Systèmes Concurrents

1h45, documents autorisés

13 décembre 2017

Les réponses doivent être justifiées. Les trois approches 2.1, 2.2 et 2.3 sont indépendantes.

1 Cours (5 pt)

1. Quelle est la différence entre famine et interblocage ?
2. On considère un système composé de 3 activités et de 4 ressources équivalentes. Chaque activité demande une ressource à la fois, mais ne peut pas demander plus de deux ressources. Montrer que ce système est nécessairement exempt d'interblocage.
3. Quelle est la différence entre l'utilisation d'une section de code en exclusion mutuelle et l'utilisation d'une transaction ?
4. Transaction : Est-ce que le contrôle de concurrence continu par estampilles restreint davantage le parallélisme que le contrôle de concurrence par certification ?
5. Transaction : dans le cas du contrôle de concurrence basé sur le verrouillage à deux phases, montrer qu'il ne peut pas y avoir d'effet domino en propagation directe (= propagation en continu).

2 Problème (15 pt)

Dans un pays voisin et à une époque guère lointaine vivaient deux familles en parfaite hostilité : les Montaigu et les Capulet. Les membres des deux familles se rendent régulièrement et individuellement en ville, depuis leur résidence située à la campagne. Pour cela, il leur faut franchir une rivière au moyen d'un bac. L'usage du bac fait l'objet d'une réglementation stricte :

— La capacité du bac est limitée à 4 passagers.

— Pour des raisons de bon emploi de l'argent public, le bac ne peut traverser qu'à plein.

— Pour des raisons de sécurité, le bac ne doit pas embarquer un membre de l'une des familles seul face à trois membres de l'autre famille. Toutes les autres combinaisons sont permises (2/2 ou 4/0 ou 0/4).

— Les actions *applicatives* possibles sur le bac sont :

— `prendre_place()` : pour qu'une personne embarque dans le bac. Comme on ne s'occupe pas des débarquements, on suppose qu'il est garanti qu'il ne reste pas de passager d'un trajet précédent quand un passager veut embarquer.

— `démarrer()` : débuter la traversée. Doit être exécutée une et une seule fois par un des passagers, et après que tous les passagers ont fini de prendre place.

— Ces opérations sont non atomiques et prennent du temps. Il est interdit d'invoquer `démarrer` pendant une invocation de `prendre_place`, et il est interdit d'invoquer `prendre_place` une fois le bac parti. Par contre, il est autorisé des invocations concurrentes de `prendre_place` (embarquements concurrents).

Les *implantations* de `prendre_place()` et `démarrer()` ne sont pas à programmer. L'objectif du problème est le contrôle de leur invocation. Il s'agit de coordonner l'activité des passagers, représentés par des activités, afin de garantir que le protocole d'usage du bac qui vient d'être présenté est bien respecté.

Note : on ne considère que la gestion des départs, dans un seul sens : les activités sont toutes sur la même berge, le bac est toujours disponible sur cette berge, et l'on ne s'intéresse pas à la gestion des arrivées. (La solution complète devrait considérer et gérer la direction de la traversée et l'état de la barque, ce qui n'est pas demandé ici).

Question (1 pt)

1. Le schéma de synchronisation présenté est proche d'un schéma présenté en cours. Lequel ? Quelles sont les différences ?

2.1 Synchronisation par sémaphores (4 pt)

Le code d'une activité passager utilisant des sémaphores a la structure ci-dessous. Attention, le lanceur n'est pas fixé a priori, c'est le code de synchronisation qui doit choisir l'une des activités pour ce rôle.

```
Activité passager
   // variables locales à l'activité
   famille : {Montaigu, Capulet} := ... // Famille de l'activité
   lanceur : booléen := false           // Sera-t-il celui qui démarre le bac ?
boucle
   // code de synchronisation à compléter (pour pouvoir embarquer et
   // éventuellement positionner lanceur)
   prendre_place();
   // code de synchronisation à compléter (pour s'assurer que tous sont
   // sont à bord et éventuellement positionner lanceur)
   si lanceur alors démarrer(); finsi;
   // code de synchronisation à compléter (pour permettre un voyage ultérieur)
   ⋮
   // débarquement et retour (hors sujet)
fin boucle
```

On dispose :

— de sémaphores, définis par une classe Sémaphore, fournissant les méthodes `up()`, `down()`, et un constructeur permettant de fixer la valeur initiale du sémaphore ;

— et de barrières, définies par une classe Barrière, fournissant la méthode `attendre()`, et un constructeur permettant de fixer la taille de la barrière (nombre d'activités à attendre avant ouverture).

Note : l'utilisation des barrières n'est ni obligatoire ni nécessaire, un schéma utilisant uniquement des sémaphores est tout aussi faisable.

Question

2. Compléter le code des activités passagers pour garantir le bon fonctionnement. Préciser les valeurs initiales des sémaphores et/ou barrières utilisées.

2.2 Synchronisation par moniteur (6 pt)

On souhaite maintenant définir un moniteur pour coordonner les activités passagers. L'interface du moniteur comporte les opérations :

— `boolean embarquer(f : {Montaigu, Capulet})` qui prend un paramètre précisant la famille du passager et retourne un booléen indiquant si le passager est celui en charge du démarrage et du pilotage du bac ;

— `void piloter(c:booléen)` ; le paramètre indique si le passager est celui en charge du pilotage.

Le code d'une activité passager utilisant le moniteur a la structure suivante :

```
Activité passager
  // variables locales à l'activité
  famille : {Montaigu, Capulet} := ... // Famille de l'activité
  lanceur : booléen := false           // Sera-t-il celui qui démarre le bac ?
boucle
  lanceur ← embarquer(famille);
  piloter(lanceur);
  ⋮
  // débarquement et retour (hors sujet)
fin boucle;
```

On suppose que les moniteurs disponibles sont des moniteurs de Hoare (exécution automatique en exclusion mutuelle des opérations, priorité au signalé, files FIFO associées aux variables condition).

Questions

— Réaliser le moniteur en suivant la démarche présentée en cours :

3. Écrire (en français) les conditions d'acceptation des opérations `embarquer` et `piloter`.
4. Définir les variables d'état permettant de représenter ces conditions.
5. Établir un invariant du moniteur liant ces variables d'état.
6. Programmer les opérations `embarquer` et `piloter` du moniteur. Préciser (sous forme de commentaire dans le code) les pré/post conditions des signaler/attendre.

— La solution développée doit garantir que seuls les 4 premiers passagers constituant un groupe valide s'embarquent (c'est-à-dire qu'aucun passager supplémentaire ne peut se substituer à l'un des membres du groupe).

7. Expliquer pourquoi c'est effectivement le cas.
8. Expliquer pourquoi cette propriété n'est pas valide si l'on considère que l'on dispose de moniteurs avec priorité au signaleur, sans file spécifique pour les signalés.

2.3 Synchronisation par processus communicants (4 pt)

Répondre au choix à l'UNE des approches (Ada 2.3.1 ou canaux 2.3.2).

2.3.1 Synchronisation par tâches Ada

Chaque passager est une tâche. Elles sont coordonnées via une tâche serveur. Cette tâche serveur a pour interface :

```
task Bac is
    entry embarquerCapulet ();
    entry embarquerMontaigu ();
    entry autoriserDemarrage (lanceur : out boolean);
end Bac;
```

Le code d'une tâche passager est :

```
task body Passager
  lanceur : boolean;       -- Sera-t-il celui qui démarre le bac ?
loop
  Bac.embarquerCapulet();  -- ou embarquerMontaigu
  prendre_place();
  Bac.autoriserDemarrage(lanceur);
  if lanceur then demarrer(); end if;
  ⋮
  -- débarquement et retour (hors sujet)
end loop
```

Note : l'interface Ada donnée est une proposition qui nous semble faciliter la solution. Vous pouvez en définir une autre si vous le souhaitez. Précisez alors son usage (code d'un passager).

Questions

9. Donner un automate décrivant les états et les transitions légales pour la barque et les passagers voulant embarquer selon les entrées de rendez-vous. (abréger les trois entrées en C, M et D)
10. En déduire le code de la tâche `Bac`. Ne donner le code que pour l'état initial et deux états successeurs. Préciser quelle(s) transition(s) renvoie(nt) `lanceur` à true.

2.3.2 Synchronisation par canaux

Chaque passager est une goroutine (une activité). Elles sont coordonnées via une activité de synchronisation. Pour cela, on utilise trois canaux :

```
capulet := make(chan bool)
montaigu := make(chan bool)
pilotage := make(chan bool)
```

Le code exécuté par une activité passager est :

```
func Passager() {
  capulet<- true  // ou montaigu<-true; la valeur envoyée est sans importance
  prendre_place()
  lanceur := <-pilotage
  if lanceur {
      démarrer()
  }
}
```

Note : les canaux spécifiés sont une proposition qui nous semble faciliter la solution. Vous pouvez en définir d'autres si vous le souhaitez. Précisez alors leur usage (code d'un passager).

Questions

11. Donner un automate décrivant les états et les transitions légales pour la barque et les passagers voulant embarquer selon les communications synchrones réalisées.
12. En déduire le code de l'activité de synchronisation. Ne donner le code que pour l'état initial et deux autres états, l'un où la communication est possible sur `capulet` (et éventuellement d'autres canaux), l'autre où la communication est possible sur `pilotage` (et éventuellement d'autres canaux).