Multiprocesseurs symétriques

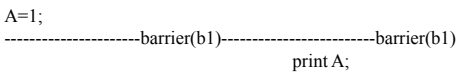
Daniel Etiemble
LRI, Université Paris Sud
de@lri.fr

Consistance mémoire

- L'écriture dans une case devient visible à tous les autres processus dans le même ordre. Mais quand une écriture devient-elle visible ?
  - Comment établir un ordre entre une écriture et une lecture par différents processeurs ?

P_1	P_2
/*Les valeurs initiales de A et flag sont 0*/	
`A = 1;`	`while (flag == 0);  /*attente*/`
`flag = 1;`	`print A;`

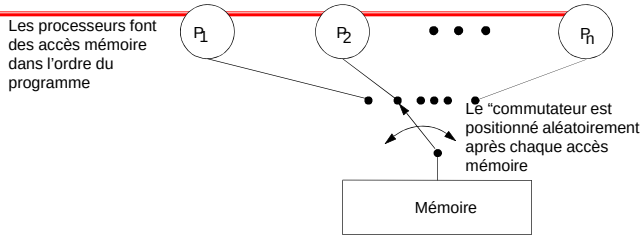
  - Utilisation de barrière de synchronisation



  - Autre exemple

P_1	P_2
/*Valeurs initiales de A et N : 0*/	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

La consistance séquentielle



- (Comme s'il n'y avait pas de caches et une seule mémoire)
- Ordre total obtenu en entrelaçant les accès des différents processus
- Maintient l'ordre du programme et les opérations mémoire de tous les processus apparaissent (début, exécution, fin) comme si elles étaient atomiques les unes par rapport aux autres
- Conserve l'intuition du programmeur
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Exemple de consistance séquentielle

L'important est l'ordre dans lequel cela semble s'exécuter, non de l'exécution réelle.

P_1	P_2
/*Les valeurs initiales de A et B sont	0*/
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

- Valeurs possibles pour (A,B): (0,0), (1,0), (1,2); impossible : (0,2)
- 1a->1b et 2a->2b d'après l'ordre du programme
- A = 0 implique 2b->1a, ce qui implique 2a->1b
- B = 2 implique 1b->2a, d'où contradiction

- MAIS, l'exécution 1b->1a->2b->2a est SC, bien que non dans l'ordre du programme
  - Apparaît comme 1a->1b->2a->2b comme le montrent les résultats
- L'exécution 1b->2a->2b-> n'est pas SC

L'implémentation de la consistance séquentielle

- Deux types de contrainte
  - Ordre du programme
    - Les opérations mémoire effectuées par un processus doivent sembler apparaître (aux autres processus et à lui-même) dans l'ordre du programme
  - Atomicité
    - Dans l'ordre total de tous les processus, une opération mémoire doit se terminer par rapport aux autres processus avant que la suivante commence
    - On doit garantir que l'ordre total est consistant entre les processus
    - La partie difficile est de rendre les écritures atomiques

Consistance séquentielle et consistance relâchée

- Problèmes pour implanter la consistance séquentielle
  - Mécanismes pour tolérer la latence mémoire dans les microprocesseurs
    - Tampons d'écriture
    - Caches
    - Lectures non bloquantes
  - Le réseau d'interconnexion à la mémoire
    - Peut modifier l'ordre des écritures envoyés par les processeurs
  - Optimisations du compilateur
    - Réordonnancement des opérations mémoire
- Problèmes de performance
- Modèles de consistance relâchés (cf poly Cappello)
  - Consistance processeur
  - Ordre de rangement partiel
  - Ordre faible
  - Consistance à la libération

Bus et cohérence

- Sérialisation des écritures ?
  - Toutes les écritures qui apparaissent sur le bus (LeExBus) sont ordonnées par le bus.
    - Ecriture effectuée dans le cache du processeur qui écrit avant qu'il gère d'autres transactions. Les écritures sont ordonnées de la même manière., même par rapport au processeur qui écrit.
  - Les lectures qui apparaissent sur le bus sont ordonnés par rapport à ces écritures.
  - Les écritures qui n'apparaissent pas sur le bus
    - Une séquence de telles écritures entre deux transactions bus pour le bloc proviennent du même processeur (P).
    - Dans la sérialisation, la séquence apparaît entre ces deux transactions bus.
    - Les lectures par P lui paraîtront dans cet ordre par rapport aux autres transactions bus.
    - Les lectures par d'autres processeurs séparés de la séquence par une transaction bus, qui les placent dans un ordre sérialisé par rapport aux écritures.
    - Les lectures par les autres processeurs verront les écritures dans le même ordre

Bus et consistance séquentielle

- 1. Par rapport à la définition
  - Le bus impose un ordre total sur les transactions bus pour toutes les cases
  - Entre les transactions, les processeurs effectuent les lectures/écritures dans l'ordre du programme
  - Toute exécution définit un ordre partiel naturel
    - M_j suit M_i si j suit i dans l'ordre du programme pour le même processeur, ou si M_j génère une transaction bus qui suit une opération mémoire pour M_i
  - Dans un segment entre deux transactions bus, toute entrelacement d'opérations provenant de différents processeurs conduit à un ordre total consistant.
  - Dans un tel segment, les écritures observées par le processeur P sont sérialisés comme suit :
    - Les écritures des autres processeurs par la transaction bus précédente générée par P
    - Les écritures de P par l'ordre du programme
- 2. Les conditions à satisfaire
  - Fin des écritures : peut détecter quand une écriture apparaît sur le bus
  - L'atomicité des écritures : si une lecture reçoit le résultat d'une écriture, cette écriture est déjà devenue visible à tous les autres processeurs.

Synchronisation

- "Une machine parallèle est un ensemble de processeurs qui coopèrent et communiquent pour résoudre rapidement de gros problèmes".

- Types of Synchronisation
  - *Exclusion mutuelle*
  - Synchronisation d'évènements
    - *Point à point*
    - groupe
    - *global (barrières)*

Les 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 quand elle ne l'est pas
- Libération
  - Permet à d'autres processeurs d'acquérir le droit à la synchronisation.

- L'algorithme d'attente est indépendant de la synchronisation

Les algorithmes d'attente

- Bloquant
  - Les processus en attente sont désordonnancés
  - Surcoût élevé
  - Permet au processeur de faire autre chose
- Attente active
  - Les processus en attente testent en permanence une case jusqu'à ce que sa valeur change
  - Le processus libérant le verrou positionne la case
  - Surcoût plus faible, mais consomme des ressources processeur
  - Peut provoquer du trafic réseau
  - L'attente active est meilleure quand
    - Le surcoût d'ordonnancement est plus grand que le temps d'attente estimé
    - Les ressources processeur ne sont pas nécessaires pour d'autres tâches
    - Le blocage par l'ordonnanceur est inapproprié (par exemple dans le noyau du SE)
- Méthodes hybrides : attente active puis blocage

Un verrou logiciel simple

```
lock:     ld      register, location     /* copy location to register */
          cmp     location, #0           /* compare with 0 */
          bnz     lock                   /* if not 0, try again */
          st      location, #1    /      * store 1 to mark it locked */
          ret                            /* return control to caller */
unlock:   st      location, #0           /* write 0 to location */
          ret                            /* return control to caller */
```

- Problème : le verrou implique l'atomicité de son implémentation
  - La lecture (test) et l'écriture (positionnement) de la variable verrou par un processus n'est pas atomique
- Solution : instructions de lecture-modification-écriture ou d'échange atomique
  - Tester la valeur d'une case et l'écriture d'une autre valeur de manière atomique, et indiquer le succès ou l'échec de l'opération.

Instruction d'échange atomique

- Spécifie une case et un registre. Dans une opération atomique
  - La valeur dans la case est lue dans un registre.
  - Une autre valeur (fonction ou non de la valeur lue) est rangée dans la case
- Beaucoup de variantes
- Exemple simple : test&set
  - La valeur de la case est lue dans un registre spécifié
  - Constante 1 rangée dans la case
  - Succès si la valeur chargée dans le registre est 0
  - D'autres constantes peuvent être utilisées au lieu de 0 et 1
- Peuvent être utilisé pour réaliser des verrous

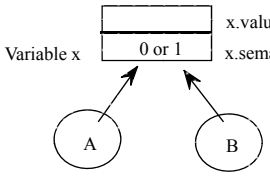
Verrou "Test & Set"

```
lock:     t&s    register, location
          bnz    lock                 /* if not 0, try again */
          ret                         /* return control to caller */
unlock:   st     location, #0         /* write 0 to location */
          ret                         /* return control to caller */
```

- D'autres primitives lecture-modification-écriture peuvent être utilisées
  - Swap
  - Fetch&op
  - Compare&swap
    - Trois opérandes : case, registre à comparer, registre pour permuter
    - Pas fournis couramment par les jeux d'instructions
- Peut être situé ou non dans le cache

Exemples d'utilisation

TEST AND SET



Test and set (x)
If x.sem. = 0**then** 0
else
x.sem :=0;
1

Process A ;
While Test&Set(x.sem) = 0 do nothing;
F(x.value) ; *travail sur x*
x.sem :=1
Process B ;
While Test&Set(x.sem) = 0 do nothing;
G(x.value) ; *Work on x*
x.sem :=1

FETCH AND ADD

F&A (V,e)
read V
V:V+e

Main program

X := 0 ; Y := N
Issue N process P i=1,N

While F&A (Y,0) ≠ 0 **do** nothing ;

For any process Pi = 1,N;

Begin

L := F&A (X,1) ;

Work on T[I]

F&A (Y,-1)
End

Amélioration du verrou simple

- Réduire la fréquence des test&sets en cours d'attente
  - Verrou Test&Set avec reprise
  - Reprise exponentielle efficace : $i^{ème}$ tentative = $k*c^i$
- Attente active avec lecture plutôt que test&set
  - *Verrou Test et Test&Set*
  - Teste avec des lectures ordinaires
    - La variable verrou dans le cache sera invalidée lors de la libération
  - Quand la valeur change (à 0), utilisation de test&set pour obtenir le verrou
    - Un seul réussira à obtenir le verrou ; les autres échoueront et recommenceront à tester.

Primitives matérielles : les instructions LL et SC

- Buts :
  - Tester avec des lectures
  - Les tentatives vaines lecture – modification - écriture ne provoquent pas d'invalidation
  - Obtenir beaucoup d'opérations lecture – modification - écriture avec une seule primitive
- *Load-Locked* (linked), *Store-Conditional*
  - LL lit la variable dans un registre
  - Puis nombre arbitraire d'instructions manipulant la variable
  - SC essaie de ranger dans la case mémoire si et seulement si personne d'autre n'a écrit dans la variable depuis le LL du processeur
    - Si succès de SC, les trois étapes se sont exécutées de manière atomique
    - Si échec, ne pas écrire ou générer des invalidations.

Verrou simple avec LL-SC

```
lock:       ll      reg1, location      /* LL location to reg1 */
            sc      location, reg2      /* SC reg2 into location*/
            beqz    reg2, lock          /* if failed, start again */
            ret
unlock:     st      location, #0        /* write 0 to location */
            ret
```

- Permettent de réaliser d'autres opérations atomiques en changeant les instructions entre LL et SC
  - Limiter le nombre d'instructions pour que SC soit un succès le plus souvent
  - Ne pas mettre d'instructions "irréversibles" comme les rangements
- SC peut échouer (sans mettre de transaction sur le bus) si
  - Il détecte une écriture en cours avant même d'essayer d'obtenir le bus
  - Il essaie d'obtenir le bus, mais le SC d'un autre processeur a déjà obtenu le bus.
- LL, SC ne correspondent pas à l'obtention ou la libération du verrou
  - Garantissent seulement qu'il n'y a pas entre eux d'écriture conflictuelle à la variable verrou.
  - Mais peuvent être utilisés directement pour implémenter des opérations simples sur des variables partagées.

Synchronisations par logiciel

- Point à point
  - Interruptions
  - Attente active : utilise des variables ordinaires comme drapeaux
  - Blocage : utilisation de sémaphores
- Barrières
  - Verrous
  - Drapeaux
  - Compteurs

Une barrière centralisée simple

- Un compteur partagé correspond au nombre de processus arrivé
  - Incrémente quand un processus arrive (lock)
  - Teste jusqu'à atteindre numprocs

```
struct bar_type {int counter; struct lock_type lock; int flag = 0;} bar_name;
BARRIER (bar_name, p) {
        LOCK(bar_name.lock);
        if (bar_name.counter == 0)
                bar_name.flag = 0;              /* reset flag if first to reach*/
        mycount = bar_name.counter++;           /* mycount is private */
        UNLOCK(bar_name.lock);
        if (mycount == p) {                     /* last to arrive */
                bar_name.counter = 0;                   /* reset for next barrier */
                bar_name.flag = 1;              /* release waiters */
        }
        else while (bar_name.flag == 0) {};   /* busy wait for release */
}
```

  - Problème
    - Les entrées consécutives de la même barrière ne fonctionnent pas
    - On doit empêcher un processus d'entrer avant que tous les processus de l'instance précédente n'ait quitté la barière

Une barrière centralisée opérationnelle

- Drapeau renversement du sens
  - Attendre que ce drapeau prenne une valeur différente avant de réutiliser la barrière
  - Le drapeau change de valeur seulement quand tous les processus ont atteint la barrière

```
BARRIER (bar_name, p) {
     local_sense = !(local_sense); /* toggle private sense variable */
LOCK(bar_name.lock);
     mycount = bar_name.counter++;          /* mycount is private */
     if (bar_name.counter == p)
            UNLOCK(bar name.lock);
            bar_name.flag = local_sense;   /* release waiters*/
     else
        {   UNLOCK(bar_name.lock);
            while (bar_name.flag != local_sense) {}; }
}
```