Théorie et pratique de la concurrence

Mihaela Sighireanu

Mihaela.Sighireanu@liafa.jussieu.fr

http://www.liafa.jussieu.fr/~sighirea/cours/concur/

Bureau 6A7, Chevaleret

Synchronisation fine

— sections critiques —

Problème des sections critiques

= implémentation d'une synchronisation par exclusion mutuelle

Schéma général :

```
proctype Proc(int i) {
    do
    :: true -> protocole_entree_sc
               section_critique
               protocole_sortie_sc
               section_non_critique
    od
}
```

Hypothèses :

- tout processus qui entre dans sa section critique va forcement sortir
- un processus peut se terminer qu'en dehors de sa section critique

Propriétés souhaitées

- **Exclusion mutuelle** : au plus un processus exécute sa section critique à un moment donné.
- **Absence de blocage** : si deux ou plusieurs processus essayent d'entrer dans leur sections critiques, au moins un réussira.
- **Absence de délai** : si un processus essaye d'entrer en section critique et les autres processus exécutent leur sections non-critiques ou sont terminés, alors le premier processus ne sera pas retardé d'entrer dans sa section critique.
- **Absence de famine** : un processus qui essaye d'entrer dans sa section critique va forcement y rentrer.

Solution binaire, non-primitive

non-primitive = utilise des instructions évolués (**atomic**).

```
bool in1 = false, in2 = false;
{ MUTEX : ! (in1 && in2) }
active proctype P1() {
  do :: true -> { MUTEX && !in1 }
                 atomic { !in2 -> in1 = true };
                 { MUTEX && in1 }
                 /* section critique */
                 in1 = false;
                 { MUTEX && !in1 }
                 /* section non-critique */
  od
}
active proctype P2() {
  do :: true -> { MUTEX && !in2 }
                 atomic { !in1 -> in2 = true };
                 { MUTEX && in2 }
                 /* section critique */
                 in2 = false;
                 { MUTEX && !in2 }
                 /* section non-critique */
  od
}
```


Propriétés

Exclusion mutuelle : invariant global MUTEX

Absence de blocage : (par exclusion de configurations)

1. P1 veut, mais ne peut pas entrer = MUTEX$\wedge$!in1$\wedge$in2
2. P2 veut, mais ne peut pas entrer = MUTEX$\wedge$!in2$\wedge$in1

comme 1 $\wedge$ 2 = F, blocage exclu.

Absence de délai : (par exclusion de configurations) pour P2

1. P1 est en dehors de sa section critique = !in1
2. P2 essaye d'entrer mais il ne reussi pas = ! (!in1)

comme 1 $\wedge$ 2 = F, délai exclu.

Absence de famine : si scheduling avec équité forte

```
P1 : atomic              !in1 && !in2 <-+
P2 : atomic                             |
P2 : section critique    !in1 && in2    |
P2 : in2 = false         !in1 && !in2 --+
```

Solution binaire, primitive

Utilise l'instruction machine "Test-And-Set" :

```
TS(bool l,bool c): atomic { c = l; l=true }
```

Solution :

```
bool lock = false;
active proctype P1() {
  bool cc;
  do :: true -> TS(lock,cc);
                do :: cc -> TS(lock,cc) od
                /* section critique */
                lock = false;
                /* section non-critique */
  od
}
active proctype P2() {
  bool cc;
  do :: true -> TS(lock,cc);
                do :: cc -> TS(lock,cc) od
                /* section critique */
                lock = false;
                /* section non-critique */
  od
}
```


Propriétés

Exclusion mutuelle : un seul processus voit lock à faux.

Absence de blocage : si les deux veulent entrer, lock est faux, donc l'un entrera.

Absence de délai : un processus entre dès que lock devient faux.

Absence de famine : si scheduling avec équité forte.

Généralisation naturelle au cas n-aire.

Problème Performances mauvaises à cause de l'accès exclusif répété à la variable partagé lock (*memory contention*).

Solution meilleure pour les caches mémoire :

```
do :: lock -> skip od;
TS(lock,cc);
do :: cc -> do :: lock -> skip od;
            TS(lock,cc)
od
```

Implémenter la synchronisation conditionelle atomique

```
atomic { B -> S }
```

C'est une attente active !

Une solution générale

```
protocole_entree_sc
do :: !B -> protocole_sortie_sc;
           protocole_entree_sc
od
S
protocole_sortie_sc
```

Problème : inefficace, car accès intense à la mémoire.
Solution : introduire un délai dans la boucle, exemple CSMA/CD.

Une solution particulière

si S est vide et B satisfait la propriété "Au plus 1"

```
do :: !B -> skip od
```


Algorithme de Peterson binaire

- = variante de la première solution non-primitive, mais avec opérations primitives.
- last introduit pour "casser" l'inter-blocage.

```
bool in1 = false, in2 = false;
int last = 1;
active proctype P1() {
  do :: true -> in1 = true; last = 1;
                do :: in2 && (last==1) -> skip od
                /* section critique */
                in1 = false;
                /* section non-critique */
  od
}
active proctype P2() {
  do :: true -> in2 = true; last = 2;
                do :: in1 && (last==2) -> skip od
                /* section critique */
                in2 = false;
                /* section non-critique */
  od
}
```

Propriétés

Exclusion mutuelle: par exclusion de configurations après avoir introduit des variables auxiliaires $mid1$ et $mid2$ comme suit (pour P1) :

```
atomic { in1 = true; mid1 = true };
atomic { last = 1; mid1 = false }
```

P1 est dans la section critique si

$$in1 \wedge \neg mid1 \wedge (\neg in2 \vee (last = 2) \vee mid2)$$

P2 est dans la section critique si

$$in2 \wedge \neg mid2 \wedge (\neg in1 \vee (last = 1) \vee mid1)$$

Les deux prédicats sont exclusifs, q.e.d.

Absence de blocage : le premier qui affecte last gagne.

Absence de délai : si in2 est faux, P1 entre sans attendre.

Absence de famine : si scheduling avec équité inconditionnelle.

Algorithme de Peterson n-aire

```
bool in[N] = ([N] -1);
int last[N] = ([N] -1);
active [N] proctype P() {
  int j, k;
  do :: true ->
    j = 0;
    do :: j < N-1 ->
          /* note que _pid est a l'etape j
           * et est le dernier y entrer */
          in[_pid] = j; last[j] = _pid;
          k = 0;
          do :: k < N && k != _pid ->
                /* attends tout k qui est devant */
                do :: in[k] >= in[_pid]
                      && last[j]==_pid -> skip od;
                k = k+1
             :: k == _pid -> k = k+1
          od; j = j+1
    od
    /* section critique */
    in[_pid] = -1;
    /* section non-critique */
  od
}
```

Algorithme du ticket n-aire

Solution non-primitive

```
int number = 1, next = 1;
int turn[N] = ([N] 0);
active [N] proctype P() {
  do :: true ->
    atomic { turn[_pid]=number; number=number + 1};
    atomic { turn[_pid]==next };
cs: /* section critique */
    atomic { next=next + 1 };
    /* section non-critique */
  od
}
```

Solution primitive

Utilise l'instruction machine “Fetch-And-Add :

```
FA(v,inc): atomic { temp=v; v=v+inc; return temp }
```

Solution :

```
    turn[_pid] = FA(number,1);
    do :: turn[_pid]!=next -> skip od
cs: /* section critique */
    next=next+1
```


Propriétés

Exclusion mutuelle : le prédicat TICKET est un invariant

$$\{ \quad (P[i]@cs) \Rightarrow (turn[i] = next) \wedge$$
$$(\forall i, j :\leq i, j < N, i \neq j : \quad (turn[i] = 0) \vee$$
$$(turn[i] \neq turn[j])) \ \}$$

Absence de blocage : les valeurs non-nulles de turn sont uniques

Absence de délai : number-next = nombre de processus en attente

Absence de famine : si scheduling avec équité faible

Problème de précision numérique : number et next sont non-bornés

Problème de matériel : si pas de FA (ou comparable), il faut utiliser une algorithme d'exclusion mutuelle !

Algorithme du boulanger (Lamport)

- calcul non-atomique d'un numéro d'ordre
- le pid du processus "casse" l'inter-blocage

```
int turn[N] = ([N] 0);
active [N] proctype P() {
  int j;
  do :: true ->
    turn[_pid]=1; turn[_pid]=max(turn[0:N-1])+1;
    j=0;
    do :: j < n ->
        do :: j!=_pid && turn[j]!=0 &&
               (turn[_pid],_pid) > (turn[j],j) -> skip
        od;
        j=j+1
    od
cs: /* section critique */
    turn[_pid] = 0;
    /* section non-critique */
  od
}
```


Prorpriétés

Exclusion mutuelle : l'invariant global BAKERY est :

$$
\begin{aligned}
(P[i]@cs) \Rightarrow \quad & (turn[i] \neq 0 \ \wedge (\forall j : 0 \leq j < N, j \neq i : \\
& \quad turn[j] = 0 \ \vee \\
& \quad turn[i] < turn[j] \ \vee \\
& \quad (turn[i] = turn[j] \ \wedge \ i < j)) \\
&)
\end{aligned}
$$

Absence de blocage : le pid des processus coupe les inter-blocages

Absence de délai : que les valeurs non-nulles de turn comptent

Absence de famine : si scheduling avec équité inconditionnelle

Problème de précision numérique : diminué, car augmentation des valeurs de turn moins importante

Matériel : pas d'instructions spéciales