Programmation parallèle en mémoire partagée

Module 3

INF8601 Systèmes informatiques parallèles

Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire

Programmation parallèle en mémoire partagée

Processus et fils d'exécution

- Continuer le traitement avec un autre fil lorsqu'un fil est bloqué; utiliser plusieurs processeurs en parallèle;
- Processus conventionnel: mémoire séparée par défaut; mémoire partagée peut être ajoutée;
- Fils d'exécution en mode usager, coopératif, préemptif;
- Processus légers en mode usager;
- Processus légers gérés par le système d'exploitation;
- Solaris: mélange de processus légers en mode usager et en mode noyau;
- Linux: tout est un processus, différents attributs (mémoire, descripteurs...) peuvent ou non être partagés.

Le vrai coût en performance?

Processeur Intel i5, cycle de 0.4ns, LMBench

Opération	Temps ns
syscall trivial	65
read	170
write	138
stat	778
fstat	205
open/close	1492
select 10 fd	515
select 500 fd	7022
signal	1348
segfault	158
délai de tuyau	22716

Opération	Temps ns
fork+exit	263000
fork+execve	269400
pthread_create	∼19000
changer de processus	∼1600
changer de fil	∼1400
iadd	.2
imul	.18
idiv	9.52
dadd	1.32
ddiv	12.05

Le vrai coût en performance?

Processeur Intel i7, cycle de 0.47ns, LMBench

Opération	Temps ns
syscall trivial	43
read	120
write	74
stat	437
fstat	115
open/close	895
select 10 fd	275
select 500 fd	4227
signal	861
segfault	380
délai de tuyau	4228

Opération	Temps ns
fork+exit	116023
fork+execve	137390
pthread_create	
changer de processus	
changer de fil	
iadd	.16
imul	.02
idiv	7.6
dadd	0.95
ddiv	4.48

Technologie Hyperthread

- Deux ensembles de registres pour un même processeur;
- Apparaît comme deux processeurs;
- Flots d'instructions entrelacés;
- Lorsqu'un processeur *virtuel* bloque sur une faute de cache, l'autre prend toute la place;
- Peut confondre le système d'exploitation ou les applications parallèles et nuire à leurs stratégies d'optimisation.

Programmation parallèle en mémoire partagée

Fils d'exécution en mode usager

- Création: allouer pile, insérer entrée dans la table des fils d'exécution, créer un contexte initial qui pointe vers la pile;
- Fin: retirer l'entrée, désallouer la pile et le contexte, ou les conserver pour recyclage;
- *Yield*: sauver le contexte courant (setjmp), changer pour le contexte d'une autre tâche (longjmp);
- Préemption: demander une interruption régulière, sauver le contexte courant, changer pour le contexte d'une autre tâche.
- Appels systèmes bloquants? Multi-processeur?

Processus régulier

- Cloner le processus courant (fork);
- Le processus courant continue et éventuellement attend le résultat de l'enfant (waitpid);
- Le processus enfant ((pid = fork()) == 0) démarre souvent un autre exécutable (exec); il peut allouer des zones de mémoire partagée;
- L'enfant a le même contenu initial de mémoire virtuelle (COW), une copie des descripteurs de fichiers...
- L'enfant n'hérite pas des signaux, temporisateurs, verrous.

Fils d'exécution

- Création: pthread_create(&tid, ...);
- Le parent attend souvent après l'enfant: pthread_join(tid, ...);
- L'enfant partage la mémoire et toutes les ressources avec le parent mais utilise une partie différente de leur mémoire pour sa pile;
- Qui reçoit le signal envoyé au processus?

Avantages et inconvénients

- Même espace mémoire et table de page pour les fils d'exécution d'un même processus; plus compact, et création et changement de contextes plus rapides.
- Taille moins flexible pour les piles;
- Tout usage concurrent de données partagées doit être protégé par des verrous ou accédé avec des opérations atomiques.

Réentrance versus concurrence

- Réentrance: une interruption peut survenir n'importe quand. La routine d'interruption pourrait accéder des structures de données qui étaient en train d'être modifiées;
- Concurrence: deux fils d'exécution s'exécutent sur deux processeurs simultanément et pourraient accéder en même temps les mêmes structures de données.

Carte de la mémoire

```
$ cat /proc/self/maps
557f45ebc000-557f45ebe000 r--p 00000000 103:02 1048728  /usr/bin/cat
557f45ebe000-557f45ec3000 r-xp 00002000 103:02 1048728  /usr/bin/cat
557f45ec3000-557f45ec6000 r--p 00007000 103:02 1048728  /usr/bin/cat
557f45ec6000-557f45ec7000 r--p 00009000 103:02 1048728  /usr/bin/cat
557f45ec7000-557f45ec8000 rw-p 0000a000 103:02 1048728  /usr/bin/cat
557f4703f000-557f47060000 rw-p 00000000 00:00 0         [heap]
7f4f65664000-7f4f65689000 r--p 00000000 103:02 1055882  /usr/lib/libc-2.31.so
7f4f65689000-7f4f65801000 r-xp 00025000 103:02 1055882  /usr/lib/libc-2.31.so
7f4f65801000-7f4f6584b000 r--p 0019d000 103:02 1055882  /usr/lib/libc-2.31.so
7f4f6584b000-7f4f6584c000 ---p 001e7000 103:02 1055882  /usr/lib/libc-2.31.so
7f4f6584c000-7f4f6584f000 r--p 001e7000 103:02 1055882  /usr/lib/libc-2.31.so
7f4f6584f000-7f4f65852000 rw-p 001ea000 103:02 1055882  /usr/lib/libc-2.31.so
7f4f65852000-7f4f65858000 rw-p 00000000 00:00 0
7f4f65867000-7f4f65889000 rw-p 00000000 00:00 0
7f4f65889000-7f4f6588a000 r--p 00000000 103:02 1055669  /usr/lib/ld-2.31.so
...
7f4f658b8000-7f4f658b9000 rw-p 00000000 00:00 0
7fff8c5e6000-7fff8c607000 rw-p 00000000 00:00 0         [stack]
7fff8c7ef000-7fff8c7f2000 r--p 00000000 00:00 0         [vvar]
7fff8c7f2000-7fff8c7f3000 r-xp 00000000 00:00 0         [vdso]
```

Programmation parallèle en mémoire partagée

Différents points de vue

- 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... 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

- 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

- Le modèle séquentiel est trop contraignant et réduit beaucoup la performance.
- Différentes contraintes peuvent être relâchées, ce qu'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

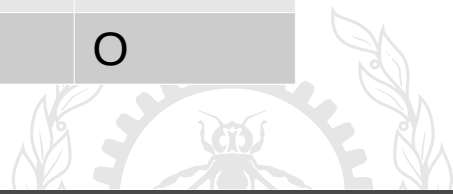
- Un processeur donné perçoit toujours ses opérations comme arrivant dans l'ordre demandé par le programme.
- Un accès mémoire ne sera réordonnancé avec une écriture que si les deux sont à des cases mémoire différentes.
- Les accès simples, alignés, sont atomiques. Un accès double mot, ou non aligné, 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

- 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

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

```
Processeur 1

Flag1 = 1
if(Flag2 == 0) {

  /* section critique */
}
```

```
Processeur 2

Flag2 = 1
if(Flag1 == 0) {

  /* section critique */
}
```

Ecritures retardées après écritures

```
Processeur 1

Data = 2000
Head = 1
```

```
Processeur 2

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

Lectures retardées après lectures

```
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
}
```

Programmation parallèle en mémoire partagée

Barrières mémoire

- 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

```
Processeur 1

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

```
Processeur 2

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

Utilisation des barrières mémoire

```
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

```
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

```
{ 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

- 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

- 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

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

Prise et relâche de verrou

- 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

```
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

```
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

- 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

```
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 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);
```

RCU: recherche dans une liste (suite)

```
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  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

```
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);
}
```

Verrous

- 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

- 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

- 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!

Programmation parallèle en mémoire partagée

Conclusion

- 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.