Systèmes Concurrents

1h30, documents autorisés

8 décembre 2023

Les réponses doivent être justifiées.

1 Programmation concurrente (5 points)

1. Dans le TP sur le problème des lecteurs-rédacteurs réalisé avec un moniteur, les stratégies priorité aux lecteurs / priorité aux rédacteurs induisent une famine possible des activités complémentaires. Expliquer en quelques phrases une stratégie équitable (= sans famine) qui ne soit pas la stratégie FIFO.

   Alterner prio lecteurs / prio rédacteurs. Quand un lecteur sort, on passe en prio rédacteurs ; aucun nouveau lecteur ne peut entrer s'il y a des rédacteurs en attente et un de ceux-ci finira par pouvoir entrer. Quand le rédacteur sort, on passe en prio lecteurs ; tous les lecteurs en attente peuvent entrer ainsi que des lecteurs supplémentaires tant qu'aucun lecteur n'est sorti. Cette solution favorise le parallélisme en autorisant les lecteurs en masse mais garantit l'absence de famine (que ce soit des lecteurs ou rédacteurs).

   On peut imaginer des variantes prenant en compte le nombre de lecteurs / de rédacteurs qui sont passés pour faire la bascule des priorités. Attention néanmoins à ne pas pénaliser une classe car l'autre classe n'atteindrait que lentement son seuil.

2. Pour quel type de problèmes le schéma fork/join est-il adapté ?

   Problèmes irréguliers, de grande taille, se décomposant récursivement en sous-problèmes indépendants.

3. On considère un serveur multi-processeurs qui peut recevoir des requêtes de deux types : des requêtes de calcul séquentiel qui utilisent intensivement le CPU ; des requêtes qui nécessitent essentiellement des entrées-sorties et consomment peu de CPU (par exemple accès à une base de données). Une entrée-sortie induit un long blocage de l'activité qui la demande, le temps que le périphérique réponde.

   3.1. Pour maximiser l'utilisation des ressources du serveur, faut-il un ou plusieurs pools d'activités, et pourquoi ?

   3.2. Pour chaque pool identifié, quelles caractéristiques a-t-il (fixe ou dynamique ? avec priorité ? si fixe, combien d'activités ? si dynamique, borné ? etc)

   3.3. Si on utilise plusieurs pools, est-il intéressant de ségréguer les différents pools, en leur réservant statiquement une partie des ressources du serveur (par exemple certains CPU ou une partie de la mémoire) ?

   (a) 2 pools, un pour les requêtes CPU, un pour les requêtes utilisant des E/S. Une activité qui sert une requête en attente d'une E/S est bloquée, il ne faudrait pas

qu'une requête ne demandant que du CPU soit empêchée de s'exécuter car toutes les activités sont déjà affectées et bloquées sur des E/S.

*(b) Pool CPU = fixe, de taille ≈ nombre de CPU (il ne sert à rien d'en mettre plus). Pool E/S = dynamique, éventuellement borné par ≈ nombre de CPU *10 ou *100 (selon la durée des E/S) pour ne pas avoir trop d'activités bloquées.*

(c) Pas vraiment, si on ne connaît rien sur l'arrivée des requêtes. Si on a des infos sur le taux / charge des requêtes, éventuellement. Par exemple, si on sait qu'il y a un flux faible mais régulier de requêtes CPU, il peut être intéressant que le pool pour ces requêtes ait quelques CPU attribués exclusivement à lui. Une autre situation où il peut être intéressant de réserver une partie des ressources à chaque pool est pour éviter la famine s'il y a de brusques changements de charge, avec arrivée soudaine de plus de requêtes d'une catégorie que ne peut en gérer le serveur : la ségrégation assurera que l'autre catégorie continue à être servie.

2 Problème (15 points)

On considère une pâtisserie où il y a des pâtissiers en nombre indéterminé (au moins un) et un unique apprenti. La pâtisserie possède quatre zones :

- une table de préparation des gâteaux limitée à 4 places (pas plus de 4 pâtissiers peuvent l'utiliser concurremment) ;
- un four de cuisson de capacité 2 ;
- une armoire de refroidissement où le gâteau cuit repose avant de pouvoir être vendu, de capacité 10 ;
- une vitrine de vente, de capacité non bornée (en réalité les gâteaux sont si bons qu'ils sont trop vite achetés pour s'accumuler).

Le « code » d'un pâtissier est le suivant (en romain les actions de synchronisation, en italique les actions applicatives dont on ne parlera plus ensuite) :

```
boucle
    1. accéder à la table de préparation
    préparer le gâteau
    2. mettre le gâteau au four
    patienter en attendant la cuisson
    3. retirer le gâteau du four
    4. déposer le gâteau dans l'armoire de refroidissement
    boire un verre d'eau
finboucle
```

Le « code » de l'apprenti consiste à prendre successivement 4 gâteaux de l'armoire de refroidissement pour les mettre en vitrine, puis à nettoyer la table de préparation. Aucun pâtissier ne doit utiliser la table pendant que l'apprenti la nettoie. Vu que la vitrine n'a pas de contrainte, il est inutile d'en parler.

```
boucle
    5. prendre un gâteau de l'armoire de refroidissement (et le mettre dans la vitrine)
    5. prendre un gâteau de l'armoire de refroidissement (          "          )
    5. prendre un gâteau de l'armoire de refroidissement (          "          )
```

```
    5. prendre un gâteau de l'armoire de refroidissement (          "          )
    6. débuter le nettoyage de la table de préparation
    nettoyer soigneusement la table
    7. terminer le nettoyage de la table de préparation
finboucle
```

2.1 Questions générales (2 points)

1. Les règles d'utilisation de la table de préparation (plusieurs pâtissiers en nombre borné ou l'apprenti seul) correspondent à un schéma de synchronisation étudié en cours, lequel ?
   Réponse exacte : lecteurs-rédacteurs avec nb de lecteurs simultanés borné
   Réponse approximative : exclusion mutuelle apprenti - pâtissiers
2. Les règles d'utilisation de l'armoire de refroidissement correspondent à un schéma de synchronisation étudié en cours, lequel ?
   producteur-consommateur

2.2 Synchronisation par moniteur (8 points)

Développer un moniteur qui réalise les sept opérations de l'interface de la pâtisserie en respectant la méthodologie :

1. Déterminer l'interface du moniteur ;
2. Énoncer informellement les prédicats d'acceptation de chaque opération ;
3. Déduire les variables d'état qui permettent d'écrire ces prédicats d'acceptation ;
4. Formuler l'invariant du moniteur et les prédicats d'acceptation ;
5. Associer à chaque prédicat d'acceptation une variable condition qui permettra d'attendre/signaler la validité du prédicat ;
6. Programmer les opérations, en suivant le schéma présenté en cours ; 2 pt
7. Vérifier que le moniteur est correct (= montrer que les appels à `signal` sont corrects).

1. c'est le sujet !

2. c'est dans l'énoncé

3.
```
int table := 4     // nb places libres
int four := 2      // idem
int armoire := 10 // idem
bool nettoyageEnCours
```

4.
1. $table > 0$ *and* $\neg nettoyageEnCours$
2. $four > 0$
3. $\top$
4. $armoire > 0$
5. $armoire < 10$
6. $table = 4$
7. $\top$

invariant $\neg(nettoyageEnCours \wedge table < 4) \wedge table \in 0..4 \wedge four \in 0..2 \wedge armoire \in 0..10$

5. *Variables conditions : AccèsTable, AccèsFour, ArmoireNonVide, ArmoireNonPleine, Nettoyage*
6. 1. `si ¬(table > 0 and ¬nettoyageEnCours) alors AccèsTable.wait finsi`
   `table--`
   `si table > 0 alors AccèsTable.signal  // réveil en chaîne`
   2. `si ¬(four > 0) alors AccèsFour.wait finsi`
   `four--`
   `table++`
   `si table = N alors Nettoyage.signal`
   3. `four++`
   `AccèsFour.signal`
   4. `si ¬(armoire > 0) alors ArmoireNonPleine.wait finsi`
   `armoire--`
   `ArmoireNonVide.signal`
   5. `si ¬(armoire < 10) alors ArmoireNonVide.wait finsi`
   `armoire++`
   `ArmoireNonPleine.signal`
   6. `si ¬(table = 4) alors Nettoyage.wait finsi`
   `nettoyageEnCours := true`
   7. `nettoyageEnCours := false`
   `AccèsTable.signal`
7. *TODO*

2.3 Processus communicants (5 points)

L'objectif est de résoudre le problème en utilisant des canaux. Il faut trois activités de synchronisation : la table, le four et l'armoire. La table peut recevoir des demandes d'utilisation et de nettoyage ; le four peut recevoir des demandes d'enfournage et de défournage ; l'armoire peut recevoir des demandes de dépôt et de retrait.

Le code d'un pâtissier est le suivant (on utilise `_` pour indiquer que la valeur est sans importance et comme précédemment, le texte en italique correspond à des actions applicatives sans intérêt pour la synchronisation) :

Version Go

```
for {
    DemanderTable <- _
    préparer le gâteau
    Enfourner <- _
    LibérerTable <- _
    attendre la cuisson
    Défourner <- _
    Déposer <- _
    boire un verre
}
```

Version CSP

```
*[
    DemanderTable!_
    préparer le gâteau
    Enfourner!_
    LibérerTable!_
    attendre la cuisson
    Défourner!_
    Déposer!_
    boire un verre
]
```

Questions Répondre en utilisant du pseudo-Go, du pseudo-CSP ou la syntaxe Java-CSP du TP. La syntaxe précise est sans importance du moment que l'intention est claire.

1. Donner le code de l'apprenti. 1,5 pt
2. Donner le code pour le four. 1 pt
3. Donner le code pour l'armoire. 1 pt
4. Donner le code pour la table de préparation. 1,5 pt

1. Code de l'apprenti :

Version Go

```
for {
    Retirer <- _ (et mettre en vitrine)
    Retirer <- _
    Retirer <- _
    Retirer <- _
    DébuterNettoyage <- _
    nettoyer la table
    TerminerNettoyage <- _
}
```

Version CSP

```
*[
    Retirer!_ (et mettre en vitrine)
    Retirer!_
    Retirer!_
    Retirer!_
    DébuterNettoyage!_
    nettoyer la table
    TerminerNettoyage!_
]
```

2. Le corrigé n'est fait qu'en Go, versions similaires en CSP ou Java.

```
Four : for {
          select {
          case <- when(four > 0, Enfourner) :
              four--
          case <- Défourner :
              four++
          }
       }
```

3.
```
Armoire : for {
              select {
              case <- when(armoire > 0, Déposer) :
                  armoire--
              case <- when(armoire < 10, Retirer) :
                  armoire++
              }
          }
```

4.
```
Table : for {
          select {
          case <- when(table > 0 ∧ ¬nettoyage, DemanderUtilisation) :
              table--
          case <- TerminerUtilisation :
              table++
          case <- when(table = 4, DébuterNettoyage) :
              nettoyage := true
```

```
  case <- TerminerNettoyage :
      nettoyage := true
  }
}
```