Travaux dirigés de Systèmes concurrents

12 octobre 2022

Résumé

Ce document présente une série de problèmes de synchronisation. Ces problèmes sont exposés de manière à pouvoir être traités dans divers contextes d'exécution (centralisé, réparti), et avec les divers constructeurs proposés par les langages et modèles de synchronisation (sémaphores, rendez-vous, moniteurs, expressions de chemins...)

1 Allocateur de ressources critiques

Un système comporte généralement des ressources critiques c'est-à-dire non partageables et non préemptibles, comme les pages mémoire. Lorsqu'un processus utilise une page mémoire, aucun autre processus ne peut et ne doit y accéder. De plus, un processus ne doit pas attendre la libération d'une page mémoire occupée s'il existe par ailleurs des pages disponibles.

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 considère donc que les actions assurant le contrôle d'accès à de telles ressources sont les suivantes :

```
ALLOUER (nbDemandé : entier);
LIBERER (nbRendu : entier);
```

Programmer les algorithmes de ces opérations, en envisageant différentes stratégies d'allocation : FIFO, priorité aux petits demandeurs, technique de l'ascenseur et ses variantes.

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 quelques instants de réflexion, les philosophes décident d'adopter le comportement suivant :

1. Chaque philosophe prend place à table.
2. Tout philosophe qui mange, utilise la fourchette de droite et celle de gauche. Il ne peut en atteindre d'autres. Deux philosophes voisins ne peuvent donc manger en même temps.
3. 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 cherche à acquérir ses fourchettes,
   — ou bien il pense (pendant un temps fini) et il a alors la politesse de ne garder aucune fourchette.
4. Initialement, tous les philosophes pensent.

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

```
process Philosophe is
    maPlace : Place;
begin
    maPlace ← PRENDRE_PLACE();
    loop
        penser;
        PRENDRE_LES_FOURCHETTES (maPlace);
        manger;
        REPOSER_LES_FOURCHETTES (maPlace);
    end loop;
end Philosophe;
```

Programmer les fonctionnalités et le protocole associés aux actions `PRENDRE_PLACE`, `PRENDRE_LES_FOURCHETTES`, `REPOSER_LES_FOURCHETTES`.

Rechercher une stratégie optimale c'est-à-dire autorisant le maximum de parallélisme et par conséquent réalisant une utilisation optimale des ressources.

Étudier les possibilités de famine et/ou d'interblocage des solutions envisagées.

3 Les lecteurs et les rédacteurs (Courtois)

On envisage deux 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 :

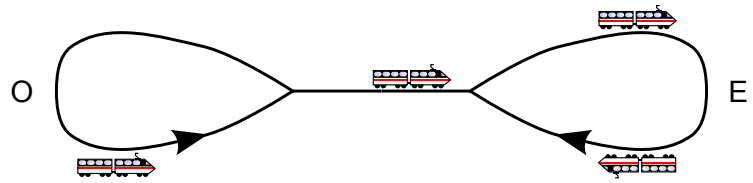
```
DEBUTER_LECTURE,
ARRETER_LECTURE,
DEBUTER_ECRITURE,
ARRETER_ECRITURE
```

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

1. 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.
2. 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.
3. 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).
   — Modifier le protocole en conséquence.
4. Accès au fichier dans l'ordre chronologique d'arrivée (ordre FIFO), indépendamment de la classe (lecteur ou rédacteur) des processus.
5. Les rédacteurs sont prioritaires sur les lecteurs. Il s'agit du cas symétrique au cas 1. Un lecteur ne peut lire que si aucun rédacteur n'est en cours d'écriture ou en attente d'accès.

4 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 : [1]



Pour s'engager sur le tronçon à voie unique les processus représentant les trains devront faire appel aux actions suivantes, où `DIRECTION` appartient à $\{OE, EO\}$:

```
ENTRER(DIRECTION)
SORTIR
```

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

```
process train is
    maDirection : Direction ;
begin
    maDirection ← DEMARRER(); -- fixe la direction initiale
    loop
        arriver_au_tronçon;
        ENTRER(maDirection);
        passer le tronçon;
        SORTIR;
        arriver_à_destination;
        maDirection ← successeur(maDirection);
    end loop ;
end train;
```

Écrire 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 famine, et donner les moyens de les éviter.

1. (train : source : wikicommons)

5 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, 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.

6 Les fumeurs de cigarettes (Patil)

Trois (gros) fumeurs, représentés par des processus FA, FF et FT, roulent leurs cigarettes eux-mêmes. Ils ont besoin de tabac, de feuilles, et d'allumettes. Chacun d'eux dispose en permanence d'un seul des trois ingrédients : le tabac pour FT, les feuilles pour FF, les allumettes pour FA. Un quatrième processus, l'agent, dispose en permanence des trois ingrédients. L'agent place sur une table deux des trois ingrédients, à sa convenance. Les fumeurs ne peuvent prendre les ingrédients placés sur la table qu'ingrédient par ingrédient. Les fumeurs doivent donc se coordonner afin que ce soit seulement le fumeur auquel manquent précisément les deux ingrédients fournis par l'agent qui se procure ces ingrédients. L'activité des fumeurs est cyclique. Un cycle consiste à se procurer les deux ingrédients manquants, à rouler une cigarette, à fumer la cigarette, et à rendre à l'agent les ingrédients empruntés. L'agent ne place une nouvelle paire d'ingrédients que lorsque les ingrédients qu'il a fournis auparavant lui ont été rendus.

Coordonner l'activité de l'agent et des trois fumeurs.

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

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

Le contrôle d'accès à la salle est assuré par l'appel des actions `ENTRER` et `SORTIR`. Le comportement d'un joueur est modélisé par le type de processus suivant :

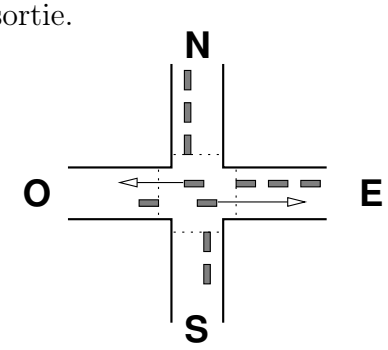
```
process Joueur is
begin
    se rendre sur le lieu du tournoi
    ENTRER;
    Jouer
    SORTIR;
    retourner chez soi
end Joueur ;
```

1. On définit deux 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 exprimer 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 vérifié par 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`. Annoter les algorithmes de manière à prouver leur correction.

8 Gestion d'un carrefour

On désire assurer la commande des feux tricolores d'un carrefour par ordinateur. On suppose pour cela que des capteurs placés sous les voies de circulation détectent le passage des véhicules en entrée et en sortie.



Pour traverser (régulièrement !) le carrefour, les processus devront faire appel aux actions suivantes :

```
ENTRER(VOIE)
QUITTER
```

où `VOIE` prend ses valeurs dans {OE, NS}.

Chaque fois qu'un véhicule entre, une des deux actions `ENTRER(OE)` ou `ENTRER(NS)` est appelée. De même, chaque fois qu'un véhicule sort, l'action `QUITTER` est appelée. Plusieurs véhicules peuvent se trouver dans le carrefour.

1. Expliquer pourquoi l'origine du véhicule (voie et direction exactes d'entrée) n'est pas une information nécessaire pour l'opération `QUITTER`
2. On définit les règles suivantes de commande des feux :
   (a) Les véhicules arrivant sur la voie ouverte peuvent passer tant qu'il n'y a aucun véhicule en attente sur l'autre voie. Cependant, s'il existe des véhicules en attente sur l'autre voie, on limite le flot de véhicules passés successivement sur la voie ouverte à une valeur maximale N.
   (b) Il y aura commutation de la voie ouverte chaque fois que le carrefour redevient vide, qu'il y ait ou non des véhicules en attente d'entrée.
   (c) Lorsque le carrefour est vide et qu'aucun véhicule n'est arrêté, le feu reste dans une position fixe autorisant le passage de futurs arrivants sur l'une des deux voies. Lorsqu'un premier véhicule arrive sur la voie ouverte, les feux restent inchangés. Mais, si ce premier véhicule arrive sur la voie fermée, les feux doivent commuter pour laisser passer ce véhicule.
   (d) Attention : lorsqu'un flot de N véhicules est entré, la voie ouverte est fermée mais on prendra soin de n'ouvrir la voie opposée que lorsque le carrefour sera vidé. Pendant un certain laps de temps, tous les feux sont alors au rouge (les deux voies sont fermées). On suppose que les automobilistes jouent le jeu, c'est-à-dire qu'ils ne grillent pas les feux.

3. Déterminer les variables d'état nécessaires au problème.
4. Définir les prédicats associés aux entrées.
5. Programmer les actions `ENTRER` et `QUITTER`. On supposera disponible les procédures permettant d'ouvrir ou fermer une voie par le passage au vert ou au rouge des feux associés à cette voie.

   ```
   procedure OUVRIR(v: VOIE);
   procedure FERMER(v: VOIE);
   ```

6. Si un automobiliste grille un feu rouge, quelle anomalie provoque-t-il dans le comportement de la tâche ? Modifier les actions pour qu'elles résistent aux feux grillés.