Théorie et pratique de la concurrence

Mihaela Sighireanu

Mihaela.Sighireanu@liafa.jussieu.fr

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

Bureau 6A7, Chevaleret

Sections critiques

— problème et solutions classiques —

Problème des sections critiques

= implémentation d'une synchronisation par exclusion mutuelle

Schéma général :

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

Hypothèses :

- tout processus qui entre dans sa section critique va forcement sortir
- un processus peut se terminer ou boucler que dans la section non-critique
- les variables utilisées par les protocoles d'entrée et sortie ne sont pas utilisées ailleurs

Propriétés souhaitées

- **Exclusion mutuelle** : au plus un processus exécute sa section critique à un moment donné.
- **Absence de interblocage** : si deux ou plusieurs processus essayent d'entrer dans leur sections critiques, au moins un réussira.
- **Absence de délai** : si un processus P essaye d'entrer en section critique et les autres processus exécutent leur sections non-critiques ou sont terminés, alors le processus P 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.

Vers une solution avec deux processus

... et que des instruction atomiques élémentaires.

Premier essai

```
int turn = 1;
active proctype P() {
  1:do :: true -> section_non_critique
  2:             (turn == 1); // protocole d'entrée
  3:             section_critique
  4:             turn = 2 // protocole de sortie
  od
}
active process Q() {
...
}
```

- utilise une attente active sur une condition (inefficace)
- variable turn indique quel processus détient la ressource

Nous allons utiliser les diagrammes d'états pour prouver :

1. la propriété d'exclusion mutuelle
2. l'absence de d'interblocage
3. la présence de famine (et donc de délai)

Deuxième essai

Idée : éviter la famine en testant une variable par process.

```
bool wantP = 0, wantQ = 0;
active proctype P() {
  1:do :: true -> section_non_critique
  2:               (wantQ == 0);
  3:               wantP = 1;
  4:               section_critique
  5:               wantP = 0
  od
}
active process Q() {
...
}
```

Exercice : Utiliser les diagrammes d'états pour prouver l'absence d'exclusion mutuelle.

Question : Et si on inverse les lignes 2 et 3 ?

Troisième essai

Idée : on inverse les lignes 2 et 3 et on évite l'interblocage en étant poli...

```
bool wantP = 0, wantQ = 0;
active proctype P() {
  1:do :: true -> section_non_critique
  2:             wantP = 1;
  3:             do :: (wantQ == 1) ->
  4:                wantP = 0;
  5:                wantP = 1
                 od;
  6:             section_critique
  7:             wantP = 0
  od
}
active process Q() {
...
}
```

Exercices : utiliser les diagrammes d'états pour prouver :

1. la propriété d'exclusion mutuelle
2. l'absence de d'interblocage
3. la présence de famine (et donc de délai)

Algorithme de Dekker

Idée : combine le premier et le troisième essais.

```
bool wantP = 0, wantQ = 0;
int turn = 1;
active proctype P() {
  do :: true -> section_non_critique
                wantP = 1;
                do :: (wantQ == 1) ->
                   if :: turn == 2 ->
                           wantP = 0;
                           (turn == 1);
                           wantP = 1
                      :: else -> skip
                fi od;
                section_critique
                turn = 2
                wantP = 0
  od
}
```

- wantP et wantQ assurent l'exclusion mutuelle
- turn indique qui a le droit d'insister pour entrer dans la section critique

Tous les propriétés sont satisfaites (preuves page 18).

Instructions atomiques complexes

Leur implémentation (matérielle) assure l'atomicité.

Quelques exemples :

- instruction "Test-And-Set" :

```
TS(int common,int local):
  atomic { local = common; common=1 }
```

- instruction "Exchange" :

```
EXG(int a,int b):
  int temp;
  atomic { temp = a; a = b; b = temp }
```

- instruction "Fetch-and-Add" :

```
FA(int common,int local,int x):
  atomic { local = common; common = common + x }
```


Section critique avec TS

```
int common = 0;
active proctype P() {
  int localP = 1;
  1: do :: true -> section_non_critique
  2:      do :: localP == 1 ->
  3:           TS(common,localP)
          od;
  4:      section_critique
  5:      common = 0
  od
}
active proctype Q() {
  int localQ = 1;
  1: do :: true -> section_non_critique
  2:      do :: localQ == 1 ->
  3:           TS(common,localQ)
          od;
  4:      section_critique
  5:      common = 0
  od
}
```

Propriétés

Exclusion mutuelle : un seul processus voit common à 0.

Absence de blocage : si les deux veulent entrer, common est 0, donc l'un entrera.

Absence de délai : un processus entre dès que common devient 0.

Absence de famine : si scheduling avec équité forte.

Généralisation naturelle au cas n-aire.

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

Solution meilleure pour les caches mémoire :

```
do :: common==1 -> skip od;
TS(common,localP);
do :: localP==1 ->
     do :: common==1 -> skip od;
     TS(common,localP)
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
```

Sections critiques

— vérification —

Vérification de programmes

Problème : étant donné un programme P et une propriété ϕ, montrer que P satisfait ϕ, càd $P \models \phi$.

Propriétés de sûreté : Soit BAD le prédicat qui décrit l'état non-sûr. Il faut montrer que à chaque point de l'exécution du programme BAD est faux.

Propriétés de vivacité : Nécessite des hypothèses sur l'équité de l'ordonnanceur du processeur par rapport à toute actions éligibles (action qui peut être exécuté à un moment donné).

Classes de politiques d'ordonnancement selon leur équité :

- équité inconditionnelle : toute action atomique non-conditionnée sera exécutée.
- équité faible : équité inconditionnelle et toute action atomique conditionnelle éligible sera exécutée si sa garde devient vraie et elle reste vraie ensuite.
- équité forte : équité inconditionnelle et toute action atomique conditionnelle éligible sera exécutée si sa garde est infiniment souvent vraie.

Exercice équité

Préciser, pour chaque classe d'équité si les programmes concurrents ci-dessous terminent ou non :

```
bool continue = 1;
proctype Loop() { do :: continue -> skip od }
proctype Stop() { continue = 0 }
init { run Loop(); run Stop() }
```

```
bool continue = 1, try = 0;
proctype Loop() { do :: continue -> try=1;try=0 od }
proctype Stop() { atomic { try -> continue = 0 } }
init { run Loop(); run Stop() }
```


Méthodes de vérification

Preuve inductive : Trouver une propriété I invariante de P (càd I est vraie à chaque point d'exécution de P) et montrer que $I \rightarrow \phi$

- difficile à trouver I (mais des outils semi-automatiques existent)
- longue si P est très long

Preuve manuelle sur le diagramme d'états : Générer le diagramme d'états complet et le regarder pour démontrer la propriété.

- diagrammes d'états grands pour des vrais programmes
- humainement difficile à traiter

Preuve automatique sur le diagramme d'états : Donner le programme et la propriété à un outil (par exemple Spin) qui simultanément génère le diagramme d'états et vérifie la propriété sur ce diagramme.

- outils existants capables de traiter diagrammes avec 10^{32} états
- utilisent des logiques spéciales : logiques temporelles

Specification logique des propriétés

Propositions atomiques :

- prédicats sur les valeurs des variables : $wantP$, $turn \neq 1$
- prédicats sur les compteurs programme des processus : $P@1$

Formules : combinaison booléenne des propositions atomiques

$$P@1 \ \wedge \ Q@1 \ \wedge \ \neg wantP \ \wedge \ \neg wantQ$$

Propriété invariante : prouver que I est invariante en P demande :

- (Cas de base) I est vraie dans l'état initial de P
- (Pas d'induction) Si I est vraie dans tous les états jusqu'à l'état courant, alors I reste vraie après l'exécution des instructions possibles dans l'etat courant.

Preuve pour l'algorithme de Dekker

Théorème : la formule $\neg(P@8 \ \wedge \ Q@8)$ est invariante pour l'algorithme de Dekker.

Preuve : Utilise la lemme suivante :

Lemme : Les formules suivantes sont egalement invariantes :

- $turn = 1 \ \vee \ turn = 2$
- $P@3..5 \ \vee \ P@8..10 \Leftrightarrow wantP$
- $Q@3..5 \ \vee \ Q@8..10 \Leftrightarrow wantQ$

On montre que $P@8 \ \wedge \ Q@8$ est faux dans chaque état :

- (Cas de base) C'est trivialement vrai pour l'etat initial.
- (Pas d'induction) Supposons que $P@8 \ \wedge \ Q@8$ est fausse. Les instructions qui peuvent la changer sont aux lignes 3 de chaque processus, quand wantP et wantQ sont fausses. Mais si wantQ est faux, par la lemme, Q ne peut pas être à l'état 8, donc l'instruction à l'état 3 de P ne peut pas être exécutée si $Q@8$. QED.

Introduction à la logique temporelle LTL

Constatation : la logique classique peut exprimer que des propriétés sur les états mais pas sur les séquences d'exécution.

Solution : utiliser des opérateurs logiques spéciaux pour les propriétés des séquences d'exécution.

Linear Temporal Logic (Manna et Pnueli '80)

Soit $\sigma = s_0\ s_1 \ldots$ une séquence d'exécution de P avec s_0, s_1, ... états de P. Soit a une formule sur les états de P.

- (toujours) la formule $\Box a$ est vraie dans un état s_i ssi a est vraie dans tous les états $s_j, j \geq i$.
- (possible) la formule $\Diamond a$ est vraie dans un état s_i ssi a est vraie dans iun état $s_j, j \geq i$.

Exemples :$\sigma = bba^*$

- $\Box a$ est vraie à partir de s_2
- $\Diamond b$ est vraie en s_0 et s_1

Propriétés des opérateurs temporels

- Réflexivité : toujours et possible incluent le moment présent.
- Dualité : $\neg\Box a = \Diamond\neg a$ et $\neg\Diamond a = \Box\neg a$

Exemples intéressants

- exclusion mutuelle : $\Box\neg(P@8\ \wedge\ Q@8)$
- progrès d'un calcul en $P@1$: $\Box\Diamond\neg P@1$
- absence de famine avec équité faible : $P@4\ \wedge\ \Box(turn = 2) \Rightarrow \Diamond P@5$
- ou : $\Box wantP\ \wedge\ \Box(turn = 1) \Rightarrow \Diamond P@5$
- ou : $P@2 \Rightarrow \Diamond P@8$

Vérification basée sur les modèles (*model-checking*)

Entrées :

- modèle de l'algorithme P
- propriété souhaitée en logique temporelle ϕ

Résultats possibles :

- Valide : si toutes les séquences d'exécution de P satisfont ϕ
- Invalide : s'il existe une séquence de P qui ne satisfait pas ϕ
- Aucun : si le diagramme d'états est trop gros

Outils : Spin, CADP, ...

Vérification avec Spin

Développé par G. Holzmann à Bell Labs.

Site Web : `spinroot.com`

Statut : logiciel libre.

Langage de modélisation : Promela.

Logique temporelle : LTL.

Utilisation :

- Lancer Spin : `xspin` (interface Tcl/tk)
- Éditer le modèle Promela et vérifier sa syntaxe (Run, Run Syntax Check).
- Ouvrir l'éditeur de propriétés et écrire la propriété (Run, LTL Property Manager)
- Vérifier la propriété (Generate puis Run Verification).
- Simuler le modelé (Run, Set Simulation Parameters, Guided, Simulate)

Sections critiques

— autres solutions —

Algorithme de Peterson binaire

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

```
bool wantP = 0, wantQ = 0;
int last = 1;
active proctype P() {
    do :: true -> section_non_critique
                  wantP = 1;
                  last = 1;
                  do :: wantQ && (last==1) -> skip od
                  section_critique
                  wantP = 0;
  od
}
active proctype Q() {
  ...
}
```

Propriétés

Exclusion mutuelle: par preuve de propriété invariantes ou en utilisant Spin.

Preuve par induction : on peut montrer que P est dans la section critique si

$$wantP \wedge \neg P@3 \wedge (\neg wantQ \vee (last = 2) \vee Q@3)$$

Pareil pour Q. Comme les deux prédicats sont exclusifs, QED.

Absence de blocage : le premier qui affecte last gagne.

Absence de délai : si wantQ est faux, P 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 -> /* section non-critique */
    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;
  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 -> /* section non-critique */
    atomic { turn[_pid]=number; number=number + 1};
    atomic { turn[_pid]==next };
cs: /* section critique */
    atomic { next=next + 1 };
  od
}
```

Solution primitive

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

```
FA(c,l,x): atomic { l=c; c=c+x }
```

Solution :

```
    FA(number,turn[_pid],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

Pour deux processus

```
int np=0, nq=0;
active proctype P() {
  do :: true ->
p1:   /* section non-critique */ skip;
p2:   np = nq+1;
p3:   ((nq==0) || (np <= nq));
p4:   /* section critique */ skip;
p5:   np = 0
  od
}
active proctype Q() {
...
}
```

Pour prouver l'exclusion mutuelle et l'absence de famine on utilise les invariants :

$$np = 0 \iff (p1 \lor p2)$$
$$nq = 0 \iff (q1 \lor q2)$$
$$p4 \Rightarrow (nq = 0) \lor (np \leq nq)$$
$$q4 \Rightarrow (np = 0) \lor (nq < np)$$

Algorithme du boulanger pour N processus

```
int number[N] = ([N] 0);
active [N] proctype P() {
  int j;
  do :: true ->
p1:  /* section non-critique */
p2:  number[_pid]=1 + max(number[0:N-1]);
p3:  j=0;
p4   do :: (j < N) && (j<>_pid) ->
p5:    ((number[j]==0) ||
        ((number[_pid],_pid) < (number[j],j)));
p6:    j=j+1
p7:    :: (j == _pid) -> j=j+1
     od
p8:  /* section critique */
p9:  number[_pid] = 0
  od
}
```

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

Propriétés

Exclusion mutuelle : l'invariant global BAKERY est :

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

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

Section critique avec "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 --+
```