Cnam-Laboratoire Cedric

Cohérence des données Réparties Partagées

Construction de Systèmes Répartis

Eric Gressier-Soudan Cnam

Plan

Introduction

Cohérences Uniformes

Cohérences Hybrides

Bases de Mise en oeuvre Logicielle des DSM

Introduction

Pourquoi répartir les données

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

- performance / efficacité
- disponibilité / réplication
- tolérance aux pannes

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

- le parallèlisme des processus[1]
- communication par variables partagées au lieu de la communication par messages
- accès aux données: producteur-consommateur

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

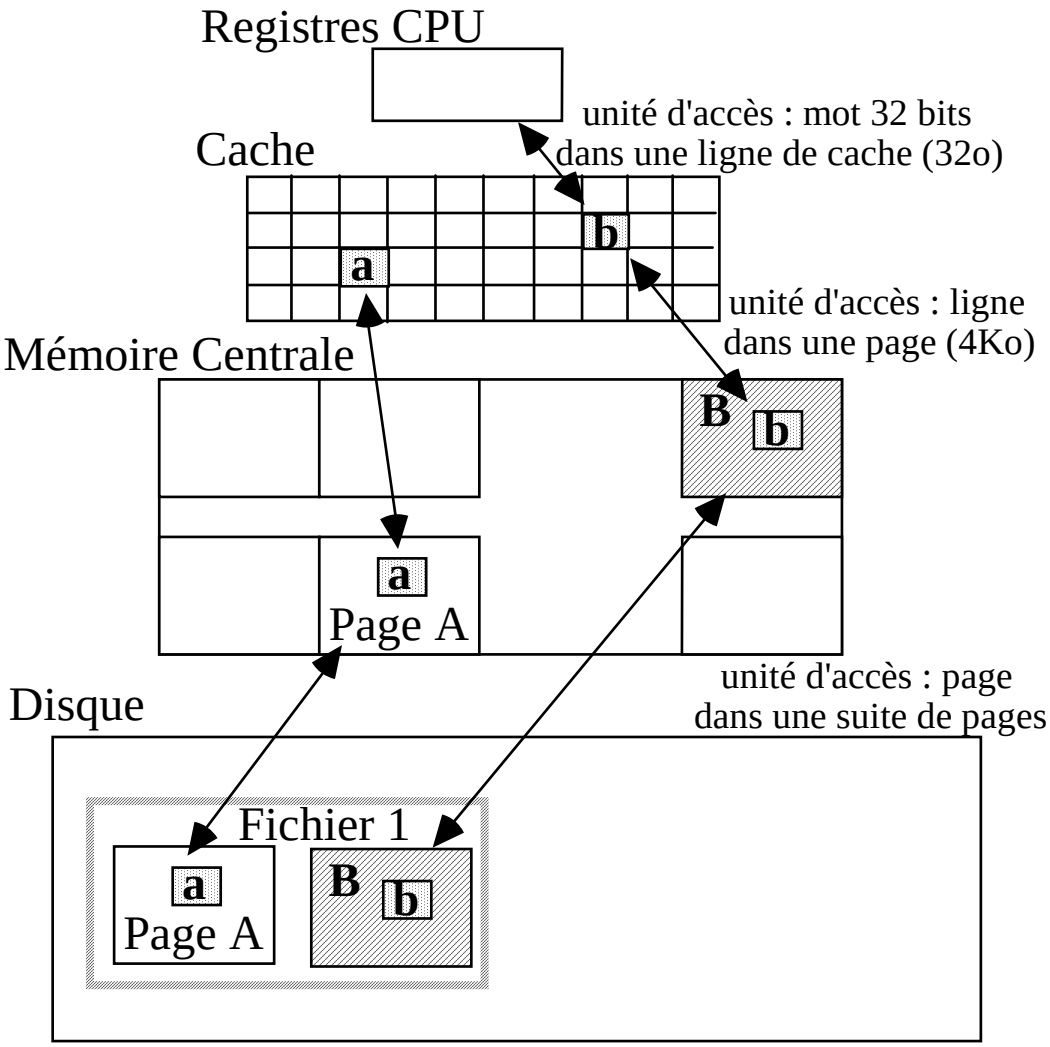
(déjà connu en univers centralisé)

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 offertes par un système

Hiérarchie de mémoires (cohérence verticale)

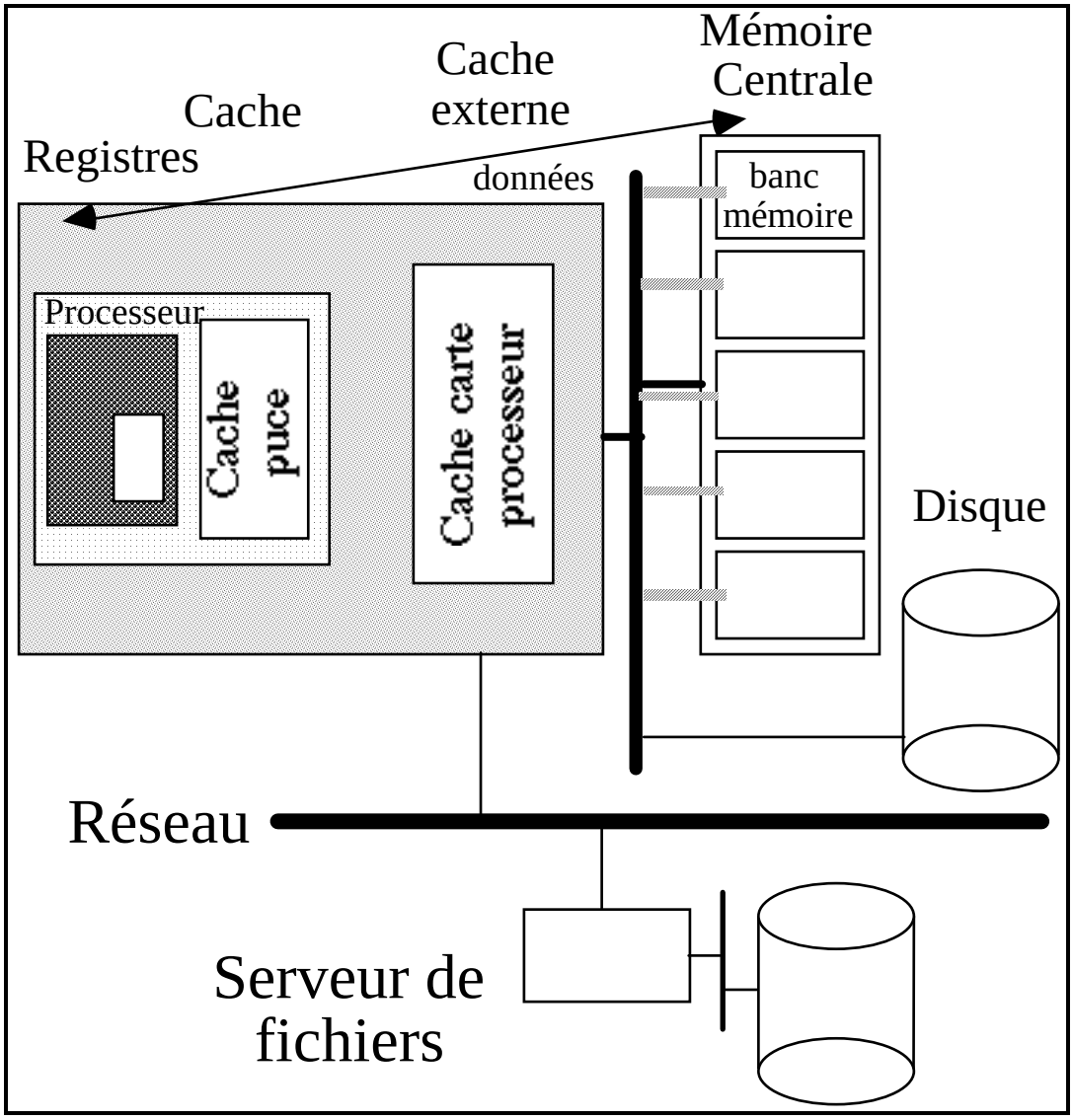


vitesse d'accès :

Processeur< Cache < Mémoire Centrale < Disque

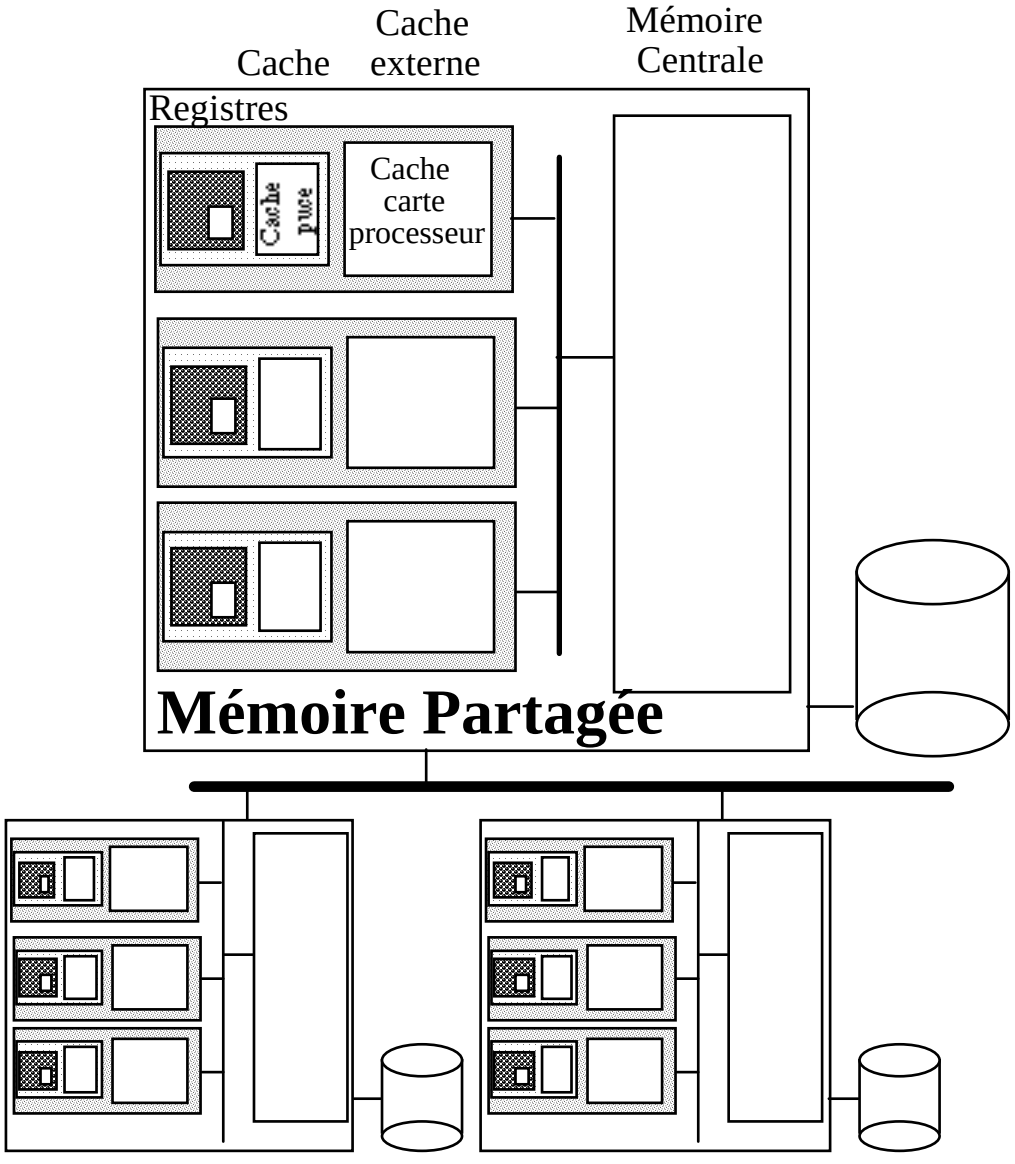
Problème de la répartition des données (1)

Problème en centralisé monoprocesseur ?



Problème de la répartition des données (2)

Cohérence verticale + horizontale



. 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 d'exécution**.

. La **Gestion de Fichiers Répartis** relève de 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 (abordé en groupe de recherche).

. Les **Bases de Données Réparties** sont intéressées par les problèmes de cohérences, mais c'est un autre cours

Architectures Considérées :

les architectures parallèles de type MIMD[3]

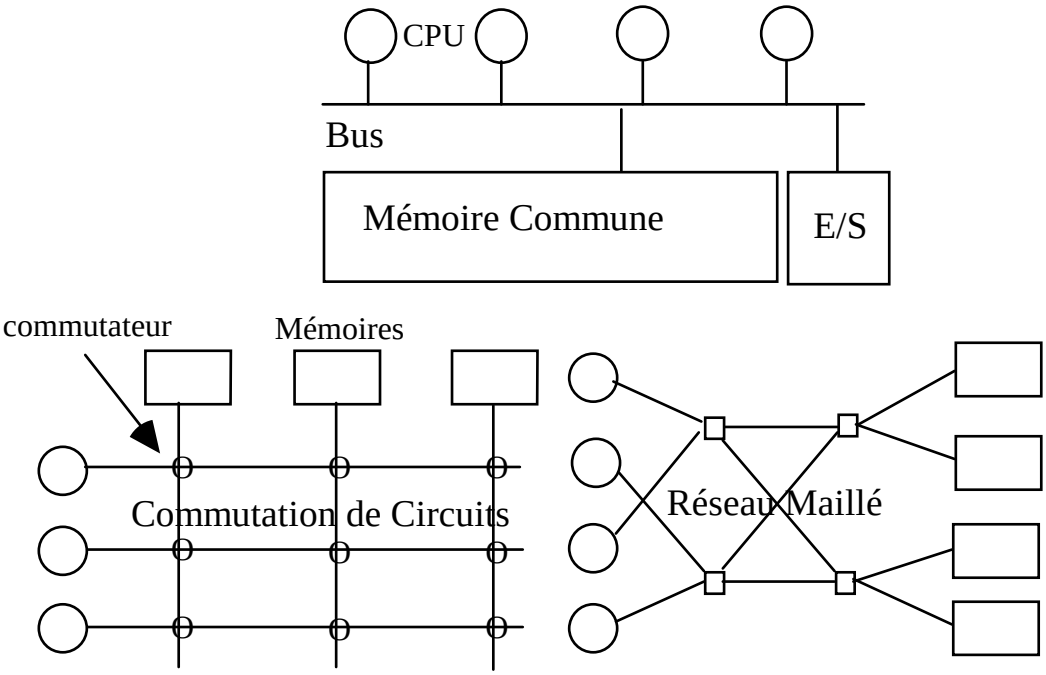
Nous ne nous intéressons qu'aux **mises en oeuvre logicielles** même si les multiprocesseurs ont inspiré le domaine et contribuent à l'alimenter.

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

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.

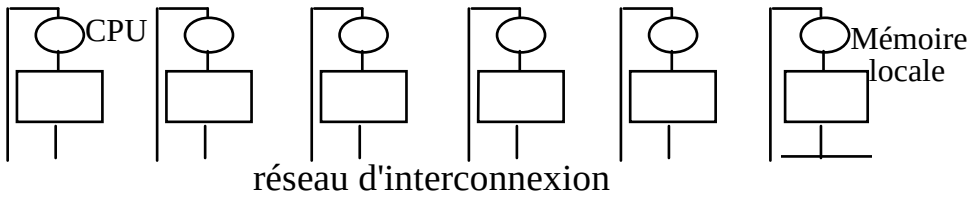
multiprocesseur asymétrique : certains processeurs sont spécialisés, un processeur exécute le système et fait des E/S, les autres sont dédiés à l'exécution de programmes utilisateur.

[4] Uniform Memory Access

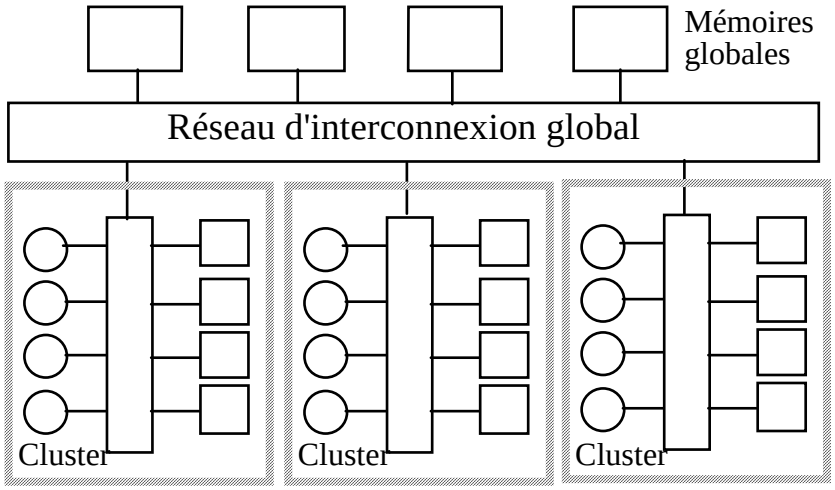
Machines Fortement Couplées (2)

. **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



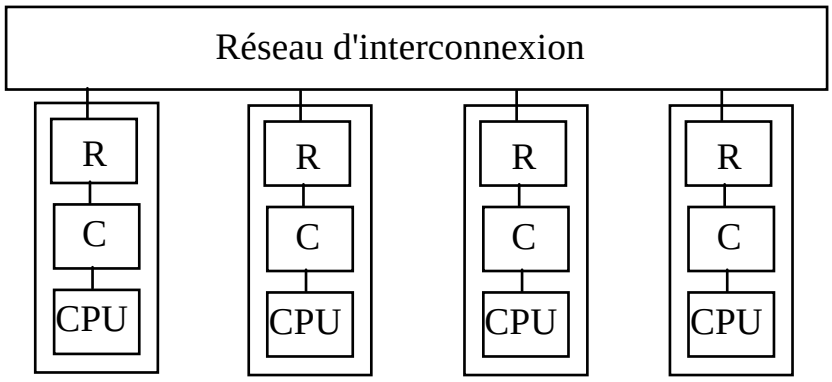
multiprocesseur Cedar (Université Illinois)

[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

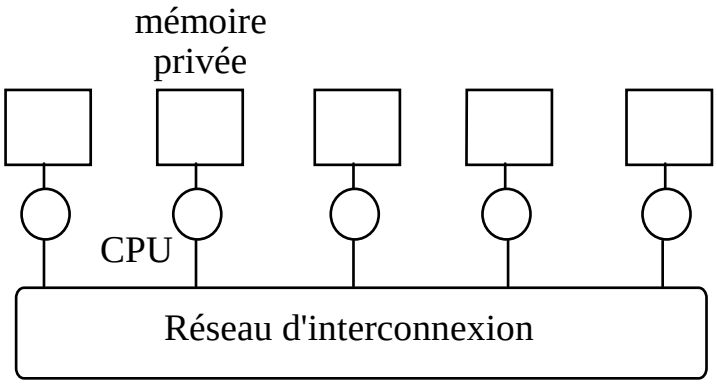
[6] Cache-Only Memory Architecture

Machines Faiblement Couplées

multiordinateur -> processeurs en réseau

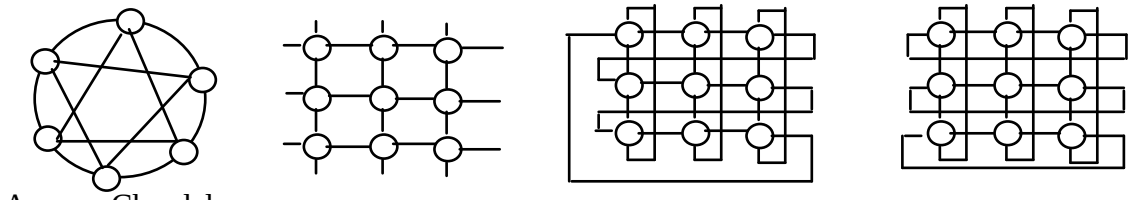
la communication entre processeurs s'effectue uniquement par messages, les mémoires sont privées et ne sont accessibles que par le processeur local

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



Réseau :

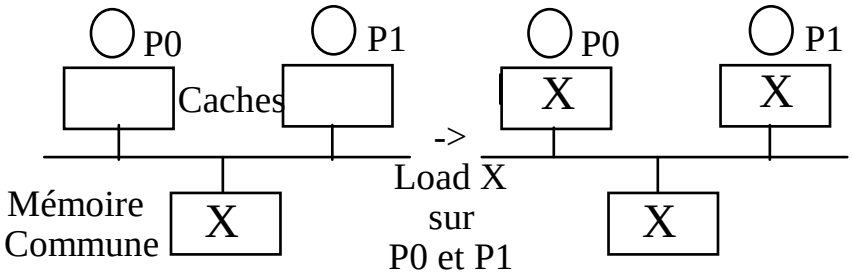
Réseau Local, Anneau, Anneau "Chordal", Arbre binaire, Etoile, Hypercube, Réseau maillé, Tore, Mesh...



[7] No Remote Memory Access

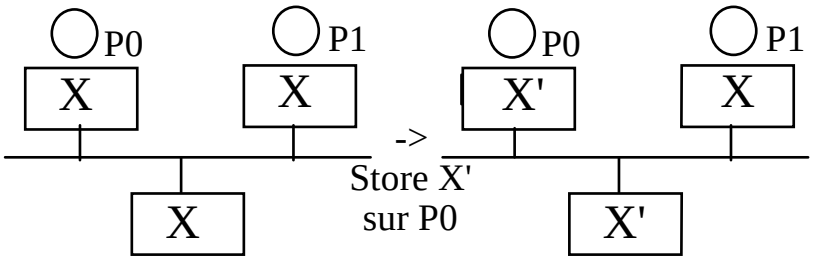
Rappel multiprocesseur (1)

Situation initiale :



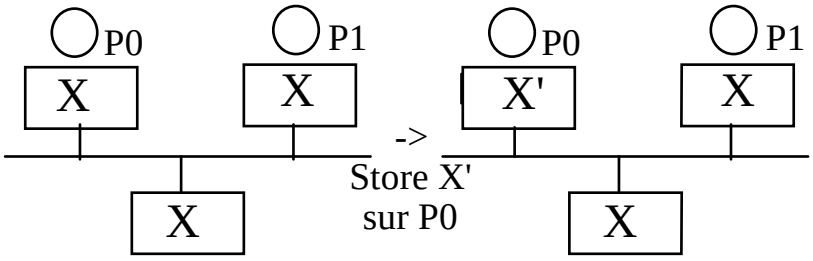
Les copies sont cohérentes

Mise à jour immédiate (Write-through):



Les copies sont incohérentes

Mise à jour retardée (Write-back):

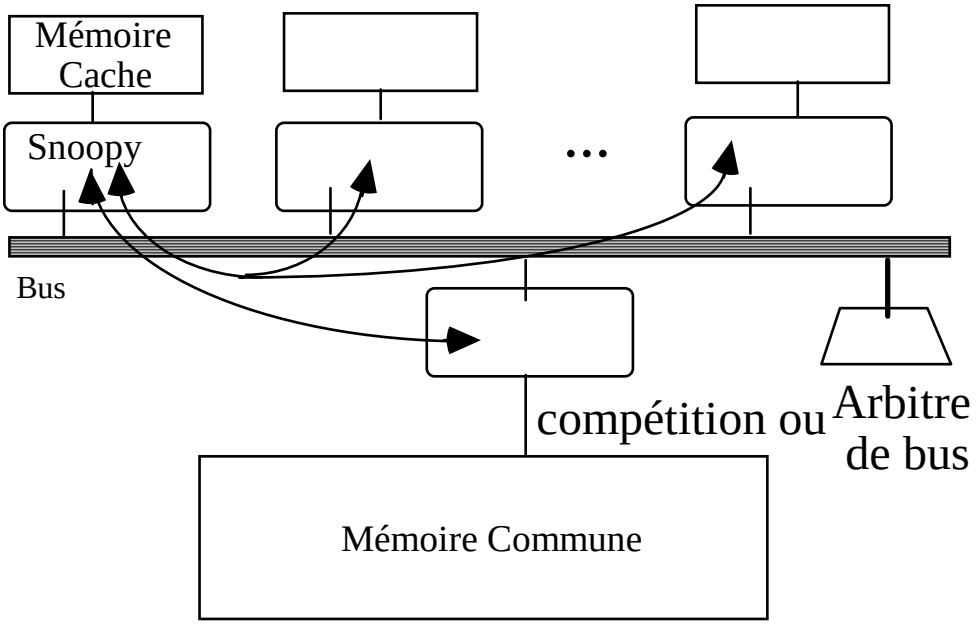


Les copies sont incohérentes

Mécanismes Insuffisants pour le maintien de la cohérence

Rappel Multiprocesseur (2)

Le maintien de la cohérence entre caches se fait par coopération des différents contrôleurs (snoopy):



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.

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

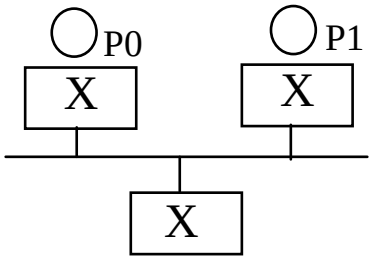
A chaque écriture sur une copie, toutes les autres copies présentes dans les autres caches sont mises à jour.

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

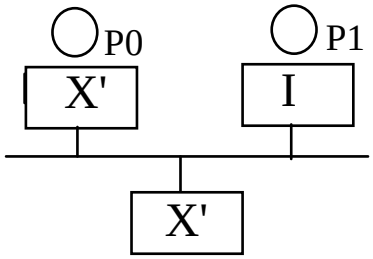
Rappel Multiprocesseur (3)

Multiprocesseur avec mémoire à mise à jour immédiate

Situation initiale :

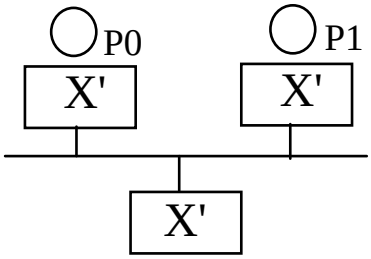


Effet d'une invalidation sur écriture lors de la modification de X en X' sur P0 :



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

Gestion des données réparties partagées

Gestion des Données Réparties Partagées
=
Résoudre des problèmes de cohérence

Convergence des Problèmes quelle que soit la base des solutions : matérielles ou logicielles

=

Algorithmique répartie

Cohérences Uniformes

Propriété de registre atomique

Cohérence des accès à la mémoire

Définition de Censier & Feautrier 1978 :

Une mémoire[8] est dite correcte si la valeur retournée par une opération de lecture à un certain emplacement de la mémoire est toujours la valeur qui correspond à la dernière écriture effectuée à ce même emplacement.

mémoire atomique[9]:

l'effet de l'écriture doit de plus être observé par tous les processeurs

Paradigme considéré par tout programmeur dans les architectures traditionnelles

[8] Une cellule mémoire peut être : un registre, une ligne de mémoire cache, une case de la mémoire principale

[9] ne pas confondre avec la cohérence atomique qui sera vue plus loin

Cohérence Forte vs *cohérence faible* (1)

Cohérence Forte :

La cohérence est plutôt attachée à l'environnement d'exécution[10] des programmes : Système et Matériel (cohérence couche basse)

=> la cohérence est assurée en toutes circonstances ... même quand ce n'est pas nécessaire

Cohérence Faible :

La **cohérence est plutôt à la charge du programme**, plus exactement du compilateur qui sait quand il doit y avoir cohérence, ... il marque de façon explicite les endroits où la cohérence est nécessaire(cohérence couche haute)

=> cohérence "autour" de points de synchronisation

[10] Aspect qui change avec les nouvelles architectures matérielles

Cohérence Forte vs *cohérence faible* (2)

Affaiblir la cohérence permet d'aller plus vite dans l'exécution des programmes.

En relachant la cohérence, on permet un plus grand parallèlisme (par une certaine réplication des données).

Les problèmes de **cohérence** se déplacent du matériel vers le logiciel et touchent le **modèle de programmation**.

=>

Concepteurs de systèmes et les concepteurs de langages

Typologie des Cohérences

L'utilisation d'une **cohérence dépend de l'application visée**, et de la sémantique de partage et d'accès aux données dont elle a besoin.

Cohérences Uniformes :

Les accès considérés sont les **lectures** et les **écritures**, les opérations de gestion de cohérence sont uniformément appliquées.

Adaptées aux applications dont on ne peut modifier le source, aux applications à comportement identifié mais sans coordination.

Cohérences Hybrides

En plus des lectures et des écritures, on considère les **opérations de synchronisation** : barrières, sémaphore, fork-join ...

Adaptées aux applications pour lesquelles on peut **contrôler la concurrence d'accès aux données**. Toute donnée partagée avec accès conflictuels doit être protégée par une section critique (DRF - Data Race Free) sinon les résultats obtenus sont imprévisibles.

Modèle d'Exécution

P1, ... , Pn processus s'exécutent[11] de façon concurrente et communiquent entre-eux par mémoire partagée.

Opérations sur un objet x par le processus Pi :

- la **lecture** retourne la valeur v : **ri (x) v**

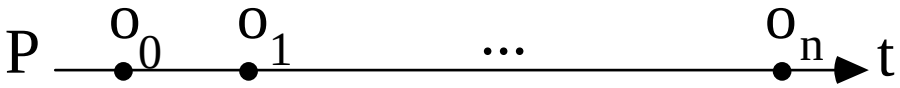
- l'**écriture** modifie à la valeur v : **wi (x) v**

simplification : 1 processus par processeur

[11] **Hypothèses sur le matériel :** (assez classiques)
- **Pas d'Etat Global**: pas d'horloge globale et de mémoire centrale
- Les processeurs sont reliés par un réseau et communiquent par des **canaux FIFO**
- Les processeurs ont une **mémoire privée locale,** c'est une mémoire de travail
- Pas de pannes
- Les processeurs ont des vitesses différentes (exécution asynchrone)

Ordre programme [9] [10]

Un processus est un programme qui exécute une suite d'opérations[12] les unes après les autres sur un même processeur P, ce qui peut se schématiser par :



ou plus formellement :
soit $\xrightarrow{po}$ une relation d'ordre[13] non réflexive qui concrètise le fait qu'une opération en précède une autre sur un processeur P :

oi $\xrightarrow{po}$ oj ssi oi se produit (juste) avant oj dans la suite d'opérations sur le processeur P (notion de précédence directe),

en fonction des cohérences mises en oeuvre, cet ordre peut être partiel ou total,
dans le cas de l'ordre total on a :

si ov?ow alors soit ov $\xrightarrow{po}$ ow, soit ow $\xrightarrow{po}$ ov

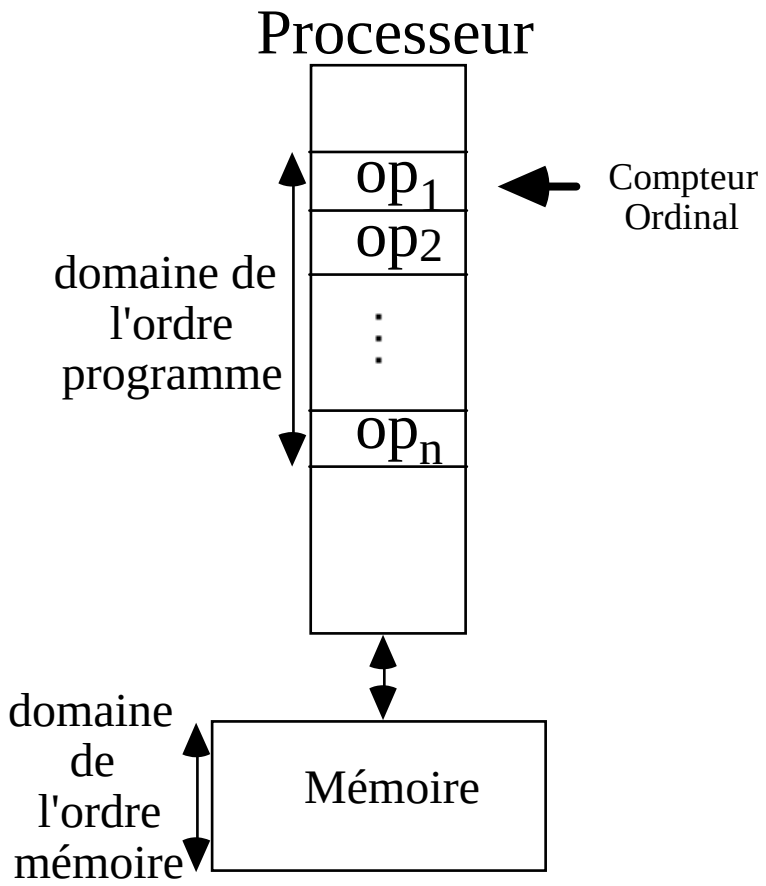
L'ordre programme peut être vu comme un ordre local au processeur. Si un programme est seul sur un processeur, et suivant le mode de fonctionnement de celui-ci, il peut se confondre avec l'ordre processeur. (Il faut avoir élminer le multi-threading, et les optimisations du compilateur).

---

[12] certains auteurs précisent "logiquement", dans ce cas il est fait référence à un ordre total

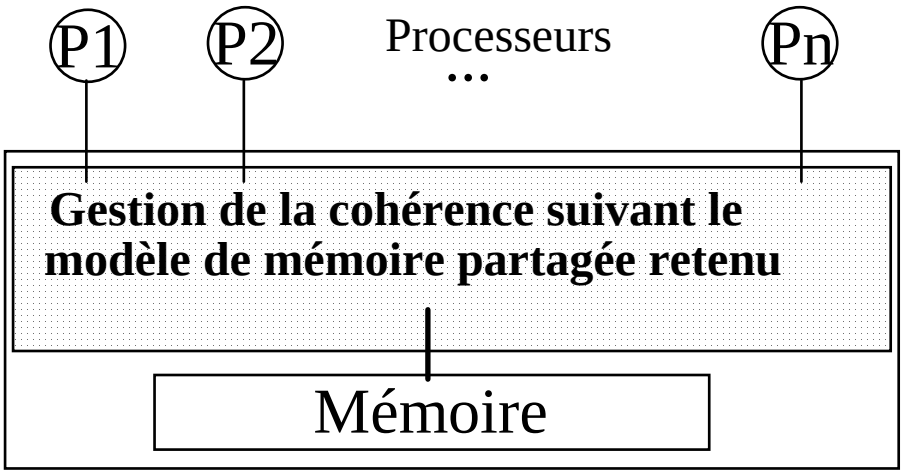
[13] une relation d'ordre est habituellement reflexive, antisymétrique, et transitive

Ordre mémoire en monoprocesseur

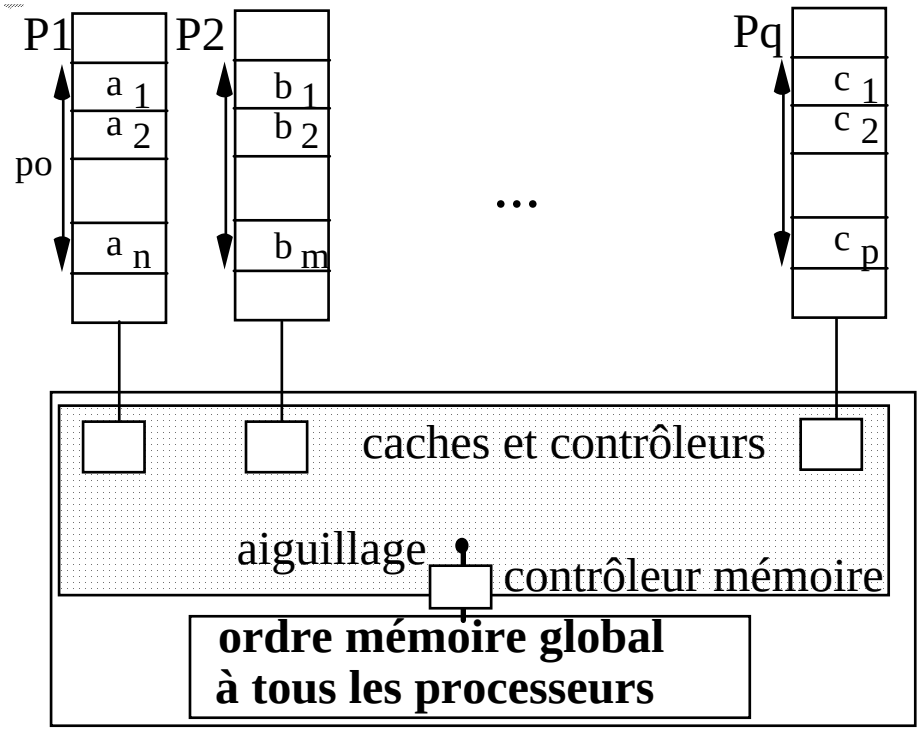


En monoprocesseur, même avec de la multiprogrammation, l'ordre mémoire (qui porte sur la suite des accès à la mémoire) est celui généré par le processeur.

Schématisation du modèle mémoire partagée



Système de Mémoire Partagée



Ordre mémoire en multiprocesseur [9]

Deux ordres associés à la mémoire partagée:

. ordre cellule mémoire

C'est un ordre associé à chaque emplacement de la mémoire. Il correspond à l'ordre dans lequel les opérations d'accès à une cellule mémoire sont effectuées. C'est un **ordre total**.

. ordre mémoire

C'est l'ordre dans lequel les opérations d'accès sont effectuées par la mémoire en temps réel. Toutes les opérations ne sont pas comparables, c'est un **ordre partiel**.

Problème de la cohérence mémoire en environnement multiprocesseur

mémoire à accès atomique : une mémoire est à accès atomique si le résultat de chaque écriture dans la mémoire est accessible (par une lecture) à tous les processeurs en même temps.

mémoire à accès non atomique : l'effet d'une écriture n'est pas perçu par tous les processeurs au même moment ... retard de la perception d'un signal du à la traversée du réseau d'interconnexion par exemple.

Trois catégories de multiprocesseurs peuvent être considérés :
. l'ordre programme est préservé[14], et tous les processus observent la même suite d'accès à la mémoire
. l'ordre programme n'est pas préservé[15], et tous les processus observent la même suite d'accès à la mémoire
. l'ordre programme n'est pas préservé, et les processus n'observent pas la même suite d'accès à la mémoire

Cohérence mémoire partagée répartie:

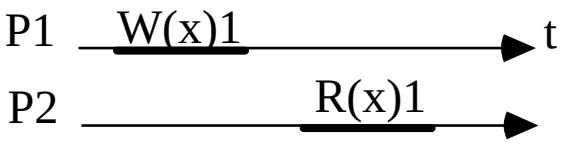
Le modèle de cohérence mémoire spécifie l'ordre dans lequel les accès à la mémoire effectués par un processus sont observés par l'ensemble des autres processus.

"Contrat entre la mémoire et le logiciel"

[14] l'ordre programme est un ordre total pour le processeur d'exécution
[15] l'ordre programme est un ordre partiel pour le processeur d'exécution

Cohérence atomique[16] (1)

Une opération de lecture ou d'écriture prend un certain temps, on peut le schématiser ainsi :



L'opération effective sur la mémoire peut avoir lieu à un instant quelconque dans l'intervalle de temps qui représente l'opération.

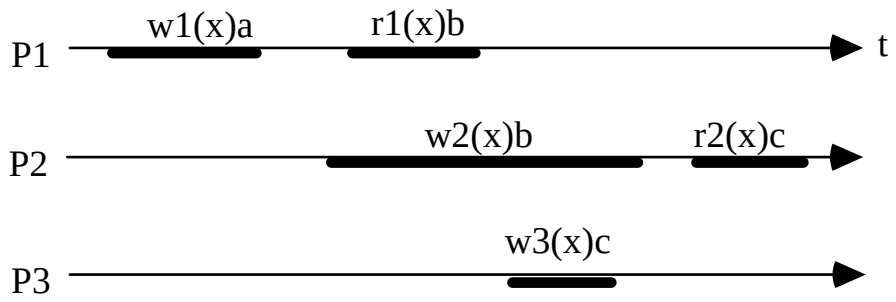
On considère l'ordre d'occurrence de ces opérations effectives observées par une horloge globale universelle virtuelle de granularité très fine (horloge absolue).

C'est de loin l'ordre le plus fort, c'est un modèle strict pas réellement implantable sinon à un prix prohibitif.

[16] appelée aussi propriété de linéarisabilité [12]

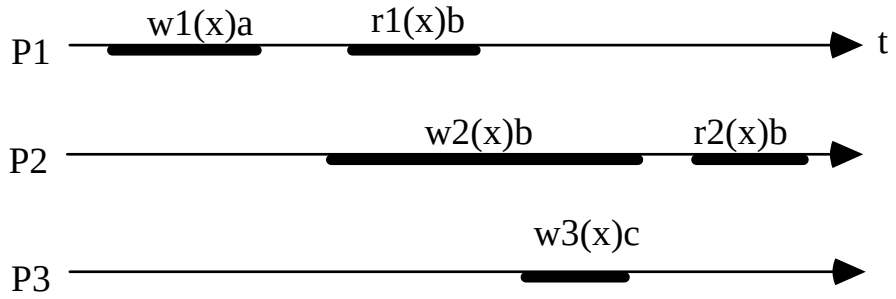
Cohérence atomique (2)

Une exécution s1[17]



S1 = w1(x)a w2(x)b r1(x)b w3(x)c r2(x)c

Une exécution s2



Exécution non atomique

Cohérence atomique préserve la Chronologie des événements

(ordre impliqué par la prise en compte d'un temps réel universel)

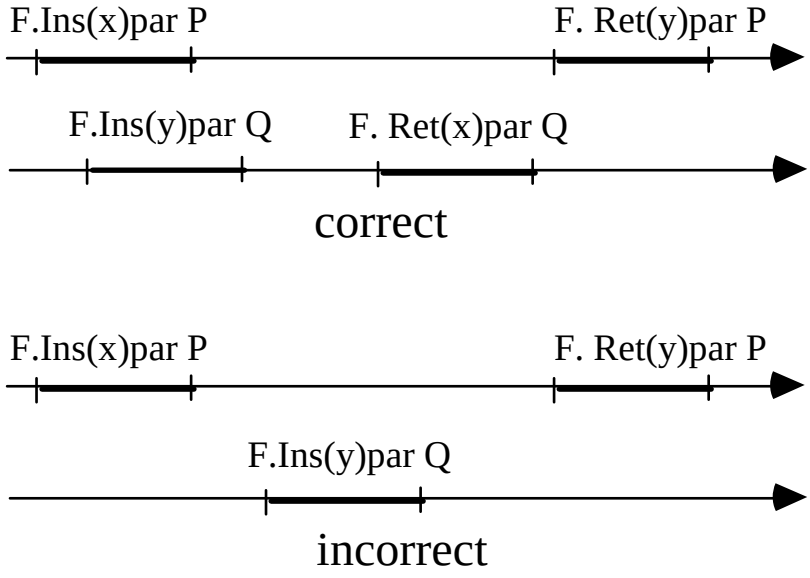
[17] Pour certains auteurs, ce qui compte, c'est le début pour une écriture et la fin pour une lecture (*repère le moment où l'effet de l'écriture est perçu*), pour d'autres c'est uniquement le début, c'est plus simple pour la cohérence atomique.

Linéarisabilité (1)

La linéarisabilité est une **autre formulation de la cohérence atomique**. Elle porte sur les objets d'un environnement de programmation accédés par des processus séquentiels concurrents:

objet = état + méthodes + spécification comportementale

Exemple : F une file FIFO avec les méthodes Ins pour insérer et Ret pour retirer



Chaque appel de méthode est délimité par son invocation (inv) et sa réponse (rep) ce qui définit les évènements contenus dans une exécution du système.

Linéarisabilité (2)

On prend H[18] l'ensemble des évènements d'une exécution.

On considère H muni de la relation $->_H$ telle que: $op_1 ->_H op_2$ si et ssi $rep(op_1)$ est avant $inv(op_2)$ dans H p/r au temps universel, à priori $->_H$ est un ordre partiel donc $(H, ->_H)$ n'est pas séquentiel.

$(H,->_H)$ est linéarisable si :
- il existe **S** un ensemble d'événements **équivalent à H** qui contient toutes les réponses aux opérations pendantes de H.
- il existe une relation d'**ordre total** $->_S$ sur les évènements de S ($(S,->_S)$ est séquentiel) tel que $->_H$ est inclu dans $->_S$.
- la spécification comportementale de tous les objets est respectée dans $(S,->_S)$ **-principe de légalité-**

Remarques :

- H/x est la restriction de H p/r à l'objet x. Si H/x est séquentiel, la spécification comportementale associée à l'objet x doit être respectée.
- H/P est la restriction de H p/r au processus P. H/P est séquentiel puisque $\xrightarrow{po}$ est un ordre total sur les évènements de P par hypothèse.

---

[18] Le premier événement de H est supposé être une invocation.

Linéarisabilité (3)

Théorème 1 :
(H,->H) est linéarisable si et ssi, pour tout objet x,
(H/x,->H) est linéarisable.
(la linéarisabilité est une propriété locale aux objets)

Théorème 2 :

Soit (H,->H) linéarisable. Si <inv(x.o)P> est l'invocation d'une opération définie sur un objet x par un processus P et est contenue dans H, il existe une réponse telle que H'=H.<rep(x.o)P>[19] (H',->H') est linéarisable.

Ces 2 théorèmes permettent d'établir que les algorithmes de gestion d'une mémoire répartie partagée proposés par Kai Li et al offrent une cohérence atomique :
Si l'objet considéré est la page, les méthodes sont les opérations de lecture et d'écriture, et l'algorithmique développée par les différentes solutions proposées par Kai Li et al linéarise les opérations qui portent sur les pages.

D'autres auteurs indiquent que les solutions de Kai Li et al offrent une cohérence séquentielle ... tout dépend de la façon dont on implante le mécanisme d'invalidation (communication synchrone ou asynchrone)

[19] Symbole de la concaténation

Cohérence Séquentielle[20] [7] (1)

Cohérence la plus connue, la plus maîtrisée

[Un système multiprocesseur est cohérent séquentiellement si] "le résultat de toute exécution est le même que si les opérations de tous les processeurs étaient exécutées dans un ordre séquentiel quelconque, et les opérations de chaque processeur apparaîtrait dans cette séquence dans l'ordre spécifié par leur programme ($\overrightarrow{po}$)"

il existe un ordre global (total) sur tous les accès à la mémoire qui préserve l'ordre programme

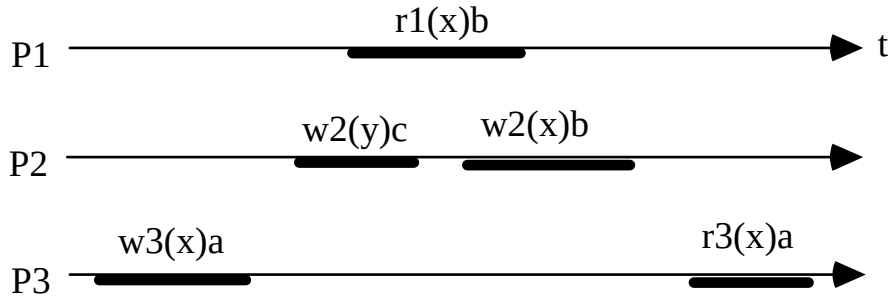
Idées d'implantation :

- On bloque tout processeur voulant accéder à une variable partagée en écriture tant que l'accès courant d'un processeur n'est pas complètement terminé.
- On sérialise les accès par un composant matériel (bus, réseau, séquenceur...)
- On conserve la valeur d'une variable tant que c'est nécessaire.

[20] La Cohérence Séquencielle est souvent comparée à la sérialisabilité(domaine des bases de données).
Point de vue discuté : la sérialisabilité ne s'occupe pas des valeurs prises par les données, elle ne traite que des conflits d'accès aux données.

Cohérence Séquentielle (2)

Une exécution s3



Exécution incompatible avec la cohérence atomique, mais cohérente séquentiellement :

S3 = w3(x)a r3(x)a w2(y)c w2(x)b r1(x)b

Ce qu'exprime la cohérence séquentielle :

Tous les objets et tous les processus sont d'accord sur un entrelacement des opérations d'une séquence, cette séquence peut différer de ce qui est réellement arrivé (au sens du temps chronologique), par contre, elle aurait pu se produire.

En fait, on a ordre programme + ordre total

Cohérence Causale [8][12][13] (1)

Causalité appliquée à la mémoire $\xrightarrow{co}$:

. si $op_1 \xrightarrow{po} op_2$ alors $op_1 \xrightarrow{co} op_2$

. si $op_1 = wj(x)v$ et $op_2 = ri(x)v$ alors $op_1 \xrightarrow{co} op_2$[21]
(il n'y a pas eu d'opération d'écriture entre op1 et op2 qui a modifié la valeur de x, principe de legalité de la lecture ri(x)v)

. si $op_1 \xrightarrow{co} op_2$ et $op_2 \xrightarrow{co} op_3$ alors $op_1 \xrightarrow{co} op_3$

C'est une adaptation de la causalité de Lamport pour les messages à la mémoire.

$\xrightarrow{co}$ est un ordre partiel, deux opérations non comparables par cet ordre sont dites concurrentes.

La cohérence causale est mise en oeuvre à l'aide d'horloges vectorielles.

[21] Remarque : "écriture" équivalent à envoi de message, "lecture" équivalent à réception de message

Cohérence Causale (3)

Definition [13]:

soit o = r(x) et o' = w(x)v, la valeur v est **viable/valide** pour o (on dit alors que v appartient à A(o)) si :

. o' est concurrent de o (o' $\not\xrightarrow{co}$ o et o $\not\xrightarrow{co}$ o')

. ou o' précède o sans qu'aucune lecture ou écriture délivrant une valeur v' soit intervenue entre o' et o. (o' $\xrightarrow{co}$ o et il n'existe pas o'' = r(x)v' ou o'' =w(x)v' tel que o' $\xrightarrow{co}$ o'' $\xrightarrow{co}$ o)

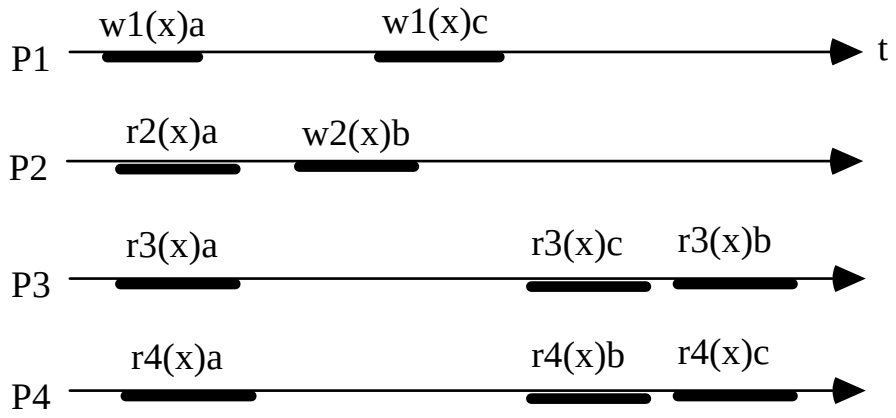
Definition :

Une exécution dans une mémoire causale est **correcte** si la valeur retournée par chaque opération de lecture dans l'exécution est **viable/valide** pour l'opération de lecture considérée.

pour tout o=r(x)v, v appartient à A(o)

Cohérence Causale (3)

Une exécution s4

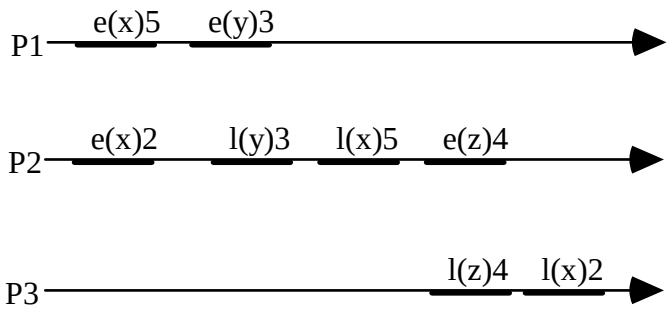


Exécution pas cohérente séquentiellement, mais cohérente causalement ...

Les dépendances causales sont bien respectées ... curiosité, P3 et P4 observent l'effet des écritures sur x dans un ordre différent.

La cohérence causale permet plusieurs écrivains. L'ordre d'observation des écritures peut différer d'un processeur à l'autre.

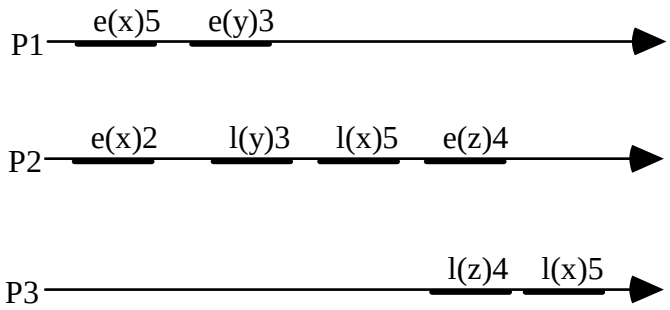
On observe la suite d'accès :



Cette exécution est-elle correcte pour une mémoire causale ?

Quelle devrait être l'exécution correcte ?

Solution :



Exécution correcte pour une mémoire causale

Les données sont répliquées sur tous les sites. On utilise une diffusion causale pour mettre à jour les données sur chacun des sites. L'utilisation d'une diffusion causale est-elle suffisante pour permettre une exécution correcte du point de vue de la cohérence causale ? Reprenez l'exemple précédent.

Exemple :

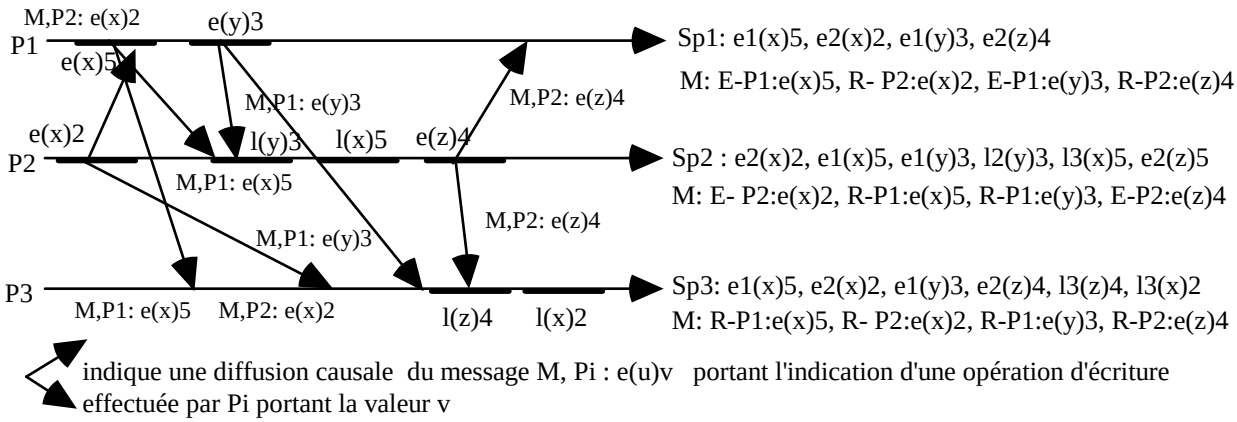


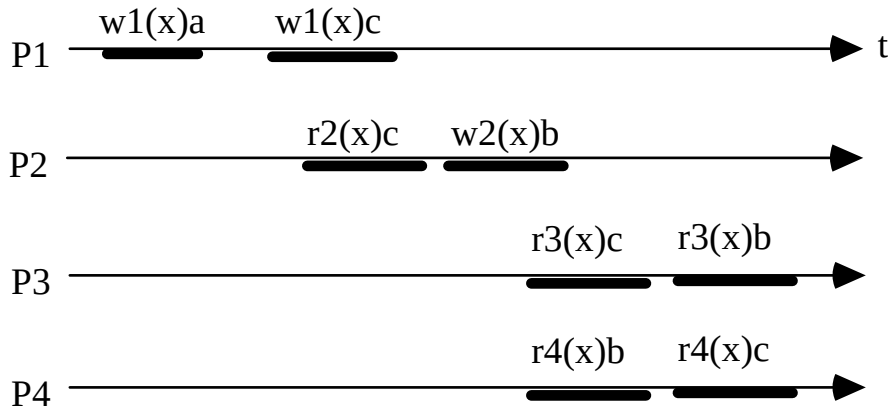
Figure précédente complétée avec les diffusions causales, les vues S_{pi}, les émissions E- et les réceptions de messages R-

plus d'informations dans RR95-01 (accès par serveur ftp.cnam.fr, répertoire Cedric)

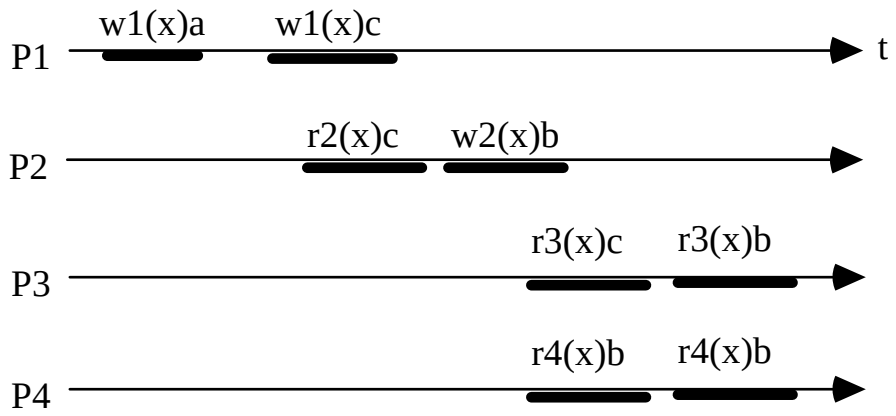
Cohérence PRAM[22] (Lipton & Sandberg 1988)

Les écritures d'un processeur sont toujours reçues par les autres processeurs dans le même ordre que son ordre programme, tandis que les écritures de différents processeurs sont reçues dans un ordre quelconque (ordre FIFO p/r à l'écrivain).

Une exécution s5



Exécution cohérente PRAM , mais pas cohérente causalement ... il faudrait :



[22] PRAM = Pipelined RAM, ne pas confondre avec Parallel Random Access Machine qui correspond à une architecture de machine parallèle

Cohérence PRAM - spécification formelle

D'après les travaux de [3]:

La cohérence PRAM est un affaiblissement de la cohérence causale, elle peut se définir à partir d'une relation d'ordre $\xrightarrow{pram}$ sur les accès mémoire:

. si $op_1 \xrightarrow{po} op_2$ alors $op_1 \xrightarrow{pram} op_2$

. si $op_1 = w_j(x)v$ et $op_2 = r_i(x)v$ alors $op_1 \xrightarrow{pram} op_2$[23]
(il n'y a pas eu d'opération d'écriture entre op1 et op2 qui a modifié la valeur de x)

. si $op_1 \xrightarrow{po} op_2$ et $op_2 \xrightarrow{po} op_3$ alors $op_1 \xrightarrow{pram} op_3$, mais ceci est déjà contenu/exprimé dans l'ordre programme

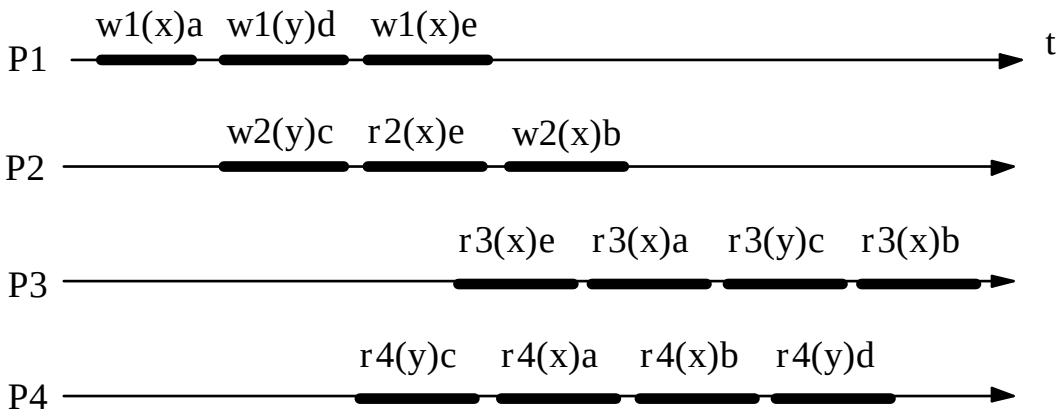
Remarque : la relation de transitivité ne "fonctionne" pas comme la définition de la relation d'ordre causal $\xrightarrow{co}$ sur les accès mémoire qui englobe plus d'opérations

[23] Remarque : "écriture" équivalent à envoi de message, "lecture" équivalent à réception de message

Cohérence Cache

Pour chaque emplacement x, il existe un ordre total des écritures sur x. Les écritures sur différents emplacements peuvent être vues dans un ordre différent par différents processeurs (Ordre FIFO sur l'emplacement mémoire).

Une exécution s6 :



Cache :

Ordre total sur x : e, a, b (ordre attribué dans une architecture NUMA par la gestion mémoire, la valeur a est connue après celle de e à cause du réseau maillé qui interconnecte les processeurs à la mémoire)

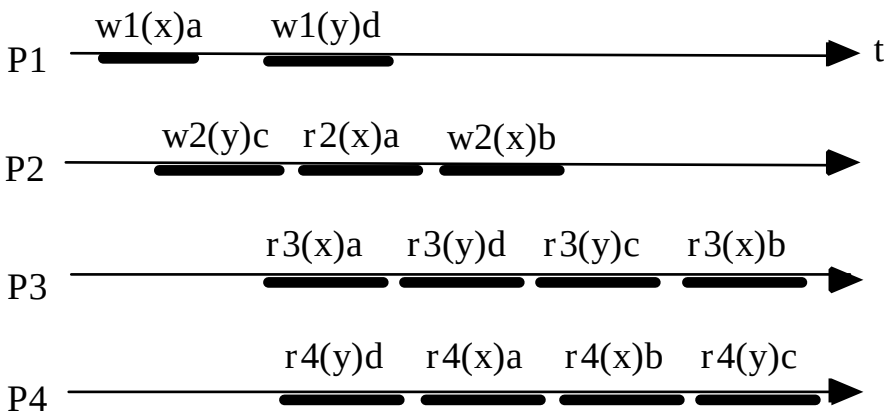
Ordre total sur y : c, d

ni séquentielle, ni causale, ni PRAM : l'ordre programme n'est pas respecté

Cohérence Processeur = Cohérences PRAM + Cache

Autres types de cohérences uniformes [11]

Exercice : Quelles cohérences ?



Cohérente Cache : Ordre sur x : a,b ; Ordre sur y : d,c

Cohérente PRAM : P1 et P2 n'écrivent qu'une fois sur des données différentes, pas assez d'écritures pour déterminer s'il y a des problèmes

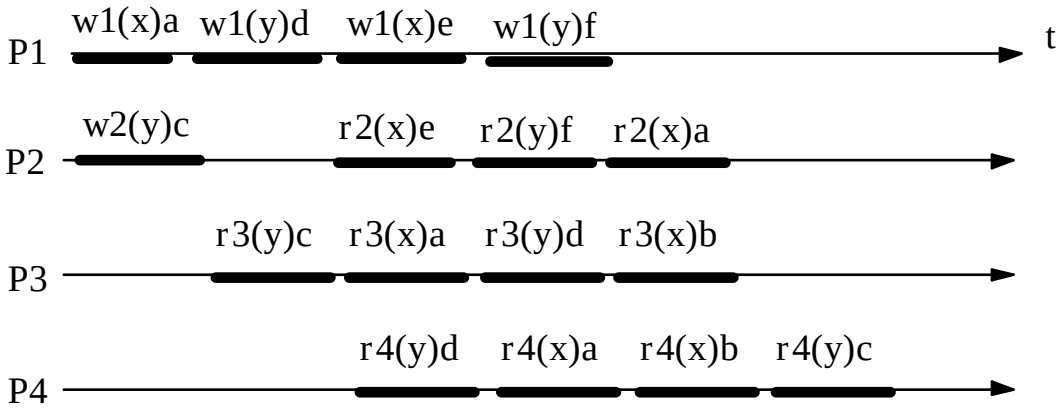
cohérente processeur car cohérent cache et cohérent PRAM

Cohérente Causale : respect de l'ordre causal des accès (ordre programme + ordre "lit de-read from")

Cohérente Séquentielle : ordre total sur les accès, w1(x)a, r3(x)a, w1(y)d, r4(y)d, r3(y)d, r4(x)a, w2(y)c, r2(x)a, r3(y)c, w2(x)b, r4(x)b, r3(x)b, r4(y)c

Cohérence séquentielle.

Quelles cohérences ?



Cohérences Hybrides

Faiblesse des Cohérences Uniformes

Même si le modèle de cohérence est affaibli, **une cohérence uniforme implique le maintien du même modèle sur toutes les opérations d'accès**, et surtout sur les écritures.

C'est un avantage pour certaines applications, mais quand le programmeur spécifie l'ordre d'accès à ses données à l'aide de **sections critiques**, ou de variables de synchronisation, il y a **double emploi**.

L'idée est d'utiliser les indications sur la synchronisation d'accès aux données pour gérer la cohérence, les opérations de mise en oeuvre de la cohérence sont faites à ce moment.

Hypothèses des mémoires à cohérence hybride:

- Puisque l'utilisateur sait ce qu'il fait, le système de gestion mémoire n'a pas besoin de s'occuper de la cohérence des données hors des moments où c'est indiqué.

- Si un utilisateur ne spécifie pas ses sections critiques, les résultats obtenus ne sont pas corrects. Le système n'en est pas responsable.

Cohérence Faible - *Weak Memory* [5][6] (1)

Conditions de mise en oeuvre [5] :

1. L'accès aux variables de synchronisation respecte la cohérence séquentielle
2. Aucun accès à une variable de synchronisation ne peut être fait tant que toutes les lectures ou écritures déjà commencées ne sont pas toutes terminées.
3. Aucune lecture ou écriture ne peut être effectuée par un processeur tant qu'un accès à une variable de synchronisation n'a pas été terminé.

Commentaires :

1. indique que **tous les processeurs** voient l'accès aux variables de synchronisation dans le même ordre (par une diffusion par exemple) et se bloquent tant que l'accès à une variable de synchronisation en cours n'est pas terminé, **la cohérence séquentielle implique une coordination globale**
2. force toutes les écritures en cours à se terminer, quand l'opération de synchronisation s'est faite on est certain que toutes les écritures ont été faites
3. indique que les opérations d'accès après une opération de synchronisation utilisent les données les plus récentes

Cohérence Faible [5][6] (2)

$\xrightarrow{hb}$ [24] = $\xrightarrow{po} \cup \xrightarrow{so}$

où $op_1 \xrightarrow{so} op_2$ ssi op_1 et op_2 sont des opérations de synchronisation sur la même variable s et $op_1(s)$ s'est terminé avant $op_2(s)$ dans l'exécution

Conditions de mise en oeuvre (DRF0) [6] :

1. Toutes les opérations de synchronisation sont reconnues par le matériel, et chaque opération d'accès ne concerne qu'un seul emplacement de la mémoire (étiquetage des opérations)
2. Toutes les opérations d'accès respectent l'ordre programme, toutes les opérations en conflit (l'une d'elles au moins est une écriture) sont ordonnées par la relation $\xrightarrow{hb}$

=> **Etiqueter toutes les opérations d'accès**

Commentaire :

Lors d'opérations de synchronisation, on veut limiter le bloquage aux processeurs uniquement concernés par la variable impliquée, les autres peuvent poursuivre leur exécution.

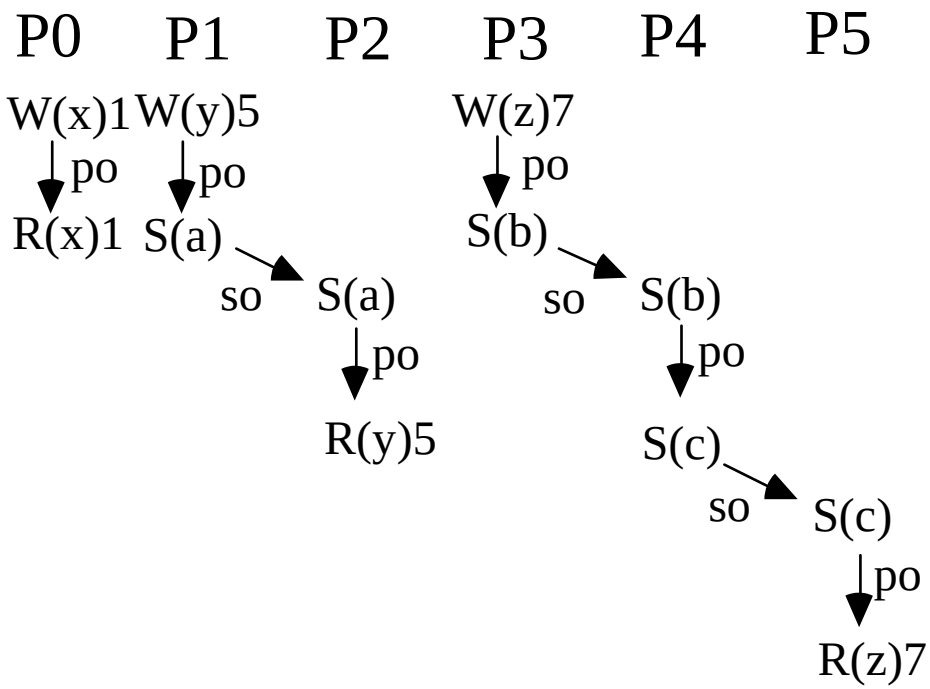
Cohérence faible = oblige le programmeur à respecter la condition DRF (propriété non décidable sur un programme) mais augmente la performance

---

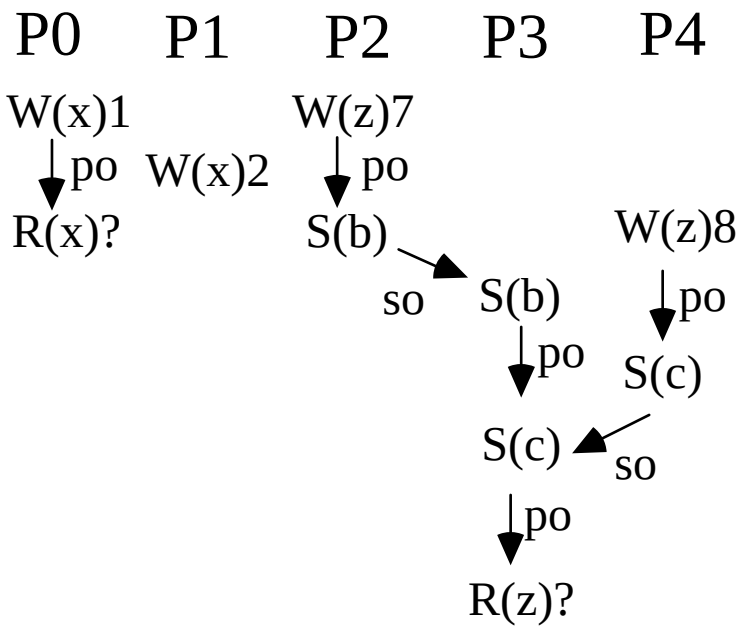
[24] hb pour happen before

Exemples d'exécutions p/r DRF0

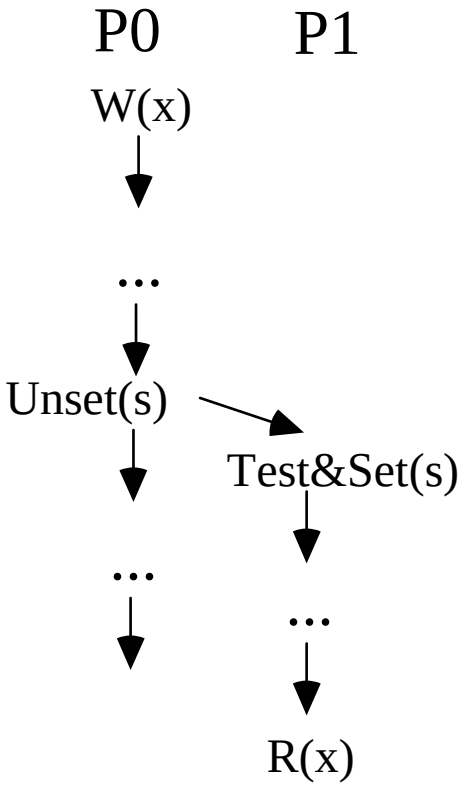
Une exécution correcte :



Une exécution incorrecte : des accès conflictuels ne sont pas ordonnés par la relation $\xrightarrow{hb}$



Eléments d'Implantation de la cohérence faible



Remarques :

- On a deux opérations de synchronisation qui sont considérées de la même façon mais qui ne portent pas la même sémantique pour un programmeur.
- Avec [6], seuls P0 et P1 sont concernés par l'opération de synchronisation.

- Après "Unset(s)", P0 poursuit sont exécution.

Etiquetage des données pour spécifier la concurrence d'accès [4]

synchronisation DRF (Data Race Free) :
Soit un emplacement x, soit opDRF(x) et op'DRF(x) deux opérations conflictuelles (une des opérations est une écriture), on a :

$$op_{DRF}(x) ->_{25} op'_{DRF}(x) \text{ ou } op'_{DRF}(x) -> op_{DRF}(x)$$

synchronisation par objet

synchronisation CWF (Concurrent Write Free) :
Soit deux emplacements x et y, soit $w_{CWF}(x)$ et $w_{CWF}(y)$ deux écritures, on a :

$$w_{CWF}(x) -> w_{CWF}(y) \text{ ou } w_{CWF}(y) -> w_{CWF}(x)$$

synchronisation entre objets

synchronisation CRF (Concurrent Read Free):
Soit deux emplacements x et y, soit rCRF(x) et rCRF(y) deux écritures, on a :

$$r_{CRF}(x) -> r_{CRF}(y) \text{ ou } r_{CRF}(y) -> r_{CRF}(x)$$

synchronisation entre objets

On peut combiner **DRF et CRF** ou **DRF et CWF**.

Toutes les opérations sont étiquetées. Le système peut déterminer le meilleur entrelacement des opérations d'accès. Ce type de technique est plutôt utilisé par les compilateurs pour des multiprocesseurs (ex DASH).

---

[25] la relation d'ordre n'est pas spécifiée mais devrait l'être, et dépend du système, $\xrightarrow{hb}$ pourrait convenir par exemple

Evolution contenue dans le modèle Cohérence Faible

La cohérence faible ne donne pas de sémantique particulière à une opération de synchronisation: **la gestion mémoire ne sait pas si la synchronisation matérialise un début ou une fin de section critique**. En fait une indication connue du programmeur n'est pas spécifiée.

Par conséquent, la gestion mémoire sur un processeur doit prévoir les deux cas : terminer ses écritures locales et les propager vers d'autres processeurs, et attendre le résultat des écritures d'autres processeurs.

L'opération de synchronisation dans le modèle Cohérence Faible porte la sémantique de deux opérations :

1. la protection de données par une section critique

2. le phasage d'algorithmes parallèles

1. mène aux verrous avec des opérations d'acquisition et relâchement

2. mène aux barrières

Cohérence relâchée DASH - *Release Consistency*

La Cohérence Relâchée (RC) de DASH distingue les primitives de synchronisation :

deux primitives sur les verrous:

- **Acquire(S)** : entrée dans une section critique
- **Release(S)** : sortie d'une section critique

Les données accédées dans une section critique sont dites protégées.

Avec un modèle de Cohérence Relachée on peut avoir aussi une primitive de phasage :

- Barrière(B) : rendez-vous qui permet de phaser les programmes parallèles

  arrivée/bloquage sur B ˜ acquire

  départ/débloquage sur B ˜ release

Cette cohérence est parfois appelée *Eager Release Consistency* (ERC) qui peut se traduire par "cohérence relâchée avec effets au plus tôt". On la trouve dans Munin.

Principes de la RC

Principe :

- lors de l'**Acquire**, les données dont on va avoir besoin vont être **mises à jour** et seront cohérentes sur tous les processeurs qui disposent d'un exemplaire
- lors du **Release**, les données qui ont été modifiées dans la section critique sont **propagées** vers les autres processeurs qui ont un exemplaire.

Pour augmenter les performances lors de l'implantation, ce modèle fait l'hypothèse que les données sont répliquées sur les processeurs qui les utilisent.

Attention :

- un Acquire ne propage pas immédiatement les mises à jour locales vers d'autres processeurs
- un Release ne provoque pas nécessairement la mise à jour des données locales qui auraient été modifiées ailleurs.

Conditions de mise en oeuvre d'une RC

Conditions de mise en oeuvre d'une RC par un système de gestion mémoire :

1. L'accès aux variables de synchronisation lors d'opérations Acquire et Release doit respecter la cohérence processeur

2. Avant de pouvoir commencer un accès normal à une donnée partagée protégée, tous les Acquire doivent avoir été complètement terminés.

3. Avant de pouvoir commencer un Release, toutes les opérations d'écriture et de lecture d'un processeur doivent avoir été complètement terminés.

Si ces conditions sont respectées et si le programmeur spécifie correctement toutes les sections critiques (condition DRF0) alors une exécution faite sous la **cohérence relâchée est équivalente à une cohérence faite sous une cohérence séquentielle**.

Cohérence relâchée paresseuse TreadMarks - *Lazy Release Consistency* (LRC) [17]

autre traduction : cohérence au relâchement

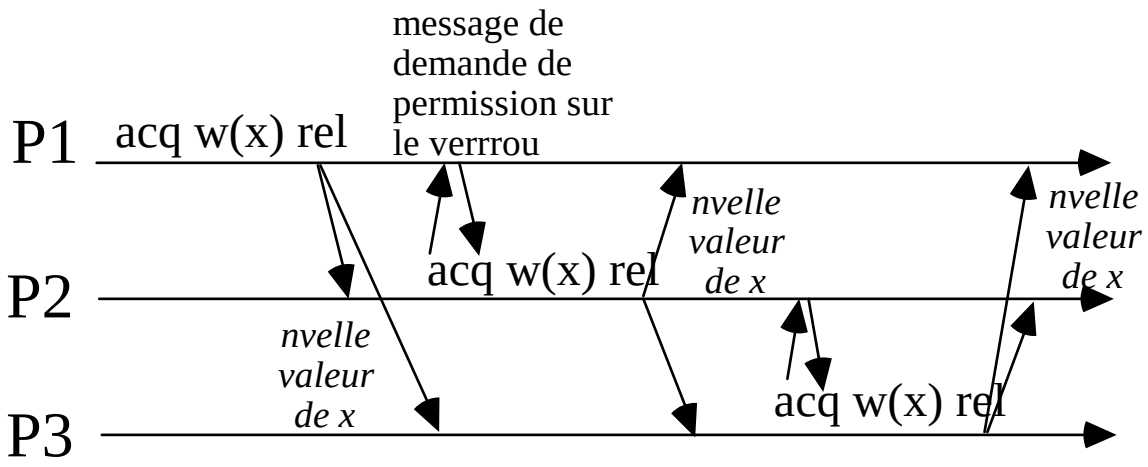
L'idée est que si on met le résultat de l'accès aux données au moment où on sort d'une section critique à disposition de tous ceux qui ont un exemplaire d'une donnée dans leur cache, on risque de travailler pour rien.

La mise à disposition des résultats est reportée le plus tard possible avec la cohérence relâchée paresseuse... c'est à dire jusqu'à la prochaine fois où on a vraiment besoin des données, donc lors du prochain Acquire. Au moment du Release, aucune modification n'est donc propagée.

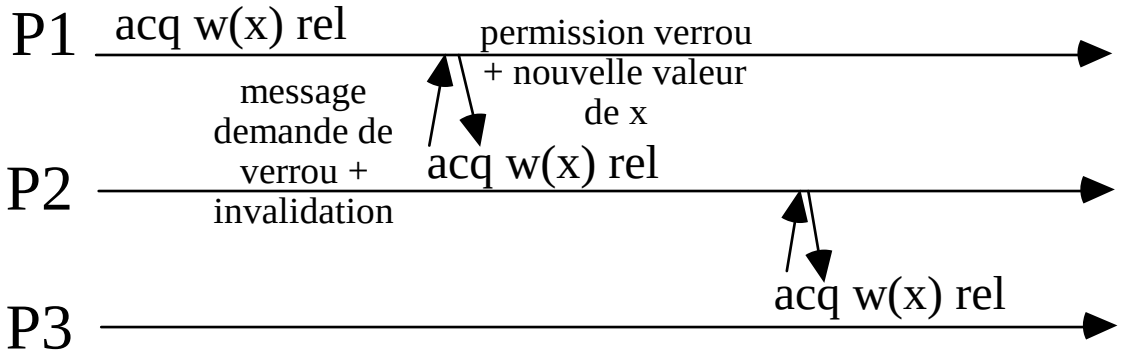
Différence entre ERC et LRC

Munin : Eager Release Consistency

On suppose que x est répliqué sur tous les processeurs



TreadMarks : Lazy Release Consistency



Cohérence relâchée causale - *Causal Release Consistency* [16]

A la définition de l'ordre causal mémoire, on ajoute les effets de la prise en compte des opérations de synchronisation, on aboutit à l'ordre causal étendu $\xrightarrow{eco}$:

On a donc toujours la relaton d'ordre $\xrightarrow{co}$:

. si $o_1 \xrightarrow{po} o_2$ alors $o_1 \xrightarrow{co} o_2$

. si $o_1 = w_j(x)v$ et $o_2 = r_i(x)v$ alors $o_1 \xrightarrow{co} o_2$

. si $o_1 \xrightarrow{co} o_2$ et $o_2 \xrightarrow{co} o_3$ alors $o_1 \xrightarrow{co} o_3$

Les auteurs ajoutent les relations d'ordre suivantes sur les accès mémoire :

. l'**ordre verrou** : $o_1 \xrightarrow{lo} o_2$ quand o_1 et o_2 sont telles que o_1 précède immédiatement un relâchement de verrou et o_2 suit immédiatement la prise de verrou correspondante, les sémaphores sont couverts aussi par $\xrightarrow{lo}$,

. l'**ordre barrière** : $o_1 \xrightarrow{bo} o_{k,j}$ avec k=1,...,n quando_1 précède immédiatement une prise de barrière par n processus et quand les $o_{k,j}$ k=1,...,n suivent immédiatement la fin de rendez-vous dans les processus Pk,

. l'**ordre fork** : $o_1 \xrightarrow{fo} o_2$ quand o_1 précède immédiatement une opération fork et o_2 est la première opération d'accès à la mémoire effectuée par le processus fils.

Cohérence relâchée causale

Les opérations de synchronisation induisent un nouvel ordre. Soit o_1 et o_2 deux opérations d'accès à la mémoire, $o1 \xrightarrow{so} o_2$ si :

- $o_1 \xrightarrow{po} o_2$ ou
- $o_1 \xrightarrow{lo} o_2$ ou
- $o_1 \xrightarrow{bo} o_2$ ou
- $o_1 \xrightarrow{fo} o_2$ ou
- il existe o' tel que $o_1 \xrightarrow{so} o'$ et $o' \xrightarrow{so} o_2$

Deux opérations sont ordonnées suivant l'ordre causal étendu, $o_1 \xrightarrow{eco} o_2$, si :

- $o_1 \xrightarrow{co} o_2$ ou
- $o_1 \xrightarrow{so} o_2$ ou
- s'il existe une opération o' tq $o_1 \xrightarrow{eco} o'$ et $o' \xrightarrow{eco} o_2$".

Cohérence Relachée Causale + Spécification des contraintes de Synchronisation = Cohérence Séquentielle [4]

Exemples de Bases de Mise en oeuvre logicielle des DSM

- Ivy
- Saffres

à compléter :

- Munin
- TreadMarks
- Orca
- Linda
- Larchant
- Sirac

Mémoire virtuelle répartie partagée

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 :

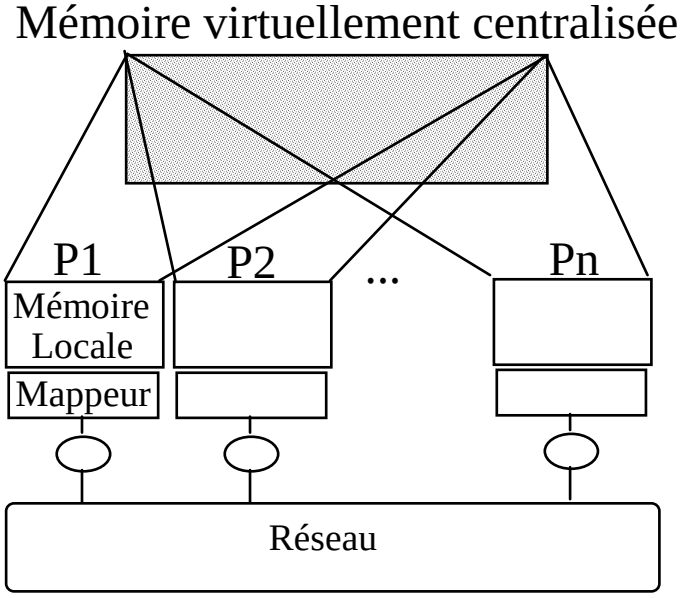
Unité de gestion mémoire (MMU[26])

-> 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

[26] Memory Management Unit

Modèle IVY



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.

Modèle d'implantation

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

Hypothèses :
- les données peuvent être répliquées
- les données peuvent migrer vers l'utilisateur, une seule copie de référence dans le système

=>1 écrivain ou plusieurs lecteurs

fondé sur la notion de **propriétaire** de la copie de référence, utilisant un mécanisme d'**invalidation sur écriture**[27].

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

Il existe une fonction de localisation du site détenteur de la copie de référence.

[27] Adaptation du protocole de Berkeley [4] pour multiprocesseur à mémoire commune à un réseau de stations

Localisation par un serveur (1)

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}[28] : n'a un sens que si le site est propriétaire de la page

Un verrou n'est relâché 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.

- pour le Serveur :

Info

N° page	propriétaire

Verrou Serveur

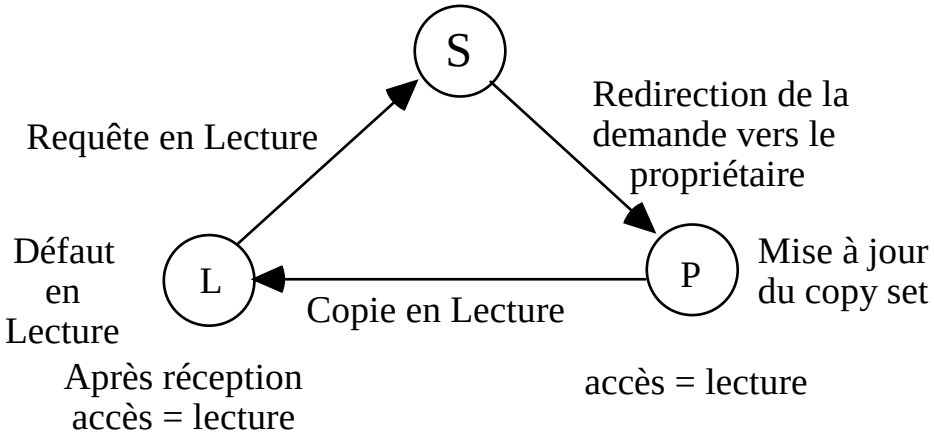
propriétaire : N° du site écrivain le plus récent
Verrou-Serveur : pour verrouiller les demandes des différents sites

[28] encore appelé *copyset*

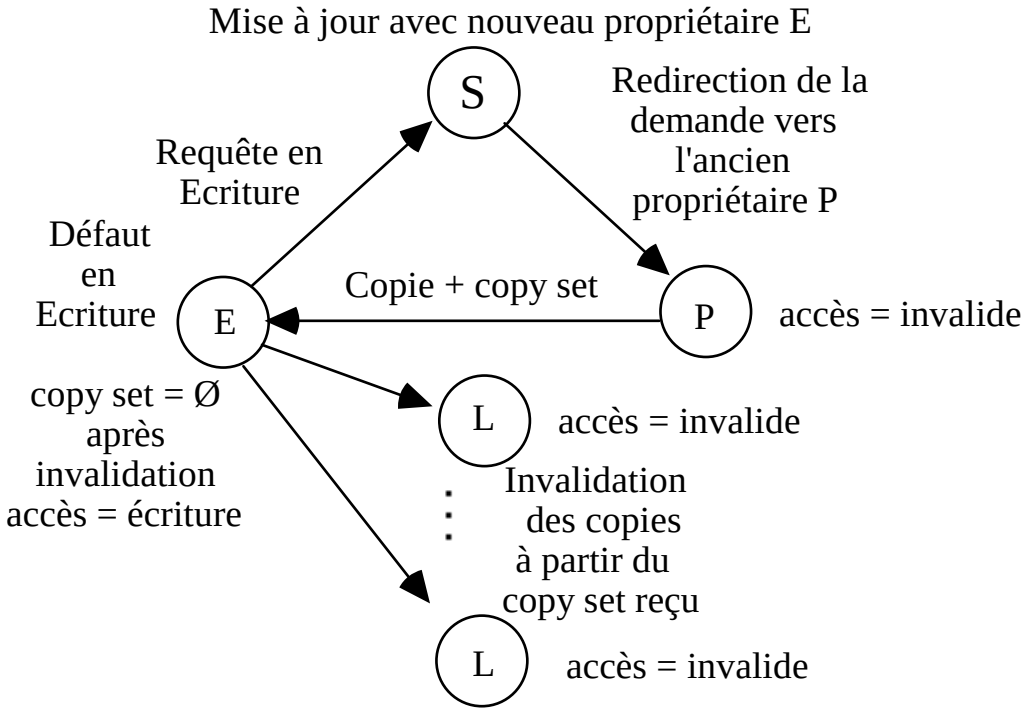
Gestion de la cohérence stricte

Protocole

Défaut en Lecture :



Défaut en Ecriture :



Distribution du serveur de localisation

Chaque processeur est le serveur de localisation d'un sous-ensemble prédéfini de pages (statique).

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

le numéro du serveur 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.

Localisation par 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

Algorithme dans le papier de Kai Li et al ne fonctionne pas.

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

Heuristique de localisation(1)

Il n'y a pas de serveur de localisation, on a une heuristique qui consiste à interroger une liste des propriétaires probables de la copie de référence

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.

Heuristique de localisation(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 :

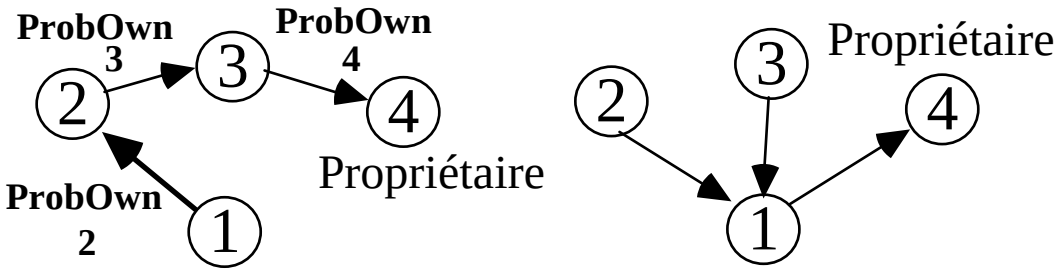
. en lecture il modifie l'accès, met à jour le copy set, et envoie une copie,

. en écriture il invalide localement, et envoie la page avec le copyset au demandeur, il mémorise que ce dernier devient le propriétaire probable.

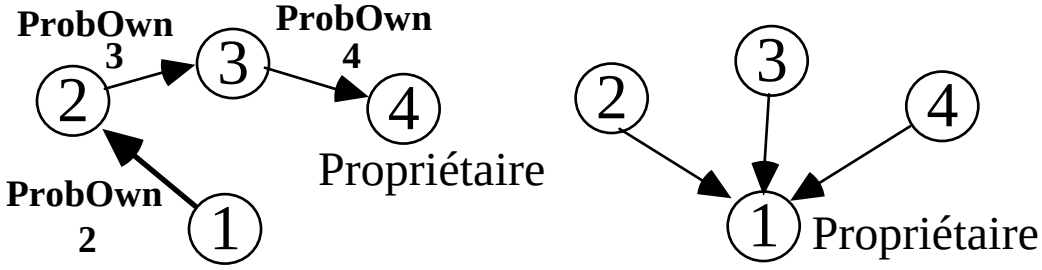
Sur réception de la page, le demandeur invalide et devient le vrai propriétaire, le copyset est vidé.

Heuristique de localisation (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[29], en moyenne O(N + K log N) après K demandes d'une page. En général pas plus de deux processeurs partagent la même page.

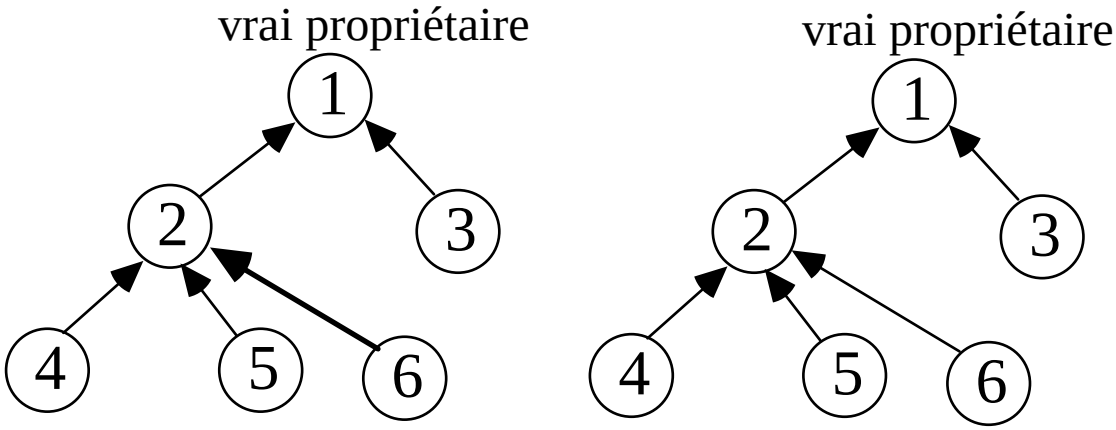
[29] Juste après une recheche du propriétaire qui a parcouru toute la chaine des propriétaires probables. Les champs de la table des pages de ces sites a été mis à jour avec la valeur du demandeur.

Heuristique de localisation avec gestion répartie de la liste des copies (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.

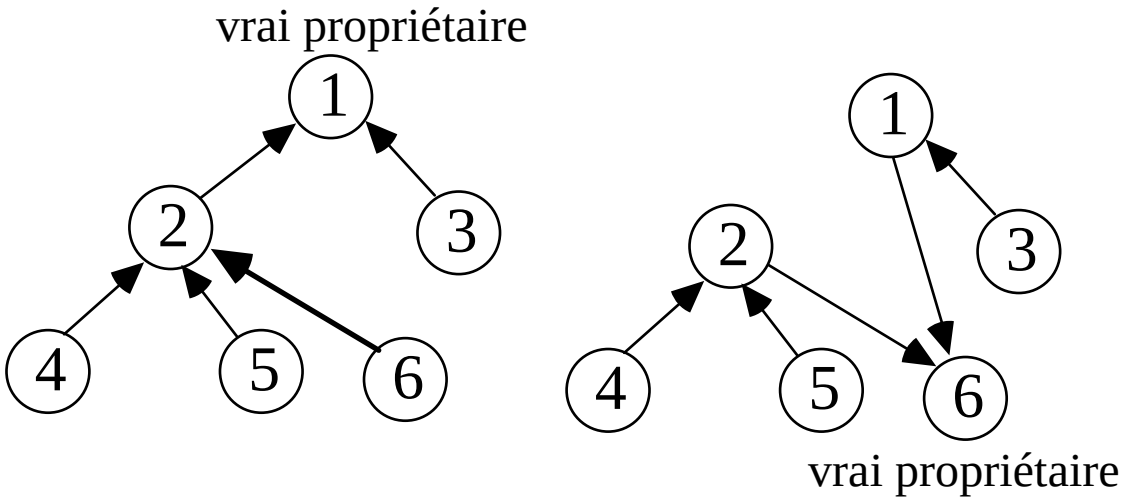
Maintenant un site qui détient un exemplaire de la page peut fournir la page au demandeur en lecture. Il ajoute alors le demandeur au copyset.

graphe des propriétaires probables :



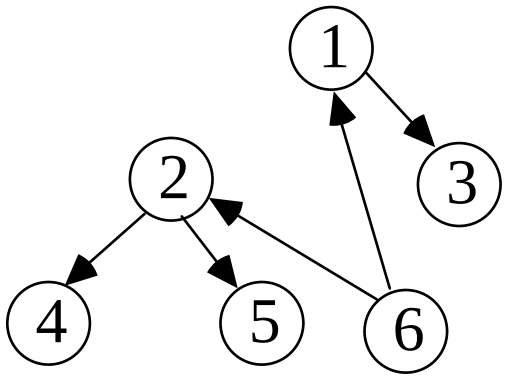
6 effectue un défaut en lecture et est servi par 2

Heuristique de localisation avec gestion répartie de la liste des copies (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.

Problèmes des protocoles de gestion de la cohérence en Mémoire Partagée Répartie

. 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 : Les données contenues dans une page ne sont pas partagées physiquement entre deux programme concurrents mais ne correspondent pas à un partage applicatif.

. DOUBLE FAUTE : Quand un défaut en lecture est suivi d'un défaut en écriture, la page est demandée 2 fois au propriétaire

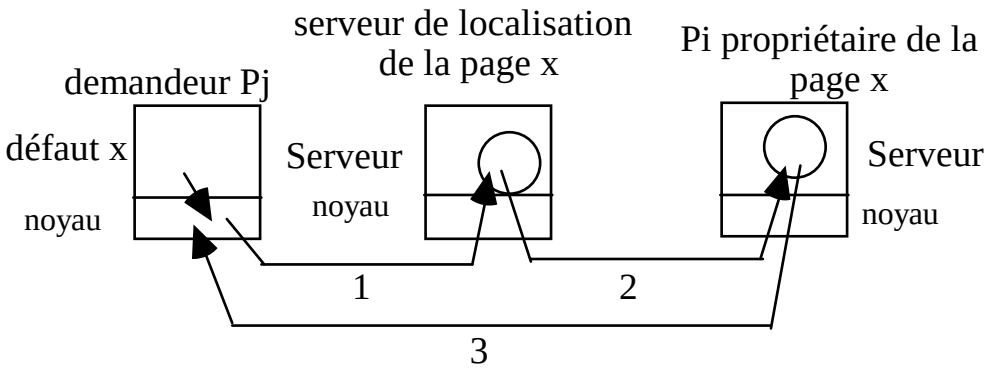
Remarque : Ces problèmes existent aussi en multiprocesseur.

Maintien de la Cohérence Causale, gestion par horloges vectorielles

La cohérence causale permet plusieurs écrivains et plusieurs lecteurs, pour des raisons d'implantation on se limite à 1 écrivain et n lecteurs.

Cet algorithme utilise le principe des **horloges vectorielles**.

Le modèle d'implantation reprend le schéma Serveur de localisation distribué de Kai Li et al.



1 : demande de page vers le gestionnaire (statique) de la page x ,
2 : redirection de la demande vers le propriétaire de la page x si le serveur de localisation n'est pas propriétaire
3 : réponse à la demande concernant la page x

LIRE : demande de page en lecture (1),
ECRIRE : demande de page en écriture (1),
REDIRIGE_ECRIRE : redirection d'une demande de page en écriture vers le propriétaire (2),
REDIRIGE_LIRE : redirection d'une demande de page en lecture vers le propriétaire (2),
REPONSE : arrivée de la page demandée et d'informations pour maintenir la cohérence causale (3),

Structures de Données de l'algorithme

Chaque serveur maintient une table des pages qui indique pour chaque page x :

x.acces : le type de l'accès (lecture, ecriture ou nul),

x.proprietaire : le propriétaire de la page, ce champ n'est utile que quand le serveur est propriétaire effectif de la page,

x.contenu : contenu de la page en mémoire physique, chargement par "x.contenu = contenudex"

x.estampille : contient la valeur de l'horloge vectorielle du site sur lequel s'est produit la dernière écriture au moment où celle-ci s'est produite, sert à matérialiser **la relation de dépendance causale entre une version d'une page (dernière écriture sur celle-ci) et toute copie**.

Le serveur de localisation d'une page x est défini à l'initialisation du système et est indiqué dans le tableau gestionnaire[x].

La section critique qui protège toute manipulation d'information d'une page na figure pas dans les algorithmes.

Pour gérer la cohérence causale, chaque serveur sur un **site i** maintient une horloge vectorielle **HF&Mem_i**. Celle-ci évolue en fonction des demandes de pages.

Traitements liés aux lectures

- défaut de page : accès en lecture
à la page x sur le site i

```
li(x)v ::
01. envoyer(LIRE,i,x) à gestionnaire[x]
02. recevoir(REPONSE,x,contenudex,HF&M')
03. x.contenu := contenudex
04. x.acces := lecture
05. x.estampille := HF&M'
06. HF&M_i := max(HF&M_i,HF&M')
07. invalider(HF&M')
```

- serveur de défauts de page: requête de lecture
d'un site j au site i pour la page x

```
[LIRE, x] ::
08. recevoir(LIRE,j,x)
09. si x.proprietaire = = i alors
10.       x.acces := lecture
11.       envoyer(REPONSE,x,contenudex,x.estampille) vers j
12. sinon
13.      envoyer(REDIRIGE_LIRE,j,x) vers x.proprietaire
14. finsi
```

Traitements liés aux redirections et aux invalidations

- serveur : redirection d'une requête d'un site j pour le site i à propos de la page x

38. recevoir (REDIRIGE,j,x)
39. x.acces := lecture
40. si REDIRIGE = = REDIRIGE_ECRIRE alors
41. x.propriétaire :=j
42. finsi
43. envoyer(REPONSE,x,contenudex,x.estampille) vers j

- invalidations

invalider(V)

44. pour tout y telle que y.estampille < V
et y. acces = = lecture
et y.propriétaire ° i
45. y. acces := nul

Traitements liés aux écritures

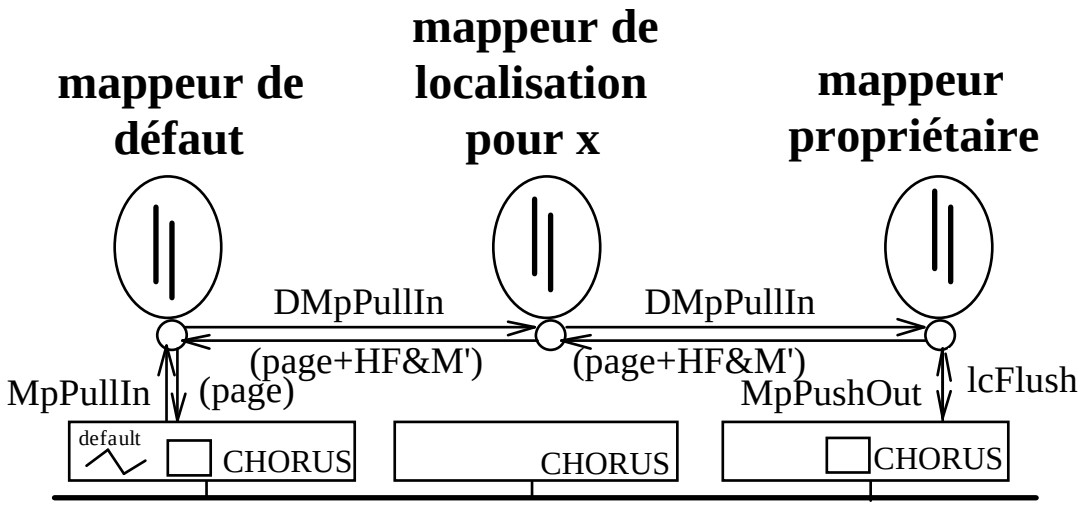
- défaut de page : accès en écriture à la page x sur le site i

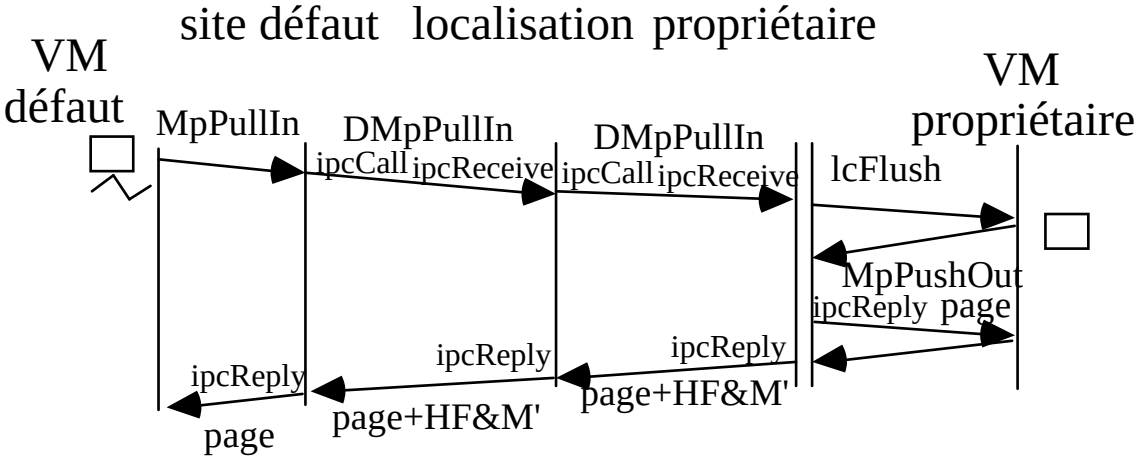
```
ei(x,v)::
15. si (x.propriétaire == i) alors
16.     x.acces := écriture
17.     HF&M_i[i] := HF&M_i[i] + 1
18.     x.estampille := HF&M_i
19. sinon
20.     envoyer(ECRIRE,i,x) à gestionnaire[x]
21.     recevoir(REPONSE,x,contenudex,HF&M')
22.     x.contenu := contenudex
23.     x.acces := écriture
24.     x.propriétaire := i
25.     HF&M_i := max(HF&M_i, HF&M')
26.     HF&M_i[i] := HF&M_i[i] + 1
27.     x.estampille := HF&M_i
28.     invalider(HF&M')
29. finsi
```

- serveur de défauts de page: requête d'écriture d'un site j au site i à propos de la page x

```
[ECRIRE, x] ::
30. recevoir(ECRIRE,j,x)
31. si x.proprietaire = = i alors
32.       x.acces := lecture
33.       envoyer(REPONSE,x,contenudex,x.estampille) vers j
34. sinon
35.      envoyer(REDIRIGE_ECRIRE,j,x) vers x.proprietaire
36. finsi
37. x.propriétaire := j
```

Cohérence Causale par vecteur sur CHORUS





Conclusion

Le problème de la cohérence des données est toujours actuel.

Il se pose par exemple dans le contexte du Cloud Computing.

Il se pose, de façon proche de celle du cours dans les architectures multi-cœur.

Un domaine prometteur !

Références Bibliographiques :

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

[2] Distributed Operating Systems. A. Tanenbaum . Prentice Hall 1995.

[3] A suite of formal Definitions for consistency Criteria in distributed shared memories.M. Raynal, A. Schiper. PI 968. IRISA. Novembre 1995.

[4] From Causal Consistency to Sequential Consistency in Shared Memory Systems. M. Raynal, A. Schiper. PI . IRISA. 1995.

[5] Memory Access Buffering in Multiprocessors M. Dubois, C. Scheurich, F. Briggs. Proc. 13th Ann Int. Symp. on Computer Architecture. ACM. pp434-442. 1986.

[6] Weak Ordering : a New Definition. S. Adve, M. Hill. Proc. 17th Ann Int. Symp. on Computer Architecture. ACM. pp2-14. 1990.

[7] How to make a Multiprocessor Computer that correctly executes Multiprocess Programs.Leslie Lamport. IEEE TOC. V C-28. N9. 1989.

[8] Memory Consistency Models. David Mosberger. ACM Op. Syst. Rev. 1993.

[9] Formal Specification of memory Models. P.S. Sindhu, J-M Frailong, C. Cekleov. Scalable shared-Memory Multiprocessors. Kluwer Academic Publishers, Boston, MA, 1992.

[10] Weak Ordering - A New Definition. S.V. Adve, M.D. Hill. Proc. 17th Annual International Symposium on Computer Architecture. 1990.

[11] Etudes des Cohérences Mémoire Uniformes - Cohérence Causale : mise en oeuvre sur CHORUS et extensions. T. Cornilleau. Thèse de Doctorat. Cnam. Paris. Janvier 1997.

[12] How to find his way in the jungle of Consistency Criteria for Distributed Objects Memories (or how to escape from Minos' Labyrinth). M. Raynal, M. Mizuno. IRISA. Rapport N° 730. May 1993.

[13] Implementing and Programming Causal Distributed Shared Memory. M. Ahamad, P.W. Hutto, R. John. Proc. of the 11th ICDCS. May 1991.

[14] Mémoire Répartie Virtuellement Partagée. M. Shapiro. Support de cours groupe de recherche SRA du DEA Sytème P6. 1995-96.

[15] Memory Consistency and Event Ordering in scalble shared-Memory Multiprocessors. K. Gharachorloo, D. Lenoski, J. Laundon, P. Gibbons, A. Gupta, J. Hennessy. Proc. 17th Annual International Symposium on Computer Architecture. 1990.

[16] Implementing and Programming Weakly Consistent Memories. J. Ranjit. PhD. Thesis. GIT-CC-95-12. Georgia Institute of Technology. Mars 1995.

[17] TreadMarks: Shared Memory on Network of Workstations C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Liu, W Yu, W. Zwanepoel. Research Report Rice University Texas USA. 1996.

Annexe – Eléments de Gestion de Caches processeur

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 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

Objectifs de la conception d'un cache

- maximiser la probabilité de trouver une référence (***hit ratio***[30]) 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

[30] "hit", succès lors d'une requête au cache, au contraire de "miss", qui est employé lors d'un défaut d'accès au cache

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

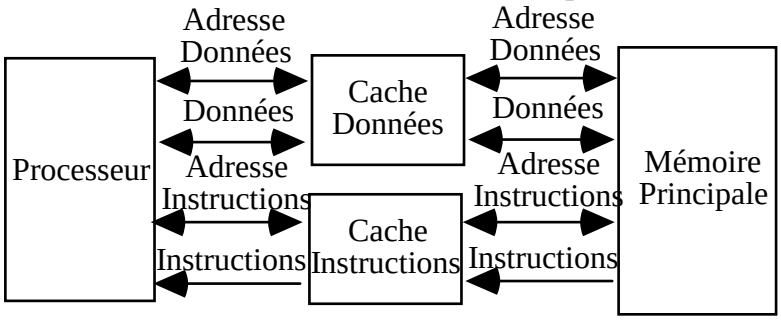
<bit de validité, bit de modification, [clé], adresse de l'information><information>

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 schéma 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 (plutôt write-back ?), ou le contrôleur d'E/S échange uniquement avec la Mémoire Principale (plutôt 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 :

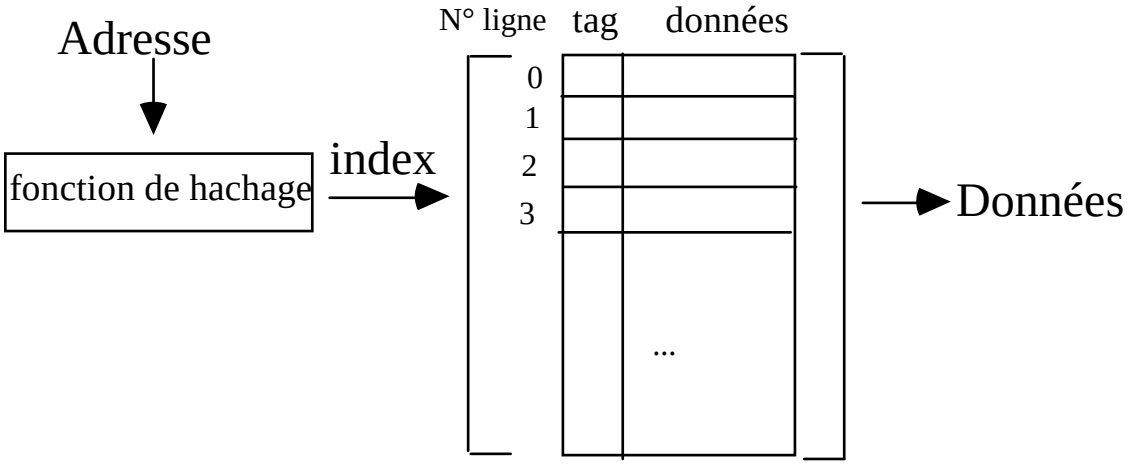
- démarrage à froid par vidage de tous les caches
- démarrage à chaud : on laisse agir l'algorithme de remplacement

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.

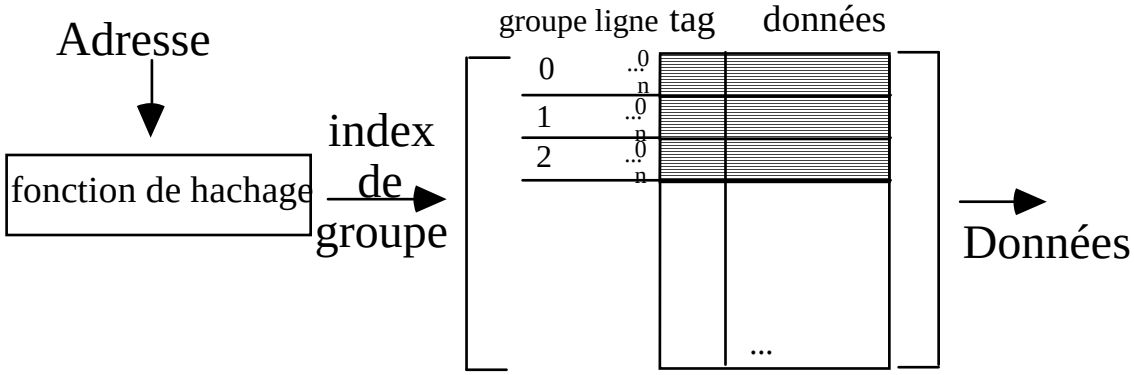
Correspondance Cache / Mémoire Centrale

1. Caches à correspondance directe :



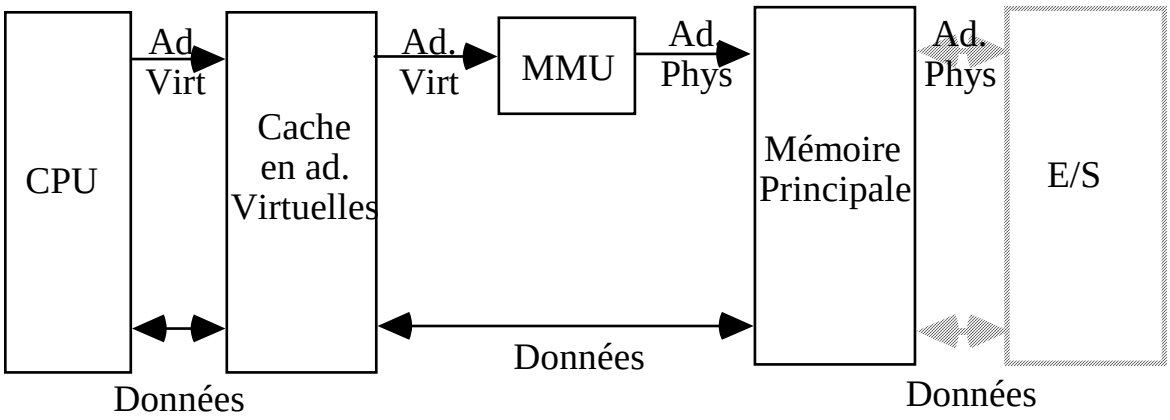
L'adresse de la donnée est prise puis soumise à une fonction de hachage qui retourne un index. Cet index indique la position de la donnée dans le cache si elle s'y trouve. Plusieurs adresses peuvent produire le même index, l'adresse demandée est donc comparée avec l'étiquette contenue dans la ligne pointée 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 s'il y a un défaut dans le cache.

Autres problèmes :

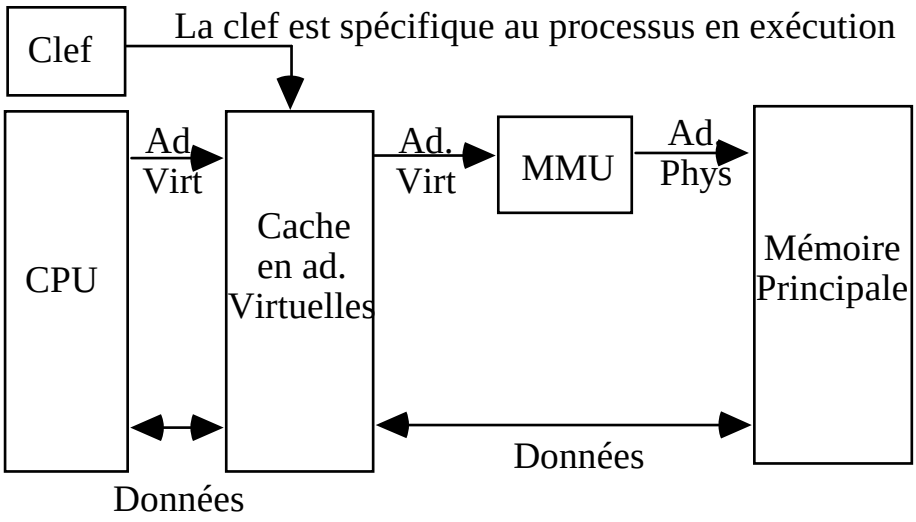
ambiguité - le cache contient une donnée référencée par une adresse virtuelle correcte, mais la correspondance adresse virtuelle/adresse physique a changé depuis le moment où la donnée a été chargée dans le cache, il est alors difficile de conserver la cohérence mémoire centrale/cache (cette situation peut se produire avec une même adresse virtuelle de deux processus différents).

alias - plusieurs adresses virtuelles référence 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

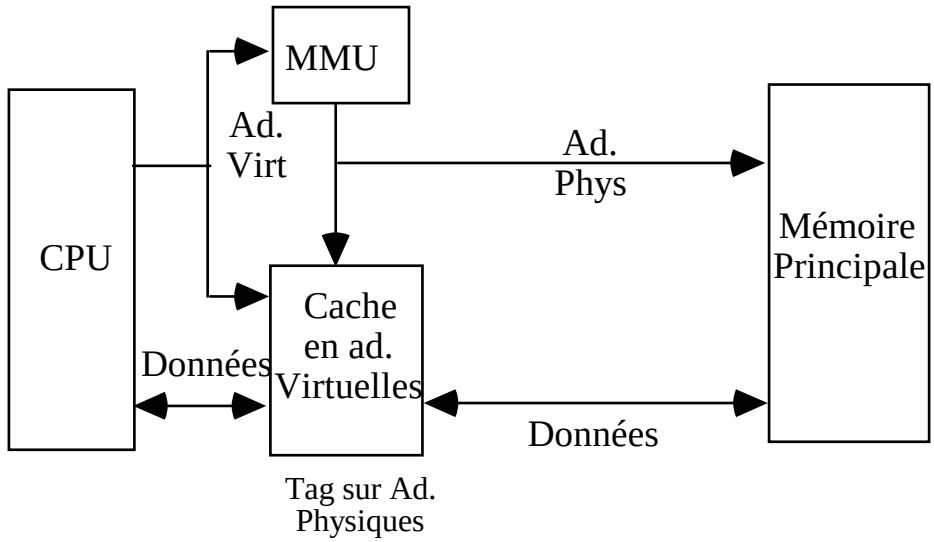


Une clef qui identifie le processus de façon unique est ajoutée à chaque ligne du cache dans l'étiquette.

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



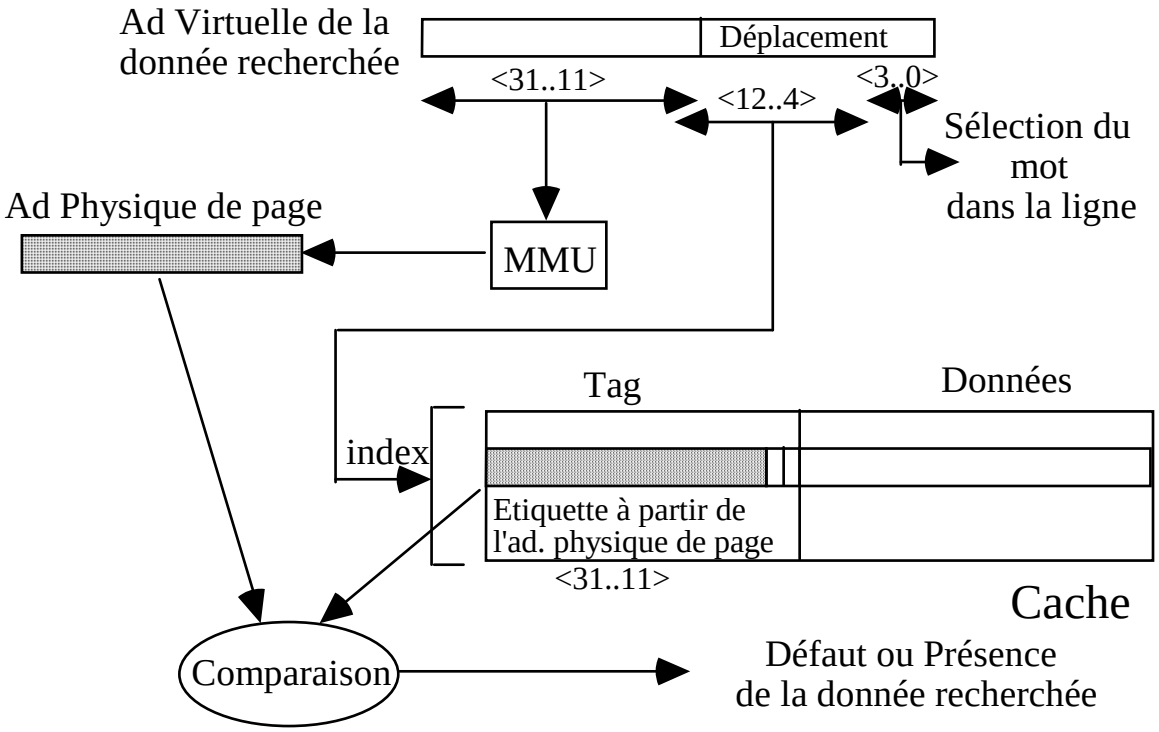
Même principe de fonctionnement qu'avec un cache virtuel. L'étiquette par contre est fabriquée à partir de l'adresse physique et non de l'adresse virtuelle.

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 obtenue 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

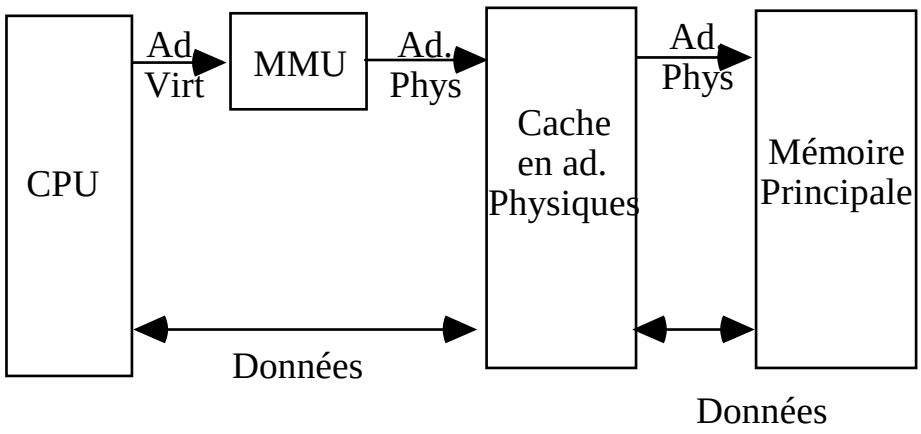
Avantages :
- Il n'y a plus d'ambiguïté sur les données accédées dans le cache, l'adresse physique est là pour les différencier.
- La MMU peut aussi vérifier les droit d'accès à la page pendant le calcul d'adresse.
- Plus besoin de calculer l'adresse physique lors de l'éjection d'une ligne du cache.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

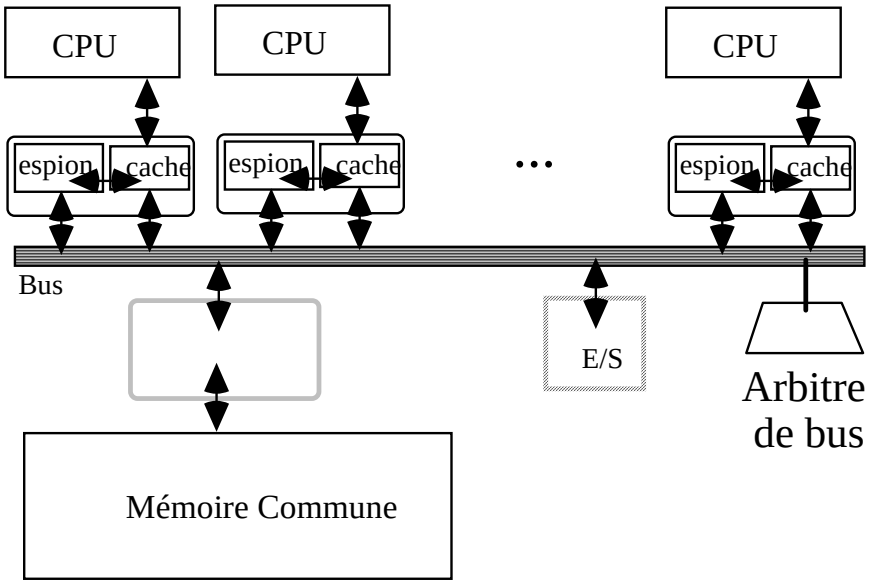


L'adresse physique est calculée à chaque accès.

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.

Protocoles Multi-processeurs par espionnage

Exemple :Firefly

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 :

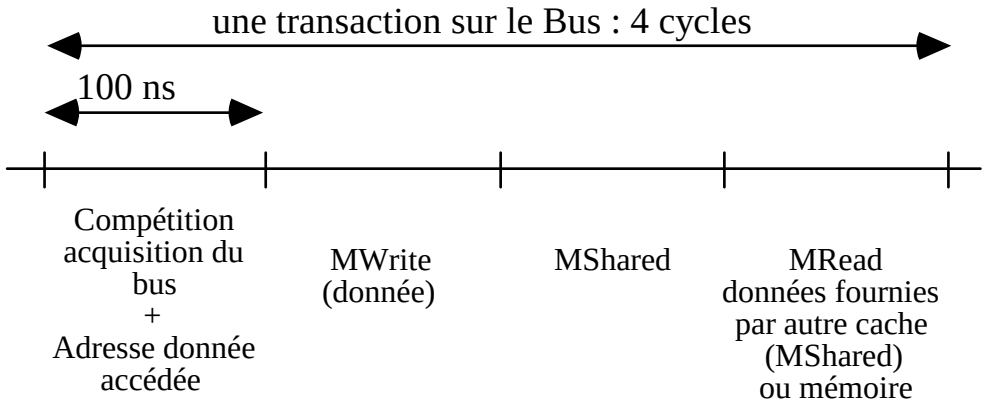
1 bit SHARED : partagé

1 bit DIRTY[31] : 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.



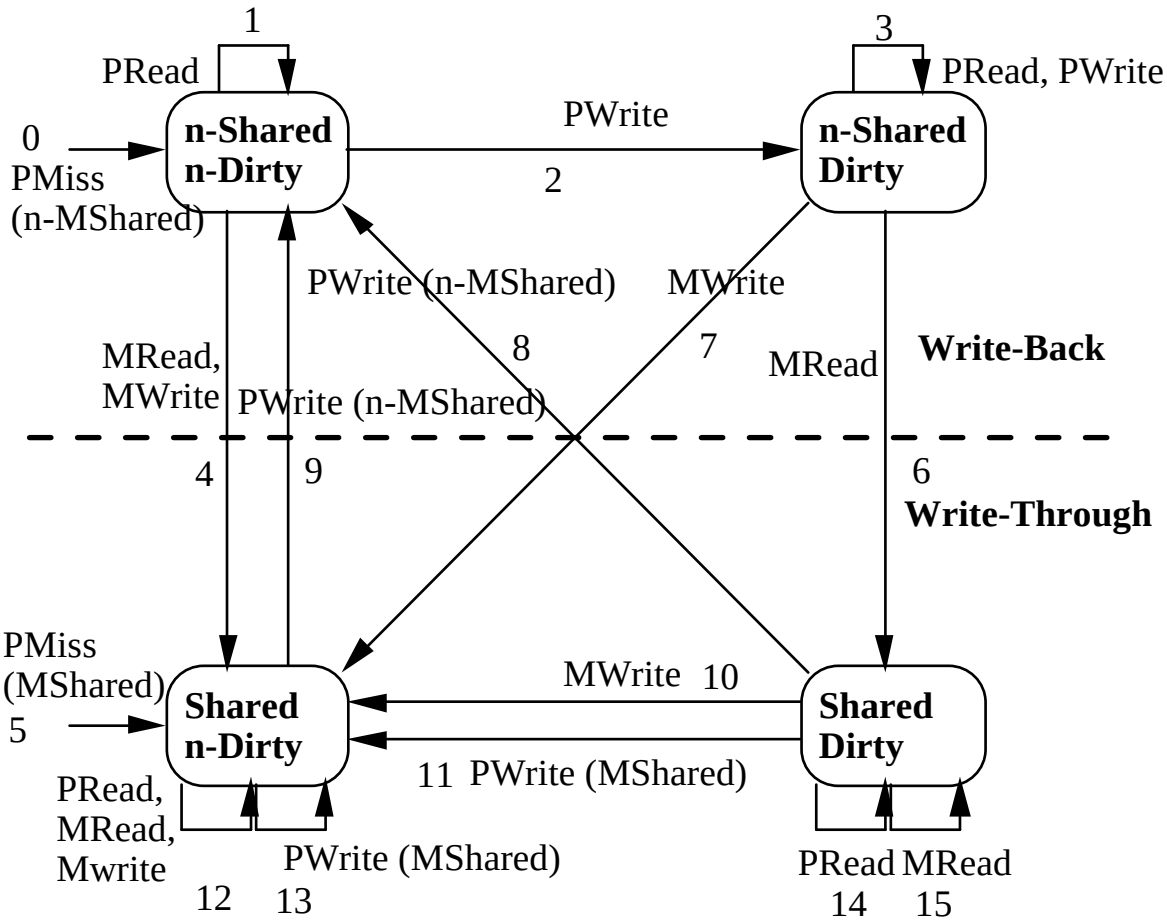
[31] seul type de bloc qui peut-être recopié en mémoire commune, lorsque le cache du dernier écrivain sera vidé

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[32].

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.

[32] 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 cohérence.

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.

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 aggravés 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.