Rapport de stage
de
D.E.A de Systèmes Informatiques

Septembre 1996

MAÎS

Philippe CADINOT,
Neilze DORTA.

Sous la responsabilité de messieurs,

Bertil FOLLIOT et Pierre SENS

Remerciements

Je tiens à remercier ....

Table des matières

Table des figures

Chapitre 1
Introduction

Les outils tels que les applications parallèles, les calculs scientifiques, le multimédia, les simulations d'ingénierie et beaucoup d'autres, ont un besoin croissant de mémoire. Ce besoin dépasse souvent le total de mémoire existant sur une station de travail. Le problème est d'autant plus important si on tient compte de la multiprogrammation et de la technique du temps partagé. L'utilisation du mécanisme de mémoire virtuel, fourni par le système d'exploitation, a pour objectif de diminuer le manque de mémoire. Pour cela, une application a une partie de ses données dans la mémoire et le restant sur le disque; ce qui lui permet de s'exécuter sur un espace total égal à "espace mémoire + espace disque".

Le mécanisme de mémoire virtuel utilise les téchniques de pagination et de *swap* entre la mémoire et le disque. Traditionnellement, les systèmes d'exploitation utilisent les disques magnétiques comme le périphérique de pagination. Malheureusement, l'écart de vitesse entre les processeurs et les disques croît verticalement, rendant le coût à payer pour la pagination inacceptable.

L'émergence des réseaux à haut débit (Ethernet 100 Base T, ATM, FDDI) modifie la hiérarchie des temps d'accès par rapport aux réseaux classiques (Ethernet 10 Base T, Token Ring):

Mémoire locale, disque, réseaux $\Rightarrow$ Mémoire locale, réseau, disque.

Ces réseaux offrent une grande bande passante et une faible latence. Dès lors, rechercher une information dans une mémoire distante devient moins coûteux qu'une lecture sur un disque local [Thekkath, 1994]. Cette caractéristique offre de nouvelles possibilités de conception de systèmes d'exploitation répartis tirant profit de la vitesse de transfert de mémoire centrale à mémoire centrale, telles que les caches disques répartis ou la mémoire virtuelle répartie.

Dans un réseau de stations de travail, la capacité totale de traitement est sous utilisée. Des mesures montrent: un taux d'occupation moyen du processeur de 20% pendant la période critique [Stumm, 1988], un tiers des stations de travail restent oisives même durant les moments de la journée les plus actifs [Theimer, Lantz, 1989], au-delà d'un giga octet de mémoire inutilisée dans une soirée banale [Felten, Zahorjan, 1991]. Il est donc possible de trouver, parmi les stations de travail, suffisamment de mémoire disponible pour servir comme périphérique de pagination.

Utiliser la mémoire physique des machines distantes pour la pagination se montre très avantageux dans un environnement réseau à haut débit. Cet avantage est d'autant plus important dans le cadre d'applications parallèles qui ont grand besoin de mémoire. Dans ce sens, l'idée de connecter des machines homogènes par un bus à très haut débit (600 Mo/s) pour permettre l'exécution des ces applications, exige du système d'exploitation une nouvelle façon de mettre en oeuvre la mémoire virtuelle.

La pagination distante sur un réseau des stations de travail est donc la réponse à un besoin technique et économique. Comment augmenter l'espace mémoire total sans pour autant changer la configuration déjà existante? Comment profiter de la mémoire oisive des autres machines du réseau? Comment envisager l'utilisation d'un grain fin pour les applications parallèles? C'est sûrement avec cette préoccupation que l'idée de créer une pagination à distance en coopération avec des nouvelles technologies hardware est si prometteuse.

Dans ce rapport, nous avons voulu donner une vision globale du problème, en parlant des techniques de *software* utilisées ainsi que des éléments de *hardware* nécessaires pour obtenir le résultat désiré. Pour comprendre pourquoi l'étude sur la pagination à distance est importante, nous donnons un aperçu des applications potentielles à son usage. Le deuxième chapitre est dédiée à expliquer le mécanisme de la mémoire virtuelle et aussi la technique choisie pour l'implanter dans le système d'exploitation utilisé dans ce projet. Le troisième chapitre montre ce qui existe dans l'univers réseau à haut débit et les implications de ce qu'il offre sur les systèmes d'exploitation. Le chapitre 4 donne un état de l'art sur le *swap* et la pagination. Enfin, dans le chapitre 5 nous présentons MAÎS, notre projet de pagination à distance, avec sa conception, son implémentation et ses performances. Le dernier chapitre parle de nos conclusions et perspectives dans le cadre de ce travail.

Chapitre 2
Gestion Mémoire

Le système de gestion de la mémoire est un composant central pour tout système d'exploitation. Le gestionnaire de mémoire est responsable pour la gestion des ressources disponibles dans la machine. Ces ressources sont typiquement classifiées de façon hiérarchique, ayant le temps d'accès à la mémoire inversement proportionnel à leur proximité de la CPU (voir Figure 1). La mémoire physique constitue le premier niveau de mémoire; le suivant est la mémoire secondaire qui correspond en général aux disques magnétiques. Dans certains environnements de stations de travail, ces deux niveaux hiérarchiques deviennent trois niveaux avec l'ajout des machines serveurs de fichier connectés à la station de travail via un réseau local.

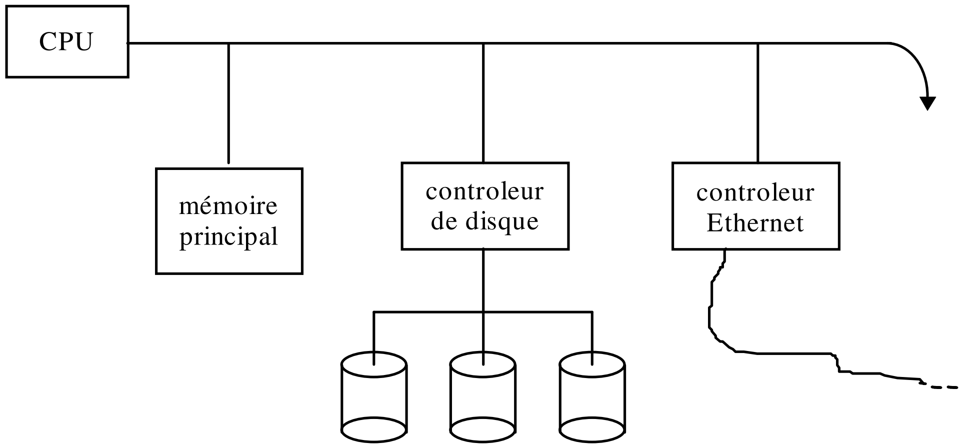

Figure 1. Classification hiérarchique de la mémoire.

Dans un environnement de multiprogrammation, il est critique pour le système d'exploitation de partager de manière efficace les ressources de mémoire disponibles entre les processus. La fonction d'arbitrage de tout gestionnaire de mémoire est directement liée à la quantité de mémoire requise par le processus pour s'exécuter. Cela veut dire que si un processus a besoin de résider entièrement en mémoire pour s'exécuter, alors le gestionnaire de mémoire doit, en principe, allouer des grandes unités de mémoire. Par ailleurs, si un processus peut s'exécuter étant partiellement résident en mémoire, alors l'arbitrage du gestionnaire de mémoire doit être totalement différent. En général, les gestionnaires de mémoire essayent d'optimiser le nombre de processus en cours d'exécution résidents en mémoire principal. Cet objectif tient compte des objectifs du processus ordonnanceur, de façon à éviter des conflits qui puissent affecter la performance du système.

La disponibilité d'une mémoire secondaire permet l'existence d'un nombre plus grand de processus que celui possible de résider dans la mémoire principal. Gérer cet espace requiert, en général, des algorithmes plus compliqués et des arbitrages différents de ceux utilisés pour la mémoire principal. Cet arbitrage doit décider quand déplacer un processus entre la mémoire principal et la mémoire secondaire.

2.1 ETAT DE L'ART.

2.1.1 LES MECANISMES DE GESTION MEMOIRE

Les algorithmes de gestion mémoire pour les systèmes d'exploitation multi-programmés vont de la simple approche du système mono-utilisateur jusqu'à la segmentation paginée. Le déterminant le plus important de la méthode employée dans un système particulier est le matériel fourni. On doit vérifier la légalité de chaque adresse mémoire générée par l'UC et éventuellement la transformer en une adresse physique. La vérification ne peut pas être implémentée (efficacement) avec un logiciel. Désormais, nous sommes limités par le matériel disponible.

Les algorithmes de gestion mémoire (allocation contiguë, pagination, segmentation et combinaison de pagination et segmentation) diffèrent par plusieurs aspects. La liste ci-dessous indique quelques considérations importantes à utiliser pour comparer les différentes stratégies de gestion mémoire:

- **Support matériel:** un simple registre de base ou une paire de registres de base et limite sont suffisantes pour les schémas de partitions uniques et multiples, tandis que la pagination et la segmentation ont besoin de tables de correspondance pour définir la conversion d'adresses.

- **Performance:** quand l'algorithme devient plus complexe, le temps requis pour convertir une adresse logique en une adresse physique augmente. Pour les systèmes simples, nous devons seulement comparer ou additionner l'adresse logique; opérations qui sont rapides. La pagination et la segmentation peuvent être rapides si la table est implémentée avec des registres rapides. Cependant, si la table est en mémoire, les accès mémoire de l'utilisateur peuvent être fortement dégradés. Un ensemble de registres associatifs peut réduire la dégradation de la performance à un niveau acceptable.

- **Fragmentation:** un système multiprogrammé fonctionne généralement plus efficacement avec un haut niveau de multiprogrammation. Pour un ensemble donné de processus, nous pouvons augmenter le niveau de multiprogrammation seulement en tassant plus de processus en mémoire. Pour accomplir cette tâche, nous devons réduire le gaspillage de mémoire et la fragmentation. Les systèmes ayant des unités d'allocation de taille fixe, comme le schéma de partition unique et la pagination, souffrent de la fragmentation interne. Les systèmes ayant des unités d'allocation de taille variable, comme le schéma de plusieurs partitions et la segmentation, souffrent de la fragmentation externe.

- **Translation:** le compactage est une solution au problème de la fragmentation externe. Le compactage implique un déplacement des programmes en mémoire. Ce déplacement doit être transparent. Cela suppose que les adresses logiques soient translatées dynamiquement au moment de l'exécution. La compression est donc impossible si les adresses ne peuvent être translatées qu'au moment du chargement.

- ***Swapping*:** on peut ajouter le *swapping* à tous les algorithmes. A des intervalles de temps déterminés par le système d'exploitation et habituellement imposés par les politiques de *scheduling* de l'UC, on copie les processus de la mémoire principale à la mémoire auxiliaire. Plus tard ils sont replacés en mémoire principale. Ce schéma permet d'exécuter plus de processus que la mémoire ne le permettait à un moment donné.

- **Partage:** un autre moyen d'augmenter le niveau de multiprogrammation est de partager le code et les données entre plusieurs utilisateurs. Le partage demande que l'on utilise la pagination ou la segmentation afin de fournir des petits paquets d'information (pages ou segments) à être partagés. Le partage est un moyen d'exécuter plusieurs processus avec une quantité limitée de mémoire, mais on doit concevoir soigneusement les programmes et les données partagés.

- **Protection:** les mécanismes de pagination et de segmentations fournissent, généralement, des droits d'accès aux utilisateurs. L'utilisation de ces droits permet d'interdire ou d'autoriser l'exécution et/ou la lecture et/ou la modification de certaines régions par certains utilisateurs. Ces restrictions, nécessaires pour gérer le partage du code et des données, sont généralement utiles pour fournir des moyens de détection des erreurs logicielles en cours d'exécution.

2.1.2 MEMOIRE VIRTUELLE

Il est souhaitable de pouvoir exécuter un processus dont l'espace adresse logique soit plus grand que l'espace adresse physique disponible. Le programmeur peut rendre ce processus exécutable en le restructurant, en utilisant des recouvrements, mais c'est généralement une tâche de programmation difficile. La mémoire virtuelle est une technique permettant d'établir une correspondance entre un grand espace d'adressage logique et une mémoire physique plus restreinte. La mémoire virtuelle permet d'exécuter des processus extrêmement grands, mais aussi d'augmenter le degré de multiprogrammation, ce qui accroît le taux d'utilisation de l'UC. De plus, cela évite aux programmeurs d'applications d'avoir à se préoccuper de la disponibilité de la mémoire.

La pagination à la demande pure ne charge jamais une page en mémoire avant qu'elle ne soit réellement référencée. La première référence provoque un défaut de page au moniteur résident du système d'exploitation. Le système d'exploitation consulte une table interne afin de déterminer l'emplacement de la page dans la mémoire auxiliaire. Il trouve ensuite un cadre de page libre et il y place la page. La table de pages est actualisée afin de refléter ce changement. Enfin, on redémarre l'instruction qui avait produit le défaut de page. Cette approche permet à un processus de s'exécuter même si son image mémoire entière ne se trouve pas tout de suite en mémoire principale. Tant que le taux de défauts de pages se maintient raisonnablement bas, la performance est acceptable.

Utiliser la pagination à la demande permet de réduire le nombre de cadres de pages alloués à un processus. Cet arrangement augmente le degré de multiprogrammation (en permettant à plus de processus de s'exécuter en une seule fois) et, au moins théoriquement, l'utilisation de l'UC du système. Il permet également aux processus d'être exécutés même si leurs besoins en mémoire dépassent la mémoire physique disponible totale. Des tels processus s'exécutent dans la mémoire virtuelle.

Si les besoins en mémoire totaux excèdent la mémoire physique, il peut être nécessaire de remplacer des pages de la mémoire afin de libérer des cadres de pages pour les nouvelles pages. On utilise divers algorithmes de remplacement de pages. Le remplacement de pages FIFO est facile à programmer, mais souffre de l'anomalie de Belady. Le remplacement de pages optimal demande des dons de voyance. Le remplacement LRU est presque optimal, mais il est encore difficile à implémenter. La plupart des algorithmes de remplacement de pages, comme l'algorithme de la seconde chance, sont des approximations du remplacement LRU.

En plus d'un algorithme de remplacement de pages, il faut une politique d'allocation de cadres de pages. L'allocation peut être fixe, suggérant ainsi le remplacement de pages local, ou dynamique, suggérant le remplacement de pages global. Le modèle de l'ensemble de travail suppose que les processus s'exécutent en possédant des localisations de références. L'ensemble de travail est l'ensemble de pages dans la localisation courante. On devrait donc allouer à chaque processus suffisamment de cadres de pages pour son ensemble de travail courant.

Si un processus ne possède pas suffisamment de cadres de pages pour son ensemble de travail, il échouera. Le fait de fournir suffisamment de cadres de pages à chaque processus, afin d'éviter l'écroulement, demande la mise en oeuvre du *swapping* et du *scheduling* de processus.

En plus de résoudre les problèmes principaux retrouvés dans le remplacement de pages et l'allocation de cadres de pages, une conception correcte d'un système de pagination demande que l'on prenne en compte la taille de la page, les entrées/sorties, le verrouillage, la prépagination, la structure du programme et d'autres problèmes encore. On peut voir la mémoire virtuelle comme un niveau dans une hiérarchie de niveaux de stockages dans un système informatique. Chaque niveau possède son propre temps d'accès, taille et paramètres de coût.

2.1.3 MEMOIRE REPARTIE PARTAGEE

La mémoire répartie partagée (MRP) est une abstraction utilisée pour partager les données entre ordinateurs qui n'accèdent pas la même mémoire physique. Les processus accèdent la MRP, par des lectures et par des écritures, comme si s'agissait de leur propre mémoire physique dans leur espace d'adressage. Cependant, un système en dessous la MRP assure la transparence, afin que les processus s'exécutant dans des machines différentes voient leurs modifications respectives. Tout se passe comme si les processus accédaient une simple mémoire partagé, mais en réalité la mémoire physique est répartie (voir Figure 2).

Le point principal de la MRP est celui de permettre au programmeur de se libérer de l'implémentation des passages de messages dans le développement des applications qui, autrement, devraient les utiliser. La MRP est originairement un outille pour des applications parallèles ou pour toute application distribuée - ou groupe d'applications - dans lesquelles les données partagées peuvent être accédé directement. La MRP est normalement moins approprié pour les systèmes client-serveur, où les clients voient les ressources en possession du serveur comme des données abstraites et les accèdent par des requêtes (pour des raisons de modularité et protection). Cependant, les serveurs peuvent fournir une MRP qui est partagée entre les clients. Par exemple, les fichiers mappés qui sont partagés et pour lesquels on maintient un degré de cohérence ont la forme d'une MRP.

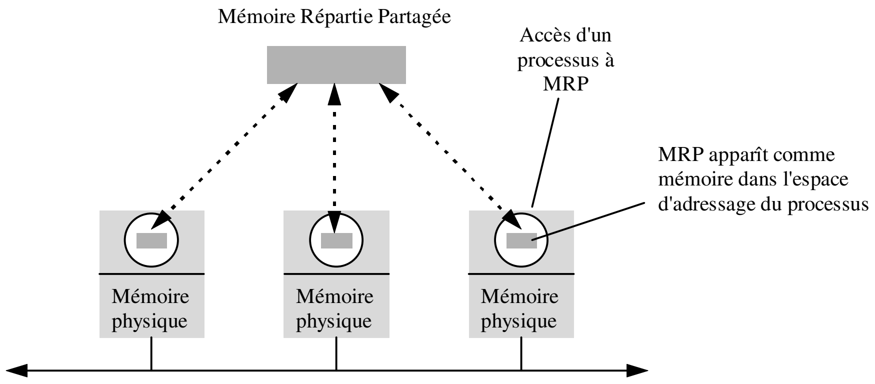

Figure 2. L'abstraction de la mémoire répartie partagée.

Il existe trois principales approches pour implémenter la MRP, qui englobent respectivement l'utilisation du *hardware*, de la mémoire virtuelle ou d'une bibliothèque de support:

hardware: Quelques architectures multiprocesseurs (Dash, PLUS) dépendent d'un équipement matériel spécialisé pour effectuer des chargements et des enregistrements d'instructions sur les adresses dans la MRP, mais aussi pour communiquer avec des modules de mémoire distants.

pagination: Ivy, Mirage, Clouds, Choices, COOL et Mether implémentent la MRP comme une région de mémoire virtuelle qui a la même porté d'adresses dans l'espace d'adressage de tout processus participant. Dans tous les cas, le noyau assure la cohérence de données dans la région de la MRP en la considérant comme une fonction supplémentaire du traitement de défaut de page.

bibliothèque: Quelques langages ou extensions de langage comme Orca et Linda supportent plusieurs formes de MRP. Dans ce type d'implémentation le partage n'est pas implémenté par le système de mémoire virtuelle mais par des communications entre objets du langage en temps d'exécution. Les processus font des appels à la bibliothèque qui le compilateur insère lorsqu'ils accèdent des données dans la MRP.

Les principales caractéristiques d'un système de MRP sont: la structure de données utilisée dans la MRP; le modèle de synchronisation permettant d'accéder à la MRP de façon cohérente du point du vue de l'application; le modèle de cohérence qui s'occupe de la cohérence des valeurs des données accédés par des ordinateurs différents; l'option de mise à jour pour communiquer les écritures faites entre ordinateurs; la granularité et l'effet correction (*thrashing*).

2.1.3.1 Structure

La vision qu'a le programmeur du contenu de la MRP change suivant la granularité utilisée. La construction de la MRP passe donc par le choix d'un grain. Les trois approches principales sont l'octet, les objets partagés ou les données dénombrées immuables.

octets: Permet aux applications d'imposer toutes sortes de structure de données voulue dans la mémoire partagée.

objets partagés: L'avantage de voir la mémoire partagée comme une collection d'objets est de pouvoir inclure la synchronisation aux opérations de manipulation de l'objet.

données immuables: Agora et Linda voient la MRP comme une collection de données immuables qui tout processus peut lire. Les processus doivent remplacer les données dans la MRP et non de les modifier (par exemple, Linda utilise le modèle avec de tuples).

2.1.3.2 Synchronisation

Pour utiliser la MRP, un service de synchronisation distribué doit être fourni, qui utilise des constructions assez connues comme les *locks* et les *sémaphores. De m*ême pour un modèle de MRP objet, les mécanismes de synchronisation sont inclus dans les méthodes de manipulation des objets. La synchronisation est mise en oeuvre par passage de messages. Un groupe d'instructions spécial tel que *test-and-set,* utilisé pour la synchronisation dans la mémoire partagé des multiprocesseurs, reste un principe applicable ‡ la MRP, mais leur emploie dans les cas distribués est très inefficace. Les cas les plus récents de MRP tirent profit des synchronisations de l'application pour réduire le montant des transmissions due aux mise ‡ jour. L'a MRP, alors, inclue la synchronisation comme un composant intégré.

2.1.3.3 Cohérence

Les principaux modèles qui peuvent être réalisé dans la pratique pour les implémentations de la MRP sont la cohérence séquentielle et les modèles basés dans la cohérence faible.

Cohérence Séquentielle

La cohérence séquentielle donne comme résultat de toute exécution, l'équivalent aux conditions suivantes:

- toutes les lectures et toutes les écritures faites par un processus individuellement sont satisfaites dans l'ordre;

- les opérations sur la mémoire appartenant à des processus différents sont exécutées dans un ordre serial.

Plus concrètement, aucun processus ne peut lire de valeur qui soit en conflit avec son ordre programme ou avec l'ordre programme d'un autre processus participant.

Cohérence faible

La cohérence faible a été développée afin d'éviter le coût de la cohérence séquentielle dans les multiprocesseurs. Ce modèle exploite les opérations de synchronisation pour relâcher la cohérence mémoire, laissant au programmeur la tâche d'implémenter la cohérence séquentielle par lui même. Par exemple, si le programmeur utilise un verrou pour implémenter une section critique, alors un système de MRP peut assumer qu'aucun autre processus ne peut accéder les données qui sont en exclusion mutuelle. Il serait alors redondant pour le système de la MRP de propager les mise-à-jour faites sur ces données avant que le processus ne sorte de la section critique. Même si certaines copies de ces données sont inconsistantes une partie du temps, elles ne sont pas accédé pendant ce temps; l'exécution semble alors être cohérente séquentiellement.

2.1.3.4 Mise-à-jour

On distingue deux principales façons de mettre en oeuvre les mise à jour entre processus: mise-à-jour à l'écriture (*write update*) et invalidation à l'écriture (*write invalidate*).

Mise-à-jour à l'écriture

Les mises ‡ jour faites localement par un processus sont diffusées sélectivement à tout autre gérant en possession d'une copie de cette donnée. Ces gérants modifient alors immédiatement leur copie locale de la donnée. Les processus lisent les copies locales des données sans avoir à communiquer. Cela permet à plusieurs lecteurs ou écrivains d'accéder simultanément à leur copie; ce mécanisme est connu comme "partage entre plusieurs lecteurs et plusieurs écrivains".

Le modèle de cohérence implémenté avec la "mise-à-jour à l'écriture" dépend principalement des propriétés de diffusion sélective. La cohérence séquentielle peut être obtenue en utilisant des diffusions sélectives totalement ordonnées, qui ne retournent pas avant que le message de mise-à-jour ne soit délivré localement. Tous les processus sont alors en accord sur un ordre des mises ‡ jour.

Dans la "mise-à-jour à l'écriture" les lectures sont peu coûteuses en termes de messages, cependant les protocoles de diffusion sélective sont relativement chers pour être implémenté par logicielle. On peut alors utiliser un support matériel pour implémenter ce protocole (PLUS) ou laisser comme option la "mise-à-jour à l'écriture" (Munin).

Invalidation à l'écriture

Son utilisation permet le partage entre plusieurs lecteurs et un seul écrivain. Un objet de donnée peut être accédé en "lecture seule" par un ou plusieurs processus, mais il ne peut être "lu et modifié" que par un seul processus. Un objet qui est accédé en "lecture-seule" peut être copié indéfiniment par les autres processus. Lorsqu'un processus veut écrire sur l'objet, il faut tout d'abord envoyer un message par diffusion sélective vers toutes les autres copies afin de les invalider. Tout processus qui veut accéder l'objet est bloqué s'il y a un écrivain. Le contrôle de l'objet est transféré et d'autres accès peuvent être faits après que la mise-à-jour a été effectuée. Cela a pour effet d'exécuter tous les accès à l'objet avec une politique "premier arrivé, premier servi". La cohérence séquentielle est alors assurée, comme cela a été prouvé par Lamport.

Dans ce protocole, la mise-à-jour est propagée seulement lorsque la donnée est lue après une écriture par un autre processus. Plusieurs mises à jour peuvent être faite avant qu'on ait à communiquer. Par contre, il faut compter le coût de l'invalidation des copies en "lecture-seule" avant qu'une écriture ne puisse avoir lieu.

2.1.3.5 Granularité

Une question en relation avec la structure de la MRP est la granularité du partage. En théorie, tous les processus partagent le contenu entier de la MRP. Cependant, tandis que les programmes qui partagent la MRP s'exécutent, seulement certaines parties des données sont réellement partagées et seulement un petit nombre de fois pendant l'exécution. Quelle doit être l'unité de partage dans la MRP? Après un processus a écrit dans la MRP, quelles données le système de la MRP doit envoyer pour fournir des valeurs cohérentes aux autres processus?

Beaucoup de mise-en-oeuvre utilise la page comme base de partage. Ce choix est obligatoire si on veut utiliser le système de mémoire virtuelle sous-jacent. La taille d'une page est de normalement quelques Koctets. C'est une quantité assez grande de données à transmettre via un réseau afin de maintenir cohérentes des copies distantes. Par défaut, le prix de transfère de toute une page doit être payé, même si un seul octet est modifié.

Utiliser une taille petite pour la page (512 octets ou 1 Koctets) ne garantit pas nécessairement une bonne performance. Dans le cas d'un processus mettant à jour beaucoup de données contiguës, le meilleur moyen est d'envoyer une grande page au lieu de plusieurs petites, afin de tenir compte du temps de latence logiciel qui est fixe. Le travail de la MRP augmente si on administre de nombreuses petites unités séparées. En outre, la probabilité que les données accédées par un processus soit dans la même page augmente de façon proportionnelle à la taille de la page. Par contre, deux processus qui accèdent la même page mais modifient des données différentes sont censés partagé la page même si leurs écritures ne sont pas en conflit; ce qui suggère un grain de page petit. On appelle cela de faux partage.

Dans la pratique, le choix de l'unité à partager doit être fait selon la taille des pages physiques disponibles. La conception des données par rapport aux limites des pages est un facteur important pour déterminer le nombre de pages à transférer lorsqu'un programme s'exécute.

2.1.3.6 Effet Correction

Un problème potentiel avec les protocoles d'invalidation à l'écriture est "l'effet correction" (*thrashing*). Cela a lieu lorsque le système de la MRP dépense une énorme quantité de son temps à invalider et à transférer de données, en comparaison avec le temps dépensé avec les applications de processus qui font le travail utile. Cela peut arriver quand plusieurs processus rentrent en compétition pour la même donnée ou pour des données en faux partage.

2.2 BSD 4.4 et Free-BSD 2.2.0.

Le système de mémoire virtuelle du BSD 4.4 est complètement différent de celui du BSD 4.3 et ses prédécesseurs. La mise-en-oeuvre est basée sur le système de mémoire virtuelle du Mach 2.0 avec des mises à jour du Mach 2.5 et Mach 3.0. Le système de mémoire virtuelle du Mach a été adopté parce qu'il a un support de partage très efficace et une séparation nette des caractéristiques "machines-dépendantes" et "machines-indépendantes". En outre, il a un support pour multiprocesseurs, même si celui-ci n'est pas souvent utilisé. Aucun appel système de l'interface original du Mach n'a été retenu. Tout a été remplacé par l'interface proposée par BSD 4.2 qui est largement utilisé dans l'industrie d'UNIX.

2.2.1 LE SYSTEME DE MEMOIRE VIRTUELLE

Le système de mémoire virtuelle fournit des espaces d'adressages protégés dans lesquels on peut "mapper" des sources de données (objets) comme des fichiers ou privés, ou des morceaux anonymes d'espace de *swap*. La mémoire physique est utilisée comme un cache des objets les plus récemment référencés et elle est gérée par un algorithme approchant celui de BSD 4.3.

L'espace d'adressage virtuelle de la plupart des architectures est divisé en deux parties: l'espace réservé du noyau, d'une taille de 30 à 100 Mo, située dans la partie haute de la mémoire; et l'espace utilisateur qui compris toute la mémoire laissée libre par le noyau. Le début des zones de texte et de données commence aux adresses les plus proche du début de la mémoire. Habituellement, les 4 ou 8 premiers Koctet de mémoire sont maintenus hors porté des processus. Les allocations mémoires faites par le processus à l'aide de la fonction de bibliothèque *malloc()* (ou l'appel système *sbrk*) sont effectuées dans le tas qui commence juste après la zone de données et croît vers les adresses les plus hautes. Les vecteurs d'arguments et les vecteurs d'environnements sont dans la partie haute de l'espace d'adressage de l'utilisateur. La pile de l'utilisateur commence juste en dessous de ces vecteurs et croît vers les adresses les plus basses. Limités seulement pour des raisons administratives, la pile et le tas peuvent chacun croître jusqu'à leur rencontre.

Dans BSD 4.4 et d'autres systèmes UNIX modernes qui supportent l'appel système *mmap*, l'espace d'adressage de l'usager est moins structuré. Les implémentations des bibliothèques partagées peuvent placer les textes et les données de façon arbitraire. La notion de régions prédefinies devient alors obsolète. Pour assurer la compatibilité, BSD 4.4 supporte encore l'appel à *sbrk, utilisé par malloc(),* de plus, le noyau possède une région de pile désignée où des allocations adjacentes sont faites automatiquement.

A tout moment, le processus courant "mappé" utilise la totalité de l'espace d'adressage virtuel (sauf l'espace du noyau). Lorsque le système décide de commuter le contexte pour un autre processus, il doit sauvegarder les informations concernant les adresses "mappées" du processus courant, par ensuite charger les adresses "mappées" du nouveau processus afin qu'il puisse s'exécuter. Les détails de cette commutation des adresses "mappées" sont dépendants de l'architecture. Certaines architectures n'ont qu'à changer quelques registres de "mappage-mémoire" qui pointent vers (base et taille) des tables de pages résidentes en mémoire. D'autres architectures stockent les descripteurs de table de page dans une partie spécial de la mémoire haute (RAM). Le fait de commuter ces tables peut exiger la sauvegarde et le rechargement de centaines d'entrées de ces tables.

Le noyau et le processus usager utilisent à la base les mêmes structures de données pour gérer leur mémoire virtuelle. Les structures de données utilisées pour gérer la mémoire virtuelle sont les suivantes:

vmspace Contient les structures dépendantes et indépendantes de la machine. Elles décrivent l'espace d'adressage d'un processus.

vmmap Le niveau le plus haut de structure de données qui décrit l'espace d'adressage virtuel indépendant de la machine.

vm_map_entry Structure qui décrit un intervalle virtuel contigu d'espaces d'adressages qui partagent la même protection et le même héritage des attributs.

object Structure qui décrit la source de la donnée pour un intervalle d'adresses.

shadow object Objet spécial qui représente la copie modifiée de la donnée originale.

vm_page Le niveau le plus bas de structure de données qui représente la mémoire physique en train d'être utilisée par le système de mémoire virtuel.

La figure (voir) montre un espace d'adressage typique d'un processus et les structures de données associées. La structure *vmspace* contient l'état de la mémoire virtuelle d'un processus en particulier, et inclut les structures de données dépendantes de la machine et celles indépendantes de la machine et aussi des statistiques. La structure *vm_pmap* et les fonctions associées fournissent une mémoire virtuelle, abstraite, commune, que toutes les architectures matérielles doivent simuler. Les structures de données indépendantes de la machine incluent l'espace d'adressage qui est représenté par la structure *vm_map*. La structure *vm_map* contient une liste chaînée de structures *vm_map_entry*, ainsi que des suggestions permettant d'accélérer la recherche pendant une allocation mémoire. Enfin, elle contient les structures nécessaires aux traitements de défauts. Toute structure *vm_map_entry* pointe vers une chaîne de structures de *vm_object* qui décrivent les sources des données (objets) "mappées" dans les intervalles d'adresses indiqués. A la queue de la chaîne on trouve la donnée (objet) "mappée" originel. Le plus souvent elle représente une source de donnée persistante, tel qu'un fichier. Interposé entre cet objet et l'entrée du *vm_map_entry* on trouve un ou plusieurs objets *shadow* transitoires qui représentent des copies modifiées de la donnée originelle.

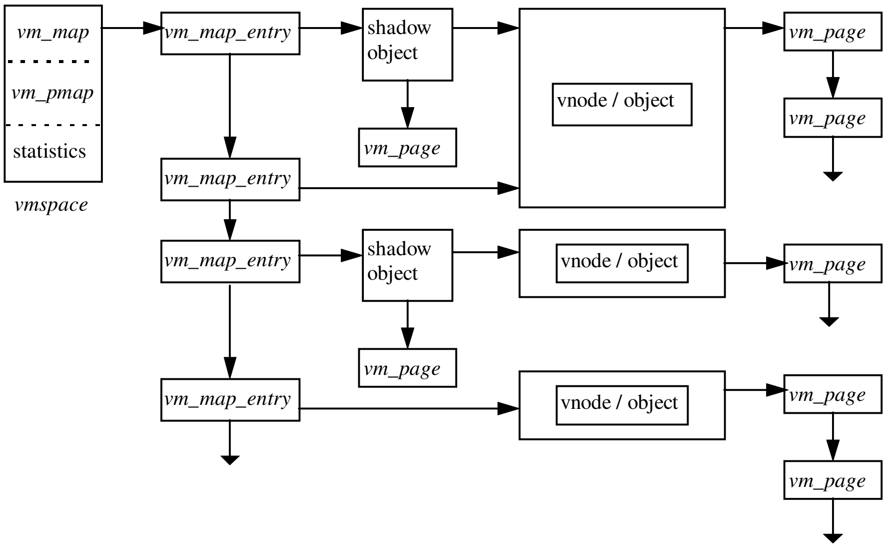

Figure 3. Structures de données qui décrivent l'espace d'adressage d'un processus.

Chaque structure de *vm_object* contient une liste chaînée de structures de *vm_page* qui représente le cache en mémoire physique de l'objet, ainsi qu'un pointeur vers la structure *pager_struct* qui détient des informations pour retrouver ("*page in"* ou "*page out")* les données dans la mémoire secondaire. Il y a une structure *vm_page* allouée pour toute page de la mémoire physique gérée par le système de mémoire virtuelle. Dans BSD, une page peut être une collection de multiples pages physiques contiguës, qui seront traitées par la couche dépendante de la machine comme si était une seule unité. La structure contient aussi le statut de la page (modifiée ou référencé) ainsi que des liens vers plusieurs queues de pagination.

Toutes les structures possèdent les "inter verrous" nécessaires pour le *multithreads* dans un environnement multiprocesseur. Le grain de verrouillage est fin, avec au moins un verrou par instance de structure de données. Plusieurs structures contiennent des verrous multiples leur permettant de protéger des champs individuels

2.2.1.1 La Gestion Mémoire du Noyau

Le noyau est "mappé" de façon permanent dans la partie haute de l'espace d'adressage de tout processus. Dans ce modèle, commuter un processus vers un autre n'affecte pas le morceau d'espace d'adressage du noyau. Avoir le noyau "mappé" de façon permanent réduit la quantité d'espace d'adressage disponible pour un processus (ainsi que pour le noyau), en compensation, le coût nécessaire pour copier les données d'un espace vers un autre, ainsi que le temps de commutation sont réduits. En effet, étant "mappé" de façon permanente, il n'est pas nécessaire de commuter vers l'espace d'adressage du noyau. De plus, les recopies de données entre l'espace d'adressage du noyau et celui du processus courant se font par simple recopie de bloc.

Les "mappes" et "sous-mappes" du noyau

Lorsque le système démarre, la première tâche que le noyau doit effectuer est celle d'initialiser les structures de données pour décrire et gérer son espace d'adressage. Comme tout processus, le noyau possède une structure *vm_map* avec un ensemble de structures *vm_map_entry* correspondantes qui décrivent un intervalle d'adresses. Les sous-mappes sont une construction spéciale au noyau, utilisé pour isoler et forcer l'allocation d'espace d'adressage pour les sous systèmes du noyau.

2.2.1.2 Ressources par Processus

Un processus requiert une entrée dans la table de processus et une pile du noyau. L'autre principal ressource qui doit être alloué est sa mémoire virtuelle. Les exigences initiales pour la mémoire virtuelle sont définies dans l'en-tête du processus exécutable. Ses exigences incluent l'espace nécessaire pour le texte du programme, les données initialisées, les données non initialisées et la pile. Au lancement du programme, le noyau construira les structures de données nécessaires pour décrire ces quatre zones. La plupart des programmes ont besoin d'allouer de mémoire additionnelle. Le noyau fournit cette mémoire additionnelle par une expansion de la zone de données non initialisée.

2.2.1.3 Défaut de Page

Lorsqu'un processus essaye d'accéder une partie de son espace d'adressage qui n'est pas résident, un défaut de page se produit. Le traitement d'un défaut de page dans le noyau a les quatre étapes suivantes:

1. Trouver la structure *vmspace* du processus qui a occasionné le défaut; à partir de cette structure, trouver l'en-tête de la liste *vm_map_entry.*

2. Traverser la liste *vm_map_entry* en partant de l'entrée indiquée par le "map hint"; pour chaque entrée, vérifier si l'adresse manquante tombe entre l'intervalle d'adresse *start* et *end*. Si le noyau atteint la fin de la liste sans trouver aucune région valide, il envoie un signal de *segment fault*.

3. Convertir l'adresse vers un offset absolu dans la page de l'objet référencé par la *vm_map_entry*.

4. Donner l'*offset* absolu pour l'objet souligné, qui alloue une structure *vm_page* et utilise son ***pager*** pour remplir la page. L'objet retourne donc un pointeur vers la structure *vm_page*, laquelle est "mappée" dans l'endroit où a eu lieu le défaut dans l'espace d'adressage du processus.

La routine de traitement de défaut de page retourne et réexécute l'instruction défaut.

2.2.2 L'INTERFACE DE PAGINATION

L'interface de pagination fournit les mécanismes par lesquels les données sont transférées entre la mémoire secondaire et la mémoire physique. L'interface de pagination du BSD 4.4 est basée dans la page, avec toutes les requêtes de données faites en multiple de la taille de la page du logiciel. Les structures *vm_page* sont passées comme descripteurs qui contiennent l'offset dans la mémoire secondaire et les adresses du cache physique des données désirées. Cette interface est "interne" dans le sens que les *pagers* sont compilés dans le noyau et les routines de pagination sont évoquées par un simple appel de fonction.

Chaque objet a une structure *pager_struct* associée, qui représente une instance du type du *pager* responsable pour fournir le contenu des pages dans l'objet. Cette structure contient des pointeurs vers des routines de type spécifique pour lire et écrire des données, ainsi qu'un pointeur vers l'instance spécifique de stockage.

Une instance de *pager* est créée au même moment que l'objet, c'est à dire, quand un fichier, un périphérique ou un morceau anonyme de mémoire est "mappé" dans l'espace d'adressage d'un processus. L'instance du *pager* existe jusqu'à que l'objet soit désaloué. Lorsqu'un défaut de page a lieu pour une adresse virtuelle qui "mappe" un objet en particulier, le code de traitement alloue une structure *vm_page* et convertie l'adresse fautive en déplacement (*offset*) dans l'objet. Ce déplacement est enregistré dans la structure *vm_page*, et la page est additionnée à la liste de pages en cache de l'objet.

L'instance du *pager* est aussi responsable de la sauvegarde du contenu d'une page modifiée (sale) si le système décide de la mettre en mémoire secondaire. Quand le démon *pageout* décide qu'une page n'est plus nécessaire, il demande à l'objet propriétaire de la page de la libérer. L'objet passe d'abord l'*offset* logique pour l'instance du *pager* pour que la page soit sauvée. L'instance du *pager* est responsable pour trouver un endroit approprié pour la sauvegarde de la page, effectuant tout entrée/sortie nécessaire et ensuite notifiant l'objet que la page peut être libérée. Après cela, l'instance du *pager* notifie le démon *pageout* pour qu'il enlève la structure *vm_page* de l'objet et la mettre dans la liste libre.

Il y a sept routines associées à chaque type de *pager*. La routine *pgo_init* est appelée au moment du démarrage pour effectuer les initialisations (une seule fois) spécifiques au type. Les routines *pgo_alloc* et *pgo_dealloc* sont appelées quand une instance de page doit être créée ou être détruite. La routine d'allocation est appelée toutes les fois qu'un objet est mappé dans un espace d'adressage par *mmap*. C'est pourquoi, seulement le premier appel devrait créer la structure; des appels successifs font qu'augmenter le compteur de référence pour l'objet associé et retourne un pointeur vers l'instance du *pager* existante. La routine pour désallouer est appelée seulement quand le compteur tombe à zéro.

Pgo_getpages est appelée pour retourner une ou plusieurs pages de données d'une instance du *pager* de façon synchrone ou asynchrone. Normalement, cette routine est appelée par le code de traitement de défaut de page pour remplir de façon synchrone des pages simples. *Pgo_putpages* est appelée par le démon *pageout* pour écrire une ou plusieurs pages de façon asynchrone, et par *msync* pour écrire de simples pages de façon synchrone ou asynchrone. Ces deux routines sont appelées avec un tableau de structures *vm_page* qui indique les pages affectées.

Pgo_cluster prend un *offset* et retourne un intervalle fermé d'*offset* représentant une unité d'entrée/sortie optimale vers la mémoire secondaire. Cet intervalle peut être utilisé avec *pgo_getpages* et *pgo_putpages* pour aider un préchargement informé ou un nettoyage des pages (clusters). La routine *pgo_haspage* porte une instance du *pager* pour voir si cette instance a des données dans un *offset* particulier de la mémoire secondaire.

Les trois types de *pagers* que le système support sont:

- **Vnode Pager**

  Le *vnode pager* traite des objets qui "mappent" de fichiers dans le système de fichiers. Chaque fois qu'un fichier est "mappé" de façon explicite par *mmap* ou de façon implicite par *exec*, la routine d'allocation du *vnode pager* est appelée. Si l'appel correspond au premier "mappage" du *vnode*, la structure nécessaire et spécifique du *vnode-pager* est créée, et un objet de taille approprié est alloué et associé avec l'instance du *pager.* Le compteur de référence du *vnode* est augmenté pour refléter la référence du *pager.*

- **Device Pager**

  Le *device pager* traite des objets qui représentent des périphériques matériel "mappés" en mémoire. Les périphériques "mappés" en mémoire fournissent une interface qui ressemble à un morceau de mémoire. Un exemple d'un périphérique "mappé" en mémoire est un tampon de fenêtre (*frame buffer),* qui fournit un intervalle d'adresses en mémoire avec un mot par pixel sur l'écran. Le noyau fournit l'accès vers un périphérique "mappés" en mémoire, en "mappant" la mémoire du périphérique dans un espace d'adressage d'un processus. Le processus peut alors accéder cette mémoire sans autre intervention du système d'exploitation. Ecrire sur un mot du tampon de fenêtre *(frame buffer)* revient à faire le pixel prendre la couleur ou le brille approprié.

  Le *device pager* est fondamentalement différent des autres deux *pagers* dans le sens qu'il ne remplit pas les pages de la mémoire avec des données. A la place, il crée et gère sa propre structure *vm_page*, où chacune décrit une page de l'espace du périphérique. Cette approche fait que la mémoire du périphérique ressemble à une mémoire physique reliée. Donc, aucun code spécial devait être nécessaire dans le restant du système de la mémoire virtuelle pour traiter le périphérique en mémoire.

- **Swap Pager**

Dans l'usage la plus fréquente, le *swap pager* correspond au *pager* utilisé par des objets qui "mappent" de la mémoire anonyme. Ce *pager* est dit le *pager* par défaut, car il est utilisé quand aucun autre *pager* a été requis. Il fournit un espace de *swap*: mémoire secondaire non persistante qui est remplie par des zéros à la première référence. Grâce à l'optimisation du remplissage par des zéros et à la nature transitoire de la mémoire secondaire, l'allocation des ressources du *swap pager* pour un objet en particulier peut être différé jusqu'à le premier *pageout*.

Le rôle du *swap pager* est la gestion du espace de *swap*: trouver où stocker les pages sales (modifiées) et comment les rencontrer quand elles sont de nouveau requises. Les objets de type *shadow* ont besoin que ces opérations soient efficaces. Déterminer si une instance du *pager* possède une page en particulière doit être rapide, de préférence sans avoir à faire d'entrée/sortie. Le *swap pager* doit faire des écritures asynchrones, en mémoire secondaire, des pages modifiées. Cela pour satisfaire une nécessité du démon *pageout*.

L'instance du *swap pager* ne sera créée que quand la page de l'objet est remplacée en mémoire. A ce moment, une structure est allouée pour décrire l'espace de *swap* qui peut tenir l'objet. Cet espace de *swap* est décrit par un tableau de blocs de *swap* de taille fixe. La taille de chaque bloc est sélectionné prenant pour base la taille de l'objet que le *swap* est en train de gérer.

Les deux structures créées par le *swap pager* sont: *swpager* et *swblock* (voir Figure 4). La structure *swpager* décrit la zone de *swap* en train d'être gérée par le *pager*. Elle enregistre la taille totale de l'objet (*object size*), la taille de chaque bloc de *swap* en gestion (*block size*), et le nombre de blocs de *swap* qui forment la zone de *swap* de l'objet (*block count*). Elle a aussi un pointeur vers un tableau de "block-count" de structures *swblock,* chacune contenant un numéro de bloc de périphérique (*device*) et un bit de masquage. Le masque d'un bit par page dans ce bloc de *swap* enregistre lesquelles des pages du bloc ont de données valides. Ensemble, le tableau de *swblock* et les bits de masquage fournissent deux niveaux de table de page qui décrivent la mémoire secondaire d'un objet. Cette structure fournit une allocation du espace de *swap* efficace pour des objets peu peuplés, puisqu'un bloc de *swap* donné n'a pas besoin d'être alloué jusqu'à la première fois qu'une page dans les limites de la taille du bloc est écrire en mémoire secondaire.

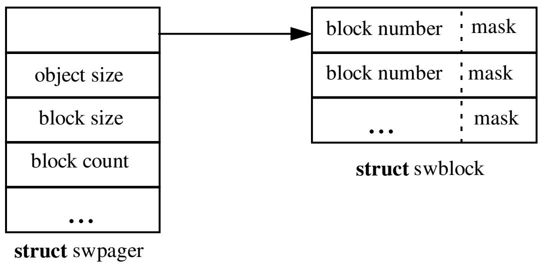

Figure 4. Les structures utilisées pour gérer l'espace du *swap*.

La taille de l'objet est figé au moment de l'allocation. Donc, si la zone anonyme continue à grandir (comme la pile ou le tas d'un processus), un nouvel objet doit être créé pour décrire la zone d'expansion. Dans un système qui a peu de mémoire, le résultat est qu'un processus grand peut acquérir plusieurs objets anonymes.

L'allocation de blocs de *swap* de la réserve d'espace de *swap* du système est gérée par une ressource *map* appelé *swapmap*. Le *swapmap* est initialisé au moment du démarrage pour contenir tout l'espace de *swap* disponible. Un indice dans le *swapmap* sur l'espace qui a été alloué, est utilisé comme un indice de l'adresse sur disque de la zone de *swap*.

Chapitre 3
Réseaux à haut débit

3.1 INTRODUCTION

Le développement accéléré vers des débits toujours plus élevés est la conséquence d'une course poursuite entre les utilisateurs et les opérateurs ou fournisseurs alimentée par de nombreuses opportunités: nouvelles applications et nouvelles technologies attractives. Toutes les entreprises aujourd'hui sont devenues dépendantes des réseaux et des échanges d'informations entre applications informatiques. La perte de services de réseaux est maintenant considérée comme un sérieux inconvénient, qui induit des délais inacceptables, des pertes de revenus quand elle n'entraîne pas carrément la faillite de l'entreprise.

L'automatisation des chaînes de production est une voie intéressante pour améliorer la productivité de l'entreprise. Cette automatisation nécessite des équipements intelligents (ordinateurs...). Une quantité d'ordinateurs exécutant des calculs distribués rend indispensables les moyens de communication donc les réseaux. Ce passage de la situation *terminal connecté à un ordinateur central* à une situation de *communication et coopération d'égal à égal dans un environnement distribué* a déterminé l'évolution des architectures de réseaux de la solution hiérarchique à la solution "à plat", courante aujourd'hui dans les topologies en bus ou en anneau.

L'évolution de l'informatique distribuée vers une informatique en réseau avec des serveurs et des appels de procédures à distance et la rapide augmentation des puissances de machines intelligentes à connecter entre elles ont causé également une modification dans les modèles de réseaux, en particulier, ces dernières années, en précipitant les besoins en matière de réseaux haut débit.

Des réseaux encore plus rapides sont nécessaires essentiellement pour trois raisons liées les unes aux autres. Tout d'abord, les réseaux grandissent. Des noeuds intelligents sont continuellement rajoutés. Les réseaux isolés sont reliés entre eux pour former un réseau intégré d'entreprise. Cette tendance à la croissance des réseaux a fait croître le trafic. La croissance du trafic nécessite d'augmenter les capacités des réseaux si l'on cherche à maintenir les délais de remise et les temps de réponse.

La deuxième raison à l'origine des besoins en débits toujours plus élevés est liée à l'évolution de la puissance des équipements informatiques connectés aux réseaux. Aujourd'hui un ordinateur personnel a typiquement une puissance de 2 à 8 Mips, il contient quelques Moctets de mémoire, dispose d'un bus interne "efficace" et peut effectuer des entrées/sorties à quelques Moctets/seconde. Les stations de travail récentes ont des puissances comprises entre 20 et 100 Mips, elles disposent de fortes extensions de mémoire et peuvent traiter les entrées/sorties à 100 Mbit/s. Les ordinateurs hôtes travaillent entre 500 MFLOPS et 2 GFLOPS, ils peuvent avoir jusqu'à 2 Goctets de mémoire et des canaux à 100 Mbit/s. Il arrive maintenant des machines de 10 à 40 GFLOPS, ayant au moins 8 Goctets de mémoire et des canaux à 800 Mbit/s. Si l'on considère ensemble ces différentes améliorations de puissance, on aboutit à une accélération vertigineuse des besoins de communication. De plus, les réseaux de transmission de données et les capacités de traitement des noeuds augmentant, les traitements applicatifs sont utilisés de manière plus intensive; ils participent ainsi à la croissance des besoins en réseau. Les serveurs de fichiers en réseau et les appels de procédure à distance sont deux éléments du modèle de réseau qui entrent dans cette catégorie.

Le facteur de coût décroissant est également une motivation pour le développement de nouveaux réseaux toujours plus rapides. Le coût par bit a toujours été en chute "exponentielle". On admet généralement que le coût par bit est divisé par trois chaque fois que le débit est multiplié par quatre. La capacité des systèmes optiques augmente rapidement. Les développements des lasers simple fréquence permettent de diminuer de manière significative le nombre des répéteurs pour une couverture donnée, réduisant ainsi les coûts de connexion dans les WAN. L'émergence de systèmes optiques à très longues portées et très haut débit a réduit le coût du bit transporté à un tel point que l'efficacité spectrale n'est plus un paramètre primordial. Les avantages des grandes bandes passantes et des faibles coûts qui profitent aux communications longues distances sont également intéressants dans le contexte des LAN et des MAN. Ces dernières années, le coût de ces réseaux a diminué de 20 à 40% par an.

La combinaison de cette croissance des réseaux, de la demande continuelle de bande passante, des besoins en débits toujours plus élevés pour interconnecter efficacement les machines et les réseaux, avec les aspects économiques des communications par fibres optiques, a créée un mouvement très fort, presque inévitable, pour augmenter sans cesse les débits des systèmes de communication.

3.1.1 LES RESEAUX LOCAUX A HAUT DEBIT

3.1.1.1 Les Réseaux Ethernet 100 Mbit/s et la Commutation

Pour satisfaire à la demande des utilisateurs, le débit des réseaux Ethernet doit augmenter de façon importante. Trois grandes catégories de solutions sont proposées:

- augmentation du débit du support partagé;
- utilisation plus optimisée du support physique par une commutation ou une commutation partielle;
- ajout de canaux synchrones.

Trois propositions de l'IEEE correspondent à ces solutions:

- **Fast Ethernet**

  Fast Ethernet est la dénomination de l'extension à 100 Mbit/s du réseau Ethernet à 10 Mbit/s. La technique d'accès est la même mais à une vitesse multipliée par 10 et les trames transportées sont identiques. La difficulté principale provient du système de câblage et de la possibilité ou non d'y faire transiter des débits aussi grands.

  La distance maximale entre les deux points les plus éloignés est fortement réduite par rapport à la version 10 Mbit/s. La plupart des constructeurs ont limité la distance maximale à 210 mètres.

  Un avantage de cette méthode est la bonne compatibilité avec la version 10 Mbit/s qui permettra de relier sur un même *hub* à la fois des stations 10 Mbit/s et 100 Mbit/s. Le coût de connexion du 100 Mbit/s ne devrait pas dépasser deux fois le coût de l'Ethernet classique, dix fois moins rapide.

- **La commutation Ethernet**

  Une deuxième solution pour augmenter le débit de l'environnement Ethernet est de passer à la commutation Ethernet ou Ethernet FDSE (Full Duplex Switched Ethernet). Historiquement, on a commencé par couper les réseaux Ethernet en des sous-réseaux autonomes reliés entre eux par des ponts en essayant de bien garder un trafic local. De ce fait, on multiplie le trafic par le nombre de sous-réseaux. La logique extrême est de découper le réseau jusqu'à n'avoir qu'une seule station par réseau Ethernet. On tombe sur la commutation Ethernet, où chaque carte coupler est reliée directement sur un commutateur Ethernet qui se charge de rediriger les trames dans la bonne direction suivant un routage qui sera mis en place dans les noeuds de commutation.

  Les difficultés proviennent des techniques de contrôle qu'il faudra mettre en place pour prendre en charge, sur les liaisons intercommutateurs, les débits provenant en même temps de tous les coupleurs Ethernet. On retrouve tous les problèmes posés par une architecture de réseau du type commutation de paquets. Par contre, il n'y a plus de limitation de distance et on peut réaliser des réseaux en commutation Ethernet au niveau mondial.

- **Ethernet IsoNet**

  Ethernet est en réseau fortement asynchrone et il est pratiquement impossible d'y transporter des applications isochrones. Une extension d'Ethernet dans ce sens est proposée et définit une interface d'accès à partir d'une station de travail vers un réseau capable de transporter simultanément des canaux asynchrones et des canaux synchrones. La capacité globale de cette interface est de 16,144 Mbit/s et la distance maximale entre deux *hubs* est de 100 mètres.

3.1.1.2 IEEE 100VG AnyLan

Le but du 100VG AnyLan est d'être une solution acceptable à la fois pour les possesseurs de réseaux Ethernet et Token-Ring pour augmenter la capacité de leur réseau. L'innovation est le protocole de gestion du droit à la parole (Demand Priority). Le principe est que chaque *hub* 100VG AnyLan interroge tout à tour les stations qui lui sont raccordées et assure un droit de parole réparti équitablement entre les machines. Il gère plusieurs niveaux de priorité. Une machine ayant un message à émettre répond à l'interrogation du *hub* par le niveau de priorité correspondant à son message. Le *hub* enregistre toutes les demandes et les satisfait par ordre de priorité. Dans le 100VG AnyLan, les *hubs* doivent contrôler les demandes d'émission en arrêtant les trames arrivant dans le noeud et qui provoqueraient des collisions. Il y a donc parallélisme entre les communications.

L'avantage est donc d'avoir des délais garantis et des trames dans le format Ethernet et Token Ring; et la contrainte est celle d'un faible diamètre.

3.1.1.3 Le réseau local ATM

Le réseau local ATM est une orientation importante de ces dernières années. Il a été propulsé sur le devant de la scène par l'ATM Forum. En fait la solution qui a été retenue est une émulation de réseau Ethernet ou Token Ring. Le transport interne s'effectue par la technique de transfert ATM. Les trames Ethernet ou Token Ring sont découpées en segments ATM. Ces segments sont transportés à l'autre extrémité où ils sont réassemblés pour retrouver la trame originale.

Les interfaces normalisées par l'ATM (haut débit) forum sont les suivantes:

- interface à 100 Mbit/s sur une fibre optique multimode pour utiliser le réseau FDDI comme transporteur. Le codage 4B/5B utilisé donne un débit utile de 88,89 Mbit/s;

- interface à 155 Mbit/s sur une fibre optique multimode ou monomode;

- interface à 155 Mbit/s sur une trame SDH. Le support physique peut être soit une fibre optique multimode, soit une paire torsadée. La distance dans ce dernier cas est limitée à une centaine de mètres. Le débit utile est de 135,63 Mbit/s.

3.2 PROBLEMATIQUE DES RESEAUX HAUT DEBIT

L'augmentation du débit du réseau améliore les possibilités des systèmes mais pose de nouveaux problèmes: la puissance des machines nécessaires à l'exploitation du haut débit, la portabilité difficile des protocoles existants, les temps importants de latence et la coexistence nécessaire avec les réseaux à faible-débit.

L'exploitation d'un réseau à haut débit n'est réellement efficace que si les stations de travail sont suffisamment puissantes pour supporter les débits offerts. Cependant ce goulot d'étranglement est temporaire car la puissance des machines croît plus rapidement que les débits des réseaux.

Le temps de latence est un facteur aussi important que le débit dans l'évaluation des performances des communications. Le protocole TCP n'est pas adapté à une exploitation optimale des performances des réseaux à haut débit car il entraîne des temps de latence important [Trecordi, 1994].

L'utilisation des réseaux à haut débit, aussi bien comme réseaux locaux que comme réseau d'interconnexion, pose le problème de la coexistence de réseaux à débits hétérogènes. A cela s'ajoute le problème de l'hétérogénéité des machines.

3.3 GAINS POUR LES SYSTEMES

L'émergence des réseaux à haut débit permet de repousser les limitations liées aux communications dans les systèmes répartis [Thekkath, 1994]. La prépondérance de ces réseaux dans un futur proche nécessite dès à présent d'étudier et de concevoir de nouveaux systèmes répartis exploitant un débit crête élevé.

Les recherches sur les possibilités des réseaux à haut débit ne concernent généralement que les protocoles de communication et sont particulièrement dédiées au multimédia [Fédaoui, 1994] A partir de ces recherches, un travail d'adaptation sur les systèmes peut être fait pour tirer parti des capacités des réseaux à haut débit.

Les applications parallèles peuvent être exploitées sur nombreuses stations de travail, obtenant des performances comparables à celles d'une machine massivement parallèle, comme nous montrent plusieurs expériences

[Zhou, Wang, Zheng, Delisle, 1992] [Lin, Hsieh, Du, Thomas, MacDonald. 1994].

3.3.1 GESTION REPARTIE DES RESSOURCES

La gestion répartie des ressources peut tirer profit des réseaux à haut débit grâce aux perspectives offertes à la gestion des accès disques et celle de la mémoire centrale. La modification de la hiérarchie des temps d'accès (mémoire, réseau, disque) promet de nouvelles techniques de cache disque et de mémoire virtuelle. Les accès disque deviennent plus coûteux que les accès réseau.

Dans la gestion des caches répartis, quand un fichier est demandé par plusieurs sites, il est moins coûteux d'aller chercher le fichier dans la mémoire d'une autre machine client que de le redemander au serveur. En ce qui concerne les disques répartis, une gestion dynamique de fichiers est aussi envisageable grâce au faible coût du transfert des données.

Lorsque la mémoire physique est saturée, la gestion de mémoire du système peut utiliser la mémoire disponible des autres machines du réseau comme zone de *swap*. Décharger les pages sur la mémoire physique d'autres machines devient moins coûteux que les décharger sur le disque local. En plus, le *swap réparti* permet de soulager le disque local du trafic engendré par le gestionnaire de la mémoire virtuelle et de réduire ainsi les coûts de lecture et d'écriture du *swap*.

3.3.2 MEMOIRE PARTAGEE ET REPARTIE

Les mémoires partagées réparties étaient peu performantes à cause de l'écart des temps d'accès entre la mémoire locale et les réseaux classiques. Maintenant, il est possible de réaliser un protocole avec un faible temps de latence qui permet d'envisager des accès répétés à distance [Thekkath, 1994].

3.3.3 APPLICATIONS PARALLELES

Les réseaux de stations de travail n'offrent pas une vue globale du système tandis que certaines machines parallèles le font. Le coût pour déterminer un état global sur un ensemble de machines géographiquement distantes est trop grand à cause des contraintes physiques. Sur les réseaux classiques, les échanges d'informations nécessaires à la connaissance permanente d'une bonne approximation de l'état global du réseau sont prohibitifs. Selon des mesures faites avec des concentrateurs Ethernet 10 Mbit/s, on ne pouvait pas envisager une répartition de charge de grain moyen au-delà d'une vingtaine de machines [Folliot, 1992]. Dorénavant, les commutateurs (ATM) permettent une augmentation du débit global du réseau qui suit le nombre de machines. C'est pourquoi l'extensibilité du placement est assurée.

Les opérations de réductions parallèles, qui peuvent profiter de l'échange simultané de données, sont optimisées grâce à la disponibilité de communications bidirectionnelles simultanées (*full duplex*) sur les réseaux à haut débit. Cela améliore encore les performances des applications parallèles.

Enfin, la capacité que les réseaux à haut débit ont de se communiquer permet l'exécution d'une nouvelle classe d'applications parallèles de grain moyen-fin fortement communicantes. Ce type d'application aurait un coût de placement trop élevé sur les réseaux à faible débit.

3.4 LE RESEAU HSL ET L'INTERFACE PCI-DDC

Le réseau High Speed Link (HSL) est un réseau local haut débit fonctionnant sur le mécanisme d'écriture en mémoire distante. L'accès au réseau "HSL" passe par l'interface matérielle PCI-DDC. Cette interface est elle même gérée par l'interface logicielle PCI-DDC.

3.4.1 Réseau HSL

Le réseau "High Speed Link" (HSL) est basé sur le standard [IEEE 1355] et sur le routeur RCUBE (Rapid Reconfigurable Router) conçu et développé au laboratoire MASI. Le routeur RCUBE offre un faible temps de latence (150ns). Il se compose de 8 liens bidirectionnels (High Speed serial Link), et d'un "cross-bar" nom bloquant 8x8 ainsi que de 8 tables de routage indépendantes (une par lien). Chaque lien offre un débit de 1Gbits/seconde (soit 80 Moctets/seconde) dans chaque sens de transfert. Cette architecture massivement parallèle permet un débit crête pouvant atteindre 640Moctes/seconde. Chaque machine est connectée au réseau par une carte interface PCI-DDC. Cette carte utilise le lien d'un routeur pour s'insérer dans le réseau. Ainsi, chaque machine dispose alors de 7 liens lui permettant de se connecter aux autres noeuds du réseau. Ce composant matériel est géré par l'interface logicielle PCI-DDC. L'interface matérielle est encore en phase finale de conception, c'est donc l'interface logicielle qui assure de plus la simulation du réseau HSL. Destiné au compatible PC, le système retenu est Free-BSD.

3.4.2 Interface Logicielle PCI-DDC

L'interface logiciel PCI-DDC offre trois niveaux d'utilisation: L'interface Haut, l'interface Basse et l'interface "Put".

♦ **L'interface Put** effectue des accès directs au matériel. Emettre un message consiste à déposer un descripteur dans la liste de pages à émettre (List of Page to Emit : LPE), via une requête a l'interface "put". Un descripteur est constitué, des adresses mémoire source et destination, de la taille du message mais aussi des informations de routage. Ces informations de routage incluent le numéro de machine destination. Le terme message signifie une zone contiguë de mémoire physique. Le principal problème réside dans le passage des adresses. La machine cible doit fournir des adresses physiques à l'émetteur, mais aussi assurer que la page n'est pas réutilisée avant la fin de l'échange (verrouillage de pages). Le protocole fonctionne donc à l'initiative de l'émetteur.

♦ **Les interfaces Basse et Haute** diffèrent entre elles par les adresses respectivement physiques ou logiques qu'elles utilisent. Elles offrent un niveau d'abstraction supplémentaire: le canal. De façon similaire au mécanisme de communication dans les machines parallèles, les messages sont transmis en séquence sur un canal. Contrairement à l'interface "put" l'initiative est au récepteur. Lorsque celui-ci effectue un appel à la fonction "receive" de l'interface, l'adresse locale est transmise à l'émetteur. Chaque nouvelle requête de l'émetteur sur un canal utilise l'adresse de réception suivante. Ce mécanisme suppose donc une parfaite synchronisation entre émetteur et récepteur.

3.4.3 L'émulation de l'interface PCI-DDC

L'interface logicielle est composée d'un gestionnaire de périphérique ("driver"), d'un client et d'un serveur RPC. Le "driver" contient le code des trois niveaux d'interface ainsi que les fonctions nécessaires à la simulation. Les écritures distantes sont réalisées via le client et le serveur RPC. Le client prend un à un les descripteurs contenus dans la LPE. Il effectue une requête de lecture à l'adresse locale indiquée, puis utilise un appel au serveur distant pour réaliser l'écriture. Le serveur distant reçoit l'adresse physique à laquelle écrire ainsi que les données, il effectue alors l'écriture physique.

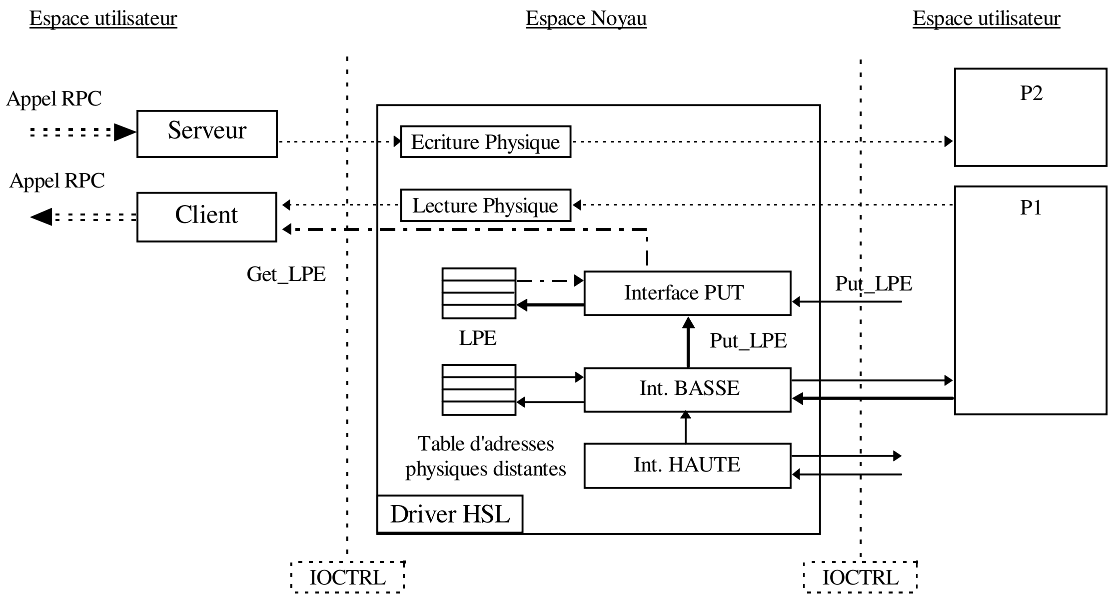

Figure 5. Architecture du "driver HSL", et du client/serveur RPC

Chapitre 4
Le *Swap* et la Pagination

4.1 *Swapping*

Un processus doit être en mémoire pour être exécuté. Cependant, un processus peut être transféré temporairement de la mémoire principale à une mémoire auxiliaire et être ensuite ramené en mémoire pour continuer son exécution. Par exemple, supposez un environnement multiprogrammé avec un algorithme de *scheduling* de l'UC *round-robin*. Quand une tranche de temps expire, le gestionnaire de la mémoire commencera à transférer en mémoire auxiliaire (*swap out*) le processus qui vient de finir et à ramener (*swap in*) un autre processus à l'espace mémoire libéré (voir Figure 6). En attendant, le *scheduleur* de l'UC allouera une tranche de temps à d'autres processus en mémoire. Quand un processus finit sa tranche de temps, il est échangé avec un autre processus. Dans le cas idéal, le gestionnaire de la mémoire peut transférer des processus suffisamment vite pour qu'il ait toujours des processus en mémoire, prêts à être exécutés quand le *scheduleur* de l'UC désire *scheduler* l'UC à nouveau.

Le *swapping* requiert de la mémoire auxiliaire qui est généralement un disque rapide. Il doit être suffisamment grand pour pouvoir ranger les copies de toutes les images mémoires de tous les utilisateurs et il doit fournir l'accès direct à ces images mémoire. Le système assure la maintenance d'une fille d'attente des processus prêts constituée de tous les processus dont les images mémoire se trouvent dans la mémoire auxiliaire ou dans la mémoire principal et qui sont prêts à être exécutés. Chaque fois que le *scheduleur* de l'UC décide d'exécuter un processus, il appelle le *dispatcher*. Celui-ci vérifie si le processus suivant dans la file d'attente est en mémoire. Si ce n'est pas le cas et qu'il n'existe pas de zone de la mémoire libre, le *dispatcher* transfère en mémoire auxiliaire un processus qui est en mémoire principale et charge le processus désiré. Ensuite, il recharge les registres de façon normale et transfère le contrôle au processus sélectionné.

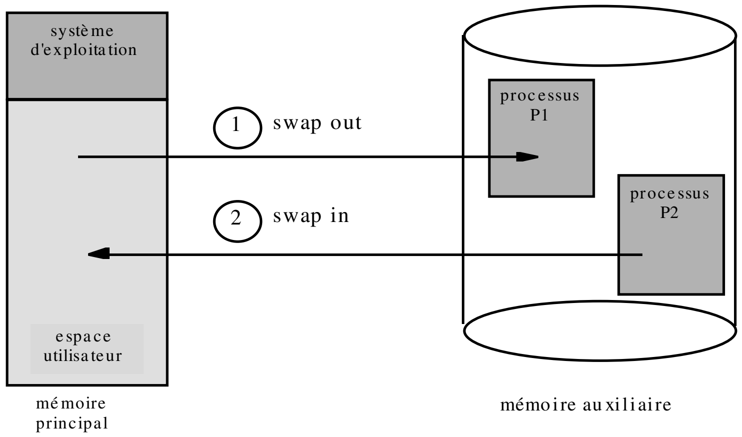

Figure 6. *Swapping* de deux processus en utilisant un disque comme mémoire.

Il est clair que le temps de commutation de contexte dans un système de *swapping* est assez élevé. Pour une utilisation efficace de l'UC, nous désirons que notre temps d'exécution pour chaque processus soit relativement long par rapport au temps de *swapping* Il existe d'autres contraintes pour le *swapping*. Si nous désirons transférer un processus, nous devons être sûrs qu'il est complètement inactif. Les entrées/sorties en cours sont d'un intérêt particulier. Cependant, si les entrées/sorties accèdent d'une manière asynchrone à la mémoire utilisateur pour utiliser les tampons d'entrée/sortie, le processus ne peut pas être transféré. Les deux solutions principales à ce problème sont: ne jamais transférer un processus avec des entrées/sorties en cours ou exécuter les opérations d'entrée/sortie que dans les tampons du système d'exploitation. Les transferts entre le système d'exploitation et la mémoire du processus ne se produisent donc que lorsque le processus est chargé en mémoire principale.

Il vaut mieux expliquer la supposition que le *swapping* requiert peu de positionnements de têtes. Généralement, on alloue une portion séparée du disque pour l'espace de *swap*, indépendamment du système de fichiers, de sorte que son utilisation soit la plus rapide possible.

Généralement, le *swapping* standard est utilisé dans peu de systèmes. Il requiert trop de temps de *swapping* et fournit un temps d'exécution trop court pour constituer une gestion de mémoire raisonnable. Cependant, on trouve sur plusieurs systèmes des versions modifiées du *swapping*.

4.2 Pagination

Un des problèmes du *swapping* est que l'unité de transfère est le processus, ce qui limite la flexibilité dans l'utilisation de la mémoire. Un système de pagination à la demande est semblable à un système de pagination avec *swapping*. Les processus résident en mémoire auxiliaire (laquelle est habituellement un disque). Lorsque nous désirons exécuter un processus, nous le transférons à la mémoire principale (*swap in*). Cependant, plutôt que de transférer le processus en entier dans la mémoire, nous utilisons une routine de *swap* paresseuse. Celle-ci ne transfère une page en mémoire que si l'on a besoin.

Quand un processus doit être transféré en mémoire, la routine de pagination devine quelles pages seront utilisées avant que le processus soit mis en mémoire auxiliaire de nouveau. Au lieu de transférer en mémoire un processus complet, la routine de pagination ramène seulement les pages qui sont nécessaires. Ainsi, il évite que l'on lise en mémoire des pages qui ne seront jamais employées, en réduisant le temps de *swap* et la quantité de mémoire dont on a besoin.

Avec ce schéma, nous devons compter avec un support matériel qui nous permette de distinguer entre les pages qui sont en mémoire, et celles qui sont sur disque. On peut utiliser pour ce propos un bit qui prendrait la valeur *valide-invalide*. Quand ce bit est fixé à "valide", cette valeur indique que la page associée est légale et en mémoire. Si le bit est fixé à "invalide", cette valeur indique que, soit la page n'est pas valide (c'est-à-dire, pas dans l'espace adresse logique du processus), soit elle est valide mais elle est sur disque. L'entrée de la table de pages pour une page qui n'est pas en mémoire est simplement marquée invalide ou contient l'adresse de la page sur disque.

L'accès à une page marquée invalide provoque un déroutement de *défaut de page*. Le matériel de pagination, en traduisant l'adresse par l'intermédiaire de la table de pages, remarquera que le bit est fixé à invalide, ce qui provoque un déroutement vers le système d'exploitation. La procédure permettant de manipuler ce défaut de page est simple:

- Nous examinons une table interne de ce processus, afin de déterminer si la référence était un accès mémoire valide ou invalide.

- Si la référence était invalide, nous terminons le processus. Si elle était valide mais que nous n'avons pas encore ramené en mémoire cette page, nous la chargeons.

- Nous trouvons un cadre de page libre.

- Nous *schédulons* une opération disque pour lire la page désirée dans le cadre de page récemment alloué.

- Quand on a fini la lecture du disque, nous modifions la table interne maintenue avec le processus et la table de pages afin d'indiquer que la page est maintenant en mémoire.

- Nous redémarrons l'instruction interrompue par le déroutement d'adresse illégale. Le processus peut accéder à la page comme si elle avait toujours été en mémoire.

Le matériel permettant de supporter la pagination est le même que celui utilisé pour le *swapping:*

- **Table de pages:** cette table possède la capacité de marquer une entrée invalide par l'intermédiaire d'un bit valide-invalide ou une valeur spéciale des bits de protection.

- **Mémoire secondaire**: cette mémoire contient les pages qui ne sont pas en mémoire principale. La mémoire auxiliaire est habituellement un disque rapide. Il est connu comme le dispositif de *swap* et la section du disque utilisée à ces fins est connue comme l'espace de *swap* ou la mémoire auxiliaire.

Il faut imposer quelques contraintes architecturales supplémentaires. La possibilité de redémarrer n'importe quelle instruction après un défaut de page est un thème crucial. Dans la plupart des cas, ce besoin est facile à satisfaire. Un défaut de page peut se produire dans toute référence mémoire. S'il se produit pendant l'extraction de l'instruction, nous pouvons redémarrer en effectuant à nouveau l'extraction. Si le défaut de page se produit pendant l'extraction d'un opérande, nous devons extraire et décoder à nouveau l'instruction et ensuite extraire l'opérande.

Redémarrer n'importe quelle instruction peut devenir difficile. La difficulté majeure survient quand une instruction peut modifier plusieurs emplacements différents. Si un bloc est à cheval sur une limite de page, il pourrait se produire un défaut de page après déplacement partiel. De plus, si les blocs source et destinataire se chevauchent, il se peut que l'on ait modifié le bloc source, auquel cas nous ne pouvons pas redémarrer simplement l'instruction.

On peut résoudre ce problème de deux manières différentes. Dans l'une des solutions, le microcode calcule et essaye d'accéder aux deux extrémités de deux blocs. S'il va se produire un défaut de page, il se produira à ce moment-là, avant que l'on ait modifié un emplacement quelconque. L'autre solution utilise des registres temporaires pour garder les valeurs des emplacements écrasés. S'il se produit un défaut de page, toutes les anciennes valeurs sont réécrites dans la mémoire avant que l'on effectue le déroutement.

Un problème architectural semblable survient dans les machines utilisant des modes d'adressage spéciaux, y compris les modes "d'auto-décrémentation" et d'"auto-incrémentation". Ces modes d'adressage emploient un registre comme pointeur et réduisent ou augmentent automatiquement le registre. "L'auto-décrémentation" réduit automatiquement le registre avant d'utiliser son contenu comme l'adresse opérande; "l'auto-incrémentation" augmente automatiquement le registre après l'utilisation de son contenu comme l'adresse opérande. Une solution consiste à créer un nouveau registre d'état spécial pour enregistrer le numéro de registre et la quantité modifiée pour tout registre changé pendant l'exécution d'une instruction. Ce registre d'état permet au système d'exploitation de "défaire" les effets d'une instruction partiellement exécutée lors d'un défaut de page.

Ceux-ci ne sont pas du tout les seuls problèmes architecturaux résultant de l'ajout de la pagination à une architecture existante afin d'obtenir une pagination à la demande, mais ils illustrent certaines des difficultés. La pagination est ajoutée entre l'UC et la mémoire dans un système informatique. Elle devrait être complètement transparente pour le processus utilisateur. Ainsi, les gens supposent souvent que la pagination pourrait être ajoutée à n'importe quel système. Bien que cette supposition soit vraie pour un environnement de pagination qui n'est pas à la demande, où un défaut de page représente une erreur fatale, ceci n'est pas vrai dans le cas où un défaut de page signifie seulement qu'une page supplémentaire doit être ramenée en mémoire et le processus redémarré.

4.2.1 PERFOMANCE DE LA PAGINATION A LA DEMANDE

La pagination à la demande peut affecter de façon significative la performance d'un système informatique. Afin de calculer le temps effectif d'accès, nous devons savoir combien de temps il faut pour traiter un défaut de page. Un défaut de page provoque la séquence suivante:

1. Effectuer un déroutement vers le système d'exploitation.

2. Sauvegarder les registres utilisateur et l'état du processus.

3. Déterminer que l'interruption a été provoquée par un défaut de page.

4. Vérifier que la référence à la page était légale et déterminer l'emplacement de celle-ci dans le disque.

5. Effectuer une lecture du disque vers un cadre de page libre:

   - Attendre dans la file d'attente de ce périphérique jusqu'à ce que la lecture soit satisfaite.

   - Attendre le positionnement du périphérique et/ou le temps de latence.

   - Commencer le transfert de la page vers un cadre de page libre.

6. Pendant l'attente, allouer l'UC à un autre utilisateur.

7. Interruption du disque (entrée/sortie terminée).

8. Sauvegarder les registres et l'état du processus pour l'autre utilisateur (si le pas 6 a été exécuté).

9. Déterminer que l'interruption venait du disque.

10. Corriger la table de pages et les autres tables afin de montrer que la page désirée se trouve maintenant en mémoire.

11. Attendre que l'UC soit allouée à nouveau à ce processus.

12. Restaurer les registres utilisateur, l'état du processus et la nouvelle table de pages, reprendre ensuite l'instruction interrompue.

Si nous prenons un temps moyen de traitement du défaut de page de 25 millisecondes et un temps d'accès à la mémoire de 100 nanosecondes, le temps effectif d'accès en nanosecondes est:

$$\text{temps effectif d'accès} = (1-p) \times (100) + p \times (25 \text{ millisecondes})$$

$$= (1-p) \times 100 + p \times 25\,000\,000$$

$$= 100 + 24\,999\,900 \times p$$

Nous voyons que le temps effectif d'accès est directement proportionnel au taux de défaut de page. Si un accès sur 1000 provoque un défaut de page, le temps effectif d'accès est de 25 microsecondes. L'ordinateur sera ralenti d'un facteur de 250 à cause de la pagination à la demande. Il est important de maintenir le taux de défaut de page bas dans un système de pagination à la demande. Sinon, le temps effectif d'accès augmente, en ralentissant l'exécution des processus dramatiquement.

Un aspect supplémentaire de la pagination à la demande est la manipulation et l'utilisation de l'espace de *swap*. Les entrées/sorties disque pour l'espace *swap* sont généralement plus rapides que celles pour le système de fichiers. Elles sont plus rapides parce que l'espace de *swap* est alloué dans des blocs beaucoup plus grands et on n'emploie pas de consultations de fichiers ni de méthodes d'allocation indirecte. Il est donc possible pour le système d'obtenir une capacité de traitement meilleure, en copiant une image fichier entière dans l'espace de *swap* quand le processus démarre et effectuer ensuite la pagination à la demande à partir de l'espace de *swap*. Les systèmes possédant un espace de *swap* limité peuvent employer un schéma différent quand on utilise les fichiers binaires. Les pages demandées pour ces fichiers sont ramenées directement à partir du système de fichiers. Cependant, quand nous requérons le remplacement de pages, celles-ci peuvent simplement être écrasées (parce qu'elles ne sont jamais modifiées) et lues à nouveau à partir du système de fichier s'il le faut. Il existe une autre possibilité: demander initialement les pages à partir du système de fichier; mais écrire les pages dans l'espace de *swap* quand elles sont remplacées. Cette approche assure que seulement les pages nécessaires sont lues à partir du système de fichiers, mais toute la pagination successive est effectuée à partir de l'espace de *swap*. Cette méthode semble être un bon compromis; on l'emploie dans le BSD UNIX.

4.2.2 REMPLACEMENT DE PAGES

Le remplacement de pages se fonde sur l'approche suivante. Si aucun cadre de page n'est libre, nous en trouvons un qui n'est pas actuellement utilisé et nous le libérons. Nous pouvons libérer un cadre de page en écrivant ses contenus dans l'espace de *swap* et en modifiant la table de pages (et toutes les autres tables) afin d'indiquer que la page n'est plus en mémoire. On peut utiliser maintenant le cadre de page libre pour ranger la page manquante au processus. La routine de traitement de défauts de page tient compte alors du remplacement de pages:

1. Trouver l'emplacement de la page désirée sur disque.

2. Trouver un cadre de page libre:
   a. S'il existe un cadre de page libre, l'utiliser.
   b. Sinon, utiliser un algorithme de remplacement de pages pour sélectionner un cadre de page *victime.*
   c. Enregistrer la page victime dans le disque; modifier les tables de pages et de cadres de pages en conséquence.

3. Lire la page désirée dans le cadre de page (récemment) libéré; modifier les tables de pages et de cadres de pages.

4. Redémarrer le processus utilisateur.

S'il n'existe pas de cadre de page libre, on a besoin de deux transferts de pages (un à partir de la mémoire et l'autre vers la mémoire). Cette situation double en réalité le temps de traitement du défaut de page et par conséquence augmentera le temps effectif d'accès.

On peut réduire cette surcharge en utilisant un bit de modification (sale). Le bit de modification d'une page est mis à 1 par le matériel chaque fois qu'un mot ou un octet est enregistré dans la page, ce qui indique qu'elle a été modifiée. Quand nous choisissons un page devant être remplacée, nous examinons son bit de modification. Si le bit est à 1, nous savons que la page a été modifiée depuis qu'elle a été ramenée du disque. Dans ce cas, nous devons enregistrer cette page sur disque. Cependant, si le bit de modification est égal à 0, la page n'a pas été modifiée depuis qu'on l'a lue dans la mémoire. Par conséquent, si la copie de la page sur disque n'a pas été écrasée, nous pouvons éviter de sauvegarder la page mémoire sur le disque; elle est déjà là.

Le remplacement de pages est essentiel pour la pagination à la demande. Il complète la séparation entre la mémoire physique et la mémoire logique. Avec ce mécanisme, on peut fournir aux programmeurs une très grande mémoire virtuelle sur une mémoire physique plus petite. Avec la pagination à la demande, la taille de l'espace adresse logique n'est plus limité par la mémoire physique.

Nous devons résoudre deux problèmes principaux pour implémenter la pagination à la demande: nous devons développer un algorithme d'allocation de cadres de pages et un algorithme de remplacement de pages. S'il existe plusieurs processus dans la mémoire, nous devons décider combien de cadres de pages allouer à chaque processus. De plus, quand on requiert un remplacement de page, il faut sélectionner les cadres de pages à remplacer. La conception d'algorithmes corrects pour résoudre ces problèmes est une tâche importante, car les entrées/sorties disque coûtent cher. Même des petites améliorations aux méthodes de la pagination à la demande conduisent à des gains considérables dans la performance du système.

4.2.3 ALGORITHMES DE REMPLACEMENT DE PAGES

Il existe plusieurs algorithmes différents de remplacement de pages. Chaque système d'exploitation possède probablement son propre schéma de remplacement unique.

4.2.3.1 Algorithme FIFO

L'algorithme de remplacement le plus simple est un algorithme *premier entré, premier sorti* (*"first-infirst-out"*). Un algorithme de remplacement FIFO associe à chaque page le moment où l'on a ramené cette page en mémoire. Quand on doit remplacer une page, c'est la plus ancienne que l'on sélectionne. Remarquez qu'il n'est pas strictement nécessaire d'enregistrer le moment dans lequel une page est chargée en mémoire. Nous pouvons créer une file d'attente FIFO pour ranger toutes les pages se trouvant en mémoire. Nous remplaçons donc la page qui se trouve en tête de la file d'attente. Quand on ramène une page en mémoire, nous l'insérons en queue de la file d'attente.

L'algorithme de remplacement FIFO est facile à comprendre et à programmer. Cependant, sa performance n'est pas toujours bonne. La page remplacée peut être un module d'initialisation pas utilisé depuis longtemps et qui n'est plus nécessaire. En revanche, il pourrait contenir une variable très employée, initialisée tôt et en constante utilisation.

L'utilisation de l'algorithme de remplacement FIFO peut occasionner l'anomalie de Belady. Cette anomalie consiste en avoir un nombre supérieur de défaut de pages quand on augmente le nombre de cadres de pages. Celle-ci reflète le fait que le taux de défaut de pages grandi au fur et à mesure que le nombre de cadres de pages alloués augmente.

4.2.3.2 Algorithme optimal

La recherche d'un algorithme de remplacement de page optimal est une conséquence de la découverte de l'anomalie de Belady. Il existe un algorithme de remplacement de pages optimal et on l'appelle OPT ou MIN. Il consiste simplement à: remplacer la page qui mettra le plus de temps à être de nouveau utilisée.

L'utilisation de cet algorithme de remplacement de pages garantit le taux de défaut de pages le plus bas possible pour un nombre fixe de cadres de pages. Malheureusement, l'algorithme de remplacement de pages est difficile à implémenter, car il requiert une connaissance future. On utilise donc l'algorithme optimal essentiellement pour effectuer des études comparatives.

4.2.3.3 Algorithme LRU

Si l'algorithme optimal n'est pas faisable, une approximation est peut être possible. Si nous utilisons le passé récent comme une approximation du futur proche, nous remplacerons donc la page qui n'a pas été utilisée pendant la plus longue période de temps. Cette approche est l'algorithme du *moins récemment utilisé* ("*last recently used"*).

Le remplacement LRU associe à chaque page le moment de sa dernière utilisation. Quand on doit remplacer une page, LRU sélectionne la page qui n'a pas été utilisée pendant la plus longue période de temps. Cette stratégie est l'algorithme de remplacement de pages optimal regardant le temps passé plutôt que celui à venir.

La politique LRU est souvent employée comme algorithme de remplacement de pages et il est considéré comme très bon. Le problème principal est de savoir comment implémenter le remplacement LRU. Un algorithme de remplacement LRU peut demander une forte assistance matérielle.

Peu de systèmes fournissent un support matériel suffisant pour implémenter un véritable remplacement de pages LRU. Cependant, beaucoup de systèmes aident avec un bit de référence. Le bit de référence d'une page est mis à 1, par le matériel, chaque fois que la page est référencée. Nous savons donc quelles pages ont été employées mais nous ne connaissons pas l'ordre d'utilisation. Cette information sur l'ordre partiel mène à beaucoup d'algorithmes se rapprochant du remplacement LRU.

4.2.4 ALLOCATION DE CADRES DE PAGES

Comment allouons-nous la quantité fixe de mémoire libre entre les divers processus? Dans la pagination à la demande pure, on mettra initialement tous les cadres de pages dans la liste de cadres de pages libres. Quand un processus utilisateur débutera son exécution, il produira une séquence de défauts de pages. Les premiers défauts de page obtiendront des cadres de pages libres à partir de la liste. Quand la liste de cadres de pages libres sera vide, on utilisera un algorithme de remplacement de pages. Quand le processus se termine, les cadres de pages seront placés à nouveau dans la liste de cadres de pages libres.

Il existe plusieurs variantes de cette stratégie si simple. Nous pouvons demander que le système d'exploitation alloue tous ses *tampons* et espace de tables à partir de la liste de cadres de pages libres. Quand le système d'exploitation n'emploie pas cet espace, on peut l'utiliser pour supporter la pagination de l'utilisateur. Nous pourrions essayer de réserver en permanence quelques cadres de pages libres dans la liste. Ainsi, quand il se produit un défaut de page, il existe un cadre de page disponible pour y charger la page. Quand le transfert de la page a lieu, on peut sélectionner un remplacement qui est ensuite écrit sur disque quand le processus utilisateur continue son exécution.

Un problème différent survient quand on combine la pagination à la demande et la multiprogrammation.

4.2.4.1 Nombre minimum de cadres de pages

Il existe plusieurs contraintes dans nos stratégies pour l'allocation de cadres de pages. Nous ne pouvons allouer plus que le nombre total de cadres de pages disponibles. Il existe également un nombre minimum de cadres de pages pouvant être allouées. Evidemment, au fur et à mesure que le nombre de cadres de pages allouées à chaque processus diminue, le taux de défaut de pages augmente, ralentissant ainsi l'exécution du processus.

En plus des propriétés de performance indésirables résultant du fait d'allouer peu de cadres de pages, il existe un nombre minimum de cadres de pages devant être alloués. Ce nombre minimum est défini par l'architecture du jeu d'instructions. Quand il se produit un défaut de page avant que l'exécution d'une instruction ne soit finie, on doit redémarrer l'instruction. Nous devons donc avoir suffisamment de cadres de pages pour ranger toutes les pages différentes qu'une instruction unique peut référencer.

Le nombre minimum de cadres de pages est défini par l'architecture de l'ordinateur. Les pires cas se produisent quand les architectures permettent plusieurs niveaux d'indirection. En théorie, une instruction de chargement simple pourrait référencer une adresse indirecte qui référencerait une adresse indirecte (sur une autre page) référençant encore une adresse indirecte (toujours sur une autre page) et ainsi de suite, jusqu'à atteindre chaque page dans la mémoire virtuelle. Ainsi, dans le pire des cas, la mémoire virtuelle devrait se trouver dans sa totalité dans la mémoire physique. Afin de surmonter cette difficulté, nous devons imposer une limite aux niveaux d'indirection.

Le nombre minimum de cadres de pages par processus est défini par l'architecture, tandis que le nombre maximum est déterminé par la quantité de mémoire physique disponible. Entre les deux, il nous reste encore à effectuer des choix significatifs pour l'allocation de cadre de pages.

4.2.4.2 Algorithmes d'allocation

La manière la plus simple de fractionner m cadres de pages entre n processus est de donner à chacun des portions égales, m / n cadres de pages. Ce schéma est appelé allocation équitable. Il existe une autre possibilité: admettre que les divers processus demanderont des quantités différentes de mémoire. Pour résoudre ce problème, nous pouvons employer l'allocation proportionnelle. Nous allouons de la mémoire disponible à chaque processus selon sa taille. Ainsi, les processus partagent les cadres de pages disponibles selon leurs "besoins", plutôt que d'une manière égalitaire.

Dans les deux types d'allocation, un processus de haute priorité est traité de la même façon qu'un de basse priorité. Cependant, à cause de sa définition, on peut souhaiter donner au processus le plus prioritaire plus de mémoire. Une approche possible consiste à utiliser un schéma d'allocation proportionnelle, où la quantité de cadres de pages dépend, non de la taille relative des processus, mais plutôt de leurs priorités ou d'une combinaison de taille et de priorité.

4.2.4.3 Allocation globale versus allocation locale

Le remplacement de pages est un autre facteur important dans la manière d'allouer les cadres de pages aux divers processus. Avec plusieurs processus en concurrence pour des cadres de pages, nous pouvons classer les algorithmes de remplacement de pages en deux grandes catégories: remplacement global et remplacement local. Le remplacement global autorise un processus à sélectionner un cadre de page de remplacement à partir de l'ensemble de tous les cadres de pages, même si ce cadre de page est à ce moment-là alloué à un autre processus; un processus peut prendre le cadre de page d'un autre. Le remplacement local demande que chaque processus choisisse un cadre à partir de son propre ensemble de cadres de pages alloués.

L'un des problèmes posés par l'algorithme de remplacement global, c'est qu'un processus ne peut pas contrôler son propre taux de défauts de page. L'ensemble de pages en mémoire pour un processus ne dépend pas seulement du comportement de la pagination pour ce processus, mais aussi du comportement de la pagination pour les autres processus. Par conséquent, le même processus peut s'exécuter très différemment à cause de circonstances complètement extérieures. De son coté, le remplacement local pourrait gêner un processus en ne lui rendant pas disponible d'autres pages de la mémoire moins utilisées. Ainsi, le remplacement global provoque en général une capacité de traitement plus importante, c'est donc la méthode la plus courante.

4.2.5 ECROULEMENT

Si la quantité de cadres de pages alloués à un processus de basse priorité descend en dessous du nombre minimum requis par l'architecture de l'ordinateur, nous devons arrêter l'exécution de ce processus. Nous devrions ensuite transférer les pages restant sur disque, libérant ainsi tous les cadres de pages qui lui ont été alloués.

Bien qu'il soit techniquement possible de réduire le nombre de cadres de pages alloués au minimum, il existe un nombre de pages en utilisation active. Si le processus ne possède pas ce nombre de cadres de pages, il produira très rapidement des défauts de page. A ce moment-là, il doit remplacer une page nécessaire tout de suite après. Il produira donc très rapidement un défaut de page à nouveau et ainsi de suite. Le processus continuera à produire des défauts de page, en remplaçant des pages dont il aura besoin tout de suite après.

Cette haute activité de pagination est appelée écroulement. Un processus s'écroule s'il passe plus de temps à paginer qu'à s'exécuter.

Pour prévenir l'écroulement, nous devons fournir à un processus autant de cadres de pages qu'il en aura besoin. La stratégie de l'ensemble de travail commence par regarder combien de cadres de pages un processus utilise en réalité. Cette approche définit le modèle de localisation de l'exécution du processus.

Le modèle de localisation dit que, lorsqu'un processus s'exécute, il se déplace de localisation en localisation. Une localisation est un ensemble de pages utilisées activement ensemble. Un programme est généralement constitué de plusieurs localisations différentes qui peuvent se chevaucher.

Par exemple, quand on appelle une sous-routine, elle définit une nouvelle localisation. Dans celle-ci, les références mémoire sont effectuées aux instructions de la sous-routine, ses variables locales et un sous-ensemble des variables globales. Quand on quitte la sous-routine, le processus part de cette localisation, puisque les variables locales et les instructions de la sous-routine ne sont plus en utilisation active. Ainsi, nous observons que les localisations se définissent par la structure du programme et ses structures de données.

Supposez que nous allouons une quantité suffisante de cadres de pages à un processus pour ranger sa localisation courante. Il effectuera des défauts de pages dans cette localisation jusqu'à ce qu'elles se trouvent toutes en mémoire; il ne réalisera donc plus de défaut de page jusqu'à son changement de localisation.

Si nous allouons moins de cadres de pages que la taille de la localisation courante, le processus s'écroulera, car il ne peut pas maintenir en mémoire toutes les pages qu'il utilise activement.

Le modèle de l'ensemble de travail a du succès et la connaissance de l'ensemble de travail peut être utile pour la prépagination, mais cela semble une manière maladroite de contrôler l'écroulement. La stratégie de la fréquence de défauts de pages adopte une approche plus directe.

Le problème spécifique consiste à prévenir l'écroulement. Celui-ci possède un taux élevé de défauts de pages. Ainsi, nous souhaitons contrôler le taux de défauts de page. Si le taux réel de défauts de pages dépasse la limite supérieure, nous allouons au processus un autre cadre de page; s'il est en dessous de la limite inférieure, nous supprimons un cadre de page de ce processus. Ainsi, nous pouvons directement mesurer et contrôler le taux de défauts de pages afin de prévenir l'écroulement.

Comme pour la stratégie de l'ensemble de travail, nous risquons de devoir arrêter un processus. Si le taux de défauts de pages augmente et que nous ne disposons pas de cadres de pages libres, nous devons sélectionner un processus à arrêter. On distribue donc les cadres libérés entre les processus ayant des taux élevés de défauts de pages.

4.2.6 PREPAGINATION

Un système de pagination à la demande pure présente une caractéristique évidente: le grand nombre de défauts de pages des produisant quand un processus démarre. Ceci peut aussi survenir quand un processus ayant été transféré sur disque est redémarré, toutes ses pages se trouvent sur le disque et on doit charger en mémoire chacune d'entre elles avec son propre défaut de page. La prépagination est une tentative de prévenir ce haut niveau initial de pagination. La stratégie consiste à ramener en mémoire en une seule fois toutes les pages nécessaires.

Par exemple, dans un système où l'on emploie le modèle de l'ensemble de travail, nous maintenons pour chaque processus une liste des pages dans son espace de travail. Si nous désirons arrêter un processus, nous enregistrons l'ensemble du travail pour ce processus. Quand on doit reprendre le processus, nous ramenons automatiquement en mémoire son ensemble de travail en entier avant de redémarrer le processus.

La prépagination peut être avantageuse dans certains cas. La question à se poser est simplement de savoir si le coût de la prépagination est inférieur au coût du traitement des défauts de pages correspondants.

4.2.7 VERROUILLAGE DES ENTREES/SORTIES

Quand on emploie la pagination à la demande, il faut quelques fois permettre à quelques pages d'être verrouillées en mémoire. Une telle situation se produit quand les entrées/sorties se produisent à partir de ou vers la mémoire (virtuelle) utilisateur.

Nous devons être sûrs qu'un processus en attente d'une entrée/sortie, n'aura pas sa page, contenant le *buffer* mémoire, remplacé par une page d'un autre processus (dans le cas d'utilisation d'un algorithme de remplacement global). Si cela arrive, un peu plus tard, quand la requête d'entrée/sortie se place en tête de la file d'attente du périphérique, l'entrée/sortie se produit dans l'adresse spécifiée. Cependant, ce cadre de page est maintenant utilisé pour une page différente appartenant à un autre processus.

Il existe deux solutions communes à ce problème. L'une consiste à ne jamais exécuter d'entrées/sorties dans la mémoire utilisateur. Au lieu de cela, on copie toujours les données entre la mémoire système et la mémoire utilisateur. Cette copie supplémentaire peut produire une surcharge élevée de façon inacceptable.

Une autre solution consiste à permettre de verrouiller des pages en mémoire. Un bit verrou est associé à chaque cadre de page. Si le cadre de page est verrouillé, on ne peut pas le sélectionner pour le remplacer. Avec cette approche, pour écrire un bloc sur la bande, nous verrouillons en mémoire les pages contenant le bloc. Le système peut donc continuer comme d'habitude. Les pages verrouillées ne peuvent pas être remplacées. Quand l'entrée/sortie est terminée, les pages sont déverrouillées.

4.3 Le *swap* distant

4.3.1 LE *SWAP* SUR DISQUE DISTANT

Initialement, la pagination à distance a été utilisée pour des machines ne disposant pas de disque local (*diskless)*. A cette époque le prix du disque était prohibitif, c'est pourquoi la préoccupation première était de diminuer le nombre de disque. Toute page est envoyée par le réseau vers un serveur de pages [Minnich, 1993]. Le temps de *swap* est alors égal au temps "accès réseau + accès disque distant". Un tel serveur est facilement réalisable via NFS [Anderson, Dahlin, Neefe, Patterson, Roselli; Wang, 1995]. Le *swap* sur disque distant améliore les performances d'un *swap* local dans le cas où le serveur distant est équipé de disques plus rapides que les disques locaux (par exemple, l'architecture RAID). Cependant ces disques peuvent difficilement être mis à la disposition de nombreux clients sans que le serveur devienne un goulot d'étranglement. En outre, la diminution du temps de pagination due aux disques plus rapides peut être absorbée par le temps de latence du réseau. Certaines méthodes de pagination procèdent par anticipation (*lookahead)*, supprimant alors le temps de latence pour les pages anticipées.

4.3.2 LE *SWAP* EN MEMOIRE DISTANTE

Le *swap* se fait uniquement dans la mémoire de serveurs distants. Ce mécanisme supprime le temps de latence induit par l'accès à un disque distant et augmente la disponibilité des serveurs auprès des clients. Cela suppose, une politique de sélection de machines "serveur de mémoire" et surtout, que la somme des mémoires des mémoires des serveurs soit suffisante. Dans le cas contraire, il est possible de voir la mémoire distante comme un premier niveau de stockage avant le *swap* sur disque. Cette méthode est souvent utilisée dans le cas de serveurs de fichiers distribués qui utilisent leur mémoire comme cache pour les disques. Les performances sont alors dépendantes du taux de la présence dans le cache (*hit)*, qui est de moins de 10% [Backer, Hartman, Kupfer, Shirriff, Ousterhout, 1991]. Ce faible taux est dû au fait que ces systèmes utilisent un même cache pour les fichiers et la pagination. Les politiques de remplacement de pages désignent les pages les moins utilisées. Les pages de la mémoire virtuelle sont donc évincées en priorité pour laisser place à celle du système de fichier.

4.3.2.1 Limites et performances

G.Bernard et S.Hamma [Bernard, Hamma, 1994], simulent une pagination mémoire à distance. L'idée de la pagination à distance est d'utiliser la mémoire oisive des autres machines. Ils examinent différents critères (taille des processus, taux de défaut de pages, taux d'utilisation du cpu, espace mémoire alloué au *swap* distant) caractérisant les politiques de *swap*, afin de découvrir l'impact que la pagination distante a sur les processus locaux et vice-versa. Ils observent que: un processus local utilisant intensivement la mémoire et déjà sujet au *swap* (sur disque) n'est pas trop pénalisé (pénalité de 7%) par la pagination d'un processus distant dans son espace mémoire. L'effet sur un processus utilisant intensivement le processeur est d'autant plus important que le processeur est réquisitionné par le système de pagination: pour 40 pf/s en lecture ou 17 pf/s en écriture la pénalité est de 10% elle croit rapidement avec le faute de pages en écriture (jusqu'à 100% pour 100 pf/s). L'effet des processus locaux, utilisant intensivement la mémoire, sur la pagination à distance est plus important. En effet dès 30pf/s on perd le bénéfice de la pagination à distance, de plus le nombre de fautes de pages que le serveur est capable de traiter diminue très rapidement. Au contraire le taux d'occupation du processeur par les processus locaux a une influence négligeable sur les temps de réponse des processus clients. Cette simulation (réalisée sur un ensemble de stations de travail SUN, sous SunOS.4.1.2, interconnectés par un réseau Ethernet à 10Mbit/s), indique qu'il est possible de concilier les besoins des utilisateurs locaux et ceux du *swap* réparti.

Remote Memory Pager [Markatos, Dramitinos, Papachritos, 1995] réalise un mécanisme de pagination à distance, en utilisant la mémoire oisive des autres stations du réseau. Toutes les stations sont des DEC-Alpha 3000, connectées par un réseau Ethernet 10 Base T. Le disque local pour la pagination est un DEC RZ55, ayant un débit de 10 Mbits/s et un temps d'accès de 16 ms. Les tests ont montré une amélioration significative: entre 39% et 112% d'accélération du temps d'exécution. Le *swap* représentait alors 62% du temps d'exécution. Le temps moyen de pagination est de 9,05 ms en mémoire distante et de 17 ms sur le disque local. Ces 9,05 ms se décomposent en 7,2 ms de temps de transfert sur le réseau et 1,85 ms de temps de latence logiciel. Augmenter le débit du réseau réduit donc le temps de pagination.

L'utilisation d'un réseau à haut débit est donc nécessaire, mais il est aussi important de réduire le temps de latence du système. Cela est possible au moyen d'un mécanisme d'écriture directe en mémoire distante (*remote write*), sans interruption des processeurs et sans recopie dans les tampons du système. Ce type de protocole prend les données à transférer directement dans l'espace utilisateur local et les place dans l'espace utilisateur distant. Un transfert se décompose en trois phases: l'utilisateur place dans la file des requêtes les adresses locales et distantes des données à transférer; le matériel vient lire directement ces données en mémoire; la machine cible reçoit ces données et les écrit à l'adresse destinataire. L'émetteur est responsable de la présence des données émises, celles-ci ne doivent pas être modifiées. Pour cela, il verrouille les pages avant d'en demander le transfert. Il est aussi responsable de la validité de l'adresse destinataire.

Notre plate-forme de test initiale est composée de 5 *Pentium* 150 Mhz interconnectés par un concentrateur Ethernet 100 Base T (100 Mbits/s). Le système retenu est une version publique, modifiable et documentée du système UNIX: le système FreeBSD. Ce système dispose d'un mécanisme de pagination en mode utilisateur, ce qui permet de limiter les modifications à apporter au noyau. Cette plate-forme constitue un modèle de ce que l'on peut trouver dans de nombreux réseaux d'entreprises. Nous y avons ajouté un emulateur du protocole DDSBP.

Par la suite nous nous placerons dans le cadre du projet Multi-PC du laboratoire MASI, ce projet vise à la conception d'une machine multi-ordinateurs. Elle est composée d'un grand nombre de processeurs standards (Pentium) reliés par un réseau haut débit. Ce réseau, le *High Speed Link* (HSL) [Lecler, Potter, Desbarbieux, Wajsbürt, Greiner, 1996], utilise le routeur *Rcube* développé au laboratoire MASI. Il a pour caractéristique un très grand débit (640 Moctets/s) et un temps de latence très faible. Il est associé à un protocole matériel de transfert très rapide du type *Direct Deposit Sender Based Protocol* (DDSBP). Ce protocole permet à l'émetteur d'écrire directement dans la mémoire distante. Ce réseau assure des transferts de messages en quelques dizaines de micro-secondes, à comparer au 10 ms d'un réseau Ethernet 10 Base T (10 Mbits/s) standard.

Chapitre 5
MAÎS

5.1 CONCEPTION

La section 5.1.1 présente le support de réalisation du mécanisme de *swap* réparti. La section 5.1.2 introduit les principes de notre solution. Les sections 5.1.3 et 5.1.4 détaillent les mécanismes, politiques et stratégies. La section 5.1.5 donne un aperçu des solutions possibles dans le domaine de la tolérance aux fautes.

5.1.1 SUPPORT DE REALISATION

Notre plate-forme de test initiale est composée de 5 *Pentium* 150 Mhz interconnectés par un concentrateur Ethernet 100 Base T (100 Mbits/s). Le système retenu est une version publique, modifiable et documentée du système UNIX: le système FreeBSD. Ce système dispose d'un mécanisme de pagination en mode utilisateur, ce qui permet de limiter les modifications à apporter au noyau. Cette plate-forme constitue un modèle de ce que l'on peut trouver dans de nombreux réseaux d'entreprises. Nous y avons ajouté un émulateur du protocole d'écriture distante (nommé DDSBP: *Direct Deposit Sender Based Protocol*) [Potter, David, Fenyö, 1996].

Par la suite, nous nous placerons dans le cadre du projet Multi-PC du laboratoire MASI, qui vise à la conception d'une machine multi-ordinateurs. Elle est composée d'un grand nombre de processeurs standards (Pentium) reliés par un réseau haut débit. Ce réseau, le *High Speed Link* (HSL) [Lecler, Potter, Desbarbieux, Wajsbürt, Greiner, 1996], utilise le routeur *Rcube* développé au laboratoire MASI. Il a pour caractéristique un très grand débit (640 Moctets/s) et un temps de latence faible (< 10 µs). Il est associé à un protocole matériel de d'écriture direct en mémoire distance nommé *Direct Deposit Sender Based Protocol* (DDSBP). Ce réseau assure des transferts de messages en quelques dizaines de micro-secondes, à comparer au 10 milli-secondes d'un réseau Ethernet 10 Base T (10 Mbits/s) standard.

Le réseau HSL de la machine MPC est un réseau de type multi-étage. Toutes les machines ne sont donc pas à égale distance les unes par rapport aux autres, nous devons donc envisager la possibilité de redistribuer les pages. Le temps libre des serveurs et du réseau peut être employé à établir des statistiques d'utilisation des pages et une éventuelle utilisation de ces résultats pour leur chargement anticipé et leur bonne localisation.

5.1.2 PRINCIPES

La mise en oeuvre de ce *swap* réparti repose sur le modèle client / serveur. Les clients, souhaitant éviter de décharger des pages sur leur disque local, utilisent la mémoire centrale de serveurs de pages. Ces serveurs sont constitués des machines peu chargées du réseau. Nous prévoyons trois étapes de conception: (1) une unique machine cliente, le reste des machines étant des serveurs dédiés, (2) plusieurs machines clientes coopérantes pour exécuter plusieurs applications, (3) un mode symétrique où les machines peuvent devenir cliente ou serveur, ou les deux à la fois. Nous ne considérons pas le cas du partage de pages entre plusieurs processus, car il suppose un modèle de mémoire virtuelle partagée répartie, en dehors de la problématique du *swap* réparti.

(1) La première étape consiste en une configuration statique composée d'une seule station cliente. Cette configuration a pour avantage de bien isoler les facteurs qui influencent la performance. Sa mise en oeuvre repose sur des mécanismes simples. On répartit l'espace mémoire virtuel du client sur les différents serveurs. Un simple entrelacement des pages permet de distribuer la charge des serveurs, ceux-ci sont alors désignés par un modulo sur l'adresse virtuelle (voir Figure 7).

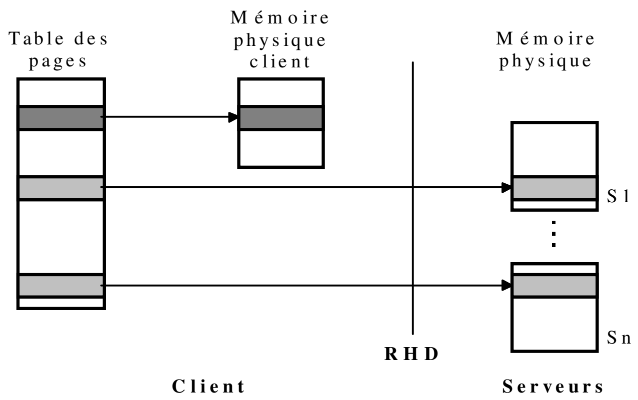

Figure 7. Pagination réparti statique à un client.

(2) Notre objectif est d'étendre ce travail à un modèle où plusieurs clients d'une même application s'exécutent sur un nombre restreint de stations (stations clientes). Les processus se trouvant sur ces stations nécessitent toujours plus de mémoire qu'il n'y en a physiquement. Ces clients utiliseront donc la mémoire des machines inutilisées du réseau (stations serveurs). Ce partitionnement est fait dynamiquement pendant l'exécution. Initialement, toutes les stations sont considérées comme serveurs. Par la suite, un processus candidat au *swap* réparti se fait connaître du serveur de pages local, celui-ci passe alors en mode client. Si ce serveur avait des pages en sa possession il les redistribue entre tous les autres serveurs. Contrairement au modèle précédent, il est possible que la mémoire globale ne soit pas suffisante. Nous allons donc utiliser un deuxième niveau de *swap* en utilisant les disques locaux des clients ou ceux des serveurs de pages afin d'y placer les pages non récemment utilisées. L'espace de *swap* réparti peut être vu comme une hiérarchie de mémoires (voir Figure 8).

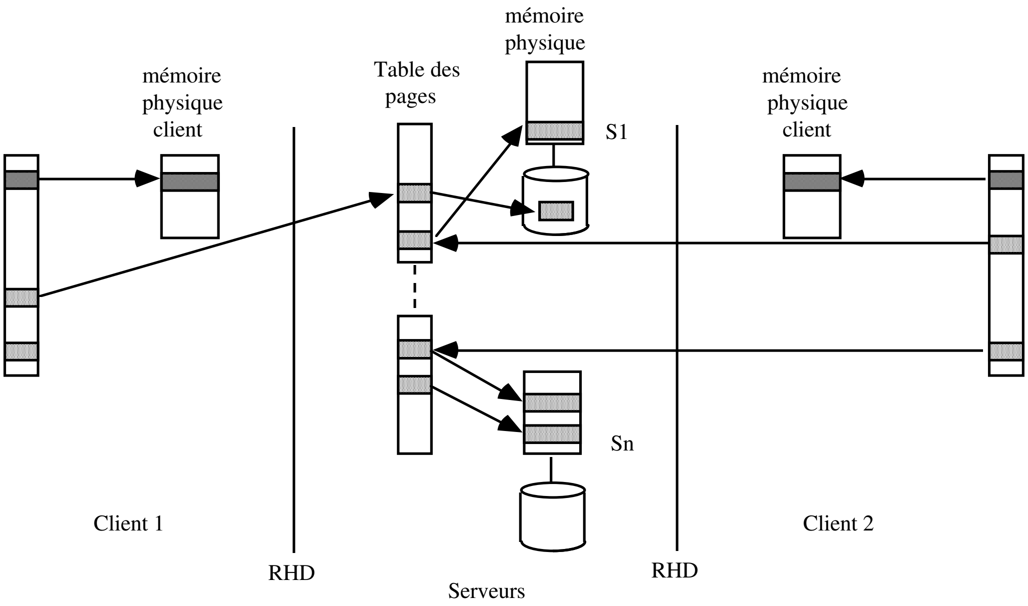

Figure 8. Pagination dynamique à deux clients.

(3) Dans le cas particulier de la machine MPC, toutes les machines sont équivalentes. Il est donc nécessaire de transformer les configurations précédentes en un modèle symétrique, où toutes les machines sont à la fois serveurs et clientes. Cela suppose un contrôle réparti dynamique du nombre de pages libres sur les serveurs.

Dans la suite, nous décrirons plus particulièrement le deuxième modèle.

5.1.3 MECANISMES

Différencier les processus est nécessaire afin de discerner ceux qui pourront participer au mécanisme de *swap* réparti. La solution choisie consiste à modifier la structure des processus (champs vers le périphérique destinataire) permettant ainsi de sélectionner entre le *swap* standard (réservé aux processus systèmes) et le *swap* réparti pour chaque processus en particulier.

La conception d'un *swap* réparti utilise les mêmes mécanismes de mémoire virtuelle que pour un *swap* local. Toutefois, il faut ajouter aux structures locales de contrôle des pages des informations sur leur localisation, à savoir, un bit de localisation (local/distant), et l'identité du serveur qui gère la page. Les serveurs de pages doivent pouvoir envoyer les pages aux processus clients mais aussi être capables de déplacer (migration) des pages dynamiquement. Pour cela, ils doivent connaître pour chaque page l'identité du processus (n° de processus + adresse de la machine) propriétaire.

Le mécanisme de *swap* réparti modifie le *swap* de FreeBSD pour utiliser le protocole DDSBP (voir section 5.1.1). Un client voulant écrire une page utilise l'adresse physique fournie par le serveur pour sauver la page à distance. Symétriquement, un défaut de page provoque l'envoi d'une requête au serveur qui la détient. Cette requête contient la référence de la page, ainsi que l'adresse d'un emplacement physique où placer localement la page manquante. Le transfert de plus d'une page (anticipation) est effectué en donnant au serveur la liste des pages et des emplacements physiques.

5.1.4 POLITIQUES & STRATEGIES

Pour réagir rapidement aux demandes de déchargement, les clients conservent un quota de pages (un*pool* contenant une liste d'adresses physiques) fourni par les machines identifiées comme serveurs. Après le transfert d'un grain de pages sur le serveur, celui-ci peut renouveler le *pool* du client en lui fournissant un nouveau quota. Les stratégies de *swap* doivent considérer la granularité des transferts, ainsi que le moment et les pages à évincer. Dans le *swap* réparti il faut en plus sélectionner une machine de destination.

Grain ? Il faut trouver un compromis entre l'optimisation de la bande passante du réseau (gros grain) et sa disponibilité (grain fin). Regrouper les pages lors des échanges entre clients et serveurs permet de diminuer le temps de latence des protocoles systèmes et réseaux proportionnellement au nombre de pages. De plus, le protocole DDSBP permet de transmettre plusieurs pages en générant une seule interruption sur le destinataire. En outre, le taux de localité (*working-set*) de l'application est un facteur essentiel à la définition de la taille du grain optimale. Un bon taux de localité permet d'augmenter la taille du grain. En revanche, l'envoi d'une requête suite à un défaut de page peut alors se trouver bloqué par l'envoi préventif d'un tel bloc de page.

Quand ? Afin de maintenir un ensemble de pages libres en mémoire et aussi de réduire le temps de latence au chargement, il est nécessaire de faire de l'évincement préventif. Un seuil calculé comme un pourcentage de la mémoire définit la limite au-delà de laquelle il faut activer ce mécanisme d'évincement préventif. Un seuil trop bas implique un gaspillage de la mémoire et à terme des va-et-vient inutiles; au contraire, un seuil trop haut risque de laisser le système bloqué en attente de pages disponibles. Par exemple, ce seuil représente 6,25% de la mémoire utilisateur sur

Système V Release 4 [Goodheart, Cox, 1994] et de 12,5% dans 4.3 BSD [Leffler, McKusik, Karels, Quarterman, 1990].

Qui ? Les algorithmes de choix d'une page à évincer ne changent pas pour un *swap* réparti. Les algorithmes généraux tel que: FIFO, NRU permettent de sélectionner la page, mais on évince (en écrivant les pages modifiées) tout le bloc dans lequel se trouve la page désignée (grain). L'utilisation du mécanisme de *swap* est restreint à l'ensemble des processus désignés explicitement. Enfin, nous avons choisi de ne jamais utiliser le mécanisme de *swap* réparti pour les processus systèmes (voir section 5.1.5).

Où ? Contrairement au *swap* local, la localisation d'un serveur est cruciale. Elle est résolue en partie par le mécanisme d'identification qui délimite l'ensemble des serveurs. Par la suite, les systèmes clients choisissent de façon circulaire le serveur sur lequel placer leurs pages. Chaque serveur décide après la fin d'un transfert de réallouer ou non un quota de pages en fonction de sa place mémoire disponible.

5.1.5 TOLERANCE AUX FAUTES

La tolérance aux fautes dans notre système de *swap* réparti peut-être assurée à quatre niveaux. Dans le premier niveau, l'utilisation du *swap* réparti par les processus critiques est interdite. Cela permet aux autres systèmes de rester intègres en présence de la panne d'un serveur.

Le deuxième niveau consiste à identifier les processus affectés par une panne et à les abandonner. Le *swap* réparti continue ainsi à fonctionner, et les applications peuvent être redémarrées.

Le troisième niveau est l'intégration d'un gestionnaire de tolérance aux fautes (comme GatoStar [Folliot, Sens, Raverdy, 1995]), qui permet de continuer l'exécution en présence de fautes au moyen d'un mécanisme de point de reprise où toutes les pages sont sauvegardées sur support stable.

Finalement, le quatrième niveau consiste à intégrer la tolérance aux fautes directement dans le gestionnaire de *swap* réparti au moyen d'un mécanisme de réplication des pages, ou par calcul de parité. Cette solution est la plus efficace mais est la plus complexe à mettre en oeuvre, c'est à terme celle que nous comptons implanter.

5.2 IMPLEMENTATION

Cette section décrit tout d'abord l'architecture globale du système, ensuite viennent: les modifications faites au noyau, le "driver MAÎS", le démon. Tout au long des ces descriptions nous exposerons les problèmes que nous avons rencontrés et justifierons les solutions choisies.

5.2.1 Architecture de MAÎS

Comme le montre la Figure 9, Maîs constitué d'un périphérique chargé dans l'espace du noyau. Ce *driver* est constitué des codes du nouveau *pager* ainsi que celui du client/serveur/démon Maîs. A ce *driver* vient s'ajouter un processus démon.

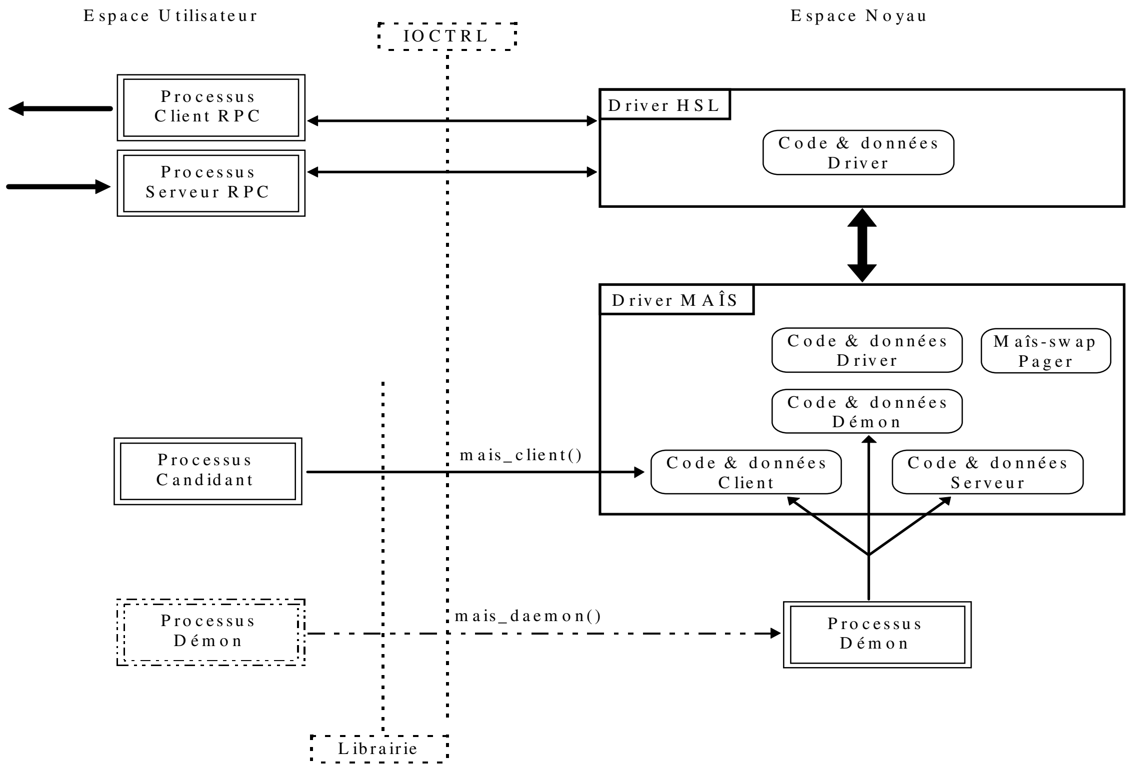

Figure 9. Architecture de MAÎS

5.2.1.1 L'intégration de Maîs au système

Le choix d'une architecture est une question difficile, de même le niveau d'intervention est crucial. En effet, il conditionne les possibilités qui sont par la suite offertes. Dans le cas de Maîs, nous avons commencé par explorer deux directions.

La première, conduit à concevoir un nouveau *driver* qui est monté[1] comme périphérique de va-et-vient. Un tel périphérique est considéré par le système comme un disque magnétique même si en réalité les écritures de pages sur disque sont remplacées par des écritures de pages en mémoire distante. Très simple de conception, ce système a l'énorme avantage d'être portable. Cependant, à cet avantage se combinent trois inconvénients : tout d'abord, la taille d'un tel

[1] Le système de fichiers d'Unix repose sur le montage dynamique de partitions (espace disque contigu), dans le système de fichier, ajoutant ainsi de nouvelles arborescences a celles existantes. De même l'espace de va-et-vient est généralement composé d'une ou plusieurs partitions qui sont montées (insérées) à l'exécution.

périphérique est limitée par la représentation des adresses disques. Celle-ci de 32 bits impose un espace swap maximal de 4Go. Il convient de remarquer que cette limite est aussi celle de l'espace d'adressage des processus. Le problème vient du fait que tous les processus doivent partager cet espace de va-et-vient. Le deuxième problème est lié aux politiques d'utilisation de l'espace de va-et-vient. Dans un système tel que FreeBSD il est possible de définir plusieurs périphériques de va-et-vient. Ces espaces sont alors utilisés de manière alternée, sans possibilité de contrôle. Dans notre cas, cela signifie que l'on ne peut imposer au système le choix d'une destination, cela nous oblige à n'avoir qu'un seul périphérique de va-et-vient et aller à l'encontre de notre idée de confinement des erreurs. Enfin, s'il n'est pas possible d'associer un *pager* à un processus il est encore moins possible d'associer type de pager à une région.

La seconde direction suppose une intégration plus fine au noyau. La difficulté de porter un tel driver sur un autre système constitue son inconvénient majeur. Mais un tel driver permet d'augmenter la taille de l'espace de va-et-vient à 16To. Dans notre cas, cela permet en plus d'agir plus finement et d'associer un *pager* particulier à une région de mémoire répondant ainsi à nos critères de sélection.

C'est cette deuxième solution, que nous avons choisie, car elle seule permet de contrôler assez finement le fonctionnement du mécanisme de pagination. Cela suppose donc de modifier ce système de pagination, tout en veillant à ce que cette adaptation modifie le moins possible le fonctionnement standard du noyau. Enfin, il faut à tout cela ajouter le mécanisme d'identification qui associera un *pager* particulier à la région à traiter.

5.2.1.2 Un périphérique chargeable dynamiquement

Tout au long du développement nous avons eu comme soucis de modifier le moins possible le code du noyau du système FreeBSD. Dans cette optique, il était naturel d'implémenter le système Maîs comme un gestionnaire de périphérique. Ce *driver* contient le code et les données du système Maîs.

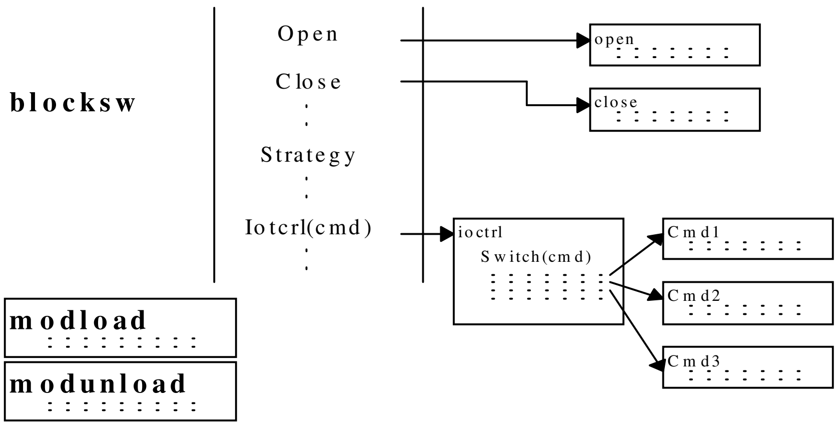

Figure 10. Structure d'un driver chargeable dynamiquement.

La structure d'un driver chargeable dynamiquement est assez simple. Elle se sompose de trois éléments :

- fonction *modload* : fonction d'initialisation du module. La valeur retournée prend la valeur nulle si tout s'est bien passé. Une valeur différente de zéro signifie une erreur d'initialisation et le module n'est pas chargée.

- fonction *modload* : fonction d'initialisation du module. La signification des valeurs de retour est similaire à celle de la fonction modload. Si la valeur de retour est différente de zéro, le module n'est pas déchargé (typiquement le cas d'un module toujours en cours d'utilisation).

- Structure *blocksw* : présente dans les périphériques de type block, cette structure contient les adresses des fonctions associées à ce périphérique :
    - open.
    - close.
    - strategy.
    - ioctrl : le rôle de cette fonction est normalement de consulter ou de modifier la configuration du périphérique adressé. Les paramètres de l'appel système sont : le n° du périphérique, le n° de la commande ainsi qu'une structure contenant les paramètres à passer à la fonction. Ces arguments peuvent êtres en entrée ou en sortie. Comme le montre la Figure 10, la fonction *ioctrl* se compose typiquement d'un *switch* qui en fonction du n° de commande passé exécute le code associé à celle-ci.
    - ...

L'accès au code du driver peut-être fait directement en mode noyau (c'est le cas pour le processus démon) ou par l'intermédiaire des appels système standard aux périphérique (c'est le cas pour les processus candidats). A cet effet, une librairie permet un accès transparent aux fonctions de Maîs à travers des appels systèmes standards : *open/ioctrl/close*.

5.2.1.3 Processus Démon (Client / Serveur)

Le code de mais est composé d'un nouveau *swap pager*, d'un démon, d'un client et d'un serveur. Un noeud se comporte différemment suivant qu'il est client ou serveur de plus un noeud peut potentiellement passer d'un mode a l'autre. Il doit donc contenir le code de toutes les fonctions qu'il peut remplir. Un *driver* est constitué exclusivement de code et de données normalement exécutées par le processus qui a réalisé l'appel système. Dans le cas de Maîs, la surveillance et l'insertion des autres noeuds, l'exécution des opérations de *page in* et *page out* est faite par un processus indépendant qui s'exécute dans le noyau (pour un accès facile aux fonctions).

5.2.2 Les modifications apportées au noyau

Ayant déjà choisi l'architecture de Maîs, nous allons aborder dans cette section les modifications du noyau nécessaires à son fonctionnement. Ces modifications portent sur le pager, les problèmes d'identification et enfin l'ajout d'un nouveau *swap pager*.

5.2.2.1 Ajout de nouveaux pagers

La mémoire virtuelle de FreeBSD est inspiré de celle de MACH. Elle en reprend le fonctionnement global, ainsi que son découpage en modules. Cependant l'aspect chargement dynamique de *pager* en mode utilisateur n'a pas été repris. FreeBSD utilise un nombre fixe de *pager* inclus dans le noyau et initialisés lors de son chargement. Les modifications portent sur l'ajout d'un *pager*, la correction de l'interface d'accès au *pager* et des ajouts à la structure *swblock*.

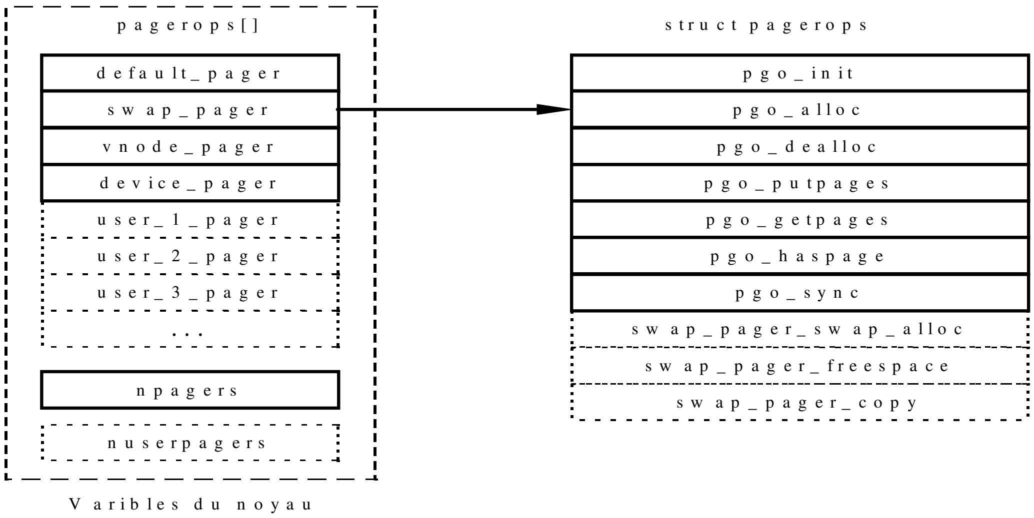

Figure 11. Ajout de nouveau *pagers* & Modification de l'interface.

Les pagers sont appelés via la structure *pagerops* (voir

Figure 11). Cette structure est composée d'une entrée par fonction du pager (init, alloc, dealloc, getpages, ...). La table *pagerops[]* contient une entrée par pager existant dans le système. Le nombre de pager indiqué par le système est fixe. Les appels aux différents *pagers* se font à travers la table et la structure pagesops. Par exemple (*pagerops[SWAP_PAGER]->pgo_alloc)(...) permet d'appeler la fonction pgo_alloc du *swap pager* et ainsi de créer une nouvelle instance de *pager*. Ajouter de nouveaux pager consiste donc à augmenter le nombre d'entrées de la table *pagerops*. Charger dynamiquement un *pager* consiste donc à placer dans la table *pagerops* un pointeur vers la structure *pagerops* contenant les fonctions du nouveau *pager*. L'initialisation de se dernier est supposée déjà faite au moment du remplacement. C'est initialisation peut être faite par la fonction de chargement du module dynamique.

Un autre problème se pose : les nouveaux pager sont destinés a remplacer un pager de type *swap pager*; hors certaines fonctions (au nombre de trois) sont appelées directement sans passer par l'interface *pagerops*. La solution à ce problème est résolue en deux temps : ajouter trois nouvelles entrées à la structure; modifier les appels de fonctions en appel à travers l'interface.

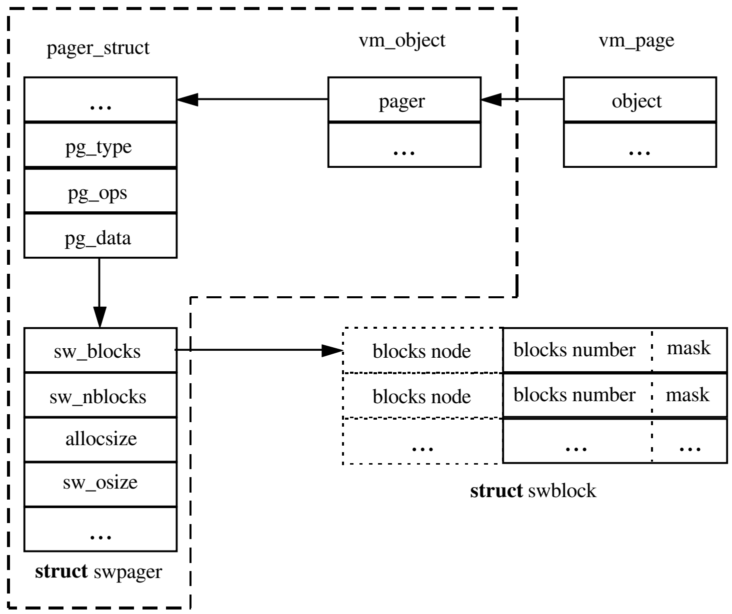

Figure 12. Organisation des structures de données du swap pager.

La dernière modification porte sur la structure swblock. Une instance de cette structure est créée et associée à un objet dès que l'une de ses pages est susceptible d'être placé en zone de *swap*. Chaque objet référence (voir Figure 11) un tableau de stucture *swblock*. Le nombre de structures allouées est fonction de la taille de l'objet. Chaque structure contient un champ de 16 bit et un tableau d'un maximum de 16 adresses disque. On y accède en utilisant l'offset dans l'objet comme index. FreeBSD associe à chaque page de la mémoire virtuelle d'un objet un couple bit/adresse disque. Le bit indique soit que la page n'a jamais été placée sur disque, soit que la page à été écrite et alors l'adresse est celle de la page sur le périphérique de *swap*. Dans le cas de Maîs, nous avons décider d'ajouter le numéro du noeud ou se trouve la page. L'adresse change de signification et représente un index de page distante.

Ces quelques modifications permettent a elles seules l'ajout dynamique de nouveau *pager*. Comme espéré, le noyau à subit un minium de changement. Ces changements ne modifient en rien sont fonctionnement normal, ni n'altèrent sa sécurité: l'ajout d'un *pager,* même dynamiquement, ne peut se faire que par un passage en mode noyau.

5.2.2.2 Identification des candidats

Les systèmes répartis doivent répondre à un impératif particulier : un système ne doit pas être mis en panne par la faute d'une machine distante. Dans ce sens, il nous a parus indispensable de pouvoir identifier les processus qui vont utiliser notre mécanisme de va et vient répartis. Cette identification permet de ne pas utiliser le *remote swap* pour des processus systèmes (ou critiques). Ainsi, la panne d'un noeud serveur n'affecte que des processus utilisateur, le système est toujours capable de s'exécuter, de détecter la panne et de mettre en oeuvre d'éventuels mécanismes de reprise.

Notre première idée était d'associer un *pager* à un processus de façon totalement transparente. Une commande supplémentaire aurait permis de lancer ce processus en mode candidat. Le fonctionnement de FreeBSD n'a pas permis une telle transparence. Tout d'abord, dans FreeBSD, un *pager* est associé à un objet et non pas à un processus. L'association ne peut donc se faire qu'entre *pager* et objet ! Un processus est composé d'un nombre quelconque d'objet, il faut donc répéter l'association pour chaque objet du processus. De plus, certains objets ne sont créés qu'à l'exécution: c'est le cas de la zone de tas. Enfin, l'utilisateur peut *mapper* des fichiers ou des zones de mémoire partagée en mémoire, ce qui a pour effet de créer des objets en cours d'exécution. Il n'est donc pas possible d'effectuer toute les associations au moment de la création du processus puisque tous les objets n'existent pas encore. Cette association ne peut être faite qu'a l'exécution par des appels plus ou moins explicites. Ces appels peuvent êtres implicites pour des objets présents dans tout processus, il suffit alors d'un seul appel à une fonction pour associer un nouveau *pager* à toutes les régions existantes. Les appels doivent êtres explicites pour des régions créées à l'exécution, il faut alors demander à associer cette région particulière à un *pager*.

Le mécanisme minimal nécessaire doit donc permettre d'associer un objet à un *pager*. L'association se fait en deux temps : Modifier les *flags* de l'objet pour indiquer quel type de *pager* on lui associe, puis créer une instance de *pager*. Ces deux étapes sont nécessaires, car le mécanisme de pagination de FreeBSD ne crée les instance de *pager* qu'à l'exécution. Un objet est associé à un type de *pager* à sa création, mais celui-ci n'est pas alloué. L'allocation ne se fait que lorsque le *pager* est réellement nécessaire. Un *pager* est utile uniquement si au moins une page est présente sur le périphérique de va-et-vient. C'est donc lors du premier *page-out* que l'on sait qu'un *pager* est nécessaire pour un objet et que l'ont créé une instance de ce pager.

L'allocation d'une instance de *pager* peut être faite, soit explicitement par le démon *pageout* lorsqu'il décide d'évincer une page d'un objet de la mémoire (version 2.1.0 de FreeBSD), soit implicitement par le *default pager* lors du premier appel à la fonction put (version 2.2.0 de mars de FreeBSD). En effet dans cette dernière version de FreeBSD lorsque aucun *pager* n'est alloué pour un objet, le default pager lui est associé. Ce default pager à ses propres fonctions *pgo_getpages* et *pgo_putpages*. Le rôle de cette dernière est d'allouer une réelle instance de *swap pager* dès le premier appel et de remplacer le *pager* de l'objet qu'elle gère par ce nouveau *pager*.

Nous avons utilisé les quatre bits de poids fort du champ *flags* de la structure *vm_object* pour contenir le type de pager à créer: 0 - *swap pager* standard, 1 - *pager* utilisateur 1, ... Une fonction permet de modifier le champ *flags* d'une structure *vm_object* et ainsi de passer la région en mode candidate. Cette modification est demandée par un appel explicite à la fonction *mais_client(@région, pager)* qui utilise les *ioctrl* pour passer en mode système et changer le *flag*. La création de l'instance de pager est automatique et s'effectue via un appel à la fonction *swap_pager_swap_alloc* de l'interface *pagerops*.

Nous avons laisser volontairement de côté les problèmes de partage de régions. En effet un objet de type mémoire anonyme peut être partagé entre plusieurs processus. Ce partage est fait par un appel à la fonction *mmap*. Ce partage se traduit par deux *vm_map_entry* de deux processus différents qui référence le même objet, le problème est alors de limiter ce partage exclusivement à des processus candidat ou à des processus non candidat.

5.2.3 Maîs

Maîs est composé d'un processus démon qui exécute le code du client ou du serveur (suivant son mode) et gère le passage d'un mode à l'autre, et d'un nouveau swap pager.

L'existence d'un processus est nécessaire. Un driver Unix est sensé réagire directement aux interruptions en provenance du matériel. Dans le cas de Maîs, ces interruptions proviennent de la couche basse de l'interface d'accès au réseau HSL. Le traitement est effectué par le processus courant (le processus qui s'exécute réellement sur le processeur au moment où l'interruption se produit). Typiquement ce processus doit traiter le message reçu, ce qui peut se traduire par l'émission d'un ou plusieurs messages. Hors, une demande d'émission ou de réception de message peut se traduire par la mise en sommeil du processus demandeur. Cette mise en sommeil est gênante, car elle agit sur le processus courant, processus qui n'a aucune raison d'être interrompus. Deux solutions existent : supprimer les attentes dans l'interface basse, gérer une file d'événement. La première solution n'est pas envisageable dans l'immédiat car, elle suppose une modification de l'interface basse pour qu'elle retourne un diagnostic en cas d'échec lors d'une demande d'émission ou de réception. Cette valeur de retour serrait alors traduite par la mise en file d'attente du message respectivement à émettre ou à recevoir. C'est la deuxième solution que nous avons adoptée. Le processus qui reçoit l'interruption se contente d'ajouter l'événement correspondant à cet événement dans une file et de réveiller le processus démon. C'est celui-ci, qui une fois éveillé prend un par un les événements présents dans la file et les traites. Ce processus doit s'exécuter en mode noyau. Hors, ajouter un processus dans le noyau n'est pas trivial. Seuls quatre processus particuliers (*pagedaemon*, *vmdaemon*, *swapdaemon*, et *backup*) sont présents en permanence dans le système. Ces quatre processus sont créés manuellement au démarrage du système. Créer un processus manuellement est contraire à la notion de portabilité de Maîs. Nous avons donc utilisé une solution plus dynamique, qui utilise l'appel système IOCTRL. Comme la plupart des appels systèmes *ioctrl* fait passer le processus appelant en mode système, exécute le code du noyau qui correspond à l'appel et enfin retourne au programme appelant en repassant en mode utilisateur. L'idée consiste donc à ajouter un nouvel appel (en plus de l'appel permettant de passer en mode candidat). Un processus utilisateur effectue un appel à *ioctrl*, passe en mode noyau, mais ne revient jamais en mode utilisateur. Une fois en mode noyau, ce processus effectue une boucle infinie d'attente d'événement. Ce processus s'endort en attente d'un événement. Dès qu'un événement est levé (événement interne : initialisation; ou arrivée de message...) le processus est éveillé. Cette boucle d'événement constitue le coeur de Maîs: les clients, serveur et démon utilisent cette même boucle d'événements.

Maîs fonctionne donc sur la base d'une file d'événements. Ces événements peuvent êtres internes (initialisations ...) ou externes (émissions ou réceptions de messages). L'adresse de la fonction d'interruption ainsi qu'une structure constituant les paramètres de celle-ci sont passés lors d'une demande d'émission ou de réception. L'interruption est traitée par le processus courant. Celui-ci reçoit en paramètre la structure passée en argument de la demande de réception (respectivement d'émission). Le processus interrompu ajoute alors dans la file l'événement correspondant à la nature de l'interruption (en passant toujours en paramètre la structure qu'il a reçu). Le processus démon est réveillé. Il prend alors un par un les événements présents dans la file et appel la fonction de traitement associé à chaque événement. S'il s'agit d'un événement externe, celle-ci appel alors la procédure de *call-back* associée (*recv_ctrl_init_node*, *send_ctrl_up_client*, ...). Le diagramme suivant résume l'organisation des appels de fonctions :

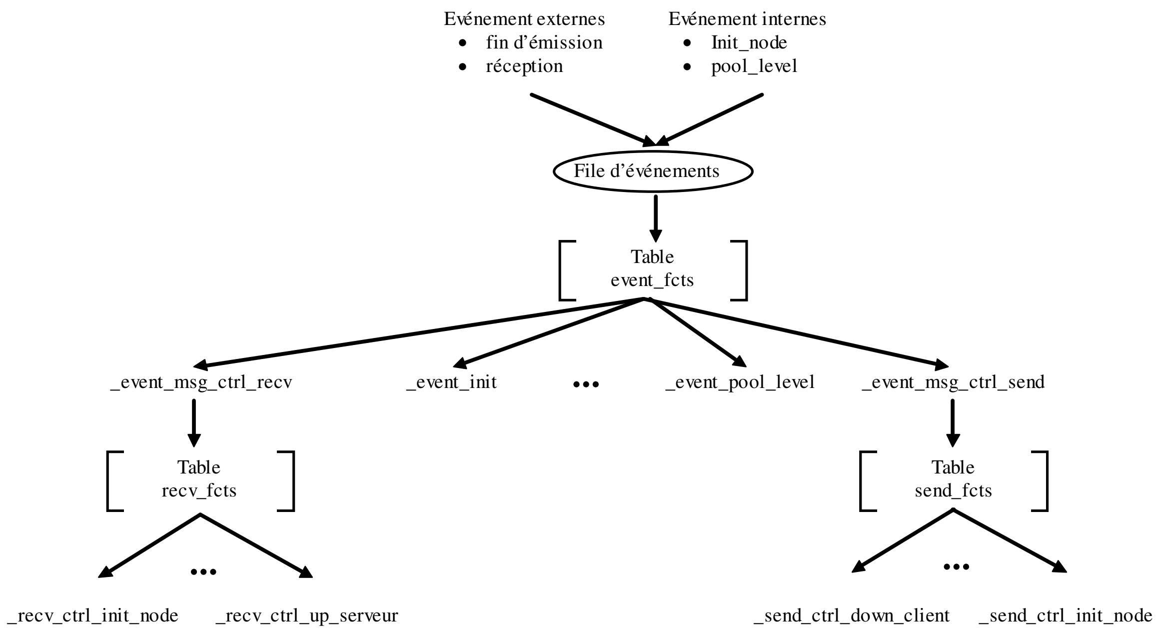

Figure 13. Gestion des événements.

Enfin, la présence de cette file d'événements permet d'ordonner naturellement les différentes sources d'événement et simplifie certain mécanisme du serveur. Dans le cas du client, les interruptions peuvent êtres traitées de deux façons différentes : soit en ajoutant un événement à la file, soit en exécutant directement le gérant d'exception. Ce dernier cas, réservé au swap pager annonce la fin des entrées sorties (réception d'un acquittement d'un message de page out). Ces deux façons de traiter les interruptions ajoutent donc des problèmes de synchronisation entre démon client et pager.

Pour assurer ses communications, Maîs utilise six différants : contrôle, commande et quatre de donnée. L'utilisation de six canaux distincts est nécessaire pour désynchroniser les opérations : les demandes de nouvelles pages (canal de commande) ne doivent pas ralentir les requêtes de *page in* (canal « *data_cmd_get_channel* »); mais aussi parce que c'est le meilleur moyen pour recevoir des données de taille différente.

5.2.3.1 Le démon Maîs

Le démon Maîs est chargé de l'insertion, du retrait des noeuds ainsi que du passage du mode client au mode serveur. L'insertion d'un noeud, le passage d'un mode actif à un mode client ou serveur, nécessite la gestion d'un état global. La connaissance d'un état global permet de traiter les changements d'état simultanés des différents noeuds. Par exemple, lors du passage d'un noeud en mode serveur, celui-ci parcourt la liste des noeuds et différencie un noeud actif, client ou serveur. Un changement d'état pendant un parcours amène le serveur à ignorer ce noeud. Nous avons volontairement négliger ces problèmes pour nous concentrer sur les mécanismes de Maîs. Dans la suite de cette section, nous décrivons sur un exemple simplifié (insertion d'un noeud : passage du mode inexistant à actif) les mécanismes minimum tout de même nécessaire au fonctionnement de Maîs.

L'algorithme d'insertion de noeud utilise une propriété de l'interface base du réseau HSL: toute demande d'émission reste pendante sur la machine sur laquelle elle est faite tant que la demande de réception correspondante n'a pas été faite. C'est la réception (lors du passage de l'adresse de réception à l'émetteur) qui déclenche l'émission. Utilisant ce phénomène chaque noeud lors de son insertion effectue des requêtes *send*, puis *receive* vers tous les émetteurs potentiels. Si le noeud est le premier, toutes les réceptions échoueront immédiatement puisque le noeud distant n'existe pas, toutes les émissions seront mises en attente. Lors de l'insertion du 2ème noeud, toutes les émissions seront mise en attente, mais la réception qui correspond au premier noeud pourrat, elle se terminer. Alors, le premier noeud est éveillé et termine l'algorithme effectuant une nouvelle requête *receive* vers le deuxième noeud. Cette deuxième requête à pour objet de terminer le *send* déjà demandé par ce second noeud et qui est toujours pendant. Cet algorithme utilise bien sur la gestion d'événement décrite précédemment.

Maîs gère une structure d'état du démon aussi qu'une structure d'état pour chaque noeud potentiel. Il faut voir cette insertion comment un mécanisme de rendez-vous. En voici l'algorithme approximatif : epuré de tous les problèmes de synchronisation et de *race contrôle*. Nous y retrouvons les quartes fonctions décrites ainsi que le rôle des drapeaux, qui sont associées aux différents noeuds.

event_init(...)
    initialiser la structure ***daemon_state***
    Pour chaque noeud potentiel
        déposer un événement ***init_node***

even_init_node(i,...)
initialiser la structure nodes_state[i]...
flags = INACTIF.
oflags = oflags **|** ***INIT_NODE_IN_PROGRESS***.
demander envoie(vers noeud i, canal de contrôle, ***MSG_CTRL_INIT_NODE***,...)
iflags = iflags **|** ***INIT_NODE_IN_PROGRESS***.
demander réception(du noeud i, canal de contrôle,...)
si erreur réception : pas de noeud distant
iflags = iflags & ***~INIT_NODE_IN_PROGRESS***

recv_ctrl_init_node(i,...)
iflags = (iflags & ***~INIT_NODE_IN_PROGRESS***) **|** ***ACTIF.***
si oflags = ***ACTIF***
flags = ***ACTIF***
demander réception(du noeud i, canal de contrôle,...)

send_ctrl_init_node(i,...)
oflags = (oflags & ***~INIT_NODE_IN_PROGRESS***) **|** ***ACTIF.***
si iflags = ***ACTIF***
flags = ***ACTIF***
demander réception(du noeud i, canal de contrôle,...)

Les algorithmes de passage du mode actif au mode serveur ou client fonctionnent sur le même principe : on informe les noeuds actifs distants, on attend les acquittements de tous les noeuds. Alors, on alloue les structures supplémentaires spécifiques au client ou serveur. Le retour au mode actif suppose la libération de toutes les pages, la terminaison des *receive* en attente sur les canaux de donnée et de commande suivie d'un rendez-vous pour repasser en mode actif. Le passage du mode actif au mode passif (pour terminer Maîs) suppose de terminer toutes les émissions et réception en cours sur le canal de contrôle. Ce n'est pas possible pour l'instant puisque les émissions faites vers les noeuds non connectés ne peuvent être retirées des tables internes du driver HSL. La version future du driver permettra d'effectuer le vidage d'un canal particulier.

5.2.3.2 Les clients / serveurs Maîs

Une fois activé un noeud passe alors soit en mode serveur, soit en mode client. En mode serveur, il fournis les pages aux noeuds clients, reçois les pages évincées, et répond aux requêtes de pagination. Le client maintient un pool de pages libres pour l'évincement de nouvelles pages, ainsi qu'un pool de *buffers* pour l'écriture de pages. Le pool de pages constitue l'espace de va-et-vient du client. Le pool de *buffer* est destiné au calcul de pages de parités. Dans le cas de pager maintenant une redondance de pages par calcul de parité, celle-ci est effectuée par une différence entre la nouvelle version de la page et l'ancienne version. Au moment de l'évincement, deux versions de la page doivent donc coexister dans le serveur. La correspondance entre les pages en faite à l'aide d'un numéro d'index. Lorsque l'on évince une page, son numéro d'index est accolé afin de d'effectuer la correspondance avec l'ancienne page. Une fois la parité propagée, l'ancienne page est remplacée par la nouvelle. Enfin, l'ancienne est replacée dans le pool de *buffer*.

Un serveur se compose principalement de deux ensembles de structures : les pages constituant l'espace de pagination, et les *buffers* sur lesquels sont réalisés les écritures par les clients. L'index d'une page est constitué de l'adresse de la structure *index to page* qui la représente. Les pages sont placées soit dans la liste des pages libre (*free index to page list*), soit dans la liste de pages occupée associée à chaque noeud. La structure *index to page* contient un certain nombre d'information destinée à identifier la page chez le client (a des fins de statistique, ou de futurs points de reprise coordonnés ou mécanisme de tolérance aux fautes). Cette structure est renseignée lors du premier évincement de la page. Le second ensemble de structures est le *pool de buffer*. A chaque noeud est associé un certain nombre de pages destinées à effectuer les écritures. Ces pages sont alors placées respectivement dans les listes *free* et *busy* de la structure *entry pool buffer* associée au noeud. Ces pages constituent le quota minimal de page fournies à un noeud client. Ce quota en augmenté automatiquement en prenant des pages dans la liste des *buffer* libres. Cela permet de fournir instantanément des pages supplémentaire à un noeud demandeur. On comprendra en fait par *busy* un *buffer* qui a été consommé par un client mais pas encore acquitté.

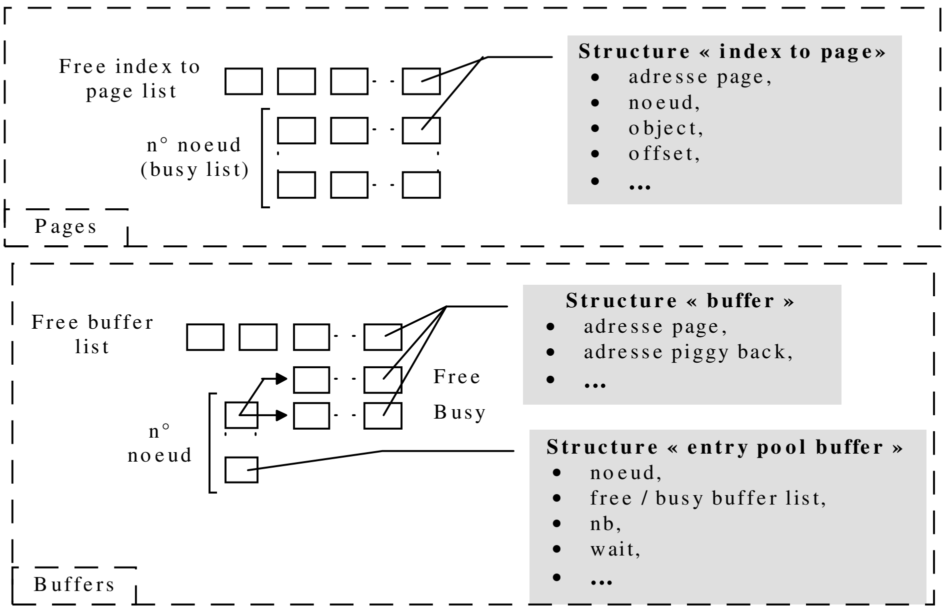

Figure 14. Structure d'un noeud serveur.

Un serveur peut répondre à trois types de requêtes :

- Demandes de nouvelles pages. Destinées au *pool* de pages du client, ces pages sont prises directement dans la liste des pages libres. Ces pages sont demandées par block de *k* pages. Un serveur répond en fournissant les index et n° de noeud des pages demander et peut si nécessaire donner moins de pages que l'on ne lui en a demandé.

- Evincement de page. Un client envoie un page, ainsi que son index. Le serveur commence par lui envoyer, si c'est possible, un nouveau *buffer*. Il valide l'écriture de la page : en échangeant ancienne et nouvelle page (à posteriori, suivant le type de pager, il devra propager au préalable la différence entre ces pages). Enfin, il acquitte cette page.

- Demande de *page-in*. Un client envoie au noeud concerné l'index de la page qu'il désire, le serveur lui répond en lui envoyant celle-ci.

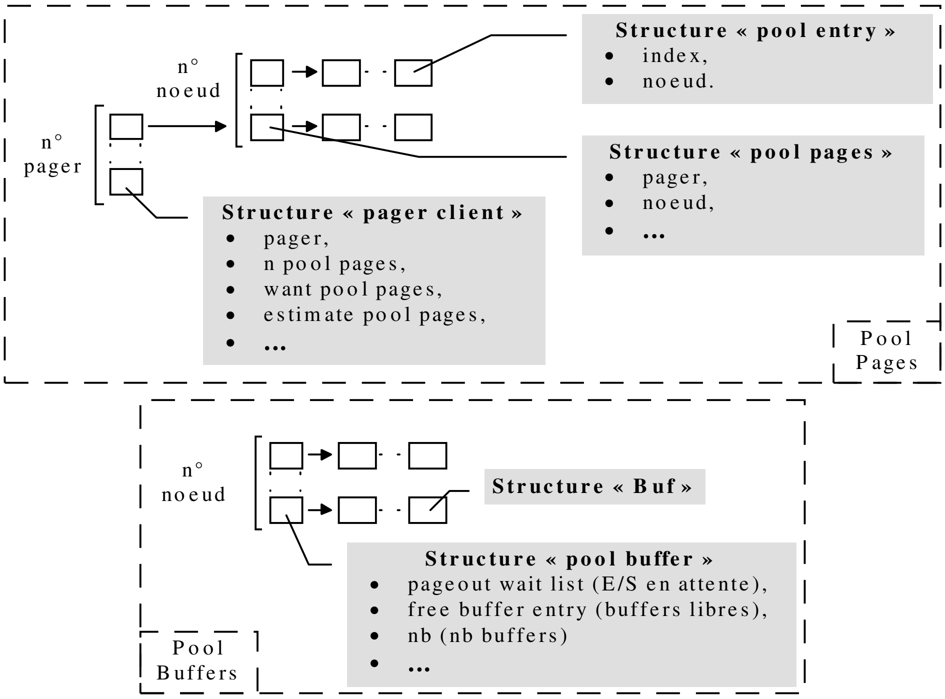

Figure 15. Structure d'un noeud client.

Comme le montre la Figure 15, un client est constitué de deux ensemble de structures : un pool de pages ainsi qu'un pool de *buffer*. Le pool de pages contient un certain nombre de pages libre. Ce pool constitue l'espace libre de va et vient visible par un client. Le *swap pager* de Maîs y prend des pages chaque fois qu'il en a besoin. Par soucis d'extensibilité, il existe un pool de pages par *pager*. Cela permet de maintenir indépendamment le *pool* des serveurs. Un client voulant renouveler un pool de page s'adresse directement au serveur associé. C'est le serveur qui s'occupe d'allouer ces pages et si nécessaire de contacter d'autres serveurs. En effet dans le cas de serveurs de pages de parité, il est nécessaire d'allouer des pages sur différents serveurs et d'en désigner une comme page de parité. Tout ce mécanisme peut de ce fait être caché du client. Les pages d'un pool sont regroupées par serveur, ce qui permet d'envisager par la suite toutes les politiques de renouvellement, autant que de consommation. Lors d'une demande d'allocation, ou de libération de page par le *swap pager* le niveau du pool est vérifié. Un dépassement, dans un sens ou dans l'autre ajoute un événement *pool level* dans la file des événements. Cet événement est alors traité de façon asynchrone par le processus démon et pas par le swap pager. Le maintient du *pool* utilise un simple mécanisme à seuil. Lorsque l'on souhaite remplir le pool, on parcourt (de façon cyclique) la liste des noeuds serveurs et leur envoie à chacun une requête de demande de pages. La difficulté provient du fait que ces serveurs répondent de façon asynchrone et peuvent fournir moins de pages qu'il ne leurs en a été demandé. Il est donc nécessaire de maintenir trois variables lors des changements de seuil : *n / want / estimate pool pages*. La variable *n pool pages* représente le nombre actuel de pages dans le pool. La variable *want pool page* indique lors d'une nouvelle demande de pages, le nombre de pages à atteindre. Enfin, *estimate pool page* indique le nombre de pages estimé. Les serveurs peuvent retourner moins de pages que demandé. Cette dernière variable permet de réévaluer les demandes si celles-ci n'ont pas étés complètement satisfaites.

5.2.2.3 Maîs swap pager

Le swap pager de Maîs est une adaptation de la version originale de FreeBSD. Les principales différences résident dans les fonctions d'allocation et de désallocation de pages, ainsi que dans les fonctions d'écriture ou de lecture de pages (respectivement *putpages* et *getpages*). Les différences entre les deux *pager* proviennent du fait que les adresses sur disque sont maintenant composées d'un couple : index et numéro de noeud. Toutes les fonctions du pager d'origine ont étées adaptées pour tenir compte de la nouvelle structure *swblock* ainsi que du numéro de noeud associé à chaque adresse (index).

Les fonctions d'allocation et de désallocation de page dans l'espace de va et vient on subit un changement important. Celles-ci n'utilisent plus la structure des *rlist* pour gérer l'espace de swap. Cette structure, constituée d'une liste chaînée de fragments libres permet de représenter une ressource locale. Hors, l'espace de swap est ici une ressource globale à laquelle le client accède à travers un pool de page. De plus cette vision est incomplète : elle ne permet par exemple pas de connaître l'espace de va et vient disponible dans le système. Cette connaissance est répartie et nécessite l'adjonction de mécanismes d'état global si on veut pouvoir l'estimer. Le *pager* utilise la fonction *mswap_pager_getswapspace* pour allouer des pages dans l'espace de va-et-vient, ainsi que la fonction *mswap_pager_freeswapspace* pour les libérer. L'interface de ces fonctions à été modifiée pour passer le numéro du pager concerné ainsi qu'un pointeur vers un ensemble de couples index et noeud (et non plus seulement adresse disque). Cette table de couple contient soit les différentes pages à libérer (*mswap_pager_freeswapspace*), soit les pages demandées (*mswap_pager_getswapspace*). La fonction d'allocation utilise une politique circulaire pour choisir des pages sur les différents serveurs. Enfin les fonctions d'allocation et de désallocation surveillent en permanence le niveau du *pool* de pages (*mswapsizecheck*). Si le niveau en insuffisant ou trop élevé, un événement du type *pool_level* est ajouté à la liste des événements à traiter. Le traitement associer à cet événement rétablira le niveau du pool par la suite.

La seconde difficulté réside dans l'écriture des fonctions *mswap_pager_getpages* et *mswap_pager_putpages*. La version initiale de ces fonctions cherche à regrouper les pages à lire (respectivement à évincer) en une seule entrée - sortie. Cette politique est dictée par les caractéristiques des accès disques. Dans le cas de Maîs, les entrées - sorties sont effectuées à travers un réseau qui n'a pas les mêmes propriétés. Maîs cherche à paralléliser les traitements des serveurs, pour cela, il utilise une politique circulaire d'allocation de pages. Bien que cela simplifie la politique de sélection des pages, les mécanismes d'entrée - sortie s'en trouvent compliqués. La mécanique d'entrée sortie utilise deux entités de *FreeBSD* : les *buffers* (représentés par une structure *buf*) et les *vnode* (qui représente un périphérique virtuel). A ces structures s'ajoutent la *swp_clean_t*. Cette dernière permet de synchroniser les opérations du *pager*. Une structure *swp_clean_t* est associé à chaque demande d'évincement ou de lecture de page. Elle contient une référence à la structure *buf* associée. Une partie de ces structures est préallouée afin d'éviter les risques d'interblockage. En effet allouer des structures lors d'une demande d'évincement peut être impossible si cet évincement est dû à l'absence de page libre. Cela permet aussi de limiter le nombre d'opérations d'évincement préventives simultanées en cours. Une structure *buf* décrit les opérations à effectuer : lecture, écriture et quel est le périphérique (*vnode*) concerné. Les champs décrivant l'entrée - sortie sont l'adresse en mémoire du *buffer* ou placer les données (pour une lecture, respectivement prendre les données : écriture), l'adresse disque, la taille de l'entrée sortie, la nature de cette entrée sortie (lecture ou écriture) et enfin le *vnode* représentant le périphérique. Dans le cas des opérations de mémoire virtuelle, l'adresse mémoire fournie n'est pas celle d'un *buffer* (les entrées sorties de type fichier passent par un cache et sont accédées à travers ce cache) mais directement celle de la page concernée. Les pages sont maintenues en mémoire afin de les rendre toujours accessibles en le *mappant* dans l'espace d'adresse du noyau. C'est l'adresse de ces pages dans le noyau qui est placée dans la structure *buf*. Une fois la structure *buf* renseignée, le *pager* appel la fonction *strategy* du *vnode* afin de lancer cette entrée sortie. La fonction *strategy* du swap pager appel ensuite la fonction *strategy* du périphérique disque associé à la partition de va-et-vient pour lancer effectivement l'entrée sortie. Dans le cas de Maîs, plusieurs choses diffèrent : vu la distribution des pages sur les serveurs, une requête de pagination à une probabilité presque certaine (sauf dans le cas de serveur unique) de concerner plusieurs serveurs et des pages non contiguës en mémoire de ces serveurs. Cela suppose autant de structure *buf* que de page disponible. La structure *swp_clean_t* doit maintenant référencer autant de structure *buf* que de pages paginables en une seule requête. Typiquement, le nombre de pages est fixé à 8 dans le noyau. De plus toutes ces pages doivent être *mappées* indépendamment dans le noyau. Enfin la taille des adresses disque pose problème : fixée à 32 bits, elle ne peut contenir à la fois l'index de la page sur 32 bits et le noeud sur 8 bits. Pour remédier à ce problème nous avons décider d'utiliser les propriétés mêmes des *vnodes*. Un *vnode* peut être associé à un périphérique physique. Celui-ci possède un numéro majeur qui l'identifie parmi les autres : par exemple gestionnaire de disquette ; mais il possède aussi un numéro mineur qui identifie les périphériques physiques qu'il gère : par exemple le numéro du lecteur. Ainsi, un gestionnaire de lecteur de disquette de numéro majeur 7 peut gérer deux lecteurs de disquette A et B de numéro mineur respectif 0 et 1. Appliqué à notre problème cela revient à associer un numéro mineur à chaque noeud potentiel du système. On crée initialement autant *vnode* que de noeuds dans le système. On leur associe à chacun le numéro majeur du driver Maîs et un numéro mineur correspondant à un numéro de noeud. Le numéro 0 étant associé au driver Maîs les noeuds 0 à k sont associée au numéro mineur 1 à k+1. Réaliser une entrée sortie à l'adresse X sur le périphérique K revient à placer *mapper* X dans le noyau, placer l'adresse dans le *buffer* y placer aussi le *vnode* numéro k+1 et appeler la fonction *strategy* du *vnode*. Celle-ci reçoit la structure *buf*, ainsi que le numéro (mineur et majeur)du périphérique et peu terminer l'entrée sortie en émettant un ou plusieurs messages vers le noeud concerné. Une requête est découpée en plusieurs autres sous requêtes vers différent noeud. Il faut donc synchroniser les réponses afin de libérer les ressources réservées (structure *swp_clean_t* ...) et terminer l'opération de pagination. Pour cela on utilise un sémaphore associé à la structure *swp_clean_t.* A la fin du traitement de chaque sous requête ce sémaphore est incrémenté. Lorsque, toute les requêtes sont terminées il est alors possible de terminer l'opération. La Figure 16 suivante décris succinctement la fonction *mswap_pager_putpages* ainsi que les fonctions *mais_strategy* et *mswap_pager_iodone1* associées.

Mswap_pager_getpages(object,m,count,reqpage)

- Repérer les structures swblocks associées aux pages m (count pages),
- Rechercher en arrière à partir de reqpage :
    - ♦ éliminer les pages qui ne sont déjà en mémoire
    - ♦ éliminer toutes le pages si l'accès précédent était séquentiel
- Rechercher en avant à partir de reqpage :
    - ♦ éliminer les pages qui ne sont déjà en mémoire

- S'il n'y a pas de structure swp_clean_t libre : en allouer une, ainsi que les buffers associés.
- ...sync_count = count : nombre d'entrée sorties à réaliser !
- Pour chaque page i sélectionnée (dans reqpages),
    - préparer la structure buffer associée (bp[i]) :
        - ♦ mapper la page dans le noyau => kva [i]
        - ♦ bp[i].flags = B_READ | B_PAGING ...
        - ♦ bp[i]..b_blkno = reqpages[i].page (index)
        - ♦ bp[i]...b_addr = kva[i].
        - ♦ bp[i].b_bcount = PAGE_SIZE
        - ♦ bp[i].b_iodone = mswap_pager_iodone1
        - ♦ associer le vnode « reqpages[i].node » à bp[i]
    - VOP_STRATEGY(&bp[i])

- Attendre la fin de l'entrée sortie : ...sync_count = 0.

- Terminer le requête de get pages
- libérer toute les structures qui doivent l'être

Mswap_pager_iodone1()

- Terminer l'entrée sortie sur la structure bp[i]
- décrémenter ...sync_count
- Si requête complète réveiller le processus demandeur pour terminer la demande de pagein.

Mais_strategy(bp)

- Suivant la valeur de bp->flags

- Si B_READ
    - demander à recevoir la réponse (la page demandée):
        - ♦ depuis le noeud : minor(bp->b_dev) - 1
        - ♦ sur le canal : DATA GET CHANNEL
        - ♦ à l'adresse : bp->b_addr
        - ♦ d'une taille taille : bp->b_bcount
    - demander à émettre la requête de page
        - ♦ vers le noeud : minor(bp->b_dev) - 1
        - ♦ de la page : bp->b_blkno

- Si B_WRITE
    - ...

Figure 16. Description de la fonction mswap_pager_putpages.

Elle se décompose en 4 phase :

- Sélection des pages à lire parmi celles proposées (seule *reqpage* est obligatoire). La variable *count* est diminuée d'autant.
- préparer et demander les entrées - sorties : une structure *swp_clean_t* et *count* structures *buf* suivies d'un appel à la fonction *strategy*.
- attendre la terminaison des Counts requête à la fonction strategy.
- Terminer la requête de *pagein*.

La fonction *Stategy* traduit le requête faites par l'intermédiaire de la structure *buf* en un envoie de message au serveur correspondant. C'est la réception de la réponse (la page attendue), qui appellera la fonction mswap_pager_iodone1 pour terminer le travail de la fonction *strategy* et éventuellement provoquer la fin de la demande de *pagein*. La fonction *mswap_pager_putpages* fonction sur le même modèle (en mode synchrone). Sa mise en œuvre est cependant plus complexe étant donné qu'elle est capable de réaliser de opération d'évincement asynchrone. Pour cela deux fonctions lui sont associée afin de terminer correctement ces opération asynchrone sans rester bloquer dans la fonction *mswap_pager_putpages* (comme c'est la cas pour la fonction précédante : phase 3).

5.3 PERFORMANCES

Compte tenue des problème de jeunesse de l'émulateur, il n'a pas été possible d'effectuer des mesures réelles. L'interface base nécessite encore quelques corrections en ce qui concerne les problème de concurrence. Cependant il à été possible de mesurer la durée même de traitement des fonctions *mswap_pager_putpages* et *mswap_pager_getpages*. Les mesures sont prise à l'aide de fonction internes du noyau. La précision de mesure est de l'ordre de la micro - seconde et l'effet de sonde négligeable (de l'ordre de 0,2%). Le déroulement d'un appel se fait en 5 phase :

- Sélection des page, préparation de structure, appels à la fonction strategy, envois de messages à travers le réseau
- traversée du réseau
- Réception de la requête, traitement, envoie de réponse
- traversée du réseau
- Réception de la réponse, terminer le traitement.

Le temps de traversé du réseau à été calculé à l'aide de simulation obtenues sur le réseau HSL. Il sont de l'ordre de 100µs pour un message de 4ko et < 10µs pour des message < 100 octets. Les valeurs données dans le tableau sont calculées sur cette base et surestime la taille des messages Les premiers et dernier temps concerne le client et ont été mesurés entre l'entrée dans la fonction jusqu'au début de la mise en attente, et du début de l'interruption de réception de la réponse jusqu'à la sortie de la fonction get (respectivement put). Le tableau suivant indique les mesure prise pour la fonction *mswap_pager_getpages*..

Nb de Pages	Page In					
	Std (ms)	MAÎS (ms)			Speed Up	
		HSL	Système			
			client	serveur	Maîs	Maximum
1	12,2	0,113	0,2	0,15	26,35	108
10x1	**24,3**	**1,13**	**2**	**1,5**	**5,25**	**21,5**
10x16	117,5	18	33	24	1,57	6,5

Dans ce tableau nous retrouvons les différents temps : réseau HSL, client et serveurs. La somme de ces temps nous donne le durée totale de traitement du *page in*. Nous avons de plus mesuré le temps mis par le swap standard pour effectuer le même travail. Enfin les colonnes *speed-up* nous donnée l'accélération obtenue ainsi que sa borne supérieure. Cette dernière est obtenue en ne tenant compte que du temps réseau. Des mesures identiques pour la fonction *mswap_pager_putpages* sont regroupées dans ce second tableau.

Nb de Pages	Page Out					
	Std (ms)	MAÎS (ms)			Speed Up	
		HSL	Système			
			client	serveur	Maîs	Maximum
1	-	0,12	0,25	0,21	-	-
10x1	13,4	1,2	2,5	2,1	2,3	11,2
10x16	135,4	19,2	5,6	33,9	2,3	7,05

Dans ces deux séries de mesures plusieurs choses sont à prendre en considération :

- Les résultat de mesures standard sont arbitrairement bons car dans tous les cas les pages demandées étaient consécutives sur disque. Dans tous les autres cas de figure, le temps de pagination serra de K*12,2ms (resp. K*13,4) où K est le nombre de page à évincer. Alors que dans le cas de Maîs il n'y a pas de détérioration.

- Les comportements les plus probables sont grisés sur les tableaux.

- Le dernier problème concerne l'émulateur. Sa mise en œuvre utilise un serveur RPC pour la réception des messages. Ce serveur provoque à chaque réception de message une double commutation de message. Pour le démon Maîs cela se traduit par le traitement de événements un par un avec une commutation de tache chaque fois qu'un message est reçu. Le serveur qui utilise la file des événements (contrairement au client : pager qui gère directement les interruptions) est plus sensible à ces commutations de taches. Sont temps devient alors proportionnel au nombre de messages et donc de pages échangées. Ces problèmes devrais diminuer dans le cas d'une interface réseau matérielle mais ne pas disparaître. Plusieurs interruption consécutives produiront des événements qui seront traités après une seule commutation.

Ces mesures nous donnent des résultats encourageants. Compte tenu de la limite supérieure à l'accélération nous avons obtenu de 24 à 32% du maximum possible. Cependant deux problèmes subsistes compte tenue du choix de l'interface basse : Celle-ci nous impose une gestion par événements de la réception des messages, ce qui provoque des commutations inutiles. Ces commutations diminuerons mais ne disparaîtrons pas avec l'interface matérielle. De plus la latence de appel à l'interface basse et très importante. Sa durée, de l'ordre de 0,1ms, comparée aux 0,2ms de latence client pour la fonction *get*, constitue la moitié de la latence système.

La dernière mesure effectuée donne une durée de prise en charge du défaut de page par le *fault_handler* (entre l'interruption et le début de la fonction *get*) de 0,67ms ce qui est bien supérieur aux 0,2ms de latence système pour Maîs. Cette latence s'ajoute bien sûr à toute les mesures déjà effectuées. Cela nous laisse supposer que si on ne peu diminuer cette latence, il faudra au moins optimiser l'évincement pour de plus gros cluster afin de diminuer le coût de cette latence et profiter réellement des performances de Maîs.

5.4 EVOLUTIONS

Les résultats obtenus nous dictent un certain nombre de modifications. Tout d'abord il faut changer le niveau d'interface au réseau utilisé en choisissant l'interface put. Ce qui nous permettra d'utiliser uniquement les interruptions comme mode de programmation. Il faut simplifier le code du *pager*. Il n'est pas nécessaire d'utiliser le *buffer* et *vnode*. En simplifiant ces fonctions nous gagnerons en portabilité et en performance. Enfin la structure actuelle est relativement statique et ne permet que difficilement de mettre en place les mécanisme de tolérance aux fautes et de reconfiguration. La solution à ce problème peu être de considérer des *cluster* de pages. Des *clusters* de pages gérés indépendamment les uns des autres par des groupes de noeuds permet un reconfiguration plus aisée. Chaque cluster est géré par un ensemble de serveurs si l'un d'entre eux veut interrompre sont fonctionnement il suffit de choisir un nouveau serveur par cluster auquel il participe. Un noeud peut alors être à un instant donné serveur, passer en mode client. Dès cet instant il commence à migrer cluster par cluster les pages qu'il détient tout en continuant à fonctionner en mode client. La fin de ce mode peut de plus interrompre le mécanisme de migration qui n'est plus justifié. C'est aussi une façon élégante d'implémenter de multiple politique de tolérance aux fautes puisque chaque cluster gère indépendamment ces propres pages.

Une seconde catégorie d'évolution concerne l'emploie de Maîs dans des mécanisme plus généraux. Il est possible de concevoir Maîs comme une interface d'application qu'il est possible de mettre à la disposition de clients. Fournissant de mécanisme de gestion de la mémoire et permettant d'ajouter la cohérence nécessaire à de la mémoire répartie partagée. De plus Maîs peut très facilement être étendu pour aider à la mise en place d'autre mécanisme tel que les points de reprise et la migration de processus. En effet pour effectuer un point de reprise il suffit d'évincer la totalité du processus et de demander au serveurs de stocker sur disque les pages du processus qu'il possèdent. Ce qui est possible car un version tolérante aux faute de Maîs peut être considérée comme support stable. De plus l'écriture en parallèle des pages devrait s'avérer très efficace. La migration de processus est presque immédiate. En effet, il suffit d'évincer la totalité du processus. Après déplacement des structures critiques, le mécanisme de pagination de Maîs permet de façon transparente de renvoyer les pages non plus au client initial mais vers un autre noeud du système. En fait l'optimisation de ces deux mécanisme dépendra de la façon dont l'API de Maîs permettra de contrôler ses mécanismes internes.

Toutes ces évolutions on pour objectif un support d'exécution pour application parallèles incluant la tolérance aux fautes, les points de reprise, la migration de processus. Toute ces mécanisme étant obtenu à partir d'une fonction de base : le mécanisme de va et vient en mémoire distante fournie par Maîs.

Chapitre 6
Conclusion

Les résultats obtenus dans cette implémentation de Maîs sont encourageants. Nous avons mis à jours de gains important (de 1,6 à 26). Maîs n'est qu'à la première étape d'une longue série de développement possibles. Nous comptons notamment développer une version totalement dynamique et tolérante aux fautes, ce qui permettra dans un avenir plus lointains d'envisager de nombreuse évolution, tel que le migration et les points de reprises. Les difficultés rencontrées concernaient pour la plupart le manque de stabilité d'interface basse. Celle-ci est sujette à des défaut de fonctionnement dans le cas d'accès concurrents. Nous somme malheureusement dans ce cas de figure puisqu'à un instant donnée au moins deux processus (le *pager* et le démon Maîs) accèdent à cette interface. Les autres difficultés étaient liée directement au développement de code dans la noyau et aux problème de mise au point de ces code. Notamment le traçage (pour le déverminage) est très difficile à mettre en place et quelque fois sa simple présence suffit à masquer des erreurs (effet de sonde).

Maîs est un projet original et d'envergure qui est appelé à se poursuivre et constitue une nouvelle opportunité de recherche au siens du laboratoire.

Références

[Coulouris, Dollimore et Kindberg, 1995] G. Coulouris, J.Dollimore et T.Kindberg. Distrubuted Systems concepts and Design (Second Edition). Addison Wesley. 1995.

[Bernard, Hamma, 1994] G. Bernard et S. Hamma. Remote memory paging in network of workstations. Dans *Proc. Symposium UNIX Users Group. International Conference on Open Systems: Solutions for Open World,* à Moscou - Russie. Avril 1994.

[Thekkath, 1994] C.A. Thekkath. System Support for Efficient Network Communication. Departement of Computer science and Engineering, University of Washington, Seattle, repport technique. Juillet 1994.

[Stumm, 1988] M. Stumm. The design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. Dans *2nd IEEE Conferencec on Computer Workstations*. Mars 1988.

[Theimer, Lantz, 1989] M.M. Theimer et K.A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Trans. on Software Engineering*, 15(11). Novembre 1989.

[Felten, Zahorjan, 1991] E.W. Felten et J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical report no. 91-03-09. University of Washington. 1991.

[Silberschatz, Galvin, 1994] A. Silberschatz, P.B. Galvin. Operating System concepts (Fourth Edition). Addison Wesley. 1994.

[Fédaoui, 1994] L. Hauw-Fédaoui. Spécification d'un schéma de gestion générique de qualité de service pour applications multimédias sur réseaux à haut-débit. PhD thesis. Université Pierre et Marie Curie, Paris VI. 1994

[Trecordi, 1994] V. Trecordi. Inter-process communication performance in high-speed networks. Dans *IEEE*, editeur, *International Conference on Communications*, volume 1, pages 57-63. 1994

[Zhou, Wang, Zheng, Delisle, S.Zhou, J.Wang, X. Zheng et P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical Report CSRI-

1992] 257. Université de Toronto, Ontario. 1992

[Lin, Hsieh, Du, Thomas, MacDonald. 1994] M. Lin, J. Hsieh, D.H.C. Du, J.P. Thomas et J.A. MacDonald. Distributed network computing over local ATM networks. Dans *IEEE*, editeur, *Supercomputing*, pages 154-163. 1994

[Folliot, 1992] B. Folliot. Méthodes et Outils de partage de charge pour la conception et la mise en oeuvre d'applications dans les systèmes répartis hétérogènes. PhD thesis. Institut Blaise Pascal, Laboratoire MASI. Université Pierre et Marie Curie, Paris VI. 1992

[Minnich, 1993] R. Minnich. Mether-NFS. A Modified NFS Which Support Virtual Shared Memory. Dans *Proc. 4th Symposium on Experiences on Distributed and Multiprocessor Systems (*SEDMS-IV), San Diego, CA. Septembre 1993.

[Anderson, Dahlin, Neefe, Patterson, Roselli; Wang, 1995] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli et R.Y. Wang. Serverless Network File Systems. Dans *ACM press, Fifteenth ACM Symposium on Operating Systems Principles,* volume 29, numéro 5, pp 109-126. Decembre 1995.

[Backer, Hartman, Kupfer, Shirriff, Ousterhout, 1991] M.G. Backer, J.H. Hartman, M.D. Kupfer, K.W. Shirriff et J.K. Ousterhout. Measurements of a Distributed File System. Dans *ACM press, Thirteenth ACM symposium on Operating Systems Principles, volume 25, numéro 5, pp 198-212. Octobre 1991.*

[Markatos, Dramitinos, Papachritos, 1995] A.P. Markatos, G. Dramatinos, K. Papachritos. Implementation and evaluation of a Remote Memory Pager. Institute of Computer Science (ICS), Foundation for Research & Technology - Hellas (FORTH). technical Report FORTH/ICS 129, Grèce. Mars 1995.

[Folliot, Sens, Raverdy, 1995] B. Folliot, P. Sens, P.G. Raverdy. Plate-forme de répartition de charge et de tolérance aux fautes pour applications parallèles en environnement réparti. Calculateurs Parallèles. Editions Hèrmes, volume 7, numéro 4, pp 345-366. 1995.

[Bretelle, Terracol, Folliot, Sens, 1996] B. Bretelle, T. Terracol, B. Folliot, P.Sens. Problématique du Placement Dynamique dans les Systèmes Répartis à Haut-Débit. Laboratoire MASI, Institut Blaise Pascal. Université Pierre et Marie Curie. RenPar'8, Bordeaux, France. 1996.

[Leffler, McKusik, Karels, Quarterman, 1990] S.J. Leffler, M.K. McKusik, M.J. Karels, J.S. Quarterman. The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley. 1990.

[Goodheart, Cox, 1994] B. Goodheart, J. Cox. The magic garden explained. The internals of UNIX System V Release 4. An open system design. Prentice Hall. 1994.

[McKusic, Bostic, Karels, Quarterman, 1994] M.K. McKusic, K. Bostic, M.J. Karels, J.S. Quarterman. The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley. 1994.

[Lecler, Potter, Desbarbieux, Wajsbürt, Greiner, 1996] J. Lecler, F. Potter, J.L. Desbarbieux, F. Wajsbürt, A. Greiner. PCI6DDC / Spécifications (rapport). Laboratoire MASI. Avril 1996.

[Potter, David, Fenyö, 1996] F. Potter, P. David, A. Fenyö. Interface Logicielle pour la machine M.P.C.. laboratoire MASI / PRiSM. 1996.

[IEEE 1355] IEEE draft std p1355 standard for heterogenoeous interconnect (hic). Technical report, Intitute of Electrical and Electronic Engineers, Inc, 345 East 47th Street New York NY 10017, USA, August 1994.