08 - Concurence

Gaétan Richard
gaetan.richard@unicaen.fr

L2 – S4 2011/2012

I. Parallélisme

”Faux parallélisme”

Principe : Le système d’exploitation donne l’impression que les programme s’exécutent en parallèle en alternant suffisamment rapidement sur une unité de calcul.

Utilisation : En général efficace pour une utilisation personnelle.

Gros calculs

Principe : Certains calculs demandent une très forte puissance de calcul (ex : prévisions météorologique, simulation de phénomènes physiques, …)

Solution : On utilise un grand nombre d'unités de calculs.

Problème : Il faut souvent échanger des informations, se synchroniser entre les unités.

Mise en réseau

Alternatives : Les unités de calcul peuvent être câblées :

- En clique ;
- En anneaux ;
- En grille (bi ou tri dimensionnelle) ;
- En hypercube ;
- …

À chaque fois, il s'agit d'un compromis entre le coût (matériel et au niveau du calcul) du câblage et la facilité de programmation.

Quelques “grosses” machines

Top 10 : (issu du top 500, novembre 2011)

n°	Pays	Fabriquant	nb de cœurs	Rmax (PFlop)	date
1	Japan	K computer	705 024	10 510.00	2011
2	China	NUDT	186 368	2 566.00	2010
3	United States	Cray Inc.	224 162	1 759.00	2009
4	China	Dawning	120 640	1 271.00	2010
5	Japan	NEC/HP	73 278	1192.00	2010
6	United States	Cray Inc.	153 408	1 054.00	2011
7	United States	SGI ALix	111 104	1 088.00	2011
8	United States	Cray	153 408	1 054.00	2010
9	France	Bull SA	138 368	1050.00	2010
10	United States	Blade Center	122 400	1 042.00	2009

Algorithmique parallèle

Problème : Pour utiliser de telles machines, il faut séparer les calculs en blocs “le plus indépendant” possibles et qui tiennent compte de la structure de la machine.

Recherche : Ces questions relèvent du domaine de l’algorithmique parallèle.

À l’avenir : Pour le moment, ces questions sont réservées à des machines “d’exception” mais la multiplication des cœurs en fait un domaine émergeant dans les machines grand publique.

Programmation parallèle

Performance : Il est quasi impossible de rendre parallèle un code initialement conçu en séquentiel.

Programmation : Si vous programmez un programme ayant besoin de puissance de calcul, il faut réfléchir à l'avance à la séparation du calcul en tâches indépendantes et parallèlisables.

2. Synchronisation

Threads

Principe : À l'opposé des processus qui sont indépendants, nous avons vu qu'il existe un moyen d'introduire du parallélisme à l'intérieur d'un même processus à l'aide de threads.

Avantage : Les threads partagent le même espace mémoire d'où une communication triviale.

Inconvénient : Les threads partagent le même espace mémoire d'où des problèmes de concurrence.

Exemple de problèmes

Démonstration

Solutions

Idée : Utiliser des primitives de synchronisation.

Principe : Les primitives permettent un accès exclusif à une valeur qui servira de barrière.

Bonus : Ces primitives mettront également en œuvre un mécanisme d'attente.

Verrous

Verrou : (**lock** en anglais) Une valeur booléenne qui peux être prise ou relâchée de façon atomique. Trois opérations :

- Création ;
- Acquisition (de façon bloquante ou non) ;
- Libération (doit être fait par celui ayant pris le verrou).

Si l'acquisition est bloquante, le thread est endormi. Il sera réveillé lorsque le verrou sera libre.

Le verrou sert à protéger des sections critiques

En python

- Création : **threading.Lock()**
- Acquisition : **Lock.acquire(*[blocking=1]*)**
- Libération : **Lock.release()**

Note : Il existe également une notion de *reentrant lock* (Rlock) qui est lié à un thread.

Sémaphores

Principe : Une sémaphore est un entier qui peut être incrémenté / décrémenté de manière atomique. Si celui-ci est à 0, la décrémentation est bloquante.

Il existe 3 opérations principales :

- Création à une valeur donnée ;
- Incrémentation (**V**, *Verhogen*, incrémenter) ;
- Décrémentation (**P**, *Proberen*, tester).

Elles ont été formalisées par *Edsger Dijkstra*.

En python

Primitives :

- Création : **class threading.Semaphore(*[value]*)** ;
- Décrément : **acquire(*[blocking]*)** ;
- Incrément : **release()**.

Conditions

Principe : Les conditions servent à synchroniser des threads : l'envoie d'une condition réveille tous les threads qui attentaient sur cette condition.

4 primitives :

- Création ;
- Attendre une condition ;
- Réveiller un thread en attente de cette condition ;
- Réveiller tous les threads en attente de cette condition.

En python

Note : En python, une condition combine également les fonctions d'un verrou.

Primitives :

- Création : **class threading.Condition(*[lock]*)** ;
- Acquisition : **acquire(**args*)** ;
- Libération : **release()** ;
- Attente : **wait(*[timeout]*)** ;
- Réveil unique : **notify()** ;
- Réveil de tous : **notify_all()** ;

Section critique

La principale utilisation des verrous est d'empêcher la concurrence dans des sections critiques. Le code est alors assuré d'être exécuté par un seul thread à la fois.

Exemple :

```
...

lock.acquire()
# section critique
lock.realease()

...
```

Deadlock

Il est possible (et même courant) d'avoir plusieurs lock différents protégeant l'accès à des ressources. Dans ce cas, il peut surgir le problème de **deadlock** (inter-blocage en français).

Thread 1

```
lock1.acquire()
lock2.acquire()
# du code
lock2.realease()
lock1.realease()
```

Thread 2

```
lock2.acquire()
lock1.acquire()
# encore du code
lock1.realease()
lock2.realease()
```

Famine

Remarque : Dans certains cas, le présence de lock et de sémaphore peut faire en sorte qu'un ou plusieurs threads n'aient jamais accès à l'unité de calcul. On parle alors de famine.

À propos de ces problèmes

Reproduction : Ces problèmes étant liés à de la concurrence, il sont très difficiles à reproduire et peuvent rester ignorés pendant de très longue périodes.

Détection : Il existe quelques outils permettant de détecter de façon statique des problèmes potentiels ou de reconnaître à l'exécution lors de l'apparition d'un deadlock.

Solution : Il faut être prudent et bien réfléchir avant de mettre en place des systèmes multiples de locks.

3. Problèmes classiques

Problèmes classiques

Principe : L'utilisation des verrous et sémaphores fait l'objet de nombreux exemples. On distingue cependant trois exemples classiques qui se retrouvent partout dans la littérature.

L'accès à une ressource partagée

Au tableau.

Lecteurs - écrivains

Au tableau.

Les Philosophes

Au tableau.

4. Mise en œuvre

Matériel

Processeurs : Dans les processeurs, il existe des instructions d'échange qui s'effectuent de façon atomique.

Multi-core : Dans le cas du multicore, ces instructions sont étendues pour rester efficace.

Mise en œuvre : les verrous et sémaphores sont mises en place à l'aide d'un jeton et de ces opérations d'échange.