Module 3: Modèle de mémoire partagée pour la programmation parallèle

INF8610: Systèmes Informatiques Parallèles

Michel Dagenais

Différents points de vue

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Programme séquentiel, suite de lectures et d'écritures en mémoire.
- Le compilateur optimiseur génère du code assembleur et peut changer la séquence s'il n'affecte pas le résultat.
- L'unité centrale de traitement, en raison de son pipeline, ses tampons d'écriture et autres peut changer la séquence des accès aussi.
- L'ordre des changements en mémoire vus par les différents processeurs peut différer.
- Si deux fils d'exécution communiquent par mémoire partagée, il ne peuvent nécessairement se fier sur l'ordre!

Modèle séquentiel

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Dans quel ordre un processeur voit (lit) le travail d'un autre (écritures).
- Modèle séquentiel, le résultat doit correspondre au cas où tous les accès de chaque processeur sont effectués dans l'ordre, un par un, sur une mémoire partagée unique.
- Attendre qu'une écriture soit effective partout avant de continuer, ou attendre que les écritures de partout soient effectives avant de faire la lecture ou écriture suivante.

Modèles plus flexibles

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Le modèle séquentiel est trop contraignant et réduit beaucoup la performance.
- Différentes contraintes peuvent être relâchées, ce à quoi il faut pallier par des instructions de synchronisation explicites.
- Le modèle varie d'une architecture à l'autre.
- Le programmeur qui utilise des variables en mémoire partagée et utilise des primitives de synchronisation de haut niveau peut obtenir performance et portabilité.

Dans tous les cas

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Un processeur donné perçoit toujours ses opérations comme arrivant dans l'ordre demandé par le programme.
- Un accès mémoire ne sera réordonnancé avec une écriture que si les deux sont à des cases mémoire différentes.
- Les accès simples, alignés, sont atomiques. Un accès double mot, ou non aligné, pourrait être à moitié complété!
- Les opérations de synchronisation offertes par les librairies ou le système d'exploitation s'occupent de mettre les instructions requises pour séquencer les opérations correctement.

Réordonnancements

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les lectures peuvent avoir lieu avant une écriture à une variable différente qui les précédait, contrainte W->R relâchée (Total Store Order).
- Les lectures et écritures à des variables différentes peuvent être réordonnancées par rapport aux écritures, contraintes W->R et W->W relâchées (Partial Store Order).
- Tous les accès à des variables différentes peuvent être réordonnancés (Weak ordering).

Modèles de différentes architectures

ÉCOLE POLYTECHNIQUE MONTRÉAL

Architecture	R->R	R->W	W->W	W->R	DR	Pipeline
Alpha	O	O	O	O	O	O
AMD64				O		
IA64	O	O	O	O		O
PA_RISC	O	O	O	O		
POWER	O	O	O	O		O
SPARC RMO	O	O	O	O		O
SPARC PSO			O	O		O
SPARC TSO				O		O
x86				O		O
x86 OOStore	O	O	O	O		O
zSeries				O		O

Ecritures retardées après lectures

ÉCOLE POLYTECHNIQUE MONTRÉAL

Processeur 1

```
Flag1 = 1

if(Flag2 == 0) {

  /* section critique */
}
```

Processeur 2

```
Flag2 = 1

if(Flag1 == 0) {

  /* section critique */
}
```

Ecritures retardées après écritures

ÉCOLE POLYTECHNIQUE MONTRÉAL

Processeur 1

Data = 2000
Head = 1

Processeur 2

while(Head == 0) {;}
Value = Data

Lectures retardées après lectures

ÉCOLE POLYTECHNIQUE MONTRÉAL

Processeur 1

```
Data = 2000
smp_wmb()
Head = 1
```

Processeur 2

```
for(;;) {
  Ready = Head
  Value = Data
  /* Si Head n'est pas prêt,
     retardé par le pipeline,
     il pourrait être lu plus
     tard à 1 alors qu'il était
     0 lorsque Data est lu */
  if (Ready) break
}
```

Barrières mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Barrière mémoire pleine: **cmm_smp_mb()** en mode usager ou **smp_mb()** dans le noyau Linux. Lectures ou écritures qui précèdent effectuées avant toutes les lectures ou écritures qui suivent.
- Barrière mémoire pour lecture: **[cmm_]smp_rmb()**. Lectures qui précèdent effectuées avant les lectures qui suivent.
- Barrière mémoire pour écriture: **[cmm_]smp_wmb()**. Ecritures qui précèdent effectuées avant les écriture qui suivent.
- Barrière mémoire pour lectures dépendantes: **[cmm_]smp_read_barrier_depends()**. Lectures qui précedent effectuées avant les lectures dépendantes qui suivent.
- Barrière mémoire pour les E/S calquées sur la mémoire: **mmiowb()**. Correspond à un nop dans la plupart des cas car jumelé à un spinlock qui vient avec les barrières voulues.

Utilisation des barrières mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

Processeur 1

```
Data = 2000
cmm_smp_wmb()
Head = 1
```

Processeur 2

```
while(Head == 0) {;}
cmm_smp_rmb()
Value = Data
```

Utilisation des barrières mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

Processeur 1

```
a = 1
b = 2
cmm_smp_wmb()
c = 3
d = 4
```

Processeur 2

```
v = c
w = d
cmm_smp_rmb()
x = a
y = b
```

Utilisation des barrières mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

Processeur 1

```
Flag1 = 1
cmm_smp_mb()

if(Flag2 == 0) {

  /* section critique */
}
```

Processeur 2

```
Flag2 = 1
cmm_smp_mb()

if(Flag1 == 0) {

  /* section critique */
}
```

Utilisation des barrières mémoire

ÉCOLE POLYTECHNIQUE MONTRÉAL

{ M[0] == 1, M[1] == 2, M[3] = 3, P == 0, Q == 3 }

Processeur 1

```
M[1] = 4;
cmm_smp_wmb()
P = 1
```

Processeur 2

```
Q = P;
cmm_smp_read_barrier_depends()
D = M[Q];
```

Barrières par architecture

ÉCOLE POLYTECHNIQUE MONTRÉAL

- X86: smp_wmb() est *nop*, smp_rmb() et smp_mb() sont réalisés avec **lock,addl**.
- AMD64: smp_rmb() est **lfence**, smp_wmb() est **sfence**, smp_mb() est **mfence**.
- PowerPC: smp_rmb() est **lwsync**, smp_wmb() et smp_mb() sont **sync**.
- SPARC: smp_rmb() est **membar #LoadLoad**, smp_wmb() est **membar #StoreStore**, smp_mb() est **membar #LoadLoad | #LoadStore | #StoreStore| #StoreLoad.**

Vérification des accès concurrents

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Une barrière mémoire oubliée ou un algorithme incomplet peuvent causer une course qui se produit une fois sur des milliards.
- Comment déboguer un problème de ce genre? Tests intensifs sur multi-coeur avec assertions/trace pendant des jours?
- Modéliser l'algorithme et le matériel et le valider formellement en essayant toutes les combinaisons possibles (Model Checking).

Verrous

ÉCOLE POLYTECHNIQUE MONTRÉAL

- spinlock
- R/W spinlock
- mutex
- semaphores
- R/W semaphores
- RCU

Prise et relâche de verrou

ÉCOLE POLYTECHNIQUE MONTRÉAL

- LOCK: les accès mémoire après seront complétés après.
- UNLOCK: les accès mémoire avant seront complétés avant.
- LOCK... UNLOCK: certaines opérations avant et après peuvent se mélanger à l'intérieur, ce n'est pas une barrière mémoire complète.
- UNLOCK... LOCK: barrière mémoire complète entre avant et après.

Prise de verrou

```
        DADDUI R2, R0, #1
lockit: EXCH   R2, 0(R1)
        BNEZ   R2, lockit
```

```
lockit: LD     R2, 0(R1)
        BNEZ   R2, lockit
        DADDUI R2, R0, #1
        EXCH   R2, 0(R1)
        BNEZ   R2, lockit
```

Opération atomique

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
void atomic_add(int i, atomic_t *v)
{
        unsigned long flags;
        int temp;

        local_irq_save(flags);
        temp = v->counter;
        temp += i;
        v->counter = temp;
        local_irq_restore(flags);
}
```

```
void atomic_add(int i, atomic_t *v)
{
  asm volatile(LOCK_PREFIX "addl %1,%0"
                : "+m" (v->counter)
                : "ir" (i));
}
```

Opération atomique

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
struct el *insert(long key, long data)
{
  struct el *p;
  p = kmalloc(sizeof(*p),
      GPF_ATOMIC);
  spin_lock(&mutex);
  p->next = head.next;
  p->key = key;
  p->data = data;
  smp_wmb();
  head.next = p;
  spin_unlock(&mutex);
}
```

```
struct el *search(long key)
{
  struct el *p;
  p = head.next;
  while (p != &head) {
    smp_read_barrier_depends();
    if (p->key == key) {
      return (p);
    }
    p = p->next;
  }
  return (NULL);
}
```

Mise à jour atomique RCU

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Les verrous de lecture sont très problématiques sur multi-processeurs: écriture et invalidation du bloc en cache à répétition.
- Structures avec surtout des lectures, accédées via un pointeur d'entrée, et retrait qui peut être différé: lecture, copie avec modification, mise à jour atomique du pointeur (Read Copy Update), retour de l'espace libéré en différé après qu'il ne reste plus de lecteur dessus.

RCU: recherche dans une liste

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
 1 struct el {
 2   struct list_head lp;
 3   long key;
 4   spinlock_t mutex;
 5   int data;
 6   /* Other data fields */
 7 };
 8 DEFINE_RWLOCK(listmutex);
 9 LIST_HEAD(head);

 1 int search(long key, int *result)
 2 {
 3   struct el *p;
 4
 5   read_lock(&listmutex);
 6   list_foreach_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       *result = p->data;
 9       read_unlock(&listmutex);
10       return 1;
11     }
12   }
13   read_unlock(&listmutex);
14   return 0;
15 }
```

```
 1 struct el {
 2   struct list_head lp;
 3   long key;
 4   spinlock_t mutex;
 5   int data;
 6   /* Other data fields */
 7 };
 8 DEFINE_SPINLOCK(listmutex);
 9 LIST_HEAD(head);

 1 int search(long key, int *result)
 2 {
 3   struct el *p;
 4
 5   rcu_read_lock();
 6   list_foreach_entry_rcu(p,&head,lp){
 7     if (p->key == key) {
 8       *result = p->data;
 9       rcu_read_unlock();
10       return 1;
11     }
12   }
13   rcu_read_unlock();
14   return 0;
15 }
```

RCU: retrait de la liste

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   write_lock(&listmutex);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del(&p->lp);
 9       write_unlock(&listmutex);

10       kfree(p);
11       return 1;
12     }
13   }
14   write_unlock(&listmutex);
15   return 0;
16 }
```

```
 1 int delete(long key)
 2 {
 3   struct el *p;
 4
 5   spin_lock(&listmutex);
 6   list_for_each_entry(p, &head, lp) {
 7     if (p->key == key) {
 8       list_del_rcu(&p->lp);
 9       spin_unlock(&listmutex);
10       synchronize_rcu();
11       kfree(p);
12       return 1;
13     }
14   }
15   spin_unlock(&listmutex);
16   return 0;
17 }
```

RCU: implémentation

ÉCOLE POLYTECHNIQUE MONTRÉAL

```
void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }

void call_rcu(void (*callback) (void *),
    void *arg)
{
  // add callback/arg pair to a list
}

void synchronize_rcu(void)
{
  int cpu, ncpus = 0;

  for_each_cpu(cpu)
  schedule_current_task_to(cpu);

  for each entry in the call_rcu list
    entry->callback (entry->arg);
  }
}
```

```
// Not called since delete() uses mutex
#define rcu_assign_pointer(p, v)
{
  smp_wmb();
  (p) = (v);
}

// Called by list_for_each_entry_rcu()
#define rcu_dereference_pointer(p)
{ typeof(p) _value = (p);
  smp_rmb();
  (_value);
}
```

Programmation en mémoire partagée

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Décomposer le travail en minimisant les interactions (écritures dans des variables partagées entre les fils d'exécution).
- Synchroniser les accès aux variables partagées.
- Lecture sans verrou, écriture atomique.
- Verrou associé à une ou des variables.
- Structures mises à jour par RCU.

Verrous

ÉCOLE POLYTECHNIQUE MONTRÉAL

- spinlock: section critique très courte qui ne peut dormir.
- R/W spinlock: beaucoup de lectures parfois un peu longues.
- mutex: section critique longue ou qui peut dormir.
- semaphores: compteur de ressources, mis à jour par plus d'un fil d'exécution.
- R/W semaphores.
- RCU: les verrous usuels font une écriture à chaque fois, très mauvais avec un grand nombre de processeurs.

Protection des variables par CPU

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Processus (noyau) avec signal (préemption), assurer néanmoins la réentrance pour les variables locales à un thread (CPU).
- Section critique du noyau modifiant des variables et utilisant le CPUid: désactiver la préemption ou verrouiller.
- Dans le noyau, désactiver les interruptions désactive la préemption si aucun appel n'est fait qui puisse causer un réordonnancement.

Barrière de rendez-vous pour fils

ÉCOLE POLYTECHNIQUE MONTRÉAL

- Attendre que plusieurs fils d'exécution aient atteint un certain point.
- Compteur: le coordonnateur attend de recevoir un message de chacun (n messages) puis envoie un message de débloquage.
- Arbre: arbre de décomposition du problème, la racine attends après les enfants récursivement puis envoie le débloquage en sens inverse.
- Papillon: chaque fil communique avec un autre à chaque étape (i+1 % n, i+2 % n, i+4 % n...). Tous reçoivent l'information que tous sont prêts en même temps. Pas de phase de d'annonce de fin!