TD 3 - LIFPCA Programmation Concurrente et Administration Système

—

Ordonnancement, Travail à la chaîne

Sylvain Brandel, Yves Caniou, Guillaume Damiand, Meriem Ghali, Laurent Lefèvre, Thibaut Modrzyk, Grégoire Pichon, Alec Sadler, Florence Zara, Jerry Lacmou Zeutouo

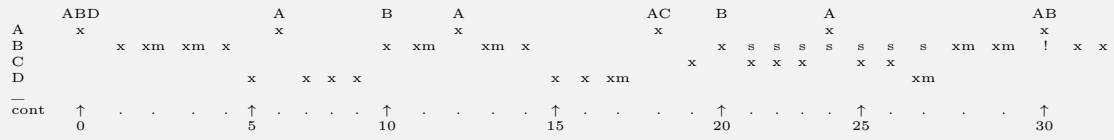
Printemps 2024

I Ordonancement avec des mutexes

Nous utilisons un ordonnancement préemptif avec priorité (Plus la valeur de priorité est importante plus la tâche est prioritaire). Nous allons utiliser un jeu de tâches qui mélange des tâches périodiques et des tâches ponctuelles. De plus, deux tâches partagent un mutex.

Tâche	Date(s) d'arrivée(s)	Priorité	Durée	Remarque
A	0, 6, 12, 18, 24, 30	10	1	Périodique
B	0, 10, 20, 30	8	4	Périodique, à chaque itération, la tâche doit acquérir le mutex à la fin du temps 1 et la libérer à la fin du temps 3.
C	18	5	6	Ponctuelle
D	0	1	8	Ponctuelle, à la fin du temps 6, la tâche acquiert le mutex et le conserve jusqu'à la fin du temps 8.

Q.I.1) - Faire l'ordonnancement de ces tâches sur 32 unités de temps.



Q.I.2) - Quel est le temps de réponse (ou latence) de chaque tâche ?

- A : 1
- B : 12 (le pire est la seconde fois). Une échéance est loupée.
- C : 9 (arrivée en 18 fini en 27)
- D : 28 (arrivée en 0 terminée en 28)

En examen, attention à bien préciser la formule utilisée, c'est-à-dire quelque chose comme Max(..,...,..).

Q.I.3) - Que remarque-t-on aux temps 21 à 27 ?

Une inversion de priorité, B est bloquée par D qui est bloquée par C. Donc C qui est moins prioritaire que B et ne partage pas de mutex avec B passe avant B.

Attention, une inversion de priorité n'est pas juste une situation où un processus de priorité faible s'exécute (ce qui arrive dès qu'un processus de priorité haute est bloqué par un mutex), c'est une situtaion à 3 processus où un processus empêche un autre, de priorité plus forte, de s'exécuter alors qu'ils ne paragent pas de mutex.
À noter qu'en examen, la question est plutôt "Que peut-on observer ?" : il peut y avoir une inversion de priorité, ou juste un thread de plus forte priorité de bloqué par un mutex ou sur un P() de sémaphore ; on peut également avoir des temps où le processeur est *idle* (aucune tâche ne s'exécute), des itérations qui ne sont pas terminées, voire même pas commencée (comme ici pour la tâche B4, qui s'exécute entre 32 et 36). Cette partie décrit le schéma réalisé (et permet d'expliquer les prises de mutex, *etc.*).

II Travail à la chaîne

Vous devez donner un programme qui simule un travail à la chaîne. 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 elles. 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 nécessaires (`NB_ETAPE`) puis les faire agir à la chaîne pendant la création de `nb_threads`. À 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 mesure aussi le temps nécessaire à sa tâche et le retourne.
- Un tableau d'entiers `nb_doubs` représente le nombre de doubitchous avant chaque étape. `nb_doubs[0]` est le nombre de doubitchous pas encore démarrés, et `nb_doubs[nb_etape]` est le nombre de doubitchous terminés.
- 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 doubitchous 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()`).
  - S'il est le dernier, il pose le doubitchou terminé sur le plat (qui correspond à la dernière case du tableau `nb_doubs`), en appelant la fonction `termine()`.

```
1   double travail_chaine(chaine *c, int pos, int nb_tours, int nb_etape) {
2     int i;
3     double temps = 0;
4     for (i=0; i<nb_tours; i++) {
5       c->attend(pos);
6       temps += travail(pos, i);
7       if (pos != nb_etape-1) {
8         c->transmet(pos+1);
9       } else {
10        c->termine();
11        fprintf(stdout, "doub %d : le doubitchou est terminé\n", i);
```

```
12          }
13        }
14        return temps;
15      }
```

Dans cette fonction, `c` est la structure de donnée nécessaire à la synchronisation (le moniteur), `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.II.1) - Expliquez la manière dont vous allez synchroniser les différents threads : les primitives utilisées, l'algorithme des fonctions `attend()` et `transmet()`.

On généralise le principe du producteur-consommateur : chaque thread sera consommateur de l'étape précédente (sauf le premier), et producteur de l'étape suivante (sauf le dernier bien sûr).

Idéalement on affecte un CPU par thread, chaque CPU fait le calcul pour une étape et on a un parallélisme efficace (dans la vraie vie, beaucoup de traitements se font de cette manière, les exemples classiques sont les algorithmes de traitement d'image ou de son qui sont naturellement pipelinées).

Le corrigé complet se trouve dans `prog_thread_chaine/chaine.cpp`, quelques extraits ci-dessous.

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

Rien de très spécial. On initialise la pile de doubitchous en entrée du premier thread avec le nombre total de doubitchous à confectionner. Ici, on ne fait que compter le nombre de doubitchous. Vous remarquez que la solution ne propose d'utiliser qu'un mutex : normal, limitant ? Par ailleurs, un shema producteur-consommateur plus réaliste devrait stocker les données dans une FIFO (cf. TD précédent).

```
1   class chaine {
2   private:
3     unsigned nb_etapes;
4     vector<int> nb_doubs; // Nombre de doubitchous disponibles en entrée de l'étape N
5     mutex m;
6     vector<condition_variable_any> t_cond;
7
8   public:
9     chaine(int nb_etapes, int nb_doubichous);
10    void affiche();
11    void attend(int pos);
12    void transmet(int pos);
13    void termine();
14  };
15
16
17  chaine::chaine(int nb_etapes, int nb_doubichous) :
18    nb_etapes(nb_etapes),
19    nb_doubs (nb_etapes+1, 0),
20    m(),
21    t_cond(nb_etapes)
22  {
23    nb_doubs[0] = nb_doubichous;
24  }
```

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

L'essentiel est donné ci-dessus. Il reste à récupérer le temps renvoyé par travail_chaine et à l'affecter à un paramètre passé par référence (car on ne peut pas récupérer la valeur de retour d'un `std::thread`) :

```
1   void fct_thread(chaine *c, int pos, int nb_tours, int nb_etape, double *temps) {
```

```
2       *temps = travail_chaine(c, pos, nb_tours, nb_etape);
3     }
```

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

```
1       chaine c(NB_ETAPES, nb_tours);
2
3       vector<thread> ths;
4       double tab_temps[nb_threads];
5
6       c.affiche();
7
8       for (i=0; i<nb_threads; i++) {
9         ths.push_back(thread(fct_thread, &c, i, nb_tours, NB_ETAPES, &tab_temps[i]));
10      }
11
12      c.affiche();
13
14      for (i=0; i<nb_threads; i++) {
15        ths[i].join();
16      }
17
18      c.affiche();
19
20      double somme = 0;
21      for (i=0; i<NB_ETAPES; i++) {
22        somme += tab_temps[i];
23      }
24      cout << "Le temps total est de " << somme << " secondes"<< endl;
```

La fonction `affiche()` n'est utile que pour le débug.

Q.II.5) - Donnez le code des fonctions `attend()` et `transmet()`.

```
1     void
2     chaine::attend(int pos) {
3       m.lock();
4
5       while(nb_doubs[pos]==0) {
6         t_cond[pos].wait(m);
7       }
8
9       nb_doubs[pos]--;
10
11      m.unlock();
12    }
13
14    void
15    chaine::transmet(int pos) {
16      m.lock();
17
18      nb_doubs[pos]++;
19      t_cond[pos].notify_one();
20
21      m.unlock();
22    }
```

Q.II.6) - Donnez le code de la fonction `termine()`.

```
1     void
2     chaine::termine() {
3       // Pas besoin de mutex ici
4       nb_doubs[nb_etapes]++;
5     }
```

On n'a pas besoin de mutex ici car seul le dernier thread va accéder à cette case du tableau. Pour les autres cases, le mutex était indispensable car on peut avoir un `nb_doubs[pos]++;` et un `nb_doubs[pos]--;` en parallèle.

On peut aussi utiliser un mutex dans `termine()` pour se simplifier la vie et appliquer « bêtement » le principe du moniteur de Hoare. C'est correct, mais un peu moins efficace en performance.

On aurait besoin d'un mutex ici dans le cas où plusieurs threads se partageraient le travail sur la dernière étape.

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
[...]
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
[...]
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
[...]
La somme est 0.154813 secondes
```