Chapitre 3

Exclusion mutuelle répartie

Chap 3 Exclusion mutuelle répartie

- Pas de mémoire commune
  => échanges de messages
- Protocole exempt d'interbolcage et équitable
  => un processus désireux de pénétrer en section critique y parvient en un temps fini
- Deux grandes catégories d'algorithmes d'exclusion mutuelle :
  - basés sur l'utilisation de variables d'état distribuées
  - basés sur la communication de messages

1. Algorithme basé sur l'utilisation de variables d'état

- Algorithme de la boulangerie [Lamport 74]

  - ne s'appuie sur aucun dispositif centralisé
  - Analogie avec le ticket d'ordre d'arrivée des clients dans un magasin

  Principe :

  - chacun des n processus P0,… Pn-1 choisit son propre numéro en fonction des numéros pris par les autres processus.

  - si deux processus ont choisi le même numéro, alors celui dont l'indice est le plus petit passe devant (symétrie de texte).

  => *simulation d'une mémoire partagée accessible aux processus*

1. Algorithme de la boulangerie

- Variables :

  choix : tableau [0 .. n-1] de booléen ; /* initialisés à faux */

  num : tableau [0 .. n-1] de entier ; /* initialisés à 0 */

- Le couple (choix [i], num [i]) est la propriété de Pi

- Implémentation distribuée de ces tableaux :
  - chaque processus P_i maintient la case du tableau correspondant à num[i] et choix[i],
  - lorsque Pi veut connaître la valeur de num[j] ou choix[j] ($j \neq i$), il la demande à Pj. (message).

- Rappel : relation d'ordre sur les couples d'entiers :

  $(a, b) < (c, d) \Leftrightarrow (a < c)$ ou $((a = c)$ et $(b < d))$

1. Algorithme de la boulangerie

- Algorithme exécuté par le processus P_i :

```
Début
        choix [i] ← vrai ;
        num [i]   ← 1 + Max (num [0], …, num [n-1] ) ;
        choix [i] ← faux ;

        pour j de 0 à n-1
            si j ≠ i alors
             attendre ¬ choix [j] ;
             attendre (num [j] = 0) ou ((num[i], i) < (num[j], j)) ;
           fsi
        fpour

< section critique > ;
num[i] ← 0 ;
Fin
```

1. Algorithme de la boulangerie

- le processus P_i est :
  - « entrant » lorsque choix[i] = vrai,
  - « dedans » entre les instructions :
    « choix [i] ← faux » et « num[i] ← 0 »
- Exemple : Cf. feuille des exercices du chapitre 3
- Preuve : [exercice]
- Inconvénient de cet algorithme : la croissance des variables num[i] peut être infinie s'il y a toujours au moins un processus dans la zone « *dedans* ».

2. Algorithmes basés sur les échanges de messages

- Les processus ne possèdent que des variables locales

  $\Rightarrow$ pas de variables d'état distribuées

- Chaque fois qu'un processus modifie son état

  $\Rightarrow$ diffuse son nouvel état aux autres processus

- Ces algorithmes minimisent le nombre de messages

  $\Rightarrow$ les messages correspondent à des modifications de l'état des processus

2.1. Algorithme de Ricart et Agrawala (Suzuki / Kasami) ⇒ jeton sur un réseau maillé

- Hypothèses :
  - le réseau de communication est complètement maillé,
  - les voies de communication assurent le transport sans erreur,
  - le délai de transmission est variable,
  - le déséquencement des message est possible

- Principe :
  - le processus qui est en section critique possède un privilège matérialisé par un jeton
  - tant qu'un processus garde le jeton, il peut pénétrer en SC sans consulter les autres

2.1. Algorithme de Ricart et Agrawala (Suzuki / Kasami)

- Principe (suite) :
  - initialement le jeton est affecté à un processus quelconque
  - le jeton est demandé par le processus Pi (i $\in$ [1..n]) à l'aide d'une requête estampillée et diffusée à tous les autres processus
  - le jeton est constitué d'un tableau dont le k-ième élément mémorise l'estampille de la dernière visite qu'il a effectuée au processus Pk
  - lorsque le processus Pj qui possède le jeton, ne désire plus accéder en SC, il cherche dans le tableau qui matérialise le jeton, le premier processus P_l (l choisi dans l'ordre j+1, … n, 1, … j-1) tel que l'estampille de la dernière requête de P_l soit supérieure à l'estampille mémorisée par le jeton lors de sa dernière visite à P_l. P_j envoie alors le jeton à P_l.

2.1. Algorithme de Ricart et Agrawala (Suzuki / Kasami)

- Algorithme :

  - var
    - horlog : entier initialisé à 0 ; /* horloge logique */
    - jetonprésent : booléen ;
    - dedans : booléen initialisé à faux ;
    - jeton : tableau[1..n] de entier initialisé à 0 ;
    - requêtes : tableau[1..n] de entier initialisé à 0 ;

- Le booléen « jetonprésent » est initialisé à faux dans tous les processus sauf un, celui qui possède initialement le jeton.

- L'opération attendre (token, jeton) permet d'attendre jusqu'à l'arrivée d'un message de type *token*, qui est alors placé dans la variable *jeton*.

2.1. Algorithme de Ricart et Agrawala (Suzuki / Kasami)

```
Début

/* début prélude */
si ¬ jetonprésent alors
            début
            horlog ← horlog + 1 ;
            diffuser (req, horlog, i) ;
            attendre (token, jeton) ;
            jetonprésent ← vrai ;
            fin ;
fsi ;
/* fin prélude */
dedans ← vrai ;
< section critique >
dedans ← faux ;
```

2.1. Algorithme de Ricart et Agrawala (Suzuki / Kasami)

```
/* début postlude */
jeton(i) ← horlog ;
pour j de i + 1 jusqu'à n, 1 jusqu'à i - 1 faire
      si requête(j) > jeton(j) et jetonprésent alors
                  début
                  jetonprésent ←  faux ;
                  envoyer (token, jeton) à j ;
                  fin
      fsi
/* fin postlude */

si réception (req, k, j) alors
      début
      requêtes(j) ← max (requêtes(j), k) ;
      si jetonprésent et ¬ dedans alors
                  < texte identique au postlude >
      fsi
fsi
Fin
```

2.1. Algorithme de Ricart et Agrawala (Suzuki / Kasami)

- Exemple : Cf. feuille d'exercices

- Preuve : [à faire sous forme d'exercice]
  - exclusion mutuelle
  - non-interblocage et équité

- Nombre de messages générés : n

- Utilisation des estampilles :
  - Les estampilles servent de compteurs qui mémorisent le nombre de fois où un processus a voulu pénétrer en SC.
  - Ces compteurs sont utilisés de manière différentielle
  - La fonction « max » utilisée lors du traitement de la réception des requêtes sert à ne considérer que la dernière requête de Pj (cas de déséquencement).

2.1. Algorithme de Ricart et Agrawala (Suzuki / Kasami)

- Cas de perte des messages :
  - perte d'une requête :
    - le processus émetteur restera bloqué sans pouvoir accéder à la SC (à moins qu'il n'utilise une temporisation).
    - l'algorithme continue à fonctionner pour les processus restants
  - perte du jeton :
    - tous les autres processus vont être inter-bloqués

      => algo de regénération du jeton.

2.2. Algorithme de Misra : regénération du jeton

2.2.1. Jeton sur un anneau logique

- La topologie de communication est un anneau
- Le privilège est matérialisé par un *jeton*

- Le protocole d'accès à la SC par le processus P_i ($i \in [0..n-1]$) est :

Début
Attendre (jeton) de Pi-1 mod n ;
< section critique>
Envoyer (jeton) à Pi+1 mod n ;
Fin

2.2. Algorithme de Misra : regénération du jeton

2.2.2. Algorithme de Misra (83)

Principe :

- Cet solution algorithme utilise deux jetons dont chacun sert à détecter la perte de l'autre selon le principe suivant :
- un jeton arrivé au processus P_i peut garantir que l'autre jeton est perdu (et alors le regénérer) si depuis le passage précédent de ce jeton dans P_i, ni lui ni le processus P_i n'ont rencontré l'autre jeton.
- Les deux jetons ont des comportements symétriques (du point de vue de la SC, le privilège est rattaché à un seul d'entre eux).
- La perte d'un jeton est détectée par l'autre jeton en un tour sur l'anneau : cet algorithme ne fonctionne donc que si, un jeton étant perdu, l'autre effectue un tour sans se perdre.
- On peut généraliser l'algo à l'utilisation de N jetons : il fonctionne alors tant qu'il reste un jeton sur l'anneau.

2.2.2. Algorithme de Misra (83)

- Soient *ping* et *pong* les deux jetons auxquels sont associés deux nombres respectivement *nbping* et *nbpong* ;

- la valeur absolue de chacun d'eux compte le nombre de fois où les jetons se sont rencontrés, et leurs valeurs sont liées par l'invariant :

  nbping + nbpong = 0

- Initialement les jetons sont dans un processus quelconque de l'anneau, et l'on a :

  nbping = 1 et nbpong = -1

- Chaque processus Pi est doté d'une variable locale m qui mémorise la valeur associée au dernier jeton qu'a vu passer le processus :

  var m : entier := 0 ;

2.2.2. Algorithme de Misra (83)

L'algorithme exécuté par le processus P_i est le suivant :

lors de

- la réception de (ping, nbping) faire

```
début
    si m = nbping alors
        début <pong est perdu, il est regénéré>
            nbping ← nbping + 1;
            nbpong ← - nbping ;
        fin
    sinon m ← nbping
    fsi
fin
```

2.2.2. Algorithme de Misra (83)

- la réception de (pong, nbpong) faire

  <traitement analogue en intervertissant les rôles de ping et pong>

- la rencontre de 2 jetons faire

  début

  nbping ← nbping + 1;

  nbpong ← nbpong - 1;

  fin

Preuve : faire sous forme d'exercice

2.2.2. Algorithme de Misra (83)

Taille des compteurs :

- La taille des compteurs *nbping* et *nbpong* n'est pas bornée a priori, ce qui constitue un inconvénient majeur.

- Il est nécessaire que lorsque les compteurs sont mis à jour, ils soient incrémentés (en valeur absolue) et prennent alors des valeurs différentes de celles prises par toutes les variables m_i des processus P_1 à P_n .

- Il est donc possible d'incrémenter le compteur *nbping modulo n+1* (idem pour nbpong).

2.3. Algorithme de Lamport : distribuer une file d'attente

- Hypothèse
  - Les messages ne se déséquencent pas (pas de croisement).

- Principe
  - Dans un système centralisé, on peut réaliser l'exclusion mutuelle en définissant un processus allocateur qui gère l'accès à la section critique en utilisant une file d'attente.
  - Pour distribuer cet algorithme centralisé, il est nécessaire de répartir la file sur tous les sites : chaque site reçoit tous les messages de requête et de libération de tous les autres sites.
  - Les messages sont totalement ordonnés en utilisant le mécanisme d'estampillage de Lamport.
  - Pour qu'un processus puisse prendre une décision au vu de l'état de sa seule file d'attente, il faut qu'il ait reçu un message « assez récent » de chacun des autres processus.

2.3. Algorithme de Lamport : distribuer une file d'attente

- Algorithme

  - Les messages peuvent être de trois types :
    - *requête* : lorsqu'un processus désire rentrer en SC il diffuse un message de type « *req* »
    - *release* : lorsqu'un processus quitte la SC il diffuse un message de type « *rel* »
    - *acquittement* : lorsque le processus P_j a reçu un message « req » de P_i , il lui signale la réception par un accusé de réception « *acq* »

  - Chaque processus possède une horloge locale et émet des messages constitués de trois champs :

    (type, horloge locale, n° du site)

2.3. Algorithme de Lamport : distribuer une file d'attente

- Chaque processus est doté d'une file d'attente, et gère les variables locales suivantes :

  var h : horloge ;

  f : tableau [0..n-1] de message ;

- L'horloge h est gérée selon le principe des estampilles de Lamport : respect de l'ordre causal.

- La gestion de la file d'attente est la suivante :
  - à tout instant l'entrée f [j] contient un message en provenance de P_j .
  - initialement on a : f [j] = (rel, 0, j).
  - quand un message est diffusé par P_i , il est également enregistré dans f [i].

2.3. Algorithme de Lamport : distribuer une file d'attente

- La mise à jour de f [j] par P_i se fait ainsi :
  - à la réception d'un message (req, k, j) ou (rel, k, j) celui-ci est placé dans f [j]. Rappelons que la réception par Pi d'un message de type (req, k, j) provoque l'émission de P_i vers P_j d'un message (acq,h,i).
  - à la réception d'un message (acq, k, j), celui-ci est placé dans f [j] si cette entrée *ne contient pas un message de type req*. Le message est ignoré sinon.
  - Un processus P_i s'octroie le droit d'entrer en SC lorsque le message contenu dans f [i] est du type *req* et que son estampille est la plus ancienne : sa requête précède alors toutes les autres.

2.3. Algorithme de Lamport : distribuer une file d'attente

- Protocole réalisé par P_i :
  - Demande d'entrée en SC :

Début

diffuser (req, h, i) ;
f [i] ← (req, h, i) ;
h ← h + 1 ;

attendre ∀ i ≠ j , estampille_de (f [i]) < estampille_de (f [j]);

< SC >

diffuser (rel, h, i) ;
f [i] ← (rel, h, i) ;
h ← h + 1 ;
Fin

2.3. Algorithme de Lamport : distribuer une file d'attente

A la reception de :

- *(req, k, j)* faire
  début
  maj (h, k) ;
  f [j] ← (req, k, j) ;
  envoyer (acq, h, i) à j ;
  fin ;

- *(rel, k, j)* faire
  début
  maj (h, k) ;
  f [j] ← (rel, k, j) ;
  fin ;

- *(acq, k, j)* faire
  début
  maj (h, k) ;
  si type de f [j] ≠ req alors
  f [j] ← (acq, k, j) ;
  fsi ;
  fin ;

2.3. Algorithme de Lamport : distribuer une file d'attente

- Remarques :

  - la fonction *estampille_de* délivre les valeurs de l'horloge h et du n° du site

  - la fonction *maj* réalise la mise à jour de l'horloge locale selon le principe de Lamport :

```
maj (h, k : horloge) :
début
si h < k alors h ← k fsi ;
h ← h + 1 ;
fin ;
```

Exemple : compléter le scénario de la feuille d'exercice

Preuve : faire sous forme d'exercice