Commentaires sur le devoir à la maison

Christophe Cérin

18 novembre 2009

1 Préambule

Premièrement j'ai vraiment l'impression que vous n'avez jamais rédigé le moindre rapport ou compte rendu de votre vie. Dans un compte rendu, il faut faire une introduction, une conclusion afin d'exposer vos choix (les hypothèses), justifier vos approches et les citer. Il faut aussi systématiquement rédiger un algorithme qui permet justement de répondre succintement aux précédents points et cela permet aussi de dégager les grandes structures d'organisation. Ensuite vous passez à la codification. La conclusion résume votre travail en montrant quel est le point fort de votre travail.

Sachez discerner ce que l'on doit faire et comment on le fait.

Deuxièmement, concernant le bout de programmation Java, personne (à part 2 ou 3 exceptions) ne s'est posé la question des différences et des approches Java pour exprimer la mémoire commune et la notion de processus. Il fallait répondre et vous poser la question de savoir comment on fait pour creer des processus en Java, comment ils se synchronisent et comment ils partagent de l'information... et en fait pourquoi on n'a pas besoin du tralala Unix (mmap, ftruncate...) que l'on a vus dans le cours et les TP. Pourquoi utilisez-vous IsAlive() qui fait de l'attente active : est-ce que cela correspond à la sémantique du wait() Unix ? Il faut justifier votre travail.

En conclusion, il y a du pain sur la planche ! nous allons voir ensemble ce qu'il est possible de faire autour des deux questions proposées.

2 La recherche de la valeur maximale

2.1 Le sujet

Vous disposez d'un tableau d'entiers T de taille N. On vous demande de trouver le maximum en déléguant le travail à plusieurs threads. Par exemple, pour N=4 et deux processus (p=2), on peut procéder comme suit : le premier processus compare T[0] et T[1] et garde le plus grand ; le deuxième processus compare T[2] et T[3] dans le même temps et renvoie le plus grand. Il reste à déterminer ensuite qui est le plus grand parmi 2.

Votre programme C utilisera de la mémoire partagée pour ranger les résultats intermédiaires et devra fonctionner quelque soient N, p

Vous commencerez par donner un algorithme en expliquant les sous-problèmes que vous traitez pour arriver à la solution.

Vous discuterez ensuite du temps d'execution de votre programme en comptant le nombre de comparaisons entre T[i] et T[j] qu'il effectue (attention, les comparaisons peuvent s'effectuer en parallele). Vous exprimerez ce temps d'execution en fonction de N et de p. S'agit-il de $(log_2 N)/p$?

2.2 La (une) réponse

Dans ce qui sui nous allons supposer que p divise N ce qui va éviter de discuter de cas particuliers. Pour trouver le plus grand dans un vecteur de N entiers, nous pouvons utiliser l'algorithme suivant :

Étape 1 : chaque processus $1 \leq i \leq p$ travaille sur une portion distincte de T de taille N/p et déroule un algorithme séquentiel qui effectue donc $N/p - 1$ comparaisons pour trouver le plus grand sur la portion de tableau. Soit m_i ce plus grand élément. Il est rangé en mémoire commune à la position i.

Étape 2 : un des processus parmi les p processus activés, par exemple le père, exécute un algorithme de recherche du plus grand sur la mémoire partagée et donc parmi p valeurs (les m_i précédents). Le coût est donc

ici de $p-1$ comparaisons.

Au total, cet algorithme parallèle a un temps d'exécution parallème de $N/p-1+p-1$. C'est pas trop mal : le problème de taille N est traité p fois plus rapidement que si on avait un seul processeur (facteur N/p) modulo un surcoût en p (si p n'est pas du même ordre de grandeur que N, alors ce surcout est négligeable).

Passons maintenant aux discussions techniques. Premièrement, un processus peut choisir de traiter N/p valeurs consécutives dans T ou bien traiter les éléments aux positions $i, i+N/p, i+2N/p\cdots$: c'est un choix à faire par le programmeur. Nous allons implémenter le premier cas.

Deuxièmement, nous devons discuter de ce que l'on va ranger en mémoire partagée. De toute évidence il nous faut y ranger le tableau des m_i. Dans le code ci-dessous nous avons aussi choisi de ranger le tableau T de taille N. Si ce tableau n'était pas rangé en mémoire partagée, il y aurait une copie locale à chaque processus et cela peut consommer beaucoup de mémoire et de temps à la création du processus puisqu'on a vu qu'avec l'interface fork(), les variables sont copiées.

Le code que nous proposons est alors le suivant :

```
/*
 * Programme qui calcule le plus grand dans un vecteur
 * d'entiers sur p processus et en utilisant la mémore
 * partagée.
 *
 *
 * Note : on suppose que n/p est en entier
 * Note : veuillez remarquer comment se font les
 *        allocations dynamiques et l gestion de la
 *        memoire partagée
 *
 *
 * Exemple d'exécution :
 * $ ./a.out
 * Enter the number of elements in array:
 * 25
```

```
 * Enter the number p of processes:
 * Shoud verify: p>1 and n modulo p == 0:
 * 5
 * 1 4 19 12 14 6 2 17 17 24 10 13 21 1 2 14 17 1 10 2 11 18 15 24 22
 * Les plus grands 'locaux' sont' : 19 24 21 17 24
 * Le plus grand est 24
 *
 *
 */

#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#define SEED 35791246

struct region {               /* Defines "structure" of shared memory */
     int             Mycase;
};


int
main(int argc, char *argv[])
{

     int             n = 0;
     int             inter, i, j, p, max = -1;
     pid_t          *nn, wpid;
     struct region  *rptr;
     int             fd, status;

     /* The number of intervals. */
```

```
printf("Enter the number of elements in array:\n");
scanf("%d", &n);

/* The number of processes we activate. */
do {
     printf("Enter the number p of processes: \n");
     printf("Shoud verify: p>1 and n modulo p == 0: \n");
     scanf("%d", &p);
} while ((p < 2) || ((n % p) != 0));

/* Create shared memory object and set its size */

fd = open("/tmp/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
if (fd == -1) {
}
/* Handle error */ ;

if (ftruncate(fd, sizeof(struct region) * (n + p)) == -1) {
     printf("ERROR\n");
}
/* Handle error */ ;

/* Map shared memory object */
rptr = mmap(NULL, sizeof(struct region) * (n + p),
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (rptr == MAP_FAILED) {
}
/* Handle error */ ;

/* pere */
/* initialize the shared memory with random numbers */
srandom(SEED);
for (i = 0; i < n; i++) {
     (rptr + i)->Mycase = 1 + (int) ((double) n * rand() / (RAND_MAX * 1));
     printf("%d ", (rptr + i)->Mycase);
}
printf("\n");

/* Allocate an array to store th (n-1) childrens PID */
nn = (pid_t *) malloc((p) * sizeof(pid_t));
```

```
if (nn == NULL) {
     perror("Problem with allocation\n");
}
/* Now compute the maximum. */
for (i = 0; i < (p); i++) {
     nn[i] = fork();
     if (nn[i] == 0) {     /* fils */
          /* initialize random numbers */
          max = -1;
          for (j = (i * n / p); j < (i * n / p) + (n / p); j++) {
               inter = (rptr + j)->Mycase;
               //printf("%d,%d\n", inter, max);
               if (inter > max)
                    max = inter;
               (rptr + i + n)->Mycase = max;
          }
          exit(0);
     }
}

/* Suspend l'exécution du processus dans lequel il est appelé */
 * jusqu'à réception d'une valeur status d'un processus enfant
 * terminé.
*/
for (i = 0; i < (n - 1); i++)
     wpid = waitpid(nn[i], &status, WUNTRACED);

/* Le pere recompose le resultat */
/* Print the results. */
printf("Les plus grands 'locaux' sont' : ");
max = -1;
for (i = 0; i < p; i++) {
     j = (rptr + i + n)->Mycase;
     printf("%d ", j);
     if (j > max)
          max = j;
}
printf("\nLe plus grand est %d \n", max);

/* on libére la place occupée */
```

```
    free(nn);
    munmap(rptr, sizeof(struct region) * (n + p));
    close(fd);
    return 0;

}
```

3 Java et processus

3.1 Le sujet

Monsieur Khafif a beaucoup apprécié notre programme de calcul d'une approximation de pi sur 2 processus (voir le code C sous http ://www.lipn.fr/~cerin/SE/pi2.c) et il voudrait une implementation en Java. Vos intervenants de Systeme n'ont jamais fait de Java. Pour ne pas décevoir M Khafif, nous vous proposons de réaliser une implémentation Java de pi2.c

Puisque vous acceptez cette mission, commencez par rechercher un peu de documentation en ligne sur la création de processus en Java (émuler fork() et wait()). Ensuite posez-vous la question de comment réutiliser le code C de mémoire partagée dans votre programme Java.

3.2 Des éléments de réponse

Cet exercice nous renvoie à la question de croiser les supports de programmation Unix et Java pour exprimer la création d'activités concurrentes/parallèles, la synchronisation d'activités concurrentes/parallèles et le partage d'information entre activités.

Nous avons vu, dans notre cours d'Unix, que les processus crées par fork() et synchronisés par wait() ne partagent pas les variables ce qui nous a conduit à implémenter de la mémoire partagée avec les instructions mmap(), ftruncate, open(), munmap().

Pour Java, vous pouviez suivre les discussions suivantes (par ordre de diffi-

cultés croissantes) :

- http ://viennet.developpez.com/cours/java/thread/
- http ://www.larcher.com/eric/guides/javactivex/VII.htm
- http ://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/Cours05html/poly003.html

Vous allez y trouver toutes les réponses aux 3 questions précédentes. Par exemple en Java, on n'a pas besoin de primitives équivalentes à mmap() pour accéder à une variable partagée : si on veut partager l'accés à une variable, on a souvent recours à une variable de classe (par définition partagée par toutes les instances de cette classe). Il reste à régler les problèmes de conflit d'accès (surtout pour les écritures) à cette variable. Dans notre problème nous n'avons pas de conflit en écriture.

Concernant la synchronisation, on peut lire sur la page Web suivante `http ://www.devarticles.com/c/a/Java/Multithreading-in-Java/5/` : *The isAlive() method determines whether a thread is still running. If it is, the isAlive() method returns a Boolean true value ; otherwise, a Boolean false is returned. You can use the isAlive() method to examine whether a child thread continues to run. The join() method works differently than the isAlive() method. The join() method waits until the child thread terminates and joins the main thread. In addition, you can use the join() method to specify the amount of time you want to wait for a child thread to terminate.*

On voit donc une distinction nette entre isAlive qui n'est pas bloquante et join qui est bloquante. L'usage pour isAlive afin d'attendre la fin d'un processus est donc au sein d'une bloucle while : c'est ce que l'on appelle « faire de l'attente active »de manière explicite.