Théorie et pratique de la concurrence

Mihaela Sighireanu

Mihaela.Sighireanu@liafa.jussieu.fr

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

Bureau 6A7, Chevaleret

Concurrence et synchronisation

— notations, sémantique, propriétés —

Introduire la concurrence (1/4)

Notation historique : (Dijkstra 1965) **parbegin...parend**

```
x = 0; y = 0;
parbegin
   x = x + 1
|| y = y + 1
parend;
z = x + y
```

Notation en C : processus lourd **vfork()** ... **wait()**

```
x = 0; y = 0;
pid = vfork();
if (pid<0)
  { printf("Cannot fork!"); exit(1); }
else if (pid==0) /* son */
  { x = x + 1; _exit(0); }
else /* father */
  { y = y + 1; }
while (wait(&status) != pid)  /* await son */;
z = x + y;
```

Remarque : lacement de processus léger (*thread*) avec **exec**

Introduire la concurrence (2/4)

Notation en Promela : **init**, **run**

```
int x = 0, y = 0, z = 0;
proctype inc_x()
{ x = x + 1; z = z + x }
proctype inc_y()
{ y = y + 1; z = z + y }
init
{ run inc_x(); run inc_y() }
```

Remarques :

- fils d'exécutions sans synchronisation sur la terminaison
- **init** lance ses instructions dans un ordre aléatoire, donc $z \in \{0, 1, 2\}$ à la terminaison du programme :

```
int x = 0, y = 0, z = 0;
proctype inc_x() { x = x + 1 }
proctype inc_y() { y = y + 1 }
init {
  run inc_x(); run inc_y();
  z = x + y
}
```

Introduire la concurrence (3/4)

Première manière en Java : par héritage, masquer la méthode **run** de la classe **java.lang.Thread**

```
public class ExThreadRun extends Thread {
  protected static int x = 0;
  protected static int y = 0;
  protected static int z;
  ...
  public void run () {
    if (v)      x = x + 1;
    else        y = y + 1;
  }
  ...
  public static void main (String args[]) {
    Thread inc_x = new ExThreadRun (true);
    Thread inc_y = new ExThreadRun (false);

    inc_x.start (); // call run()
    inc_y.start (); // call run()
    while(inc_x.isAlive() || inc_y.isAlive());
    z = x + y;
```

- (+) simple,
- (-) code dans le contrôleur

Introduire la concurrence (4/4)

Deuxième manière en Java : implanter la méthode **run** de l'interface **java.lang.Runnable** :

```
public class ExRunnableRun implements Runnable {
  protected static int x = 0;
  protected static int y = 0;
  protected static int z;
  ...
  public void run () {
    if (v)     x = x + 1;
    else       y = y + 1;
  }

  public static void main (String args[]) {
    ExRunnableRun incX = new ExRunnableRun(true);
    Thread inc_x = new Thread (incX);
    ...
    inc_x.start ();
    ...
```

- (-) complexe,
- (+) permet l'héritage d'une autre classe,
- (+) code et contrôleur dissociés,
- (+) naturel de partager les champs d'un objet.

Actions atomiques

actions atomiques = transformations indivisibles de l'état

Pour un programme concurrent, l'axiome de l'affectation (C-1:p16) n'est plus valide, car non-atomique.

```
init { run inc_x(); run inc_y(); z = x + y }
```

Actions atomiques fines = implémentées dans le matériel :

- lecture et écriture de valeurs dans la mémoire/registres (load, store) pour les types simples
- opérations simples sur les registres

Hypothèses sur le matériel :

1. chaque processus a son propre ensemble de registres
2. les valeurs mémoire sont lues et modifiées dans les registres, puis écrites dans la mémoire
3. sauvegarde/restauration des registres à l'interruption d'un processus
4. résultats intermédiaires des opérations complexes dans les registres

Atomicité de l'affectation

Notations :

- $R(P, S)$ variables lues par l'instruction S de P
- $W(P, S)$ variables affectées par l'instruction S de P

Exemples de conditions suffisantes pour l'atomicité de $x = e$ de P_i:

Disjoint : $\forall j : j \neq i : R(P_i, e) \cap W(P_j, *) = \emptyset$

Au plus 1 : ("Disjoint" et x est simple) ou

(e fait référence à au plus une variable externe modifiée y, une seule fois, et x non lue par un autre processus)

Exemples :

```
x = 0; y = 0;
/* Disjoint */
parbegin x = x + 1 || y = y + 1 parend
/* Au plus 1 */
parbegin x = y + 1 || y = y + 1 parend
/* ??? */
x = 0; y = 0;
parbegin x = y + 1 || y = x + 1 parend
z = 0; y = 0;
parbegin x = y + z || y = 1; z = 2 parend
```

Introduire la synchronisation

Notations en Promela :

- exclusion mutuelle (ou atomicité) : **atomic** $\{ \ S \ \}$
- sur condition : $B \ \rightarrow \ S$

Variantes :

- synchronisation atomique sur condition : **atomic** $\{ \ B \ \}$
  autre formulation ??
- action atomique conditionnelle : **atomic** $\{ \ B \ \rightarrow \ S \ \}$

Règle d'inférence de la synchronisation :

$$\frac{\{ \ P \wedge B \ \} \ S \ \{ \ Q \ \}}{\{ \ P \ \} \ \textbf{atomic} \ \{ \ B \ \rightarrow \ S \ \} \ \{ \ Q \ \}}$$

Sémantique de la concurrence

Une assertion critique dans $\{ \ P \ \} \ S \ \{ \ Q \ \}$ est :

- Q et
- pour toute instruction T en S qui n'est pas dans une section atomique, le plus faible prédicat $pre(T)$ tel que $\{ \ pre(T) \ \} \ T \ \{ \ post(T) \ \}$ est un théorème de la preuve.

Une action d'affectation est une action atomique qui contient une ou plusieurs instructions d'affectation.

Une action d'affectation a n'interfère pas avec une assertion critique C si $NI(a, C) \ : \ \{ \ C \wedge pre(a) \ \} \ a \ \{ \ C \ \}$.

$\{ \ P_i \ \} \ S_i \ \{ \ Q_i \ \}$, $1 \leq i \leq n$ sont théorèmes sans interférence si

pour toute action d'affectation a dans S_i

pour toute assertion critique C dans la preuve de $S_j, \ j \neq i$

$NI(a, C)$ est un théorème

Règle d'inférence de la concurrence

$$\frac{\{\ P_i\ \}\ \ S_i\ \ \{\ Q_i\ \}, 1 \leq i \leq n \quad \text{sont théorèmes sans interférence}}{\{P_1 \wedge \cdots \wedge P_n\} \left(\begin{array}{l}\textbf{parbegin}\\ \ S_1 \,||\, \cdots \,||\, S_n\\ \textbf{parend}\end{array}\right) \{Q_1 \wedge \cdots \wedge Q_n\}}$$

Exercices :

1. Pour n processus, chacun contenant a actions d'affectation et c assertions critiques, le nombre de preuves de non-interférence à faire est ... ?
2. Prouver le théorème suivant :

$$\{x = 0 \wedge y = 0\} \left(\begin{array}{l}\textbf{parbegin}\\ \quad x = x + 1\\ ||\ \ y = y + 1\\ \textbf{parend}\end{array}\right) \{x = 1 \wedge y = 1\}$$

Techniques pour éviter les preuves de non-interférence

Variables disjointes : $RC(P_i) \cap W(P_j, *) = \emptyset, i \neq j$ où $RC(P_i)$ variables dans les assertions critiques de P_i, $RC(P_i) \subseteq R(P_i, *)$.

Invariant global : I pour un ensemble de processus est un prédicat sur les variables globales tel que :

- I est vrai au début de l'exécution de chaque processus
- I est invariant pour l'exécution de toute action d'affectation

Exemple : producteur/consommateur avec tampon partagé.

Introduire la synchronisation : car les assertions dans les instructions atomique ne sont pas critiques.

$$\{P\}\ \textbf{atomic}\{\ S_1;\ \ \{Q\}\ S_2\ \}\ \{R\}$$

P critique, Q non-critique.

Exemple : transfert bancaires avec audit.

Exemple : producteur/consommateur (1/2)

```
int buf, p=0, c=0;
{PC:c<=p<=c+1 & a[1:N]=A[1:N] & ((p=c+1)=>(buf=A[p]))}
proctype Prod() {
  int a[N];
  { IP: PC & p<=N }
  do :: p < N -> { IP & p<N }
                 atomic { (p == c) };
                 { IP & p<N & p=c }
                 buf = a[p+1];
                 { IP & p<N & p=c & buf=A[p+1] }
                 p = p+1
                 { IP }
  od
  { PC & p=N }
}
```


Exemple : producteur/consommateur (2/2)

```
proctype Cons() {
  int b[N];
  { IC: PC & c<=N & b[1:c]=A[1:c] }
  do :: c < N -> { IC & c<N }
                 atomic { (p > c) };
                 { IC & c<N & p>c }
                 b[c+1] = buf;
                 { IC & c<N & p>c & b[c+1]=A[c+1] }
                 c = c+1
                 { IC }
  od
  { IC & c=N }
}
```

Exemple : transfert bancaire avec audit

```
int compte[N];
proctype Transfert() {
  { compte[x]=X & compte[y]=Y }
  atomic { compte[x]-=100; compte[y]+=100 }
  { compte[x]=X-100 & compte[y]=Y+100 }
}
proctype Audit() {
  int total = 0, i=1;
  bool detourne = 0;
  { total=compte[1]+...+compte[i-1] }
  do :: i <= N ->
           { total=compte[1]+...+compte[i-1] & i<=N }
           total += compte[i]; i+=1
           { total=compte[1]+...+compte[i-1] }
  od
  { total=compte[1]+...+compte[N] & i=N }
  if :: total != CASH -> detourne = 1 fi
}
```


Prouver des propriétés de sûreté

Soit BAD le prédicat qui décrit l'état non-sûr.

Méthode générale : Soit $\{P\}\ S\ \{Q\}$ une preuve de S telle que P caractérise l'état initial. S satisfait la propriété de sûreté exprimé par $\neg BAD$ si, pour chaque assertion critique C de la preuve, $C \Rightarrow \neg BAD$.

Méthode par invariant : Soit $\{P\}\ S\ \{Q\}$ une preuve de S telle que P caractérise l'état initial et I est un invariant global dans cette preuve. S satisfait la propriété de sûreté exprimé par $\neg BAD$ si $I \Rightarrow \neg BAD$.

États exclusifs : Pour une preuve donnée, un processus ne peut pas être dans un état satisfaisant P quand un autre processus est dans un état qui satisfait Q si $P \wedge Q = F$

Prouver des propriétés de vivacité

Nécessite des hypothèses sur l'équité du *scheduling* du processeur (FIFO, round-robin, time-slicing, etc.).

Action éligible : peut être exécuté à un moment donné.

Classes d'équité pour les politiques de *scheduling* :

- 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 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() }
```