Travaux Dirigés

Introduction

Ce document présente une série de problèmes de synchronisation. Ces problèmes sont exposés de manière à pouvoir être traités dans divers contextes d'exécution (mémoire partagée, processus communiquantsv), et avec les diverses primitives proposées par les langages et modèles de synchronisation (sémaphores, rendez-vous, moniteurs, expressions de chemins...).

Une application réalisée par plusieurs activités concurrentes comportera en général

- la description du code exécuté par chacune des activités ;
- la description des variables globales/objets partagés par les activités ;
- un programme principal, qui sera lui même une activité, à partir de laquelle seront lancées, implicitement ou explicitement les activités constituant l'application proprement dite.

Par exemple, une application réalisant un schéma d'interaction producteur/consommateur au moyen d'un tampon de taille bornée N (et utilisant des sémaphores pour coordonner les activités) pourra avoir la structure suivante :

```
Programme Prod_Conso;
importe lancer_activité, sémaphore, tampon, ligne, créer, utiliser;

const  C = …; {nombre de consommateurs}
       P = …; {nombre de producteurs}
       N = …; {taille du tampon}

var    t : tampon[N] <ligne>;
       mutex_conso, mutex_prod, vide, plein : sémaphore;

procédure producteur;
var  lgn : ligne;
début {producteur}
  cycle
       créer(lgn) ;
       vide.P();
       mutex_prod.P()
            t.déposer(lgn)
       mutex_prod.V()
       plein.V();
  fin cycle
fin producteur;

procédure consommateur;
var  lgn : ligne;
début {consommateur}
  cycle
       plein.P();
       mutex_conso.P()
            t.retirer(lgn)
       mutex_conso.V()
       vide.V();
       utiliser(lgn);
  fin cycle
fin consommateur;

début {Prod_Conso}
  mutex_prod.init(1);
  vide.init(vide,N);
  plein.init(0);
  mutex_conso(1);
  pour i := 1 à C faire lancer_activité(consommateur); {lance C consommateurs}
  pour i := 1 à P faire lancer_activité(producteur); {lance P producteurs}
fin Prod_Conso.
```

Variables/objets partagés

procédures exécutées par les activités

Programme/activité principal

Problèmes de synchronisation

1 Le barbier (d'après Dijkstra)

Il s'agit de modéliser l'activité coordonnée d'un barbier et de ses clients, à l'aide de processus synchronisés.

1. Pour commencer, le barbier dispose d'un unique siège, et travaille sans local. Le siège ne peut accueillir qu'un client à la fois : lorsqu'un client se présente et trouve le siège libre, il peut s'installer, pour se faire raser ; sinon il attend son tour. Un client installé sur le siège ne le quitte qu'une fois rasé. Le client suivant peut alors s'installer, pour se faire raser.
2. La boutique du barbier comporte maintenant un salon, avec un unique siège, et une salle d'attente, avec N chaises. Lorsqu'un client se présente, il entre dans la salle d'attente, s'il y a des chaises libres, sinon il attend dehors qu'une chaise se libère. Les clients entrent dans le salon lorsque le siège est libre. Un client installé sur le siège ne le quitte qu'une fois rasé. Le client suivant peut alors entrer dans le salon et s'installer, pour se faire raser.
3. Variante du scénario 2) : dans le cas où un client se présente et trouve toutes les chaises de la salle d'attente occupées, il va chez un autre barbier.
4. La boutique est maintenant prospère : si la salle d'attente comporte toujours N chaises, et S sièges ont été installés dans le salon, où B barbiers officient. Quand un client se présente, il entre dans la salle d'attente, s'il y a des chaises libres, sinon il va chez un autre barbier. Lorsqu'un siège est libre (et que son tour est venu), il quitte la salle d'attente, s'installe dans le siège et attend qu'un barbier vienne le raser. Lorsqu'un barbier a fini de raser un client, il rejoint les barbiers inemployés, qui attendent qu'un client s'installe dans un siège libre.

2 Les philosophes et les spaghettis (Dijkstra)

Cinq philosophes, réunis pour philosopher, ont, au moment du repas, un problème pratique à résoudre. En effet, le repas est composé de spaghettis qui, selon le savoir-vivre de ces philosophes, se mangent avec deux fourchettes. Or, la table n'est dressée qu'avec une seule fourchette par couvert. Après réflexion et débat, les philosophes conviennent de suivre le protocole suivant :

1. A tout instant, chaque philosophe se trouve dans l'un des trois états suivants :
   — ou bien il mange (pendant un temps fini),
   — ou bien il attend la disponibilité de ses fourchettes,
   — ou bien il pense (pendant un temps fini) et il a alors la politesse de ne garder aucune fourchette.
2. Initialement, tous les philosophes pensent.
3. Pour manger, chaque philosophe dispose d'une place à table, qui lui est allouée exclusivement.
4. Tout philosophe qui mange, utilise sa fourchette de droite et celle de gauche. Il ne peut en atteindre d'autres. Deux philosophes voisins ne peuvent donc manger en même temps.

Chaque philosophe peut être représenté par un processus cyclique dont le code est le suivant :

```
processus Philosophe
  maPlace : Place;

  PrendrePlace(maPlace);
  répéter
    penser;
    PrendreFourchettes (maPlace) ;
    manger;
    ReposerFourchettes (maPlace) ;
  sans fin ;
fin Philosophe;
```

— Programmer les fonctionnalités et le protocole associés aux actions PrendrePlace, PrendreFourchettes, ReposerFourchettes. L'action PrendrePlace est appelée lorsqu'un nouveau philosophe est lancé. Elle permet en particulier d'initialiser les variables locales au processus, mais peut aussi comporter d'autres actions, selon la stratégie envisagée.
— Rechercher une stratégie permettant d'assurer simplement l'allocation exclusive des fourchettes aux philosophes, puis une stratégie optimale c'est-à-dire autorisant le maximum de parallélisme et par conséquent réalisant une utilisation optimale des ressources.
— Etudiez les possibilités de famine et/ou d'interblocage des solutions envisagées.
— Rechercher, enfin, une stratégie assurant l'absence de famine.

3 Les lecteurs et les rédacteurs (Courtois)

On considère 2 classes de processus, des lecteurs et des rédacteurs, accédant à un fichier partagé.

Ce fichier peut être lu simultanément par plusieurs lecteurs, mais un seul rédacteur peut écrire à un instant donné, à condition qu'aucun lecteur ne lise (exclusion mutuelle entre lecteurs et rédacteurs).

Les actions assurant le protocole d'accès au fichier partagé sont les suivantes :

DébuterLecture,
TerminerLecture,
DébuterEcriture,
TerminerEcriture,

Programmer la synchronisation des lecteurs et rédacteurs dans les cas suivants :

Les lecteurs sont prioritaires sur les rédacteurs : un rédacteur ne peut accéder au fichier que si aucun lecteur n'est en cours ou en attente de lecture. Un rédacteur ne peut cependant pas être interrompu par un lecteur.

Lecteurs coalisés : si un rédacteur possède l'accès au fichier, lecteurs et rédacteurs sont mis en attente, puis servis dans l'ordre chronologique de leur requête d'accès. Cependant, dès qu'un lecteur obtient l'accès au fichier, il libère l'ensemble des lecteurs en attente. Enfin, lorsqu'un lecteur possède l'accès au fichier, seuls les rédacteurs sont mis en attente.

Lecteurs coalisés, avec équité. La solution précédente n'évite pas la possibilité d'une famine pour les rédacteurs.

- Pourquoi ?
- Amender la politique précédente, afin d'assurer une forme d'équité dans l'accès au fichier (tout processus demandant un accès finit par l'obtenir).
- Corriger le protocole en conséquence.

Accès dans l'ordre chronologique d'arrivée (ordre FIFO) sans considérer la classe (lecteur ou rédacteur) des processus.

Les rédacteurs sont prioritaires sur les lecteurs Il s'agit du cas symétrique au premier cas. Un lecteur ne peut lire que si aucun rédacteur n'est en cours d'écriture ou en attente d'accès.

4 Allocateur de ressources

Un système comporte souvent des ressources critiques c'est-à-dire non partageables et non préemptibles, comme les imprimantes. Lorsqu'un processus utilise une imprimante, aucun autre processus ne peut et ne doit y accéder. Par ailleurs, il est courant qu'un même service puisse être assuré de manière équivalente par n'importe laquelle/lesquelles des ressources d'un pool de ressources. Ainsi, un utilisateur peut avoir le choix entre plusieurs imprimantes, ce qui lui évite d'avoir à attendre la disponibilité d'une imprimante occupée, ou éteinte, ou à court de fournitures.

On envisage un protocole permettant à un processus d'acquérir par une seule action plusieurs ressources (d'une même catégorie). Dans la mesure où l'on ne souhaite pas programmer effectivement l'allocation des ressources, mais où l'on se concentre sur la coordination entre demandeurs et libérateurs de ressources, on ne considère qu'une catégorie de ressources, et on ne manipulera pas les références aux ressources proprement dites. On considèrera donc que les actions assurant le contrôle d'accès à de telles ressources sont les suivantes :

— **Allouer** (nbDemandé : entier) ;
— **Libérer** (nbRendu : entier) ;

Programmer les algorithmes de ces opérations, en envisageant différentes politiques d'allocation :

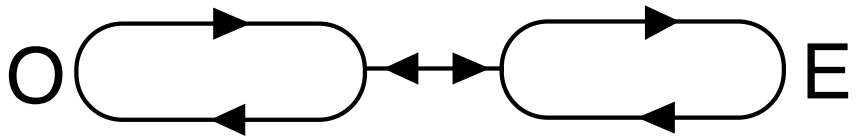
ordre FIFO : les demandes sont servies dans l'ordre chronologique d'arrivée. Cette stratégie permet de garantir l'absence de famine sur toute demande.

priorité aux petits demandeurs : lorsque des ressources sont libérées, la demande portant sur le plus petit nombre de ressources est servie en priorité. Cette stratégie permet de servir un maximum de demandes par unité de temps.

stratégie de l'ascenseur et ses variantes : les demandes sont rangées dans des files d'attentes associées aux nombre de ressources demandées (un niveau/file par valeur possible). Les demandes sont servies en parcourant les différents niveaux et en servant chaque niveau selon une politique fixée (par exemple : une seule demande par niveau, à chaque passage). Cette politique permet de combiner une forme de priorité, et une forme d'équité.

5 Voie unique (Brinch Hansen)

Deux villes E et O sont reliées par une ligne de chemin de fer qui comprend un tronçon à voie unique.



Pour s'engager sur le tronçon à voie unique les processus représentant les trains devront faire appel aux actions suivantes :

Entrer(direction)
Sortir()

avec : **type** direction = (OE, NS)

Les trains font la navette entre E et O. Leur comportement peut donc être simulé par le type de processus suivant :

```
processus train
  maDirection : direction ;

  Démarrer(maDirection); -- fixe la direction initiale
  répéter
    arriver_au_tronçon();
    Entrer(maDirection) ;
    passer le tronçon;
    Sortir() ;
    arriver_à_destination();
    maDirection := successeur(maDirection);
  sans fin ;
fin train;
```

Ecrire les algorithmes des actions Entrer et Sortir lorsque :

1. un train seulement peut se trouver sur le tronçon à voie unique ;
2. un nombre illimité de trains peuvent circuler en même temps sur le tronçon à voie unique (à condition qu'ils aillent dans le même sens) ;
3. au plus N trains peuvent circuler en même temps sur le tronçon à voie unique. Considérer les risques de privation dans ce cas, et donner les moyens d'éviter celle-ci.

6 Un tournoi de bridge

On désire modéliser le déroulement d'un tournoi de bridge. L'organisation du tournoi impose d'assurer que le nombre de personnes présentes dans la salle du tournoi soit toujours un multiple de 4. Il faut donc contrôler les entrées et les sorties selon les contraintes suivantes :

- Un joueur peut entrer si un autre est prêt à sortir (échange) ou s'il y a déjà 3 autres joueurs en attente d'entrée (le groupe de 4 peut alors entrer).
- Un joueur peut sortir si un autre est prêt à entrer (échange) ou s'il y a déjà 3 autres joueurs en attente de sortie (le groupe de 4 peut alors sortir).

Remarque : dans le cas de la sortie de 4 joueurs appartenant à des tables différentes, on ne s'intéresse pas à la réorganisation des tables de jeu pour continuer le tournoi.

L'accès à la salle est contrôlé par l'appel des actions actions Entrer et Sortir.

Le comportement d'un joueur est modélisé par le type de processus suivant :

```
processus Joueur
  {se rendre sur le lieu du tournoi()}
  Entrer() ;
  {Jouer()}
  Sortir() ;
  {rentrer chez soi()}
fin Joueur ;
```

Questions

1. On définit 2 variables d'état ne et ns variant dans l'intervalle [0..4]. Lorsqu'aucune opération Entrer ou Sortir n'est en cours :
   - La variable ne compte le nombre d'entrants en attente ;
   - La variable ns compte le nombre de sortants en attente.

   A l'aide de ces variables exprimez les prédicats suivants :
   (a) La condition pour qu'une opération Entrer provoque un échange avec un sortant.
   (b) La condition pour qu'une opération Sortir provoque un échange avec un entrant.
   (c) La condition de blocage d'un entrant.
   (d) La condition de blocage d'un sortant.
   (e) L'invariant global liant les variables d'état ne et ns.
2. Déduire des spécifications précédentes, les conditions de synchronisation nécessaires.
3. En déduire les algorithmes des opérations Entrer et Sortir. Annotez les algorithmes de manière à prouver leur correction.