

# HMIN101M TD-TP

## Processus multitâches ou *multi-threads* et synchronisation

Durée : 2 séances

### 1 Introduction

Ce TD-TP étudie le fonctionnement des processus multi-tâches et la synchronisation entre activités parallèles d'un même processus. On retrouve souvent ce fonctionnement dans la littérature sous le vocabulaire *multi-threads*. Les exercices suivants ont pour objectif d'apprendre à utiliser les outils vus en cours pour résoudre différents problèmes.

#### 1.1 La base : Parallélisme de tâches

1. Proposer une solution permettant de montrer (voir à l'écran) qu'un processus peut lancer plusieurs tâches (*threads*) en parallèle et que ces tâches se déroulent bien en parallèle.
2. Que se passe-t-il si la tâche principale se termine sans attendre la fin des autres tâches ?
3. Que se passe-t-il si une des tâches fait un appel à `exit()` ?
4. Proposer un schéma algorithmique ainsi que son implémentation permettant de constater que lorsqu'un processus crée une donnée en mémoire puis génère plusieurs tâches, toutes les tâches ont accès à cette même donnée. Pour mémoire, ceci permettra de constater un fonctionnement opposé à celui des processus enfants d'un même parent.

### 2 Exercice : produit scalaire multi-threads

On souhaite réaliser le produit scalaire (somme de produits) de deux vecteurs en parallèle (à l'aide de plusieurs threads). On suppose que si les vecteurs sont de dimension  $N$ , il y aura  $N$  threads dédiés aux multiplications et un thread dédié à la somme.

Dans la solution demandée :

- l'initialisation des deux vecteurs est à faire par le thread principal ;
- les données nécessaires au calcul des threads sont à passer en paramètre ;
- une fois les multiplications terminées, le thread d'addition pourra commencer son calcul ;
- le résultat final est à afficher par le thread principal.

1. Proposer et implémenter un premier schéma algorithmique, en supposant que le thread d'addition est le thread principal.
2. Proposer et implémenter un deuxième schéma algorithmique, en supposant cette fois, que le thread d'addition est un thread secondaire (en plus des threads de multiplication). Ce thread n'aura pas connaissance des identifiants des autres threads (il ne pourra pas attendre la fin de leur exécution).
3. (séance 2) Maintenant que le CM sur la programmation multi-threads est terminé, pensez vous avoir proposé une solution efficace ?

On suppose maintenant que la somme n'est plus effectuée par un thread séparé mais que chaque thread de multiplication ajoute son résultat à une somme globale :

4. Proposer et implémenter un schéma algorithmique respectant cette spécification.
5. (séance 2) Parmi les solutions possibles, laquelle choisiriez vous ?

### 3 Rendez-Vous

Dans un problème classique de rendez-vous entre tâches, chaque tâche lancée effectue un travail puis se synchronise (donc attend celles qui ne sont pas encore arrivées au rendez-vous) avant de continuer.

1. Peut-on résoudre simplement le problème du rendez-vous entre tâches avec seulement des outils simples d'attente de fin de tâche et/ou les verrous ? Si oui, proposer une implémentation.
2. (séance 2) Proposer une solution pour ce problème et pour  $n$  tâches, utilisant une variable conditionnelle et écrire le schéma algorithmique correspondant. Implémenter votre solution.
3. (séance 2) Comparer les différentes solutions.

### 4 Traitement synchronisé

On envisage un traitement parallèle d'une image par plusieurs activités d'un même processus, chacune ayant un rôle déterminé. Plus précisément, il s'agit d'effectuer une suite ordonnée de traitements d'image, chaque traitement est implémenté par une activité et chaque activité peut travailler sur un sous ensemble de points (pixels) de l'image, appelée *zone*, en parallèle avec un autre traitement.

En supposant que l'image est une suite de *zones* ordonnées, les traitements doivent se faire :

- avec garantie d'exclusivité : une activité ne doit pas accéder à une zone en cours de traitement par une autre activité,
- en respectant un ordre bien déterminé entre les activités : sur toute zone, l'activité  $T_1$  doit passer en premier, puis  $T_2$  etc.

On se limite d'abord à deux activités.

1. Proposer une solution permettant un fonctionnement correct pour deux activités (et sans variables conditionnelles).

On passe maintenant à trois activités (et plus) et on suppose que chaque activité traite les zones dans leur ordre successif (1, 2, 3, ...). Considérons la solution suivante :

À chaque activité  $T_i$  est associée une donnée **commune**  $d_i$ , telle que  $d_i$  contient le numéro de zone en cours de traitement par  $T_i$ .  $T_{i+1}$  peut savoir, en consultant  $d_i$  si elle peut traiter la zone à laquelle elle veut passer.

**Exemple :** si l'activité  $T_2$  est en train de traiter la zone  $Z_5$ ,  $d_2$  contient la valeur 5, positionnée par  $T_2$  avant de commencer le traitement de  $Z_5$ .  $T_3$  ne pourra travailler sur  $Z_5$  que lorsque  $d_2$  sera positionnée à  $> 5$ .

Conséquence : il y a un « espace commun » à toutes les activités (tableau commun  $d[n]$ ), contenant autant d'éléments que d'activités traitant l'image. Chaque activité peut consulter ces données mais ne peut modifier que la donnée correspondant à son rang (chaque  $T_k$  peut modifier seulement  $d_k$ ).

2. On n'utilise qu'un seul verrou, protégeant l'accès au tableau. Écrire le schéma algorithmique d'une activité. Montrer que cette solution fonctionne correctement (répond à la contrainte d'exclusivité et d'ordre). Cette solution est-elle efficace ?
3. Si ce n'est pas déjà fait, préciser comment se passe le démarrage et la fin, en indiquant aussi ce que fait le thread principal.
4. (séance 2) On souhaite améliorer le fonctionnement en le rendant plus efficace, par l'utilisation d'une ou de plusieurs variables conditionnelles. Proposer une solution. Montrer en quoi elle est plus efficace.
5. (séances 1 et 2) Implémenter progressivement vos solutions, en simulant un temps de travail aléatoire pour chaque activité travaillant sur l'image.