HMIN101M - TD/TP 3

Processus multitâches ou *multi-threads* et synchronisation

1 Introduction

Ce TD étudie le fonctionnement des processus multi-tâches et comment gérer des activités parallèles d'un même processus. On retrouve souvent ce fonctionnement dans la littérature sous le vocabulaire *multi-threads*. Après quelques exercices de base (section 1.1), des exercices proposés dans les TD précédents sont repris pour proposer des solutions différentes. On trouvera un rappel du problème puis des questions permettant d'étudier les caractéristiques des solutions.

1.1 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 ?
3. Que se passe-t-il si une des tâches fait un appel à `exit()` ?
4. Proposer un schéma algoritmique ainsi que son implémentation permettant de constater que lorsqu'un processus crée un objet en mémoire (à vous de voir où exactement) puis génère plusieurs tâches, toutes les tâches ont accès à ce même objet. Pour mémoire, ceci permettra de constater que ce fonctionnement est opposé à celui des processus enfants d'un même parent.

2 Allocation de matricules et mémoire partagée

On veut mettre en place un système d'allocation de matricules par plusieurs tâches (*threads*) d'un même processus. Par exemple, un système d'immatriculation de véhicules à partir de plusieurs guichets, chaque guichet étant lié à une tâche/un thread.

Le processus contient dans ses données propres le prochain matricule à attribuer : une donnée qu'il initialise. Son rôle ensuite consiste à créer les tâches. Chaque tâche récupère la valeur courante du matricule, l'attribue à l'opération d'immatriculation qu'elle est en train de faire, et procède l'incrémentation du matricule.

Un matricule est structuré comme une structure de donnée « à la française », comportant trois chiffres incrémentables, trois lettres incrémentables et deux chiffres fixes.

1. Étudier les éléments partagés entre les tâches et montrer qu'il est nécessaire de mettre en place une exclusion mutuelle entre les tâches.
2. Proposer une solution utilisant un verrou et le schéma algorithmique du processus et ses tâches. Quel rôle faut-il attribuer à la tâche principale ?
3. Est-ce qu'une tâche peut disparaître ? Que se passe-t-il dans ce cas ?
4. Comparer cette solution avec une solution utilisant un segment de mémoire partagée (donc, un processus par guichet). Il n'est pas demandé de réaliser une implémentation.
5. Réaliser l'application, en générant un temps de travail artificiel suffisamment long par tâche (de sorte à pouvoir montrer que la protection mise en place fonctionne).

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 les verrous ?
2. Proposer une solution pour ce problème et pour n tâches, utilisant une variable conditionnelle et écrire le schéma algorithmique correspondant.
3. Implémenter votre solution.

4 Traitement synchronisé

On envisage à nouveau le traitement parallèle d'une image cette fois-ci par plusieurs activités d'un même processus, chacune ayant un rôle déterminé. On rappelle que chaque tâche travaille sur un ensemble de points (pixels) de l'image appelée *zone*. On peut donc considérer l'image comme une suite de *zones* ordonnées.

Le travail doit se faire :

- avec garantie d'exclusivité : une tâche ne doit pas accéder à une zone en cours de traitement par une autre tâche,
- dans un ordre déterminé entre les taches : sur toute zone, la tâche T_1 doit passer en premier, puis T_2 etc.

On se limite d'abord à deux tâches.

1. Proposer une solution permettant un fonctionnement correct pour deux tâches (et sans variables conditionnelles), sachant que l'image est stockée dans l'espace d'adressage du processus générateur.

   Nous passons maintenant à trois tâches (et plus) et supposons que chaque tâche traite les zones dans leur ordre successif (1, 2, 3, ...). Considérons la solution suivante :

   À chaque tâche 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 : la tâche T_2 est en train de traiter la zone Z_5. d_2 contient la valeur 5, mise par T_2 avant de commencer le traitement de Z_5. T_3 ne pourra travailler sur Z_5 que lorsque d_2 sera > 5.

   Conséquence : il y a un « espace commun » à toutes les tâches (tableau commun $d[n]$), contenant autant d'éléments que de tâches traitant l'image. Chaque tâche peut consulter ces données et ne modifie que la donnée correspondant à son rang (chaque T_k modifie seulement d_k).
2. On n'utilise qu'un seul verrou, protégeant l'accès au tableau. Écrire le schéma algorithmique d'une tâche. Montrer que cette solution fonctionne correctement (répond à la contrainte d'exclusivité et d'ordre) et profiter pour décider de l'initialisation. Illustrer au moins une situation montrant que cette solution est inefficace.
3. Si ce n'est déjà fait, préciser comment se passent le démarrage et la fin, en indiquant aussi ce que fait la tâche principale.
4. On veut améliorer le fonctionnement en le rendant plus efficace, par l'utilisation d'une ou plusieurs variables conditionnelles. Proposer une solution. Montrer en quoi elle est plus efficace.
5. Implémenter progressivement vos solutions, en simulant un temps de travail aléatoire pour chaque tâche accédant à l'image.