Parallélisme, suite

Exécutions, entrelacement, équité

- on peut décrire les exécutions d'un programme en considérant tous les *entrelacements* possibles d'atomes: l'ordre est fixé au sein d'un flot, pas entre les flots *(cf. threads)*
- diverses formes de *synchronisation* peuvent être mises en œuvre pour contrôler l'ordre d'exécution entre les flots
- l'*ordonnancement* détermine 'qui a droit à un tour de manège' parmi les flots susceptibles de s'exécuter à un moment donné
  - cela peut être un programme ou un oracle
- une version de l'équité:
  si à partir d'un certain moment une instruction est susceptible d'être exécutée *en continuation*, alors elle le sera un jour *(+ de nombreuses déclinaisons)*

Algorithme de Peterson, preuve

- au début, `a0 = a1 = false`

$\neg a0 \wedge c0 \neq 2$	$\neg a1 \wedge c1 \neq 2$
`a0 := true ; c0 := 2;`	`a1 := true; c1 := 2;`
$a0 \wedge c0 = 2$	$a1 \wedge c1 = 2$
`turn := 1; c0 := 1;`	`turn := 0; c1 := 1;`
$a0 \wedge c0 \neq 2$	$a1 \wedge c1 \neq 2$
`while a1 and turn <> 0 do (* rien *);`	`while a0 and turn <> 1 do (* rien *);`
$a0 \wedge c0 \neq 2 \wedge (\neg a1 \vee turn = 0 \vee c1 = 2)$	$a1 \wedge c1 \neq 2 \wedge (\neg a0 \vee turn = 1 \vee c0 = 2)$
C_0	C_1
`a0 := false;`	`a1 := false;`
$\neg a0 \wedge c0 \neq 2$	$\neg a1 \wedge c1 \neq 2$

- sûreté
  - `c0, c1`: compteurs de programme *(variables fraîches)*
  - on a $(turn = 0 \vee turn = 1) \wedge a0 \wedge c0 \neq 2 \wedge (\neg a1 \vee turn = 0 \vee c1 = 2) \wedge a1 \wedge c1 \neq 2 \wedge (\neg a0 \vee turn = 1 \vee c0 = 2)$
    ssi
    $(turn = 0 \vee turn = 1) \wedge turn = 0 \wedge turn = 1$ impossible
- progression: personne n'attend indéfiniment

Primitives pour la synchronisation

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;
```

- "algorithme", ou *protocole*
- il y a de l'*attente active*: un flot interroge constamment la valeur de `turn` tandis que l'autre s'exécute
- si l'ordonnancement est *équitable*, l'algorithme l'est

Algorithme de Peterson (1981)

- au début, `a0 = a1 = false`

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

- plus compact et simple à justifier que l'algorithme de Dekker
- attente active encore

Le sujet d'examen de l'an dernier est en ligne

Il est en ligne

Sémaphores

- un *sémaphore généralisé* est une variable entière `s` avec deux opérations *(sémaphore tout court: `s=1`)*
  - `acquire(s)`:
    - si `s`>0 alors `s := s-1` — de manière atomique
    - sinon rester bloqué sur `s`
  - `release(s)`:
    - si un flot est bloqué sur `s`, le réveiller
    - sinon `s := s+1` — de manière atomique
- l'exclusion mutuelle devient naturelle: on part de `s=1`

```
...acquire(s); C0; release(s);...
   ||  ...acquire(s); C1; release(s);...
```

- néanmoins, risques d'*interblocage*
  - si A verrouille $s1$, B verrouille $s2$, puis A veut $s2$ et B veut $s1$
  - discipline courante: un ordre partiel *(relation réflexive transitive antisymétrique)* sur les sémaphores, respectée au sein de chaque flot

Primitives pour la programmation concurrente

- Modula-3, librairies (POSIX threads, ...)
  - on pense à des *threads* (*tâches*) s'exécutant sur un ensemble (éventuellement singleton) de processeurs
- entités
  - un *mutex* est libre, ou verrouillé par un thread; naît libre
  - une *condition* est un ensemble de threads en attente; naît vide
- fonctions
  - `acquire(m)`: attend que le mutex `m` soit libre et le verrouille
  - `release(m)`: déverrouille le mutex `m`
  - `wait(m,c)`: déverrouille le mutex `m` et attend sur la condition `c` atomiquement; ensuite revérouille `m` et rend la main
  - `signal(c)`: un ou plusieurs threads attendant la condition `c` peuvent redémarrer
  - `broadcast(c)`: *tous* les threads attendant la condition `c` peuvent redémarrer

Mutexes – pile concurrente

- idiome classique de manipulation des mutex

```
VAR nonEmpty := new(condition);
```

  - pop

```
acquire(m);
while p = NIL do wait(m,nonEmpty);
topElement := p.head;
p := p.next;
release(m);
```

  - push

```
acquire(m);
p := newElement(v,p);
signal(nonEmpty);
release(m);
```

- Rq: il vaut mieux mettre un `while` qu'un `if` dans 'pop'

Les cinq philosophes chinois (1)

- cinq philosophes ($\Phi_0, \ldots, \Phi_4$) qui pensent et mangent, cinq baguettes à disposition sur une table ronde
- problème dû à Dijkstra pour tester les primitives concurrentes: verrous, conditions, sémaphores, sémaphores généralisés
- première solution: un tableau `s` de mutex

```
PROCEDURE Philosophe (i) =
BEGIN
  (* penser *)
acquire(s[i]);
acquire(s[(i+1) % 5]);
  (* manger *)
release(s[i]);
release(s[(i+1) % 5]);
END Philosophe;
```

- risques d'interblocage

Philosophes chinois (2)

- deuxième solution

```
PROCEDURE Philosophe (i) =
BEGIN
while true do begin
  (* penser *)
  PrendreBaguettes(i);
  (* manger *)
  RelacherBaguettes(i);
end;
END Philosophe;
```

- on se donne les tableaux suivants
  - `f`, avec `f[i]` le nombre de fourchettes disponibles pour Φ_i
  - `manger`, un tableau de conditions

Philosophes chinois (3)

```
PROCEDURE PrendreBaguettes(i) =
BEGIN
acquire(m)
  while f[i] != 2 do
    wait (m, manger[i]);
  f[(i-1) % 5] := f[(i-1) % 5]-1;
  f[(i+1) % 5] := f[(i+1) % 5]-1;
release(m);
END PrendreBaguettes;
```

```
PROCEDURE RelacherBaguettes(i) =
BEGIN
VAR g := (i-1)%5, d := (i+1)%5;
acquire(m);
  f[g] := f[g]+1; f[d] := f[d]+1;
  if f[d] = 2 then
    signal(manger[d]);
  if f[g] = 2 then
    signal(manger[g]);
release(m);
END RelacherBaguettes;
```

- l'invariant $\Sigma_{i=0}^{4} f[i] = 10 - 2 * \mathit{mangeurs}$ est vérifié
- si interblocage, alors $\mathit{mangeurs} = 0$, et donc $\forall i.\, f[i] = 2$ donc pas d'interblocage pour le dernier à demander à manger
- *famine* possible, si par exemple Φ_1 et Φ_3 complotent contre Φ_2, qui mourra de faim

Philosophes chinois (4)

- on reprend la première solution + sémaphore généralisé `salle` au début `salle = 4`
  - pour manger, les philosophes rentrent dans la salle
  - il y a au plus 4 philosophes dans la salle
  - ils sortent de la salle après le repas
  - et retournent penser dans leur cellule (ce sont en fait des moines philosophes)

```
PROCEDURE Philosophe (i) =
BEGIN
  (* penser *)
acquire(salle) ; (* ------ début zone critique ---- *)
acquire(s[i]);
acquire(s[(i+1) % 5]);
  (* manger *)
release(s[i]);
release(s[(i+1) % 5]);
release(salle) ; (* ------ fin zone critique ---- *)
END Philosophe;
```

Philosophes chinois (5)

- 4 philosophes au plus dans la salle $\Rightarrow$ pas d'interblocage
- l'invariant suivant est vérifié:
  salle + nombre de tâches dans la zone critique = 4
- si Φ_i exécute `acquire(s[i])`, alors il finira cette instruction
- si Φ_i attend indéfiniment sur `acquire(s[(i+1)%5])`, alors Φ_{i+1} attend indéfiniment sur `acquire(s[(i+2)%5])`
- si Φ_i exécute `acquire(s[(i+1)%5])`, alors il finira cette instruction
- pas de famine

Morceaux de code synchronisable

- C# (dans l'infrastructure .NET) ou Java fournissent des `monitors` *(constructions pour programmer la synchronisation)*
- exemple: un canal à une place

```
monitor channel {
  int contents
  boolean full := false
  condition snd
  condition rcv

  function send(int sent) {
    while full do wait(rcv)
    contents := sent
    full := true
    signal(snd)
  }
```

```
  function receive() {
    var int received

    while not full do wait(snd)
    received := contents
    full := false
    signal(rcv)
    return received
  }
}
```

  - une sorte de *module pour l'exclusion mutuelle*
  - le mutex est implicite (`wait(rcv)`, `wait(snd)`)
  - rien ne garantit qu'un `monitor` est écrit correctement
    - il doit y avoir un *invariant*, vérifié à chaque fois qu'on relâche le mutex *(ici, `full` ssi une valeur est stockée)*

Passage de messages

Un autre modèle

- montons encore d'un niveau: on laisse tomber le modèle de la mémoire partagée, les flots se *communiquent* des données à l'aide de primitives
- on travaille en fait volontiers avec des *architectures hybrides*: un réseau connectant des machines ayant chacune plusieurs processeurs
  - localement: mémoire partagée
  - entre les machines: communications

Diverses formes d'interaction à distance

- communications point à point (entre 2 machines)
  - envois asynchrones (les canaux sont des files)
  - envois synchrones: l'envoi est bloquant
- le standard le plus utilisé est MPI (*Message Passing Interface*)
  - librairies (Fortran, C, C++) – on 'programme l'interaction'

```
send( dest, type, address, length)
```

· `dest` is an integer identifier representing the process to receive the message.
· `type` is a nonnegative integer that the destination can use to selectively screen messages.
· `(address, length)` describes a contiguous area in memory containing the message to be sent.

- des langages: Erlang, Occam
- invoquer un calcul à distance: RPC (Remote Procedure Call), la RMI (Remote Method Invocation) de Java
  - ex. de mise en œuvre: Corba – mettre en relation des serveurs de calcul qui proposent des méthodes
- NB: l'interaction n'a pas nécessairement lieu à distance
  - il existe des utilisations monoprocesseur des librairies pour le passage de messages *(si le code se conçoit naturellement comme ça)*

La librairie Event de Caml

- adaptation de la librairie de J. Reppy, développée pour SML
  - Concurrent ML, qui a notamment été utilisé pour développer le système de fenêtres X eXene
- les événements sont l'entité élémentaire pour programmer la synchronisation *(cette vision se marie bien avec ML)*
- interface:

```
type 'a channel
val new_channel :  unit -> 'a channel
type 'a event
val send :  'a channel -> 'a -> unit event
val receive :  'a channel -> 'a event
val choose :  'a event list -> 'a event
val wrap :  'a event -> ('a -> 'b) -> 'b event
val sync :  'a event -> 'a
val select :  'a event list -> 'a
...
```

- exemples: DÉMO `events.ml, cell[1,2,3].ml`

Il n'a pas été question ici ...

- de différentes architectures pour les parallélismes
  du parallélisme matériel (au sein d'un processeur) aux supercalculateurs, et aux approches en réseau (peer to peer, la grille, etc.)
- des questions algorithmiques et/ou pragmatiques liées à l'exécution de calculs bestiaux (parallélisme de données, questions d'équilibrage de charge, etc.)
- de la loi d'Amdahl:

$$\text{amélioration maximum} = \frac{N}{b*N+(1-b)} \qquad \begin{cases} b & \text{part séquentielle du pg.} \\ N & \text{nombre de processeurs} \end{cases}$$

  - questions de découpage des problèmes, de passage à l'échelle,...
- cf. en particulier
  - ASR2, Eddy Caron, L3IF.2, MIF
  - algorithmique parallèle, Yves Robert, MIF

Vers davantage d'abstraction

- couches sédimentaires

...
notions 'de haut niveau'
canaux, événements, ...
passage de messages
librairies, RMI, etc
primitives de synchronisation
sémaphores, mutexes, moniteurs, etc
mémoire partagée
programmer la synchronisations, busy waiting loops, etc

- en particulier, le modèle d'architecture sous-jacent est *déterminant* *(un bout de la lorgnette)*
- de nombreux défis
  - *construire/maintenir* des applications parallèles/distribuées
  - *gérer/maîtriser/prouver* le parallélisme à toutes les échelles
    (p.ex. parallélisme dans les processeurs)
  - prog: *unifier* les modèles de programmation?

Programmation parallèle

- les programmes parallèles sortent "beaucoup" du cadre séquentiel
  - non déterminisme
  - non terminaison (p.ex. un serveur)
  - des irrégularités apparaissent
- comprendre (*définir*) "ce que fait" un programme parallèle est difficile
  - le parallélisme peut être une manière d'écrire un programme (calcul) séquentiel
  - on peut aussi vouloir développer des applications intrinsèquement parallèles *(sans référence séquentielle)*

- R. Milner (très à peu près)
  "pour le séquentiel, on est guidé par la logique;
  pour le parallèle (l'interaction), on est face à la réalité"