Parallélimse

merci: JJ Lévy, L Marchal

Mémoire partagée, exclusion mutuelle

Programmes parallèles

- un programme est *parallèle* s'il y a *plusieurs flots d'exécution*
  (par opposition à un programme séquentiel)
- les programmes parallèles s'expriment 'naturellement' comme composition *(mise en parallèle)* de programmes séquentiels (ici, impératifs)

$$\left(\begin{array}{l}\texttt{x := y+1;}\\ \texttt{z := x;}\end{array}\right) \quad \| \quad \left(\begin{array}{l}\texttt{t := f(u);}\\ \texttt{t' := 52;}\end{array}\right)$$

  (langage impératif imaginaire)
- deux "directions"
  - composition séquentielle ;
  - composition parallèle ||
- exemple élémentaire: paralléliser pour accélerer

```
a := x+32; y := t*52; b := u*t; k := k+1;
```

  est remplacé par

```
( (a := x+32;|| j := t;) ; (y := j*52;|| b := u*j) ) ||  k := k+1;
```

Modèle, interaction

- mémoire partagée: les flots qui s'exécutent en même temps peuvent accéder à une partie commune de la mémoire
  - plusieurs processeurs sur la même machine
  - des threads s'exécutant sur un même processeur
  - généralement le compilateur traduit les variables en des adresses mémoire *globales*
  - ...: on reste vague sur les considérations matérielles
- interaction:
  - au début, $x = 0$
  - puis on exécute `(x := x+1; x := x+1;)` || `x := 2*x;`
  - à la fin, $x \in \{2, 3, 4\}$ *les deux composantes peuvent interagir en x*

Atomicité

- atomicité:
  - au début, `x`$=0$
  - après l'exécution de `x := x+1;` || `x:= x+1;`, on a `x`$=2$
  - mais si `x := x+1;` est compilé en `R := x; x:= R+1;`
    BOUM
- la description des exécutions d'un programme parallèle suppose la définition de ce qu'est l'*atome*, i.e. l'opération indivisible dans un flot: entre le début et la fin d'un événement atomique, rien ne peut se passer dans les autres flots
  - typiquement, le matériel définit l'atome: un accès mémoire est atomique, pas une multiplication sur un format avec grande précision

Sections critiques, exclusion mutuelle

- deux portions de programmes accédant à des ressources communes

  $P_0 = \ldots C_0 \ldots \qquad P_1 = \ldots C_1 \ldots$

  - C_0 et C_1 sont des *sections critiques*
  - on veut "rendre ces sections *atomiques*", pour éviter les interférences
  - exclusion mutuelle: C_0 et C_1 ne s'exécutent jamais en même temps *(ne font jamais la course)*
  - on a intérêt à réduire la taille de C_0 et C_1, afin de ne pas trop réduire le parallélisme
- première solution: au début, `turn = 0`

```
while turn <> 0 do (* rien *) ;      while turn <> 1 do (* rien *) ;
C0                                   C1
turn := 1;                           turn := 0;
```

  - NB: il se peut que l'on doive passer *plusieurs fois* dans C_0 et/ou C_1
  - pas très satisfaisant: non *équitable*

D'autres tentatives

- au début, `a0 = a1 = false`

```
while a1 do (* rien *) ;          while a0 do (* rien *) ;
a0 := true ;                      a1 := true ;
C0;                               C1;
a0 := false;                      a1 := false;
```

  - pas très correct
- au début, `a0 = a1 = false`

```
a0 := true;                       a1 := true;
while a1 do (* rien *) ;          while a0 do (* rien *) ;
C0;                               C1;
a0 := false;                      a1 := false;
```

  - pas terrible, risques d'*interblocage*: chaque flot attend l'autre

Algorithme de Dekker (1965)

- au début, `a0 = a1 = false`, `turn`$\in \{0,1\}$

```
a0 := true;                          a1 := true;
while a1 do                          while a0 do
  if turn <> 0 then begin              if turn <> 1 then begin
    a0 := false;                         a1 := false;
    while turn <> 0 do (* rien *) ;      while turn <> 1 do (* rien *) ;
    a0 := true;                          a1 := true;
  end;                                 end;
C0                                   C1
turn := 1; a0 := false;              turn := 0; a1 := false;
```