persistance est intimement liée au stockage secondaire. En d'autres termes, l'allocation d'un espace persistant nécessite la réservation d'espace sur le stockage secondaire, ce qui induit un surcoût important. Dans Arias, la persistance n'est pas synonyme de stockage secondaire. Seuls certains segments, rendus *permanents* à l'initiative des applications, sont stockés sur un support secondaire pour résister à certains types de pannes localisées. Cela dit, l'autre source de délais est bien sûr l'accès au réseau de communication. Notre système doit donc être capable de faire l'allocation de segments de manière autonome, c'est-à-dire sans accès au réseau ou aux disques.

Or, nous avions introduit dans l'interface applicative (section II-1.1) un paramètre facultatif lors de la création de segment par lequel l'application peut spécifier dans quelle partition le segment doit être créé. Le défi est de pouvoir faire cette allocation même quand la demande de création provient d'un site autre que le site allocateur, qui est le seul maître de la partition demandée.

À travers le paramètre maîtrisant la partition dans laquelle le segment doit être créé, l'application peut spécifier trois types de desiderata :

1. choix d'une partition,
2. choix d'un site, ou

3. choix indifférent.

L'objectif est d'obtenir la maîtrise d'un bloc provenant d'une partition demandée pour pouvoir utiliser l'algorithme d'allocation dans le buddy-system II-4.2) qu'il constitue.

Dans la suite, nous décrivons les moyens par lesquels le système satisfait à ces spécifications en restant autant que possible autonome. Pour simplifier le discours, nous adoptons quelques raccourcis de langage :

site local désigne le site d'où la demande de création émane ;

partition locale désigne une partition pour laquelle le site local est le site localisateur ;

partition distante désigne une partition dont le site localisateur n'est pas le site local.

II-5.1 Choix d'une partition

L'application peut, a priori, demander la création d'un segment dans n'importe quelle partition existante. Pour pouvoir répondre à une telle demande, un site peut *emprunter* un bloc depuis une partition distante. Le principe de l'emprunt consiste :

- pour le site localisateur de la partition, à donner temporairement le droit à un site distant de modifier l'occupation d'un bloc ; et
- pour le site emprunteur, à gérer l'allocation dans ce bloc jusqu'à épuisement, puis de rendre le bloc au site localisateur de la partition dont il provient.

Une fois le bloc rendu, le site localisateur le réintègre dans sa table des segments locaux et reprend le droit exclusif d'en manipuler les données de localisation (c'est-à-dire de répercuter les migrations et les destructions de segments).

Pendant la durée de l'emprunt, le bloc est temporairement géré par l'emprunteur. Si un site tiers a besoin d'identifier des segments dans ce bloc, il passe en premier lieu par le site localisateur d'origine, qui lui indiquera que ce bloc a été emprunté et lui fournira l'identité de l'emprunteur.

Il faut noter que l'action d'emprunter ou de rendre un bloc ne concerne que les données de localisation. La création et la migration des descripteurs des segments qui sont référencés par ces blocs suit les règles normales (II-1.2). Par exemple, le scénario suivant ne pose aucun problème au site U.

- Le site E (emprunteur) emprunte un bloc du site L (localisateur).
- Une application de E crée un segment S dans le bloc.
- Un site U (utilisateur) recherche S, donc contacte L puis E.
- E répond à la requête de U : E est le site primaire de S.

- U stocke la localisation du segment d'après la réponse de E.
- E rend le bloc à L ; E reste le site primaire de S.

Il reste un problème de délai au moment où un site se rend compte qu'il doit emprunter un nouveau bloc. Cela se produit soit parce qu'il ne possède encore aucun bloc provenant de la partition demandée, soit parce qu'il ne reste pas assez de place dans le bloc qu'il a déjà emprunté pour effectuer l'allocation. Dans ce dernier cas, il doit en plus rendre le bloc plein au localisateur.

Ce problème peut être résolu par l'emprunt de deux blocs. Un site emprunteur possède en permanence deux blocs empruntés : l'un courant dans lequel les allocations se déroulent, l'autre de réserve dans le cas ou le premier viendrait à être plein. Il peut alors commencer immédiatement à utiliser le bloc de réserve. Pendant ce temps, il peut envoyer un message asynchrone pour rendre le bloc utilisé et en emprunter un nouveau, qui devient le nouveau bloc de réserve.

Cette solution pose toutefois de nombreux problèmes d'implémentation qui seront détaillés en III-2.3, de même qu'elle nécessite deux blocs empruntés par partition distante. Le nombre de blocs empruntés devient important.

II-5.2 Choix d'un site

Nous avons vu comment se déroule l'allocation lorsque l'on sait exactement dans quelle partition il faut créer le nouveau segment. Si l'application spécifie uniquement un site privilégié, il faut en plus procéder au choix d'une partition parmi celles qui sont gérées par le site demandé. Pour commencer, il est facile d'obtenir la liste des partitions gérées par un site donné. Il suffit pour cela de consulter la table de partition, non pas directement indexé par partition, mais en extrayant les partitions par site localisateur.

Une fois constituée la liste des partitions candidates, l'application de deux règles simples suffit à faire un choix :

- s'il existe déjà localement un bloc emprunté provenant d'une des partitions candidates, on utilise cette partition ;
- sinon, le plus simple est de choisir la partition d'adresse la plus basse.

On peut trouver plusieurs manières de choisir une partition en particulier en fonction de leur taux de remplissage ou leur taux d'utilisation, le nombre de blocs qu'elle exporte, etc. Aucun algorithme n'apporte un avantage vraiment important. En privilégiant simplement les adresses les plus basses, on encourage le remplissage des partitions par ordre d'adresses. On peut jouer finement avec la fonction de hachage (cf. III-1.2) pour optimiser alors le taux d'utilisation des tables de hachage.

II-5.3 Partition indifférente

Dans le cas où l'application n'a pas de préférence particulière, le plus simple est de créer le segment sur une des partitions locales, toujours en privilégiant les adresses basses. En pratique, nous supposons que chaque machine possède au moins une partition locale. Cela peut ne pas être toujours le cas.

Même s'il existe une partition locale, on peut discuter de l'avantage à allouer sur une partition distante, par exemple sur une partition dont le site localisateur est particulièrement performant. La recherche d'une bonne répartition des charges semble introduire des solutions trop complexes pour en valoir la peine.

II-6 Vue d'ensemble

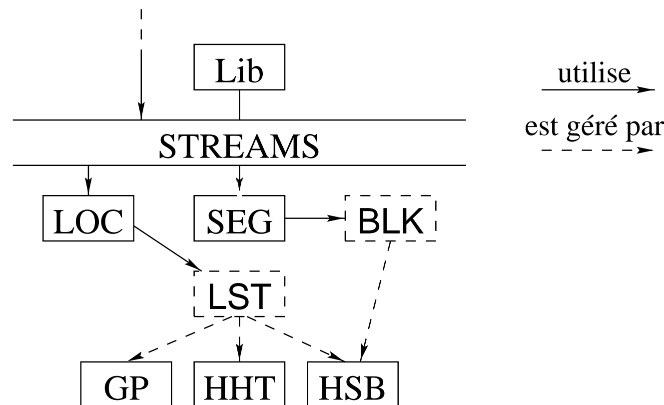

Figure II-16 : Vue d'ensemble des composants de la gestion de segments

Nous avons vu à la section I-6 une représentation de l'architecture générale. L'architecture interne du module de localisation est réalisée à l'aide de cinq composants logiciels :

LOC Point d'entrée

SEG Allocateur de segments

GP Gestion de la table de partition

HHT Gestion des tables de hachage hiérarchiques

HSB Gestion des blocs

et deux structures de données :

LST Fichier du site : références sur tous les segments locaux

BLK Ensemble des blocs d'allocation

Par ailleurs, le module d'allocation de segment comprend une bibliothèque de fonctions (Lib) qui doit être liée à l'application et qui est chargée d'effectuer la transformation des appels de création/destruction (SEG_Create et SEG_Destroy) en messages STREAMS à destination du module SEG.

Tout cela est résumé sur la figure II-16. Les flèches sur cette figure indiquent les dépendances des modules et des structures de données.

II-7 Interface d'administration

Nous avons décidé de partitionner l'espace global (II-4.3) de manière à pouvoir déplacer les partitions et par là répartir la charge des méta-données de localisation. Pour gérer cette répartition, il existe une interface d'administration réservée à un logiciel spécialisé pour permettre la reconfiguration de la partition. Cette interface permet d'activer et déplacer des partitions sur n'importe quel site.

Activation d'une partition Le nombre de partitions existantes est déterminé au départ et ne peut être changé, mais certaines partitions peuvent rester inactives jusqu'à ce qu'une activation ait lieu, c'est-à-dire jusqu'à ce qu'on lui assigne un site localisateur.

Déplacement d'une partition Pour reconfigurer le réseau, par ajout, retrait ou reconfiguration d'une machine, il est nécessaire de pouvoir déplacer des partitions en cours d'utilisation, c'est-à-dire sans arrêter Arias sur les machines. Il faut noter que le déplacement d'une partition est une opération lourde. De plus, elle nécessite un protocole de synchronisation qui garantisse le bon fonctionnement des requêtes de localisation tout au long du déplacement.

Exemple Prenons l'exemple du remplacement d'une machine A par une machine B plus puissante. Il faudra procéder en plusieurs étapes :

- début : gel de la table de partition ;
- branchement de la machine B sur le réseau et démarrage d'Arias ;
- déplacement des partitions de A vers B ;
- arrêt d'Arias sur A et coupure de A ;
- éventuellement, déplacement d'autres partition vers B (puisque cette machine est plus puissante, elle peut prendre en charge plus de partitions) ;

- éventuellement, activation de nouvelles partitions sur B ;
- fin : dégel de la table de partition.

II-8 Couplage et protection

Le système de protection d'Arias [73] définit deux notions fondamentales : la capacité et le domaine.

II-8.1 Des capacités

Le principe des capacités [49, 77] a été appliqué de longue date. Brièvement, une *capacité* est une sorte de super-pointeur qui contient, en plus des informations d'adressage, des indications de protection. Ces indications spécifient l'étendue des droits d'accès aux données pointées par les informations d'adressage. Typiquement, les droits sont du type lecture, écriture, et exécution.

Un système de capacités logicielles nécessite un cryptage à un certain niveau. En l'absence d'un support de matériel[11], une capacité ne peut se manipuler comme un pointeur, car il est impossible d'en garantir la validité et l'intégrité. On fait alors appel au cryptage ou à la dissimulation. En fait de capacité, le système fournit aux application un code (une *capacité logicielle*) qui peut représenter une capacité cryptée ou un mot de passe référençant une capacité stockée dans le noyau. Cette capacité peut être librement échangée entre les applications. Pour donner à un processus l'accès à un segment, il suffit de lui donner la capacité correspondante ; ce dernier peut alors demander au système de coupler le segment dans son propre espace d'adressage, dans la limite des droits accordés par la capacité qu'il détient.

Un tel système de capacités logicielles est très souple d'emploi, et a été adopté par exemple par le projet Mungi [79]. Mais ce système impose que les applications aient connaissance des capacités et donc du système de protection, ce qui viole nos objectifs. En effet,

- les applications doivent gérer elles-mêmes leurs listes de capacités ;
- l'utilisation des capacités doit faire l'objet d'appels explicites, par exemple pour ouvrir l'accès à un segment ;
- de plus, une partie de la protection repose sur la sûreté des applications elles-mêmes, puisqu'elles peuvent disséminer les capacités ou les laisser à la vue d'applications malicieuses.

[11] Le Plessey 250 [30] et l'IBM System/38 [38] sont des exemples d'architectures qui fournissent un support matériel pour des capacités sur des segments de mémoire.

II-8.2 Des domaines

Dans Arias, les capacités ne quittent jamais l'environnement protégé du noyau système. Un domaine de protection est une structure interne au noyau représente un environnement d'exécution regroupant des capacités. Les capacités possédées par un domaine définissent exactement les droits d'accès aux segments de la mémoire partagée des processus exécutant dans ce domaine.

La protection passe par le principe de la *vérification au couplage*. Lorsqu'un processus tente un accès à un segment qui est protégé par une capacité, le système vérifie que le domaine dans lequel s'exécute ce processus possède une capacité adéquate, puis couple le segment dans l'espace d'adressage du processus. Une fois le couplage effectué, aucune autre vérification de protection n'a lieu.

II-8.3 Couplage implicite

Partons du point où une application vient de créer un segment. Ce segment existe dans l'espace Arias, mais aucun processus n'y accède encore. Le couplage de ce segment dans l'espace d'adressage Unix d'un processus qui y accède se fait au moment du premier accès, de manière implicite. Évidemment, le mécanisme de couplage est le même pour le processus qui a créé le segment que pour tout autre processus qui y accède. Le module de gestion des couplages, que l'on appelle le *paginateur*, est en effet très fortement lié au système de mémoire virtuelle et au mécanisme de pagination de l'architecture et du système sous-jacents.

Le système de protection entre en jeu au moment du couplage implicite. Lorsqu'un processus accède à une région de la mémoire correspondant à un segment non encore couplé, il provoque un défaut de mémoire virtuelle. Ce défaut, via le système de gestion d'exceptions, a pour effet d'alerter le paginateur Arias. Ce dernier fait appel au module de protection pour vérifier que le domaine dans lequel le processus fautif s'exécute possède une capacité avec des droits suffisants sur le segment demandé. Si ce n'est pas le cas, la faute provoque une exception normale, du type «faute de segmentation» (*segmentation fault*). Si les droits sont suffisants, le paginateur fait appel au système de mémoire virtuelle d'AIX pour coupler le segment dans l'espace d'adressage du processus fautif. Par la suite, le paginateur instancie les pages du segment au fur et à mesure des défauts de pages provoqués par l'application, et par l'intermédiaire du système de mémoire virtuelle d'AIX.

Le fonctionnement du paginateur et son interface avec AIX sont décrits plus en détail en section III-3.

II-8.4 Capacités et ramasse-miettes

L'utilisation de capacités logicielles gérées dans le noyau ouvre la possibilité d'une technique de ramasse-miettes de segments. En effet, il est clair que si aucun domaine

ne possède de capacité sur un segment, ce dernier n'est accessible par personne et peut donc être supprimé. Cette méthode de destruction de segment vient en complément de l'interface de destruction de segments que nous avons introduit en section II-1.1, mais ne l'invalide pas, car le module de protection est facultatif ; en son absence, il faut conserver un moyen de détruire des segments à l'initiative des applications.

Chapitre III

Réalisation

Ce chapitre décrit en détail l'implémentation des idées dégagées dans le chapitre précédent. Il s'articule en trois sections sur la localisation, l'allocation, et le couplage. Certaines décisions qui ne peuvent être prises par une analyse qualitative sont identifiées et laissées ouvertes pour le chapitre suivant, qui traite du paramétrage de l'implémentation.

III-1 Localisation

L'opération de localisation peut être considérée, vue de l'extérieur (cf. la section II-1.2 sur l'interface externe), par une fonction qui prend en entrée une adresse virtuelle et retourne la fiche[12] (II-1) du segment contenant cette adresse. L'ensemble des segments connus par un site (les *segments locaux* de ce site) est référencé dans une hiérarchie de structures de données, le *fichier* (II-4.5).

III-1.1 Gestion des accélérateurs

Comme nous l'avons vu en II-4, cette recherche est en principe structurée en trois étapes. Les trois étapes consistent à identifier la partition à laquelle appartient le segment, puis à rechercher le bon bloc dans cette partition et enfin à parcourir le contenu du bloc jusqu'à atteindre le grain du segment. Cet algorithme est en fait précédé de la recherche d'un accélérateur (II-4.5). Ces accélérateurs étant intégrés dans le fichier lui-même, les étapes de la recherche sont donc :

1. consultation du fichier à la recherche d'un descripteur ou d'un accélérateur ;
2. en cas d'échec, consultation de la table de partition $\rightarrow$ site localisateur ;

[12]Rappelons que la fiche d'un segment contient son adresse de début, sa taille et la localisation de son descripteur.

3. obtention de la localisation du segment depuis le site localisateur ;
4. création des structures locales concomitantes au référencement d'un nouveau segment (cache de descripteur, accélérateur).

Le cycle d'utilisation d'un segment sur un site non primaire est le suivant :

1. Lorsqu'un segment est référencé pour la première fois, la recherche normale, via le site localisateur, a lieu.
2. Ensuite, un cache local est immédiatement créé.
3. Lorsque le segment n'est plus utilisé par aucun processus sur la machine, ce cache local est détruit et l'information de localisation, qui était une référence au cache local, devient alors un accélérateur.
4. Enfin, lorsque le segment migre par la suite, l'accélérateur est invalidé.

Par ailleurs, lorsqu'un segment migre (II-1.2-e), l'ancien site primaire devient un simple site client. Le descripteur qu'il contenait est transformé en cache, et ce dernier entre dans le cycle précédent au point 2.

La caractéristique importante de l'accélérateur est que s'il est présent, alors il est exact. Cela découle de son mode de mise à jour : un accélérateur est simplement conservé lorsqu'un segment cesse d'être utilisé. Il est par la suite invalidé au moment où le segment migre (cf. II-1.2 sur la migration). Or, tant qu'un segment n'a pas migré, sa dernière localisation reste exacte, par définition de la migration.

Nous justifions ce mode de mise à jour par plusieurs observations :

- la migration d'un segment est un événement relativement rare par rapport aux échanges au niveau des zones ;
- la migration est aussi une opération relativement lourde et lente, donc un envoi de messages supplémentaires compte peu ;
- le stockage d'un accélérateur entraîne un surcoût d'occupation en mémoire relativement faible par rapport à son utilité.

Cette dernière justification est légèrement spéculative. Le coût de stockage d'un accélérateur est très variable en fonction du taux d'utilisation de l'espace d'adressage au voisinage du segment qu'il référence. Si le stockage d'un accélérateur doit mobiliser un bloc entier, son coût peut paraître élevé. Cependant, la structure des blocs est extrêmement économe en mémoire, comme nous le voyons en II-4.2-c et en III-1.3. Cela justifie de garder un accélérateur aussi longtemps que possible.

La recherche de la fiche d'un segment à partir d'une adresse virtuelle s'appuie sur trois structures de données pour la recherche : les tables de hachage hiérarchiques, le buddy-system, et la table de partition. La suite de cette section détaille ces structures et leur utilisation.

III-1.2 Tables de hachage hiérarchiques

Le *fichier* d'un site regroupe l'ensemble des segments localement référencés du fait que le site courant en est le site primaire ou le site localisateur, ou simplement pour référencer un cache local ou un accélérateur (II-4.5). Le premier niveau de structuration est fait par une hiérarchie de tables de hachage, comme décrit en II-4.1 et illustré sur la figure II-7, qui permet d'identifier le bloc dans lequel se trouve le segment recherché.

Un certain nombre de paramètres doivent être fixés :

- le nombre de niveaux de tables de hachage, ainsi que la taille des tables à chaque niveau ;
- la quantité d'adresses couverte par un bloc ;
- la taille maximale d'un segment.

La taille maximale de segment détermine la taille minimale d'un bloc, puisqu'un segment doit être entièrement contenu dans un bloc. Cette taille est imposée par AIX à 256 Mo (28 bits). À la manière dont nous couplons les segments Arias aux segments AIX (cf. III-3), nous simplifions la structure du paginateur en nous imposant une taille maximale de segment Arias au plus égale à la taille des segments AIX[13]. Partant de là, il reste à la hiérarchie de tables de hachage à décoder $64 - 28 = 36$ bits d'adresse.

III-1.2-a Fonction de hachage

Pour un ensemble de clés indifférenciés, il est généralement conseillé [82, 4.6] d'utiliser une fonction de hachage basé sur des nombres premiers. Un exemple de fonction simple est d'interpréter la clé comme un nombre entier et de calculer le modulo par un nombre premier, ce dernier étant naturellement la taille de la table de hachage. Une telle fonction, bien qu'arithmétiquement simple, est en réalité une fonction complexe, qui nécessite de nombreux cycles machine. Cela peut se ressentir particulièrement lors du rehachage d'une table.

Un autre type de fonction de hachage repose sur une composition binaire plus ou moins sophistiquée. Généralement, une telle fonction doit faire intervenir l'ensemble des bits significatifs de la clé ; on utilise donc généralement la fonction OU exclusif. Les tables de hachage résultants ont des tailles qui sont des puissances de 2. Pour une hiérarchie de tables de hachage, il faut prêter attention à ces tailles.

Donnons-nous une clé quelconque et imaginons une table de premier niveau à 256 entrées. La fonction de hachage donne la valeur 3 pour cette clé, par exemple, et la troisième table de second niveau a aussi une taille de 256 entrées. Si nous utilisons la même fonction de hachage sur la même clé avec la même taille de table, nous trouverons

[13] Nous voyons en III-3.7 ce qui serait nécessaire pour avoir des segments Arias plus grand que la limite imposée par AIX.

fatidiquement la même valeur 3, et ceci pour toutes les clés dont le résultat est 3. Il s'ensuit une gigantesque collision sur la valeur 3, et la table de second niveau sera gérée comme une liste.

Pour sa rapidité et sa simplicité, le choix s'est porté sur une fonction de hachage par composition binaire faisant intervenir tous les bits de la clé. Cette fonction calcule la valeur du OU exclusif par découpage de la clé en blocs de bits de taille fixe. La figure III-1 illustre cette fonction sur une clé de 12 bits, calculant une valeur finale sur 4 bits.

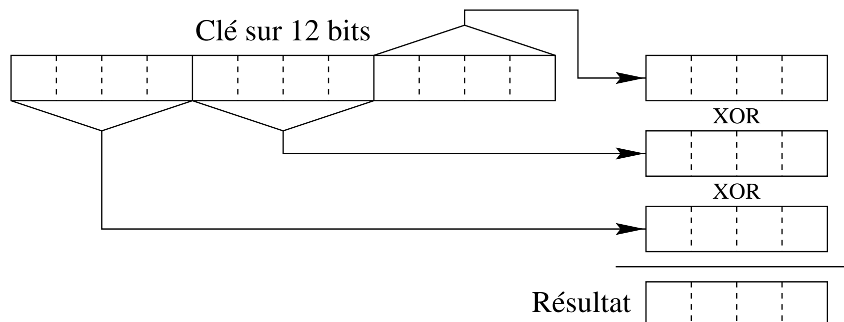

Figure III-1 : Fonction de hachage par composition binaire, exemple sur 12 bits

III-1.2-b Traitement des collisions – itérateur

Du point de vue de la table de premier niveau, les collisions sont gérées de manière externe par une autre table de hachage. Ainsi, la table de premier niveau a une taille fixe. Il en est de même jusqu'au dernier niveau de tables.

Au dernier niveau, les collisions sont gérées en interne en utilisant un simple itérateur. Pour éviter de trop nombreuses collisions, les tables de dernier niveau sont agrandies lorsque le nombre d'éléments dépasse un certain seuil, exprimé en pourcentage de la taille de la table. Inversement, pour éviter de gâcher de la place, la table est rétrécie lorsque ce nombre passe en-dessous d'un autre seuil. Comme les tables à tous les niveaux ont des tailles de puissance de 2, l'agrandissement et le rétrécissement se font d'un facteur 2.

La seconde fonction de hachage intervient justement pour la résolution des collisions. Lorsqu'une clé donne lieu à une collision avec la fonction de hachage, la seconde fonction donne successivement des positions différentes dans la table (d'où son nom d'*itérateur*). Un bon itérateur nécessite deux caractéristiques essentielles.

Exhaustivité Il doit finir par trouver une entrée libre, s'il y en a.

Disjonction Il doit fournir, autant que possible, une suite de valeurs différente pour des clés différentes.

Dans le cas de la résolution interne des collisions, comme c'est le cas pour nos tables de hachage aux feuilles, N. Wirth [82, p. 273] calcule une valeur caractéristique, la longueur moyenne de chaîne de recherche, qui représente le nombre moyen de tentatives d'insertion. La figure III-2 montre le cas d'une chaîne de recherche de longueur 3. f est la première fonction de hachage, est g_n la n-ième itération de la seconde fonction de hachage.

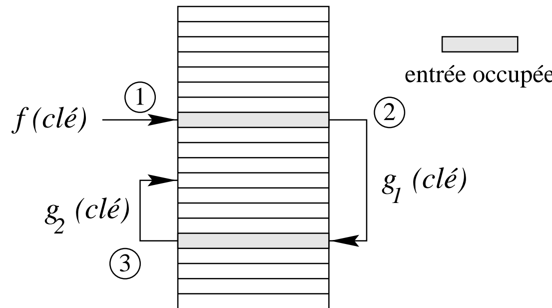

Figure III-2 : Exemple de chaîne de recherche de longueur 3

Un itérateur uniforme est une fonction qui répartit les clés de manière uniforme dans toute la table. Une autre forme d'itérateur, dite linéaire, se contente de traverser la table linéairement à partir de la première collision. Un itérateur linéaire est naturellement la plus simple à programmer, et donne des résultats qui sont moins bons, sans être catastrophiques.

Wirth montre que pour une distribution de clé aléatoire et une fonction de hachage aléatoire, cette longueur moyenne ne dépend que du taux de remplissage. La table III-a résume quelques valeurs typiques de longueur moyenne de chaîne de recherche, d'une part avec un itérateur uniforme et d'autre part avec un itérateur linéaire.

Taux de remplissage	Itérateur uniforme	Itérateur linéaire
10%	1,05	1,06
50%	1,39	1,50
75%	1,85	2,50
90%	2,56	5,50
95%	3,15	10,50

Tableau III-a : Quelques valeurs typiques de collisions dans une table de hachage

III-1.2-c Géométrie des tables

Nous appelons la *géométrie* des tables l'ensemble des paramètres tels que la taille des tables à chaque niveau. Nous avons plusieurs possibilités pour décoder 36 bits par la fonction de composition binaire :

1. en deux niveaux, par exemple 10 et 26 bits, ou
2. en trois niveaux, par exemple 10, 10 et 16 bits.

Dans notre structure, les tables de niveaux intermédiaires sont de taille fixe (par exemple 2^{10} pour le premier cas ci-dessus), alors que les tables de dernier niveau ont une taille variable en fonction du taux d'utilisation. Le compromis à trouver doit tenir compte des deux considérations contradictoires : une table de taille fixe peut occuper inutilement de l'espace, alors qu'une table de taille variable nécessite des rehachages.

Dans le premier exemple, la structure entière nécessite une table de taille fixe $2^{10} = 1024$ entrées et jusqu'à 1024 tables de taille variable. Le second exemple occupe une table de 1024 entrées également, puis 1024 tables de 1024 entrées, et enfin jusqu'à 1024×1024 tables de taille variable. En supposant un taux d'occupation identique dans les tables de dernier niveau, les tables du premier exemple sont en moyenne 1000 fois plus grandes que celles du second exemple.

On peut ajouter à cette réflexion quelques intuitions :

- dans le cas d'une hiérarchie à deux niveaux, le problème provient plutôt des tables variables du second niveau, qui doivent décoder trop de bits ;
- dans le cas d'une hiérarchie à trois niveaux, le problème est surtout l'occupation mémoire des tables des deux premiers niveaux ;
- une table de taille fixe de 2^n occupe à peu près le même espace qu'une hiérarchie à deux niveaux de tables de taille 2^m et 2^p si $n = m + p$.

Pour déterminer la viabilité d'une solution à deux niveaux, il faut savoir si une table variable décodant par exemple 26 bits n'est pas susceptible de contenir trop d'entrées, donc trop coûteux à rehacher. Dans le cas extrême, si tous les segments sont regroupés dans l'espace d'adresses de manière à n'utiliser qu'une entrée dans la table de premier niveau, ces segments sont tous référencés dans la même table de second niveau. La table de premier niveau est alors pratiquement inutilisée, et la table de second niveau est énorme, comme illustrée en figure III-3.

La solution à trois niveaux est quant à elle susceptible d'utiliser trop de mémoire si les deux premiers niveaux sont volumineux et si les segments sont très dispersés dans l'espace d'adresses. La configuration extrême est celle dans laquelle 1024 segments suffisent à occuper chacun une entrée dans la table de premier niveau, ce qui entraîne l'utilisation d'une table de 1024 entrées pour référencer chaque segment (figure III-4).

L'étude des cas extrêmes ne nous apprend que ce qu'il ne faut pas faire. Le choix d'une géométrie de tables de hachage doit donc être déterminé par une étude statistique sous une utilisation réaliste. Ce sera fait dans le chapitre suivant.

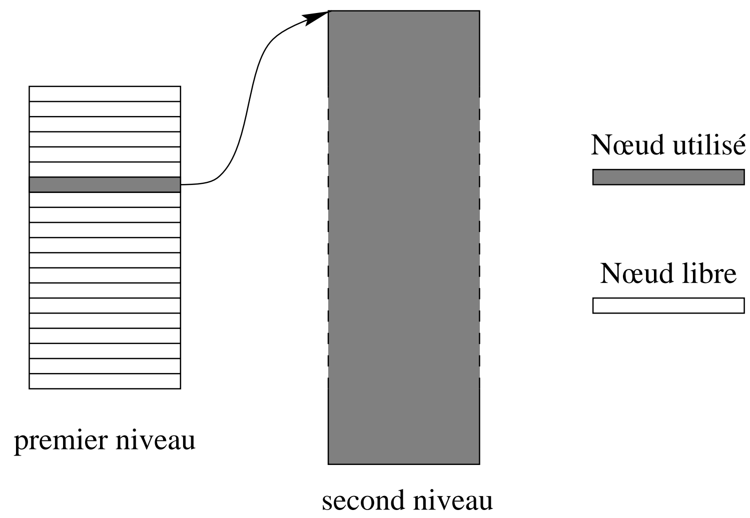

Figure III-3 : Cas extrême de hiérarchie de tables de hachage à deux niveaux

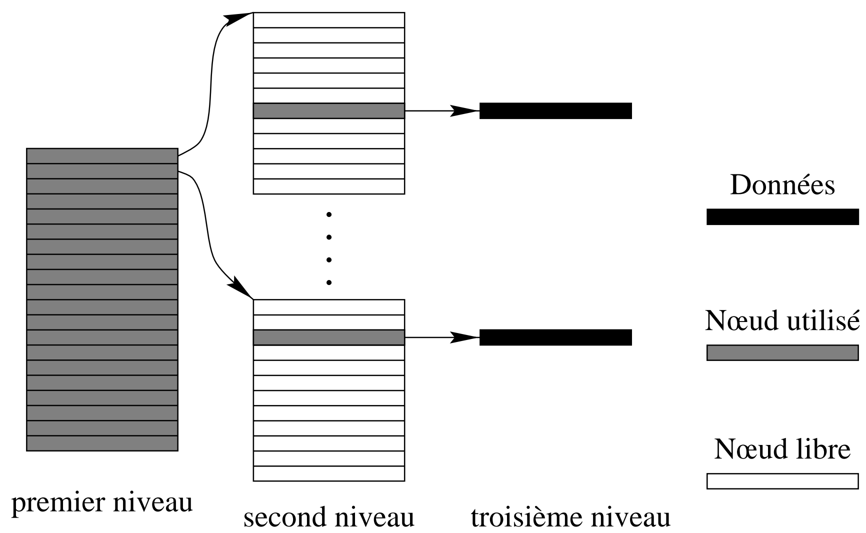

Figure III-4 : Cas extrême de hiérarchie de tables de hachage à trois niveaux

III-1.3 Blocs, buddy-system

Une fois le bloc identifié à travers la hiérarchie de tables de hachage, il faut traverser sa structure en buddy-system pour atteindre le grain du segment. Nous avons déjà discuté des avantages du buddy-system et de ses variantes non binaires (II-4.2). Il s'en est dégagé que si un buddy-system binaire est plus facile à gérer, un système non binaire permet une meilleure utilisation de l'espace d'adressage, au prix d'algorithmes de recherche et d'allocation plus complexes.

Outre une meilleure utilisation de l'espace d'adresses virtuelles, la raison principale qui justifie le choix d'un buddy-system non binaire est que la traversée d'un buddy-system de base n nécessite un plus grand nombre de poursuites de pointeurs que pour un buddy-system de base $m > n$. Par exemple, un système binaire nécessite trois déréférencements pour obtenir le même résultat qu'un calcul de déplacement pour un système de base $16 = 2^4$ (voir figure III-5).

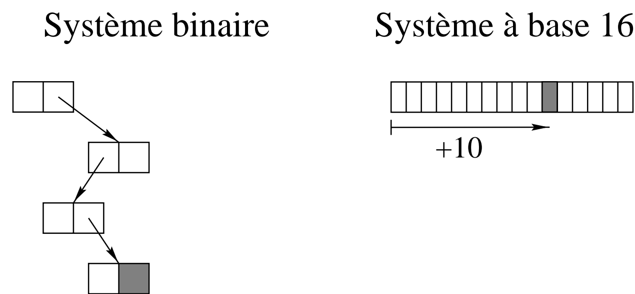

Figure III-5 : Comparaison entre un buddy-system binaire et un buddy-system de base 16 pour le décodage de la valeur $\texttt{1010}_2 = 10$.

Comme nous l'avons déjà vu, la quantité d'adresses virtuelles couverte par un bloc (que nous appelons la «*taille*» du bloc) est limitée vers le bas par la taille maximale d'un segment, soit dans notre cas 2^{28} octets. La géométrie exacte du buddy-system doit aussi tenir compte de la *consommation minimale* (ou CM) d'un segment. La consommation minimale d'un segment est la plus petite quantité d'espace virtuel utilisée pour chaque segment. En d'autres termes, tout segment est aligné sur un multiple de cette quantité. La seule limite est une limite inférieure, celle de la page ($4096 = 2^{12}$ octets pour AIX). Les effets comparés du paramètre CM sont illustrés sur la figure III-6. Cette figure montre deux cas, l'un avec une CM de 2^{12} (4096), l'autre avec 2^{14} (16384). Dans chaque cas nous montrons l'allocation de mémoire virtuelle pour la création de deux segments, l'un de 2^{12} octets, l'autre de 2^{13}.

Si l'on se fixe 2^{28} comme limites de taille de segment et 2^{12} comme consommation minimale, et que l'on fixe à 2^{28} la «taille» du bloc, cela signifie que le bloc doit pouvoir décoder 16 bits d'adresse. C'est possible en quatre étapes de 4 bits, soit quatre niveaux de tables à $2^4 = 16$ entrées.

La figure III-7 illustre un bloc dont l'adresse de début est $\texttt{X000.0000}_{16}$ et contenant un segment S1 de 4096 octets (2^{12}) à l'adresse $\texttt{X000.0000}_{16}$ et un segment S2 de 128 Mo

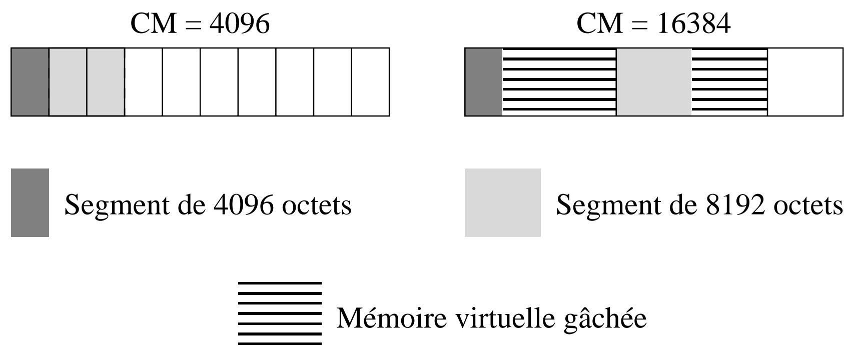

Figure III-6 : Effet de la consommation minimale d'allocation de segment. Les lignes verticales représentent les adresses alignées sur la CM.

(2^{27}) à l'adresse $\texttt{X100.0000}_{16}$. Ne nous préoccupons pas, pour l'instant, de savoir par quel algorithme d'allocation ces segments se trouvent à ces adresses. Imaginons que ces segments sont des ≪constantes≫ qui se trouvent être là, et concentrons-nous sur l'arborescence de buddy-system dont on a besoin pour les représenter.

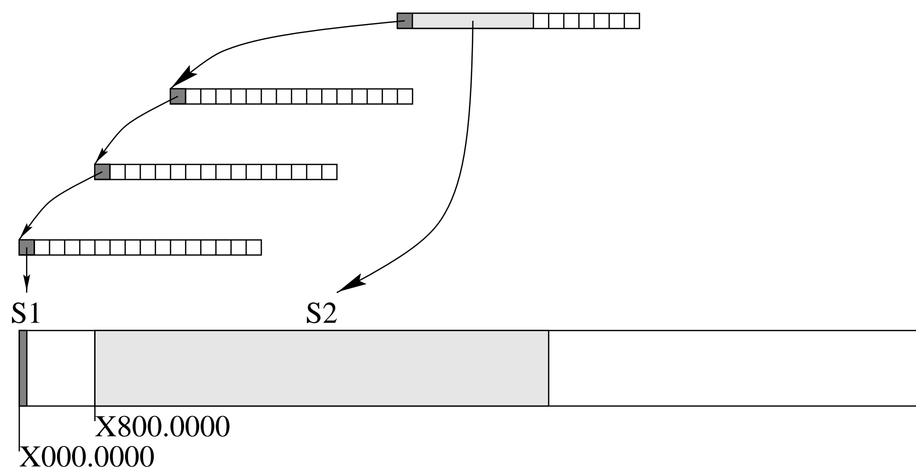

Figure III-7 : Exemple de buddy-system de base 16 couvrant 16 bits d'adresse

Une variation possible sur ce système est d'être plus généreux en espace virtuel, et de décider d'une consommation minimale de segment plus grande. Fixons par exemple cette consommation à $2^{16} = 65536$ octets. Un segment de taille 4096 octets (une page) consomme quand même 65536 octets, soit un gâchis d'espace virtuel correspondant à $65536 - 4096 = 61440$ octets. Dans cette configuration, un bloc n'a plus à décoder que 12 bits d'adresse, ce qui est faisable en trois étapes de 4 bits, soit trois niveaux de tables à 16 entrées.

Le choix d'une géométrie de bloc est donc un compromis entre trois facteurs :

- la taille moyenne des segments et

- le taux de gâchis d'espace virtuel, ainsi que
- la taille maximale de segment.

III-1.4 Table de partition

La table de partition permet d'identifier le site localisateur d'un segment. Comme nous l'avons vu en II-4.3, cette table est répliquée sur tous les sites et modifiée atomiquement à travers l'interface d'administration (II-7).

Le nombre de partitions possibles dans la mémoire partagée globale est fonction de plusieurs considérations pratiques :

- le nombre de machines constituant le réseau,
- la variation dans les capacités de stockage et de traitement parmi ces machines, et
- la fréquence de reconfiguration du réseau.

En effet, il est raisonnable de placer au moins une partition par machine, de manière à gérer la création de segments de manière entièrement autonome sur toutes les machines (cf. II-5.3). Cela dit, certaines machines peuvent être destinées à être des serveurs de données, auquel cas elles seront susceptible de ≪consommer≫ plus de partitions. Enfin, un plus grand nombre de partitions, de manière générale, facilite une reconfiguration du réseau. Par exemple lors de l'ajout d'une nouvelle machine serveur, il peut être utile de transférer un certain nombre de partitions depuis l'ancien serveur vers le nouveau, de manière à équilibrer la charge.

L'objectif d'Arias étant de fonctionner sur un réseau d'au plus quelques dizaines de machines, on peut prévoir que le nombre de partitions ne dépasse pas quelques centaines. Dans ces conditions, la table de partition peut être une simple table linéaire qui fait correspondre une partition (ou les bits de poids fort la caractérisant) à son site localisateur.

La mise à jour atomique se fait par un protocole à cinq phases :

1. verrouillage de la table ;
2. si nécessaire, transfert des données de localisation ;
3. modification de la table ;
4. si nécessaire, effacement des données de localisation de l'ancien site localisateur ;
5. déverrouillage.

Durant toute cette mise à jour, Arias reste entièrement disponible. S'il s'agit de l'affectation d'une nouvelle partition, aucun segment n'y existe encore, et il n'y a pas de risque d'interférence entre la modification de la table de partition et son utilisation. S'il s'agit du transfert d'une partition, l'ancien site localisateur continue à fournir les informations de localisation sur la partition en transit jusqu'à l'accomplissement de la 4e étape.

III-2 Allocation

La création d'un segment (cf. section II-1.1 pour l'interface de création/destruction de segments) nécessite d'abord l'allocation d'une plage d'adresses correspondant à sa taille, puis la création de son descripteur, pris en charge par la CGC (II-1.2). Comme l'allocation se fait dans un bloc éventuellement emprunté, et que ce bloc peut arriver à épuisement, nous mettons ensuite en œuvre l'algorithme de remplacement de blocs.

III-2.1 Algorithme d'allocation dans un bloc

Nous partons de l'hypothèse que le bloc à utiliser est déterminé (voir II-5). On exploite le buddy-system qui référence ce bloc pour trouver un espace de taille adéquate au segment à allouer.

L'algorithme d'allocation de base repose sur un pointeur qui indique la prochaine adresse libre, un peu comme l'appel système `sbrk` d'Unix. Mais la structure du buddy-system n'est pas optimalement utilisée si l'on se contente d'un seul pointeur. En effet, prenons par exemple l'allocation dans le bloc à l'adresse $\texttt{X000.0000}_{16}$ d'un segment de 4096 octets suivi de celle d'un segment de 127 Mo. L'allocation du premier segment S1 se fait naturellement à l'adresse la plus basse du bloc, soit $\texttt{X000.0000}_{16}$. Ce segment occupe une entrée dans la première table ; le segment suivant S2 doit donc utiliser les 8 entrées suivantes, en commençant à l'adresse $\texttt{X100.0000}_{16}$. Le pointeur d'allocation se trouve alors en $\texttt{X900.0000}_{16}$. On constate que toute la plage $\texttt{X000.1000}_{16}$–$\texttt{X0FF.FFFF}_{16}$ est définitivement inutilisée (figure III-8).

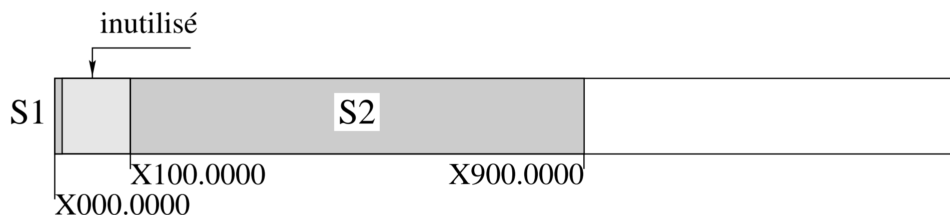

Figure III-8 : Allocation dans un bloc en utilisant un seul pointeur

Une technique plus souple consiste à maintenir autant de pointeurs d'allocation qu'il existe de niveaux de décodage. Par exemple, pour un bloc organisé en 4 niveaux de base 16 (décodant chacun 4 bits), il y aura quatre pointeurs d'allocation, un pour chacun des niveaux. Le pointeur de premier niveau sert à l'allocation des segments dont la taille est au moins égale à la plage couverte par une entrée dans la table de premier niveau, soit 2^{24} octets. Le pointeur de second niveau sert aux segments de moins de 2^{24} octets et d'au mois 2^{20} octets, et ainsi de suite jusqu'au pointeur de quatrième niveau, qui sert aux segments de moins de 2^{16} octets.

Reprenons notre exemple précédent en utilisant le nouvel algorithme. Les segments S1 et S2 sont alloués aux mêmes adresses, mais si le pointeur de premier niveau se trouve

bien en $\texttt{X900.0000}_{16}$ après l'allocation de S2, le pointeur de quatrième niveau se trouve en $\texttt{X000.1000}_{16}$ après celle de S1. Si l'on veut allouer un autre segment S3 de 4096 octets, il sera placé en $\texttt{X000.1000}_{16}$ et le pointeur de quatrième niveau se retrouvera en $\texttt{X000.1000}_{16}$. La figure III-b indique les états successifs des pointeurs de chaque niveau au cours de l'allocation des trois segments.

	Adresse allouée	Taille	Niveau 1	Niveau 4
Départ			$\texttt{X000.0000}_{16}$	inutilisé
Après S1	$\texttt{X000.0000}_{16}$	4096	$\texttt{X100.0000}_{16}$	$\texttt{X000.1000}_{16}$
Après S2	$\texttt{X100.0000}_{16}$	2^{27}	$\texttt{X900.0000}_{16}$	$\texttt{X000.1000}_{16}$
Après S3	$\texttt{X000.1000}_{16}$	4096	$\texttt{X900.0000}_{16}$	$\texttt{X000.2000}_{16}$

Tableau III-b : États des pointeurs d'allocation après l'allocation de trois segments S1, S2 et S3, de tailles respectives 4096, 2^{27} et 4096 octets. Les niveaux 2 et 3, inutilisés dans cet exemple, ont été omis.

À noter qu'à cet instant les pointeurs de niveaux 2 et 3 ne sont pas encore utilisés. Ils le seront si un segment de taille approprié doit être alloué. Par exemple, si l'on doit créer un segment S4 de taille 2^{20} (correspondant au deuxième niveau), la séquence suivante a lieu :

1. le pointeur de premier niveau est avancé en $\texttt{XA00.0000}_{16}$,
2. le pointeur de deuxième niveau est activé en $\texttt{X900.0000}_{16}$,
3. le segment S4 est alloué en $\texttt{X900.0000}_{16}$, et enfin
4. le pointeur de deuxième niveau est avancé en $\texttt{X910.0000}_{16}$.

La figure III-9 illustre l'état final des pointeurs et la position des segments.

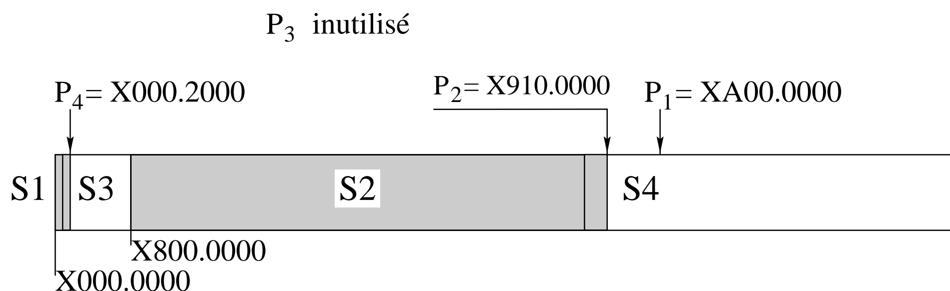

Figure III-9 : Allocation d'un segment dans un bloc encombré

Dans cet état, le plus grand espace contigu se trouve en $\texttt{XA00.0000}_{16}$ et a une taille de $\texttt{600.0000}_{16} = 6 \times 2^{24} =$ octets. Si une application demande la création d'un segment d'une taille strictement supérieure à cela, cette demande ne peut être satisfaite dans ce bloc. À ce moment, nous changeons de bloc d'allocation. La section III-2.3 ci-dessous en décrit la méthode.

III-2.2 Interaction avec la CGC

La création d'un segment débute par l'allocation de l'espace virtuel, qui se déroule comme indiqué dans la section précédente. Ensuite, il faut insérer la fiche du segment dans le bloc. Cette fiche comprend l'adresse du descripteur du segment (cf. II-2) ; or, ce descripteur n'est pas encore créé à ce stade.

Seule la CGC peut créer ce descripteur, puisqu'elle est la seule à en connaître la structure. De ce fait, les opérations suivantes sont effectuées en séquence :

1. l'adresse allouée est transmise à la CGC ;

2. la CGC crée un descripteur de segment et en renvoie l'adresse ;

3. la fiche (II-2) du segment peut alors être complétée et insérée dans le bloc.

À ce moment, le segment est complètement créé. Son adresse de début peut être renvoyée à l'application.

III-2.3 Protocole de réservation de bloc

Tout ce qui précède est parti de l'hypothèse que le bloc dans lequel l'allocation devait se faire était déjà disponible. Ici, nous détaillons le protocole de réservation de bloc qui permet de faire cette hypothèse. La section II-5 décrit en détail comment se fait le choix du bloc à utiliser pour la création d'un segment donné. Partons du cas le plus simple, où la partition choisie est locale.

Une partition en cours d'allocation possède un pointeur d'allocation courant. Ce pointeur est avancé par pas d'un bloc au fur et à mesure que la partition est ≪consommée≫. Lorsqu'un bloc est prélevé d'une partition, il est rattaché au fichier du site et devient le bloc d'allocation courant pour cette partition. Lorsque ce bloc est rempli, c'est-à-dire quand une demande de création de segment ne peut être satisfaite dans ce bloc faute de place, un nouveau bloc est prélevé de la même partition pour servir de bloc courant.

Dans le cas des blocs dans une partition distante (c'est-à-dire une partition dont le site actuel n'est pas le site localisateur), le prélèvement d'un nouveau bloc dans la partition est en fait un *emprunt* de bloc (cf. II-5.1, et nécessite un échange de message. La séquence des événements est en effet :

1. constatation que le bloc courant ne possède pas l'espace contigu nécessaire (cf. III-2.1) ;

2. remise du bloc vers son site localisateur et demande d'un nouvel emprunt ;

3. réception du nouveau bloc emprunté et

4. nouvelle tentative d'allocation.

On voit donc que l'échange de message intervient au cours de l'allocation, ce qui augmente le délai d'une quantité intolérable.

Pour remédier à ce problème, chaque bloc emprunté est accompagné d'un *bloc de réserve*, dont le rôle est de remplacer immédiatement le bloc emprunté quand ce dernier est plein. Cela permet de satisfaire la demande d'allocation immédiatement. Le bloc usé peut être remis à son site localisateur de manière asynchrone, de même que la demande d'un nouveau bloc de réserve (opérations 2 et 3 ci-dessus).

III-3 Couplage

Le *couplage* d'un segment en mémoire consiste à rendre accessible un segment de mémoire Arias à un processus. Par principe, le couplage de la mémoire est *implicite*, c'est-à-dire qu'il ne nécessite pas de traitement particulier de la part de l'application. Simplement, au moment où celle-ci accède à des zones de mémoire, les segments sous-jacents sont couplés dans son espace d'adresse à la demande (cf. II-8).

Cette section traite du fonctionnement du module de pagination (PGR) qui est chargé de réaliser cela. Bien qu'un peu éloignée du sujet central de ce rapport, elle donne une idée plus concrète du fonctionnement de la mémoire virtuelle et elle représente une partie des investigations qui ont été menées au sein du projet.

III-3.1 Principes du couplage

On distingue deux types de tentatives d'accès : le défaut de segment et le défaut de page.

III-3.1-a Défaut de segment

Le défaut de segment intervient lorsqu'une application accède à un segment qui n'est pas encore couplé dans son espace d'adressage. Ce type de tentative d'accès provoque une exception (*Data Storage Interrupt*, ou DSI) que l'on peut intercepter à l'aide d'un traitant approprié. Ce traitant doit alors identifier le segment qu'il doit coupler. Ensuite, il couple le segment de telle manière que les accès suivants à des pages de ce segment provoque non plus une DSI mais un défaut de page, dont le traitement est décrit ci-après.

III-3.1-b Défaut de page

Quand une application accède à une page qui n'est pas en mémoire physique, mais qui appartient à un segment déjà couplé, il s'agit d'un défaut de page. Dans ce cas, le système de gestion de mémoire virtuelle d'AIX identifie le segment concerné et appelle un point d'entrée de pagination (appelée la *stratégie*). Par ce biais, le paginateur identifie la page à instancier.

Le paginateur lui-même ne se charge généralement pas de remplir de données la page nouvellement instanciée. Il ne fait qu'allouer une page physique et l'attacher dans le segment à l'adresse appropriée. Normalement, c'est la CGC qui gère le remplissage des zones contenues dans cette page.

* * *

Dans la suite, nous décrivons d'abord une vue générale de l'organisation de la mémoire virtuelle dans AIX. En gardant cette organisation en tête, nous décrivons le déroulement du traitement des défauts de segment et celui des défauts de page. Ensuite, nous étudions deux problèmes supplémentaires associés au paginateur, à savoir : le mode de pagination à cohérence minimale, c'est-à-dire sans intervention de la CGC, et enfin la vérification de la protection.

III-3.2 L'organisation de la mémoire virtuelle AIX

Comme dans tout Unix, les processus AIX possèdent chacun leur propre espace d'adressage. La symbiose entre AIX et la famille de processeurs PowerPC se manifeste par un espace d'adressage par segments. Chaque espace est découpé en 16 segments dont le contenu est indiqué par un jeu correspondant de 16 registres. Un segment dans ce contexte est appelé *segment effectif*, pour le distinguer du *segment réel*, qui représente l'ensemble des pages qui sont effectivement couplées aux adresses correspondant à un segment effectif d'un processus. Un segment réel peut être par exemple un segment de mémoire partagée (`shmat`) ou un fichier couplé (`mmap`), ou encore un tampon d'entrée-sortie. Un segment réel a une existence en-dehors des processus qui le couplent, et est identifié par un *Segment Identifier* ou SID unique.

Chaque registre de segment contient une valeur parmi les suivantes.

- Rien : le segment effectif est vide.
- Réservé : le segment effectif est réservé par le système ; ce type est utilisé pour les segments fondamentaux contenant les données de gestion utilisées par le système telles les descripteurs de fichier.
- Code/données : chaque processus possède au moins un segment de code et deux segments de données, pour la pile et le tas, propres au processus.
- Segment : le registre de segment pointe directement vers un segment réel.
- Couplage : le registre de segment pointe vers une table de couplage qui découpe ce segment plus finement en un ensemble de segments réels. La figure III-10 présente ce cas.

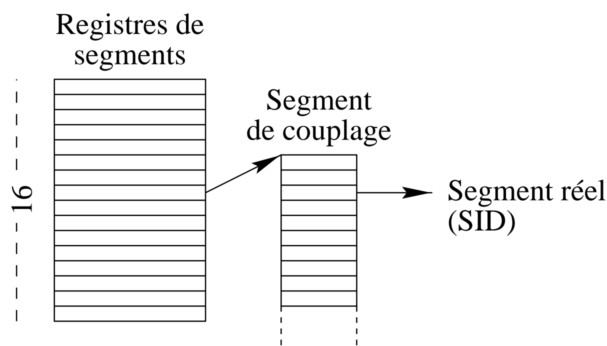

Figure III-10 : La mémoire virtuelle AIX 4.1 (32 bits)

Il faut noter que, dans sa version 32 bits, AIX 4.1 permet l'utilisation libre de 10 segments effectifs, les autres étant réservés pour son usage. Parmi ces segments libres, un certain nombre est réservé pour le couplage de la mémoire Arias, les autres restant disponibles pour l'usage propre de l'application. Il est important pour le couplage uniforme que les segments (c'est-à-dire les numéros de registres de segments) utilisés pour Arias soient toujours les mêmes pour tous les processus. Un processus qui utilise la mémoire Arias doit démarrer en appelant une fonction d'initialisation. Cette fonction a pour effet, entre autres, de réserver les segments qui servent à la mémoire Arias.

Dans son mode 64 bits, le processeur PowerPC 620 donne virtuellement à chaque processus 4096 segments effectifs en ajoutant une indirection supplémentaire. L'espace d'adressage est toujours couvert par 16 registres, mais chaque registre pointe en réalité vers une page d'indices de segments, qui sont les équivalents des registres de segments dans AIX 4.1. À raison de 256 indices par pages, on obtient donc l'équivalent de 4096 registres de segments. Chaque indice de segment peut pointer à son tour vers une table de couplage. Cette structure est illustrée sur la figure III-11.

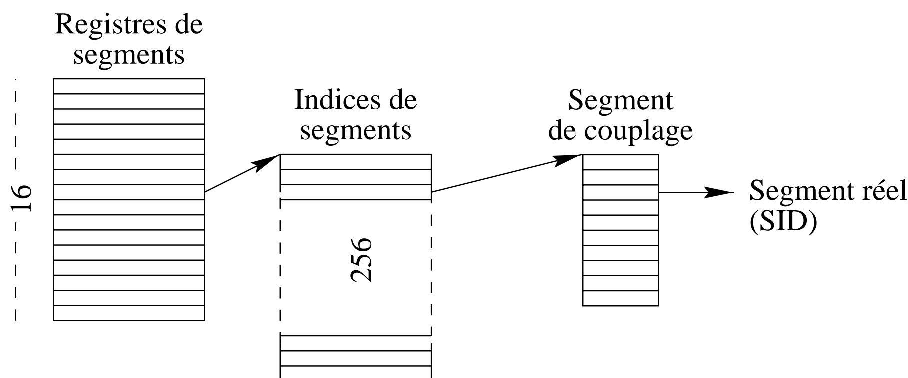

Figure III-11 : La mémoire virtuelle sur PowerPC 620 (64 bits)

Nous utilisons cette organisation en associant simplement un segment réel AIX à un

segment Arias. De cette manière, nous assurons toutes les caractéristiques suivantes :

- chaque processus a sa propre table de couplage,
- tout segment couplé par plus d'un processus est automatiquement partagé, et
- tout le mécanisme de couplage est assuré par des mécanismes natifs à AIX.

Gardons donc en mémoire qu'un segment Arias est désigné par un SID du point de vue d'AIX.

III-3.3 Les défauts de segment

Un défaut de segment signifie qu'un segment n'est pas encore couplé dans l'espace d'adresse du processus fautif. Ce segment peut cependant avoir déjà été couplé dans l'espace d'un autre processus, auquel cas il faut en partager la table de pages. Pour gérer l'association segment Arias $\leftrightarrow$ SID, le paginateur fait appel au module de localisation qui mémorise aussi ces associations parmi ses tables de localisation. Si un segment à coupler possède déjà un SID correspondant, il faut l'utiliser. Sinon, il faut créer un nouveau segment réel pour représenter le segment Arias.

Sous AIX, l'accès à une adresse qui ne correspond à rien dans la table de couplage provoque une exception de type DSI. Cette exception peut être interceptée par un traitant au niveau utilisateur s'exécutant dans un contexte restreint. Ce traitant se charge alors :

- d'identifier le segment correspondant, s'il existe (sinon, l'exception correspond à une erreur applicative fatale) ;
- de retrouver l'identifiant du segment AIX correspondant (c'est-à-dire son SID) ou d'en créer un nouveau si le segment n'avait jamais été couplé sur cette machine auparavant ; et
- d'insérer ce SID dans la table de couplage appropriée.

La première opération fait l'objet d'un appel au module de localisation de segments. En partant d'une adresse quelconque, ce dernier retrouve l'identité du segment Arias, c'est-à-dire son adresse de début et sa taille. Si ce segment possède déjà un SID, ce dernier est également renvoyé au paginateur. Dans le cas contraire, le paginateur doit, une fois le segment réel AIX créé, enregistrer la nouvelle association segment $\leftrightarrow$ SID auprès du module de localisation.

Muni du SID, le traitant d'exception doit réaliser le couplage effectif, c'est-à-dire insérer ce SID dans la table de couplage. Cependant, le contexte restreint dans lequel s'exécute le traitant ne lui permet pas d'invoquer l'appel système `mmap` qui permet de réaliser ce couplage. Cet appel système ne peut être utilisé que dans le contexte d'exécution normale du processus.

Par conséquent, chaque processus Arias possède une activité noyau (*kernel thread*) dont le rôle principal est d'effectuer le couplage. Cette activité noyau est créée par le biais de la fonction d'initialisation mentionnée plus haut. Elle est dormante la plupart du temps, mais pour effectuer un couplage de segment, le traitant d'exception lui communique le SID à insérer, la réveille et se bloque lui-même en attendant que la fin de l'opération de couplage. L'activité noyau, réactivée, invoque `mmap` puis se rendort à nouveau après avoir débloqué le traitant. Cela réactive le traitant, qui peut se terminer en signalant au système une exception correctement résolue.

III-3.4 Les défauts de page

À la sortie du traitant, le segment est couplé dans l'espace d'adressage du processus fautif. Le segment réel identifié par le SID n'est en réalité qu'une entrée dans la table de couplage qui pointe vers un traitant de défaut de page, sous la forme d'un point d'entrée appelé la *stratégie*.

Or le segment, bien que couplé en mémoire, n'est pas sous-tendu par des pages de données physiques. Le couplage effectué, le traitant d'exception rend la main après avoir déclaré l'exception résolue. Le système tente alors une ré-exécution de l'instruction fautive, qui cette fois-ci provoque un défaut de page, qui est aiguillée vers la stratégie.

La fonction de stratégie reçoit essentiellement deux paramètres :

- le SID du segment concerné et
- les coordonnées de la page manquante (déplacement et taille) dans ce segment.

Muni de ces informations, la stratégie a pour mission de remplir la page avec les données qu'elle doit contenir. Dans le cas d'un fichier couplé, par exemple, la stratégie doit lire les blocs correspondants depuis le disque. Dans le cas d'Arias, aucun transfert de données n'a à avoir lieu à ce moment. En effet, les données sont effectivement placées dans la page qu'en fonction du protocole de mise en cohérence de zones.

III-3.5 Le transfert de données

Une fois la page physique instanciée, on peut commencer à la remplir de données. En réalité, il faut revenir tout en arrière pour reprendre la description, et supposer que le défaut de page (ou de segment) n'était pas provoqué directement par l'application. En effet, cette dernière commence normalement par appeler une méthode de synchronisation dans un module spécifique, qui, via la CGC, provoque entre autres une requête de données. Ces données, une fois reçues, sont directement recopiées dans le segment destinataire par le pilote de communications (COM). C'est alors que se déclenche le défaut de page. L'instanciation d'une page physique a lieu, et la recopie est finalisée. À la suite de cela, l'opération de synchronisation s'achève, et l'application peut immédiatement

accéder aux données dont elle vient d'acquérir le droit d'accès par son appel de synchronisation.

Le seul point délicat dans cet enchaînement est le moment où le pilote de communications s'apprête à recopier les données. Le pilote COM réside dans le noyau en tant que pilote STREAMS. De ce fait, il n'a ni l'utilité ni les moyens de coupler dans son propre espace la mémoire Arias. Mais pour pouvoir recopier les données entrantes directement dans la page destinataire, il doit pouvoir coupler le segment réel temporairement. Il fait appel pour cela au paginateur, qui lui indique pour une adresse Arias donnée le SID correspondant. Il couple alors directement ce SID sur un registre de segment libre et effectue la copie, puis relâche le segment. Cette technique de couplage temporaire minimise le nombre de recopies de données par rapport à une technique de double copie qui passerait par un segment tampon créé par le pilote COM.

III-3.6 Interface de couplage

Comme le module PGR n'est pas un module STREAMS mais un ensemble de points d'entrées (traitant d'exception, système de fichiers externe et appels systèmes), son interface avec LOC est constituée d'un fonction directement appelable[14].

Comme nous l'avons vu en section II-8, le couplage des segments en mémoire est implicite, et est effectué au moment du premier accès par l'application. À ce moment, le paginateur a besoin de connaître les caractéristiques du segment (adresse de début, taille). Il s'adresse pour cela au localisateur.

LOC_Find (adresse, début, taille, ident) Cette fonction est appelée par le paginateur pour demander l'identification du segment qui correspond à l'adresse donnée en paramètre. Les paramètres **début**, **taille** et **ident** sont des paramètres de sortie renvoyés au retour de la fonction. Ils identifient le segment recherché par son adresse de début et sa taille, ainsi qu'un identifiant qui avait été enregistré via la fonction ci-dessous.

LOC_Register (adresse, ident) Cette fonction permet au paginateur d'≪enregistrer≫ un identifiant **ident** (en fait, le SID) d'un segment Arias. Cet identifiant est unique pour ce segment sur une machine donnée, et permet au paginateur de savoir qu'un segment est déjà couplé sur un processus sur la même machine.

* * *

Typiquement, le cycle de vie d'un segment couplé comporte trois étapes.

1. Tout premier accès à ce segment sur ce site : **LOC_Find** renvoie un **ident** nul.
2. Le paginateur couple ce segment sur un processus, et obtient d'AIX un SID, qu'il enregistre à l'aide de **LOC_Register**.

[14] De tels points d'entrée intra-noyau sont appelés *kernel service* dans la nomenclature AIX.

3. Pour les défauts de segment subséquent sur ce même segment, **LOC_Find** renvoie le SID du segment déjà couple en paramètre **ident**.

En utilisant le même SID, le partage des pages du segment entre plusieurs processus sur la même machine est automatiquement géré par AIX.

III-3.7 Segment Arias et segment AIX

Nous avons vu (III-1.2) qu'AIX impose une limite de 256 Mo aux segments, dans son mode 32 bits comme en 64. Nous avons vu que le mécanisme de couplage profite de l'identification unique segment Arias $\leftrightarrow$ SID (III-3.2). Si nous voulons des segments Arias de taille supérieure à 256 Mo, il faut mettre en œuvre une correspondance entre un segment Arias et plusieurs SID.

Une telle mise en œuvre rend la tâche du paginateur plus complexe, sans être impossible. Au lieu que l'identifiant de segment **ident** soit toujours le SID, il faut ajouter un niveau d'indirection et utiliser un identifiant indépendant, géré par le paginateur.

Nous n'avons pas, pour l'instant, poursuivi plus loin cette ligne de recherche. En effet, d'un point de vue pratique, les limitations du mode 32 bits rendent de tels segments gigantesques peu intéressants.

III-4 Synthèse

Nous avons décrit en détail la mise en œuvre des structures de données et des algorithme pour la localisation et l'allocation des segments. Un certain nombre de paramètres restent à régler. Récapitulons-les.

- Tailles maximale et minimale des segments.
- Géométrie des tables de hachage ; seuils de rehachage.
- Géométrie du buddy-system.

Le chapitre suivant décrit et analyse des mesures et des simulations qui ont été effectués en vue d'apporter des indications sur le comportement de ces structures et de ces algorithmes pour divers jeux de paramètres et de types d'utilisation.

Chapitre IV

Paramétrage, mesure et évaluation

Ce chapitre répond par l'expérimentation aux questions énumérées en III-4. Nous avons concentré les expériences sur deux axes : les mesures temporelles et les mesures spatiales[15]. À propos des mesures temporelles, nous commençons par analyser les performances de l'environnement STREAMS sous AIX 4.1. Cela nous permet de mettre en valeur le rapport réseau/processeur dans les performances.

IV-1 AIX 4.1 et STREAMS

Avant de mesurer les performances propres du module étudié, nous commençons par une présentation critique du système utilisé, à savoir AIX 4.1 et son environnement STREAMS.

IV-1.1 Pourquoi STREAMS ?

Le choix de l'environnement STREAMS [4] pour la mise en œuvre d'Arias découle de plusieurs contraintes et objectifs :

- ne pas modifier le système AIX lui-même,
- favoriser la portabilité,
- favoriser la modularité,
- permettre l'exécution sur un multiprocesseur.

En particulier, la première contrainte était importante, dans la mesure où nous ne disposions pas librement du code source d'AIX.

[15] Par «mesure spatiale» nous entendons les caractéristiques telles que la taille des méta-données ou consommation d'espace virtuel.

Les avantages de l'environnement STREAMS reflètent presque exactement les points énumérés en haut :

- c'est un environnement standard pour l'insertion de fonctions dans le noyau sans modifier celui-ci ;
- il s'articule autour de la notion de modules logiciels communicant à travers une interface standard ;
- dans sa version AIX 4.1, il fonctionne aussi bien sur un multiprocesseur que sur un mono-processeur.

Ces avantages sont contrebalancés par un défaut majeur du point de vue des performances. Comme le démontre en détail la thèse de V. Roca [69], l'environnement STREAMS n'est pas toujours propice aux applications de hautes performances. Ce jugement est plus mitigé dans le cas des multiprocesseurs, car la parallélisation vient naturellement dans l'environnement STREAMS, alors qu'avec une structuration monolithique elle nécessite un portage pour le moins complexe[16].

IV-1.2 Chiffres clés

Le tableau IV-a résume les quelques chiffres clés qui caractérisent les STREAMS sous AIX 4.1. La signification de ces chiffres est détaillée ensuite.

a. Aller-retour de message vide sur pile STREAMS minimale	160μs
b. Transfert de message vide module à module	13μs
c. Temps de traitement du multiplexeur Arias	245μs
d. Aller-retour de message de 150 octets par le pilote COM	1500μs
e. Appel système	20μs

Tableau IV-a : Chiffres clés des performances des STREAMS sous AIX 4.1

IV-1.2-a Pile STREAMS minimale

Ce test révèle le temps de traitement minimal pour l'acheminement d'un message de longueur nulle depuis l'espace utilisateur vers une pile STREAMS minimale constituée d'un *Stream-head* (interface application/noyau) et d'un pilote (figure IV-1).

[16]Cela dit, nous ne disposions pas de multiprocesseurs pour nos essais. Cependant, la possibilité de l'utilisation d'Arias sur un multiprocesseur le rendait attrayant pour certaines applications industrielles comme les très grandes bases de données.

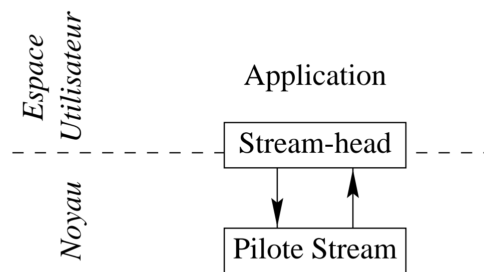

Figure IV-1 : Pile STREAMS minimale

IV-1.2-b Module à module

À la pile STREAMS minimale sont ajoutés deux modules «vides» qui se contentent de laisser passer le message. Le temps de transfert de module à module mesure le temps de traitement des fonctions STREAMS qui permettent aux modules de communiquer entre eux.

IV-1.2-c Multiplexeur

La présence de plusieurs modules spécifiques de cohérence (I-4.6) est gérée par le module multiplexeur AGM (cf. architecture générale, section I-6). Le temps de traitement dans un multiplexeur STREAMS, ainsi que le temps de traitement spécifique au décodage du multiplexage Arias, sont représentés par cette mesure.

IV-1.2-d Pilote de communications

L'architecture STREAMS d'Arias (I-6) a pour fondation un pilote STREAMS chargé d'aiguiller les communications d'une machine à une autre (le module COM). Ce test mesure le temps de transfert pour l'aller-retour d'un message de longueur nulle.

IV-1.2-e Appel système

Pour comparaison, un appel système simple sans paramètres prend environ 20μs sous AIX.

* * *

On remarque immédiatement, et ce n'est pas une surprise, que la latence du réseau est un ordre de grandeur supérieur au temps de transit dans la pile STREAMS, qui lui-même est très long par rapport à un simple appel système. La conséquence en est que l'évaluation temporelle du prototype actuel est délicate, dans la mesure où il est difficile de séparer les temps de traitement réel des surcoût causés par les STREAMS. Nous devons tenir compte de ces réserves en analysant les mesures obtenues.

IV-2 Paramétrage

Récapitulons les paramètres à régler, leur incidence probable sur le comportement du système et leurs valeurs admissibles. Gardons en tête que l'objectif des structures de localisation (le *fichier*) est de retrouver la fiche (II-2) correspondant à un segment.

IV-2.1 Partitions

Le choix du partitionnement, bien que modifiable en cours d'exécution, doit répondre à deux contraintes :

- au moins une partition par machine (III-1.4), et
- suffisamment de partitions pour une marge d'opération : l'ajout et la suppression de nouvelles partitions ; en effet, la partition est le grain de distribution de charge de localisateur (II-4.3).

En réalité, le partitionnement est très restreint en mode 32 bits. À cause de l'étroitesse de l'espace d'adressage, il n'est pas très significatif d'avoir un grand nombre de partitions artificiellement réduites.

IV-2.2 Blocs

Taille du bloc La taille du bloc (III-1.3) est limitée par la taille maximale d'un segment AIX, si l'on évite la complication des blocs multi-segments (III-3.7). Sous AIX 4.1 cette taille est donc de 256 Mo, soit 2^{28} octets.

Taille maximale de segment La taille du bloc est aussi la taille maximale d'un segment. Pour beaucoup d'application, la taille de 256 Mo est adéquate. Néanmoins, pour certaines applications, il peut être intéressant de diminuer cette taille, de manière à pouvoir avoir des blocs plus petits.

Consommation minimale de segment (CM) La CM est au moins une page (4096 octets).

Nous avons vu les avantages d'un buddy-system non binaire (II-4.2-b). Le choix d'une base doit tenir compte de cet avantage d'une part, mais aussi du fait que plus la base est grande, moins il faut de niveaux pour décoder le même nombre de bits total.

IV-2.3 Tables de hachage

IV-2.3-a Fonction de hachage

Dès que la taille de la table de hachage est choisie, la fonction de hachage (III-1.2-a) en découle immédiatement. En effet, une table de taille 2^n impose une fonction de hachage qui découpe la clé par blocs de n bits (voir figure III-1). Cela implique bien entendu que les tables aient pour taille une puissance de 2.

IV-2.3-b Table de premier niveau

La taille de la première table est fixe (III-1.2-c). Le choix de cette taille dépend du type d'utilisation, en particulier du nombre de blocs dont on anticipe la présence à tout instant sur un site. Voici quelques exemples.

- Si l'on anticipe un très grand nombre de blocs (beaucoup de segments très dispersés, très peu utilisés), par exemple un million, on prendra une taille plutôt grande pour la première table, par exemple 1024 entrées. Cela entraîne, si l'on dispose d'une bonne fonction de hachage, des tables de second niveau de taille moyenne de l'ordre 1000 entrées.
- Pour un nombre de blocs plus raisonnable, par exemple 100.000, on choisira une taille plus petite, par exemple 512 entrées, qui coïncide avec la taille d'une page physique (chaque entrée nécessite 16 octets).

IV-2.3-c Niveaux supplémentaires

Le nombre de niveaux supplémentaires dépend de la taille de la première table. Si la première table est petite, il faut le compenser en ajoutant éventuellement un niveau. Même pour un très grand nombre de fiches[17], donc de blocs, il semble qu'il soit rare d'avoir besoin de deux niveaux supplémentaires, et qu'un seul niveau secondaire de tables suffise.

Dans tous les cas, les tables du dernier niveau sont de taille variable. La seule contrainte est que cette taille ne coïncide pas avec une taille déjà utilisé en amont dans la hiérarchie, par souci de collision (III-1.2-b).

IV-2.3-d Seuils de rehachage

Pour qu'une technique de hachage fonctionne de manière satisfaisante, c'est-à-dire avec un bon compromis espace/temps [74, p. 241], les tables de hachages doivent maintenir leur taux de remplissage dans une certaine plage, généralement comprise entre 50 et 90%.

[17] N'oublions pas que la table de localisation comprend les accélérateurs (II-4.5).

Pour maintenir les tables à des taux de remplissage raisonnables, nous utilisons deux seuils :

- un seuil supérieur S_1, au-delà duquel la table est agrandie d'un facteur 2 ;
- un seuil inférieur S_0, en-deçà duquel la table est réduite d'un facteur 2.

Il est évident que pour éviter des allers-retours incessants, il faut avoir une hystérésis, c'est-à-dire que $2S_0 < S_1$. Cette inégalité n'est pas respectée si nous prenons par exemple $S_1 = 80\%$ et $S_0 = 45\%$. Lorsqu'une table atteint un taux de remplissage de 80%, sa taille est doublée, et son taux de remplissage chute alors à 40%, qui est en-dessous du seuil inférieur. On voit donc qu'il est important que l'inégalité soit respectée strictement, voire avec une petite marge.

Un autre paramètre peut conditionner le seuil de rehachage : la longueur maximale de chaîne de collision. En fixant cette limite relativement bas, on encourage le rehachage vers les tables plus grandes et donc moins remplies. En fixant cette limite plus haut, on favorise la compacité des tables au détriment du temps de recherche.

IV-2.4 Paramétrage de base

Nous avons choisi un paramétrage de base en mode 32 bits, choisi de manière intuitive pour être raisonnablement équilibré, pour servir de point de départ des expériences. Sa limitation principale tient dans la taille maximale des segments, qui est de 1 Mo.

- Taille des partitions : 28 bits (256 Mo).
- Taille des blocs (taille maximale des segments) : 20 bits (1 Mo).
- Tables de hachage : deux niveaux, 64 entrées dans la première table.
- Seuils de rehachage : 90 et 40% ; longueur maximale de chaîne : 30.
- Buddy-system : deux niveaux de 16 entrées.

IV-3 Évaluation temporelle

Notre «banc d'essai temps» se compose d'une série de micro-tests destinés à mesurer les opérations fondamentales.

IV-3.1 Micro-test : création de segment

Le test de base a été effectué sans mémoire réelle, pour examiner uniquement le processus de création de segment dans le fichier. Ensuite, nous avons ajouté la CGC dans notre environnement d'essai : la création de segment est alors accompagnée de celle de son descripteur.

IV-3.1-a Paramétrage de base

En utilisant le paramétrage indiqué en IV-2.4, nous avons testé la création de segments toujours locale (c'est-à-dire sans emprunt). Ce micro-test donne les résultats indiqués dans la table IV-b, où l'on a également reporté les temps d'aller-retour simple de message STREAMS.

	Sans CGC	Avec CGC
Aller-retour simple	245μs	270μs
Création de segment	280μs	325μs

Tableau IV-b : Création de segments, paramétrage de base

Ces tests ont été effectués en utilisant deux tailles de segments, 4 Ko et 64 Ko, et donnent des résultats tout à fait similaires dans les deux cas. De plus, ces temps restent remarquablement constants quel que soit le nombre de segments déjà présents dans le système : la création du 100.000-ième segment prend autant de temps que celle du premier.

IV-3.1-b Variation de paramétrage

Nous avons essayé une variation du paramétrage en passant le buddy-system à quatre niveaux de quatre entrées, en gardant la CGC. Cela donne les résultats résumés dans la table IV-c. Cette fois-ci, nous avons distingué les cas de segments de 4 Ko des segments de 64 Ko, car les résultats sont légèrement différents.

	Sans CGC	Avec CGC
4 Ko	280μs	330μs
64 Ko	275μs	320μs

Tableau IV-c : Création de segment, changement de paramètre

IV-3.2 Micro-test : localisation

La mesure du temps de localisation a fait l'objet de la même procédure que pour la création. La CGC n'a pas été chargée pour ce test, car sa contribution au temps de traitement est pratiquement constant. Les résultats sont résumés dans la table IV-d.

IV-3.3 Analyse

Ces chiffres donnent quelques indications :

	4 Ko	64 Ko
Paramètres de base	265μs	265μs
Paramètres modifiés	275μs	260μs

Tableau IV-d : Localisation de segment

- l'intervention de la CGC dans la création de segment contribue environ 40μs au temps de traitement, parmi lesquels il faut compter 25μs pour la simple présence du module ;
- les temps de traitement spécifiques au module étudié sont petits par rapport au temps global de traitement STREAMS, ce qui rend leur analyse difficile ;
- à cette résolution temporelle, les temps de traitement sont pratiquement indépendants du nombre de segments présents dans le fichier.

Toutes ces mesures ont été effectuées dans le cadre strictement local. Au vu de l'ordre de grandeur des variations observées, nous n'avons pas étudié le comportement temporel dans les cas où des messages réseau sont nécessaires. Le temps de transfert des messages noierait complètement les variations que nous cherchons à mesurer. Pour ces cas, le comportement temporel est mieux caractérisé par le nombre et éventuellement la taille des messages nécessaire que par les temps de calcul.

Or, de par le système des emprunts (II-5.1), l'activité réseau au cours de la création de segments est

- soit entièrement asynchrone, si le mécanisme d'emprunt utilise des blocs de réserve,
- soit épisodiquement bloquante, si les emprunts sont faits à la demande[18].

Il s'avère donc que la fréquence des messages réseau est fonction du taux de remplissage des blocs, que nous étudions dans la section suivante.

IV-4 Évaluation spatiale : système de fichiers

Les caractéristiques générales des tables de hachages et du buddy-system ont déjà été analysées aux sections III-1.2 et III-1.3. Pour résumer, le paramétrage des tables de hachages est essentiellement un compromis entre le taux de remplissage (compacité) et la vitesse de recherche (le nombre de collisions). Le paramétrage du buddy-system est quant à lui un compromis entre la compacité et la consommation d'espace virtuel.

Les caractéristiques que nous utilisons pour cette évaluation sont les suivants.

[18] Lorsqu'un bloc emprunté est plein, il faut le rendre et en emprunter un autre.

- Blocs : nombre de blocs utilisés (N_{bloc}), taux de remplissage des blocs ($\%_{bloc}$), nombre moyen d'entrées utilisées par segment (B_{seg}).
- Hachage : Nombre de tables de taille n (N_n), taux de remplissage des blocs ($\%_{hash}$), longueur moyenne des collisions (ℓ_{coll}).
- Utilisation de l'espace virtuel, exprimée en pourcentage d'espace virtuel réellement utilisé par rapport à l'espace consommé ($\%_{VM}$).

IV-4.1 Jeux d'essai

Pour affiner ces compromis, nous avons soumis ces structures à un jeu d'essai de segments. À la manière des expériences de Baker sur les systèmes de fichiers [5], notre jeu d'essai est constitué d'un ensemble de répertoires trouvés sur notre serveur de fichiers. Nous avons construit trois exemples ayant des caractéristiques différentes :

arias un état des sources d'Arias, comme exemple de fichiers de petite taille ;

bin un répertoire de fichiers exécutables (`/usr/local/bin`), pour ses grands fichiers ;

home une partition de répertoires personnels (`/home`) pour son très grand nombre de fichiers.

Les caractéristiques de ces exemples sont regroupées dans la table IV-e. La distribution des tailles de segment est illustrée sur la figure IV-2 pour le premier exemple (**arias**). L'allure de la distribution est très similaire pour les quatre exemples.

		Taille (octets)		Médiane	
	Segments	Totale	Moyenne	Segments	Octets
arias	715	5.496.375	7.687	2.345	18.313
bin	865	106.030.743	122.578	27.898	967.806
home	17.396	438.357.514	25.198	2.355	498.518

Tableau IV-e : Exemples de systèmes de fichiers

Dans ce tableau, on a indiqué deux médianes, par segments et par octets. Leur définition peut être exprimée ainsi.

- La moitié des segments ont une taille inférieure à la médiane par segments.
- La moitié des octets utilisés le sont par des segments de taille supérieure à la médiane par octets.

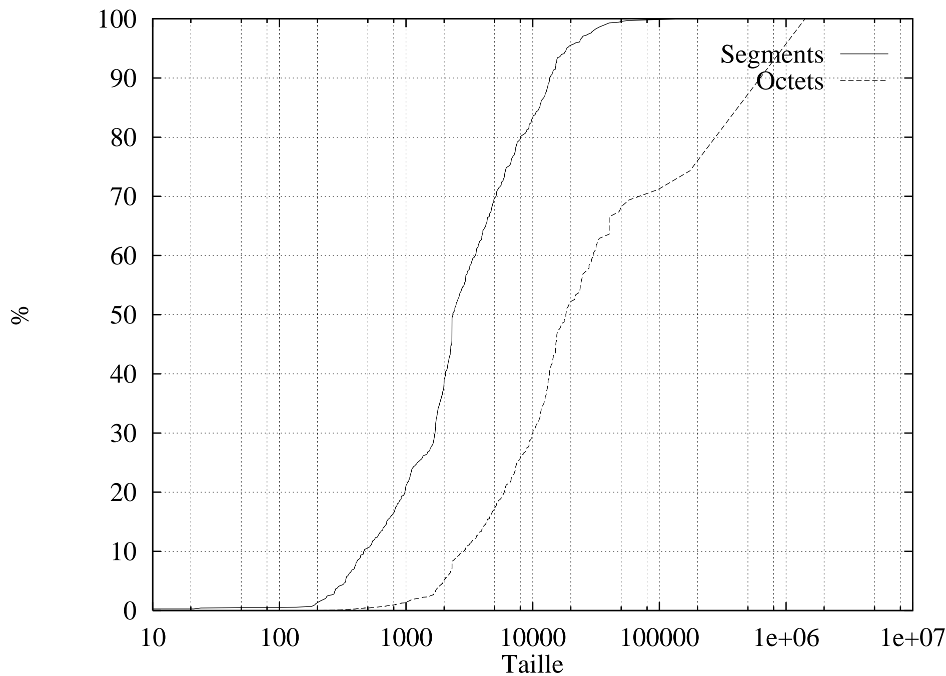

Figure IV-2 : Distribution des tailles de fichier dans un jeu d'essai

IV-4.2 Mode 32 bits

En mode 32 bits, la taille des segments est limitée à 1 Mo. Certains jeux d'essai ci-dessus contiennent des segments qui ont dû être rejetés à cause de cela. Ces jeux d'essai ajustés pour le mode 32 bits ont les caractéristiques de la table IV-f.

			Taille (octets)		Médiane	
	Segments	Rejetés	Totale	Moyenne	Segments	Octets
arias	714	1	4.085.053	5.721	2.343	12.978
bin	849	16	53.203.594	62.666	27.672	198.952
home	17.236	160	278.621.168	16.165	2.346	147.063

Tableau IV-f : Jeux d'essai ajustés pour le mode 32 bits

Le tableau IV-g résume les résultats pour les trois jeux d'essai, chacun en utilisant trois structures de bloc :

- en quatre niveaux de 4 entrées,
- en deux niveaux (16×16), et
- en un seul niveau (de 256 entrées).

		N_{bloc}	$\%_{bloc}$	B_{seg}	$\%_{VM}$
arias	$4 \times 4 \times 4 \times 4$	7	90,3	2,2	85,3
	16×16	6	92,4	2,0	92,2
	256	6	91,4	2,0	99,7
bin	$4 \times 4 \times 4 \times 4$	84	80,8	3,0	62,7
	16×16	72	74,5	6,0	73,2
	256	59	89,0	15,9	89,6
home	$4 \times 4 \times 4 \times 4$	414	88,1	2,0	73,9
	16×16	379	86,2	2,2	80,5
	256	342	89,3	4,5	89,3

Tableau IV-g : Utilisation spatiale de l'allocation

L'analyse de ce tableau permet de remarquer que l'augmentation de la taille des niveaux individuels :

- augmente le taux d'utilisation de l'espace d'adressage,
- augmente le nombre d'entrées utilisées, même si leur taux d'utilisation augmente, et
- diminue le nombre de blocs utilisés.

On constate que le jeu d'essai **arias** est une exception, car la taille des méta-données reste remarquablement constante quelle que soit la géométrie du bloc. Cela est dû au fait que la taille des segments qui constituent ce jeu d'essai est très faible. Environ 70% des segments sont de taille inférieure à celle d'une page, donc n'occupent qu'une entrée dans le buddy-system. De plus, dans le cas de la géométrie $4 \times 4 \times 4 \times 4$, un segment occupe au plus 3 entrées, puisque s'il devait occuper 4 entrées, alors il serait pris en charge par une entrée au niveau supérieur (voir figure IV-3).

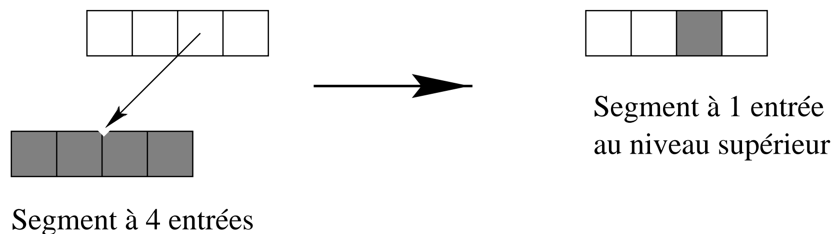

Figure IV-3 : Segment à 4 entrées dans un buddy-system de largeur 4

Le cas du jeu **home** semble le plus intuitif. En effet, au fur et à mesure que la géométrie «s'élargit» (passant de 4 niveaux à 4 entrées en un seul niveau à 256 entrées), le taux d'utilisation de l'espace virtuel augmente, et même temps que le nombre d'entrées par segment, indicatif de la taille des méta-données, augmente aussi.

Enfin, le cas **bin** est assez particulier. Il est constitué de segments relativement grands (cf. figure IV-4), dont environ 50% des segments ont une taille comprise entre 16.000 et 65.000 octets. Cela signifie qu'avec une géométrie de 16×16, 50% des segments utilisent plus de deux entrées au second niveau.

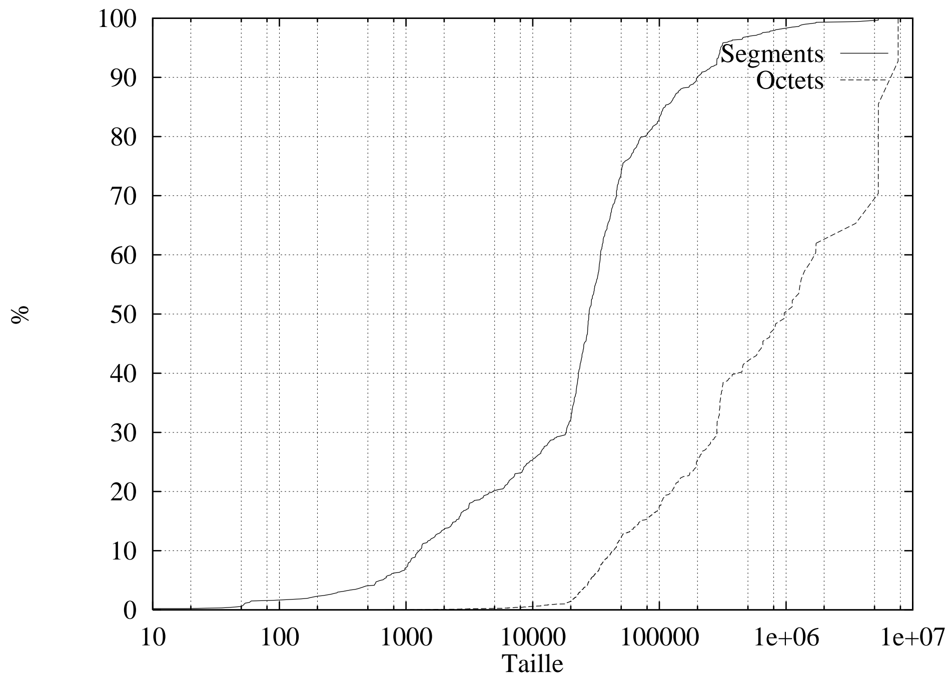

Figure IV-4 : Distribution des segments dans le jeu d'essai **bin**

* * *

Le tableau IV-h donne deux résultats pour un buddy-system binaire (que nous avions écarté d'emblée). On constate que ces résultats sont assez proches de ceux obtenus pour la configuration $4 \times 4 \times 4 \times 4$.

	N_{bloc}	$\%_{bloc}$	B_{seg}	$\%_{VM}$
bin	82	86,3	3,5	64,8
home	427	89,9	2,6	71,7

Tableau IV-h : Utilisation spatiale de l'allocation (buddy-system binaire)

Pour le cas **home**, en comparaison avec la géométrie 4^4, la solution binaire est moins bonne sur tous les plans : le nombre d'entrées par segment est plus grand, et le taux d'utilisation de l'espace virtuel est plus faible. Mais ces effets sont très faibles, et même dans le cas de **bin** l'utilisation de l'espace virtuel est meilleure.

IV-4.3 Mode 64 bits

Pour réellement utiliser le grand espace, nous avons synthétisé un jeu d'essai particulièrement exigeant, constitué de l'ensemble des fichiers d'un serveur de fichiers NFS sur notre site. Il comporte 100.916 fichiers pour une taille totale de 3.913.045.805 octets, soit une moyenne de 38.775 octets par segment. La moitié des fichiers font moins de 4438 octets, et la moitié des octets occupés le sont par des fichiers de plus de 859.676 octets. La distribution des tailles de fichiers est illustrée sur la figure IV-5.

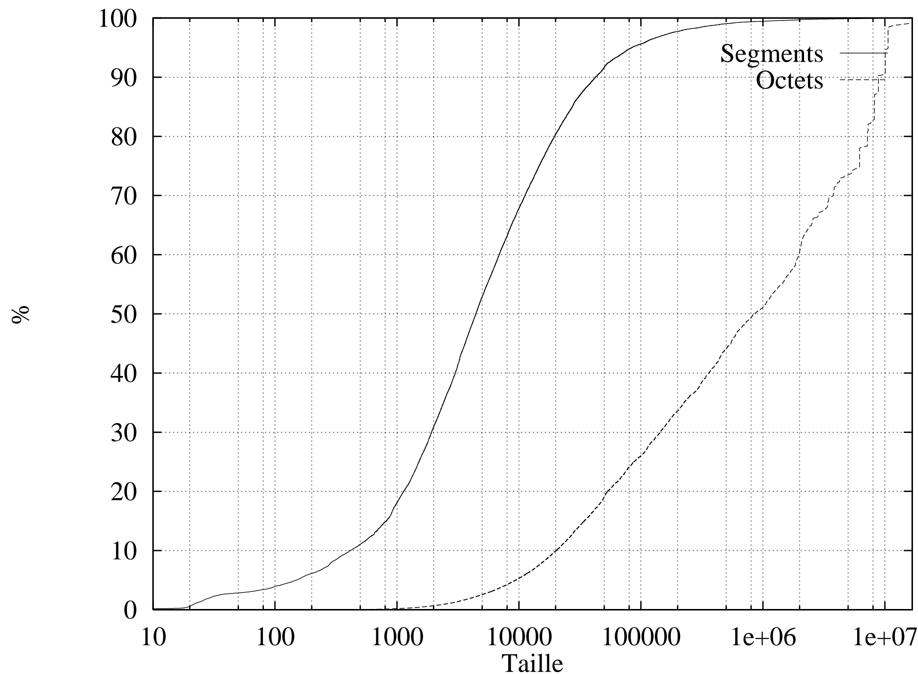

Figure IV-5 : Distribution des tailles de fichiers dans une grande partition

Les résultats sont regroupés sur le tableau IV-i. La colonne ≪CM≫ indique la consommation minimale de mémoire virtuelle par segment, qui est soit d'une page (2^{12} octets), soit de 2^{16} octets dans nos essais. La colonne ≪Taille≫ indique la *taille* des blocs, c'est-à-dire la quantité d'espace virtuel couvert par un bloc (cf. III-1.3).

On constate que ces résultats (en mettant de côté le nombre de blocs) sont assez similaires à ceux obtenus dans le mode 32 bits pour des configurations comparables. Il est clair que si l'on passe la consommation minimale à 2^{16} octets, $\%_{\mathrm{VM}}$ s'effondre. En effet, ce jeu d'essai est composé pour 50% de segments de taille inférieur à 2.400 octets.

Si l'on prend le jeu d'essai **bin**, qui contient des segments plus grands, on obtient des résultats plus satisfaisants (tableau IV-j) même avec un CM de 2^{16} octets (la taille des blocs étant toujours égale à 2^{28} octets).

Le tableau IV-k récapitule les résultats obtenus pour le jeu d'essai **serveur**, spécialement construit pour le mode 64 bits. Pour commencer, on peut constater que même ce jeu

CM	Taille	Structure	N_{bloc}	$\%_{bloc}$	B_{seg}	$\%_{VM}$
2^{12}	2^{28}	$16 \times 16 \times 16 \times 16$	3	88,9	2,2	76,7
		$256 \times 16 \times 16$	3	88,5	2,2	78,8
		$16 \times 256 \times 16$	3	89,8	2,3	87,2
		256×256	3	89,1	4,5	87,2
	2^{24}	$16 \times 16 \times 16$	38	88,6	2,2	75,8
		256×16	33	89,6	2,3	87,2
2^{16}	2^{28}	$16 \times 16 \times 16$	6	97,5	1,2	30,9
		256×16	6	97,3	1,2	31,4
	2^{24}	16×16	92	97,3	1,2	31,2

Tableau IV-i : Utilisation spatiale (64 bits, **home**)

Structure	$\%_{bloc}$	B_{seg}	$\%_{VM}$
$16 \times 16 \times 16$	91,9	1,8	64,8
256×16	87,0	1,7	66,9
16×256	93,8	2,5	74,2

Tableau IV-j : Utilisation spatiale (64 bits, **bin**)

d'essai apparemment gigantesque (il n'aurait pas pu fonctionner en mode 32 bits, puisqu'il nécessite plus de 2^{32} octets d'espace virtuel) est aisément géré dans une vingtaine de blocs de 2^{28} octets. Nous avons réduit la taille des blocs à 2^{24} octets pour «pousser» le système[19].

Relevons deux faits notables.

- Le nombre d'entrées utilisées en moyenne par segment (B_{seg}) est remarquablement bas dans les configurations «étroites» (c'est-à-dire avec des niveaux de 16 entrées).
- Le taux d'utilisation de l'espace virtuel ($\%_{VM}$) est plus faible pour les configurations «étroites».

De manière générale, ces deux caractéristiques sont contradictoires. En effet, on ne peut pas à la fois diminuer B_{seg} et augmenter $\%_{VM}$. Notre préférence va vers plus de gâchis d'espace d'adressage en faveur d'une plus grande compacité des blocs. Mais cette démarche a aussi ses limites, comme le montre le fait que l'on ne gagne pas en compacité en passant de $256 \times 16 \times 16$ à $16 \times 16 \times 16 \times 16$: les deux premières lignes du tableau IV-k indiquent les mêmes valeurs de $\%_{bloc}$ et B_{seg}.

[19]Notons que même dans ce jeu d'essai, aucun fichier ne dépassait la taille de 2^{24} octets (16 Mo), ce qui a permis d'utiliser cette taille comme taille de bloc.

CM	Taille	Structure	N_{bloc}	$\%_{bloc}$	B_{seg}	$\%_{VM}$
2^{12}	2^{28}	$16 \times 16 \times 16 \times 16$	20	86,8	2,8	80,8
		$256 \times 16 \times 16$	18	86,8	2,8	86,4
		$16 \times 256 \times 16$	18	86,9	3,0	87,9
		256×256	17	90,4	5,1	93,6
	2^{24}	$16 \times 16 \times 16$	321	86,2	3,0	77,1
		256×16	294	86,3	3,3	84,3
2^{16}	2^{28}	$16 \times 16 \times 16$	38	97,3	1,2	41,6
		256×16	37	97,4	1,2	42,7
	2^{24}	16×16	612	96,8	1,2	40,4

Tableau IV-k : Utilisation spatiale (64 bits, **serveur**)

IV-4.4 Synthèse

Nous nous sommes attachés dans cette section à étudier le comportement spatial du buddy-system à géométrie variable face à une charge de création de segment réaliste. Nous avons fait varier d'une part la géométrie des blocs, d'autre part le paramètre CM dans le mode 64 bits.

Une première analyse a montré que, dans presque tous les cas, il existe un compromis à trouver entre le taux d'utilisation de l'espace d'adressage et la taille des méta-données.

Un premier ajustement est possible en «rétrécissant» le buddy-system, c'est-à-dire en utilisant plusieurs niveaux de petites tables, comme par exemple la géométrie $16 \times 16 \times 16 \times 16$. Ce faisant, nous obtenons deux effets :

- diminution du taux d'utilisation de l'espace virtuel, et
- diminution du nombre moyen d'entrées utilisées par segment.

Un second ajustement consiste à augmenter la consommation minimale d'espace virtuel par segment. Cet ajustement revient à diminuer le nombre de bits à faire décoder par le buddy-system, donc le nombre de ses niveaux. On obtient alors :

- une forte diminution du taux d'utilisation de l'espace virtuel, et
- une diminution aussi forte du nombre moyen d'entrées par segment.

Comparons les deux extrêmes pour estimer la plage des possibilités. Le tableau IV-l reproduit les deux lignes significatives du tableau IV-k (les quatrième et septième lignes). Le taux d'utilisation de l'espace virtuel chute de 93,6% à 41,6%, ce qui diminue la «durée de vie» du système de moitié environ (cf. I-3.1). Cependant, le nombre moyen d'entrées présents par segment, donc la taille des méta-données, est divisé par 4, ce qui peut être un gain appréciable de mémoire physique.

CM	Structure	N_{bloc}	$\%_{bloc}$	B_{seg}	$\%_{VM}$
2^{12}	256×256	17	90,4	5,1	93,6
2^{16}	$16 \times 16 \times 16$	38	97,3	1,2	41,6

Tableau IV-l : Deux cas extrêmes d'organisation des blocs

IV-5 Évaluation spatiale : insertion aléatoire

Cette étude porte surtout sur le comportement des tables de hachage en présence d'une grande dispersion des clés. Elle n'a vraiment de sens que pour le mode 64 bits, puisqu'en mode 32 bits les tables de hachage ne gèrent que relativement peu d'entrées (12 bits à décoder).

Nous avons testé l'insertion de blocs de manière aléatoire dans l'ensemble de l'espace virtuel. Nous avons simulé l'insertion de 100.000 à 500.000 blocs et mesuré le nombre, la taille et le taux de remplissage des tables de hachage intermédiaires[20]. Nous avons également mesuré le taux de collision par le biais de la longueur moyenne des chaînes de recherche. Enfin, la première table de hachage avait 1024 entrées.

Les résultats sont regroupés dans le tableau IV-m. Ces expériences ont été faits avec des seuils de rehachage de 90 et 40%, et une longueur maximale de chaîne de 30.

- N_n : nombre de tables de taille 2^n.
- $\%_{hash}$: taux de remplissage.
- ℓ_{coll} : longueur moyenne des chaînes de recherche (serait égal à 1 s'il n'y avait aucune collision).

N_{bloc}	$\%_{hash}$	ℓ_{coll}
100.000	72.7	1.5
200.000	75.6	1.5
300.000	57.1	1.2
400.000	75.8	1.5
500.000	49.5	1.4

Tableau IV-m : Allocation aléatoire

On constate qu'il y a de nets effets de seuil (cas de 300.000 et 500.000), mais qui se résolvent au fur et à mesure des insertions. On constate aussi qu'avec les paramètres choisis, le taux de collision est très bas. Le choix de seuils de rehachage moins exigeants (95 et 45%, longueur de collision illimitée) ne donne pas de résultats très différents : le taux d'utilisation est légèrement meilleur, et ℓ_{coll} ne dépasse pas 2.

[20]Pour mémoire, la taille de la première table de hachage (III-1.2-c) est fixe.

Nous avons également vérifié que le partitionnement n'avait pas d'impact sur les caractéristiques des tables de hachage. Pour cela, nous avons effectué la même expérience en limitant les adresses produites par le générateur aléatoire à un nombre restreint de partitions. Cela a encore produit des résultats très semblables au tableau IV-m.

* * *

Cet effet de seuil noté ci-dessus est dû au fait que la distribution des accès était supposée totalement uniforme. En effet, une distribution d'accès uniforme va entraîner une croissance uniforme des tables de hachage de second niveau par l'insertion des fiches et des accélérateurs correspondant. C'est donc toutes au même moment que ces tables atteindront leur seuil de rehachage.

La recherche d'une structure qui permettrait d'éviter cet effet figure parmi les projets d'amélioration du système.

Chapitre V

Conclusions et perspectives

V-1 Récapitulatif

Nous avons réalisé un service de mémoire partagée répartie appelé Arias, dans le cadre du projet SIRAC sur les supports logiciels pour les systèmes répartis.

Nous avons commencé ce rapport avec un historique des systèmes de mémoire partagée répartie, puis avons mis en perspective les objectifs d'Arias par rapport aux projets passés et actuels. Nous avons déterminé l'intérêt d'un service de MPR répondant aux objectifs dont nous rappelons les principaux :

- réaliser une MPR persistante et uniforme fondée sur l'allocation de mémoire indifférenciée,
- rendre transparents la localisation et le nommage par adresse des données en mémoire,
- intégrer ce service à l'intérieur d'un système existant en minimisant l'impact sur celui-ci.

Ces objectifs sont réalisés par une conception s'articulant autour d'une entité de grain intermédiaire, le *segment*. Le problème central auquel nous nous sommes intéressé est celui de l'allocation et la localisation de segments de mémoire. Nous avons passé en revue plusieurs structures d'indexation qui permettraient à la fois de référencer et d'allouer un grand nombre de segments dispersés dans un grand espace d'adressage. Notre solution comporte trois niveaux de grain qui résout chacun un problème particulier.

- La *partition* permet de répartir la gestion des segments à travers les machines.
- Le *bloc*, de taille fixe, se prête bien à la technique de hachage d'adresse et facilite l'allocation distante.
- Le segment dans les blocs sont référencés par un buddy-system, qui procure une simplification de l'allocation tout en étant compact.

Cette solution a été décrite et justifiée dans tous ses détails techniques, et a fait apparaître un certain nombre de paramètres ajustables. Nous avons alors vu que le bon choix de ces paramètres dépend des types d'applications à servir, et nous avons rapporté un certain nombre de mesures temporelles et spatiales pour acquérir une intuition sur le comportement du système sous divers paramétrages.

V-2 Quelques leçons apprises

Une des contraintes importantes que nous nous étions fixée est celui de ne pas modifier le système préexistant, à savoir AIX 4.1. Cette contrainte découle d'un souci de portabilité, mais aussi et surtout de la raison pragmatique que nous ne disposons pas librement des sources d'AIX. Malgré cela, grâce à l'environnement STREAMS et aux possibilité d'extension du noyau AIX[21], nous avons réalisé un prototype d'Arias fonctionnant au sein du système et s'appuyant sur ses mécanismes fondamentaux. Il est important de souligner que l'extensibilité des systèmes d'exploitation est de plus en plus considérée comme une caractéristique essentielle. Or, une MPR générique nécessite d'avoir accès à un environnement d'exécution privilégié. L'évolution s'oriente donc dans la bonne direction.

* * *

Le comportement d'un système est très dépendant du style d'utilisation par les applications, et cela à tous les grains d'accès (zone, segments, partitions). Il existe deux moyens (non mutuellement exclusifs) de faire face à cette dépendance : soit en trouvant un compromis qui donne des résultats acceptables dans tous les cas, soit en créant un système qui soit adaptable. Arias utilise les deux moyens. Par exemple, la structure des données de localisation de segments part d'un compromis entre simplicité, fonctionnalité et généricité. En même temps, plusieurs paramètres permettent, si nécessaire, d'adapter le système aux applications à servir.

Pour bien comprendre ces variations de comportement, il faut pouvoir utiliser le service dans un véritable environnement d'utilisation, avec des applications en grandeur réelle. Bien que l'on commence à bien comprendre et utiliser la MPR pour les applications scientifiques, il existe encore très peu d'applications réparties. C'est une sorte de cercle vicieux : en l'absence d'applications phares, les MPR ne parviennent pas à démontrer leurs avantages, ce qui à son tour n'encourage pas le portage d'applications vers l'utilisation de MPR.

* * *

La notion de segment a facilité l'organisation et la modularisation d'Arias. Cependant, la mémoire est naturellement constituée d'octets (ou de mots), et l'introduction d'un

[21] Nous avons été impressionné par la quantité de fonctions qui sont intégrées dans AIX. Le seul point que nous avons regretté à son sujet est l'étonnante lacune de documentation fiable.

regroupement n'a pas toujours été sans quelques contorsions. En particulier, la gestion des zones, des pages et des segments gagne à être bien intégrée. Malheureusement, ce n'est pas sans raison que les systèmes de mémoire (à commencer par la mémoire virtuelle d'AIX) s'appuient sur des notions similaires à nos segments. De tels regroupements simplifient beaucoup la gestion de grandes quantités de mémoire. Les relations entre

- les pages ≪physiques≫ de mémoire,
- la correspondance entre adresse virtuelle et mémoire physique (tables de pages),
- la détection et la protection d'accès,
- le stockage secondaire (*swap*) et le chargement sur demande,
- le cache mémoire, etc.,

sont intimes et complexes. La solution du regroupement en segments simplifie le problème mais impose des règles de programmation supplémentaires.

V-3 Directions futures

Les perspectives que l'on peut envisager pour Arias et les MPR en général sont les réponses aux remarques émises ci-dessus. Plus précisément, les idées suivantes sont à étudier.

- Comment diminuer ou s'affranchir de la lenteur de l'environnement STREAMS ; un changement d'architecture, toujours dans le cadre de STREAMS mais en intégrant plus intimement les différents modules, semble souhaitable.
- Il est important d'encourager la production d'applications utilisant la MPR. Or, le meilleur moyen d'y arriver semble non pas d'écrire les applications nous-mêmes, mais de fournir des environnements de programmation et des *middle-ware* qui soient en mesure d'attirer les programmeurs.
- Une ≪mémoire de zones≫ a-t-elle un sens ? Comment s'affranchir de la notion de segment et rendre totalement transparente l'existence des pages physiques ?

Cette notion de mémoire de zones est particulièrement attrayante. Dans une MPR organisée de cette manière, les applications pourraient ne manipuler que des zones, en allouant directement des zones et non des segments. Les zones seraient alors l'entité de gestion unique. D'unité d'accès et de synchronisation, la zone deviendrait aussi l'unité d'allocation, de protection, de stockage, etc. Cette organisation a été rejetée très tôt par peur des surcoûts de gestion. Nous avons craint que les méta-données dépassent en taille les données elles-mêmes. Il peut être intéressant de tenter néanmoins l'expérience et d'évaluer ce surcoût de manière précise, ainsi que l'utilité réelle d'un tel système.

Bibliographie

[1] AHO ALFRED, HOPCROFT J., ULLMAN J. *Structures de données et algorithmes.* InterEditions 1987 (édition originale Addison-Wesley 1983).

[2] AMZA C., COX A., DWARKADAS S., KELEHER P., LU H., RAJAMONY R., YU W., ZWAENEPOEL W. Treadmarks : Shared Memory Computing on Networks of Workstations. *IEEE Computer* 29(2) :18–28, Février 1996.

[3] ANDERSON T., CULLER D., PATTERSON D., ET AL. A Case for NOW (Networks of Workstations). In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*. Août 1994.

[4] AT&T UNIX SOFTWARE OPERATION. *Unix System V Release 4 Programmer's Guide : STREAMS.* Prentice-Hall, 1990.

[5] BAKER M., HARTMAN J., KUPFER M., SHIRRIFF K., OUSTERHOUT J. Measurement of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 198–212, Octobre 1991.

[6] BERSHAD B., ANDERSON T., LAZOWSKA E., LEVY H. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1) :37–55. Février 1990.

[7] BERSHAD B., ZEKAUSKAS M., SAWDON W. The Midway Distributed Shared Memory System. *IEEE* Octobre 1993.

[8] A. D. BIRRELL A., NELSON B. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1) :39–59. Février 1984.

[9] CARTER J., BENNETT J., ZWAENEPOEL W. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 152–164, Octobre 1991.

[10] CARTER J., KHANDEKAR D., KAMB L. Distributed Shared Memory : Where We Are and Where We Should Be Headed. In *Proceedings of the Fifth International Workshop on Hot Topics in Operating Systems (HotOS-V)*, Mai 1995.

[11] CHASE J., AMADOR A., LAZOWSKA E. The Amber System : Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 147–158, Décembre 1989.

[12] CHASE J., FEELEY M., LEVY H. Some Issues for Single Address Space Systems. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*. Octobre 1993.

[13] CHASE J., ISSARNY V., LEVY, H. Distribution in a Single Address Space Operating System. In *Proceedings of the Fifth ACM SIGOPS European Workshop*, Septembre 1992. (Aussi dans *Operating Systems Review*, Avril 1993).

[14] CHASE J., LEVY H., BAKER-HARVEY M., LAZOWSKA E. How to use a 64-bit Virtual Address Space. University of Washington, Department of Computer Science and Engineering, Technical Report 92-03-02. Mars 1992.

[15] CHASE J., LEVY H., LAZOWSKA E., BAKER-HARVEY M. Opal : A Single Address Space System for 64-bit Architectures. In *Proceedings of the IEEE Workshop on Workstation Operating Systems*, Avril 1992.

[16] CHASE J., LEVY H., LAZOWSKA E., BAKER-HARVEY M. Lightweight Shared Objects in a 64-bit Operating System. In Archibald J. and Wilkes M. (Ed.) *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, pp. 397–413. Octobre 1992. Vancouver, BC, Canada.

[17] CHERITON D., ZWAENEPOEL W. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the Ninth Symposium on Operating System Principles*, pp.129–140, Octobre 1983.

[18] CHLAMTAC I., GANZ A., KIENZLE M. G. An HIPPI interconnection system. *IEEE Transactions on Computers*, C-42(2) :138–150. Février 1993.

[19] CRAY RESEARCH. *The CRAY T3E Scalable Parallel Processing System.* `http://www.cray.com/PUBLIC/product-info/T3E/CRAY_T3E.html`

[20] DALEY R., DENNIS J. Virtual Memory, Processes, and Sharing in MULTICS. *Communication of the ACM*, Mai 1968, pp. 306–312.

[21] DASGUPTA P., ET AL. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems Journal*, 3, Winter 1990.

[22] DAVID E., FANO R. Some Thoughts About the Social Implications of Accessible Computing. In *Proceedings of the 1965 Fall Joint Computer Conference*, pp. 243–247.

[23] DECHAMBOUX P., HAGIMONT D., MOSSIÈRE J., ROUSSET DE PINA X. Objectifs et plan de travail du projet Sirac. Rapport Technique RT-Sirac-1-95, LSR-IMAG. Disponible électroniquement sur ftp://ftp.imag.fr/pub/SIRAC/doc/reports/95-1-sirac-plan-RT.ps.gz, Juin 1995.

[24] DIGITAL EQUIPMENT CORPORATION. *Alta Vista*. http://www.altavista.digital.com/

[25] DIJKSTRA E. Co-operating sequential processes. In *Programming Languages*, Genuys F., Ed., Academic Press, 1968, pp. 43–112.

[26] DIJKSTRA E. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, 11(5) :341–346. Mai 1968.

[27] ELPHINSTONE K. Address Space Management Issues in the Mungi Operating System. SCS&E Report 9312, School of Computer Science and Engineering, University of New South Wales, Novembre 1993.

[28] FEELEY M., MORGAN W., PIGHIN F., KARLIN A., LEVY H., TEKKATH C. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, pp. 201–212, Décembre 1995.

[29] FERREIRA P., SHAPIRO M. Distribution and Persistence in Multiple and Heterogeneous Address Spaces. In Cabrera L. and Hutchinson N. (Ed.) *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pp. 83-89. IEEE, Décembre 1993. Asheville, North Carolina, USA.

[30] FERRIÉ J., LANCIAUX, D. *Le système Plessey 250.* Rapport de recherche IRIA R 168, Avril 1976.

[31] FREYSSINET A., KRAKOWIAK S., LACOURTE S. A Generic Object-Oriented Virtual Machine. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS '91)*.

[32] GEIST G., SUNDERAM V. The PVM system : Supercomputer level concurrent computaion on a heterogeneous network of workstation. In *Sixth Annual Distributed-Memory Computer Conference*, pp. 258–261, 1991.

[33] HAGIMONT D., CHEVALIER P.-Y., FREYSSINET A., KRAKOWIAK S., LACOURTE S., MOSSIÈRE J., ROUSSET DE PINA X. Persistent Shared Object Support in the Guide System : Evaluation & Related Work. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 129–144. Octobre 1994.

[34] HAN J., KNAFF A., PÉREZ-CORTÉS E., SAUNIER F. Arias : A Generic Support for Persistent Runtimes. Présenté lors de l'*European Research Seminar on Advances in Distributed Systems*, Avril 1995. Disponible sur http://

www-bi.uinrialpes.fr/SIRAC/pub/ ou ftp://ftp.imag.fr/pub/SIRAC/publications/95-ersads.ps.gz

[35] HEINRICH J. *MIPS R4000 User's Manual.* Prentice-Hall, 1993.

[36] HEISER G., ELPHINSTONE K., RUSSELL S., VOCHTELOO J. Mungi : A Distributed Single Address-Space Operating System In *Proceedings of the 17th Australasian Computer Science Conference*, pp. 271–280. Janvier 1994. Une version plus longue est le Technical Report 9314, University of New South Wales, School of Computer Science and Engineering.

[37] HOARE, C. A. R. Communicating Sequential Processes. *Communications of the ACM* 21(8) :666–677. Août 1978.

[38] INTERNATIONAL BUSINESS MACHINES. *IBM System/38 Introduction.* Septembre 1980.

[39] JONES A., CHANSLER R., DURHAM I., SCHWANS K., VEGDAHL S. STAROS, A Multiprocessor Operating System for the Support of Task Forces. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pp. 117–127. Décembre 1978.

[40] JUL E., LEVY H., HUTCHINSON N., BLACK A. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1) :109–133, Février 1988.

[41] KELEHER P., COX A., DWARKADAS S., ZWAENEPOEL W. TreadMarks : Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pp. 115–131, Janvier 1994.

[42] KNAFF A. *Conception et réalisation d'un service de stockage fiable et extensible pour un système réparti à objets persistants.* Thèse de doctorat. Université Joseph Fourier – Grenoble I, Octobre 1996.

[43] KNUTH D. *The Art of Computer Programming, Vol. III : Sorting and Searching.* Addison-Wesley, 1973.

[44] KNUTH D. *The TeXbook. Computers & Typesetting, vol. I.* 3e édition. McGraw-Hill, 1984.

[45] KOLDINGER E., CHASE J., EGGERS S. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 175–186. ACM, Octobre 1992.

[46] LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9) : 690-691, Septembre 1979.

[47] LEE R. Precision Architecture. *IEEE Computer*, Janvier 1989.

[48] LENOSKI D., LAUDON J., GHARACHORLOO K., WEBER W.-D., GUPTA A., HENNESSY J., HOROWITZ M., LAM M. S. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3) :63–79, Mars 1992.

[49] LEVY H. *Capability-Based Computer Systems.* Digital Press, 1984.

[50] LI K. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* Thèse de doctorat, Department of Computer Science, Yale University, Septembre 1986.

[51] LI K. IVY : A Shared Virtual Memory System for Parallel Computing. *Proceedings of the 1988 International Conference on Parallel Processing*, vol. 2, pp. 94–101, Août 1988.

[52] LI K., HUDAK P. Memory Coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4) :321–359, Novembre 1989.

[53] LIEDTKE J. Address space sparsity and fine granularity. In *6th SIGOPS European Workshop*, pp. 78-81. Aussi in *Operating Systems Review* 29(1) : 87–90.

[54] LIEDTKE J., ELPHINSTONE K. Guarded Page Tables on Mips R4600 or An Exercise in Architecture-Dependent Micro Optimization. *Operating Systems Review* 14(1).

[55] LOUCKS L., SAUER C. Advanced Interactive Executive (AIX) Operating System Overview. *IBM Systems Journal* 26(4) :326–345, Décembre 1987.

[56] MAY C., SILHA E., SIMPSON R., WARREN H. (Ed.) *The PowerPC Architecture : A Specification for a New Family of RISC Microprocessors.* International Business Machines, Morgan Kaufmann Publishers, Mai 1994 (seconde édition).

[57] MINNICH R., FARBER D. Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory.

[58] MINZER S. E. Broadband ISDN and Asynchronous Transfer Mode (ATM). *IEEE Communications*, Septembre 1989, pp. 17–24, 57.

[59] MOSSIÈRE J., ROUSSET DE PINA X. Single Address Space or Private Address Spaces? *Sixth ACM SIGOPS European Workshop*, pp. 72-77, Septembre 1994.

[60] MURRAY K., WILKINSON T., OSMON P., SAULSBURY A., STIEMERLING T., KELLY P. Design and Implementation of an Object-Oriented 64-bit Single Address Space Microkernel. In *Proceedings of the Second USENIX Symposium on Microkernels and Other Kernel Architectures*, pp. 31–43, 1993.

[61] GUEDES P., CASTRO M. Distributed Shared Object Memory. In *Proceedings of the Fourth Workshop on Workstation Operating Systems.* Octobre 1993.

[62] ORGANISATION INTERNATIONALE DE NORMALISATION (ISO). *Fibre Distributed Data Interface.* ISO 9314-1&2. 1989.

[63] OUSTERHOUT J., SCELZA D., SINDHU P. Medusa : An Experiment in Distributed Operating System Structure. *Communications of the ACM* 23(2). Février 1980.

[64] ÖZDEN B., SILBERSCHATZ A. A Taxonomy of Shared Address Space Systems. Rapport technique TR-92-33, Department of Computer Sciences, University of Texas at Austin, Juillet 1992.

[65] PÉREZ-CORTÉS E. *La cohérence sur mesure dans une mémoire partagée répartie.* Thèse de doctorat, Institut National Polytechnique de Grenoble. Novembre 1996.

[66] PÉREZ-CORTÉS E., DECHAMBOUX P., HAN J. Generic Support for Synchronization and Consistency in Arias. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Mai 1995.

[67] PÉREZ-CORTÉS E., HAN J., MOSSIÈRE J. Construction de protocoles de cohérence sur une interface générique de mémoire partagée répartie. Présenté au séminaire *MPR'96* du Groupe de Recherche en Informatique Parallèle et Distribuée, Mai 1996.

[68] REDELL D., DALAL Y., HORSLEY T., LAUER H., LYNCH W., MCJONES P., MURRAY H., PURCELL S. Pilot : an Operation System for a Personal Computer. *Communications of the ACM*, Février 1980, pp. 81–92.

[69] ROCA V. *Architecture Hautes Performances pour Systèmes de Communication.* Thèse de doctorat. Institut National Polytechnique de Grenoble, Janvier 1996.

[70] ROZIER M. Chorus (Overview of the Chorus Distributed Operating System). In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 39–70. Avril 1992.

[71] RUSSELL S., SKEA A., ELPHINSTONE K., HEISER G., BURSTON K., GORTON I., HELLESTRAND G. Distribution + Persistence = Global Virtual Memory. In *International Workshop on Object-Orientation in Operating Systems*, vol. 2, pp. 96–99, 1992.

[72] SALTZER J. Protection and the Control of Information Sharing in Multics. *Communication of the ACM* 17(7) : 388–402. Juillet 1974.

[73] SAUNIER F. *Protection d'une mémoire virtuelle répartie par capacités implicites.* Thèse de doctorat, Institut National Polytechnique de Grenoble. Octobre 1996.

[74] SEDGEWICK R. *Algorithmes en langage C.* InterEditions, 1991 (édition originale Addison-Wesley, 1990).

[75] SINGH J., JOE T., HENNESSY J., GUPTA A. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. In *Proceedings of the Supercomputing '93 Conference*, pp. 214–229, Novembre 1993.

[76] SITES R. (Ed.) *Alpha Architecture Reference Manual.* Digital Press, 1992.

[77] TANNENBAUM A., VAN RENESSE R., VAN STAVEREN H., SHARP G., MULLENDER S., JANSEN A., VAN ROSSUM G. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM* vol 33, pp. 46–63, Décembre 1990.

[78] TREMBLAY M., O'CONNOR J. UltraSPARC-1 : A Four-Issue Processor Supporting Multimedia. *IEEE Micro* 16(2), Avril 1996.

[79] VOCHTELOO J., RUSSELL S., HEISER G. Capability-Based Protection in a Persistent Global Virtual Memory System. University of New South Wales, School of Computer Science and Engineering, Technical Report 9303. Mars 1993.

[80] WHITE, S. J. *Pointer Swizzling Techniques for Object-Oriented Database Systems.* Thèse de doctorat, Université du Winsconsin à Madison. Septembre 1994.

[81] WILSON P. Pointer Swizzling at Page Fault Time. *ACM SIGARCH Computer Architecture News* 19(4), Juin 1991.

[82] WIRTH, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

[83] WOLMAN, VOELKER, THEKKATH. Latency Analysis of TCP on an ATM Network. Rapport technique TR-03-03. Université de Washington, Seattle, Washington, USA. Mars 1993.

[84] WULF W., COHEN E., CORWIN W., JONES A., LEVIN R., PIERSON C., POLLACK F. HYDRA : The Kernel of a Multiprocessor Operating System. *Communications of the ACM* 17(6) : 337–345. Juin 1974.

[85] WULF W., LEVIN R., HARBISON S. *HYDRA/C.mmp : An Experimental Computer System.* McGraw-Hill, 1981.

Table des matières

Introduction **3**

I La mémoire partagée répartie **11**

I-1 Du client/serveur à la MPR 11

I-1.1 Le modèle client/serveur 11

I-1.2 L'appel procédural à distance 12

I-1.3 Le modèle de MPR 12

I-1.4 Les modèles à objets 14

I-2 L'évolution des MPR 14

I-2.1 L'ancêtre : MULTICS 15

I-2.2 La MPR de pages 15

I-2.3 Les modèles de cohérence 17

I-2.4 Les systèmes à objets 17

I-2.5 Les SASOS et la protection inter-applications 18

I-2.6 Synthèse 20

I-3 Pourquoi une nouvelle MPR ? 20

I-3.1 L'adressage uniforme 20

I-3.2 Les réseaux rapides 21

I-3.3 L'économie d'échelle 22

I-3.4 La maturation de la MPR 22

I-4 Objectifs d'Arias 22

I-4.1 Objectifs 23

I-4.2 Définitions 23

I-4.3 Sémantique des manipulations 25

I-4.4 Protection inter-application 26

I-4.5 Résistance aux pannes 27

I-4.6 Synchronisation et cohérence 27

I-4.7 Répartition et migration 28
I-5 La problématique de la gestion de segments 29
I-5.1 Interface du module de gestion de segments 29
I-5.2 La localisation dans un grand espace d'adressage 30
I-5.3 L'allocation et la localisation 30
I-6 Architecture 31
I-6.1 La pile STREAMS 32
I-6.2 PGR : le paginateur 33
I-6.3 SEG : l'interface applicative 33
I-6.4 Les moniteurs 33
I-7 Transition 33

II Conception de la gestion de segments 35
II-1 Interfaces 35
II-1.1 Interface applicative 35
II-1.2 Interface CGC — LOC 37
II-2 Identification des segments : la fiche 41
II-3 La localisation dans un grand espace 41
II-3.1 Tables de pages hiérarchiques 42
II-3.2 Tables de pages gardées 44
II-4 Conception du système en trois niveaux 46
II-4.1 Tables de hachage hiérarchiques 46
II-4.2 Blocs 47
II-4.3 Partitionnement 51
II-4.4 Résumé 53
II-4.5 Accélérateurs de localisation 53
II-5 L'allocation 54
II-5.1 Choix d'une partition 55
II-5.2 Choix d'un site 56
II-5.3 Partition indifférente 57
II-6 Vue d'ensemble 57
II-7 Interface d'administration 58
II-8 Couplage et protection 59
II-8.1 Des capacités 59
II-8.2 Des domaines 60
II-8.3 Couplage implicite 60
II-8.4 Capacités et ramasse-miettes 60

III Réalisation **63**
III-1 Localisation 63
III-1.1 Gestion des accélérateurs 63
III-1.2 Tables de hachage hiérarchiques 65
III-1.3 Blocs, buddy-system 70
III-1.4 Table de partition 72
III-2 Allocation 73
III-2.1 Algorithme d'allocation dans un bloc 73
III-2.2 Interaction avec la CGC 75
III-2.3 Protocole de réservation de bloc 75
III-3 Couplage 76
III-3.1 Principes du couplage 76
III-3.2 L'organisation de la mémoire virtuelle AIX 77
III-3.3 Les défauts de segment 79
III-3.4 Les défauts de page 80
III-3.5 Le transfert de données 80
III-3.6 Interface de couplage 81
III-3.7 Segment Arias et segment AIX 82
III-4 Synthèse 82

IV Paramétrage, mesure et évaluation **83**
IV-1 AIX 4.1 et STREAMS 83
IV-1.1 Pourquoi STREAMS ? 83
IV-1.2 Chiffres clés 84
IV-2 Paramétrage 86
IV-2.1 Partitions 86
IV-2.2 Blocs 86
IV-2.3 Tables de hachage 87
IV-2.4 Paramétrage de base 88
IV-3 Évaluation temporelle 88
IV-3.1 Micro-test : création de segment 88
IV-3.2 Micro-test : localisation 89
IV-3.3 Analyse 89
IV-4 Évaluation spatiale : système de fichiers 90
IV-4.1 Jeux d'essai 91

IV-4.2 Mode 32 bits 92
IV-4.3 Mode 64 bits 95
IV-4.4 Synthèse 97
IV-5 Évaluation spatiale : insertion aléatoire 98

V Conclusions et perspectives 101
V-1 Récapitulatif 101
V-2 Quelques leçons apprises 102
V-3 Directions futures 103

Références 105

Liste des figures

I-1 Exemples d'architectures multiprocesseurs 13
I-2 Architecture d'un réseau local de stations 13
I-3 Le partage d'une région de l'espace d'adressage entre deux applications 24
I-4 Composition d'un objet de grande taille 28
I-5 Architecture générale 32

II-1 Relation entre une zone, son site maître, le segment et son site primaire 38
II-2 Migration de segment 40
II-3 Table de pages référençant un segment 43
II-4 Structure générale d'une table de pages hiérarchiques 44
II-5 Exemple de décodage avec des tables de pages linéaires 45
II-6 Décodage avec des tables de pages gardées 45
II-7 Tables de hachage hiérarchiques 47
II-8 Buddy-system binaire 48
II-9 Gâchis d'espace d'adressage dans un buddy-system binaire 48
II-10 Buddy-system de base 16 49
II-11 Une table de 16 entrées dont 11 occupées et 5 libres. 50
II-12 Destruction d'un élément dans un buddy-system binaire 50
II-13 Compactage d'un buddy-system de base 16 51
II-14 Exemple de partitionnement 52
II-15 Exemple de partition 52
II-16 Vue d'ensemble des composants de la gestion de segments 57

III-1 Fonction de hachage par composition binaire, exemple sur 12 bits . . . 66
III-2 Exemple de chaîne de recherche de longueur 3 67
III-3 Cas extrême de hiérarchie de tables de hachage à deux niveaux 69
III-4 Cas extrême de hiérarchie de tables de hachage à trois niveaux 69
III-5 Buddy-systems binaire et non binaire : exemple 70

III-6 Effet de la consommation minimale d'allocation de segment 71
III-7 Exemple de buddy-system de base 16 couvrant 16 bits d'adresse 71
III-8 Allocation dans un bloc en utilisant un seul pointeur 73
III-9 Allocation d'un segment dans un bloc encombré 74
III-10La mémoire virtuelle AIX 4.1 (32 bits) 78
III-11La mémoire virtuelle sur PowerPC 620 (64 bits) 78

IV-1 Pile STREAMS minimale 85
IV-2 Tailles de fichier dans un jeu d'essai 92
IV-3 Segment à 4 entrées dans un buddy-system de largeur 4 93
IV-4 Distribution des segments dans le jeu d'essai **bin** 94
IV-5 Tailles de fichiers dans une grande partition 95

Liste des tableaux

III-a Quelques valeurs typiques de collisions dans une table de hachage . . . 67
III-b Exemple de mouvement des pointeurs d'allocation 74

IV-a Chiffres clés des performances des STREAMS sous AIX 4.1 84
IV-b Création de segments, paramétrage de base 89
IV-c Création de segment, changement de paramètre 89
IV-d Localisation de segment 90
IV-e Exemples de systèmes de fichiers 91
IV-f Jeux d'essai ajustés pour le mode 32 bits 92
IV-g Utilisation spatiale de l'allocation 93
IV-h Utilisation spatiale (bis) 94
IV-i Utilisation spatiale (64 bits, **home**) 96
IV-j Utilisation spatiale (64 bits, **bin**) 96
IV-k Utilisation spatiale (64 bits, **serveur**) 97
IV-l Deux cas extrêmes d'organisation des blocs 98
IV-m Allocation aléatoire 98

Résumé

Arias est un système de mémoire partagée répartie (MPR) réalisé dans le cadre du projet SIRAC, qui étudie les supports logiciels pour les applications distribuées.

Une MPR facilite grandement la programmation de telles applications. Nous retraçons l'évolution des systèmes distributés qui ont abouti à cette idée, et caractérisons les diverses particularités de quelques projets antérieurs. Nous édictons nos objectifs propres et dégageons les spécifications qui sont souhaitables pour notre projet.

Partant de là, et après avoir esquissé l'architecture générale d'Arias, nous nous concentrons sur les problèmes liés à la répartition de la mémoire. En particulier, l'allocation et la localisation de la mémoire dans une MPR à grand espace d'adresses posent des difficultés de réalisation qui ont des conséquences importantes dans la conception de l'interface logicielle avec l'application et le système d'exploitation sous-jacent. L'analyse de ces difficultés nous mène au module d'allocation et de localisation que nous décrivons.

Une fois son implémentation décrite dans ses détails techniques, nous en explicitons les paramètres d'ajustement et soulignons leur impact à travers des mesures et des simulations. Nous montrons plusieurs configurations envisageables, dont certaines sont spécialement adaptées à certains types d'utilisation.

Pour finir, nous concluons sur les leçons apprises de ce travail, tant du point de vue de ce qui a été réalisé que sur les travaux futurs et les perspectives générales des MPR.

Summary

Arias is a distributed shared memory (DSM) system designed as part of project SIRAC, whose purpose is to study a software support for distributed applications.

Programming distributed applications is greatly facilitated by a DSM support. The evolution of distributed systems leading to the idea of DSM is outlined. A number of past and current DSM projects are studied and their particular features highlighted. The goals of our own project are then put into perspective, and its spefication defined.

From there, the general architecture of Arias is described. We focus our attention on the problems related to the distribution of memory. Specifically, allocating and locating memory in a large-address space DSM presents some major difficulties that impact the application interface and the utilization of the underlying operating system. The close examination of those problems leads to the design of the memory service that is described.

The implementation of that service is then fleshed out in its technical details. This implementation has a number of adjustable parameters whose impact on the general behaviour is refined through measurements and simulations. Some potential configurations, each adapted to certain usage patterns, are shown.

In conclusion, we revisit lessons learned from this work. From what has been completed, we derive a view of future study topics and general perspectives on DSM.