

FSAB 1402: Informatique 2

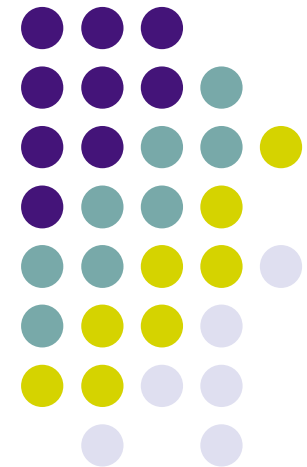
Concurrence et Systèmes Multi-Agents



Département d'Ingénierie Informatique, UCL

Peter Van Roy

pvr@info.ucl.ac.be





Ce qu'on va voir aujourd'hui

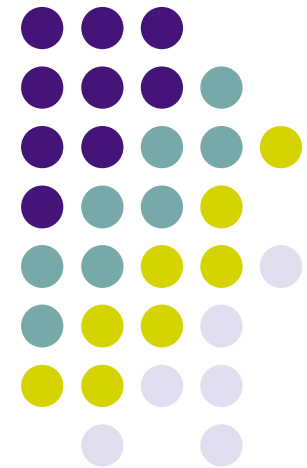
- La programmation concurrente
 - Motivation
 - L'exécution dataflow
 - Les fils ("threads")
 - Les flots de données
- Les systèmes multi-agents
 - Producteur/consommateur
 - Pipeline (comme dans Unix!)



Suggestions de lecture

- Chapitre 4 (introduction et section 4.1)
 - Les fils, leur sémantique
- Chapitre 4 (section 4.2)
 - Programmer avec les fils
- Chapitre 4 (sections 4.3)
 - Agents, flots, pipelines

Motivation pour la concurrence





Le monde est concurrent!

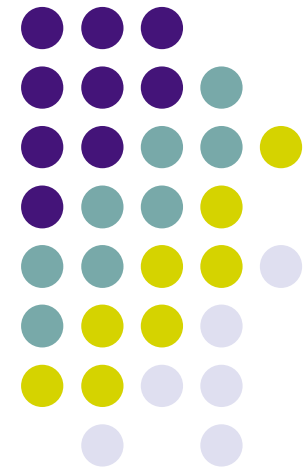
- Le monde réel est **concurrent**
 - Il est fait **d'activités qui évoluent de façon indépendante**
- Le monde informatique est concurrent aussi
 - **Systeme réparti**: ordinateurs liés par un réseau
 - Une activité concurrente s'appelle un **ordinateur**
 - **Systeme d'exploitation** d'un ordinateur
 - Une activité concurrente s'appelle un **processus**
 - Les processus ont des mémoires indépendantes
 - **Exécution d'un programme** dans un processus
 - Typiquement, dans les browsers Web chaque fenêtre correspond à une activité!
 - Une activité concurrente s'appelle un **fil**
 - Les fils partagent la même mémoire

La programmation concurrente



- La concurrence est naturelle
 - Deux activités qui sont indépendantes sont concurrentes!
 - Lien fort entre **indépendance** et **concurrence**
 - Comment faire un programme qui a deux activités indépendantes?
 - La concurrence doit être soutenue par les langages de programmation!
- Un programme concurrent
 - Plusieurs activités s'exécutent simultanément
 - Les activités peuvent communiquer et synchroniser
 - **Communiquer**: les informations passent d'une activité à une autre
 - **Synchroniser**: une activité attend une autre

Concurrence déclarative





Concurrence déclarative

- Il y a **trois manières principales** de programmer avec la concurrence
- La manière la plus simple s'appelle **la concurrence déclarative**
 - C'est ce que nous allons voir aujourd'hui
- Il y a deux autres manières principales (voir INGI1131)
 - **Concurrence par envoi de messages**
 - Les activités s'envoient des messages comme par courrier
 - Toujours relativement simple
 - **Concurrence par état partagé**
 - Les activités se partagent des données, elles essaient de travailler ensemble sans se marcher sur les pattes
 - Beaucoup plus compliquée!
 - Malheureusement, c'est ce que fait Java :-)

Une variable libre (non-initialisée)



- Une variable libre est créée en mémoire mais n'est pas encore affectée à une valeur
- Qu'est-ce qui se passe si on essaie de faire une opération avec une variable libre?

```
local X Y in
```

```
    Y=X+1
```

```
    {Browse Y}
```

```
end
```

- Qu'est-ce qui se passe?
 - **Rien!** L'exécution attend juste avant l'addition.

Que faire avec une variable non-initialisée?



- Différents langages font des choses différentes
 - **En C**, l'addition continue mais X contient une valeur "au hasard" (= le contenu de la mémoire à ce moment)
 - **En Java**, l'addition continue avec 0 comme valeur pour X (si X est l'attribut d'un objet avec type entier)
 - **En Prolog**, l'exécution s'arrête avec une erreur
 - **En Java**, il y a une détection d'erreur par le compilateur (si X est une variable locale)
 - **En Oz**, l'exécution attend juste avant l'addition et peut continuer quand X est lié (exécution dataflow)
 - **Dans la programmation par contraintes**, l'addition " $Y=X+1$ " est ajoutée à l'ensemble de contraintes et l'exécution continue! (voir cours "Programmation par Contraintes")

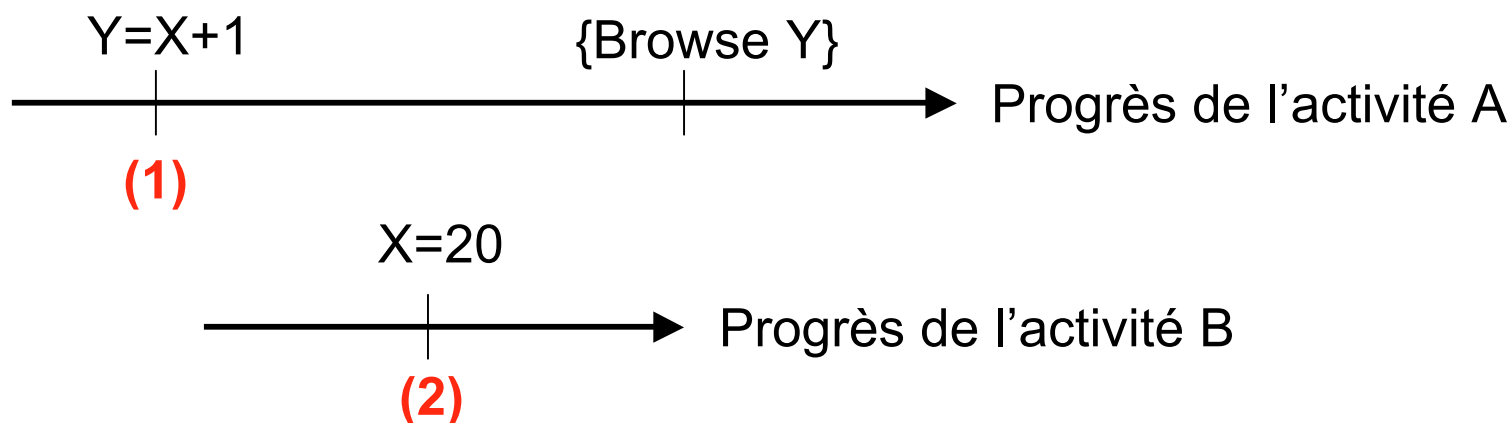


Faire continuer l'exécution

- L'instruction qui attend:
declare X
local Y **in**
 Y=X+1
 {Browse Y}
end
- Si quelqu'un d'autre pourrait lier X, alors l'exécution pourrait continuer!
- Mais qui peut le faire?
- Réponse: une autre activité concurrente!
- Si une autre activité fait:
 X=20
- Alors l'addition continuera et on affichera 21!
- Cela s'appelle de **l'exécution dataflow**

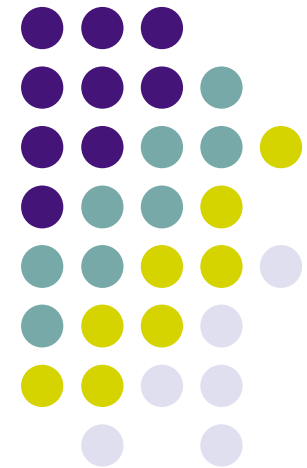


L'exécution dataflow



- L'activité A attend sagement au point (1) juste avant l'addition
- Quand l'activité B fait X=20 au point (2), alors l'activité A peut continuer
- Si l'activité B fait X=20 avant que l'activité A n'arrive au point (1), alors l'activité A n'attendra pas du tout

Fils (“threads”)





Fils (“threads”)

- Une “activité” est une séquence d’instructions en exécution
 - On appelle cela un fil
- Chaque fil est indépendant des autres
 - Entre deux fils il n’y a pas d’ordre
 - Le système garantit que chaque fil reçoit une partie équitable de la capacité de calcul du processeur
- Deux fils peuvent communiquer s’ils partagent des variables
 - Par exemple, la variable qui correspond à X dans l’exemple qu’on vient de voir



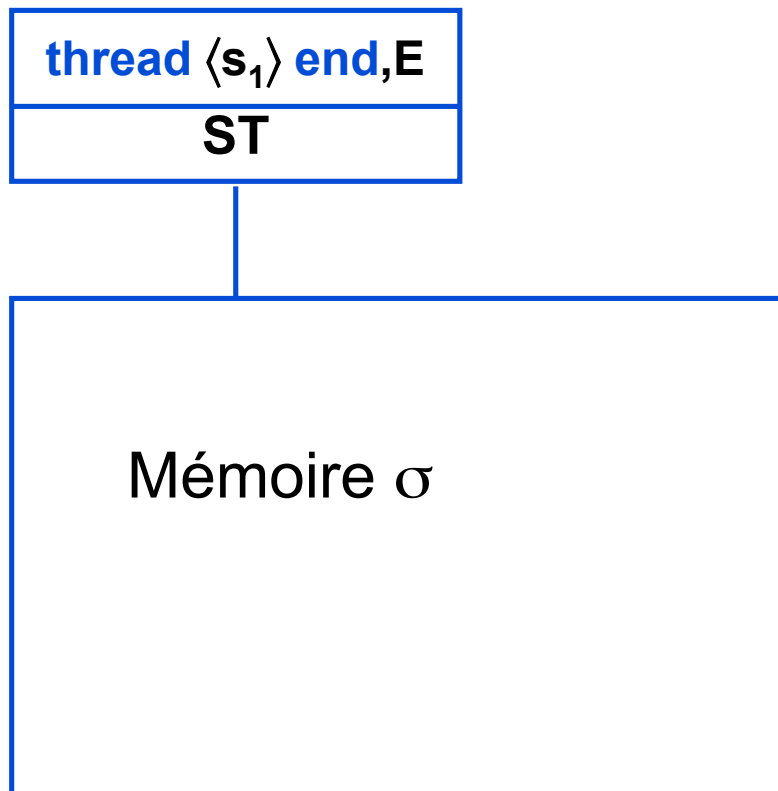
La sémantique des fils (1)

- Chaque fil correspond à une pile sémantique
- L'instruction **thread** <s> **end** crée une nouvelle pile sémantique
- Tous les fils se partagent la même mémoire



La sémantique des fils (2)

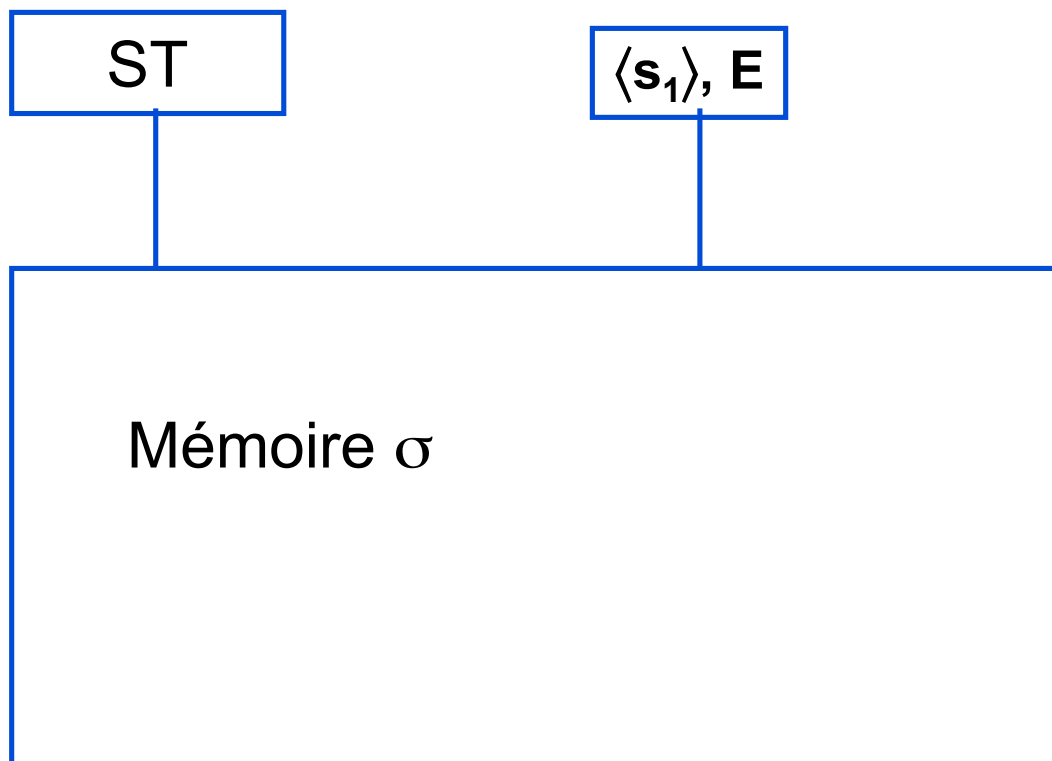
Une pile sémantique
avec une instruction
pour créer un fil





La sémantique des fils (3)

Deux piles!





La création des fils

- En Oz, la création d'un fil est simple
- On peut exécuter n'importe quelle instruction <s> dans un nouveau fil:
`thread <s> end`
- Par exemple:
`declare X`
`thread {Browse X+1} end`
`thread X=1 end`
- Qu'est-ce que fait ce fragment de programme?
 - Il y a plusieurs exécutions possibles, mais elles arrivent toutes au même résultat: tôt ou tard, on affichera 2!



Un petit programme (1)

- Voici un petit programme avec des fils:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Le Browser affiche [X0 X1 X2 X3]
 - Les variables ne sont pas encore affectées
 - Le Browser utilise aussi le dataflow: quand une variable est affectée, l'affichage est mis à jour



Un petit programme (2)

- Voici un petit programme avec des fils:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Les deux fils attendent
 - X1=1+X0 attend (X0 n'est pas affectée)
 - X3=X1+X2 attend (X1 et X2 ne sont pas affectées)



Un petit programme (3)

- Voici un petit programme avec des fils:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Nous faisons une affectation
 - Faites X0=4



Un petit programme (4)

- Voici un petit programme avec des fils:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Nous faisons une affectation
 - Faites X0=4
 - Le premier fil peut exécuter, il fait X1=5
 - Le Browser montre [4 5 X2 X3]



Un petit programme (5)

- Voici un petit programme avec des fils:

```
declare X0 X1 X2 X3  
thread X1=1+X0 end % fil terminé  
thread X3=X1+X2 end  
{Browse [X0 X1 X2 X3]}
```
- Le second fil attend toujours
 - Parce que X2 n'est toujours pas affecté



Un petit programme (6)

- Voici un petit programme avec des fils:

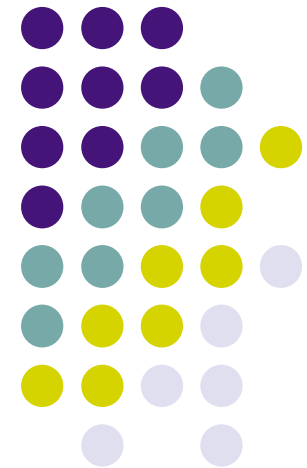
```
declare X0 X1 X2 X3  
thread X1=1+X0 end % fil terminé  
thread X3=X1+X2 end  
{Browse [X0 X1 X2 X3]}
```
- Nous faisons une autre affectation
 - Faites X2=7
 - Le second fil peut exécuter, il fait X3=12
 - Le Browser montre [4 5 7 12]



Le Browser est concurrent

- Le Browser exécute avec ses propres fils
- Pour chaque variable libre, il y a un fil dans le Browser qui attend sur cette variable
 - Quand la variable est affectée, l'affichage est mis à jour
- Attention: cela ne marche pas avec les cellules!
 - Le Browser ne regarde pas le contenu d'une cellule

Ordre total, ordre partiel et non-déterminisme





Exécution "simultanée"

- Avec la concurrence, on peut avoir plusieurs activités qui s'exécutent "en même temps"
- On peut imaginer que tous les fils s'exécutent vraiment en parallèle, chacun avec son propre processeur mais partageant la même mémoire
 - Ce n'est pas vraiment le cas, mais c'est une bonne manière de fixer les idées
- La lecture et l'écriture des variables et des cellules peuvent être faites simultanément dans différents fils
 - Lire une cellule peut être fait de façon simultanée
 - Ecrire une cellule est fait de façon séquentielle

Ordre des états d'exécution

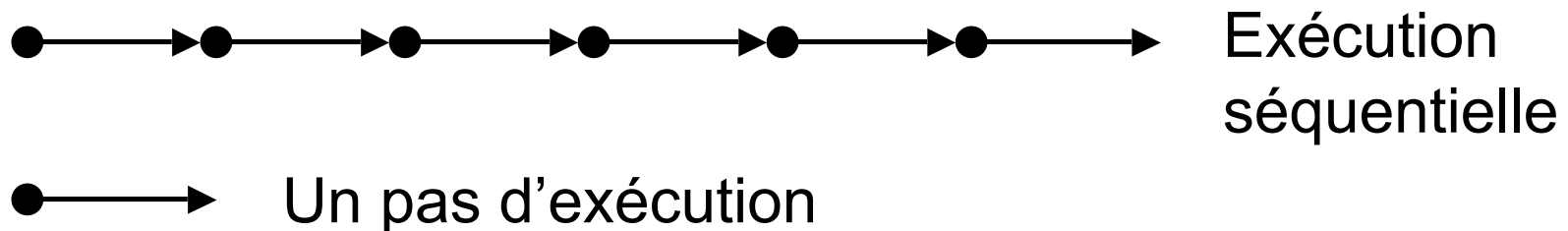


- Dans un programme séquentiel, tous les états d'exécution font un **ordre total**
 - Ordre total = il y a un ordre entre chaque paire d'états
- Dans un programme concurrent, tous les états d'exécution **du même fil** font un ordre total
 - Les états d'exécution du programme complet (avec plusieurs fils) font un **ordre partiel**
 - Ordre partiel = il y a un ordre entre une partie des paires d'états (certaines paires n'ont pas d'ordre entre eux)

Ordre total dans un programme séquentiel



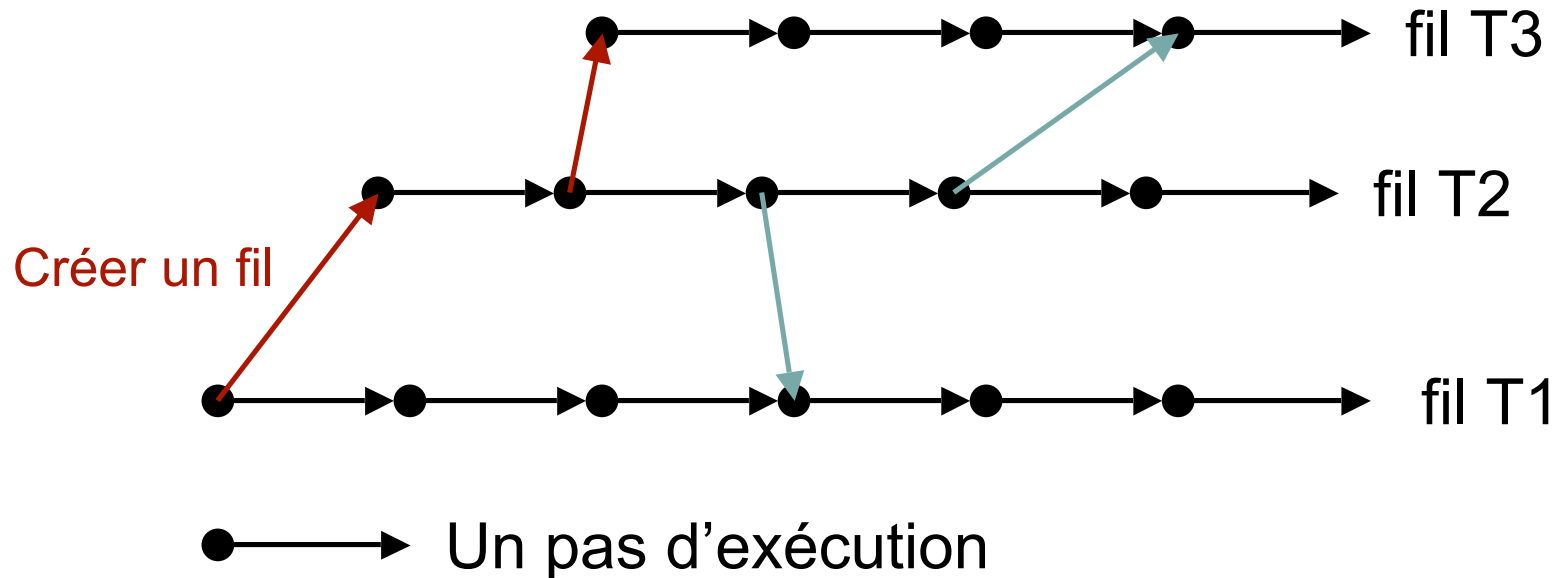
- Dans un programme séquentiel, tous les états d'exécution sont dans un **ordre total**
- Un programme séquentiel a **un fil** seulement
- C'est ce que nous avons vu jusqu'à aujourd'hui



Ordre partiel dans un programme concurrent



- Tous les états d'exécution du même fil sont dans un **ordre total**
- Les états d'exécution du programme complet (avec plusieurs fils) sont dans un **ordre partiel**



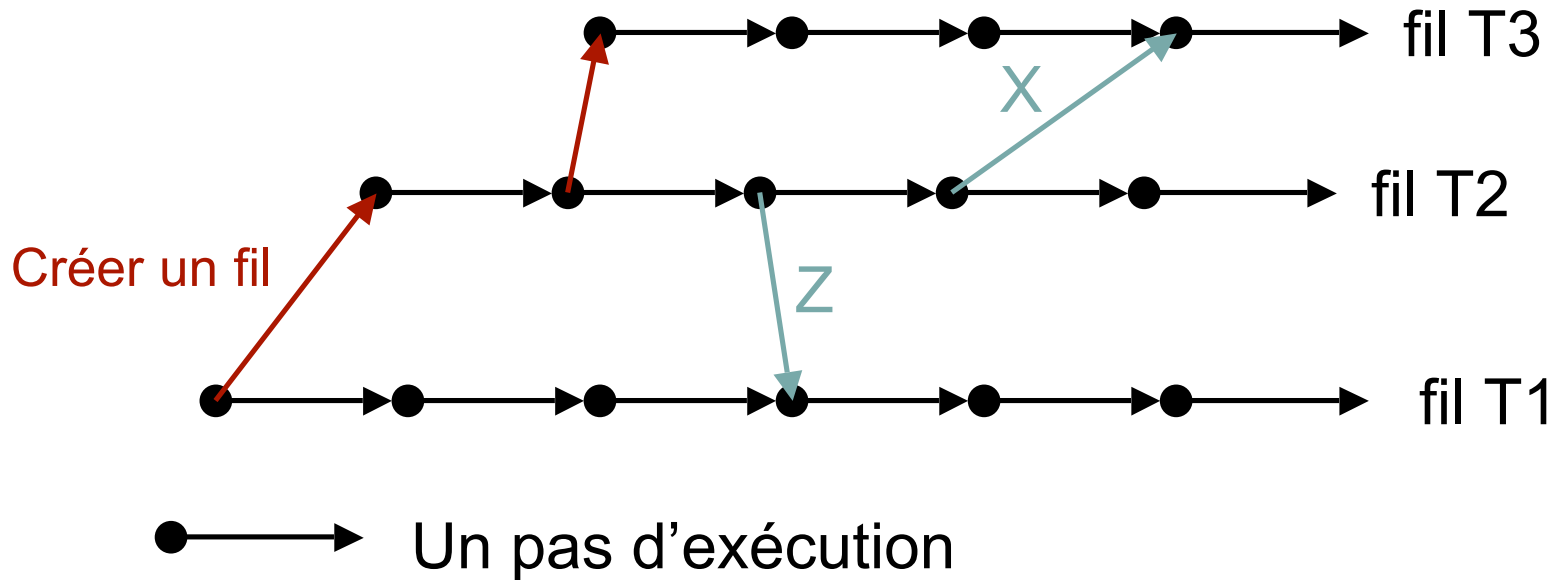
Ordre partiel dans un programme concurrent



Attendre la valeur d'une variable dataflow ("Y=X+1")

X

Lier une variable dataflow ("X=20")





Non-déterminisme (1)

- Qu'est-ce que fait le programme suivant?
`declare X`
`thread X=1 end`
`thread X=2 end`
- L'ordre d'exécution des deux fils n'est pas déterminé
 - X sera lié à 1 ou à 2, on ne sait pas à quelle valeur
 - L'autre fil **aura une erreur (une exception sera levée)**
 - On ne peut pas affecter une variable deux fois
- Cette incertitude s'appelle le **non-déterminisme**
 - L'exécution peut choisir l'une ou l'autre possibilité!



Non-déterminisme (2)

- Qu'est-ce que fait le programme suivant?
declare X={NewCell 0}
thread X:=1 **end**
thread X:=2 **end**
- L'ordre d'exécution des deux fils n'est pas déterminé
 - La cellule X sera affectée à une valeur, puis à l'autre
 - Quand les deux fils sont terminés, X aura le contenu 1 ou 2, on ne sait pas quelle valeur
 - Cette fois, il n'y a pas d'erreur
- Cette incertitude s'appelle le **non-déterminisme**



Non-déterminisme (3)

- En général, il faut **éviter le non-déterminisme**
 - Ce n'est pas toujours facile
 - C'est compliqué si on mélange les fils et les cellules (concurrence par état partagé)
 - Malheureusement, beaucoup de langages utilisent la concurrence par état partagé :-)
- Le modèle déclaratif a un avantage
 - **Le modèle déclaratif n'a pas de non-déterminisme** (sauf s'il y a une erreur comme dans l'exemple précédent)



L'ordonnancement des fils (1)

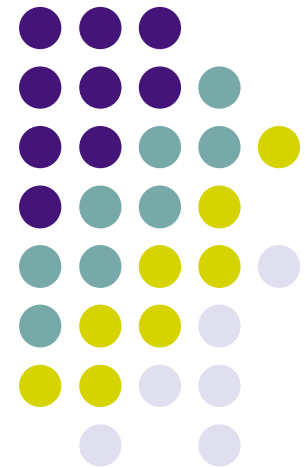
- Si le nombre de fils est plus grand que le nombre de processeurs (souvent le cas), alors les fils se partagent les processeurs
 - Chaque fil est exécuté pendant des courtes périodes qui s'appellent des **tranches de temps** ("time slices")
- Le choix du fil qui sera exécuté à chaque moment et pour combien de temps est fait par une partie du système qui s'appelle **l'ordonnanceur** ("scheduler")
- Un fil est **exécutable** ("runnable") si l'instruction au sommet de sa pile n'attend pas sur une variable dataflow. Sinon, le fil est **suspendu** ("suspended"), en d'autres mots **bloqué sur une variable** ("blocked on a variable")



L'ordonnancement des fils (2)

- Un ordonnanceur est **équitable** ("fair") si chaque fil exécutable sera tôt ou tard exécuté
 - En général, on donne des garanties sur le pourcentage du processeur qui est donné à tous les fils de la même **priorité**
- Si l'ordonnancement est équitable, on peut raisonner sur l'exécution des programmes
- Sinon, un programme parfaitement bien écrit peut ne pas marcher

Flots et agents





Flot (“stream”)

- Un **flot** (“stream”) est une liste dont l’extrémité est une variable libre
 - $S=a|b|c|d|S2$
 - Un flot peut être étendu avec des nouveaux éléments indéfiniment
 - On peut fermer le flot en terminant la liste avec nil
- Un flot peut servir comme un **canal de communication** entre deux fils
 - Le premier fil ajoute des éléments au flot
 - Le second fil lit le flot



Exemple d'un flot

- Voici un programme qui affiche tous les éléments d'un flot:

```
proc {Disp S}  
  case S of X|S2 then {Browse X} {Disp S2} end  
end  
declare S  
thread {Disp S} end
```

- On ajoute des éléments au flot:
 declare S2 **in** S=a|b|c|S2
 declare S3 **in** S2=d|e|f|S3
- Essayez par vous-même!



Producteur/consommateur (1)

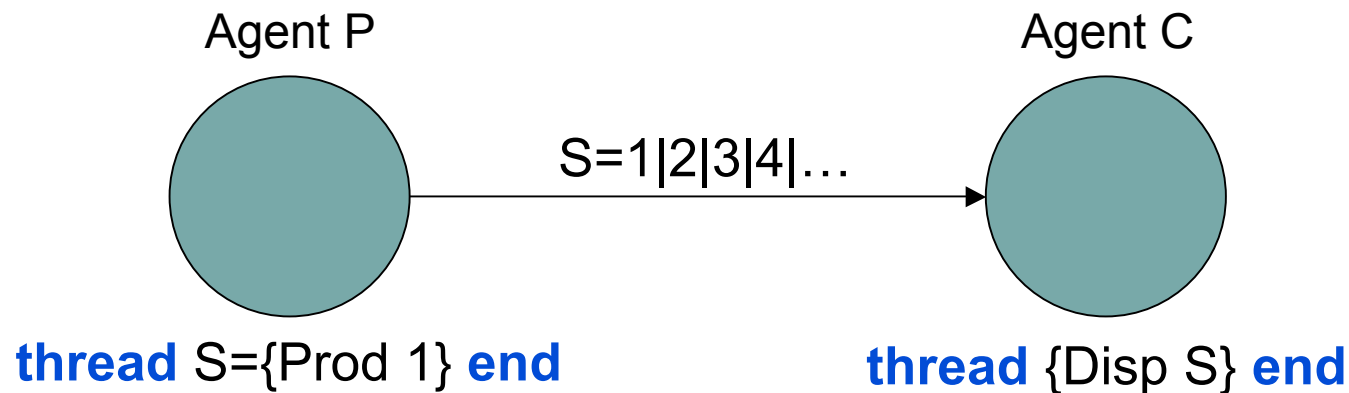
- Un **producteur** génère un flot de données

```
fun {Prod N} {Delay 1000} N|{Prod N+1} end
```

 - Le {Delay 1000} ralentit l'exécution pour qu'on puisse voir!
- Un **consommateur** lit le flot et fait quelque chose (comme la procédure Disp)
- Un programme producteur/consommateur:

```
declare S  
thread S={Prod 1} end  
thread {Disp S} end
```


Producteur/consommateur (2)



- Chaque cercle est une **activité concurrente avec un (ou plusieurs) canaux de communication**
 - On appelle ça aussi un **agent**
- Les agents communiquent par le flot S
 - Le premier fil crée le flot, le second le lit



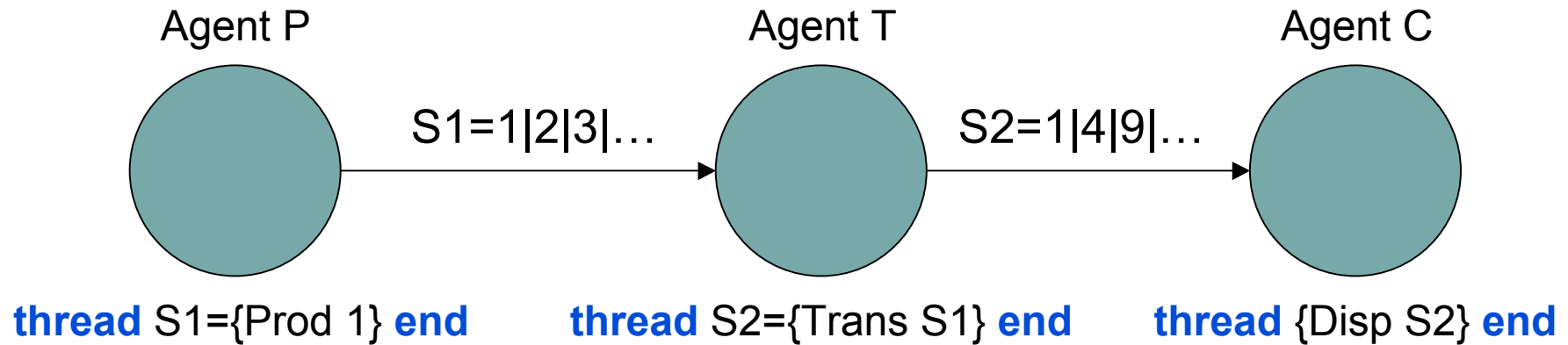
Pipeline (1)

- On peut ajouter d'autres agents entre P et C
- Voici un **transformateur** qui modifie le flot:

```
fun {Trans S}  
    case S of X|S2 then X*X|{Trans S2} end  
end
```
- Voici un programme avec trois agents:

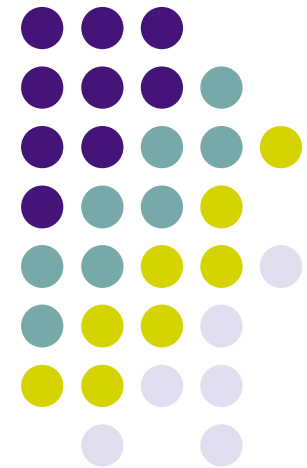
```
declare S1 S2  
thread S1={Prod 1} end  
thread S2={Trans S1} end  
thread {Disp S2} end
```

Pipeline (2)



- Nous avons maintenant créé trois agents
 - Le producteur (agent P) crée le flot S1
 - Le transformateur (agent T) lit S1 et crée S2
 - Le consommateur (agent C) lit S2
- La technique du pipeline est très utile!
 - Par exemple, il est **omniprésent en Unix**

Résumé





Résumé

- Concurrence
 - Activités qui évoluent de façon indépendante
 - Exécution dataflow avec variables libres
 - Non-déterminisme: incertitude sur quelle action sera exécutée
- Fil (“thread”)
 - Pour modéliser une activité concurrente dans le langage
 - Une séquence d’instructions en exécution
 - Dans la sémantique, c’est une pile sémantique
- Flot (“stream”)
 - Une liste dont l’extrémité est une variable libre
 - Un canal de communication entre agents
- Agent
 - Une activité concurrente avec un ou plusieurs canaux de communication
 - Programme multi-agent: producteur/consommateur, pipeline