Théorie et pratique de la concurrence

CM7 Exclusion mutuelle

Algorithme de Dekker

Lundi 25·10·2021

Exclusion mutuelle

Un problème de base dans les systèmes concurrents est de garantir qu'une certaine section critique, une certaine partie (une ressource par exemple) doit être en exclusion mutuelle entre les processus·

On a vu les choses abstraites comme les réseaux de Petri, mais comment ça se fait concrètement ? Si on veut programmer ça, comment faire ?

Donc, on va essayer d'aller plus dans les détails du problème·

Processus

Répéter
  non critique
  critique

On doit faire un protocole pour gérer l'entrer et la sortie à cette section critique

Processus

Répéter
  non critique
  pré-protocole
  critique
  post-protocole

Comment on va réaliser le « pré-protocole » et le « post-protocole » ?

On va ajd regarder une certaine solution·

Lorsque on fait un protocole comme ça, faut faire attention à un certain nombre de choses :

- Permettre le plus de parallélisme possible.
- Faut avoir une solution correcte.
    - Une solution qui fait ce qu'elle doit faire. Donc, on parle de la propriété sureté (safety) : Tous ce qui est fait, est admissible. Quelque chose de mauvais n'arrive jamais. A chaque fois qu'on avance on reste dans un domaine sur. Ici, notre domaine sur est que 2 processus ne doive jamais être tous les 2 en même temp en section critique (jamais 2 processus ne sont dans leur section critique en même temp).
    - Mais, ce n'est pas suffisant, car on risque de crée un protocole qui ne permet à personne d'avancer. C'est pour ça que faut avoir aussi de la vivacité (livness) : un terme plutôt général qui dit que quelque chose de bon doit arriver de temp en temp, c.-à-d. : faut que ça avance, il est tjr possible que quelque chose de bon arrive. Donc, ça veut dire qu'on est dans une situation où on attend, rien de mauvais nous arrive, mais rien de bon non-plus. Il ne faut pas qu'on reste éternellement en attente.

Parmi les propriétés de vivacité :

- **Blocage** : il y a 2 situations : deadlock et livelock
    - **Deadlock** : une situation où personne ne peut avancer, aucune action est possible pour personne (pour l'ensemble des processus).
    - **Livelock** : Ce n'est pas qu'on ne bouge pas, ce n'est pas qu'il n'y a pas d'action, mais ils n'avancent pas, ils tournent en rond, y'a pas de progrès vers le but, y'a rien qui fait avancer vers quelque chose de bon. Par exemple : j'essaie de laissai l'autre passer, et lui essaie

de me laissai passer.

- **La famine (starvation)** : il ce peut qu'une parti des processus, un certain groupe de processus, avance au détriment d'autres. C'est une propriété locale pour un processus ou plusieurs, alors que d'autres peuvent avancer.

De manière général,

Lorsque on va vouloir écrire un protocole faut :

1. Qu'il soit correct du point de vue de sûreté
2. Sans blocage
3. Sans famine

} Ou au moins les éviter le plus possible.

1er solution

On écrit en pseudo-code

```
turn : integer

pro p1 // procédure (une function)
begin
  repeat
    while turn = 2 do {} ; // Il fait rien. C'est de l'attente active.
     crit 1 ;
     turn := 2 ;
     noncrit 1 ;
  for ever
end

pro p2
begin
  repeat
    while turn = 1 do {} ; // Il fait rien. C'est de l'attente active.
     crit 2 ;
     turn := 1 ;
     noncrit 2 ;
  for ever
end


begin
  turn := 1 ;
  cobegin
    p1 ; p2
  coend
end
```

- *Les 2 procédures vont s'exécuter en parallèle.*

- Dans cet exemple on suppose qu'il y a quelque chose qui fait en sorte que la lecture / écriture de **trun** est sérialiser. Il y a une procédure qui aura accès avant l'autre, mais on ne sait pas qui avant l'autre.
- Est-ce que notre solution assure la sureté ? Oui.
- Et l'exclusion mutuelle ? Oui.

Avec ce code on n'a jamais 2 procédures en même temp en section critique. Comme c'est un programme simple, ça se voit à l'œil nu.

Pour p1, pour entrer en section critique, faut que **turn** soit égal à 1, et la procédure qui va mettre la valeur a 1, c'est p2 et cela veut dire que p2 est pas en section critique.

Mtn, on a d'autres critères :

Il peut y avoir un blocage ?

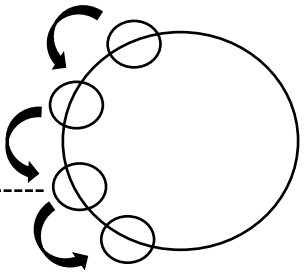
Oui, si la section critique dur infiniment ou si le processus meure en section critique. Donc, pour pouvoir résonner sur ce pbm de blocage, on va supposer que le processus ne peut pas mourir.

Cependant, c'est plus dur de faire cette hypothèse sur la partie non-critique. Si p1 se plante en section non-critique, est-ce que c'est un danger ? OUI, car p2 a besoin que p1 lui "rend la clé" pour qu'il puisse avoir accès à sa section critique.

Si on attend que l'autre nous passe le tour, et il n'existe plus : c'est un pbm.

De manière général, cette solution a un défaut : il est vulnérable au fautes, si un meure, tlm est bloquer.

Si lui meure,
on est bloqué.



Le 2eme défaut du programme :

La séquentialisation des processus.

- Une fois que j'ai utilisé une fois la section critique, je dois attendre que les autres aussi entre en section critique même s'ils ne sont pas intéressés.
- Donc, on a un problème de manque de parallélisme car on a rendu le programme complètement séquentiel, ça ne sert à rien de lancer les procédures en parallèle car on a réglé l'ordre d'entrer en section critique, donc pas de concurrence.

Donc, le problème est le manque de parallélisme et le risque de blocage suite a des pannes.

- La séquentialisation enlève le problème d'exclusion mutuelle, mais c'est trop brutal.
- Donc, une idée qu'on peut avoir est de paralléliser les sections non-critique. On va créer 2 variables, chaque un va signaler qu'il est intéressé a entrer en section critique à l'aide de ces variables et en fonction de ça je vais décider si entrer ou pas en section critique. Les variables on va les appeler c1, c2.

2eme solution

Pseudo-code

```
c1, c2 : integer

proc p1 // procédure 1
begin
  repeat
    while c2  = 0 do {} ; // L'autre est intéresser par la section critique.
     c1 := 0 ;
     crit 1 ;
     c1 := 1 ;
     non critique
  for ever
end

proc p2 // procédure 2
begin
  repeat
    while c1  = 0 do {} ; // L'autre est intéresser par la section critique.
     c2 := 0 ;
     crit 2 ;
     c2 := 1 ;
     non critique 2
  for ever
end

begin
  c1 := 1 ; c2 := 1 ;
   cobegin
     p1 ; p2
   coend
end
```

- *Il y a problème· Si tous les 2 arrive en même temp au* **while-do**, *chaque un met sa variable a 0, et comme le test a déjà été effectué, ils vont tous les deux entrer en section critique·*

```
proc p2 // procédure 2
begin
  repeat
    while c1  = 0 do {} ; // L'autre est intéresser par la section critique.
    c2 := 0 ;
    crit 2 ;
    c2 := 1 ;
    non critique 2
  for ever
end
```

Il y a un trou entre le test et le moment où je déclare que je suis intéressée.

Et si on inverse l'ordre de ces 2 lignes ?

3eme solution

Pseudo-code

```
c1, c2 : integer

proc p1 // procédure 1
begin
  repeat
    c1 := 0 ;
    while c2  = 0 do {} ; // L'autre est intéresser par la section critique.
    crit 1 ;
    c1 := 1 ;
    non critique
  for ever
end
```

```
proc p2 // procédure 2
begin
  repeat
    c2 := 0 ;
    while c1  = 0 do {} ; // L'autre est intéresser par la section critique.
    crit 2 ;
    c2 := 1 ;
    non critique 2
  for ever
end

begin
  c1 := 1 ; c2 := 1 ;
   cobegin
     p1 ; p2
   coend
end
```

Problème : livelock. Si pc1 fait `c1 := 0 ;` *et pc2 fait* `c2 := 0 ;` *alors : livelock.* ***Donc, la solution est pas bonne.***

4eme solution

En bleu à droite : la solution précédente ou il y a l'exclusion mutuelle mais avec livelock.

```
c1, c2 : integer

proc p1
begin
 repeat
    c1 := 0 ; // Je suis intéressé.
    while c2  = 0 do // Si l'autre est intéresser
     begin
       c1 := 1 ;
       c1 := 0 ;
     end
    crit 1
    c1 := 1 ;
    non crit 1
 for ever
end
```

```
repeat
   c1 := 0 ;
   while c2  = 0 do {} ;
   crit 1 ;
   c1 := 1 ;
   non critique
for ever
```

```
proc p2
begin
 repeat
    c2 := 0 ; // Je suis intéressé.
    while c1  = 0 do // Si l'autre est intéresser
     begin
       c2 := 1 ;
       c2 := 0 ;
     end
    crit 2
    c2 := 1 ;
    non crit 2
 for ever
end

begin
 c1 := 1 ; c2 := 1 ;
 cobegin
   p1 ; p2
 coend
end
```

```
repeat
   c2 := 0 ;
   while c1  = 0 do {} ;
   crit 2 ;
   c2 := 1 ;
   non critique 2
for ever
```

Dans cette solution l'idée est de ne pas insister : abandonner et réessayer.

- ***Exclusion mutuelle ?*** ***(Quand y'a pas de blocages)***
  Quand je suis dans la section critique, l'autre ne peut pas être.
  Donc : safety : OK
- ***S'il fond tous les deux la même chose en même temp : alors ça va durer à l'éternité.***
- *En réalité, les chances que ça ne débloque pas jamais sont rare, mais on ne sait pas quand ça va se débloquer (combien de temp on va mettre avant de débloquer ?)*
- *Donc : system imprédictible (imprévisible) au niveau de performance.*
- *Problème de livelock.*
- *Il y a trop de symétrie. Comment casser cette symétrie ?*

- ***Donc, la seul bonne solution (malgré les pannes) est la première. Mais, il nous faut un système de tour, ce qui n'est pas le cas pour la première solution.***

Algorithme de Dekker

c1, c2 : integer

```
proc p1
begin
  repeat
    c1 := 0 ; // Je suis intéressé.
    while c2  = 0 do // Tant que l'autre est aussi a 0
      if turn = 2 then // Qui a le tour
       begin
        c1 := 1 ;
        while turn = 2 do {} // Jattend
        c1 := 0 ;
       end
    crit 1 // Section critique
    turn = 2 // Le tour est a p2
    c1 := 1 ;
    non crit 1
  for ever
end
```

En situation de conflit on regarde à qui est le tour (et que dans ce cas la , contrairement a la solution 10

```
proc p2
begin
  repeat
    c2 := 0 ; // Je suis intéressé.
    while c1  = 0 do // Tant que l'autre est aussi a 0
      if turn = 1 then // Qui a le tour
       begin
        c2 := 1 ;
        while turn = 1 do {} // Jattend
        c2 := 0 ;
       end
    crit 2 // Section critique
    turn = 1 // Le tour est a p1
    c2 := 1 ;
    non crit 2
  for ever
end
```

```
begin
  c1 := 1 ; c2 := 1 ; turn := 1 ;
  cobegin
    p1 ; p2
  coend
end
```

- *Sans blocages*
- *Exclusion mutuelle*
- ***Donc, une solution intéressante pour résoudre notre problème.***