Notes Multiprocessors

Cache Coherence

On a plusieurs processeurs qui ont chacun leur cache.

Coherent Memory System

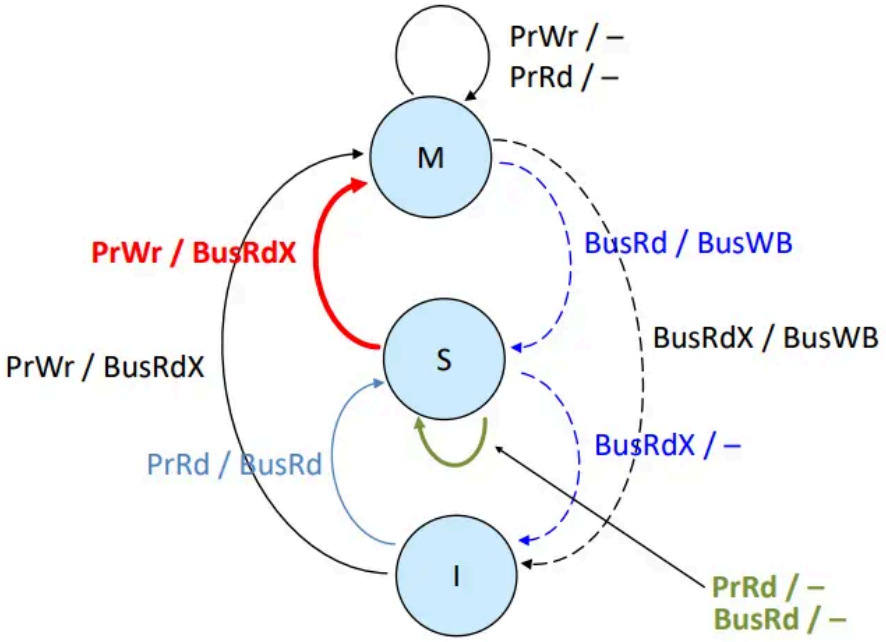
- on doit **préserver l'ordre** du programme (si A écrit X dans M puis lit dans M, A doit lire X).
- **vue cohérente** (maintenant on a aussi un B qui arrive puis qui lit dans M, après un moment plus ou moins loin il doit aussi lire X -- notons que la cohérence n'impose aucune contrainte de temps)
- **sérialisation des écritures** (si A et B écrivent successivement dans M, les autres processeurs doivent voir ces écritures dans l'ordre)

Snoopy Cache

Chaque processeur a son propre cache. Le protocole de snooping permet à chaque cache de surveiller le bus pour voir si une adresse est modifiée par un autre processeur.

- **write-through** : on écrit à chaque fois dans le cache et dans la mémoire.
- **write-back** : on écrit dans le cache et on met un bit dirty, puis on écrit dans la mémoire quand on doit écrire autre chose à la place (demande des states M, S, I).
- **write-allocate** : on remplit le cache après un write-miss (on passe en valid)
- **write-no-allocate** : quand on fait un write, si on est en invalid, on reste en state invalid (on ne cache pas la valeur)

Protocole MSI



Quand on veut lire avec intention de modifier (**RdX**), tous les autres doivent passer en invalide.
Lire avec intention de modifier ça veut dire :

- lire les autres adresses du bloc si elles ne sont pas encore dans notre cache
- et surtout, informer les autres qui lisent le bus qu'on va avoir notre cache modifié par rapport à la mémoire

Quand il y a un **BusRd** ou **BusRdX**, le cache qui est en **M** (modified) doit faire un **WriteBack**.

MESI ajoute un état **E** (exclusive) qui est un peu comme **S** mais on sait qu'on est le seul à avoir cette valeur (sans l'avoir modifiée).

Directory-based Cache Coherence

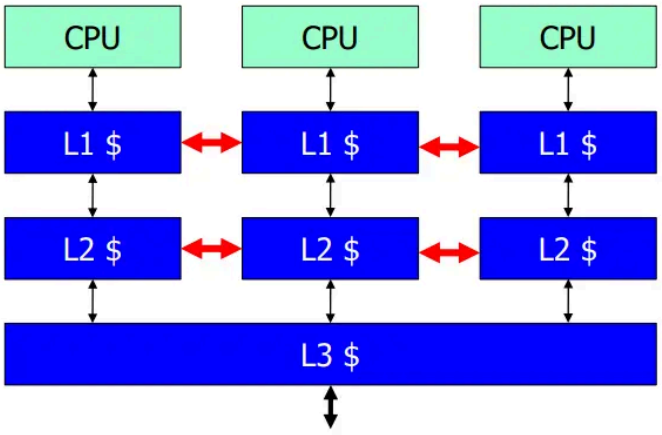
Snoopy est possible avec peu de processeurs (**toutes** les transactions doivent être broadcast sur le bus -- ça fait un goulot d'étranglement tellement il y a d'info qui doivent circuler sur ce bus).

Chaque processeur P_i va avoir son propre **cache**, sa propre **mémoire**, et un dossier, qui va stocker la liste des processeurs qui ont en cache une valeur stockée dans la mémoire de P_i.

Par exemple si P_1 stocke la valeur à l'adresse A, et P_2 demande à lire A, alors P_1 va stocker dans son dossier "Shared : A_P_2". Quand P_1 veut écrire dans A, il va (par exemple, on peut l'implémenter différemment) notifier tous les processeurs qu'il faut invalider leur cache, puis passer en state Modified.

Multi-level caches

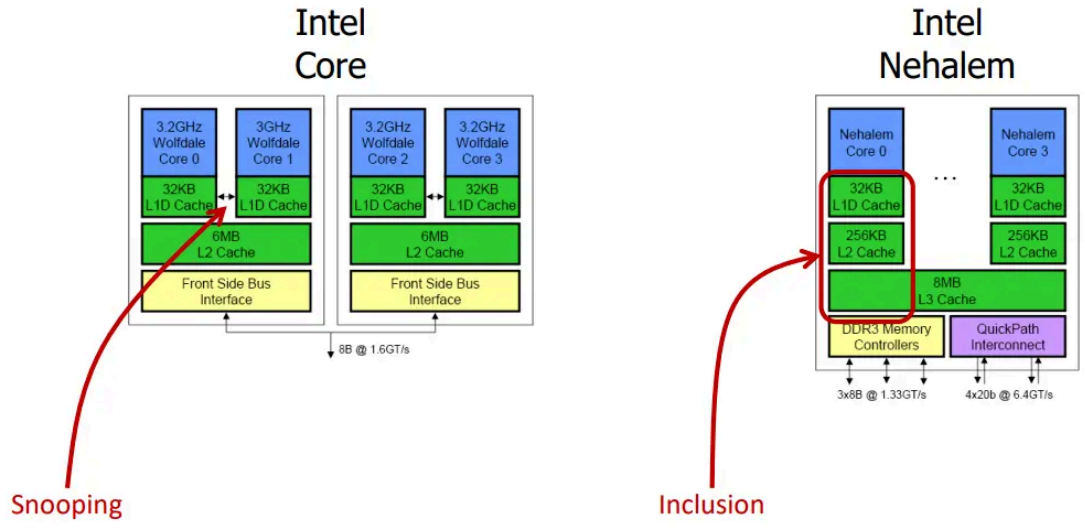
What happens if instead of **CPU → $ → Mem**
we have **CPU → L1 $ → L2 $ → L3 $ → Mem**?
One could just replicate snooping at all levels



Inclusion property

- la couche N-1 contient toujours un sous-ensemble de ce qui a déjà été cached dans une couche plus basse (N2 contient tout N1 + d'autres choses).
- même état, l'état N doit toujours avoir le même état que le N-1, etc.

→ inclusion is **not** naturally maintained! we need to propagate invalidations and evictions up to the hierarchy!



Coherence and Consistency

La **consistency** concerne l'ordre de toutes les opérations de mémoire effectuées par différents processeurs (vers différents emplacements de mémoire). Ordre global pour tous les emplacements de mémoire.

La **cohérence** concerne l'ordre des opérations de différents processeurs (vers le même emplacement de mémoire). Il s'agit d'un ordre local pour un seul emplacement de mémoire.

Le prof nous donne cet exemple :

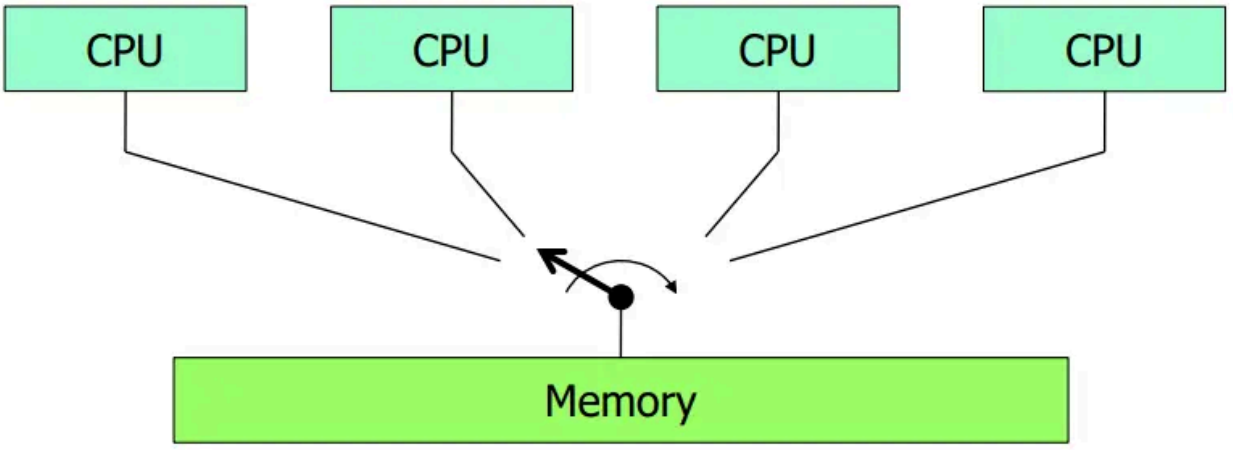
Processor or Thread #1	Processor or Thread #2
`A = 0;`	`B = 0;`
`...`	`...`
`A = 1;`	`B = 1;`
`if ( B == 0 ) ...`	`if ( A == 0 ) ...`

et on se demande si ça peut être à ce point le bazar que les deux reads renvoient 0 ? par exemple si les deux reads numéro 2 sont très lents ?

Nous n'avons pas de **strict** consistency (toute opération de lecture retourne toujours la valeur la plus récente écrite sur une adresse mémoire, peu importe le CPU qui a écrit dedans et celui qui lit la valeur).

Sequential consistency

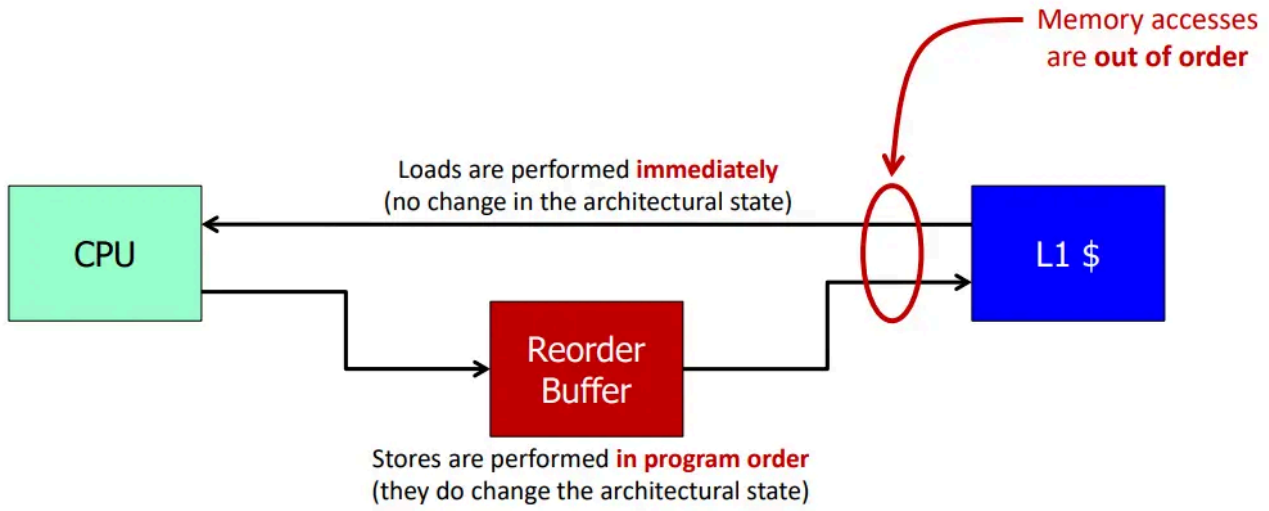
mais nous pouvons avoir de la **sequential** sequency! dans chaque CPU, les opérations s'exécutent au moins dans l'ordre prévu dans le programme (le write de 1 dans A sera fait après le write de 0 dans A). Par contre, les opérations des différents processeurs n'ont pas d'ordre prédéfinis.



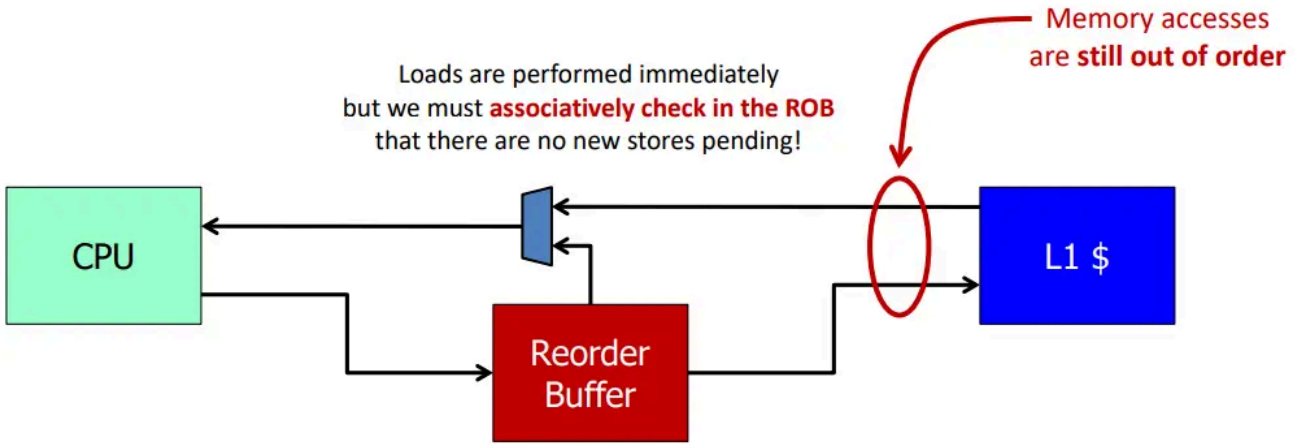
- après chaque accès mémoire, l'aiguille est changée aléatoirement vers un nouveau CPU (comme dit, pas d'ordre défini).
- donc on charge et on stocke dans l'ordre au sein du CPU

Mettre à jour le ROB

Si on veut pouvoir charger depuis les autres CPUs, on doit toujours charger depuis le cache... mais avec le ROB on a plus les bonnes valeurs dans l'ordre ! (certaines valeurs sont pas encore commit mais devraient être accessibles par l'instruction suivant).



donc on ajoute un MUX pour continuer (comme avant) à récupérer les valeurs du ROB... oui mais du coup on a toujours pas les dernières valeurs compute par les autres processeurs ?!



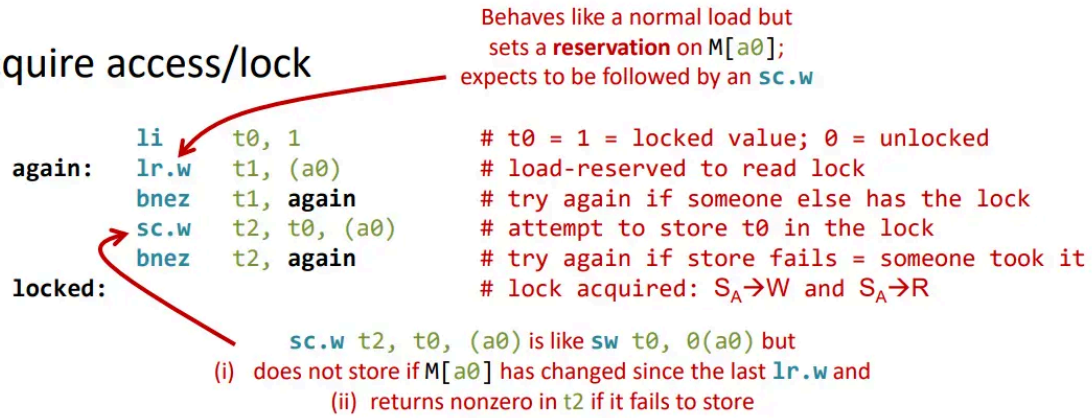
Revenir à la gestion de la sequential consistency

Il y a quand même un problème → comment on sait que le store est fini (que tous les autres CPU ont mis à jour leur cache ?)

- on peut implémenter un système d'acknowledgement pour que la prochaine instruction se fasse uniquement lorsque le cache a été mis à jour, mais **mauvaises performances**
- on peut implémenter un system de locks! en fait quand un programme veut lire une adresse mémoire (pendant un while par exemple) sans que les autres ne la modifie, il

peut utiliser un lock :

- Acquire access/lock



- Release access/lock

```
sw      zero, 0(a0)        # free lock by writing 0 = unlocked
```