TD 3 - ASR7 Programmation concurrente

25 avril 2017

I Travail à la chaine

Vous devez donner un programme qui simule un travail à la chaine. Il s'agit d'organiser le travail à l'usine de Doubitchous Preskovitch S.A. à Sofia.

Le travail est décomposé en 6 étapes :

1. mélanger la farine et le cacao ;
2. pétrir la pâte avec la margarine et l'huile ;
3. ajouter le sucre et la cannelle ;
4. ajouter un morceau de chocolat ;
5. rouler le doubitchou ;
6. cuire le doubitchou .

Chaque thread est en charge de l'une d'entre elle. Le travail se fait à la chaîne c'est-à-dire que le thread en charge de la cuisson ne peut le faire tant que le doubitchou n'est pas correctement roulé par le thread qui se trouve à la position précédente dans la chaîne.

Votre programme doit lancer le nombre de threads nécessaire (NB_ETAPE) puis les faire agir à la chaîne pendant la création de nb_threads. A la fin, le programme doit afficher le temps passé à la confection (c'est à dire la somme des temps passés par chaque thread).

Pour faire ce travail, vous disposez déjà des fonctions :

— double travail(int pos, int num_tours) qui effectue le travail pour le thread à la position pos dans la chaîne, de plus cette fonction affiche un message permettant de savoir ce qu'on fait. Par exemple, pour le thread 5 durant la confection du 3e doubitchou (le numéro 2), cette fonction affichera :
`2 : le thread 5 cuit le doubitchou`
Cette fonction de plus mesure le temps nécessaire à sa tâche et le renvoie.

— Vous disposez aussi de la fonction travail_chaine qui effectue la tâche. L'algorithme utilisé par ces threads est simple, tant que le nombre voulu de doubitchou n'est pas préparé, le thread :
  — attend que le thread précédent ait fini sa préparation (fonction attend) ;
  — effectue sa tâche (fonction travail) ;
  — s'il n'est pas le dernier, il transmet son travail au suivant (fonction transmet).

```
double travail_chaine(chaine *c, int pos, int nb_tours, int nb_etape) {
  int i;
  double temps = 0;
  for (i=0; i<nb_tours; i++) {
    attend(c, pos);
    temps += travail(pos, i);
    if (pos != nb_etape-1) {
      transmet(c, pos+1);
    } else {fprintf(stdout, "le doubitchou %d est terminé\n", i);}
  }
  return temps;
}
```

Dans cette fonction, c est la structure de donnée nécessaire à la synchronisation, pos la position du thread dans la chaîne, nb_tours le nombre de tours demandés et nb_etape le nombre d'étapes nécessaires à la préparation.

Q.I.1) - Expliquez la manière dont vous allez synchroniser les différents threads : les primitives utilisées, l'algorithme des fonctions attend et transmet.

Q.I.2) - Donnez le code des fonctions d'initialisation et de destruction.

Q.I.3) - Donnez le code de la fonction exécutée par chacun des threads.

Q.I.4) - Donnez le code de la fonction principale qui lance les threads et récupère leur résultat.

Q.I.5) - Donnez le code des fonctions attend et transmet.

Pour élaborer 3 doubitchous le programme que vous fournissez doit par exemple afficher :

```
0 : le thread 0 mélange la farine et le cacao
1 : le thread 0 mélange la farine et le cacao
0 : le thread 1 pétrie la pâte avec la margarine et l'huile
2 : le thread 0 mélange la farine et le cacao
1 : le thread 1 pétrie la pâte avec la margarine et l'huile
0 : le thread 2 ajoute le sucre et la cannelle
2 : le thread 1 pétrie la pâte avec la margarine et l'huile
1 : le thread 2 ajoute le sucre et la cannelle
0 : le thread 3 ajoute un morceau de chocolat
0 : le thread 4 roule le doubitchou sous les aisselles
1 : le thread 3 ajoute un morceau de chocolat
0 : le thread 5 cuit le doubitchou
1 : le thread 4 roule le doubitchou sous les aisselles
le doubitchou 0 est terminé
2 : le thread 2 ajoute le sucre et la cannelle
2 : le thread 3 ajoute un morceau de chocolat
1 : le thread 5 cuit le doubitchou
2 : le thread 4 roule le doubitchou sous les aisselles
le doubitchou 1 est terminé
2 : le thread 5 cuit le doubitchou
le doubitchou 2 est terminé
Le thread 0 s'est terminé et a travaillé pendant 0.038278
Le thread 1 s'est terminé et a travaillé pendant 0.023248
Le thread 2 s'est terminé et a travaillé pendant 0.02927
Le thread 3 s'est terminé et a travaillé pendant 0.013315
Le thread 4 s'est terminé et a travaillé pendant 0.021332
Le thread 5 s'est terminé et a travaillé pendant 0.02937
La somme est 0.154813 secondes
```

II Choix d'ordonancement

Sur le noyau linux (après le 2.4.4) une option est apparue pour l'ordonnanceur : il s'agit de child_run_first . Cette option concerne l'ordonnancement lorsqu'un processus fait un **fork**. Comme son nom l'indique, cette option signifie qu'à la suite du **fork**, c'est le processus fils qui prend la main.

Q.II.1) - Rappelez le comportement des **fork** vis-à-vis de la mémoire et la façon dont le système le gère.

Normalement, le fils copie la mémoire du père. Cela peut être coûteux surtout pour un **fork** suivit immédiatement d'un **exec** (écrasement de la mémoire pour faire autre chose).
Le système utilise le `copy-on-write` c'est à dire qu'il ne copie que la structure de données de gestion de la mémoire, de manière à ce que les processus partagent la même mémoire physique. Mais celle-ci est marquée Read-Only. Si l'un des processus fait une modification, la page est alors réellement copiée ailleurs.

Q.II.2) - Expliquez l'intérêt de l'option child_run_first en considérant le programme suivant.

```
int pid_t pid_com;
int nb_fils = 0;
...
pid_com = fork();
if (pid_com < 0) {
  perror("Il y a eu un problème durant le fork()");
  exit(1);
}
if (pid_com == 0) {
  execvp(arg_com1[0], arg_com1);
  exit(2);
} else {
  nb_fils++;
  printf("Lancement du fils %d de pid %d\n", nb_fils, pid_com);
  ...
}
```

Le père change des données juste après le **fork** imposant un `copy-on-write` inutile puisque le fils n'en a pas besoin. Lancer le fils en premier résoud le problème.

Q.II.3) - Pourtant depuis le noyau 2.6.32 cette option est désactivée par défaut. Proposez une raison.

À cause de l'évolution des processeurs le gain apporté par cette politique n'est plus avantageux par rapport au coût :

- Gain : seulement quelques pages par le principe de localité. Le nombre de pages potentiellement changée par le père est faible puisque un processus ne reste que peu de temps dans le processeur et donc n'exécute que peu d'instructions. De plus (pp de localité) pendant un temps court, les adresses mémoire utilisées sont prochent les unes des autres (donc sur les mêmes pages mémoire)
- Coût : un changement de contexte : le fait de changer de processus n'est pas gratuit (En plus, les processeurs deviennent de plus en plus complexes et cela coute de plus en plus cher) :
  - le pipeline instruction (les instructions en préparation) est vidé or il est de plus en plus important
  - le TLB est invalidé (celui qui permet de lire une case mémoire sans avoir à lire la table des pages) or ce dernier est de plus en plus gros
  - cela stop tout un tas de petites optimisations fait à la volée par le processeur (sauvegarde du microcode, réordonancement des instructions, execution en avance ...)

Des mesures ont montré que le cout est supérieur au gain.
On peut supposer que comme le `copy-on-write` est très utilisé, il y a eu un investissement matériel pour l'améliorer. De plus, à cause de l'apparition des multi-coeurs, il y a une bonne chance pour que le fils et le père soit de toute façon exécutés immédiatement et simultanément.