INGI 1113 - Systèmes d'exploitation

Rapport final
-
Projet TracesWeb

LAMOULINE Laurent
3597-05-00

NUTTIN Vincent
5772-05-00

25 mars 2009

```
Welcome in 'TraceWeb Group 14'!
=> 325 items have been processed by dezipMan !

90 % remaining ...
80 % remaining ...
70 % remaining ...
60 % remaining ...
50 % remaining ...
40 % remaining ...
30 % remaining ...
20 % remaining ...
10 % remaining ...
0 % remaining ...

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Results ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Rank                 HashCode            Timestamp   ClientID
----                 --------            ---------   --------
1       0xf9b0f7a140564787819f6537b92a2c8e 893972518    248
1       0xf9b0f7a140564787819f6537b92a2c8e 893973432    248
2       0xf530314b60b5fe0f0721a5ce3a64416c 893972510    439
2       0xf530314b60b5fe0f0721a5ce3a64416c 893973410    439
3       0xf1499bd9d7a565c5db30ab2b26a092a7 893972382    34602
3       0xf1499bd9d7a565c5db30ab2b26a092a7 893973597    34602
3       0xf1499bd9d7a565c5db30ab2b26a092a7 893972996    34602

>>> >>> Execution time : 11.675000 <<< <<<
```

1 Introduction

Dans le cadre du cours *INGI 1113 - Systèmes d'exploitation*, nous avons été amenés à concevoir un programme destiné à analyser des requêtes effectuées par des clients sur les serveurs du site web officiel de la coupe du monde de football 1998. Ce rapport présente brièvemenent les différentes étapes par lesquelles nous sommes passés afin de mener à bien ce projet. Nous commencerons par justifier rapidement les choix que nous avons fait concernant l'architecture globale du programme ; et principalement la façon dont nous avons choisi de gérer la communication entre les diverses parties concurrentes de notre programme. Ensuite, nous présenterons quelques données de performances de notre solution. Et enfin, nous proposerons des améliorations que nous pourrions apporter à notre programme afin d'en améliorer les performances.

2 Architecture globale

Afin de pouvoir traiter efficassement les demandes de l'utilisateur de notre programme, nous avons choisi une architecture concurrente. Nous avons utilisé plusieurs threads qui effectuent des opérations en parrallèle.

L'architecture de base est illustrée sur le schéma suivant :

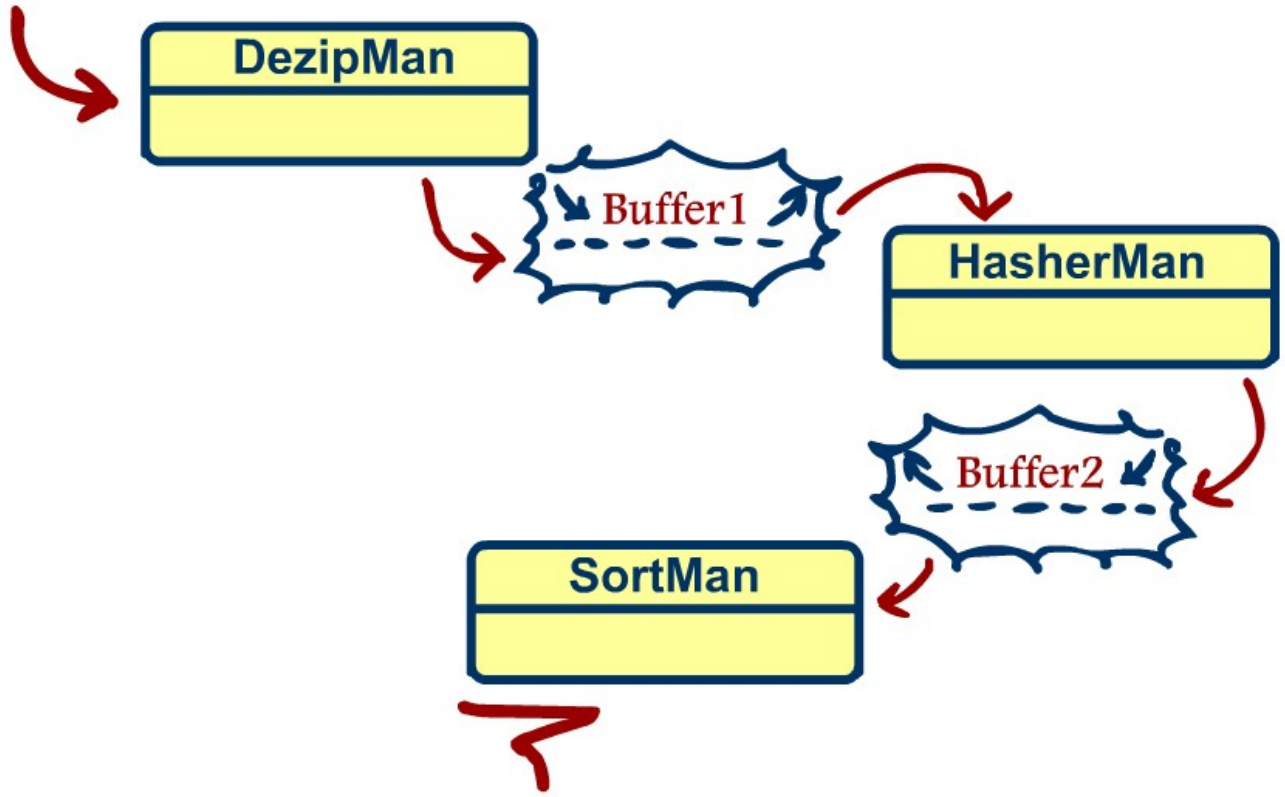

FIGURE 1 – Vue d'ensemble

Comme on peut le constater, notre programme est constitué de trois principaux agents : `DezipMan`, `HasherMan`, `SortMan` dont nous allons détailler le rôle ci-dessous ; ainsi que deux buffers.

2.1 DezipMan

Ce thread se charge tout d'abord de lire le fichier donné en argument à notre programme afin d'identifier les archives à traiter. Le nom de ces archives est lu un à un dans le fichier et pour chaque archive, nous procédons à la lecture des requêtes ci-trouvant. Ces dernières sont ensuite placées dans une structure de type `request`. Il s'agit alors de créer une nouvelle structure `request_perso` qui ne contiendra que les données dont nous avons besoin dans la suite du traitement afin d'éviter de communiquer des informations inutiles aux autres threads. Cette structure sera alors placée dans un buffer (`Buffer1`). Ce buffer sera partagé avec un autre thread (`HasherMan`) qui s'occupera de la suite du traitement. Une fois les données mises dans le buffer, le thread `DezipMan` poursuit la lecture du fichier de départ et recommence les opérations précitées avec l'archive suivante. Quand il arrive à la fin du fichier, il place dans le buffer une structure spécifique qui servira de balise de fin de traitement.

2.2 Buffer1

Ce buffer est rempli par `DezipMan` et lu par `HasherMan`. Il contient un certain nombre de structures de type `request_perso`. Nous avons décidé de borner ce buffer à une taille spécifique ($2000 \times sizeof(struct\ \ request\_perso)$) pour éviter que si `HasherMan` est vraiment trop lent, ce buffer ne se remplisse et prenne toute la mémoire. Ce paramètre (la taille du buffer) peut être facilement modifié et nous montrerons l'effet de ces changements dans la partie consacrée aux performances de notre application.

2.3 HasherMan

Il lit les structures présentes dans `Buffer1` afin de calculer la valeur de hash correspondant à l'ID du client. Quand ce calcul est fait, il place ce résultat dans la structure concernée. Par la suite, il place à nouveau la structure dans un buffer (`Buffer2`) afin d'essayer d'amortir les différences de vitesses entre les sous-traitements.

2.4 Buffer2

Ce buffer est rempli par `HasherMan` et lu par `SortMan`. Il contient un certain nombre de structures de type `request_perso`. Nous avons également décidé de borner ce buffer à une taille spécifique ($2000 \times sizeof(struct\ \ request\_perso)$) pour éviter que ce buffer ne se remplisse et prenne toute la mémoire au cas où la suite du traitement prendrait beaucoup de retard.

2.5 SortMan

`SortMan` récupére les structures placées dans `Buffer2`. Son rôle est de tenir à jour une liste doublement chaînée de taille fixée par le second argument `X` donné à notre programme. `SortMan` doit en réalité trier les différentes requêtes en fonction de leur valeur de hash. Ce seront donc les `X` requêtes présentant les plus grandes valeurs de hash qui seront conservées dans la chaîne à chaque étape du traitement. `SortMan` pourra repérer la balise de fin de traitement grâce aux valeurs spécifiques qui la carractérise. Quand `SortMan` a détecté la structure de fin de traitement, le résultat est affiché à l'écran.

2.6 Commentaires relatifs à l'architecture en général

Nous allons ici détailler quelques éléments évoqués ci-dessus.

2.6.1 request_perso

La structure `request_perso` a typiquement l'aspect suivant :

```
struct request_perso {
       uint32_t timestamp;
       uint32_t clientID;
       unsigned char *hashCode;
}
```

Dans un premier temps, quand `DezipMan` doit remplir cette structure, il met le champ `hashCode` (encore inconnu à cette étape du traitement) à `NULL` et les champs `timestamp` et `clientID` seront simplement copiés de la structure de base `request`. C'est ensuite `HashMan` qui désignera la bonne valeur à donner à ce champ `hashCode`.

2.6.2 Buffers

Les deux buffers sont protégés avec des sémaphores. Cela nous permet d'éviter que différents threads n'accèdent à une zone critique de la mémoire partagée en même temps. Nous avons utilsé les sémaphores de la librairie POSIX `<semaphore.h>`. Etant donné que les sémaphores sont des resources limitées sur les machines, nous avons veillés à utiliser ceux-ci de manière optimale et sans oublier de les libérer à la fin de nos différents traitements.

2.6.3 Threads

L'ensemble du programme s'éxécute dans un seul processus. Néanmoins, chaque partie possède un certain nombre de threads qui est adapté en fonction du temps que prend chaque opération : Une opération qui s'éxécute très rapidement par rapport à une autre (par exemple, `dezipMan` est beaucoup plus rapide que `hasherMan`) nécéssitera moins de thread. Des mesures expérimentales nous ont permis de determiner un nombre spécifique de threads amenant notre application à des résultats rapides. Dans le cas d'une machine dotée de moins de quatre coeurs CPU, nous avons opté pour 3 threads pour l'opération de dézippage, 5 threads pour l'opération de hash et 5 threads pour tenir la liste chaînée triée. Ce qui nous fait un total de $3 + (3 + 5 + 5) = 16$ threads comme nous pouvons le visualiser sur la figure suivante.

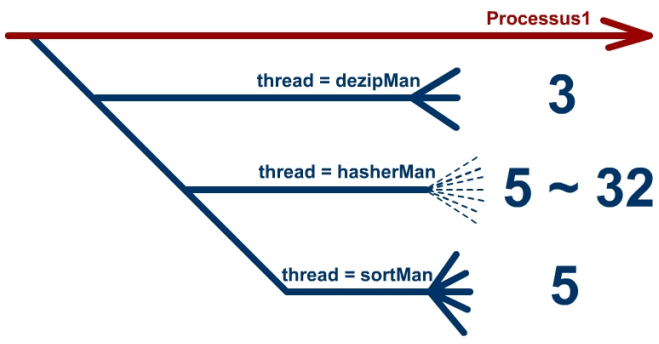

FIGURE 2 – Schéma global : threads

Dans le cas de l'utilisation de notre application sur `Sirius` (qui dispose de 8 processeurs capables de faire tourner 4 threads en parrallèle chacun ; ce qui correspond à 32 processeurs virtuels), notre programme s'adapte automatiquement à la compilation afin de garder le même nombre de threads pour les opérations de dézippage et de triage, mais augmente son nombre de threads pour calculer les valeurs de hash à 32. Ce qui nous fait un total de $3 + (3 + 32 + 5) = 43$ threads.

3 Performances

3.1 Temps d'éxécution

Voici quelques données expérimentales concernant les performances de notre application :

Performances sur Sirius avec 'test_log.gz' (1000 requêtes)					
Buffer1	Buffer2	#threads dezipMan	#threads hasherMan	#threads sortMan	Temps (sec)
20	1500	3	32	5	17
1500	20	3	32	5	17.2
2000	2000	1	1	1	327
2000	2000	1	20	1	20.3
2000	2000	3	1	1	324
2000	2000	1	1	5	319
2000	2000	**3**	**32**	**5**	**16.8**

Performances sur un PC Dual Core 2,4GhZ avec 'test_log.gz' (1000 requêtes)					
Buffer1	Buffer2	#threads dezipMan	#threads hasherMan	#threads sortMan	Temps (sec)
20	1500	3	5	5	20
1500	20	3	5	5	21.3
2000	2000	1	1	1	19.3
2000	2000	1	20	1	20.3
2000	2000	3	1	1	19
2000	2000	1	1	5	19.5
2000	2000	3	5	5	18.3

La première observation est que la taille des buffers n'est pas un élément déterminant pour les performances de notre programme. Aussi bien sur Sirius que sur notre ordinateur personnel, nous remarquons que le fait d'échanger les tailles des buffers 1 et 2 a peu d'impact sur le temps d'éxécution. Il apparait cependant que la façon dont les threads sont répartis dans les trois entités joue un rôle prépondérant. Nous parlons ici de petites différences de quelques secondes pour traiter 1000 requêtes mais nous imaginons bien que pour traiter des millions de requêtes, ces différences de temps vont vite avoir leur importance. Enfin, comme dernière observation, nous pouvons dire que les meilleurs performances ont été obtenues sur Sirius comme vous pouvez le voir sur le tableau ci-dessus.

3.2 Particularité

Une particularité interessante de notre application se situe au niveau de l'algorithme de tri. En effet, celui-ci est implémenté via une liste doublement chaînée dont les noeuds contiennent encore eux-même des références vers une autre liste doublement chaînée destinée à contenir les noeuds dont les valeurs de hash sont identiques. On remarque que l'ajout et la suppression d'un noeud à la liste se fait en $O(n)$. La méthode de tri consiste à comparer les valeurs de hash du noeud à ajouter et du dernier de la liste. Si le nouveau

noeud présente une valeur de hash plus petite, le noeud n'est pas pris en compte. Par contre, s'il présente une valeur de hash égale au noeud courrant, il est ajouté à la sous-liste (dont nous avons parlé ci-dessus) de ce dernier. Enfin, s'il présente une valeur de hash plus grande, nous nous déplaçons vers le haut de la liste pour effectuer une nouvelle comparaison... jusqu'à, peut-être, arriver tout en haut de la liste.

4 Problèmes et améliorations

Le principal problème rencontré fut la synchronistion des threads. En effet, il fallait à tout prix empêcher plusieurs threads de lire ou écrire sur la même zone de mémoire en même temps. Comme expliqué précédement, nous avons utilisé des sémaphores afin de protéger les accès aux zones critiques du programme. Un autre problème qui s'est posé à nous était la gestion de la mémoire. Nous avons utilisé `Valgrind` afin de trouver les fuites possibles de mémoire.

Ce point particulier nous amène aux améliorations possibles de notre programme. Il serait interessant d'encore se plonger dans le code car, malheureusement, nous ne sommes pas parvenus à libérer totalement la mémoire que nous utilisons. Une autre amélioration possible serait d'éviter de faire appel à la fonction de hash à chaque requête lue dans l'archive. En effet, nous calculons la valeur de hash de l'ID des clients effectuants les requêtes mais beaucoup de ses ID sont redondants. Ce serait donc un gain de temps de conserver les valeurs de hash des ID précédement rencontrés.

5 Conclusion

Pour conclure, nous ne pretendons pas avoir trouvé la solution optimale au problème posé. Comme nous l'avons dit, certaines améliorations sont envisageables, voire nécéssaires. Cependant, les resultats qu'il est possible d'obtenir avec notre réalisation peuvent l'être en un temps raisonnable et de manière fiable.