Processus et mutex

Partie 1 – Optimisation de calculs avec plusieurs threads

Matériel à disposition. Pour cette partie, on vous fournit :

— Un fichier `template.c` en langage C qui montre comment lancer trois threads et initialiser, prendre et libérer un mutex ;
— deux fichiers `boite_noire1.h` et `boite_noire2.h` qui sont des interfaces pour des bibliothèques qui correspondent respectivement au premier et au second exercice ;
— deux fichiers `boite_noire1.o` et `boite_noire2.o` qui sont des fichiers compilés correspondant aux interfaces décrites ci-haut et dont le code source n'est pas fourni.

Note des concepteurs du sujet : Le jour de l'oral, les fichiers .o étaient fournis mais pas les fichiers .c. Comme le .o à fournir dépend de la machine, le jury fournit les fichiers .c plutôt que les fichiers .o mais l'objectif n'est PAS que les élèves lisent les fichiers .c ou adaptent leur solution en fonction de ces fichiers.

Partage de variables entre threads. Pour exercices de cette partie, il est possible d'accéder à des données (par exemples des variables globales ou tableaux) depuis plusieurs threads. Il n'y a pas besoin de faire attention aux données auxquelles on n'accède qu'en lecture (quand leur contenu n'est jamais modifié une fois que les threads sont lancés) mais, pour toute mémoire qui est modifiée après le lancement des threads, **tout accès** à cette variable (ou case de tableau) que ce soit en lecture **ou** en écriture, ne peut se faire que depuis un seul thread à la fois. Il est demander de garantir cette unicité d'accès à l'aide de mutex.

Note des concepteurs du sujet : Il est tout à fait possible que les solutions proposées par les élèves marchent même si elles ne vérifient pas cette contrainte. Que ce soit par chance (la race condition n'est pas déclenchée ou ne créé pas de problème), ou parce que le code est en fait correct (pour des raisons parfois compliquées qui peuvent être liées aux mutex utilisés par la bibliothèque ou aux garanties séquentielles de x86). Comme le sujet donne explicitement la consigne de ne pas faire de tels accès les examinateurs ont pu demander à des candidats de corriger leur code même quand, en pratique, il semblait fonctionner correctement.

Compilation séparée. Pour compiler vos programmes, il est conseillé d'utiliser la ligne de commande suivante (`main1.c` est le nom de votre fichier C ici et `boite_noire1.o` est le nom de la bibliothèque que l'on veut utiliser dans le premier exercice) :

```
gcc main1.c -Wall -Wextra boite_noire1.o  -lpthread
```

Note des concepteurs du sujet : Ici les fichiers C sont fournis mais le jour de l'oral les fichiers C n'étaient pas fournis il fallait donc utiliser une telle ligne de commande.

Exercice 1 : Gestion de tâches indépendantes

L'objectif de ce premier exercice est de faire un programme C qui fait un appel à la fonction `demarre_boite` puis fait `OBJECTIF` (qui ici vaut 100) appels à la fonction `boite_noire` puis un appel à la fonction `eteint_boite`. Toutes ces fonctions sont définies dans le fichier `boite_noire1.h`.

L'objectif de l'exercice est de faire le code qui fait le plus rapidement possible les 100 appels à la fonction `boite_noire` sachant que chaque appel à `boite_noire()` prend un certain temps. Votre programme peut utiliser plusieurs threads pour accélérer le temps d'exécution mais il ne doit pas utiliser plus de 10 threads (en comptant le thread principal).

Pour cette exercice uniquement, on ne connaît pas la durée que met chaque tâche mais on peut supposer qu'une tâche met autant de temps à s'exécuter quelque soit le thread qui l'exécute et quelque soit ce que les autres threads font (peu importe qu'ils travaillent sur une tâche ou non). Les différentes tâches ne prennent pas forcément toutes le même temps à s'exécuter (certaines peuvent durer 5s et d'autres 0.5s).

Pour cet exercice ainsi que pour les suivants, il est demandé de commencer par des solutions simples mais qui marchent puis d'améliorer progressivement vos solutions. On commencera donc par une version qui ne lance aucun thread supplémentaire pour s'assurer que l'on a bien compris le fonctionnement de la bibliothèque puis on pourra tester des versions qui lancent des threads avant d'écrire des solutions qui font des choses compliquées avec les threads lancés. Il est fortement recommandé de présenter aux examinateurs vos idées avant de vous lancer dans du code compliqué. Si on propose plusieurs solutions aux examinateurs, on ne supprimera pas les différentes versions mais on fera des copies (par exemple dans des fichiers `exo_1_naif.c`, `exo_1_thread.c`, `exo_1_thread_efficace.c`).

Note des concepteurs du sujet : On rappelle que les fichiers .c n'étaient pas fournis. Il n'était donc pas attendu que les candidats devinent leur fonctionnement pour en déduire la bonne stratégie à adopter. Les élèves devaient considérer la bibliothèque comme une boite noire dont on ignore le fonctionnement (qui n'est d'ailleurs pas entièrement déterministe).

Exercice 2 : Gestion de tâches avec prérequis et temps de calcul

Dans ce deuxième exercice, on utilise maintenant `boite_noire2.h` et `boite_noire2.o`. Dans cet exercice on a toujours `OBJECTIF` tâches à accomplir (ici `OBJECTIF` vaut 1000) mais cette fois les tâches ne sont plus indépendantes et l'on a de l'information sur les tâches à accomplir.

Plus précisément, on a, pour chaque tâche, une liste de ses prérequis et le temps (en microsecondes) qu'elle prend. Quand une tâche A a une tâche B pour prérequis, il faut que B soit finie avant de pouvoir lancer A. De nouveau, on veut une solution qui finisse l'ensemble des tâches le plus rapidement possible et l'on se donne comme limite de ne pas utiliser plus de deux threads (en comptant le thread principal).

Question préliminaire. Avant d'implémenter votre solution, prouver que déterminer la solution optimale pour cet exercice est un problème difficile (au sens de la NP-complétude). Pour cela, on pourra utiliser la NP-complétude du problème PARTITION. Le problème PARTITION est le suivant : étant donnés n entiers $v_1 \dots v_n$, déterminer s'il existe une partition de $1..n$ en deux ensembles S_1 et S_2 tels que $\sum_{i \in S_1} v_i = \sum_{i \in S_2} v_i$.

Utilisation de la bibliothèque. Chaque fonction prend en paramètre un entier entre 0 (inclus) et OBJECTIF (exclu) qui correspond à l'identifiant d'une tâche. L'objectif de cet exercice est de proposer des heuristiques qui permettent de résoudre toutes les tâches le plus rapidement possible avec 2 threads. Consulter le fichier `boite_noire2.h` pour une documentation sommaire de la biblothèque.

Recommandation. Comme pour l'exercice 1, on essaiera d'abord d'avoir une solution correcte avant d'avoir une solution qui essaie d'utiliser des threads ou d'être optimale. Les fonctions de la boite noire sont "thread safe" c'est à dire que vous n'avez pas besoin de prendre des mutex avant des les appeler.

Partie 2 – vérification statique des interblocages

L'objectif de cette partie est de travailler sur la vérification de programmes avec mutex. Comme la plupart des propriétés sur les programmes sont indécidables, on ne s'intéressera qu'à une version très simplifiée de nos programmes. Dans un premier temps on ne travaillera même que sur des programmes qui ne sont pas mis en parallèle (c'est à dire avec un seul thread où les mutex ne sont donc pas vraiment utiles). Nos programmes correspondront aux types sommes OCaml suivants :

```
type code =
  | Take of int
  | Release of int
  | Call of string
  | Sequence of code*code
  | IfThen of code*code

type programme = (string*code) list
```

Un terme `programme` décrit – en simplifiant – le comportement d'un programme qui utilise des mutex. Un terme de type `programme` c'est une liste $(s_1, c_1) \ldots, (s_k, c_k)$ de paires. Pour $1 \leq i \leq k$ cela signifie que la définition de la fonction dont le nom est s_i est le code c_i (correspondant au type `code`). On supposera qu'il existe une fonction `main` qui est appelée au début de chaque exécution et donc qu'il existe forcément un des s_i est la chaîne de caractères `"main"`. On supposera aussi que tous les fonctions ont des noms différents, c'est à dire que les s_i sont tous distincts. Voici maintenant la signification des termes `code` :

- le terme `Take` i correspond à une instruction qui prend le mutex numéro i (les mutex sont numérotés, il n'y a pas besoin de les créer) ; comme nous le verrons plus loin, pour les besoins de l'exercice et contrairement à ce qui arriverait avec des mutex en C ou en OCaml, on va supposer, dans un premier temps, que l'exécution du programme continue même si l'on prend un mutex qui est déjà pris ;
- le terme `Release` i correspond à une instruction qui libère le mutex numéro i ;
- le terme `IfThen(a,b)` correspond à un bloc d'instruction if-then-else dans le programme. Ici nous faisons abstraction de la condition, on pourra supposer que le choix est fait au hasard entre exécuter le code `a` ou le code `b` ;
- le terme `Sequence(a,b)` correspond à l'enchaînement séquentiel de deux blocs de code : le programme lance d'abord `a` et ensuite `b` ;
- enfin le terme `Call "s"` décrit que le programme appelle la fonction dont le nom est `"s"`. On suppose que nos termes sont bien définis et donc que la fonction nommée `"s"` existe bien dans le programme (c'est à dire que s'il y a un `Call "s"` alors un et un seul des s_i est `"s"`).

Question 1. Donner le programme associé au pseudo-code suivant :

```
fonction main() {
  a() ;
}

fonction a() {
  If (cond) {
    Take 0 ;
  } else {
    Take 0 ;
    b() ;
    Release 0
```

```
  }
}

fonction b() {
  Release 0 ;
  a() ;
  Take 0 ;
}
```

On dit qu'il y a un risque de blocage si le programme tente de prendre deux fois le même mutex sans que celui soit libéré entre les deux prises. Ce programme a-t-il une chance de bloquer (si les conditions sont défavorables) ?

Question 2. Écrire une fonction qui prend un programme et renvoie la liste de tous les numéros de mutex qui apparaissent syntaxiquement dans le programme. On ne cherchera donc pas à savoir si ces mutex peuvent en pratique être déclenchés, un mutex qui n'apparaîtrait que dans une fonction qui n'est jamais appelée devra donc être dans la liste renvoyée.

Trace d'un programme. Étant donné un programme P, la *trace* d'une exécution de P c'est la séquence des `Take i` et `Release i` que le programme fait. Par exemple si l'on considère le programme suivant :

```
fonction main() {
  If (cond) {
    Take 0 ;
  } else {
    Take 0 ;
    b() ;
    Release 0
  }
}

fonction b() {
  Take 0
}
```

En appelant `main()` le programme ira soit dans la branche then et sa trace sera juste `Take 0`, soit le programme exécutera la branche else et sa trace sera `Take 0 Take 0 Release 0`. Noter que cette deuxième trace correspond à un risque de blocage car on prend deux fois le mutex 0 mais l'on considère tout de même que c'est une trace du programme et `Release 0` fait aussi partie de cette trace (le programme ne s'arrête pas même en cas de blocage).

On rappelle que la structure if-then-else doit être considérée comme si le choix était fait au hasard et donc peut changer à chaque exécution. Ainsi, pour un programme qui utilise la structure if-then-else, il peut y avoir plusieurs exécutions différentes et donc plusieurs traces différentes. Remarquer aussi qu'une exécution d'un programme peut ne pas terminer, dans ce cas on considérera qu'il n'existe pas de trace car on ne s'intéresse qu'aux traces des exécutions qui finissent.

Question 3. Donner 5 traces différentes du programme de la question 1.

Question 4. On va vouloir écrire un algorithme qui prend un programme P et renvoie une grammaire qui produit l'ensemble des traces de P. Dans cette grammaire, les terminaux sont donc les `Take i` et les `Release i` et on demande à ce qu'il y ait au moins un symbole non-terminal pour chaque fonction (mais vous pouvez introduire d'autres symboles si vous le jugez utile).

Commencer par écrire à la main une grammaire correspondant aux traces des deux programmes exemples donnés en pseudo-code ci-haut. Notez que l'on ne s'intéresse qu'aux exécutions de programmes qui terminent (mais, de la même manière, tous les mots produits par les grammaires ont des dérivations finies).

Question 5. Proposer un type pour représenter les grammaires non contextuelles.

Question 6. Écrire une fonction qui retire d'une grammaire tous les symboles non terminaux qui n'ont pas de dérivation finie ainsi que les symboles non accessibles. La grammaire obtenue doit accepter le même langage que la grammaire d'entrée.

Question 7. Écrire une fonction qui prend un `programme` et produit la grammaire associée.

Question 8. Justifier que, pour un programme donné, il existe une exécution qui termine et rencontre un blocage si et seulement si la grammaire associée au programme contient un mot qui contient un facteur contenant deux `Take i` et pas de `Release i` pour un certain `i`.

Question 9. Écrire une fonction qui prend un `programme` et détermine si ce programme peut se bloquer tout seul.

Semantique des mutex. Pour la suite du sujet, on reviendra au fonctionnement classique des mutex où un programme attend indéfiniment s'il prend un mutex qui n'est pas disponible.

Question 10. On a deux programmes séquentiels (c'est à dire deux termes `code` uniquement composé de `Sequence`, `Take` et `Release`) mais qui partagent des mutex. Proposer et implémenter un algorithme qui détermine s'il est possible que les deux programmes lancés en parallèle se bloquent. Quelle est la complexité de votre algorithme (on supposera qu'il n'y a qu'un petit nombre de mutex) ?

Question 11. On considère deux programmes : un séquentiel et un non séquentiel. Proposer un algorithme qui détermine s'il y a une chance que ces deux programmes se bloquent (si les conditions sont défavorables).