UNIVERSITÉ PARIS SUD
IUT D'ORSAY

DUT Informatique
M3101 Système S3
2015 / 2016

Travaux Pratiques n° 4 : Threads

Nom(s) :
Groupe :
Date :

Objectifs : apprendre à créer, travailler avec et arrêter des threads (ou processus légers). Savoir reconnaître les données partagées entre différents threads. Être capable d'orchestrer la synchronisation de threads au moyen des primitives de terminaison et d'exclusion mutuelle.

Nous souhaitons implémenter l'algorithme classique de tri à bulle avec une particularité : le tableau à trier, `tab`, est parcouru en permanence par deux threads : « bulle » qui procède `tab` de bas en haut, et « plomb » qui le procède de haut en bas.

Le squelette du programme est donné dans la Figure 1 et déposé dans l'espace du cours sur Dokeos. Initialement, le tableau `tab` est initialisé par les nombres de 1 à `SIZE`, dans l'ordre croissant. Ensuite, le programme répète plusieurs fois le même procédé : le tableau est permuté d'une façon aléatoire avec la fonction `shuffle()` ; puis deux threads, `bulle` et `plomb`, sont générés. Le thread principal attend la fin de deux threads avant de commencer l'itération suivante.

N.B. : Notez l'algorithme de permutation du tableau implémenté dans la fonction `shuffle()`. Cet algorithme, connu sous les noms de *Fisher-Yates shuffle* et *Knuth shuffle* est une méthode à la fois simple et efficace (de complexité linéaire) de permuter une suite de nombres de manière non-biaisée (c'est-à-dire que toute permutation se réalise avec la même probabilité).

Attention : tout programme doit être compilé avec `gcc -Wall -pthread -o prog prog.c`

Exercice 1. Implémentez la procédure de tri à bulle dans les fonctions `bulle()` et `plomb()` sans utiliser de mesures d'exclusion mutuelle. Rappelons que `bulle()` doit parcourir le tableau de 0 vers (`SIZE`−1), et `plomb()` de (`SIZE`−1) vers 0. N'oubliez pas d'arrêter la boucle principale quand le tableau est déjà en bonne ordre (c.-à-d. aucun échange n'est fait au dernier passage).

Pouvez-vous prédire le résultat de votre programme ? Faites plusieurs lancements. Si vous travaillez sur une machine uni-processeur, ajoutez les instructions `sched_yield()` entre les opérations qui effectuent l'échange des éléments du tableau (pour forcer l'ordonnanceur à changer la main) et refaites l'expérience.

Que constatez-vous ? Expliquez le comportement du programme.

```c
#define SIZE 20
int tab[SIZE];

void * bulle(void * arg) { .., }
void * plomb(void * arg) { ... }

void shuffle() {
  int i, j, temp;
  for (i = SIZE - 1; i > 0; i--) {
    j = random() % (i + 1);
    temp = tab[i];
    tab[i] = tab[j];
    tab[j] = temp;
  }
}

int main() {
  int i;
  pthread_t bulle_id, plomb_id;
  srandom(time(NULL));

  for (i = 0; i < SIZE; i++) tab[i] = i + 1;

  for (i = 0; i < 3000; i++) {
    shuffle();
    if (pthread_create(&bulle_id, NULL, bulle, NULL) != 0) exit(1);
    if (pthread_create(&plomb_id, NULL, plomb, NULL) != 0) exit(1);
    pthread_join(bulle_id, NULL);
    pthread_join(plomb_id, NULL);
  }

  for (i = 0; i < SIZE; i++) printf("%d ", tab[i]);
  putchar('\n');
  return 0;
}
```

FIGURE 1 – Tri à bulle parallélisé (schéma).

Identifiez la ressource partagée (ou les ressources partagées) et les sections critiques dans le code.

Exercice 2. Essayons de régler le problème en protégeant les sections critiques (telles que vous les avez identifiées) par verrouillage et déverrouillage d'un mutex global :

```c
pthread_mutex_lock(&mutex);
/* section critique */
pthread_mutex_unlock(&mutex);
```

Est-ce que cette solution est correcte ? Est-ce qu'elle est acceptable ?

Dans la méthode de tri à bulle on compare toute paire de cases voisines, mais on ne fait l'échange que quand les valeurs sont dans le mauvais ordre, donc moins souvent en général. Pourrait-on améliorer la solution précédente (avec un mutex global) en faisant la comparaison en dehors de la section critique, c'est-à-dire avant la prise de verrou, et ne protéger que l'échange de valeurs ?

Est-ce que cette solution est correcte ? Est-ce qu'elle est acceptable ?

Exercice 3. Entre Bob. « Tant que nos `bulle()` et `plomb()` comparent et échangent les éléments aux deux extrémités du tableau, il n'y pas d'interférences », dit il. Selon Bob, ce n'est que *les deux cases impliquées dans la comparaison-échange* qu'il faut protéger à chaque itération.

Comment peut-on protéger l'accès à une case particulière du tableau ?

L'ordre dans lequel on verrouille les cases, est-il important ? Comparez les schémas de verrouillage dans `bulle()`, `plomb_left()` et `plomb_right()` dans la Figure 2. Pouvez-vous prédire le comportement de votre programme dans le cas où on utilise `bulle()` avec `plomb_left()` ? avec `plomb_right()` ? Testez les deux implémentations et expliquez les résultats.

Joignez le code source de l'implémentation correcte à votre compte-rendu.

```
void * bulle(void * arg) {
  ...
      pthread_mutex_lock( /* i-ème case */ );
      pthread_mutex_lock( /* (i+1)-ème case */ );
      /* comparaison et échange */
      pthread_mutex_unlock( /* (i+1)-ème case */ );
      pthread_mutex_unlock( /* i-ème case */ );
  ...
}

void * plomb_left(void * arg) {
  ...
      pthread_mutex_lock( /* i-ème case */ );
      pthread_mutex_lock( /* (i-1)-ème case */ );
      /* comparaison et échange */
      pthread_mutex_unlock( /* (i-1)-ème case */ );
      pthread_mutex_unlock( /* i-ème case */ );
  ...
}

void * plomb_right(void * arg) {
  ...
      pthread_mutex_lock( /* (i-1)-ème case */ );
      pthread_mutex_lock( /* i-ème case */ );
      /* comparaison et échange */
      pthread_mutex_unlock( /* i-ème case */ );
      pthread_mutex_unlock( /* (i-1)-ème case */ );
  ...
}
```

FIGURE 2 – Ordre de verrouillage.

Exercice 4. Si vous avez accès à une machine multi-processeur, ajoutez à votre programme la mesure du temps total de tri, c.-à-d. le nombre de microsecondes passées entre la création et la fin des threads `bulle` et `plomb` pendant une itération de la boucle principale dans `main()`. Indice : utilisez la primitive `gettimeofday()`.

Comparez la performance avec la version séquentielle du programme. Voyez-vous un gain ? Essayez avec `SIZE` égal à 2000 et un seul tour de permutation-tri.