LINFO1104 TP8: Programmation concurrente et systèmes multi-agents

Cette séance introduit la concurrence comme technique de programmation, et son utilisation pour l'implémentation de systèmes multi-agents. Le monde réel étant concurrent, la programmation concurrente le modèle plus naturellement et s'y intègre mieux.. En outre, la concurrence est nécessaire pour tirer pleinement profit des processeurs multi-core.

Enoncés des exercices

1. Expliquez ce qui se passe lors de l'exécution du code suivant:

```
declare A B C D
thread D = C+1 end
thread C = B+1 end
thread A = 1 end
thread B = A+1 end
{Browse D}
```

   - Dans quel ordre les fils d'exécution (*threads*) sont-ils créés?
   - Dans quel ordre terminent-ils leur exécution?
   - Dessinez la machine abstraite de ce programme après la création de tous les threads, mais avant l'exécution de leur contenu. C'est à dire, sans exécuter l'instruction Browse ni les affectations contenues dans les threads.
     Dans cet état, donnez toutes les instructions qui peuvent s'exécuter.
     Toutes les exécution de ce programme passent-elles forcément par cet état ? Si non, donnez un contre-exemple.

2. Analysez le code suivant et donnez les valeurs qui sont liées aux variables `X`, `Y` et `Z`, à la fin de l'exécution.

```
local X Y Z in
    thread if X==1 then Y=2 else Z=2 end end
    thread if Y==1 then X=1 else Z=2 end end
    X=1
end
```

   Faites de même pour le code suivant:

```
local X Y Z in
    thread if X==1 then Y=2 else Z=2 end end
    thread if Y==1 then X=1 else Z=2 end end
    X=2
end
```

   1. Vérifiez vos réponses à l'aide de Mozart. Bien sûr, vous devrez ajouter un appel à `Browse`. Dans quelle partie du code pouvez-vous le placer ?
   2. Dessinez pour chacun de ces deux programmes l'état de la machine abstraite à la fin de l'exécution par la sémantique, c'est à dire quand plus aucune instruction ne peut être exécutée.

3. **Producteur/Consommateur**. Un producteur est un morceau de code qui génère un flot (*stream*) de données, par exemple des entiers, en tournant dans son propre fil d'exécution. Un consommateur lit un flot dont il traite les données ; par exemple, il calcule la somme de tous les entiers dans le flot. Le consommateur tourne également dans son propre fil. En d'autres termes, le producteur et le consommateur sont des agents concurrents.

1. Implémentez une fonction `{ProduceInts N}` qui génère un flot contenant les `N` premiers entiers consécutifs, à partir de 1.
2. Implémentez la fonction `{Sum Str}` qui reçoit une liste d'entiers et qui renvoie la somme des ses éléments.
3. Connectez le producteur avec le consommateur en tant qu'agents concurrents, afin que chacun d'eux ait son propre fil d'exécution, de la manière suivante.

```
declare Xs S
thread Xs = {ProduceInts 666} end
thread S = {Sum Xs} end
{Browse S}
```

   Quelle est la différence avec le code suivant?

```
declare Xs S
Xs = {ProduceInts 666}
S = {Sum Xs}
{Browse S}
```

4. Considérez maintenant que `ProduceInts` prenne 1 seconde pour produire chaque entier, et que `Sum` prenne 1 seconde pour faire chaque addition. Combien de secondes au total prendra chacune des versions ci-dessus?

4. **Filtres** Nous pouvons également implémenter un consommateur qui soit aussi un producteur, comme un filtre. Soient, par exemple, un agent qui génère un flot de 10 000 entiers, un autre agent filtre les nombres impairs dans le flot et un dernier agent additionne les éléments du flot résultant. Le producteur, le filtre, et le consommateur fonctionnent dans des fils distincts, en formant un système multi-agent.

   La figure suivante décrit le montage. `Producer` est le processus générant le flot d'entiers. `Consumer` additionne les éléments filtrés. `Filter` est à la fois un producteur *et* un consommateur.

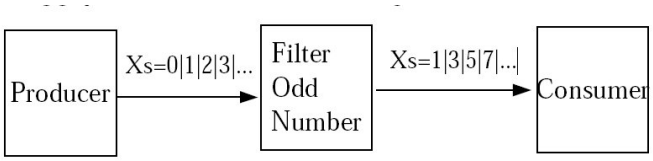

Figure 1:

1. Implémentez le programme décrit par la figure ci-dessus ; autrement dit, un filtre que ne garde que les nombres impairs.
2. Faites maintenant l'exercice suivant qui suit un schéma similaire. Charlotte a été une étudiante légendaire, renommée pour sa capacité à reconnaître une bière rien qu'à son odeur (et non pour une sombre histoire de carotte). Implémentez une fonction `{Barman N}` qui produise un flot de `N` bières, avec un intervalle de 3 secondes entre chaque bière. Implémentez Charlotte comme une fonction qui lit un flot de bières, sent chaque bière, et décide de boire uniquement les trappistes. Charlotte passe les autres bières à son ami qui, lui, boit tout ce qu'il reçoit. À la fin du programme, vous devez montrer séparément combien de bières Charlotte et son ami ont bu.

   Pour implémenter le barman, utilisez la fonction `{ServeBeer}` qui renvoie immédiatement une bière. Pour implémenter Charlotte, utilisez la fonction `{SmellTrappist Beer}` qui renvoie `true` ou `false` selon que `Beer` soit une trappiste ou non.

5. **Tracking information** Implémentez un programme qui permette de suivre l'apparition de caractères dans un flot. Le programme doit générer un flot de sortie où chaque position correspond aux caractères vu à ce moment. Chaque position du flot de sortie est une liste de tuples `C#N`, où `N` est le nombre de fois que `C` apparaît dans le flot d'entrée.

   Exemple:

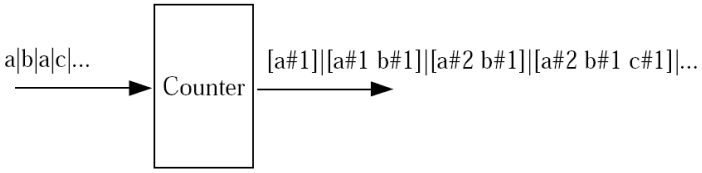

Figure 2:

   Notez que le `Counter` doit être un processus concurrent. En particulier, si la région suivante est exécutée,

```
local InS in
 {Browse {Counter InS}}
 InS=a|b|a|c|_
end
```

   le browser doit afficher `[a#1]|[a#1 b#1]|[a#2 b#1]|[a#2 b#1 c#1]|_`.

Exercice supplémentaire

1. **Simulation de la logique digitale**. Une porte logique est une fonction concurrente qui prend un ou plusieurs streams comme entrée. Les streams ne contiennent que des valeurs binaires (0 ou 1). Pour chaque éléments du stream, la porte applique une fonction logique et retourne un stream correspondant au calcul de la fonction. La porte la plus simple NotGate prend un stream en input et applique la négation de chaque éléments du stream.

   (a) Implémentez NotGate en commencant par définir Not, qui reçoit un nombre binaire et retourne sa négation.

   (b) Implémentez les portes logiques AndGate et OrGate. Ces portes prennent deux streams en input et retournent le résultat de l'application des opérations logiques `and` and `or` sur chaque pair d'inputs. La figure 3 montre un exemple d'une porte `and`.

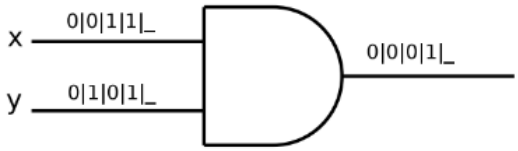

Figure 3: La porte logique `and`

   (c) Une porte logique à sortie unique peut se représenter sous la forme d'un arbre binaire : `gate(value:<operateur logique> <input 1> ... <input n>)`. Le nombre d'inputs varie en fonction de l'opérateur logique. Les inputs peuvent être soit des autres portes logiques ou bien `input(v)` où `v` est un stream binaire.

   Par exemple, la porte `NotGate` peut être représentée comme `gate(value:'not' input(x))`. La porte `and` de la figure 3 se représente par `gate(value:'and' input(x) input(y))`. La figure 4 représente un circuit plus complexe. Il peut être représenté par :

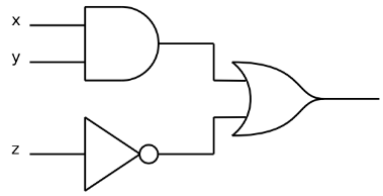

Figure 4: Composition de porte logique

```
gate(value:'or'
     gate(value:'and'
          input(x)
          input(y))
     gate(value:'not' input(z)))
```

Implémentez une fonction {Simulate G Ss} qui retourne la sortie du circuit correspondant à G sous les inputs Ss. Un exemple d'exécution : assumez que G est lié à l'arbre défini pour la figure 4. L'appel : {Simulate G `input(x: 1|0|1|0|_ y:0|1|0|1|_ z:1|1|0|0_)`} doit retourner `0|0|1|1|_`. Simulate doit être un processus concurrent. C'est-à-dire que plus l'input avance, plus l'output doit être calculé (l'input ne doit pas être totalement déterminé pour que la fonction commence à calculer l'output). Si le code suivant est exécuté :

```
declare Ss
{Browse {Simulate G Ss}}
Ss = input(x: 1|0|1|0|_ y:0|1|0|1|_ z:1|1|0|0_)
```

`0|0|1|1|_` doit apparaître dans le Browser.