Archi 3 : Architecture, Programmation et Compilation des processeurs embarqués Haute Performance

Bertrand Granado - Quentin Meunier - Lionel Lacassagne

Consistance Mémoire

Quentin Meunier

Laboratoire d'Informatique de Paris 6
Équipe Alsoc
4 Place Jussieu, 75252 Paris, France

Octobre 2018

Système multiprocesseur à mémoire partagée

- Modèle plus intuitif que le passage de messages
- Transition plus facile depuis un système monoprocesseur
- Problème sémantique : quel est le comportement en cas de lectures / écritures concurrentes ?
- Exemple :

Initialement, le pointeur `head` et les pointeurs `my_task` valent NULL

P1

```
while (have_task(task_list)) {
  task = get_task(task_list);
  task->data = ...;
  add(task_queue, task);
}
head = task_queue->head;
```

P2, P3, ..., Pn

```
while (my_task == NULL) {
  <begin critical section>
  if (head != NULL) {
    my_task = head;
    head = head->next;
  }
  <end critical section>
}
my_data = my_task->data;
```

- **Quelles sont les valeurs possibles pour `my_data` ?**

Modèle de consistance

- Intuitivement, une lecture devrait retourner la "dernière" valeur écrite
- Difficile à définir de façon précise
- 3 ordres différents pour les opérations mémoire :
  - Ordre du programme : ordre des instructions écrites par le programmeur ou produites par le compilateur
  - Ordre d'exécution : ordre des références mémoire – différent du précédent, par exemple à cause des exécutions *out-of-order*
  - Ordre perçu : ordre, propre à un processeur, de la perception des opérations mémoire des autres processeurs

Définition

- Un **modèle de consistance** est une spécification des comportements de la mémoire autorisés, tels que vus par les processus ($\Leftrightarrow$ processeurs) : ordre perçu
- Il s'agit d'une spécification sur la vue du programmeur $\Rightarrow$ Pas d'hypothèses sur le matériel

Impact du modèle de consistance

- Sur le logiciel de haut-niveau (le programmeur)
  - Validité de la synchronisation entre les threads
- Sur le logiciel de bas-niveau (le programmeur ou le compilateur)
  - Validité du code de synchronisation bas-niveau (ex : bibliothèque de locks)
- Sur le matériel
  - Réordonnancement possible ou non des instructions dans le processeur, des transactions sur le bus, des accès mémoire dans les caches ou la mémoire
  - Spécification du protocole de cohérence de cache
- Sur la performance : supporter un modèle de consistance fort peut être plus complexe en matériel (ou alors interdire certaines optimisations $\Rightarrow$ dans ce cas plus simple), coute plus cher en temps, et interdit des optimisations du compilateur
- Sur la portabilité : le même code n'est pas valide sur tous les processeurs

Le modèle de consistance séquentiel

Définition générale

- Un système supporte la **consistance séquentielle** si toutes les opérations mémoire apparaissent s'effectuer atomiquement, et les opérations émises par un seul processeur apparaissent s'effectuer dans l'ordre du programme

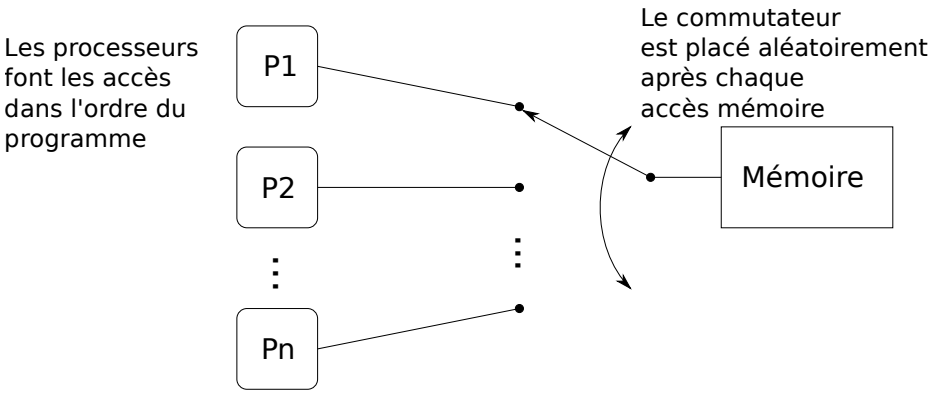
- Pour un programme séquentiel, il suffit de maintenir les dépendances de données pour assurer la consistance séquentielle
- Le compilateur et le matériel peuvent donc exécuter une liste d'instructions dans un ordre différent de l'ordre du programme
- Exemple : un cache *write-through* peut envoyer un miss sans attendre la fin de l'écriture précédente (et les deux peuvent se doubler) ; il suffit de bloquer le miss si une écriture est en cours sur la même ligne.

Le modèle de consistance séquentiel

Définition pour un système multiprocesseur

- Le résultat de n'importe quelle exécution est le même que si toutes les opérations de tous les processeurs étaient effectuées dans un ordre séquentiel (quelconque), et les opérations de chaque processeur individuellement apparaissent dans cette séquence dans l'ordre spécifié par son programme.

- Exemple d'un système simple qui garantirait la consistance séquentielle : système sans cache et une seule mémoire avec un "commutateur"



- Ordre total obtenu en entrelaçant les accès des différents processeurs
- Les opérations mémoire de tous les processus aparaissent comme si elles étaient atomiques les unes par rapport aux autres
- Conserve l'intuition du programmeur

Le modèle de consistance séquentiel

Remarque 1

- L'ordre des accès mémoire n'est pas nécessairement le même entre 2 exécutions

Remarque 2

- Le modèle de consistance mémoire séquentiel ne protège pas contre les **race conditions**

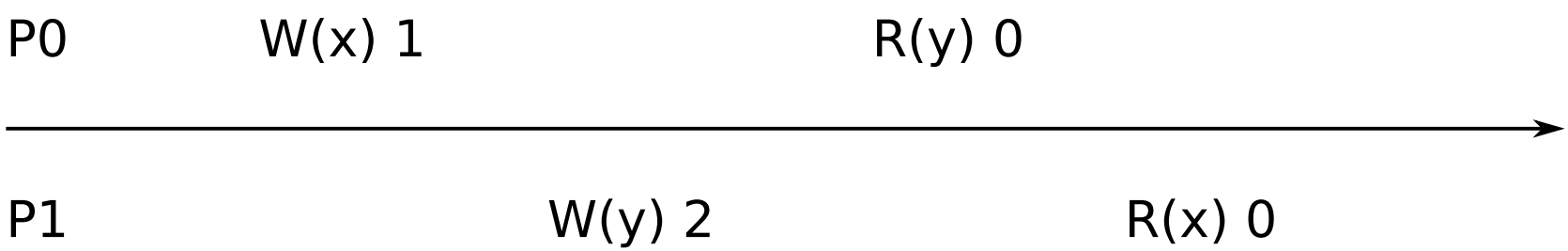
Définition

- Une **race condition** est une erreur dans l'écriture d'un programme, suite à laquelle le résultat/bon fonctionnement d'une application dépend de l'entrelacement de plusieurs accès mémoire concurrents

- Exemple de *race condition* : `a++; || a++;`
- `a++` est traduit en un `load`, un `add` et un `store` : 2 opérations mémoire
- N'importe quel entrelacement dans lequel le 2^e load a lieu avant le 1^{er} store donnera un résultat incorrect : les opérations mémoire sont atomiques, mais pas la séquence des 2

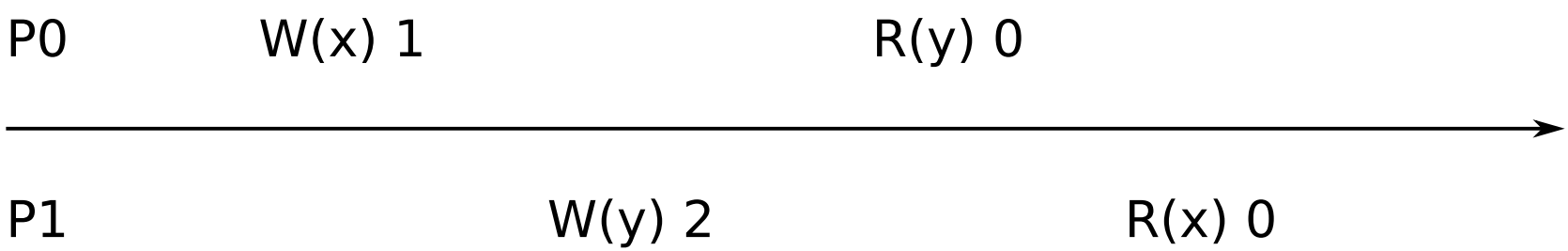
Exemple

- L'exécution suivante est-elle valide pour le modèle de consistence séquentiel ? (initialement $x = y = 0$)



Exemple

- L'exécution suivante est-elle valide pour le modèle de consistence séquentiel ? (initialement $x = y = 0$)



- Pas d'ordre séquentiel équivalent à un entrelacement :
  - R(y) 0 doit être avant W(y) 2
  - Par ailleurs, W(x) 1 doit être avant R(y) 0 et W(y) 2 avant R(x) 0 (ordre du programme) $\Rightarrow$ W(x) 1 doit être avant R(x) 0
  - Or R(x) 0 doit être avant W(x) 1 : contradiction
- Les exécutions R(x) 0 R(y) 2, R(x) 1 R(y) 0 et R(x) 1 R(y) 2 sont valides

Implémentation de la consistance séquentielle

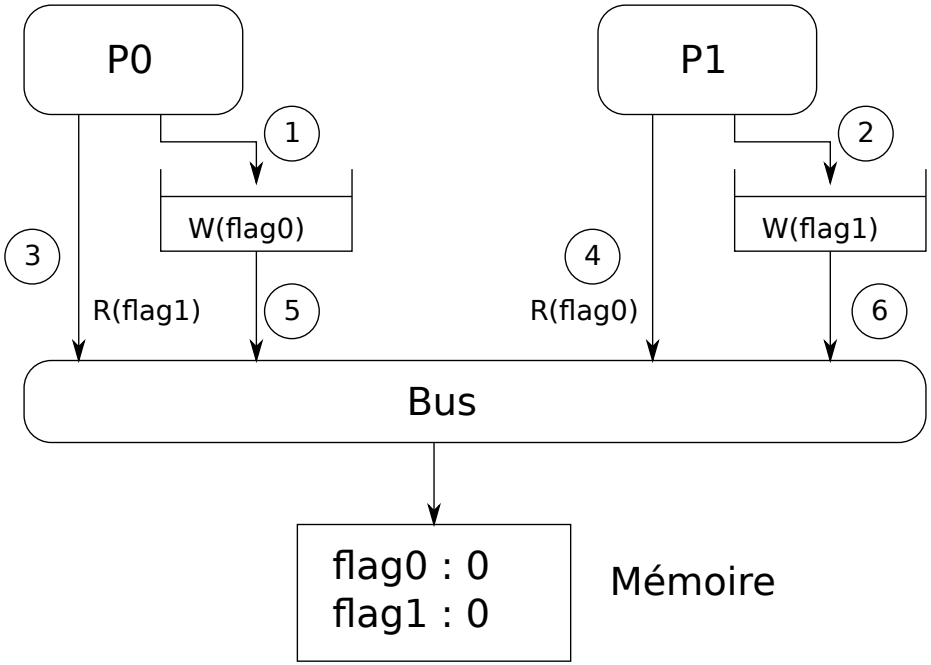
- Implémenter la consistance séquentielle impose beaucoup de contraintes sur le matériel : cela est complexe et couteux en temps car peu d'opérations peuvent se recouvrir
- Dans cette section, description de ces contraintes selon deux hypothèses :
  - L'architecture ne comporte pas de caches (plus simple)
  - L'architecture comporte des caches

Implémentation de la consistance séquentielle dans une architecture sans caches

- De nombreuses optimisations impossibles
- En monoprocesseur, seules les dépendances sur la même variable et autres que RAR sont à considérer
- Pour garantir la consistance séquentielle en multiprocesseur, il faut garantir tous les types de dépendances (RAR, WAW, WAR et RAW), y compris sur des données différentes
- Exemple avec 3 optimisations courantes :
  - Les buffers d'écriture (ne respecte pas les dépendances RAW)
  - Les écritures recouvrantes (ne respecte pas les dépendances WAW)
  - Les lectures non bloquantes (ne respecte pas les dépendances RAR)

Le buffer d'écriture

- Le buffer d'écriture permet de retarder une écriture pour favoriser les lectures, en permettant que les lectures doublent les écritures (à des adresses différentes)



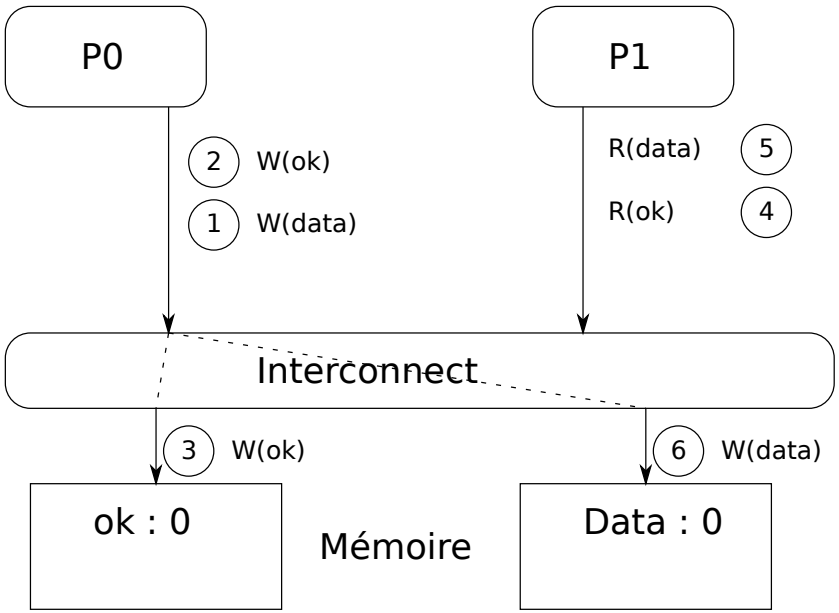

```
P0                          P1

flag0 = 1                   flag1 = 1
if (flag1 == 0) {           if (flag0 == 0) {
  <section                    <section
       critique>                   critique>
}                           }
```

- Avec la consistance séquentielle, impossible que les deux tests réussissent
- Cela est possible avec un buffer d'écriture

Les écritures recouvrantes

- Architecture comportant deux bancs de mémoire ansi qu'un interconnect autre qu'un bus (NoC, crossbar)
- Le processeur exécute ses accès mémoire dans l'ordre du programme, mais n'attend pas la fin d'une écriture pour continuer



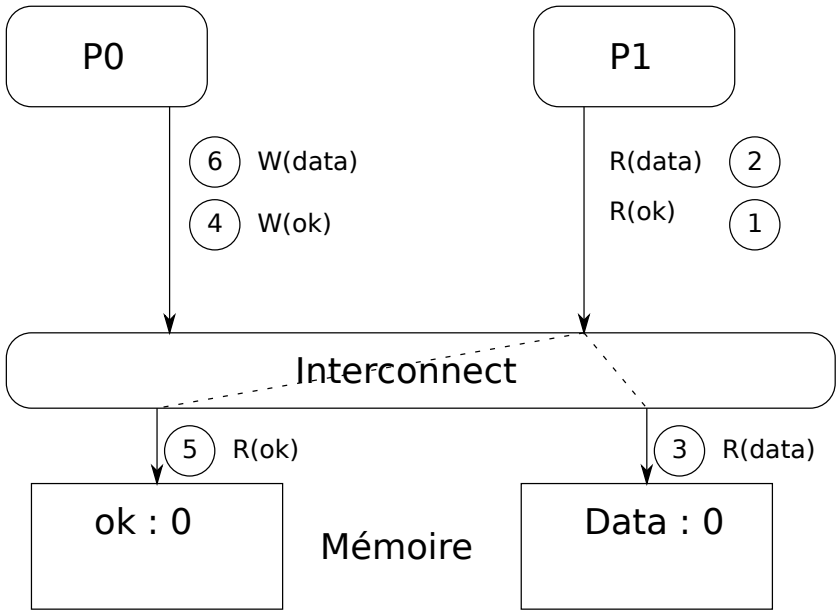

```
P0                          P1

data = 1500                 while (ok == 0);
ok = 1                      local_var = data;
```

- Avec la consistance séquentielle, P1 est sûr de lire la valeur 1500
- Cela peut ne pas être le cas ici

Les lectures non-bloquantes

- Architecture comportant deux bancs de mémoire ansi qu'un interconnect autre qu'un bus (NoC, crossbar)
- Le processeur a la capacité de faire des lectures spéculatives (même exemple possible avec des caches non-bloquant)




```
P0                          P1

data = 1500                 while (ok == 0);
ok = 1                      local_var = data;
```

- Avec la consistance séquentielle, P1 est sûr de lire la valeur 1500
- Cela peut ne pas être le cas ici

Synthèse

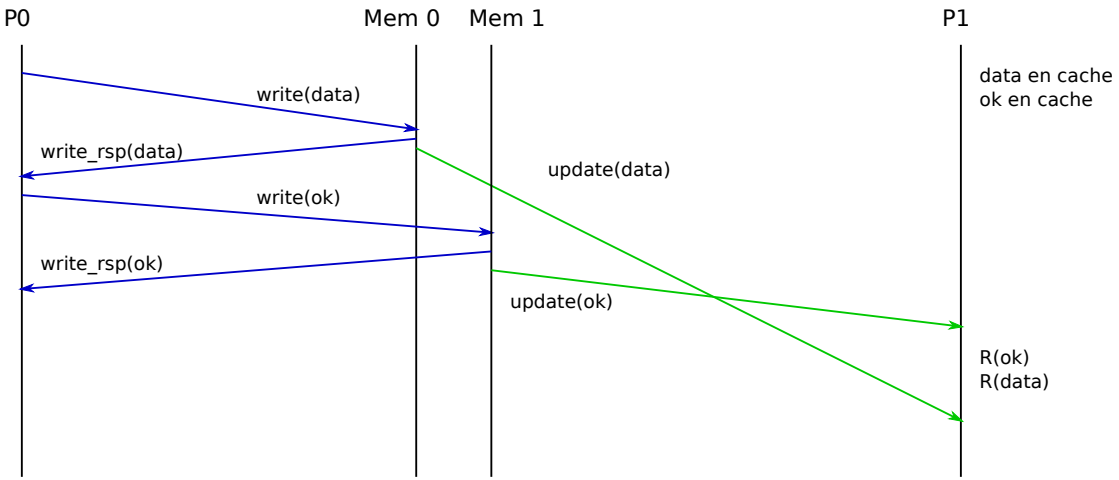
- Ces 3 exemples montrent qu'il faut impérativement forcer les opérations mémoire à avoir lieu dans l'ordre du programme pour garantir la consistance séquentielle
- $\Rightarrow$ Attendre la fin de chaque requête avant de commencer la suivante

Implémentation de la consistance séquentielle dans une architecture avec caches

- Respecter les contraintes précédentes ; ne pas utiliser de buffer d'écriture, attendre la fin d'une requête d'écriture avant de commencer la suivante
- 2 difficultés supplémentaires :
  - Détecter qu'une écriture est finie/complétée
  - Rendre/Faire paraitre les écritures atomiques (*via* les invalidations ou les mises à jour)
  - $\Rightarrow$ Ces deux points concernent plus particulièrement les protocoles de types write-through

Détecter la fin d'une écriture avec des caches : problème

- Hypothèses :
  - Les accès mémoire ont lieu dans l'ordre du programme et le processeur attend la fin d'une écriture avant de faire l'accès suivant
  - La mémoire répond à la requête d'écriture dès que celle-ci est effectuée
- Protocole *write-through write update*, mais situations équivalentes avec des invalidations



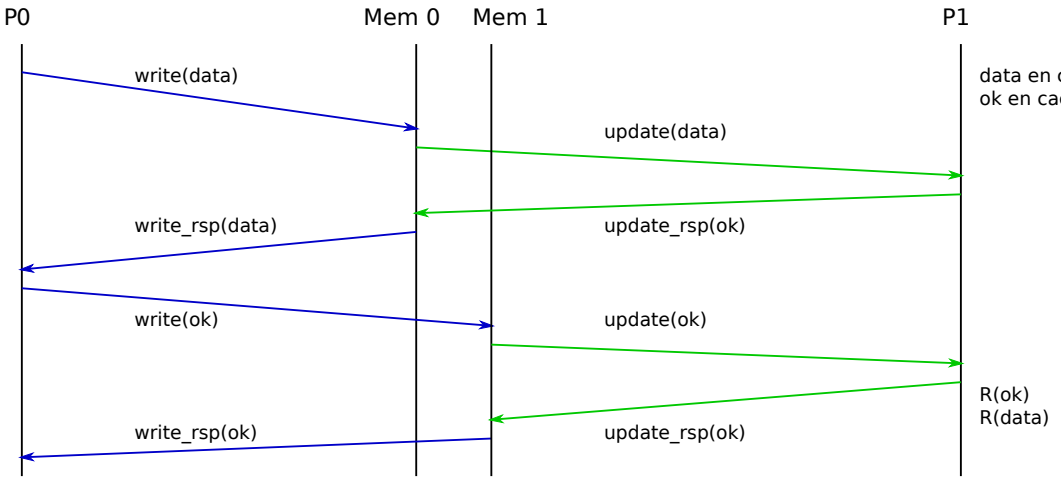

```
P0                  P1

data = 1000         while (ok == 0);
ok = 1              local_var = data;
```

- P1 peut ne pas lire la valeur 1000 ⇒ Il faut que la mémoire ne réponde que lorsque tous les caches sont à jour

Détecter la fin d'une écriture avec des caches : solution

- Hypothèses :
  - Les accès mémoire ont lieu dans l'ordre du programme et le processeur attend la fin d'une écriture avant de faire l'accès suivant
  - La mémoire répond à la requête d'écriture après réception de toutes les réponses aux *updates*




```
P0                  P1

data = 1000         while (ok == 0);
ok = 1              local_var = data;
```

- Inconvénient : ajoute 2 indirections pour chaque écriture qui contient des copies

Maintenir l'atomicité des écritures

- Hypothèses
  - Les accès mémoire ont lieu dans l'ordre du programme et le processeur attend la fin d'une écriture avant de faire l'accès suivant
  - La mémoire répond à la requête d'écriture après réception de toutes les réponses aux *updates*
- Avec la consistance séquentielle, `local_var2` et `local_var3` doivent obtenir la même valeur

P0

```
data = 1000
ok0 = 1
```

P1

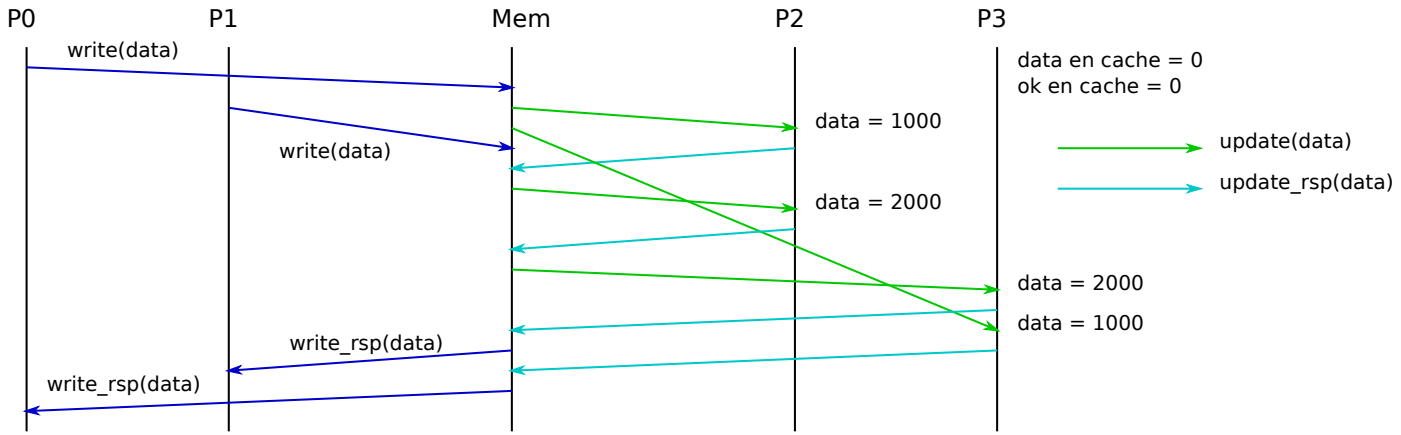
```
data = 2000
ok1 = 1
```

P2

```
while (ok0 == 0);
while (ok1 == 0);
local_var2 = data;
```

P3

```
while (ok0 == 0);
while (ok1 == 0);
local_var3 = data
```



- Le problème vient du fait que l'interconnect n'est pas FIFO point à point
- Autre possibilité pour résoudre le problème : sérialiser les *updates* à une même adresse au niveau de la mémoire (très couteux en latence si beaucoup d'*updates*)

Maintenir l'atomicité des écritures

- Hypothèses
  - Les accès mémoire ont lieu dans l'ordre du programme et le processeur attend la fin d'une écriture avant de faire l'accès suivant
  - La mémoire répond à la requête d'écriture après réception de toutes les réponses aux *updates*
  - Interconnect FIFO point à point, *updates* sérialisés
- Avec la consistance séquentielle, `local_var2` doit valoir 1 ; Montrer que ce n'est pas forcément le cas avec les hypothèses définies

P0

```
a = 1;
```

P1

```
while (a != 1);
b = 1;
```

P2

```
while (b != 1);
local_var2 = a;
```

Maintenir l'atomicité des écritures

- Hypothèses
  - Les accès mémoire ont lieu dans l'ordre du programme et le processeur attend la fin d'une écriture avant de faire l'accès suivant
  - La mémoire répond à la requête d'écriture après réception de toutes les réponses aux *updates*
  - Interconnect FIFO point à point, *updates* sérialisés
- Avec la consistance séquentielle, `local_var2` doit valoir 1 ; Montrer que ce n'est pas forcément le cas avec les hypothèses définies

P0

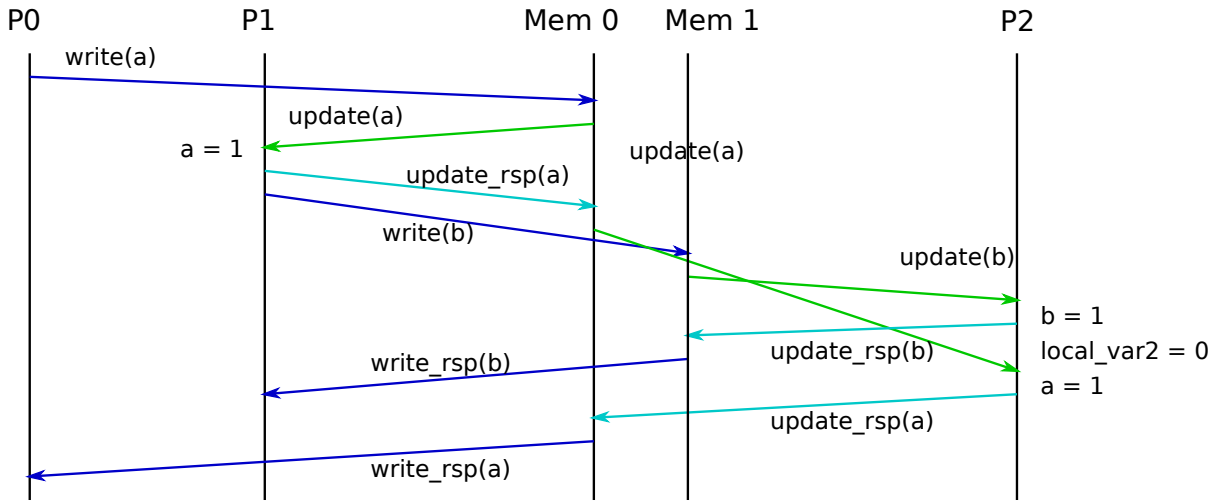
```
a = 1;
```

P1

```
while (a != 1);
b = 1;
```

P2

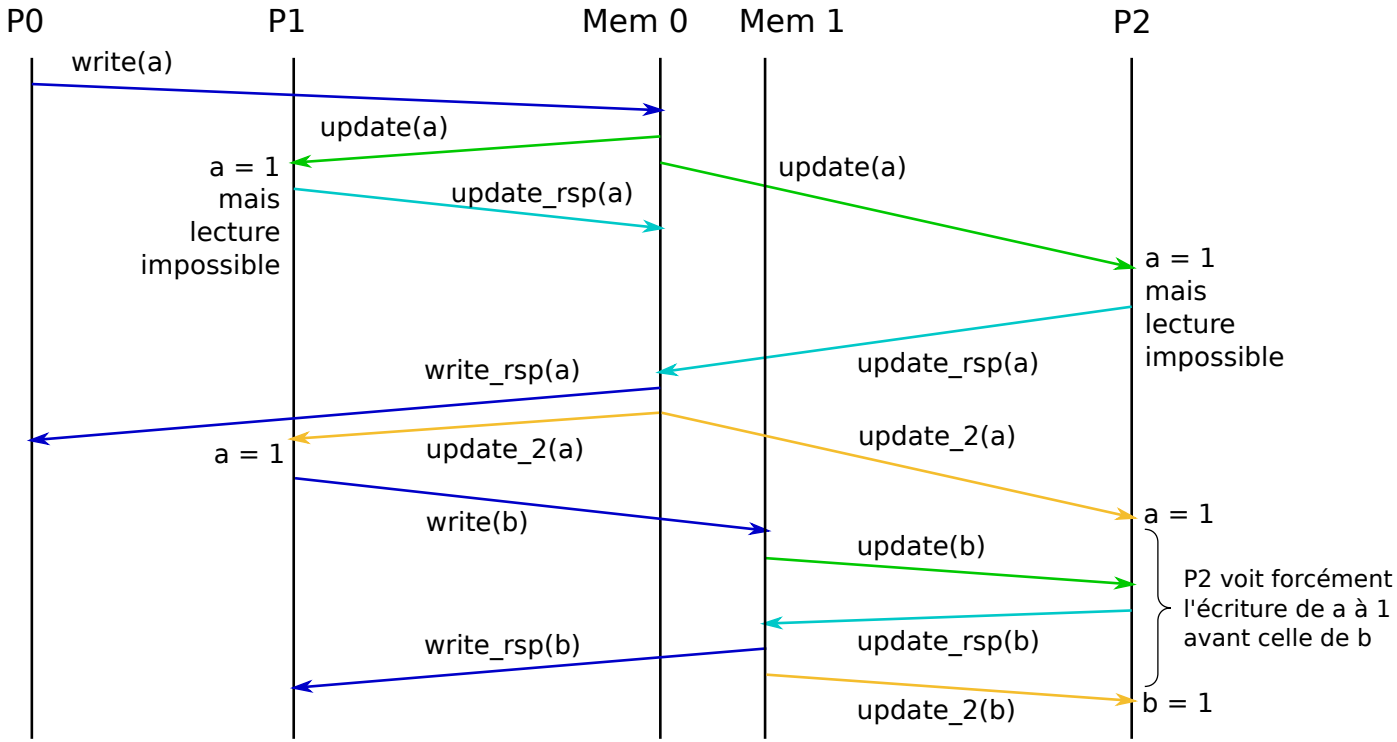
```
while (b != 1);
local_var2 = a;
```



- Problème : il ne faut pas que P1 puisse lire la valeur de l'*update* `[a = 1]` avant que les autres caches (ici P2) l'aient reçu

Maintenir l'atomicité des écritures : solution

- *update* en 2 phases : mise à jour de la donnée, puis autorisation de lecture



- Autre solution : utiliser des invalidations et bloquer les miss tant que tous les caches n'ont pas répondu à l'invalidation

Synthèse sur l'implémentation de la consistance séquentielle

- Maintenir la consistance séquentielle dans une architecture multiprocesseur est difficile
- De plus, cela ajoute beaucoup d'indirections et d'attentes qui empêchent une exécution efficace
- En pratique, les architectures multiprocesseur n'implémentent pas la consistance séquentielle
- $\Rightarrow$ Il existe un certain nombre de modèles de consistance plus faibles que le modèle séquentiel
- $\Rightarrow$ Les synchronisations doivent en conséquence être adaptées avec des primitives spécifiques
- $\Rightarrow$ Plus généralement, le problème est partiellement repoussé vers le logiciel, qui ne peut plus en faire abstraction

Le modèle Local

Définition

- L'ordre perçu pour les opérations mémoire locales à un processeur est l'ordre du programme

- Ne dit rien sur les opérations mémoire des autres processeurs
- Tous les modèles (utiles) sont plus forts que le modèle local et plus faibles que le modèle séquentiel

Le modèle Slow

Définition

- Les lectures doivent retourner une valeur précédemment écrite. Une fois qu'une valeur à été lue, aucune valeur précédemment écrite par le processeur qui a écrit la valeur lue ne peut être retournée. Les écritures d'un processeur doivent être visibles immédiatement à lui-même

Le modèle PRAM

Définition

- Les écritures effectuées par un même processeur sont vues par les autres processeurs dans l'ordre dans lequel elles ont été effectuées, mais les écritures venant de processeurs différents peuvent être vues dans des ordres différents par différents processeurs

Le modèle Cache

Définition

- Toutes les écritures au même emplacement mémoire sont effectuées dans un ordre séquentiel

- Ne dit rien pour des écritures à des emplacements mémoire différents

Le modèle Processeur

Définition

- L'exécution est consistente PRAM et toutes les écritures au même emplacement sont vues dans le même ordre par tous les processeurs

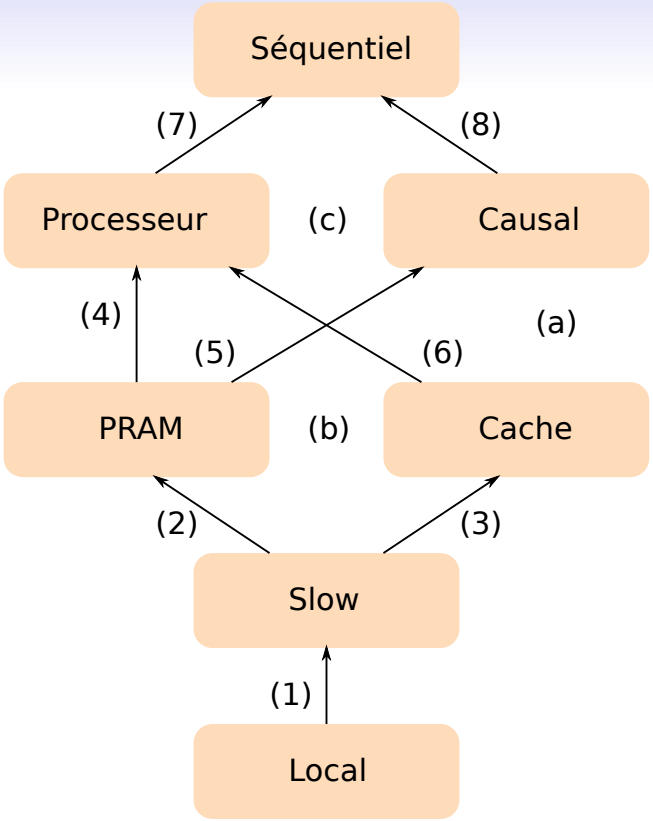
- Combinaison de PRAM et Cache

Le modèle Causal

Définition

- Pour chaque processeur, les opérations de ce processeur et toutes les écritures “connues” de ce processeur (dépendances RAW) apparaissent pour ce processeur dans un ordre qui respecte la causalité.

Graphe de relations entre ces modèles



- Exercice : Pour chaque couple de modèles reliés par une flèche, écrire un exemple d'exécution qui est valide pour le modèle du bas (le plus faible) et invalide pour le modèle du haut (le plus fort)
- De plus, pour les 3 couples (a), (b) et (c), montrer que les modèles du couple sont incomparables

Autre approche : Modèle “Weak”

Idée

- Les accès aux variables globales de synchronisation (locks) sont fortement ordonnés : introduction de la notion de *variable de synchronisation*

Définition du modèle de consistence Weak

- Tous les accès aux variables de synchronisation sont vus dans le même ordre (ordre séquentiel) par tous les processeurs
- Tous les autres accès peuvent être vus dans des ordres différents par différents processeurs
- L'ensemble des variables accédées entre 2 opérations de synchronisation doit être le même pour tous les processeurs (i.e. une opération de synchronisation ne peut pas être réordonnée vis-à-vis d'autres opérations mémoire)

Modèle “Weak”

Conséquences

- Il ne peut pas y avoir d'accès à une variable de synchronisation tant qu'il y a des opérations d'écriture en cours
- Il ne peut pas y avoir de nouvelles opérations de lecture et d'écriture lancées quand une opération de synchronisation a été commencée et n'est pas terminée

En résumé

- Les accès aux variables de synchronisation empêchent le reordonnancement et la consistance séquentielle est assurée pour ces variables
- $\Rightarrow$ C'est ça qui est souvent utilisé en pratique car cela ne fait pas d'hypothèses sur le modèle implémenté par le matériel, et les synchronisations sont à la charge du programmeur
- $\Rightarrow$ Existence de primitives spécifiques : on appelle ces instructions des instructions de “barrière mémoire”
- Exemple : `sync` en mips, `mfence` en x86

Modèle “Weak” : exemple

Producer P0

```
lock(l);                Pas de réordonnancement
read(status);
write(data0);
write(data1);           } Réordonnancement possible
write(data2);
write(status);
unlock(l);              Pas de réordonnancement
```

Consumer P1

```
lock(l);                Pas de réordonnancement
read(status);
read(data0);
read(data1);            } Réordonnancement possible
read(data2);
write(status);
unlock(l);              Pas de réordonnancement
```

Motivation

- Même le modèle séquentiel ne protège pas des *race conditions* : les accès mémoire sont atomiques “un par un”, pas par groupe
- $\Rightarrow$ Nécessité de synchroniser les accès à l'aide de primitives
- Types de synchronisation
  - Exclusion mutuelle
  - Synchronisation d'évènements (ex : barrières)

Composantes de la synchronisation

- Acquisition
  - Acquérir le droit à la synchronisation (entrée d'une section critique, aller au-delà d'un évènement)
- Algorithme d'attente
  - Attendre que la synchronisation devienne disponible si elle ne l'est pas
- Libération
  - Permettre à d'autres processeurs d'acquérir le droit à la synchronisation
- Remarque : l'algorithme d'attente est indépendant de la synchronisation

Les algorithmes d'attente

- Bloquant
  - Les processus en attente sont désordonnancés
  - Surcout élevé
  - Permet au processeur de faire autre chose
- Attente active
  - Les processeurs en attente testent en permanence une case mémoire jusqu'à ce que sa valeur change
  - Le processus libérant le verrou écrit la case
  - Surcout plus faible, mais consomme les ressources du processeur
  - Peut provoquer du traffic réseau
- L'attente active est meilleure quand
  - Le surcout d'ordonnancement est plus grand que le temps d'attente estimé
  - Les ressources processeurs ne sont pas nécessaires pour d'autres tâches
  - Le blocage par l'ordonnanceur n'est pas possible (ex : dans le noyau de l'OS)
- Méthodes hybrides : attente active puis blocage

Le problème de l'exclusion mutuelle

- Empêcher l'accès concurrent à plus d'un thread à une section de code donnée

Pourquoi est-ce un problème ?

- Pour la même raison que celle qui nous empêchait d'incrémenter une variable partagée avec a++;

```
lock_acquire:
    lw $8, 0($4) # $4 contient l'adresse du lock
    bne $8, $0, lock_acquire
    li $8, 1
    sw $8, 0($4)
    jr $31
```

- La lecture (test) et l'écriture (prise du lock) ne sont pas atomiques
- Deux processeurs peuvent prendre le lock en même temps

Le problème de l'exclusion mutuelle

Types d'implémentation des solutions

- Uniquement logicielle
  - Algorithmes complexes (ex : Dekker, Peterson)
  - Contraintes supplémentaires (ex : le nombre de threads doit être connu à l'avance)
  - Dépendant du modèle de consistance
- Avec l'aide du matériel
  - Pas standard : différents types de primitives de synchronisation selon les architectures
  - Exemple de primitives : Test&Set, Compare-and-Swap, LL/SC

Instruction Test & Set

- La valeur de la case mémoire est lue dans un registre (comme pour un `load`)
- La constante 1 est écrite dans la case mémoire
- Peut être utilisé pour réaliser un lock :
  - Prise du lock réussie si la valeur chargée dans le registre est 0
  - Pour relâcher le lock, il suffit d'écrire la valeur 0

```
lock_acquire:
    tas $8, 0($4) # n'existe pas en Mips
    bne $8, $0, lock_acquire
    jr $31
```

Instruction Compare-and-Swap

- Sémantique

```
bool cas(int * addr, int old_val, int new_val) {
    <atomic>
    if (*addr == old_val) {
        *addr = new_val;
        return true;
    }
    else {
        return false;
    }
    <end atomic>
}
```

Exercice

- Donner le code des fonctions `lock_acquire(int * lock)` et `lock_release(int * lock)` en C en utilisant la primitive cas

Instructions LL/SC

- LL(x) (*Load Link*) : Lecture à l'adresse x avec un effet de bord
- SC(x, v) (*Store Conditional*) : Écriture de la valeur v à l'adresse x s'il n'y a pas eu d'autre écriture (éventuellement, à l'adresse x) ou de SC depuis le LL(x) fait par ce processeur
  - Renvoie succès (1) ou échec (0) pour en informer le processeur
  - En Mips, `sc $x, 0($y)`, avec résultat dans `$x`

Exercice

- Donner le code des fonctions `lock_acquire` et `lock_release` en assembleur mips en utilisant les instructions LL/SC

Instruction de barrière mémoire

- Synchronise tous les accès mémoire en cours pour le processeur qui l'exécute
- $\Rightarrow$ Bloque l'exécution jusqu'à ce que toutes les écritures en cours soient terminées et visibles des autres processeurs

Exercice

- Montrer avec un exemple que la primitive `lock_release()` écrite précédemment peut nécessiter l'utilisation d'une barrière mémoire dans certains modèles de consistance mémoire
- Précisez ces modèles

Exemple : Algorithme de Peterson

But

- Implémenter une section critique entre 2 threads sans primitive spécifique (intialement, `flag0 = flag1 = false`)
  - Pas utilisé en pratique : complexe et peu d'avantages en comparaison de l'utilisation de primitives matérielles

```
// T0
flag0 = true;
turn = 1;
while (flag1 && turn == 1);
// Début de la section critique
...
// Fin de la section critique
sync(); // ajouté pour l'exercice
flag0 = false;
```

```
// T1
flag1 = true;
turn = 0;
while (flag0 && turn == 0);
// Début de la section critique
...
// fin de la section critique
sync(); // ajouté pour l'exercice
flag1 = false;
```

Exercice

- Pour quels modèles de consistence mémoire cet algorithme marche-t-il ?
- Que faut-il changer pour le faire marche ?

Influence de l'architecture sur la synchronisation : locks à ticket

Problème courant des spinlocks

- Souvent, les implémentations de spinlock ne respectent pas une prise de lock correspondant à l'ordre d'appel à la fonction `spin_lock`

Idée des locks à ticket

- Spin lock qui conserve l'ordre FIFO entre la demande et l'obtention du lock
- Besoin de 2 variables

```
typedef struct {
  int now_serving;
  int next_ticket;
} tlock;
...
ticket_init(tlock * lock) {
  lock->now_serving = 0;
  lock->next_ticket = 0;
}
```

```
void ticket_lock(tlock * lock) {
    // repose sur la primitive
    // atomique fetch_and_inc
    int my_ticket = fetch_and_inc(&lock->
        next_ticket);
    while (my_ticket != lock->now_serving);
}

void ticket_unlock(tlock * lock) {
    lock->now_serving += 1;
}
```

Influence de l'architecture sur la synchronisation : locks MCS

Problème avec les ticket locks

- Lors d'un relâchement de lock, une invalidation (ou une mise à jour) est envoyée à tous les caches qui sont en attente du lock
- $\Rightarrow$ Sérialisation de tous les miss au niveau du cache L2 : ne passe pas bien à l'échelle quand on augmente le nombre de coeurs

Idée des locks MCS

- Chaque thread boucle sur une variable privée et non partagée

```
typedef struct _node {
  struct _node * next;
  bool is_locked;
} mcs_node;

typedef struct {
  mcs_node * queue;
} mcs_lock;
```

```
void lock(mcs_lock * lock, mcs_node * mynode) {
  mynode->next = NULL;
  // opération fetch_and_store atomique
  mcs_node * pred = fetch_and_store(&lock->queue
       , mynode);
  if (pred != NULL) {
    mynode->is_locked = true;
    pred->next = mynode;
    while (mynode->is_locked);
  }
}
```

Locks MCS (suite)

```
void unlock(mcs_lock * lock, mcs_node
      * mynode) {
  if (mynode->next == NULL) {
    // compare_and_swap
    if (cas(&lock->queue, mynode,
        NULL)) {
      // lock remis à NULL
      return;
    }
    else {
      // un thread s'est enregistré
      // mais n'avait pas encore
      // affecté le champ next au
      // moment du test
      while (mynode->next == NULL);
    }
  }
  mynode->next->is_locked = false;
}
```

Remarques

- Nécessite un mécanisme d'allocation de noeuds (il peut s'agir de variables en pile – un des rares cas où la sémantique de partage de pile est utile)
- Les primitives `fetch_and_store` et `compare_and_swap` peuvent être réalisées à partir des instructions LL/SC
- Il manque deux instructions de barrière mémoire dans le code pour supporter des modèles de consistance mémoire faibles