Théorie et pratique de la concurrence

Mihaela Sighireanu

Mihaela.Sighireanu@liafa.jussieu.fr

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

Bureau 6A7, Chevaleret

Introduction

Concurrence

Programme concurrent = deux ou plusieurs processus séquentiels qui coopèrent pour accomplir une tâche.

- en utilisant les mêmes variables (variables partagées)
- en échangeant des messages

Concurrence = abstraction du parallelisme

- parallelisme = exécution concomitante sur plusieurs processeurs
- concurrence = parallelisme potentiel, par entrelacement des exécutions

Exemples :

- SE : calcul et opérations d'entrées-sorties
- SE : multi-tâches, multi-thread
- multi-processeurs
- grappes d'ordinateurs
- réseau d'ordinateurs

Programmation concurrente = jeu

- règles = outils formels d'analyse
  - état du programme $\longrightarrow$ assertion (prédicat)
  - instruction $\longrightarrow$ transformateur de prédicat
- pieces = méchanismes d'un langage de programmation
  - sémaphore, region critique, moniteur, rendez-vous, réception/envoi message
- stratégies = paradigmes de programmation

Historique

Origines : années 1960 $\longrightarrow$ révolution technologique du matériel $\longrightarrow$ systèmes d'exploitations

Quelque noms :

- Edsger Dijkstra : 1965 exclusion mutuelle, 1968 sémaphores, 1972 class, ...
- C.A.R. Hoare : 1969 "Hoare logic", 1972 région critique, 1974 moniteur, 1978 CSP, ...
- Brinch Hansen : 1970 SE, 1972 région critique, 1978 moniteur, ...
- Leslie Lamport : 1974 exclusion mutuelle, 1977 logic and program properties, 1980 horloges logiques, ...
- Dekker, Petterson, Floyd, ...

Synchronisation

Coopération $\Rightarrow$ synchronisation

Rôle de la synchronisation = restreindre les exécutions possibles d'un programme concurrent à ceux qui sont désirables.

Exemples :

- synchronisation par exclusion mutuelle = l'accès à des régions critiques d'instructions est fait de manière exclusive
- synchronisation sur condition = un processus est mis en attente jusqu'une condition devient vraie

Problème important de la programmation concurrente : implémenter correctement la synchronisation

Abstraction des programmes concurrents

Permet de raisonner sur le comportement des programmes.

Éléments de l'abstraction :

- Instruction atomique = instruction effectuée sans interruption.
- Processus séquentiel = séquence d'instructions atomiques.
- Programme concurrent = ensemble fini de processus séquentiels.
- Exécution d'un programme concurrent = entrelacement arbitraire des actions atomiques des processus séquentiels qui respecte l'ordre imposé par chaque processus
- Calcul ou scénario = une séquence d'exécution

Variables :

- globales (partagées)
- locale aux processus

Exemple :

- Processus `p = p1;p2`
- Processus `q = q1;q2`
- Programme concurrent = $\{$`p,q`$\}$

Questions :

- Quelles sont les exécutions possibles ?
- Est-ce que `p2;p1;q1;q2` est un scénario ?

Remarque : pour un programme avec n processus, chacun ayant m actions atomique, le nombre de scénarios du programme est $(n * m)!/(m!)^n$.

Définition formelle des éxécutions

Exécution d'un programme concurrent = états et transitions entre les états

Etat d'un programme concurrent = tuple dont les éléments sont

- compteur de programme pour chaque processus
- une valeur pour chaque variable globale
- une valeur pour chaque variable locale

Transition entre deux états s_1 et s_2 = éxécution d'une instruction atomique au compteur programme de s_1 qui change l'état s_1 en s_2

Diagramme d'états = graphe orienté défini par :

- noeud initial = état initial des processus et des variables;
- à chaque état lui correspond un noeud du graphe;
- si s_1 est un état d'un noeud du graphe et il existe une transition de s_1 vers s_2, alors il existe un arc orienté du noeud qui contient s_1 vers le noeud qui contient s_2.

Instructions atomiques

Par abstraction :

Toute instruction d'affectation et l'évaluation d'une expression booléenne.

Remarque importante : abstraction non réaliste mais utile.

Exemples qui contredisent cette abstraction :

- calculs complexes d'expressions
- variables non-atomiques = variables sur plusieurs mots de mémoire
- optimisation du compilateur

Pour obtenir l'atomicité :

- déclaration de type volatile pour une variable
- synchronisation par exclusion mutuelle sur un calcul

Propriétés des programmes

= attributs vrais pour toute trace possible du programme.

Deux classes spéciales :

- sûreté (*safety*) = “le programme ne passe jamais dans un état indésirable”
  - correction partielle = si un programme termine, alors il calcule le résultat attendu
  - exclusion mutuelle = un seul processus exécute une section critique à un moment donné
  - absence de blocage = dans tout état on peut exécuter une action
- vivacité (*liveness*) = “le programme passera forcement dans un état désirable”
  - terminaison = toute trace d’un programme est finie
  - absence de famine = tout processus peut entrer dans la section critique

Théorème : Toute propriété d’un programme peut être exprimée en termes de propriétés de sûreté ou de vivacité.

Démontrer des propriétés

Choix 1 : tester et déboguer

- usuel
- méthode partielle

Choix 2 : tester exhaustivement

- méthode automatisable = *model-cheking* (SPIN)
- demande l’utilisation de logiques *temporelles*
- peut exploser

Choix 3 : analyse par raisonnement mathématique

- représentation compacte, proportionnelle à la taille du programme
- méthode non-automatisable

Propriétés de vivacité et Equité

Prouver des propriétés de vivacité demande 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 le programme concurrent ci-dessous termine ou non :

- une variable globale :

```
bool continue = 1;
```

- un processus `Loop` qui execute :

```
while (continue) ;
```

- un autre processus `Stop` qui execute :

```
continue = 0;
```

Et pour les processus suivants ?

```
bool continue = 1, try = 0;
Loop() { while (continue) { try=1;try=0 } }
Stop() { if(try) continue = 0; }
```

Processus séquentiels
— construction et analyse —

Langage de programmation (1/2)

Types de base : **bit** ou **bool**, **byte**, **short**, **int**, **mtype**

```
mtype = { ack, nak, err };
```

Déclaration de constantes : macro-définition C

```
#define N 10
```

Déclaration de variables : C-like

```
bool in1, in2;
int turn;
byte ticket[N];
```

Déclaration de processus/procédures : **proctype**

```
proctype A (byte state; short foo)
{ (state == 1) -> state = foo }
```

avec instanciation dans le processus **init** :

```
init
{ run A (2, 3) /*; run A (1, 5) */ }
```

Langage de programmation (2/2)

Instructions :

- vide : **skip**
- affectation : **=**

```
ticket[i] = state + 1
```

- séquencement : **;** ($\neq$ avec C)

```
x = x+1; y=y+1
```

- sélection : **if**

```
if
:: x >= y -> m = x
:: y >= x -> m = y
fi
```

- répétition : **do**

```
do
:: x > y -> x = x-y
:: y > x -> y = y-x
/* :: x == y -> break */
od
```


Exemple de preuve de programmes séquentiels

On veut prouver $\{P\}\ S\ \{LS\}$ où

- P est $n > 0 \wedge (\exists j : 0 \leq j < n : a[j] = x)$
- S est

```
i=0;
do
:: a[i] != x -> i=i+1
od
```

- LS est $a[i] = x \wedge (\forall j : 0 \leq j < i : a[j] \neq x)$

Exemple d'esquise de preuves dans la LP

$\{\ P:\ n>0\ \wedge\ (\exists\ j:\ 0\leq\ j<n:\ a[j]=x)\ \}$
`i = 0;`
$\{\ P\ \wedge\ i=0\ \}$
$\{\ I:\ P\ \wedge\ (\forall\ j:\ 0\leq\ j<i:\ a[j]\neq x)\ \}$
do
`:: a[i] != x ->` $\{\ I\ \wedge\ a[i]\neq x\ \}$
`i=i+1`
$\{\ I\ \}$
od
$\{\ I\ \wedge\ a[i]=x\ \}$
$\{\ LS:\ a[i]=x\ \wedge\ (\forall\ j:\ 0\leq\ j<i:\ a[j]\neq x)\ \}$

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
```

Programmation en C : processus léger (*thread*, mémoire commune) : **pthread**

```
#include <pthread.h>
pthread_t tid[2]; /* identificateurs de Pthreads */
int x = 0, y = 0;
void *inc(void *arg)
{ (*(int*)arg) = (*(int*)arg) + 1; return NULL; }
int main()
{ printf("Processus %d lance\n", getpid());
  if (pthread_create(tid, NULL,inc,&x) == 0)
    printf("Pthread 1 cree\n");
  if (pthread_create(tid, NULL,inc,&y) == 0)
    printf("Pthread 2 cree\n");
  sleep(10); // atente fin threads, ou appel pthread_join
  printf("x=%d, y=%d\n",x,y); }
```


Introduire la concurrence (2/4)

Notation en Promela :

active proctype ou **init** et **run**

```
int x = 0, y = 0, z = 0;
active proctype inc_x()
{ x = x + 1; z = z + x }
active proctype inc_y()
{ y = y + 1; z = z + 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;
  private bool v;  ...
  public void run () {
    if (v)     x = x + 1;
    else       y = y + 1;
  }  ...
  public static void main (String args[]) {
    Thread inc_x = new ExThreadRun (true); // v = true
    Thread inc_y = new ExThreadRun (false); // v = false
    inc_x.start (); // call run()
    inc_y.start (); // call run()
    while(inc_x.isAlive() || inc_y.isAlive()); // or p.join()
    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;
  private bool v; ...
  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 dans la vie réelle

actions atomiques = transformations indivisibles de l'état

Pour un programme concurrent général, l'axiome de l'affectation n'est plus valide, car non-atomique.

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

En Promela, les affectations sont atomiques.

Comment obtenir l'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 (x non lue par un autre processus et e fait référence une seule fois à au plus une variable y modifiée dans 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 : B;
  autre formulation ??
- action atomique conditionnelle : **atomic** $\{ B \rightarrow S \}$

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 }
}
```