Processus multitâches ou *multi-threads*

1 Introduction

Ce TP étudie le fonctionnement des processus multi-tâches et la synchronisation entre activités (threads) 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

Téléchargez les fichiers sources fournis sur Moodle.

1. Compléter le squelette pour que le programme principal lance plusieurs tâches s'exécutant en parallèle et pour montrer (voir à l'écran) 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 ? Compléter le code pour éviter cette situation.
3. Que se passe-t-il si une des tâches fait un appel à `exit()` ?
4. Compléter l'implémentation de sorte à constater que lorsqu'un processus crée une donnée en mémoire puis génère plusieurs tâches, toutes les tâches ont accès à cette même donnée.
5. Y a t-il un problème de synchronisation à résoudre lors de la précédente question ?

2 Rendez-Vous

Dans un problème classique de rendez-vous entre tâches, chaque tâche lancée effectue un premier travail puis attend que toutes les autres aient aussi terminé leur premier travail (point de rendez vous) avant de poursuivre leur exécution pour effectuer un second travail.

1. Quel paradigme de synchronisation utiliserez vous pour mettre en place ce rendez-vous : exclusion mutuelle ou attente d'un événement ?
2. Proposer une solution pour ce problème et pour n tâches sous forme de schéma algorithmique puis implémenter votre solution.

3 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 impose aussi le fait que chaque activité traite les zones dans leur ordre successif (1, 2, 3, ...)

1. Proposer une architecture (en terme de hiérarchie de threads, le rôle de chacun et les données partagées) de l'application et une solution algorithmique permettant un fonctionnement correct.

   Pour vous aider, on considère la solution suivante (d'autres solutions sont possibles) :
   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 juste avant de commencer le traitement de Z_5. T_3 ne pourra travailler sur Z_5 que lorsque d_2 sera positionnée à une valeur > 5.
   Dans cette solution, 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. Implémenter votre solution, en simulant un temps de travail aléatoire pour chaque activité travaillant sur l'image.