Programmation des Systèmes

Processus légers

Samuel HYM, Giuseppe LIPARI, Philippe MARQUET, Jean-Stéphane VARRÉ*

Ce document est le support de travaux dirigés et de travaux pratiques relatifs aux processus légers et à leur synchronisation.

1 Création de processus légers

Exercice 1 (Arbre de calcul)
Soit le programme suivant de calcul du n^e nombre de Fibonacci :

```
int fib(int n) {
    if (n<2)
        return n;
    else {
        int x, y;
        x = fib(n-1);
        y = fib(n-2);
        return x + y;
    }
}
```

```
int main(int argc, char *argv[]) {
    int n, res;

    n = strtol(argv[1], NULL, 10);
    res = fib(n);

    printf("fib(%d)=%d\n", n, res);
    exit(EXIT_SUCCESS);
}
```

Question 1.1 On désire attacher un processus léger à chacune des instances de la fonction `fib()`. On construit ainsi un arbre de processus légers. Avant de retourner son résultat, un processus léger attend que ses deux fils aient terminé. □

Exercice 2 (Recherche parallèle)
On désire réaliser une implantation multithreadée de la recherche d'une valeur `v` dans un tableau de valeurs selon la méthode suivante :

- on délègue la recherche de la valeur dans la première moitié du tableau à un thread ;
- on recherche la valeur dans la seconde moitié du tableau (en utilisant la même méthode de recherche parallèle) ;
- on récupère le résultat de la recherche effectuée par le thread pour le combiner au résultat de notre recherche.

Soit la fonction

```
int search(int *tab, unsigned int size, int v);
```

de recherche de la valeur entière `v` dans le tableau `tab` de `size` éléments. Cette fonction retourne l'indice du tableau contenant `v` ou `-1` si `v` n'apparaît pas dans `tab`.

Question 2.1 Donnez l'implantation d'une fonction `search_wrapper()` ayant un prototype de fonction principale d'un thread en vous appuyant sur l'exemple `adaptation` du cours. Cette fonction `search_wrapper()` réalise la recherche par un simple appel à `search()`. □

Édition de liens avec la bibliothèque `pthread`

L'option `-pthread` doit être ajoutée lors de l'édition de liens pour explicitement indiquer l'utilisation de la bibliothèque `pthread`.
On ajoutera en conséquence la ligne suivante dans le `Makefile` :

```
LDFLAGS = -pthread
```

Vous pourrez alors vérifier, en lançant `ldd` sur votre programme, qu'il sera lié dynamiquement avec la bibliothèque `pthread`.

Question 2.2 Donnez une implantation de `search()`. □

2 Exclusion mutuelle

Exercice 3 (Compteur)
Il s'agit de développer une version « MT-Safe » de la fonction suivante qui retourne un numéro unique (à cette exécution du programme) à chaque appel.

```
int unique() {
    static int count = 0; /* variable globale à visibilité locale */
    count++;
    return count;
}
```

Question 3.1 Quel problème peut survenir lors de l'utilisation de cette fonction dans un programme multithreadé ? □

Question 3.2 Quelle structure utiliser pour protéger les accès à `count` ? □

Question 3.3 Comment retourner la valeur de `count` ? □

Exercice 4 (Retour sur l'arbre de calcul)
On propose d'améliorer la solution de l'exercice 1 de calcul du n^{e} nombre de Fibonacci de la manière suivante : plutôt que de créer un processus léger alors que la valeur a déjà été produite auparavant par un autre processus léger, on va récupérer cette valeur. Pour cela, on garde les résultats intermédiaires produits dans un tableau partagé entre les threads. (Les éléments de ce tableau sont par exemple initialisés à une valeur `INCONNUE` pour dénoter que la valeur n'a pas encore été calculée.)

```
#define INCONNUE       -1
static int fibtab [];
```

Proposez un mécanisme pour assurer le partage des accès à ce tableau `fibtab` entre tous les processus légers. Donnez la nouvelle implémentation de la fonction `fib()`. □

3 Synchronisation d'activités concurrentes

Dans l'exercice suivant, l'énoncé est centré sur la synchronisation de processus légers : le code nécessaire pour créer des processus légers et y déclencher les fonctions `p1` et `p2` n'est pas donné explicitement.

Exercice 5 (Rendez-vous)
Soient les deux fonctions `p1` et `p2` suivantes. Elles se partagent deux sémaphores, `sem1` et `sem2` initialisés à 0.

```
sem_t sem1, sem2;

/* dans main, avant la création des threads */
  sem_init(&sem1, 0, 0);
  sem_init(&sem2, 0, 0);

void *p1(void *arg) {           void *p2(void *arg) {
    a(1);                           a(2);
    sem_post(&sem2);                sem_post(&sem1);
    sem_wait(&sem1);                sem_wait(&sem2);
    b(1);                           b(2);
    return NULL;                    return NULL;
}                               }
```

Question 5.1 Quelle synchronisation a-t-on imposée sur les exécutions des `a(1)`,`a(2)`,`b(1)` et `b(2)` ? □

Question 5.2 Donner le code qui impose la même synchronisation pour 3 processus légers en utilisant 3 sémaphores. □

Question 5.3 Généraliser le code précédent pour synchroniser N processus légers en utilisant N sémaphores. On généralisera pour cela les fonctions `p1`, `p2`, ... en une fonction `p` recevant en argument le numéro du processus léger. □

Question 5.4 On peut résoudre le problème pour N activités avec uniquement deux sémaphores (et un compteur). Donner le code des activités dans ce cas. □

4 Applications à des exemples concrets

L'objectif de cette section est de mettre en œuvre des threads sur quelques cas concrets. Vous trouverez sur le portail des canevas de code correspondant à ces exercices.

Exercice 6 (Calcul du taux de GC)
Étant donné un génome (c'est-à-dire un fichier contenant une liste de bases qui peuvent être A, C, G ou T), il est intéressant de calculer le taux d'apparition des bases G et C sur le nombre total de bases.

Question 6.1 Implémentez une commande prenant en argument le nom d'un fichier contenant un génome et un nombre n de threads telle que :
- charge le contenu de ce fichier en mémoire,
- découpe le calcul du nombre de G et C en n threads, chacun traitant un n-ième du contenu du fichier,
- affiche le taux une fois tous les calculs terminés.

□

Question 6.2 Expérimentez la commande précédente, en faisant varier la taille du génome à analyser et le nombre de threads pour trouver le nombre optimal de threads suivant la taille du génome.

Dans cet exercice, on supposera que le génome peut être placé entièrement en mémoire, pour éviter les conflits entre les threads dans l'accès aux entrées/sorties. □

Code initial pour le calcul du taux de GC & Expérimentation

Vous trouverez sur le portail un code calculant séquentiellement le taux de G et C. Vous pourrez ainsi vous concentrer sur l'ajout des threads à ce code. Vous pourrez aussi vérifier le résultat de votre code multithreadé.
Par ailleurs, vous trouverez aussi un générateur de génome aléatoire qui vous permettra d'expérimenter sur des génomes de toutes les tailles (plus petites que la mémoire de votre machine, vu le modèle très simple de chargement du génome...).
Vous expérimenterez en lançant `time ./compteur-gc` pour observer comment se passe le temps écoulé pendant le calcul. Vous pourrez aussi utiliser la commande `/usr/bin/time -f %e ./compteur-gc` pour n'afficher que le temps réel écoulé.
Comprenez-vous comment ont été placés les appels à `clock_gettime` dans le code initial ?
Pour mener la campagne d'expériences, vous devez faire varier à la fois la taille n des génomes et le nombre t de threads (par exemple pour $n = 10^2$ à 10^9 et $t = 2^0$ à 2^5 cela donnera 48 expérimentations). En principe, pour qu'une mesure de temps soit valable il faut, pour chaque expérience taille de génome/nombre de threads, réaliser la moyenne sur plusieurs tests.
Vous pourrez avantageusement utiliser Gnuplot pour représenter votre graphique en 3 dimensions. En admettant disposer d'un fichier `res.dat` contenant sur chaque ligne la taille du génome, le nombre de threads et le temps (séparés par des espaces), vous pourrez avec les commandes suivantes obtenir une belle courbe.

```
gnuplot> set logscale x
gnuplot> set dgrid3d 30,30
gnuplot> set pm3d
gnuplot> splot 'res.dat' using 1:2:3 with lines
```

Exercice 7 (Tri rapide multithreadé)
Le principe du tri rapide est bien connu [1]. Rappelons juste qu'à chaque étape la partie du tableau en cours de tri est découpée en deux sous-tableaux : les éléments inférieurs au pivot et ceux supérieurs au pivot. Les deux sous-tableaux sont ensuite eux-mêmes triés, de façon indépendante l'un de l'autre. Il est donc possible de charger deux threads différents d'effectuer le tri de chaque sous-tableau.

On peut aussi imaginer implémenter le tri rapide en remplaçant chaque appel récursif par la création d'un nouveau thread. Mais cette stratégie est très coûteuse en nombre de threads, donc en temps (créer un thread prend du temps, etc.) et en mémoire.

Une stratégie plus efficace est d'utiliser une pile des tâches à faire, c'est-à-dire des sous-tableaux (que nous appellerons des *blocs*) du tableau à trier. Chaque thread répète alors les opérations suivantes :

— dépiler un bloc,
— découper ce bloc en deux sous-blocs (c'est-à-dire effectuer une étape de l'algorithme)
— empiler les deux nouveaux blocs,

jusqu'à ce que le tableau soit entièrement trié.

Comment un thread peut-il savoir que le tableau est entièrement trié ?

Les threads doivent s'arrêter lorsqu'il n'y a plus de bloc à trier *et* qu'aucun autre thread n'est en train de découper un bloc. Nous utiliserons donc un compteur du nombre de threads qui sont en train de découper un bloc.

Question 7.1 Mettez en œuvre cet algorithme en :
— créant une pile et n threads dans la fonction `rapide`,
— créant une fonction adaptée à la signature attendue dans un thread qui effectue en boucle :
  — vérifie si le tri est terminé,
  — dépile un bloc dès qu'un bloc est disponible,
  — le découpe,
  — empile les deux (ou moins) sous-blocs.

1. Si besoin, vous trouverez sur wikipédia tout ce qu'il faut.

Expérimentations du tri rapide

Pour tester votre tri rapide, vous pourrez utiliser `dd if=/dev/urandom of=/tmp/alea bs=1048576 count=128` qui permet de créer un fichier `/tmp/alea` contenant 128Mo de données aléatoires. Vous choisirez des tailles de données assez grandes mais toujours plus petites (au maximum la moitié ?) que la mémoire de votre machine.
Vous testerez aussi votre algorithme sur des données déjà triées.
Quel gain de performance arrivez-vous à obtenir par rapport à l'algorithme séquentiel ?

Vous veillerez à ce que les accès aux structures de données partagées soient protégés par un ou plusieurs verrous et que les endormissements (quand la pile est vide au moment où un thread veut dépiler un bloc) et les réveils (quand de nouveaux blocs doivent être traités) soient gérés grâce à une ou plusieurs conditions.

Quand les blocs sont petits, les threads passent leur temps à se synchroniser pour dépiler et empiler des blocs dans la structure de données partagée.

Question 7.2 Modifiez votre implémentation pour que les blocs qui sont plus petits qu'une valeur seuil soient triés en utilisant l'algorithme séquentiel plutôt que l'algorithme multithreadé.

Mettez en place une série de tests pour mesurer la vitesse de votre algorithme en fonction de la taille choisie comme seuil et du nombre de threads utilisés.