Cnam - Laboratoire Cedric

Mémoire Répartie Partagée

Eric Gressier-Soudan

Cnam - 1996

Plan

Introduction

Environnement Multiprocesseur

Environnement Faiblement Couplé

Conclusion

INTRODUCTION

Copyright CNAM

Pourquoi répartir les données

a) **améliorer les services** rendus par le système:

- performance / efficacité
- disponibilité
- tolérance aux pannes

b) meilleure **adéquation au modèle d'exécution** :

- le parallèlisme des processus[1]
- communication par variables partagées vs communication par messages
- objets

c) représentation/manipulation physique de **données abstraites** -> utilisation d'adresses virtuelles
- correspondance avec pages réelles

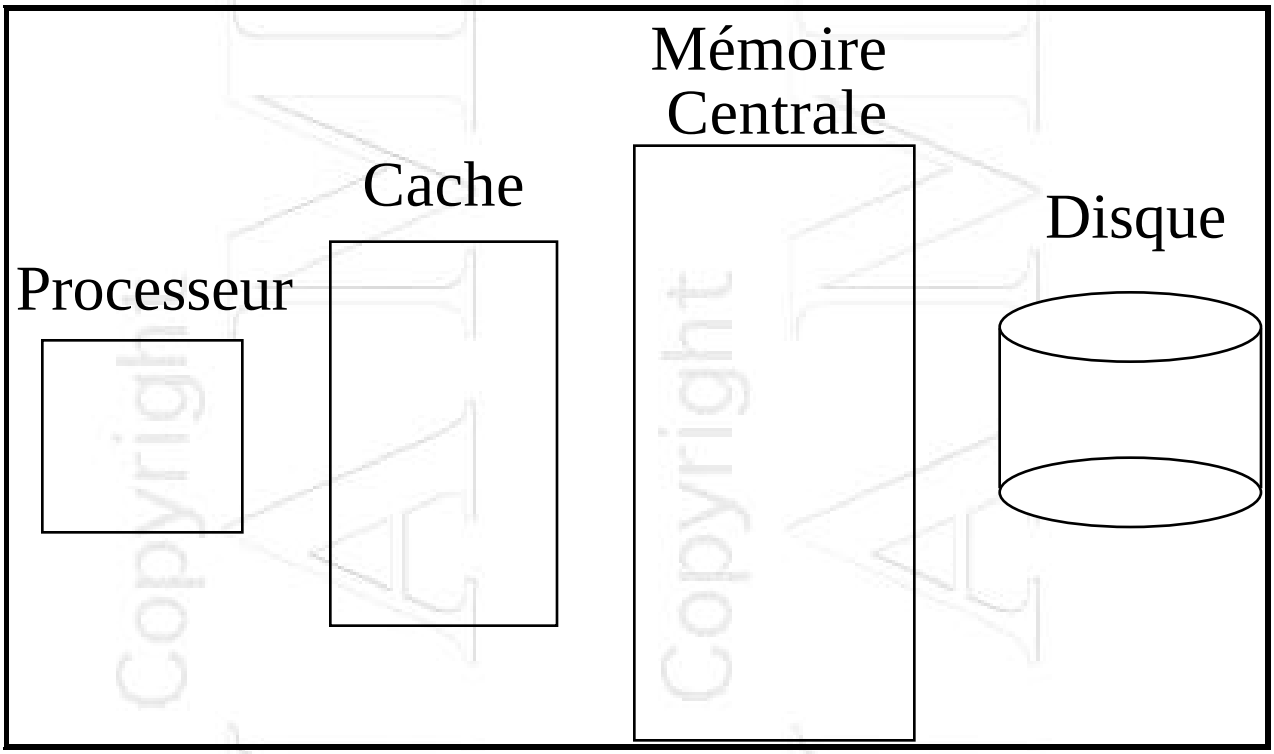
(déjà connu en univers centralisé)

offrir aux programmes **l'illusion**[2] qu'ils sont les seuls à utiliser une mémoire uniforme centralisée

[1] Migration de processus plus facile avec une Mémoire Virtuelle Répartie

[2] Toujours la propriété "Single System Image" vision uniformisée des ressources

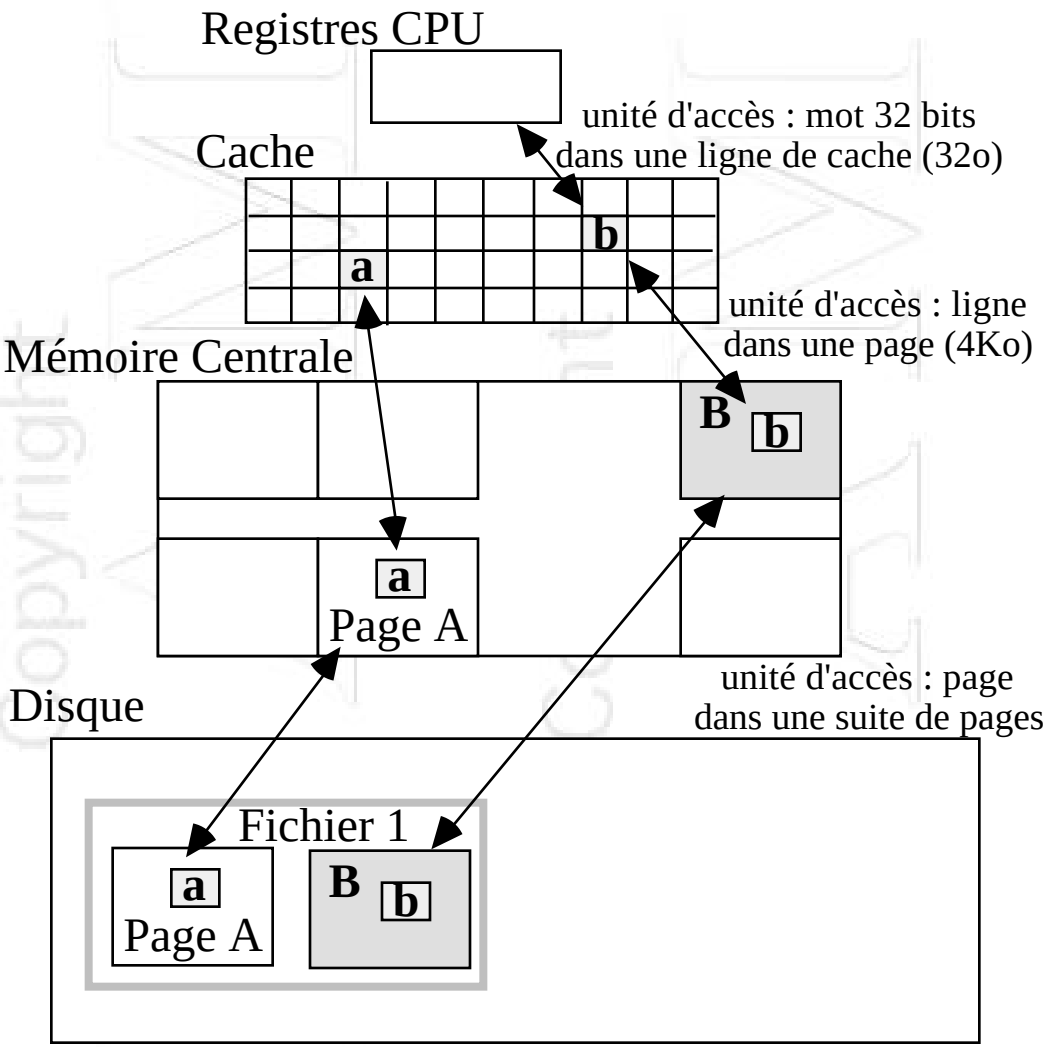
Hiérarchie de mémoire (1)



vitesse d'accès :

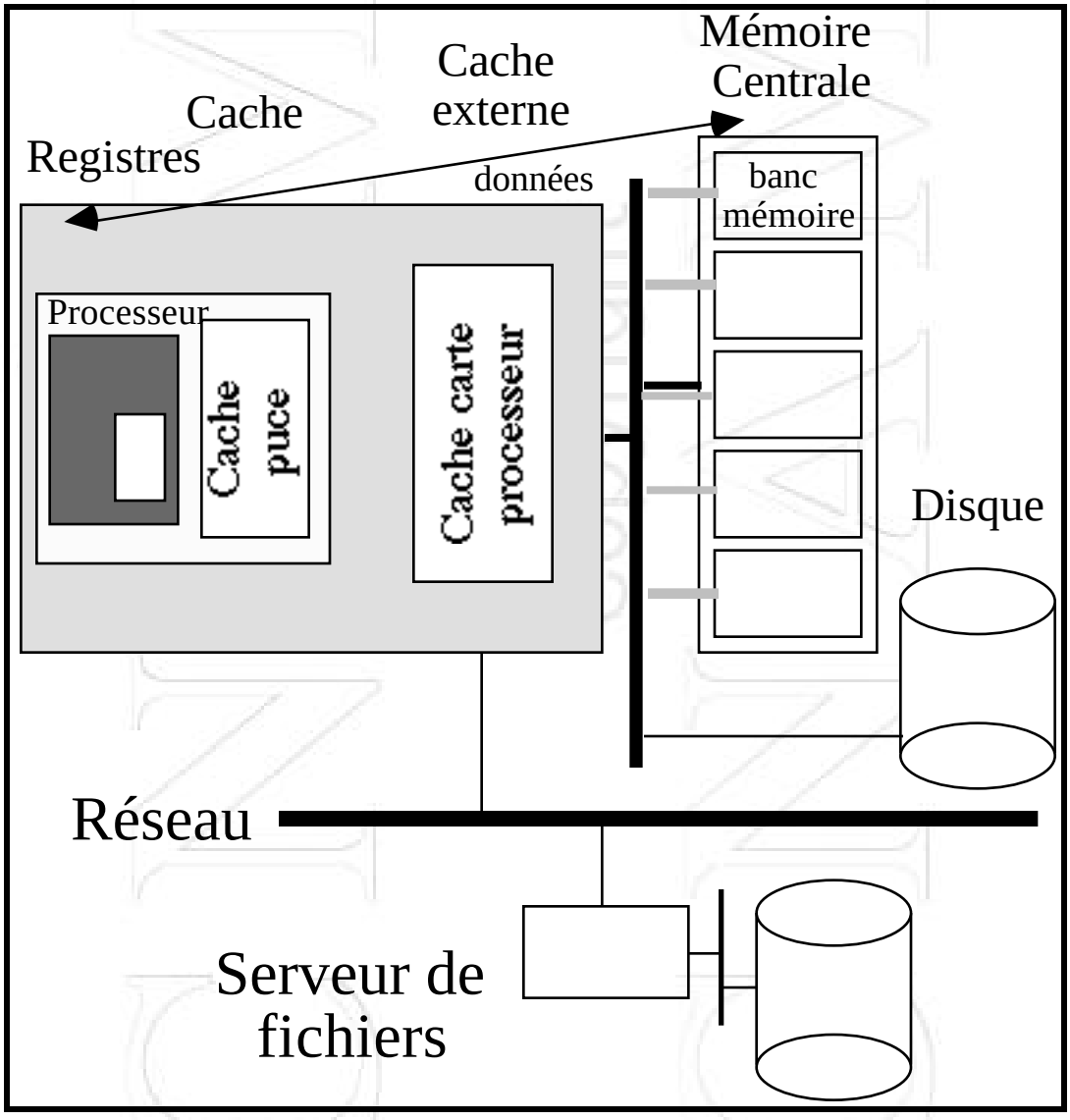
Processeur< Cache < Mémoire Centrale < Disque

Hiérarchie de mémoire (2)



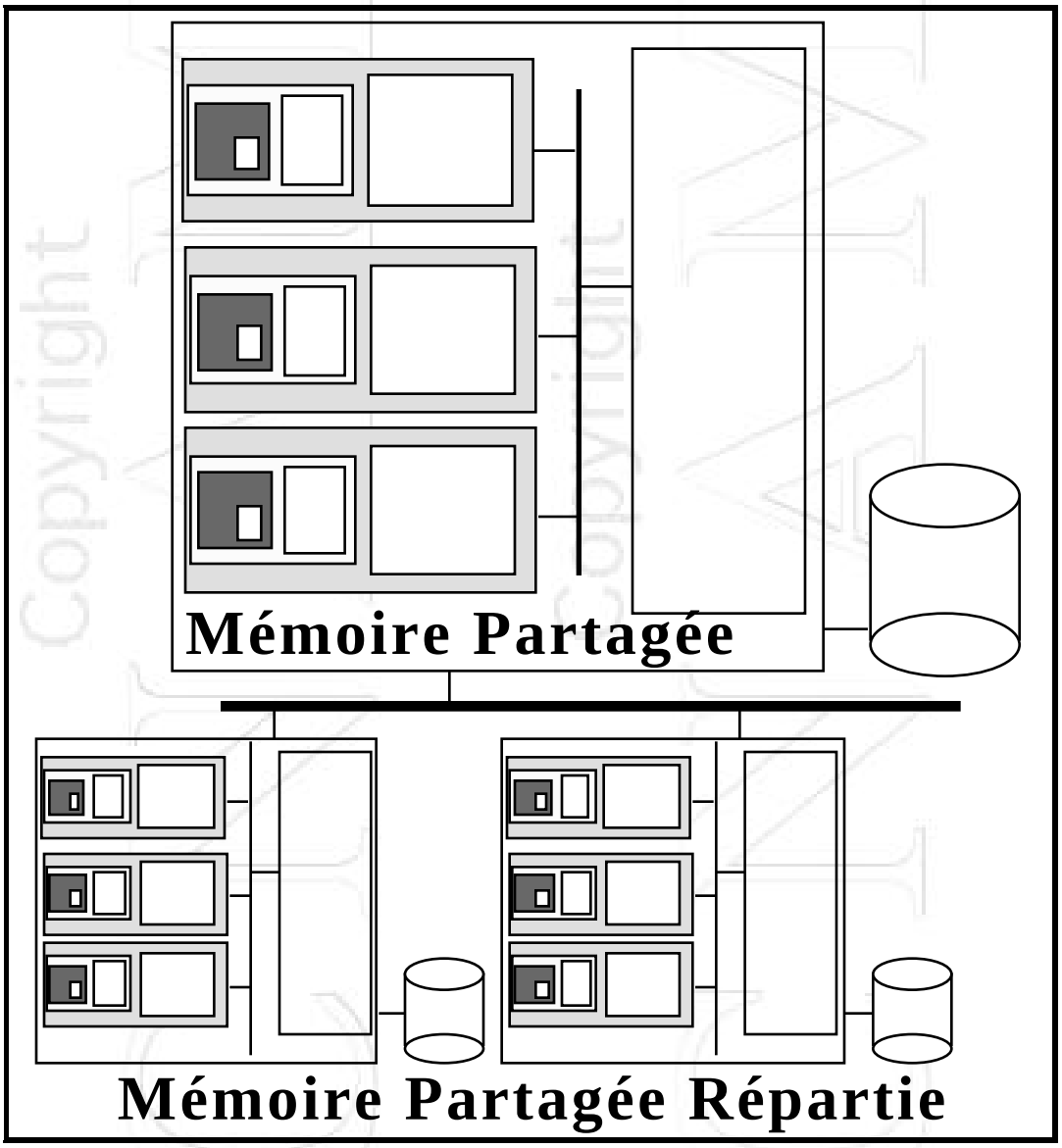
Problème de la répartition des données (1)

Problème en centralisé monoprocesseur ?



Problème de la répartition des données (2)

Il faut considérer le problème en multiprocesseur et en réparti !



. problème de copies multiples (caches/mémoires)

. de cohérence des copies

Orientation du cours

Nous ne traitons que des problèmes de gestion répartie des données au niveau de la **mémoire**.

. La **Gestion de Fichiers Répartis** concerne ce sujet mais est hors du propos de ce cours.

. La **Gestion de Transactions** pourrait aussi concerner ce sujet, mais elle ne sera pas traitée.

Architectures Considérées :

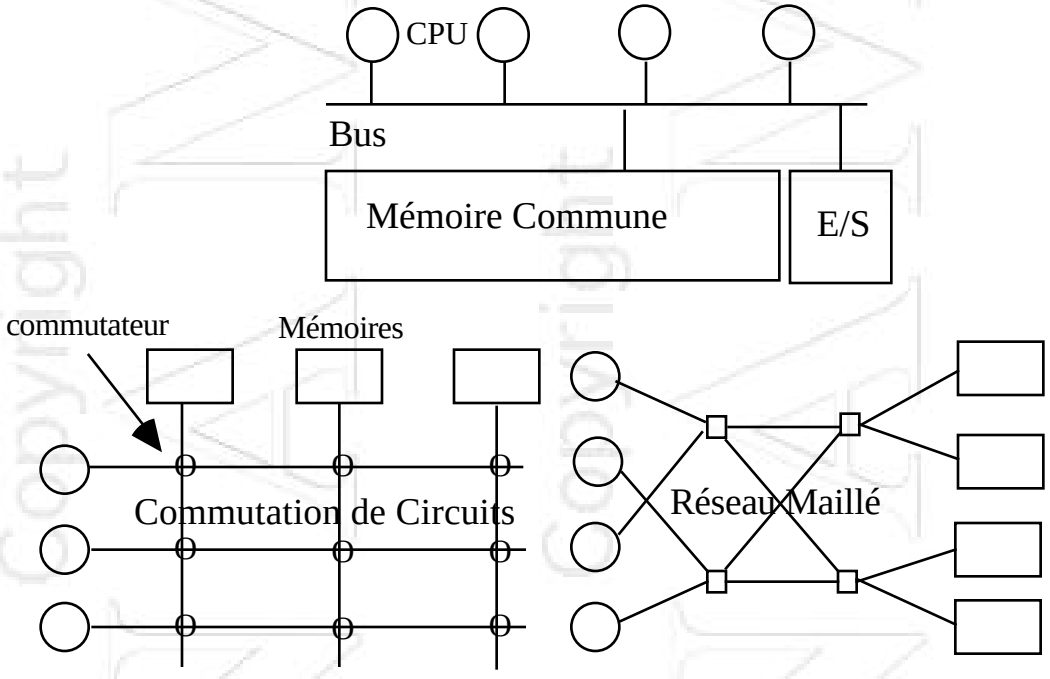
les architectures parallèles de type MIMD[3]

[3] Multiple Instruction streams over Multiple Data streams (flots d'instructions et de

Machines Fortement Couplées (1)

Multiprocesseur => la mémoire est partagée

. **Accès uniforme à la mémoire** (UMA[4]) : tous les processeurs accèdent à la mémoire physique, le temps d'accès à un mot mémoire est identique



multiprocesseur symétrique : les processeurs ont les mêmes capacités, ils peuvent exécuter le système et faire des E/S.

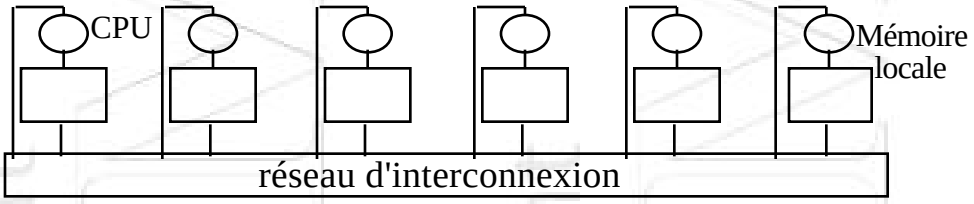
multiprocesseur asymétrique : certains processeurs sont spécialisés, un processeur exécute le système et fait des E/S, les autres sont dédiés à l'exécution de programmes utilisateur.

[4] Uniform Memory Access

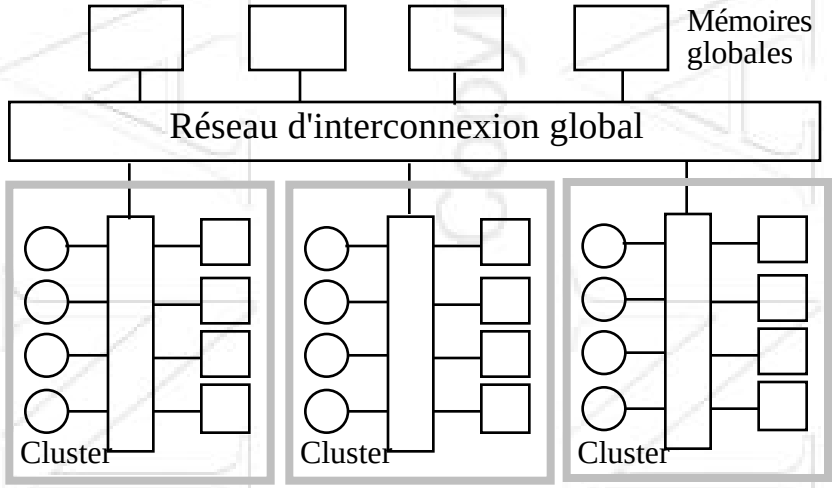
Machines Fortement Couplées (2)

. **Accès non uniforme à la mémoire** (NUMA[5]): le temps d'accès à la mémoire diffère en fonction de la localisation des données à cause de la traversée du réseau d'interconnexion

Les mémoires locales sont adressables par les processeurs distants, elles forment un espace d'adressage global.



multiprocesseur BBN Butterfly



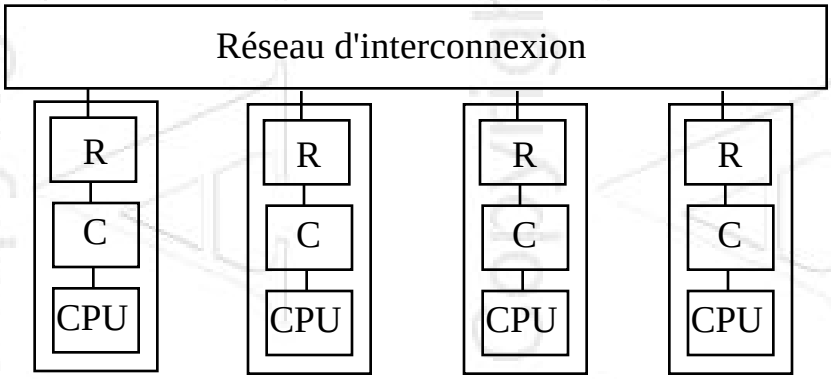
multiprocesseur Cedar (Université Illinois)

[5] Non Uniform Memory Access

Machines Fortement Couplées (3)

. **Architecture à mémoire cache uniquement** (COMA[6]): c'est un cas particulier de machine NUMA

L'accès à un cache distant utilise un mécanisme de répertoire. Le répertoire contient pour chaque bloc des informations sur son état et sa localisation. Il peut y avoir une hiérarchie de plusieurs niveaux de répertoires.



C : Cache
R : Répertoire

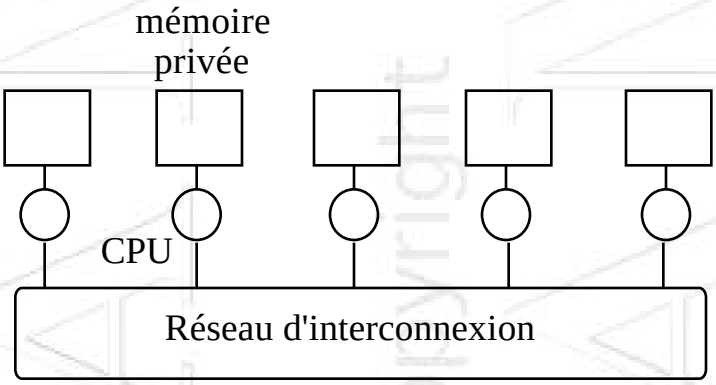
[6] Cache Only Memory Architecture

Machines Faiblement Couplées

multiordinateur -> processeurs en réseau

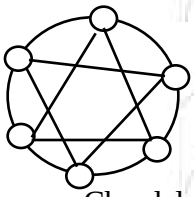
la communication entre processeurs s'effectue uniquement par messages, les mémoires sont privées et ne sont accessibles que par le processeur local

pas d'accès à distance à la mémoire (NORMA[7])

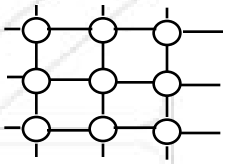


Réseau :

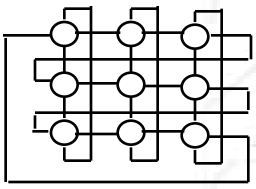
Réseau Local, Anneau, Anneau "Chordal", Arbre binaire, Etoile, Hypercube, Réseau maillé, Tore, Mesh...



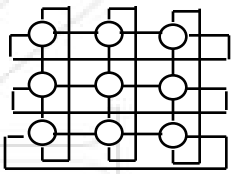
Anneau Chordal de degré 2



Mesh 3x3



Mesh (Illiac) 3x3



Tore 3x3

---

[7] No Remote Memory Access

Principes de la répartition des données en mémoire

1. L'efficacité de la répartition des données repose sur la **propriété de localité des références** aux informations en mémoire.

-**Localité dans le temps:** Pendant un certain temps, les mêmes informations seront accédées.

-**Localité dans l'espace:** Les références faites par un programme, données comme instructions, se trouvent dans des zones d'adresses proches.

"espace de travail - *Working Set* "

2. Gestion de la concurrence des accès :

-> ajout de mécanismes de synchronisation à côté du partage de la mémoire

-> sémantique de cohérence des copies plus ou moins forte

Mémoire virtuelle répartie partagée (1)

Mémoire virtuelle pour offrir un espace d'adressage plus grand que la mémoire physique disponible.

On considère une mémoire virtuelle lorsqu'on travaille en univers réparti (multiordinateur) :

l'espace d'adressage est global à un ensemble de processeurs, il recouvre les espaces d'adressages virtuels privés pouvant être associés à chaque processeur.

-> Mise en œuvre Matérielle :

Unité de gestion mémoire (MMU[8])

-> Mise en œuvre Logicielle :

Gestionnaire de Mémoire(externe au noyau) -> . indépendance vis à vis du matériel . répartition facile .sémantique de la cohérence spécifique à l'utilisateur
Micro-Noyau
Hardware - MMU

[8] Memory Management Unit

Mémoire virtuelle répartie partagée (2)

Convergence des Problèmes quelle que soit la base des solutions : matérielle ou logicielle

=

Algorithmique répartie

Environnement Multiprocesseur

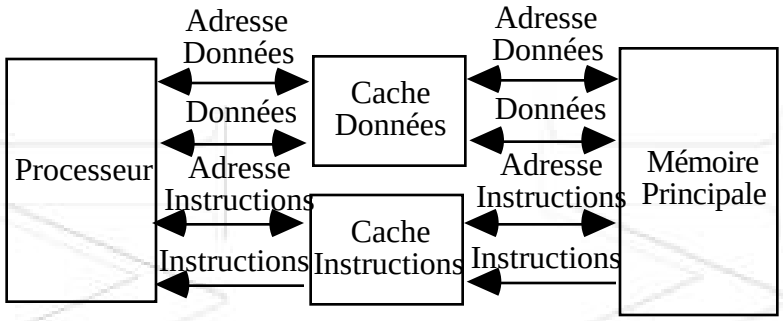
Objectifs de la conception d'un cache

- maximiser la probabilité de trouver une référence (***hit*** [9]***ratio***) dans le cache
- minimiser le temps d'accès à l'information
- minimiser le délai introduit par un défaut (***miss***) dans le cache
- minimiser le temps de mise à jour de la copie de référence (la mémoire)

invisible à l'utilisateur ou aux programmes

[9] "hit", succès lors d'une requête au cache, au contraire de "miss", qui est employé

Points principaux d'un cache



1. Les données sont accompagnées d'une **marque/étiquette** "tag". Un tag contient l'adresse de l'information archivée dans le cache.

Lorsque le processeur effectue un accès à une variable en mémoire, l'adresse de cette information est envoyée au cache, une recherche s'effectue alors, l'information est retrouvée grâce à sa marque, si elle est présente dans le cache.

Si l'adresse ne correspond à aucun tag, l'adresse est passée à l'extérieur pour être servie soit par la mémoire principale, soit par un autre cache dans le cas d'un multiprocesseur. Au retour, l'information est stockée dans le cache.

2. L'unité mémoire de gestion d'un cache est une **ligne** ou un **bloc** (16 à 32 octets voire 128 et même 256 octets pour mieux bénéficier des effets de localité), il y a une marque associée à chaque ligne.

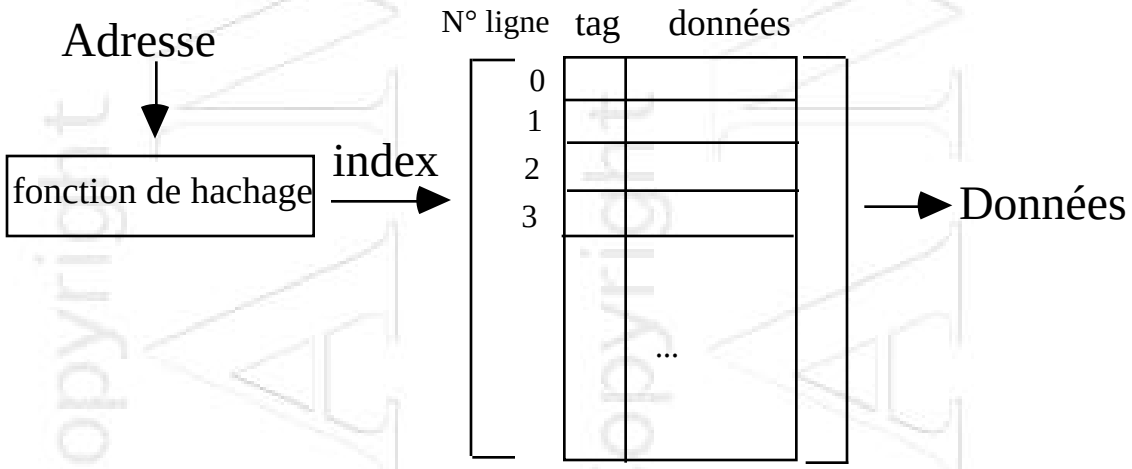
ligne de cache = tag + information

<bit de validité, bit de modification, [clé], adresse de l'information>
<information>

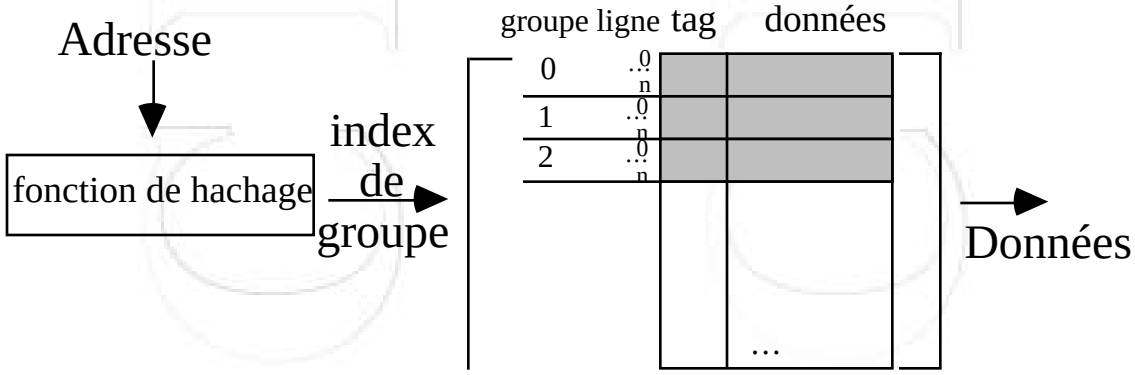
bit de validité : indique si la ligne contient des informations en service, valides
bit de modification : indique si la ligne a été modifiée depuis son chargement dans le cache, ce bit sert pour la mise à jour d'autres caches ou de la mémoire principale suivant la stratégie de cohérence choisie.

Correspondance Cache / Mémoire Centrale

1. Caches à correspondance directe



2. Caches associatifs



Mécanismes de base d'un cache

1. Accès aux données

2. Cache avec adresses virtuelles ou physiques

3. Cache données-cache instructions

4. Stratégie de remplacement des informations dans le cache / vidage du cache

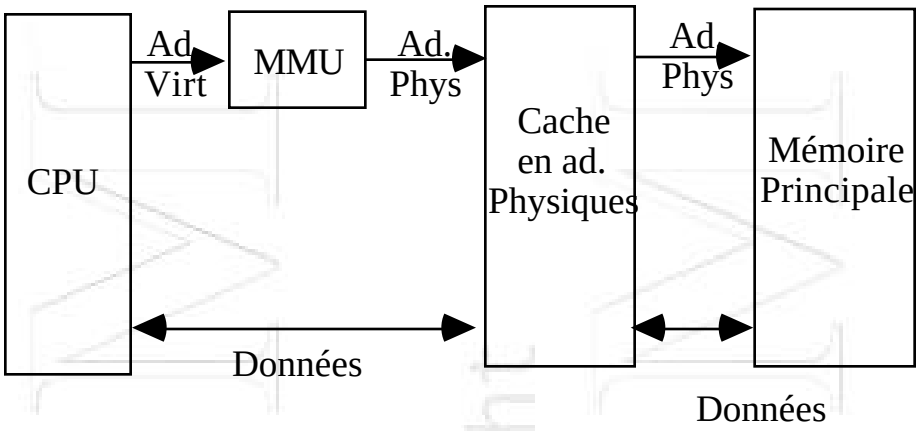
5. Mise à jour de la mémoire : cache à **mise à jour immédiate**[10] ou cache à **mise à jour retardée**[11]

6. Intégration des opérations d'E/S avec la cohérence du cache

[10] fonctionnement du cache en mode Write-through

[11] fonctionnement du cache en mode Write-back

Caches sur adresses Physiques

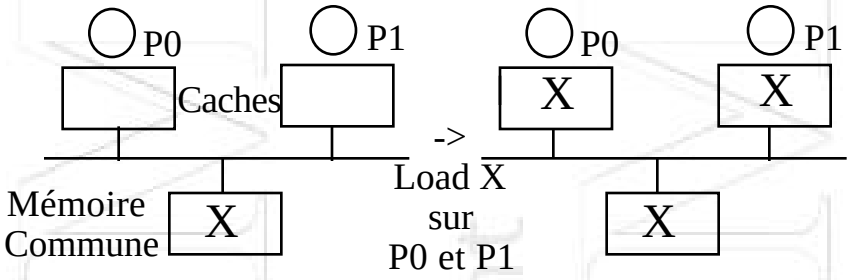


L'adresse physique est calculée à chaque accès.

Type de cache réservé au caches hors puce processeur.

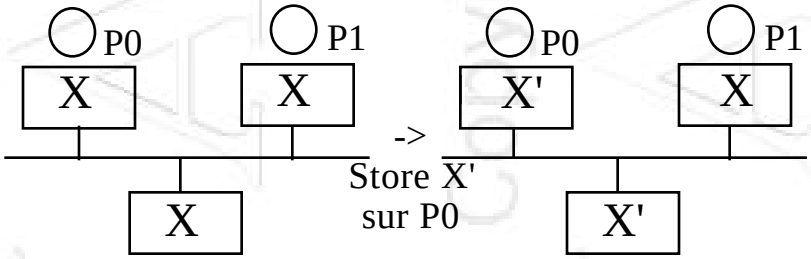
Faiblesse des politiques de mise à jour cache-mémoire pour le maintien de la cohérence

Situation initiale :



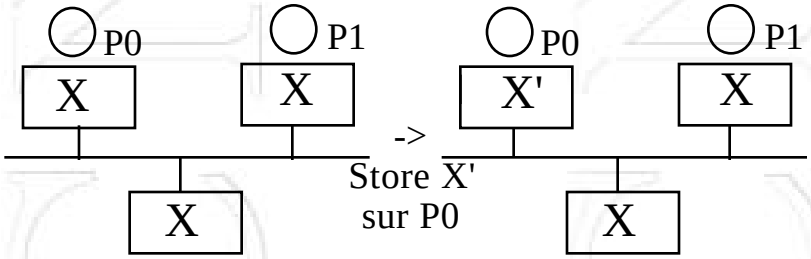
Les copies sont cohérentes

Mise à jour immédiate :



Les copies sont incohérentes

Mise à jour retardée :



Les copies sont incohérentes

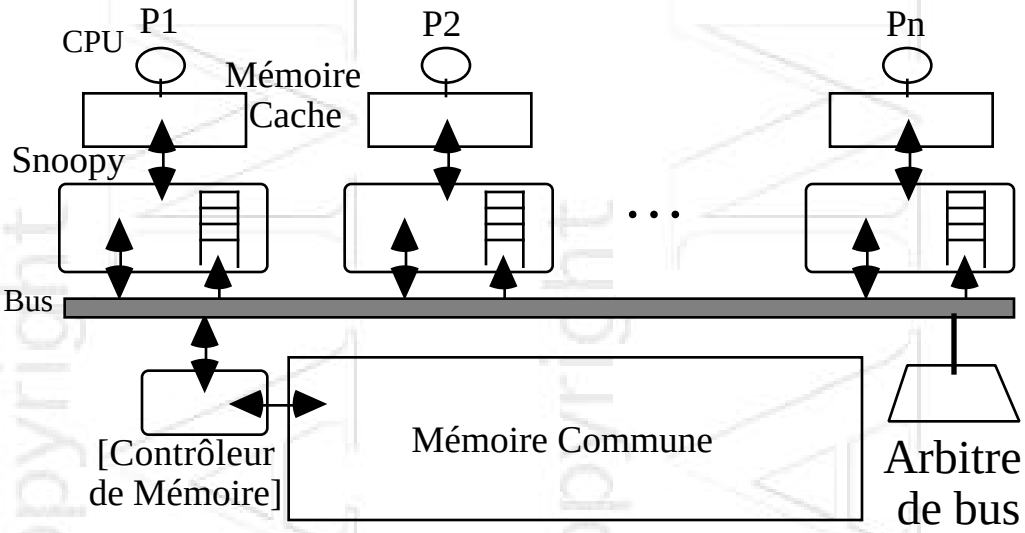
Mécanismes Insuffisants pour le maintien de la cohérence

Protocoles Multiprocesseur par espionnage

Exemple :Firefly

Copyright CNAM

Le contrôleur de cache espion - Snoopy Cache Controler

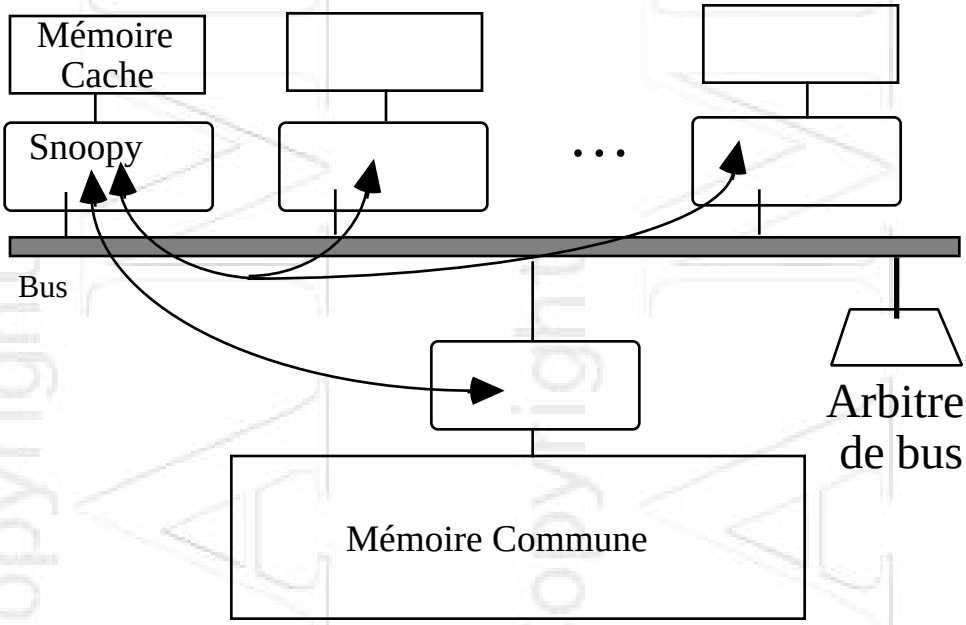


Le processeur cherche la donnée dont il a besoin dans le cache, s'il ne la trouve pas, il est mis en attente jusqu'à ce que celle-ci y soit amenée depuis un autre cache ou depuis la mémoire commune.

Principe :
Le contrôleur de cache gère le protocole de cohérence pour un processeur, il maintient la cohérence en espionnant les transactions sur le bus effectuées par les autres processeurs.
L'espionnage porte sur le tag des données contenues dans le cache.

Protocole de maintien de la Cohérence (1)

Le maintien de la cohérence se fait par coopération des différents contrôleurs:



L'arbitre de bus a un rôle prépondérant puisqu'il attribue le bus à un processeur ou à un autre quand il y a conflit.

Les protocoles sont dépendants de la stratégie de mise à jour de la mémoire commune adoptée pour les caches.

Protocole de maintien de la Cohérence (2)

Deux protocoles :

1. Protocoles à **invalidation sur écriture** :

Lorsqu'un processeur modifie une donnée toutes les copies de cette donnée sont invalidées, seul l'écrivain possède une copie à jour.

Repose sur la notion de propriétaire (dernier écrivain sinon mémoire commune)

Protocole Berkeley [4]

2. Protocole à **diffusion des écritures** :

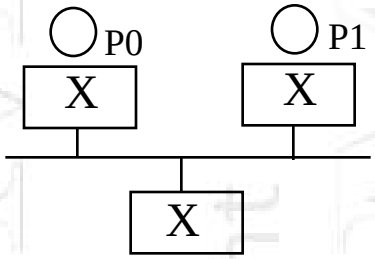
A chaque écriture sur une copie, toutes les autres copies présentes dans les autres caches sont mises à jour.

La mémoire commune est mise à jour en fonction de la gestion de cache adoptée : à chaque écriture (write-through), au vidage du dernier cache (write-back).

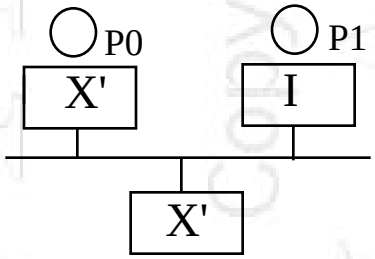
Protocole de maintien de la Cohérence (3)

Multiprocesseur avec caches de type mise à jour immédiate

Situation initiale :

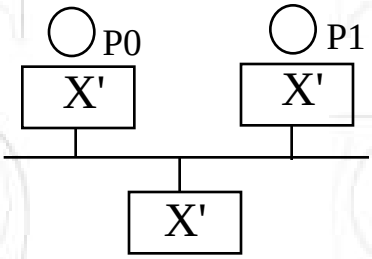


Effet d'une invalidation sur écriture lors de la modification de X en X' sur P0 :



I pour invalide

Effet d'une diffusion de l'écriture lors de la modification de X en X' sur P0 :



NB : La mémoire centrale est mise à jour à cause de la politique write-through

L'algorithme du Firefly - DEC [5] (1)

2 stratégies simultanément:

. Mise à jour retardée pour les données non partagées

. Mise à jour immédiate pour les données partagées et donc Diffusion des Ecritures

économise l'utilisation du bus, et tente un bon compromis entre performance et maintien de la cohérence entre la mémoire et le cache

L'algorithme du Firefly (2)

Etats d'un Bloc dans un cache :

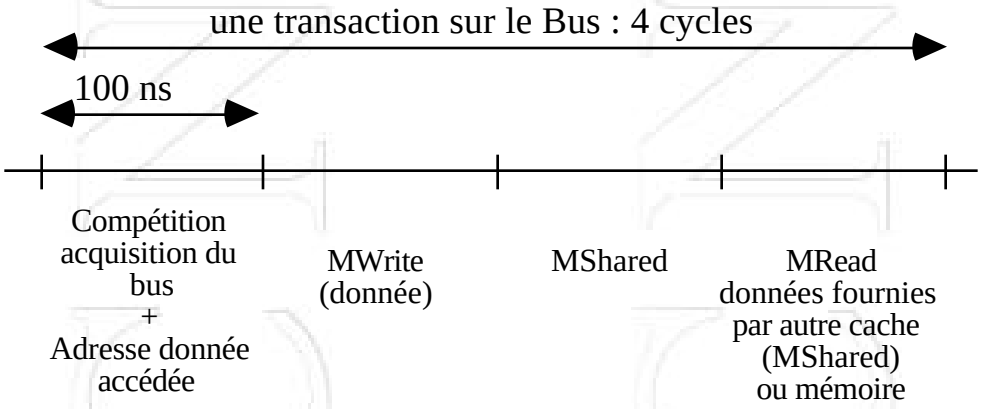
1 bit SHARED : partagé

1 bit DIRTY[12] : modifié

Important :

Un signal sur le bus, "**MShared**", sert à détecter si un bloc est partagé par d'autres caches. Quand un cache effectue une transaction sur le bus, les caches qui possèdent une copie de la donnée associée à cette transaction positionnent Mshared pour indiquer qu'ils la partagent.

La gestion du bus est faite de telle façon que tous les caches puissent répondre pendant un cycle.



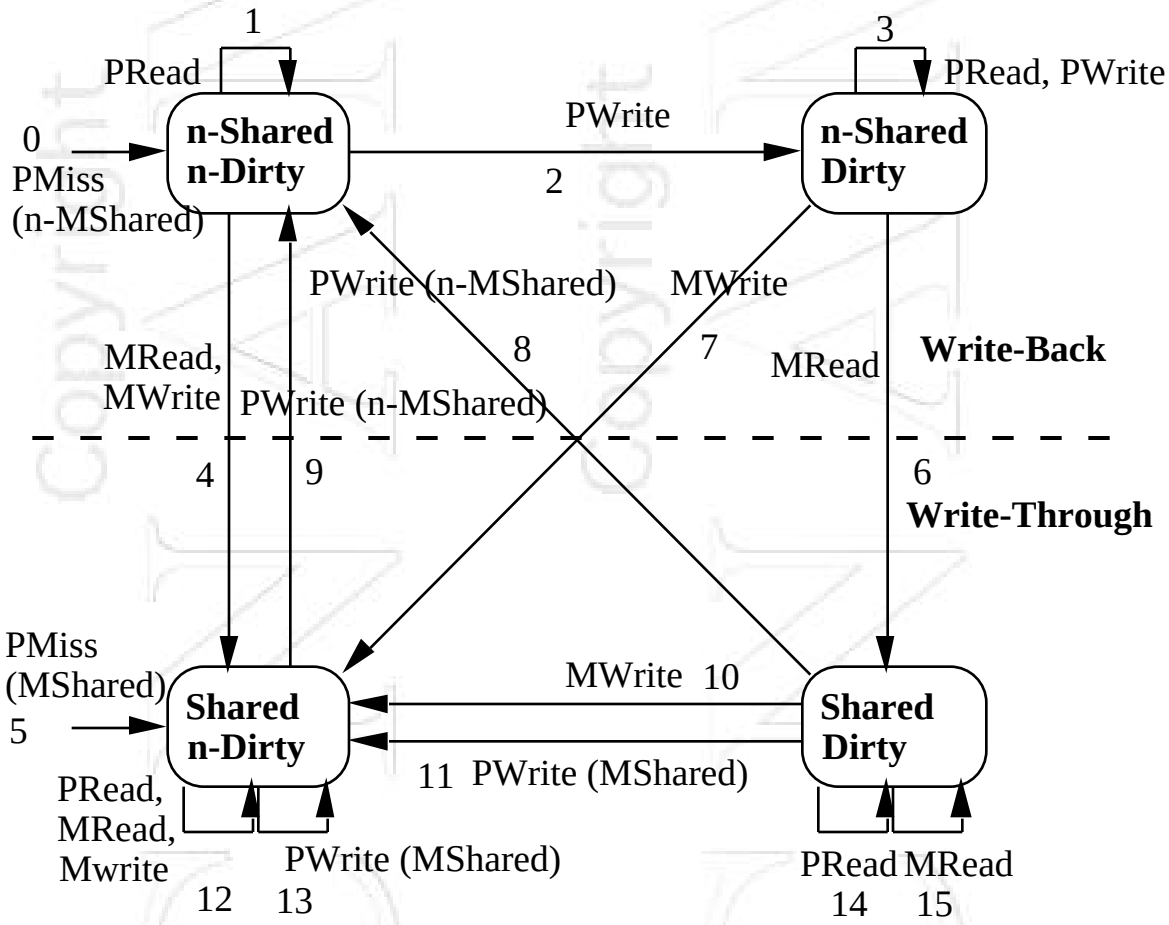
[12] seul type de bloc qui peut-être recopié en mémoire commune, lorsque le cache du

L'algorithme du Firefly (3)

Les états-changements d'états sont causés soit :

- par le processeur associé au cache (**P**),
- par des transactions observées sur le bus mémoire (**M**).

Situations observées :



"n-X" veut dire "not X"

L'algorithme du Firefly (4)

Actions liées à la scrutation du bus :

Lorsqu'un contrôleur de cache détecte une lecture sur le bus (MRead), et qu'il possède une copie chez lui, il sert la copie au demandeur[13].

Lorsqu'un contrôleur de cache détecte une écriture (MWrite), il met à jour sa copie, pendant le cycle suivant il positionne MShared.

Vidage du cache :

Lors d'un défaut (Read Miss ou Write Miss), s'il y a une place vide dans le cache, la donnée est chargée à cet endroit. S'il n'y a pas de place vide, la cellule sélectionnée (cellule victime) par l'algorithme de remplacement est d'abord recopiée dans la mémoire commune avant que la lecture ne soit terminée. Cette recopie en mémoire n'a lieu que si ce processeur est le dernier à avoir une copie de la cellule dans son cache.

Ecriture sur un bloc partagé :

La modification n'est pas faite tant que le contrôleur de cache n'a pas pu diffuser sa mise à jour, ce qui suppose l'acquisition du bus.

---

[13] Plusieurs caches peuvent servir la requête simultanément ... le résultat sur le bus est bon puisque toutes les copies sont identiques d'après le protocole de gestion de la

L'algorithme du Firefly (**Hypothèses de fonctionnement d'après [5]**)

a. L'unité de donnée sur laquelle s'applique le protocole est une cellule ("line" dans la terminologie de l'article). Elle "mesure/pèse" 4 octets soit un mot mémoire VAX.
b. Lors d'un défaut, si la donnée demandée n'est dans aucun cache, la mémoire commune fournit la donnée. Si elle est présente dans un ou plusieurs caches, elle est écrite sur le bus de données pendant le 4ème slot du cycle de transaction-bus. Tous ceux qui détiennent un exemplaire écrivent ... ils fournissent la même valeur à cause du protocole de cohérence. La mémoire commune est inhibée et ne peut servir la donnée.
c. Les données non partagées accédées en lecture comme en écriture le sont dans le cache local. Dans ce cas, aucune action sur le bus n'est provoquée... en particulier, il n'y a pas d'acquisition du bus par le "snoopy" pour informer les autres caches des opérations qu'il est en train d'effectuer. La cellule ne sera recopiée en mémoire que lorsqu'elle quittera le cache pour faire de la place (politique write-back).
d. Les données partagées (présentes simultanément dans plusieurs caches) sont accédées en lecture dans le cache local au processeur. Dans le cas d'une écriture, le cache diffuse son écriture aux autres caches. L'écriture n'a d'effet localement qu'après le succès de la diffusion de la nouvelle valeur (hypothèse d'atomicité) Compte tenu du fonctionnement des transactions sur le bus, la diffusion de l'écriture a lieu pendant le 2ème cyle. Si l'écrivain doit découvrir qu'il ne partage plus la cellule, il ne peut le faire qu'au 3ème cycle. Si tel est le cas, il modifie le bit de partage en indiquant qu'elle n'est plus partagée. Il passe en politique write-back. Les opérations en mode partagé sont toujours vues sur le bus.
e. Le déclenchement de la politique write-through est lié à la valeur du bit Shared et **à l'opération d'écriture proprement dite**.

L'algorithme du Firefly (Transitions de la figure précédente)

0. Défaut, qq soit la nature du défaut la page mémoire commune était à jour. La cellule est donc non-modifiée par rapport à la copie de référence en mémoire commune.
1. Le processeur effectue une lecture sur une cellule non paratagée, rien ne change.
2. Il effectue une écriture, la cellule est donc modifiée par rapport à la copie de référence en mémoire commune.
3. Pas de modification par lecture ou écriture, la copie est déjà modifiée.
4. Le contrôleur de cache (snoopy) a détecté qu'un autre cache demandait la copie qu'il détient pour la lire ou pour écrire ... on passe de la politique de mise à jour retardée (write-back) à mise à jour immédiate (write-through) ... la cellule n'était pas modifiée, pas de pb.
5. Défaut sur une page partagée par d'autres processeurs.
6. La copie était modifiée, le snoopy détecte une lecture, c'est nécéssairement un défaut puisqu'il était le seul à avoir la copie ... il fournit sa version au cache demandeur. La cellule devient partagée, la mémoire commune n'a pas encore été mise à jour, elle le sera à la prochaine écriture à cause de **la politique write-through qui pourra être appliquée seulement à partir de ce moment** (pour écrire en mémoire il faut une écriture effective)... Soit deux caches voient la même cellule dans deux état différents : l'un Shared-Dirty, l'autre Shared-n-Dirty ... Il faut remarquer dans la description du protocole Firefly [3], que l'état Shared-Dirty n'existait pas !!!
7. La copie était modifiée, et non partagée ... le snoopy a détecté qu'un autre cache veut la page pour y effectuer une écriture, il va lui fournir sa copie, mais l'hypothèse 3 implique une mise à jour de la copie de référence en mémoire commune.

8. La copie est partagée, la copie est écrite localement, la mémoire commune est mise à jour, le snoopy détecte que plus personne ne la partage, qu'il est le seul à détenir un exemplaire. La cellule passe à l'état non partagé, et non-modifiée.
9. idem
10. Première écriture, effectuée par un autre processeur. C'est détecté lors de la transaction bus ... la politique write-through implique la mise à jour de la version de référence dans la mémoire commune.
11. Idem, sauf que l'écriture est locale.
12, 13. Lecture locale, Ecriture locale avec poursuite du partage, lecture ou écriture dans un autre cache ne changent pas l'état de la cellule.
14,15. Lecture locale, lecture dans un autre cache ne déclenche pas la mise à jour de la copie de référence en mémoire commune ... l'état de la cellule ne change pas.

Copyright CNAM

Faiblesses des solutions à base d'espions

- mauvaise extensibilité : la bande passante du bus est limitée, 32 processeurs ... c'est déjà beaucoup

- suivant le protocole :

    * trop de diffusions qui vont mettre à jour des données qui ne seront pas utilisées.

    * les invalidations d'une ligne vont provoquer des défauts de lecture.

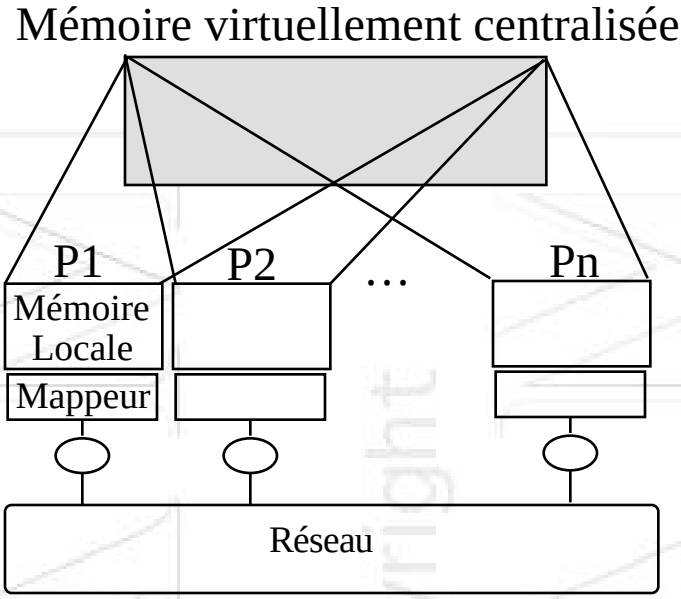
Ces deux derniers problèmes peuvent être agravés si le grain des lignes/blocs est trop grand, en effet des mots peuvent être en mémoire et ne pas être utilisés.

On veut viser plusieurs centaines de processeurs.

solutions à base de répertoires qui contiennent des informations de localisation sur les lignes/blocs.

Environnements Faiblement Couplés

Hypothèse d'architecture



Architecture de type NORMA :

. On raisonne en adresses virtuelles

. Communication par message fiable

. L'accès s'effectue à l'échelle de la page

Objectif :

Offrir l'abstraction d'une mémoire commune masquant les mémoires privées.

Origine de la solution

Algorithme de gestion de la mémoire virtuelle répartie partagée établie par Kai Li et Paul Hudak [6], solution de type **Cohérence Forte**

fondé sur la notion de **propriétaire**,
utilisant les mécanismes d'**invalidation sur écriture**[14].

=> **La diffusion des écritures est beaucoup trop coûteuse**

3 solutions :
- . Gestionnaire centralisé
- . Gestionnaire avec répartition statique
- . Gestion Répartie Dynamique

Les auteurs proposent des optimisations.

[14] Adaptation du protocole de Berkeley [4] pour multiprocesseur à mémoire

Gestionnaire Centralisé (1)

1 site, prédéfini, assure la gestion de la cohérence pour l'ensemble des pages

Structures de Données (version optimisée):

- pour chaque site :

Table des Pages

N° page	verrou	{sites avec une copie}	droits d'accès

verrou : pour verrouiller les accès locaux/distants à une page
droit d'accès : lecture, écriture, invalide
{sites avec une copie}[15] : n'a un sens que si le site est propriétaire de la page

- pour le Gestionnaire :

Info

N° page	propriétaire

Verrou Gestionnaire

propriétaire : N° du site écrivain le plus récent
Verrou-Gestionnaire : pour verrouiller les demandes des différents sites

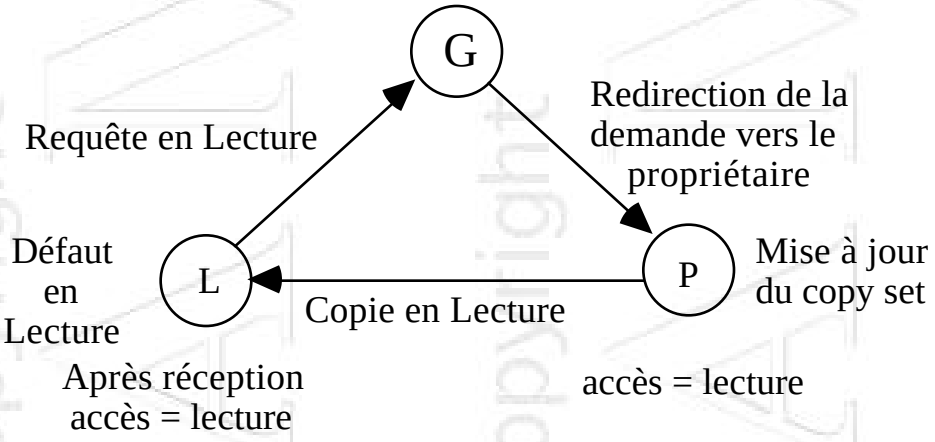
Un verrou n'est relaché qu'une fois l'opération d'accès à la page correspondante terminée, c'est à dire qu'un exemplaire de la page est arrivé sur le site.

[15]

Gestionnaire Centralisé (2)

Protocole

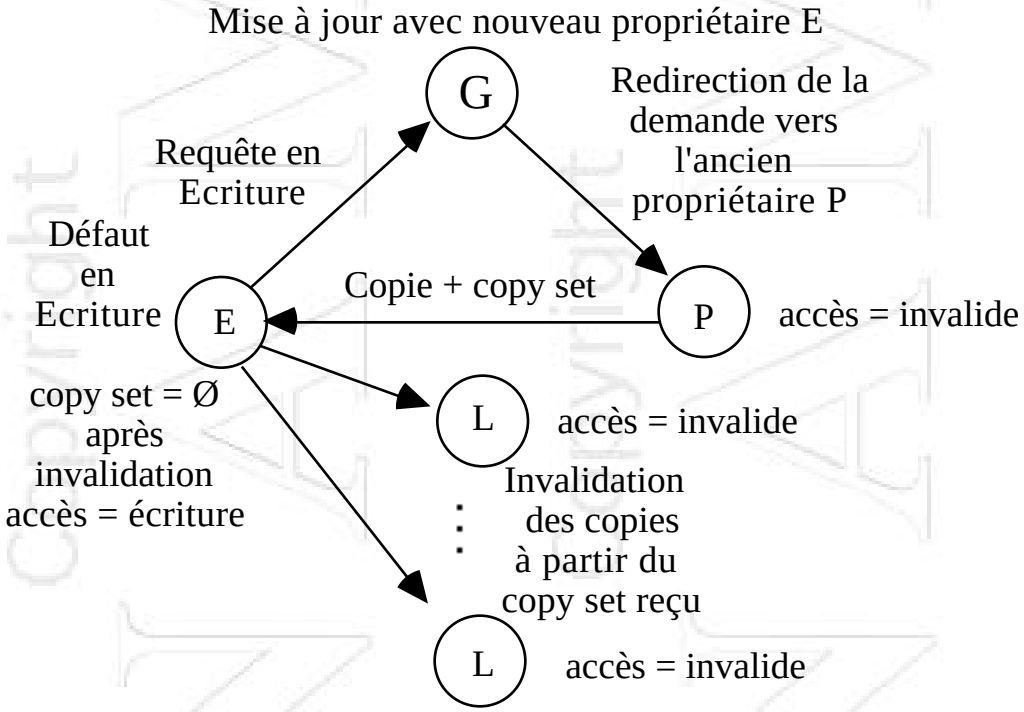
Défaut en Lecture :



Gestionnaire Centralisé (3)

Protocole

Défaut en Ecriture :



Extension du Gestionnaire Centralisé au Gestionnaire Réparti Statique

Chaque processeur est le gestionnaire d'un ensemble prédéfini de pages (statique).

Lorsqu'un accès est effectué sur une page p, la requête est soumise au processeur gestionnaire correspondant,

le numéro du processeur est obtenu par une fonction **f (p , N)**

où N est le nombre de processeurs qui participent à la gestion de la mémoire virtuelle répartie.

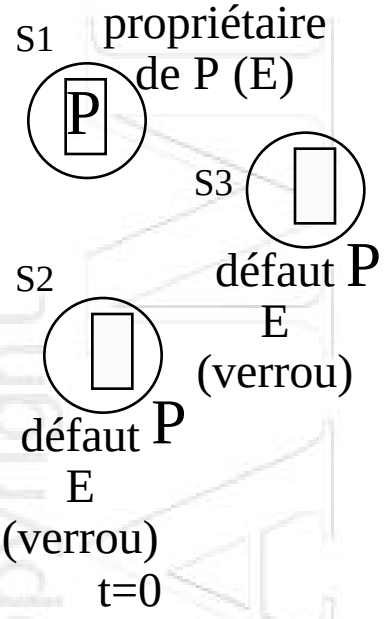
Gestion Répartie avec Diffusion

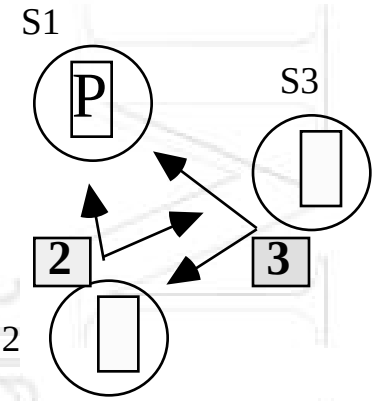
idée : LAN => utiliser les possibilités de diffusion du support de communication pour invalider ou localiser un propriétaire de page (Broadcast ou Multicast)

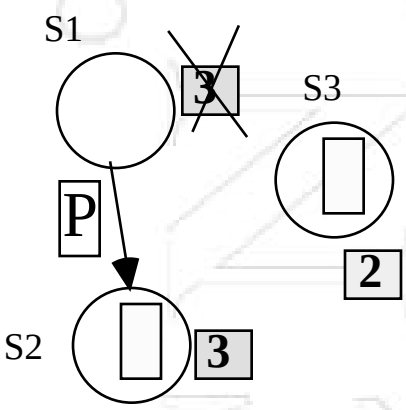
problème : diffusion des demandes, on peut être amené à traiter des demandes trop anciennes et qui n'ont plus de sens

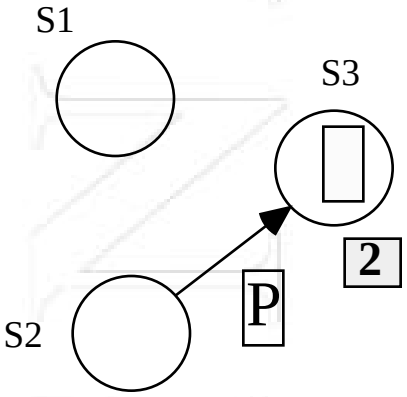
solution : ordonner les demandes et les réponses par une Diffusion Ordonnée ou des Vecteurs d'horloge sur le contenant des pages

Gestion Répartie avec Diffusion : Problème









S3 envoie la page à S2, cette page ne correspond plus à aucune demande en cours, la requête de S2 par rapport à la réception de la page P est un évènement trop ancien, il aurait du l'oublier.

Gestion Répartie Dynamique (1)

La fonction de gestionnaire est répartie et dynamique

Structure de données d'un site :

Table des Pages

N° page	verrou	{sites avec une copie}	propr probable	droits d'accès

Notion de **propriétaire probable** : le site n'a qu'une estimation du véritable propriétaire de la page (suggestion - *hint*).

Le *copyset* n'a de sens que quand on est le véritable propriétaire de la page.

Gestion Répartie Dynamique (2)

Défaut de page (écriture ou lecture) :

Le processeur envoie sa requête au propriétaire probable de la page qui est indiqué dans sa table.

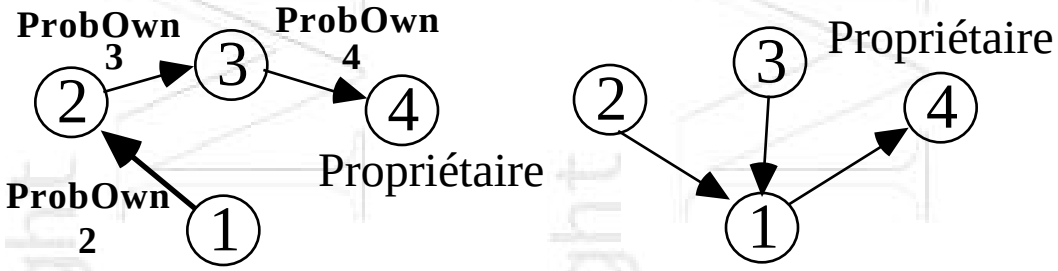
- Si le propriétaire probable n'est pas le vrai propriétaire, il propage la requête.

- Sinon il est le vrai propriétaire, il fait comme dans l'algorithme précédent :
    . en lecture il modifie l'accès, met à jour le copy set, et envoie une copie,
    . en écriture il invalide localement, et envoie la page avec le copyset au demandeur, il mémorise que ce dernier devient le propriétaire probable.

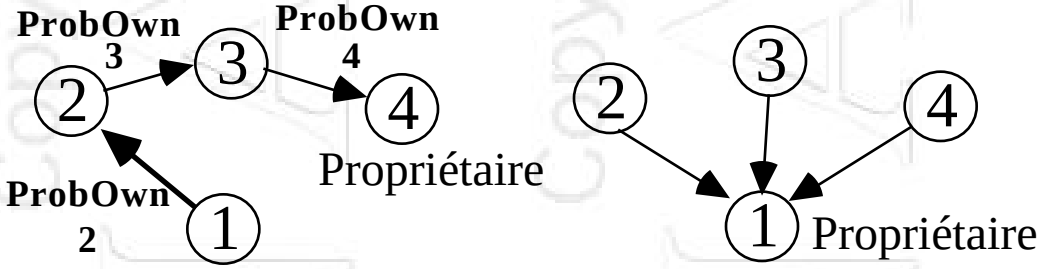
Sur réception de la page, le demandeur invalide et devient le vrai propriétaire, le copyset est vidé.

Gestion Répartie Dynamique (3)

Modification du champ propriétaire probable : Sur réception d'une demande d'écriture, de lecture (sauf pour le propriétaire), ou d'invalidation -> le demandeur



Défaut de page en lecture de 1



Défaut de page en écriture de 1

Problème : Borner le nombre de sites parcourus pour trouver la page requise.

Au pire N-1, au mieux 2[16], en moyenne O(N + K log N) après K demandes d'une page. En général pas plus de deux processeurs partagent la même page.

[16] Juste après une recheche du propriétaire qui a parcouru toute la chaine des propriétaires probables. Les champs de la table des pages de ces sites a été mis à

Algorithme [6]:

La mémoire fonctionne sur le principe d'une mémoire paginée. Lors d' un défaut de page, le processeur s'adresse à un **handler** qui recherche la page demandée auprès des autres processeurs du réseau. Le **mappeur** est la partie du système attachée à un processeur qui répond aux demandes de pages provenant des autres processeurs du réseau. Les mappeurs communiquent entre eux pour maintenir la cohérence entre les différentes copies des pages.

Les communications sont fiables: pas de perte de messages, pas de duplication de messages, pas de corruption de messages, pas de déséquencement des messages, ou alors, le protocole de transport est capable de le détecter et de le corriger.

Un processeur effectue une invalidation lorsqu'il devient le propriétaire d'une page suite à un défaut en écriture. Il effectue alors une demande d'invalidation. L'invalidation consiste à envoyer à l'ensemble des sites (copyset) qui ont une copie de la page une requête pour qu'ils changent le mode d'accès de celle-ci à "nil". Cette demande s'effectue à l'aide de la primitive :

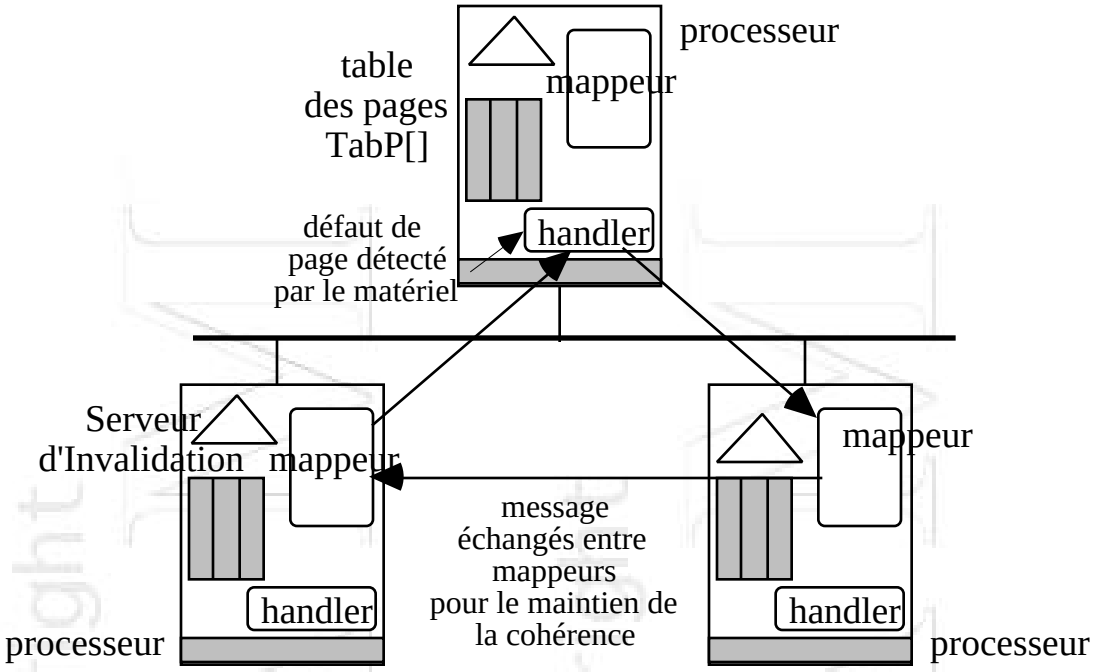
Invalider(page, ensemble de processeurs destinataires).

La réception et l'exécution de la requête d'invalidation sur un processeur est effectué par le **serveur d'invalidation** propre à celui-ci.

Les informations que maintiennent les mappeurs sur les pages sont contenues dans une table **TabP[]** , table des pages qui contient les champs suivants :

- **access** représente le mode d'accès à la page : **read**, **write**, **nil**,
- **copyset** contient l'ensemble des processeurs qui ont une copie (en lecture) de la page,
- **lock** joue le rôle de variable sémaphore pour l'accès en exclusion mutuelle à la page pendant un défaut, deux primitives sont utilisées :
  LOCK() pour tester le sémaphore et éventuellement bloquer le demandeur,
  UNLOCK() pour déverrouiller l'accès,
- **probOwner** indique un propriétaire estimé (probable) de la page.

Le schéma page suivante illustre ces éléments.



L'algorithme de gestion distribuée de la mémoire virtuelle répartie est le suivant :

Handler de défaut de page en lecture :

LOCK (TabP[p].lock);
Demander à TabP[p].probOwner l'accès en lecture pour la page p;
TabP[p].probOwner := processeur qui a répondu;
TabP[p].access := read;
UNLOCK (TabP[p].lock);

Mappeur en lecture:

LOCK (TabP[p].lock);
IF je suis le propriétaire THEN BEGIN
TabP[p].copyset := TabP[p].copyset U {processeur demandeur};
TabP[p].access := read;
Envoyer (p au processeur demandeur);
END
ELSE BEGIN
Rediriger la requête vers le processeur (TabP[p].probOwner);
TabP[p].probOwner := processeur demandeur;
END;
UNLOCK (TabP[p].lock);

Handler de défaut de page en écriture :

```
LOCK ( TabP[p].lock);
Demander à TabP[p].probOwner l'accès en écriture pour la page p;
Invalider (p, TabP[p].copyset);
TabP[p].probOwner := moi-même;
TabP[p].access := write;
TabP[p].copyset := {Ø};
UNLOCK ( TabP[p].lock);
```

Mappeur en écriture:

```
LOCK ( TabP[p].lock);
IF je suis le propriétaire          THEN BEGIN
        TabP[p].access := nil;
        Envoyer (p et TabP[p].copyset au processeur demandeur);
        TabP[p].probOwner := processeur demandeur;
        END
                                    ELSE BEGIN
        Rediriger la requête vers le processeur (TabP[p].probOwner);
        TabP[p].probOwner := processeur demandeur;
        END;
UNLOCK ( TabP[p].lock);
```

Serveur d'Invalidation :

```
TabP[p].access := nil;
TabP[p].probOwner := processeur demandeur;
```

On forme le graphe des propriétaires probables pour une page p de la façon suivante:

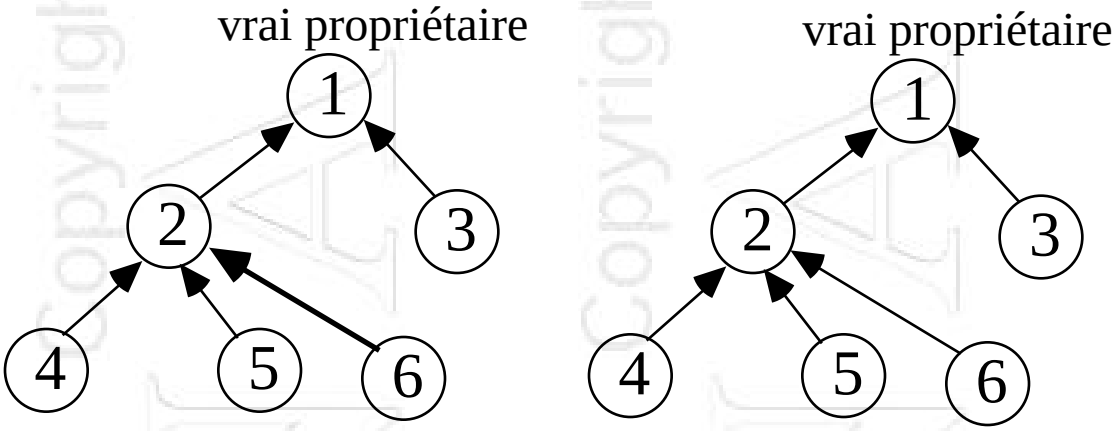
Pi->Pj si TabP[p].probOwner de Pi := Pj.

Gestion Répartie Dynamique avec copyset réparti (1)

Le copyset ne sert qu'au moment de l'invalidation, on peut imaginer une autre façon de le gérer pourvu que l'invalidation s'effectue toujours correctement.

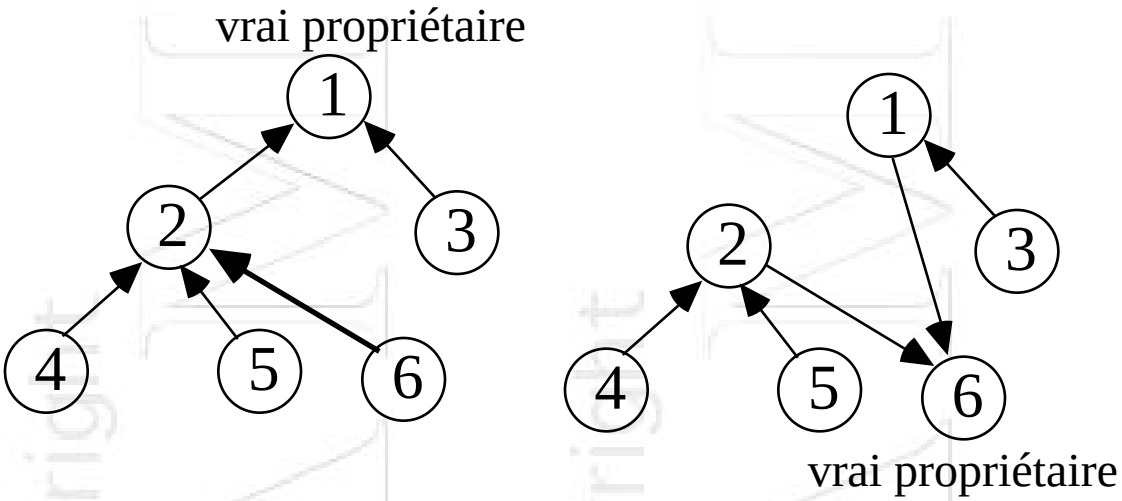
Maintenant un site qui détient un exemplaire de la page peut fournir la page au demandeur en lecture. Il ajoute alors le demandeur au copyset.

graphe des propriétaires probables :



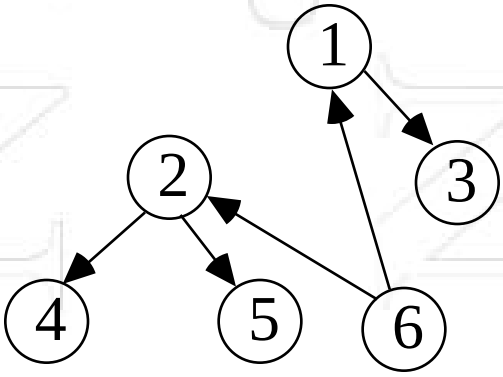
6 effectue un défaut en lecture et est servi par 2

Gestion Répartie Dynamique avec copyset réparti (2)



6 effectue un défaut en écriture et est servi par 1

Propagation des demandes d'invalidation



La distribution du copyset organise l'ensemble des noeuds en arbre.

Le graphe des propriétaires probables va des feuilles vers la racine.

Le graphe des invalidations va de la racine vers les feuilles sens oppé au précédent.

Algorithme (suite) [6]:

L'invalidation est plus efficace si le réseau qui relie les processeurs supporte la diffusion. En effet, le serveur qui déclenche une invalidation n'a qu'à diffuser en un seul message sa requête vers tous les processeurs. Les auteurs proposent une nouvelle version de leur algorithme pour tenir compte des réseaux qui ne supportent pas la diffusion.

Handler de défaut de page en lecture : (inchangé p/r à la version optimisée précédente)

LOCK (TabP[p].lock);
Demander à TabP[p].probOwner l'accès en lecture pour page p;
TabP[p].probOwner := processeur qui a répondu;
TabP[p].access := read;
UNLOCK (TabP[p].lock);

Mappeur en lecture:

LOCK (TabP[p].lock);
IF TabP[p].access ≠ nil THEN BEGIN
    TabP[p].copyset := TabP[p].copyset U {processeur demandeur};
    TabP[p].access := read;
    Envoyer (p au processeur demandeur);
    END
        ELSE BEGIN
Rediriger la requête vers le processeur (TabP[p].probOwner);
    TabP[p].probOwner := processeur demandeur;
    END;
UNLOCK (TabP[p].lock);

Handler de défaut de page en écriture : (inchangé)

LOCK (TabP[p].lock);
Demander à TabP[p].probOwner l'accès en écriture pour pagep;
Invalider (p, TabP[p].copyset);
TabP[p].probOwner := moi-même;
TabP[p].access := write;
TabP[p].copyset := {Ø};
UNLOCK (TabP[p].lock);

Mappeur en écriture : (inchangé)

```
LOCK ( TabP[p].lock);
IF je suis le propriétaire          THEN BEGIN
        TabP[p].access := nil;
        Envoyer (p et TabP[p].copyset au processeur demandeur);
        TabP[p].probOwner := processeur demandeur;
        END
                                    ELSE BEGIN
        Rediriger la requête vers le processeur (TabP[p].probOwner);
        TabP[p].probOwner := processeur demandeur;
        END;
UNLOCK ( TabP[p].lock);
```

Serveur d'Invalidation :

```
IF TabP[p].access ≠ nil   THEN BEGIN
        Invalider (p, TabP[p].copyset);
        TabP[p].copyset := {};
        TabP[p].access := nil;
        TabP[p].probOwner := processeur demandeur;
        END
```

On définit la notion de graphe des invalidations de la façon suivante :

Pk -> Pm si Pk envoie une requête d'invalidation vers Pm.

Problèmes des protocoles de gestion de la cohérence

. TRASHING : on invalide une page de la mémoire locale alors qu'on va en avoir besoin ...

pb qui se pose en gestion de mémoire centralisée, pb du Working Set, l'adaptation de ce problème aux caches est connu sous le nom de **Working Set Restauration**

. PING-PONG : voyage d'une page d'une mémoire à l'autre sous l'effet de la demande

-> par exemple plusieurs processeurs font une exclusion mutuelle :

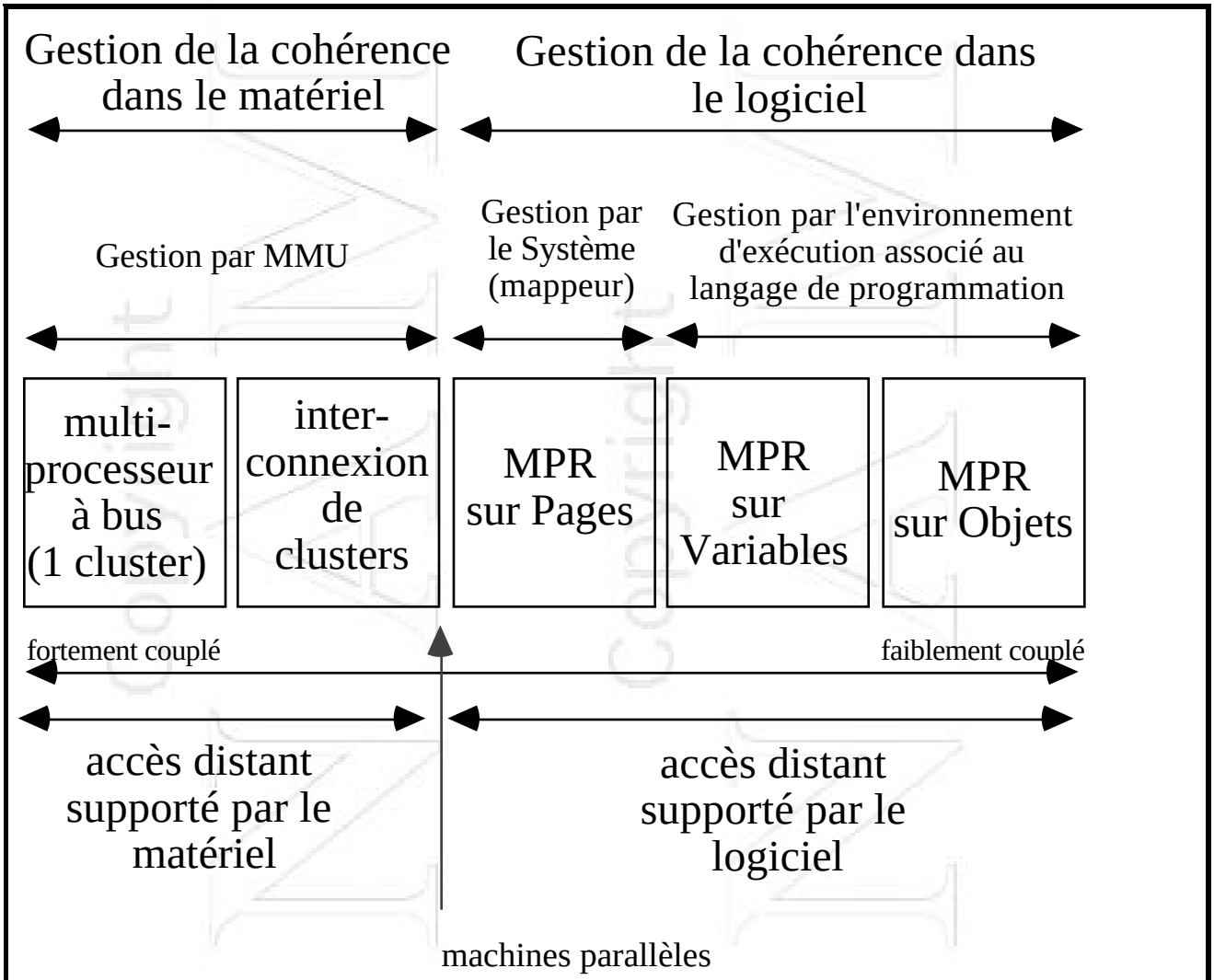
while (TEST&SET (lock) = 1) **do** nothing ;
< section critique>
RESET (lock) ;

La page qui contiendrait la variable lock n'arrête pas de se promener entre chaque processeur!

. FAUX PARTAGE : Une partie des données contenues dans une page n'est pas partagé alors que le reste l'est.

Remarque : Ces problèmes existent aussi en multiprocesseur.

Typologie des architectures gérant les données réparties



liste indicative des projets de recherche (nécessairement incomplète) :

FIREFLY
DASH
KSR
IVY
MIRAGE
CLOUDS
LEASES
KOAN(IRISA)
MYOAN(IRISA)
GOTHIC(IRISA)
CRL
MUNGI
SAFFRES(CNAM)
MUNIN
TREADMARK
MIDWAY
PHOSPHORUS(ENST)
ARIAS(IMAG)
ORCA
LARCHANT(INRIA)

Conclusion

- Convergence des Architectures Multiprocesseur et Réseau d'ordinateurs

=> Similitude des problèmes à Résoudre
(KSR, Dash)

- Cohérence des données

=> Affaiblir la cohérence pour augmenter les performances

-Modèle d'exécution

=> Meilleur support du Parallèlisme et des Objets

- Prise en compte de l'extensibilité

Références Bibliographiques

Aspects Généraux :

[1] Advanced Computer Architecture, Parallelism, Scalability, Programmability. Kai Hwang. Mac Graw Hill.1993.

[2] Distributed Systems : concepts and Design. George Coulouris, Jean Dollimore, Tim Kindberg. Addison Wesley 1994.

Multiprocesseurs

[3] Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model. James Archibald, Jean-Loup Baer. ACM TOCS. V4. N4. November 1986.

[4] Implementing A Cache Consistency Protocol. Katz, S. J. Eggers, D.A. Wood, C.L. Perkins, R.G. Sheldon. Proceedings of the 12th ISOCA. IEEE. New York 1985.

[5] Firefly : A Multiprocessor Workstation. Charles P. Thacker, Lawrence C, Stewart, Edwin Satterthwaite. IEEE TOC. V37.N8. August 1988.

Systèmes faiblement couplés

[6] Memory Coherence in Shared Virtual Memory Systems. Kai Li, Paul Hudak. ACM TOCS, V7, N4, November 1989.

Annexe

ASR-Données réparties

Références Bibliographiques pour l'annexe :

[1] Advanced Computer Architecture, Parallelism, Scalability, Programmability. Kai Hwang. Mac Graw Hill.1993.

[2] Unix Systems for Modern Architectures. C. Schimmel. Addison Wesley. 1994.

[3] Organisation et Conception des Ordinateurs. D. Patterson, J. Hennessy. Dunod. 1994.

Pour plus d'informations consulter ces ouvrages. La référence [2] est celle qui a été la plus utilisée.

Principes fondamentaux des caches

1. Un cache contient un sous-ensemble de la mémoire centrale. Il faut des informations supplémentaires pour retrouver dans la mémoire ce qui est stocké dans le cache. Pour cela les données sont accompagnées d'une **marque/étiquette** "*tag*". Un tag contient l'adresse de l'information archivée dans le cache.

Lorsque le processeur effectue un accès à une variable en mémoire, l'adresse de cette information est envoyée au cache, une recherche s'effectue alors, l'information est retrouvée grâce à sa marque, si elle est présente dans le cache.

Si l'adresse ne correspond à aucun tag, l'adresse est passée à l'extérieur pour être servie soit par la mémoire principale, soit par un autre cache dans le cas d'un multiprocesseur. Au retour, l'information est stockée dans le cache.

Lors d'un chargement du cache, un peu plus d'information que ce qui est demandé est ramené dans le cache (souvent un mot mémoire est accèdé même si on a besoin que d'un demi-mot). Pour charger le cache deux stratégies : **à la demande** ou **préchargement.**

L'unité mémoire de gestion d'un cache est une **ligne** ou un **bloc** (16 à 32 octets voire 128 et même 256 octets pour mieux bénéficier des effets de la localité), il y a une marque associée à chaque ligne.

Une ligne de cache c'est donc : tag + information

<bit de validité, bit de modification, [clé], adresse de l'information><information>

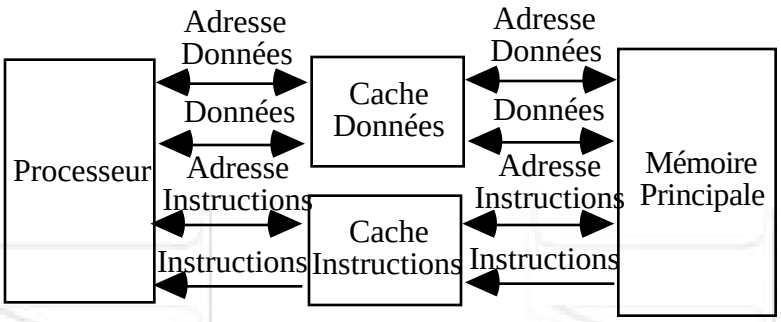
bit de validité : indique si la ligne contient des informations en service, valides

bit de modification : indique si la ligne a été modifiée depuis son chargement dans le cache, ce bit sert pour la mise à jour d'autres caches ou de la mémoire principale suivant la stratégie de cohérence choisie.

On parle de **grain** d'une ligne quand on parle de sa taille, il faut trouver un bon rapport effet de localité/temps de transfert entre mémoire et cache. Quand le grain est gros, la ligne peut être découpée en **sous-lignes**. il reste toujours qu'une marque par ligne.

2. Le cache peut être accédé soit par l'adresse virtuelle d'une information, soit par son adresse physique.

3. Cache Intructions et Données : ensembles ou séparés ?



Ce shéma est fréquent

4. Stratégie de Remplacement des blocs dans un cache quand il est plein :
FIFO : simple à implanter
LRU ou pseudo LRU : plus efficace, **rapide si bien implanté**, requiert beaucoup d'information supplémentaire donc **coûteux**
Le cache est trop petit pour pouvoir tenir compte de la notion de Working Set.
La stratégie de remplacement dépend de l'organisation du cache.

Il faut vider le cache : lors du remplacement d'une ligne, ou lors d'un changement de contexte du au changement de processus élu.

5. Politiques de mise à jour de la mémoire suite à modification(s) du cache. On examine ici uniquement le cas monoprocesseur. Le cas multiprocesseur est vu dans le cours.
Cas d'une donnée dans le cache :
- mise à jour immédiate **"write through"** ou "write-update": à chaque écriture dans le cache, la mémoire est systématiquement mise à jour. Mémoire et cache contiennent des données identiques, mais le bus est soumis au trafic des mises à jour.
-mise à jour retardée **"write-back"** ou "copy-back": la mémoire n'est mise à jour que quand une ligne modifiée (bit ligne modifiée positionné) quitte le cache. On évite des transferts sur le bus, mais cache et mémoire peuvent contenir des données différentes, et la mémoire principale qui sert généralement de référence ne plus être à jour. Lors du remplacement d'une ligne suite à un défaut cache, le défaut ne peut être terminé tant que la mémoire n'a pas été mise à jour.

Cas d'une donnée absente du cache :
Tout dépend si le matériel supporte l'écriture avec allocation **"write-allocate"**. Les processeurs qui ne le supportent pas écrivent directement dans la mémoire principale.
Pour ceux qui possèdent la stratégie write-allocate :

. soit l'information à écrire correspond à une ligne entière du cache, dans ce cas on fait de la place dans le cache avec recopie de la ligne vidée vers la mémoire principale si elle a été modifiée et que la politique write-back est de rigueur
. soit la l'information à écrire est plus petite qu'une ligne de cache, dans ce cas on fait de la place dans le cache comme au cas précédent, et on lit la ligne de la mémoire principale vers le cache, enfin on écrit l'information dans la ligne qui vient d'être chargée.
En général, write-allocate va avec write-back ... mais toute autre combinaison est possible.

6. Les E/S malmènent la cohérence mémoire/cache. Deux possibilités : le contrôleur d'E/S échange directement avec le cache (plutot write-back ?), ou le contrôleur d'E/S échange uniquement avec la Mémoire Principale (plutot write-through ?)

7. Gestion de la Multiprogrammation
. orienté temps de réponse -> 1 programme à la fois utilise tous les processeurs
. orienté débit -> plusieurs programmes utilisent en même temps tous les processeurs

Multi-programmation => il faut un certain temps pour recharger l'environnement d'un processus dès qu'il est élu, taux de multi-programmation augmente =>**miss ratio** (taux de défauts) augmente, d'où la nécéssité d'avoir une stratégie de démarrage :

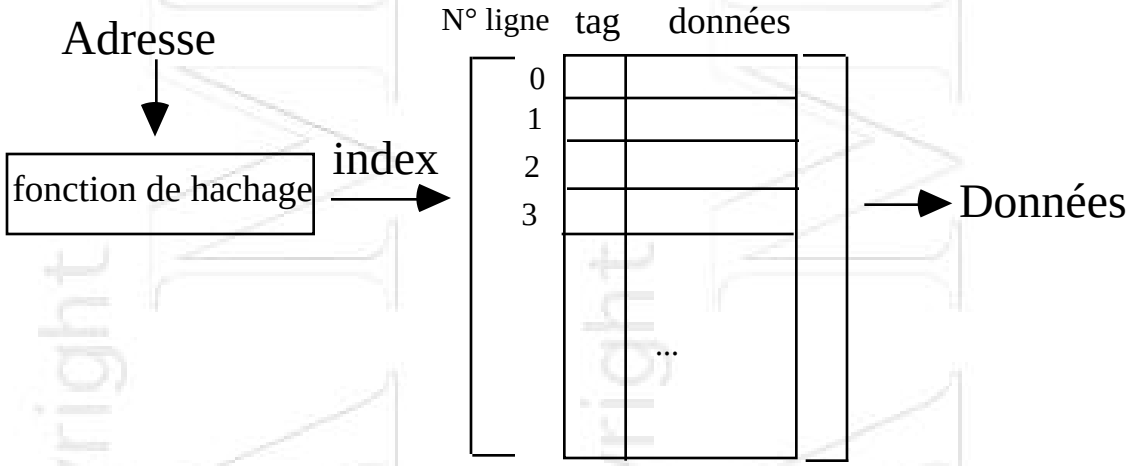
- démarrage à froid par vidage de tous les caches
- démarrage à chaud : on laisse agir l'algorithme de remplacement

Mode système/mode utilisateur => un cache réservé à l'exécution du code système ? dépend du choix du type de multi-processeur symétrique/asymétrique

8. Translation des adresses virtuelles en adresses physiques : le cache peut contenir une table de translation d'adresses (TLB - Translation Lookaside Buffer) qui contient les correspondances <ad virtuelles-ad réelles> les plus fréquentes qu'utilise le processeur.

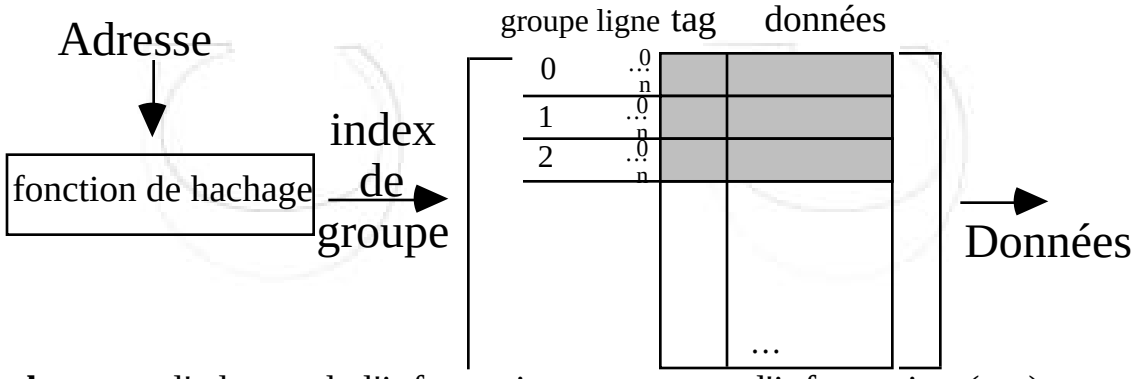
Correspondance Cache / Mémoire Centrale

1. Caches à correspondance directe :



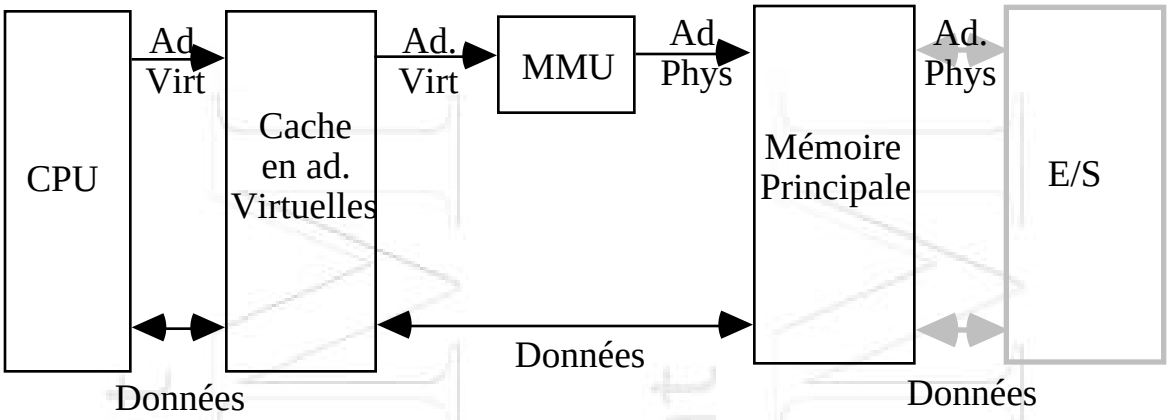
L'adresse de la donnée est prise puis soumise à une fonction de hachage qui retourne un index. Cet index indique la position de la donnée dans le cache si elle s'y trouve. Plusieurs adresses peuvent produire le même index, l'adresse demandée est donc comparée avec l'étiquette contenue dans la ligne pointée par l'index. Deux adresses qui donnent le même numéro de ligne sont dites de la même **couleur**. Quand la donnée recherchée est trouvée dans le cache, elle est envoyée au processeur. Sinon, la mémoire centrale est sollicitée pour fournir la donnée.
Les algorithmes de hachage peuvent prendre une partie de l'adresse de la donnée pour retrouver la ligne dans le cache (TI MicroSPARC-> 2Ko de données, 128 lignes de 16 octets, mot adressable de 2 octets: les bits 10 à 4 pour choisir la ligne, et les bits 3 à 0 pour le mot dans la ligne).

2. Caches associatifs



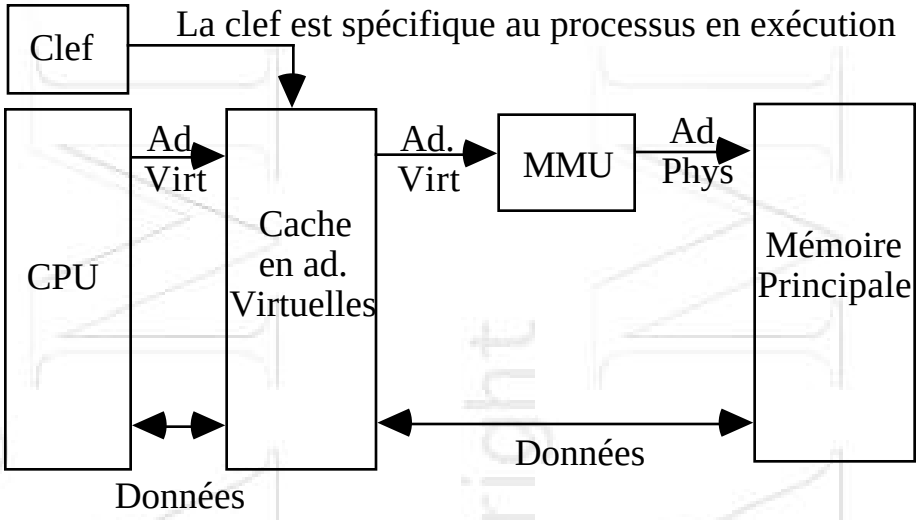
1. **hachage** sur l'adresse de l'information => groupe d'information (**set**) contenu dans une **mémoire associative** (**set-associative memory**)
2. accès parallèles au groupe d'information sélectionné
Toute la mémoire cache peut être une seule mémoire associative :solution lente qui coûte cher (solution admissible pour la TLB 64 entrées dans celle du SuperSPARC)

Caches virtuels



Pour adresser le cache on se sert des adresses virtuelles utilisées par le programme... pas de translation d'adresse avant l'accès au cache, c'est donc plus rapide. Ce type de cache pose des problèmes, car c'est la MMU qui vérifie les droits d'accès d'un programme à une donnée, hors celle-ci n'est sollicitée que si il y a un défaut dans le cache.
Autres problèmes :
ambiguité - le cache contient une donnée référencée par une adresse virtuelle correcte, mais la correspondance adresse virtuelle/adresse physique a changé depuis le moment où la donnée a été chargée dans le cache, il est alors difficile de conserver la cohérence mémoire centrale/cache (cette situation peut se produire avec une même adresse virtuelle de deux processus différents).
alias - plusieurs adresses virtuelles réference la même adresse physique, pour chaque adresse virtuelle une ligne du cache peut contenir une version différente de la donnée référencée (segments de mémoire partagée d'Unix).
Le système doit alors vider le cache pour toute opération qui risque de modifier la cohérence cache/mémoire. Ceci peut être fréquent, et on peut perdre l'avantage d'une gestion de cache à base d'adresses virtuelles.
Le cache est vidé dès qu'on change de processus élu puisque les adresses virtuelles n'ont plus de sens. On perd ainsi le bénéfice du principe de localité.

Caches virtuels avec identificateurs

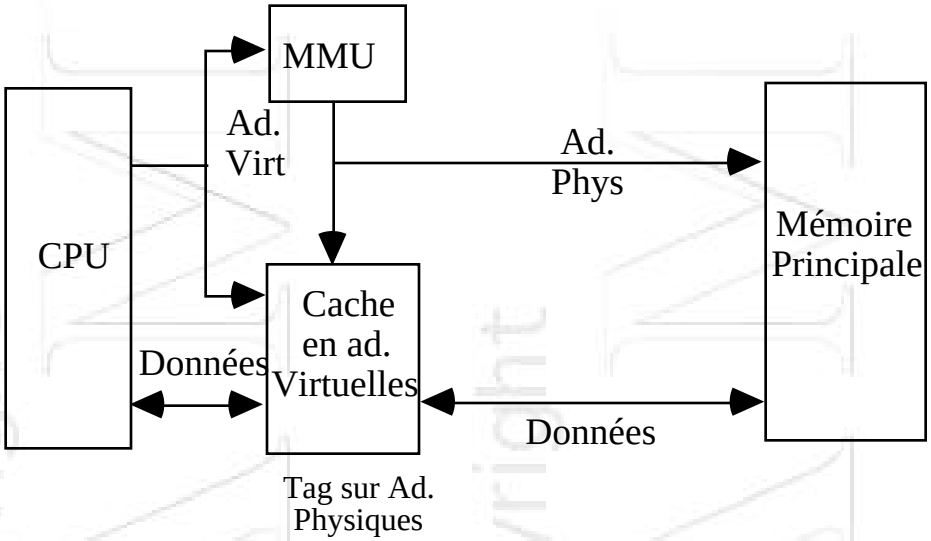


Une clef qui identifie le processus de façon unique est ajoutée à chaque ligne du cache dans l'étiquette.

Même principe que le mécanisme de cache précédent. Mais pour qu'un accès au cache réussisse, il faut que l'adresse virtuelle corresponde et que la clef du processus concorde avec celle de l'étiquette. Le cache n'a plus besoin d'être vidé dès qu'on change de processus.

Les deux techniques suivantes sont préférées pour les architectures multiprocesseur.

Caches virtuels avec marquage par adresses physiques



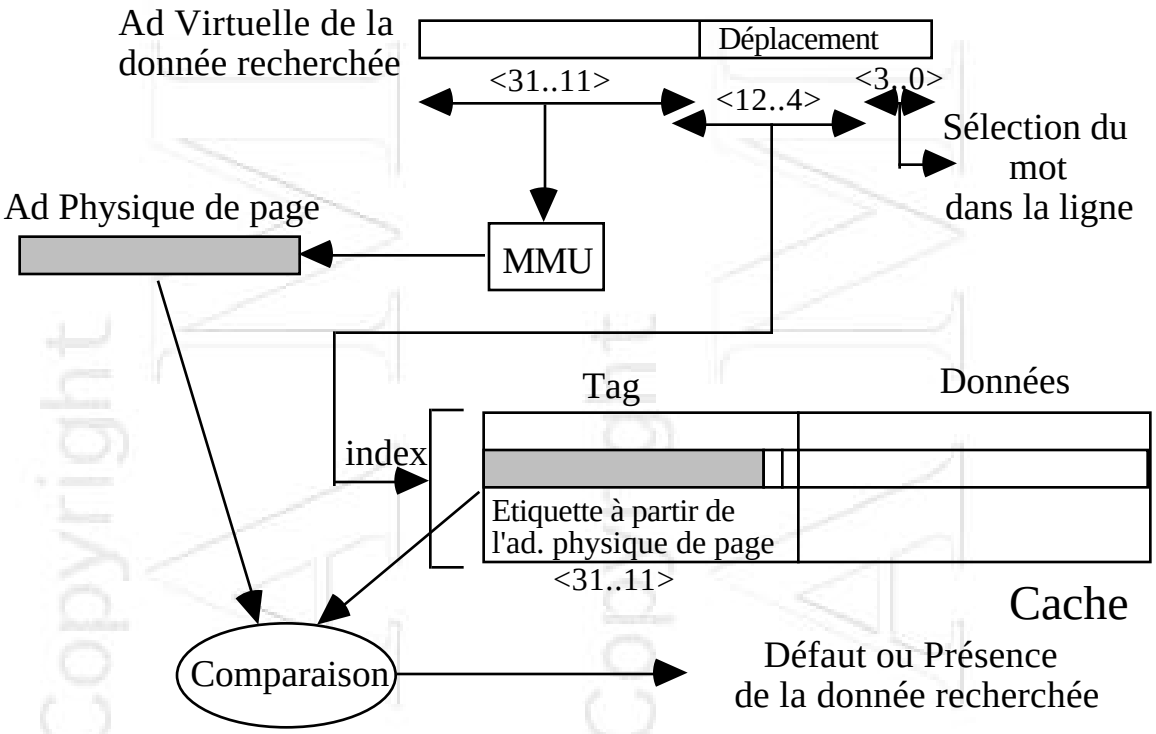
Même principe de fonctionnement qu'avec un cache virtuel. L'étiquette par contre est fabriquée à partir de l'adresse physique et non de l'adresse virtuelle.
Recherche d'une ligne dans le cache, à chaque accès :
- l'adresse virtuelle est transmise au cache et à la MMU
- parallèlement :
    . la MMU calcule l'adresse physique à partir de l'adresse virtuelle
    . le cache calcule l'indexe pour rechercher la ligne accédée
- la MMU envoie l'adresse physique obtebue au cache qui la compare à celle qu'il trouve dans l'étiquette
- si c'est bon, il fournit la donnée sinon, elle est ramenée depuis la mémoire

Avantages :

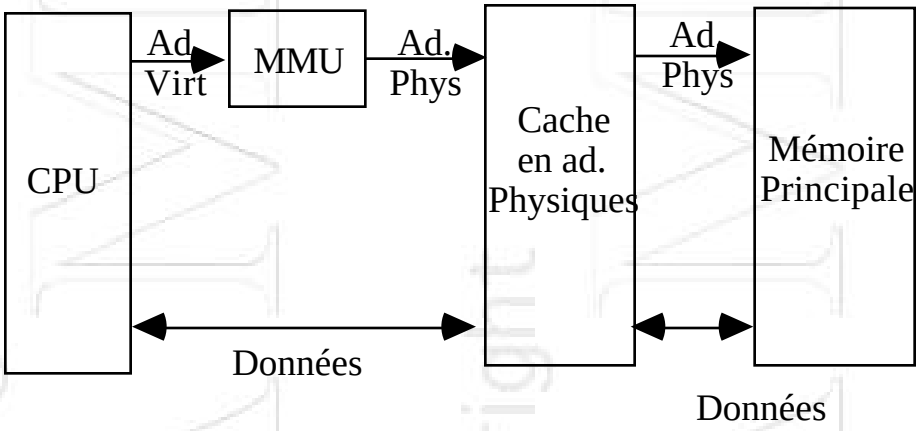
- Il n'y a plus d'ambiguite sur les données accédées dans le cache, l'adresse physique est là pour les différencier.
- La MMU peut aussi vérifier les droit d'accès à la page pendant le calcul d'adresse.
- Plus besoin de calculer l'adresse physique lors de l'éjection d'une ligne du cache.
- Le changement de processus ne nécessite plus l'invalidation du cache.

Par contre, c'est plus lent à cause du calcul par la MMU.

Translation d'adresse pendant la recherche dans le cache :



Caches physiques

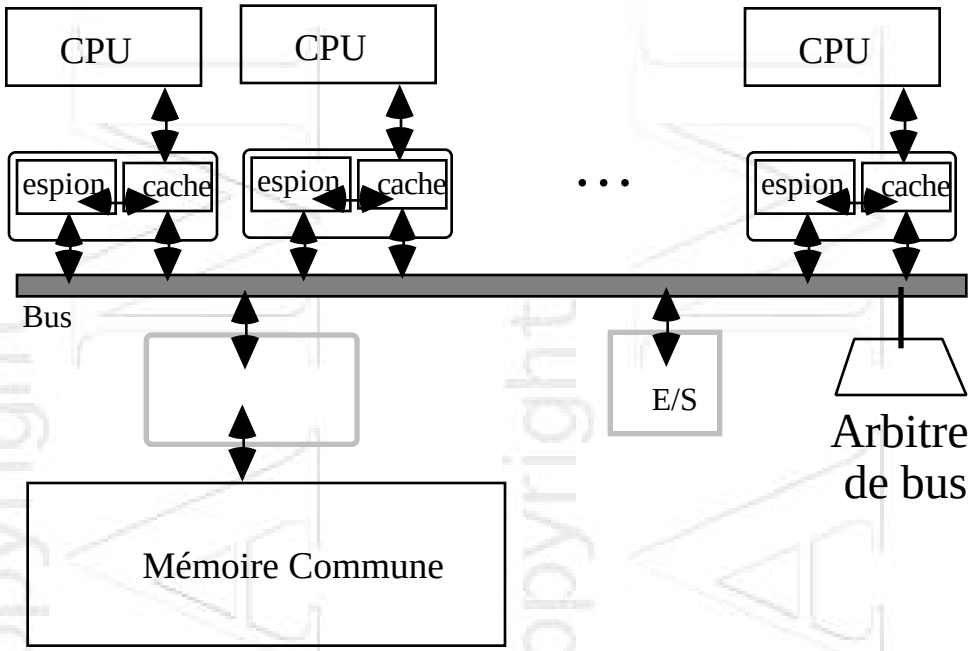


L'adresse physique est calculée à chaque accès.

Type de cache réservé au caches hors puce processeur.

Le contrôleur de cache-Snoopy Cache Controler

Snoopy ou encore Espion => hypothèse de Machines MIMD à bus :



Que contient-il ? :

- un répertoire des blocs contenus dans le cache (mémoire privée), avec des informations supplémentaires sur l'état des blocs, l'espion accède au contenu du cache comme le fait le processeur.
- une TLB, cela dépend si le cache fonctionne sur la base d'adresses virtuelles ou d'adresses physiques
- un traducteur qui calcule la correspondance <ad virtuelle- ad réelle> qd celle-ci n'est pas dans la TLB

[- Buffers liés aux écritures : contient des invalidations ou des mises à jour du cache, mécanisme prioritaire par rapport aux accès du processeur]

[- Filtre des Invalidations -> permet de regrouper les invalidations identiques]

Rôle :
Scruter le bus pour connaître les transactions effectuées par les autres caches et ainsi participer au maintien de la cohérence des copies des données.

Certaines informations sont partagées entre le Processeur et son Espion. Si l'espion travaille sur la mémoire privée, le processeur ne peut pas le faire :
- par exemple quand l'espion n'a pas terminé d'acquérir une donnée liée à un accès du processeur.
- autre exemple : conflit d'accès au répertoire des blocs contenus dans la mémoire privée. Une solution consisterait à dupliquer le répertoire => gain vitesse, mais pb de cohérence du répertoire.