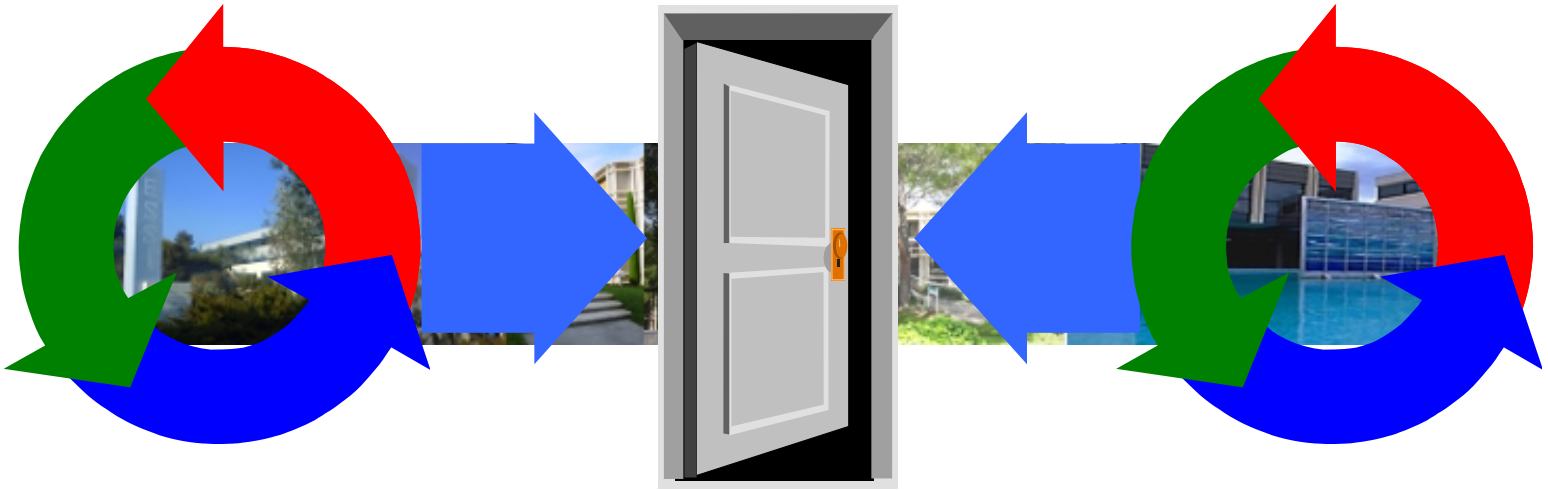
Mise en oeuvre de section critique

riveill@unice.fr
http://www.i3s.unice.fr/~riveill

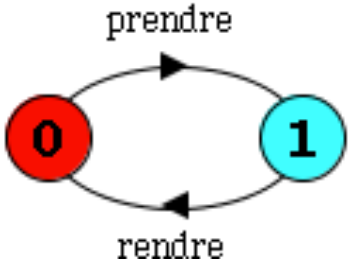


Université Nice Sophia Antipolis

Rappel du cours précédent
(mes fiches de révision)

Principaux éléments du langage FSP et leur représentation graphique

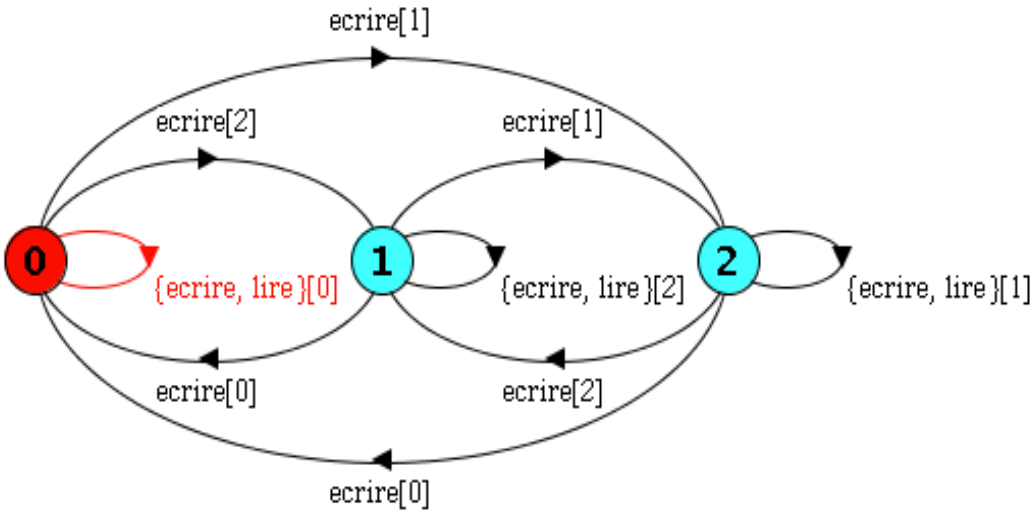
```
VERROU = (prendre -> rendre -> VERROU).
```



- Action en minuscule
- Processus en MAJUSCULE
- Séquence '->'
- Récursivité/boucle : on rappelle le même processus

```
VAR(N=2) = VAR[0],
VAR[i:0..N] = (lire[i] -> VAR[i]
      | ecrire[v:0..N] -> VAR[v]).
```

- Action indexée
- Alternative



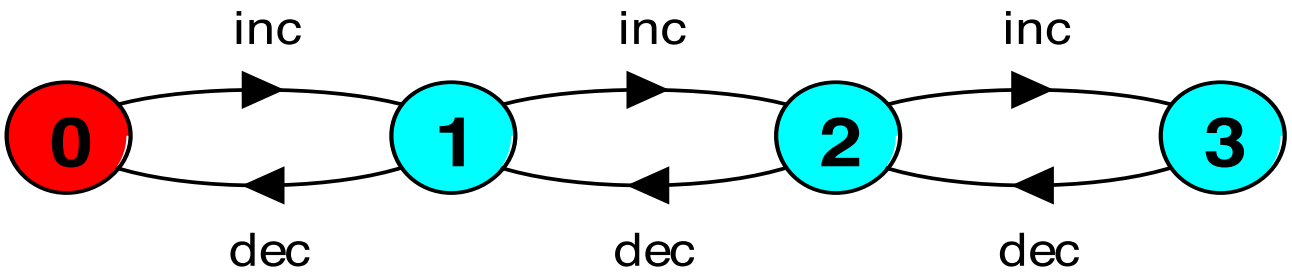
Université Nice Sophia Antipolis

i3S sophia antipolis

Principaux éléments du langage FSP

- Garde

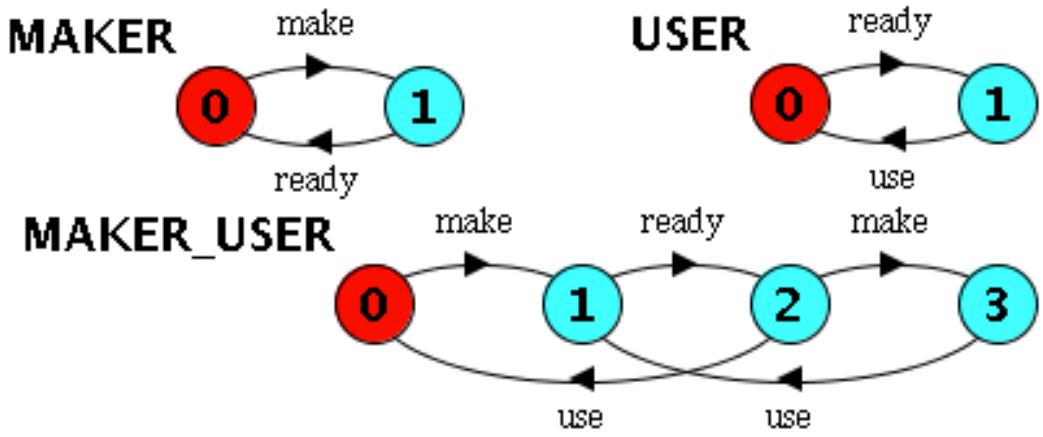
```
COUNT  (N=3)    = COUNT[0],
COUNT[i:0..N] = (when(i<N)  inc -> COUNT[i+1]
                |when(i>0)  dec -> COUNT[i-1]).
```



Principaux éléments du langage FSP

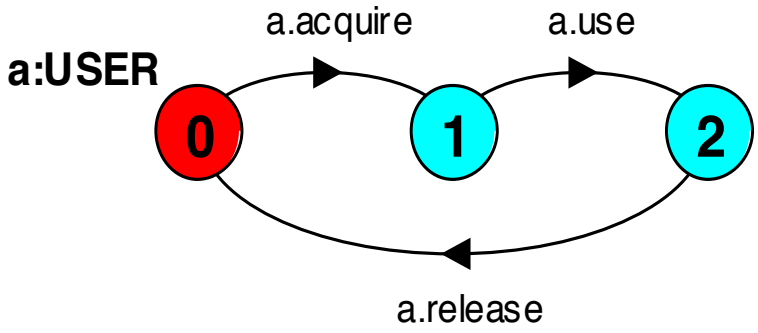
- **Parallélisme : ||**
  - **Commutative:** `(P||Q) = (Q||P)`
  - **Associative:** `(P||(Q||R)) = ((P||Q)||R) = (P||Q||R).`
  - **Les actions de noms différents sont indépendantes**
  - **Les actions de mêmes noms sont exécutées simultanément**

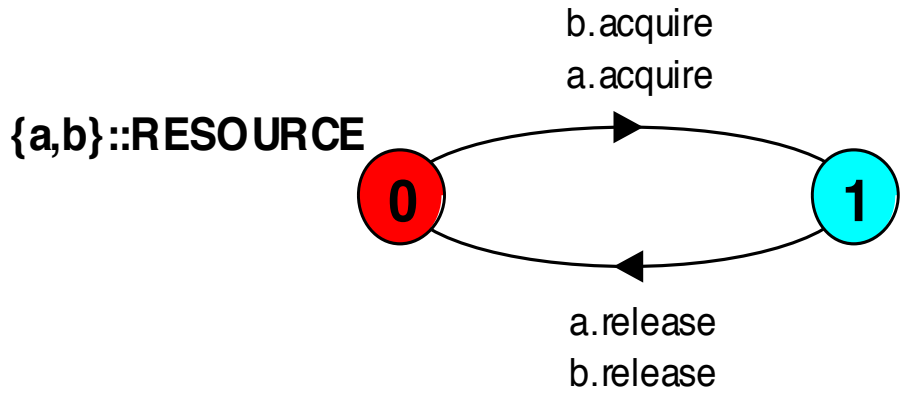
```
MAKER = (make->ready->MAKER).
USER  = (ready->use->USER).
||MAKER_USER = (MAKER || USER).
```

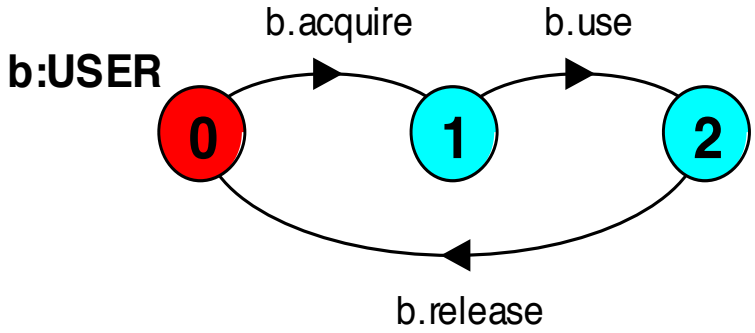


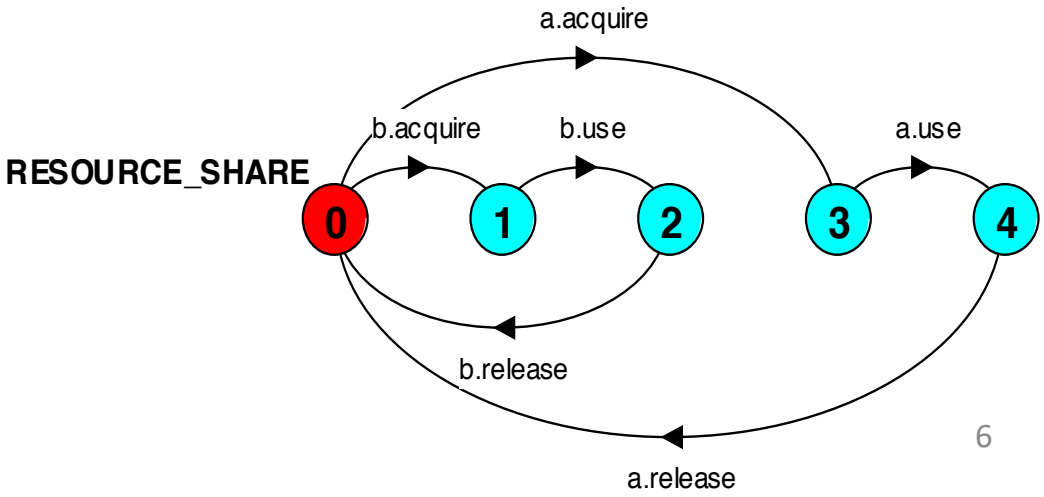
Préfixer un processus : ':' versus '::'

```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).
||RESOURCE_SHARE = (a:USER || b:USER
                    || {a,b}::RESOURCE).
```









Principaux éléments du langage FSP

- Renommage d'une action

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).

||CLIENT_SERVER = (CLIENT || SERVER)
          {call/request, reply/wait}
```

- Masquage d'une action

```
PHIL = (reflechir -> manger -> dormir -> PHIL)
          \{manger}.
```

  - Alphabet (PHIL) = *{dormir, prendre}*

- Exposition d'une action

```
PHIL = (reflechir -> manger -> dormir -> PHIL)
          @{reflechir, dormir}.
```

  - Alphabet (PHIL) = *{dormir, prendre}*

Section critique

Shared Objects & Mutual Exclusion

- **Concepts**: process interference.

  mutual exclusion refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time

- **Models**: model checking for interference

  modeling mutual exclusion

- **Practice**: thread interference in shared Java objects

  mutual exclusion in Java

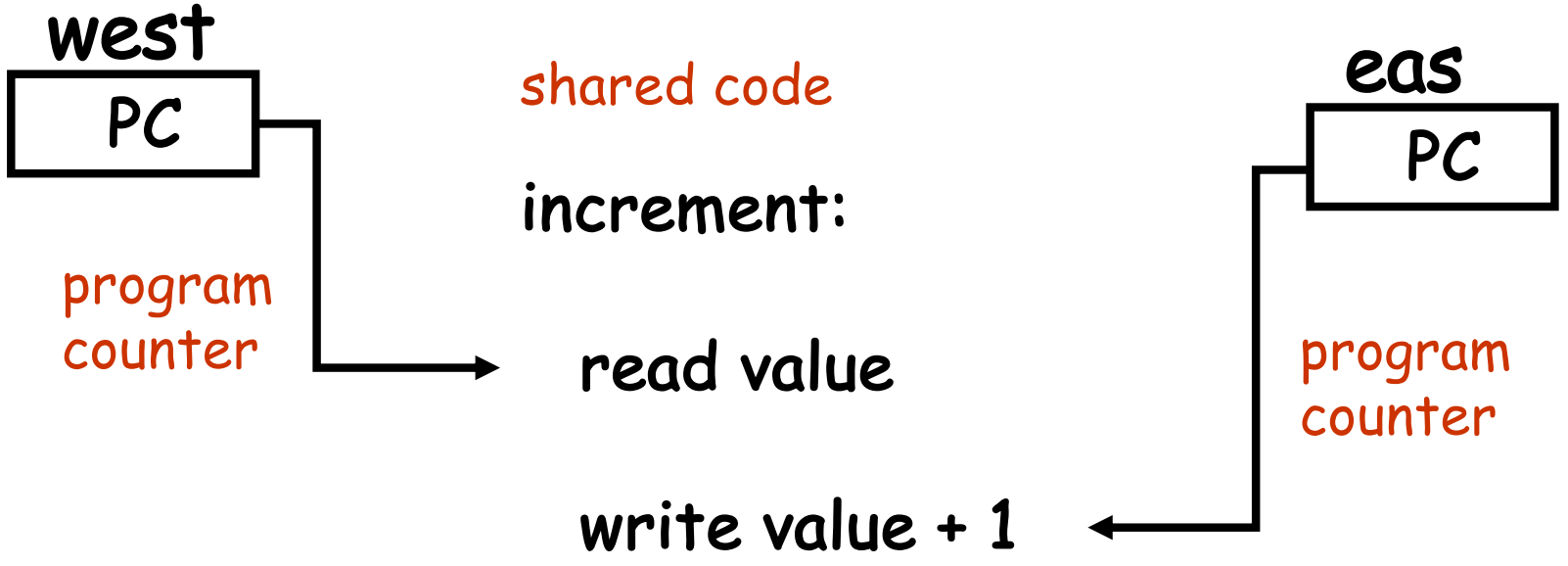
  1. synchronized objects/methods
  2. lock object
  3. semaphore object
  4. peterson algorithm (wait free synchronisation)

Ornamental garden program - display

- After the East and West turnstile threads have each incremented its counter 20 times, the garden people counter is not the sum of the counts displayed. Counter increments have been lost. *Why?*

concurrent method activation

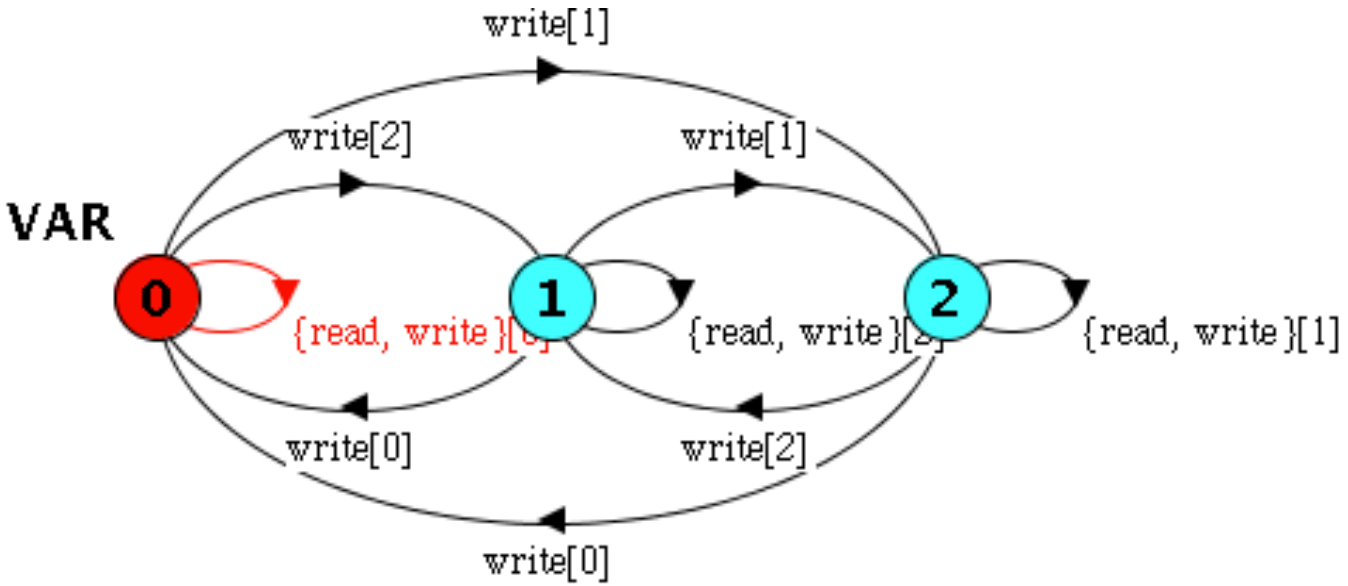
- Java method activations are not atomic - thread objects **east** and **west** may be executing the code for the increment method at the same time.



Modeling mutual exclusion

- The shared **VAR**:

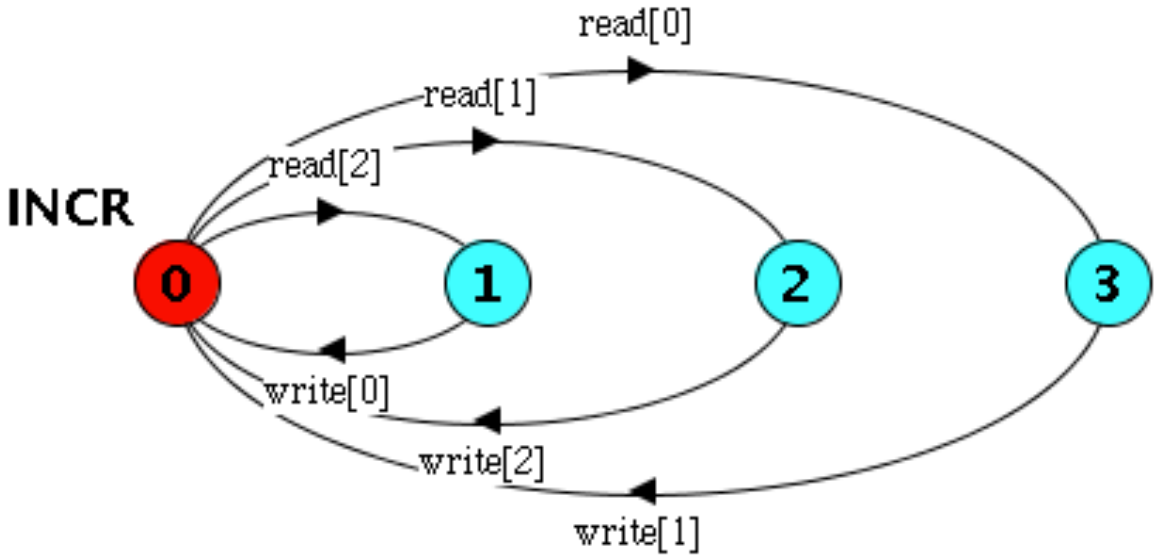
```
const N = 3
range T = 0..(N-1)
VAR = VAR[0],
VAR[i:T] = (read[i] -> VAR[i]
           |write[u:T] -> VAR[u]).
```



Modeling mutual exclusion

- The `INCR` process:

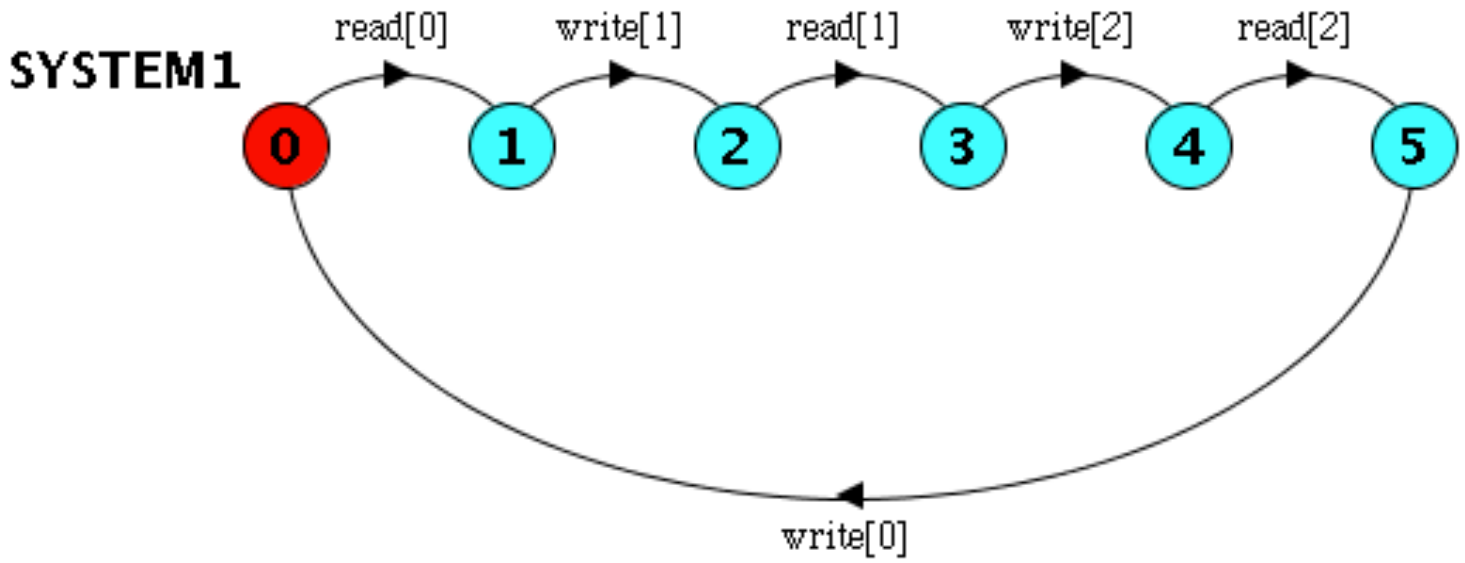
```
INCR = (read[x:T] -> write[(x+1)%N] -> INCR).
```



Modeling mutual exclusion

- Si on compose : la variable et le processus d'incrémentation

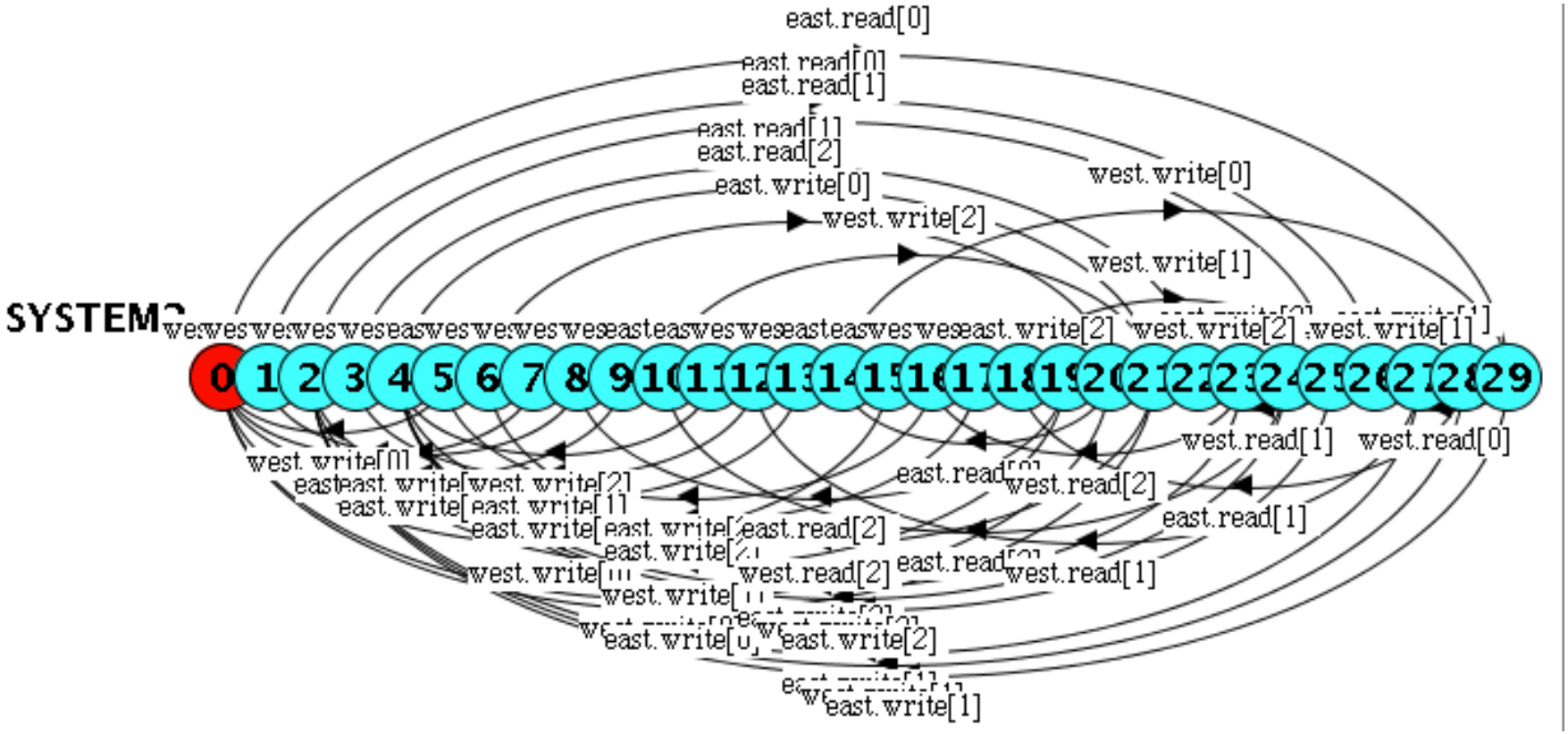
```
||SYSTEM1 = (INCR || VAR).
```



Modeling mutual exclusion

- Et si on compose : la variable et deux processus d'incrémentation

```
||SYSTEM2 = ({east, west}:INCR || {east, west}::VAR).
```



Modeling mutual exclusion

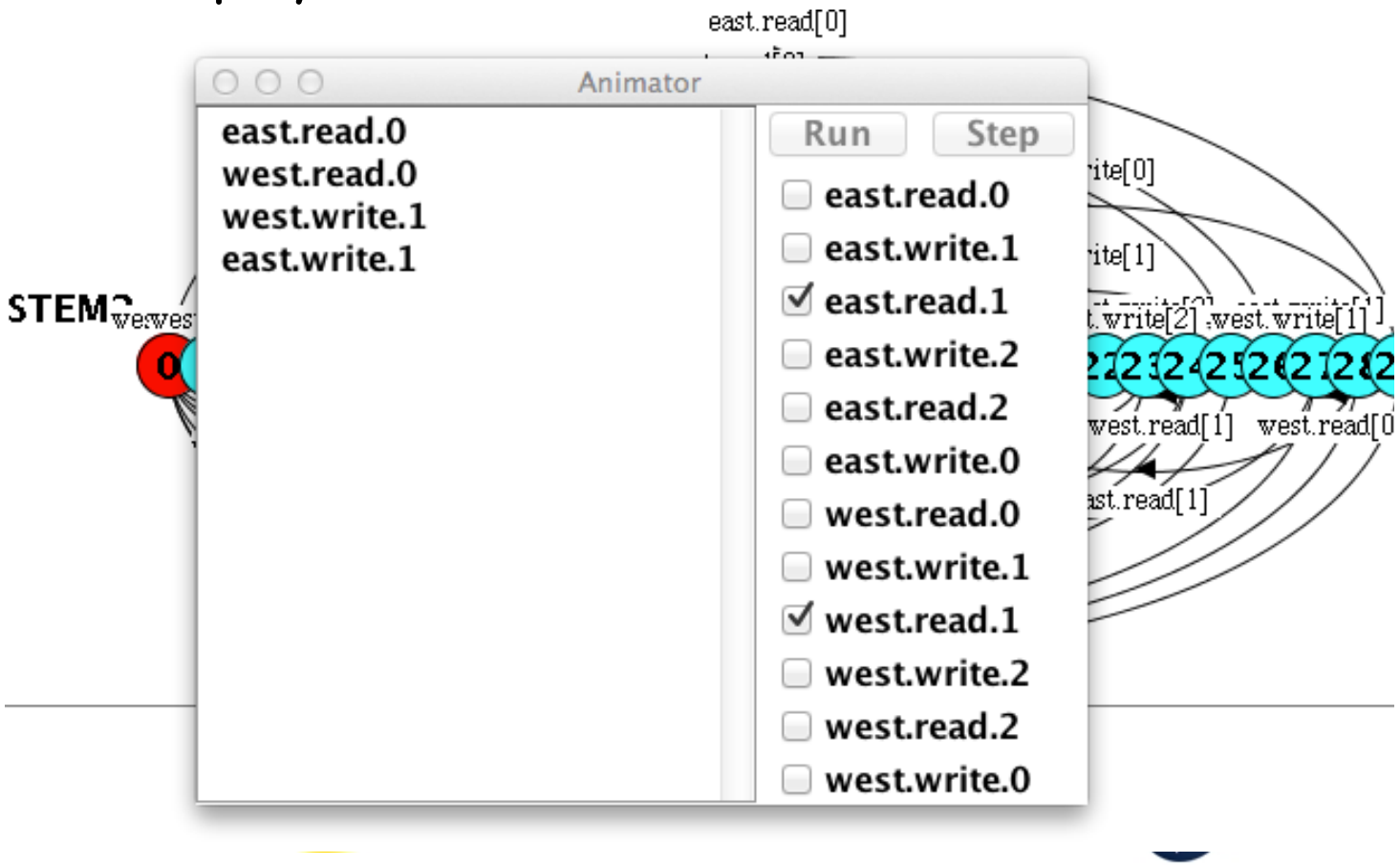
- Et si on compose : la variable et deux d'incrémentation

```
||SYSTEM2 = ({east, west}:INCR
```

Pas facile de savoir si tout se passe bien.
Cf. TD 2 pour la mise en oeuvre de la preuve en FSP

Modeling mutual exclusion

- As we know that this is not correct:
  - Uncontrolled access to a shared variable by multiple processes
  - We can replay error

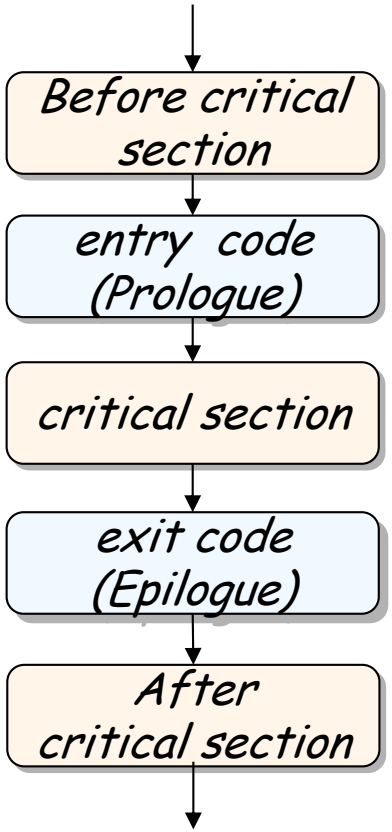


Modeling mutual exclusion

- Mutual exclusion refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time
  - What is the critical section in the ornemental garden program?

```
read[x:T] -> write[(x+1)%N]
```

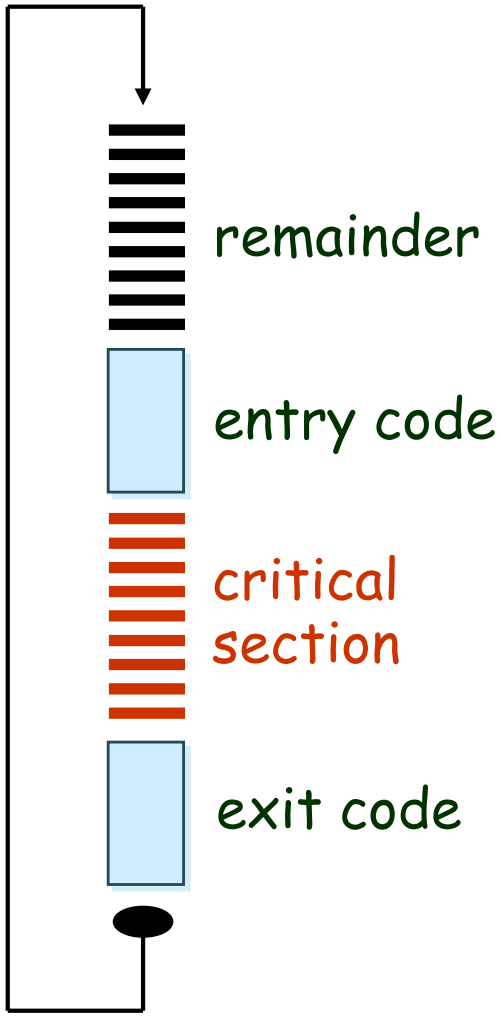
- Generic solution



The mutual exclusion problem

- Mutual Exclusion: No two processes are in their critical sections at the same time.
- Deadlock-freedom: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- Starvation-freedom: If a process is trying to enter its critical section, then this process must eventually enter its critical section.

– No assumption time
– All process execute an equivalent algorithms



Mise en œuvre d'une section critique à l'aide d'un verrou

Verrou

- Principe
  - Objet primitif ayant 2 **opérations atomiques**
  - **Prendre** : si le verrou est déjà pris alors bloque la thread sinon la thread prend le verrou
  - **Rendre** : rend le verrou ; si une thread est bloquée alors libère une thread
- En FSP

```
LOCK = (acquire->release->LOCK).
```

  - Mise en oeuvre d'une section critique

```
PROCESSUS = (acquire
             -> section_critique
             -> release->PROCESSUS).
```

- `LOCK = (acquire->release->LOCK).`

- ```
  NEW_INCR = (acquire->read[x:T]->write[(x+1)%N]
              ->release->NEW_INCR).
  ```

- ```
  ||SYSTEM3 = ({east, west}:NEW_INC
              || {east, west}::LOCK
              || {east, west}::VAR).
  ```

How do you program a “Lock” in Java ?

- lock in Java (synchronized objects/methods).
  - `synchronized aMethod () { blabla }`

  → lock on “this” object, mutual exclusion for all method of the “this” object

  - `synchronized (anObject) { blabla }`

  → lock on “anObject” object, mutual exclusion for all synchronized block of the “anObject”

  → The object could be this

  → Only one lock by synchronized objects

How do you program “Lock” in Java ?

- Java provides also 3 kind of Lock
- **Lock**
- **ReentrantLock**

```
Lock l = new blabla;
l.lock();
try {     // access the resource
          // protected by this lock
} finally { l.unlock();}
```

- **ReentrantReadWriteLock**

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.readLock().lock(); rwl.readLock().unlock();
rwl.writeLock().lock(); rwl.writeLock().unlock();
```

How do you program « Lock » in Python

- This is the subject of the second stage of the project. What are the different types of Lock that can be used.

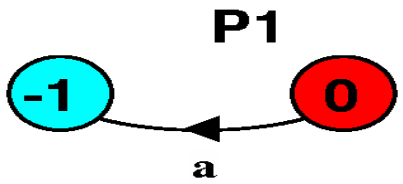
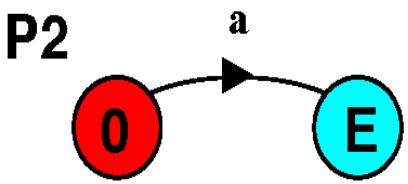
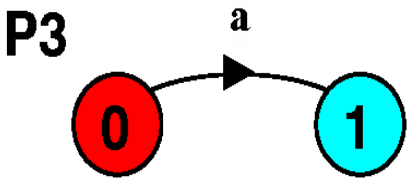
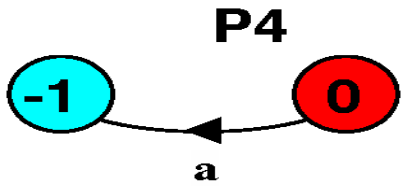
- Of course, make this study according to the library chosen in the previous step.

Preuve de programme en FSP et utilisation dans le cadre de la mise en œuvre de section critique

Méthodologie de preuve

- Pour prouver qu'un processus P est correct, il faut
  1. Construire le processus P à prouver
  2. Construire le processus Q décrivant les comportements corrects attendus mais aussi tous les comportements incorrects
     - La difficulté est de construire tous les comportements incorrects
  3. Composer P avec Q et vérifier qu'il ne reste pas de comportement incorrect.
     - **Si existe état -1** ☒
     - **Si existe état puit** ☒
     - **Sinon** ☑

Comportement correct / incorrect

	OK	Erreur	Check -> safety
P1 = (a -> Q).			Property violation
P2 = (a -> END).			No deadlocks/errors
P3 = (a -> STOP).			DEADLOCK
P4 = (a -> ERROR).			Property violation

Méthodologie de preuve

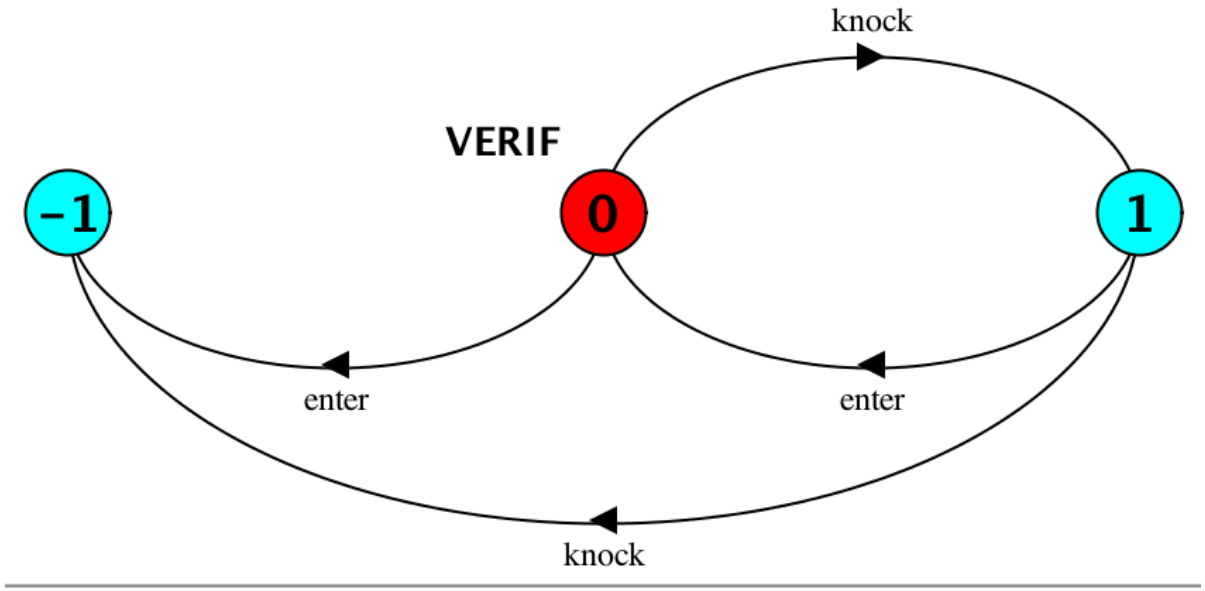
- On prouver que le processus P est correct
- Pour cela construire le processus VERIF décrivant les comportements corrects et incorrects attendus
- Composer P avec VERIF et vérifier qu'il ne reste pas de comportement incorrect.

- Exemple :
  - Toutes les personnes polies frappent 1 seule fois avant d'entrer
  - Voici quelques exemples de personnes
    - `P1 = (enter -> do_something -> P1).`
    - `P2 = (enter -> knock -> do_something -> P2).`
    - `P3 = (knock -> enter -> do_something -> P3).`
    - `P4 = (knock -> do_something -> P4).`

Exemple (30a-preuve.lts)

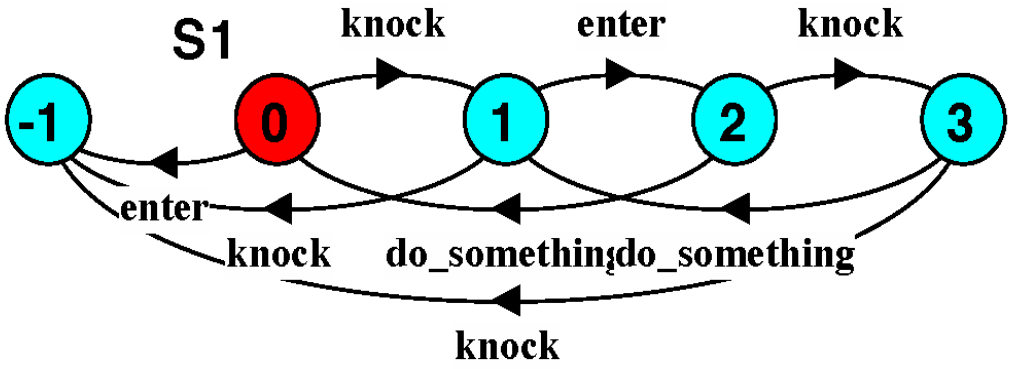
- Toutes les personnes polies frappent 1 seule fois avant d'entrer

- Le processus verificateur
  - `VERIF = (enter -> ERROR | knock -> P),`
    `P = (knock -> ERROR | enter -> VERIF).`

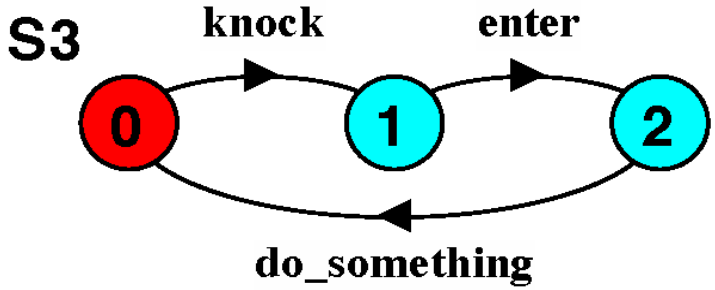


Exemple

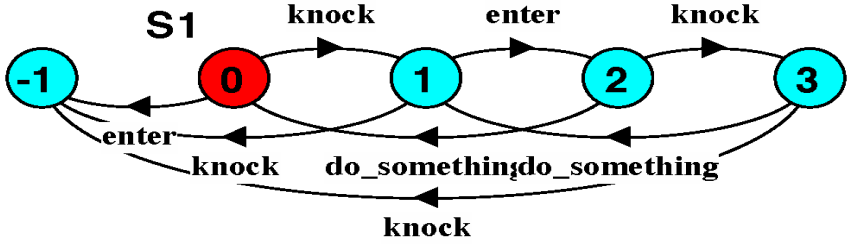
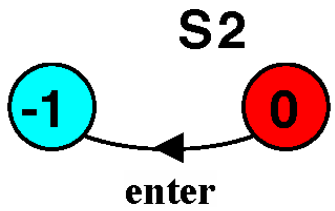
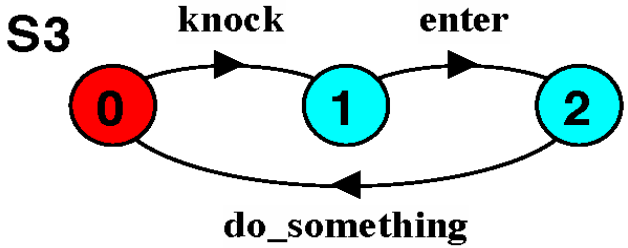
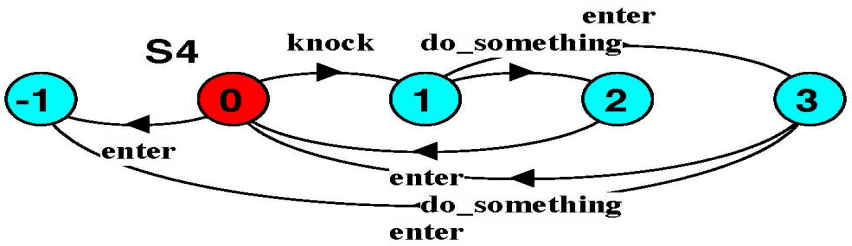
- On compose le processus de preuve avec le processus à vérifier
  - `P1 = (enter -> do_something -> P1).`
  - `||S1 = (P1 || VERIF).`



  - `P3 = (knock -> enter -> do_something -> P3).`
  - `||S3 = (P3 || VERIF).`



Comportement correct / incorrect

		Check -> safety
||S1 = (P1 || VERIF).		Property violation
||S2 = (P2 || VERIF).		Property violation
||S3 = (P3 || VERIF).		No deadlocks/ errors
||S4 = (P4 || VERIF).		Property violation

Exemple : entrée en section critique sans 'précaution'

```
PROCESSUS = (entre_SC -> sort_SC -> PROCESSUS).
||SYSTEM = ({a,b}:PROCESSUS).
```

Target properties :

- Mutual Exclusion: No two processes are in their critical sections at the same time.
- Deadlock-freedom: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- Starvation-freedom: If a process is trying to enter its critical section, then this process must eventually enter its critical section.
- No assumption time
- All process execute an equivalent algorithms

Preuve 'Mutual Exclusion'

- Mutual Exclusion: No two processes are in their critical sections at the same time.
  - Property of safety

```
PREUVE =
     (a.entre_SC -> A          // A entre en SC
     | b.entre_SC -> B         // B entre en SC
     | {a, b}.sort_SC -> ERROR),
                  // comportement incorrect des processus
A = (a.sort_SC -> PREUVE       // A sort de la SC
     | b.entre_SC -> ERROR     // B entre en SC
     | {a.entre_SC, b.sort_SC} -> ERROR),
                  // comportement incorrect des processus
B = (b.sort_SC -> PREUVE       // B sort de la SC
     | a.entre_SC -> ERROR     // A entre en SC
     | {a.sort_SC, b.entre_SC} -> ERROR).
                  // comportement incorrect des processus

||TEST = (SYSTEM || PREUVE).
```

Compilation

```
Compiled: PROCESSUS
Compiled: PREUVE
Composition: TEST =
                a:PROCESSUS || b:PROCESSUS
                || PREUVE_Mutual_Exclusion
State Space: 2 * 2 * 3 = 2 ** 4
Composing...
  property PREUVE_Mutual_Exclusion violation.

Check -> safety
Trace to property violation in PREUVE_Mutual_Exclusion :
       a.entre_SC
       b.entre_SC
```

- Manque la preuve des propriétés :
  - 'Deadlock-freedom' et 'Starvation-freedom'
  - Il nous manque encore quelques éléments en FSP

Preuve 'Deadlock-freedom' or 'Starvation-freedom'

- Deadlock-freedom: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
  - Blocking solution : property of safety detected by deadlock
  - Wait-free solution : property of liveness
- Starvation-freedom: If a process is trying to enter its critical section, then this process must eventually enter its critical section.
  - Property of liveness
- Safety :
  - No ERROR or STOP states
- Liveness :
  - We do not have the language elements from FSP to prove it

Contrôle d'une section critique par un verrou

Exemple de verrou en Java

Mot clé : synchronized

Objet : Lock, ReentrantLock, ReadWriteLock

Verrou – fichier 30b-increments.lts

- Principe
  - Objet primitif ayant 2 **opérations atomiques**
  - **Prendre** : si le verrou est déjà pris alors bloque la thread sinon la thread prend le verrou
  - **Rendre** : rend le verrou ; si une thread est bloquée alors libère une thread
- En FSP

```
LOCK = (acquire->release->LOCK).
```

  - Mise en oeuvre d'une section critique

```
PROCESSUS = (acquire
             -> section_critique
             -> release->PROCESSUS).
```

Utilisation d'un verrou

```
LOCK = (acquire -> release -> LOCK).

PROCESSUS = PROLOGUE,
PROLOGUE = (acquire -> SC),
SC = (entre_SC -> sort_SC -> EPILOGUE),
EPILOGUE = (release -> PROCESSUS).

||SYSTEM = ({a,b}:PROCESSUS || {a,b}::LOCK).
```

Utilisation d'un verrou - preuve

```
PREUVE =
     (a.entre_SC -> A          // A entre en SC
     | b.entre_SC -> B         // B entre en SC
     | {a, b}.sort_SC -> ERROR),
                // comportement incorrect des processus
A = (a.sort_SC -> PREUVE     // A sort de la SC
     | b.entre_SC -> ERROR   // B entre en SC
     | {a.entre_SC, b.sort_SC} -> ERROR),
                // comportement incorrect des processus
B = (b.sort_SC -> PREUVE     // B sort de la SC
     | a.entre_SC -> ERROR   // A entre en SC
     | {a.sort_SC, b.entre_SC} -> ERROR).
                // comportement incorrect des processus

||TestMutualExclusion = (SYSTEM || PREUVE).
```

How do you program a “Lock” in Java ?

- lock in Java (synchronized objects/methods).
  - `synchronized aMethod () { blabla }`

  → lock on “this” object, mutual exclusion for all method of the “this” object

  - `synchronized (anObject) { blabla }`

  → lock on “anObject” object, mutual exclusion for all synchronized block of the “anObject”

  → The object could be this

  → Only one lock by synchronized objects

The mutual exclusion problem

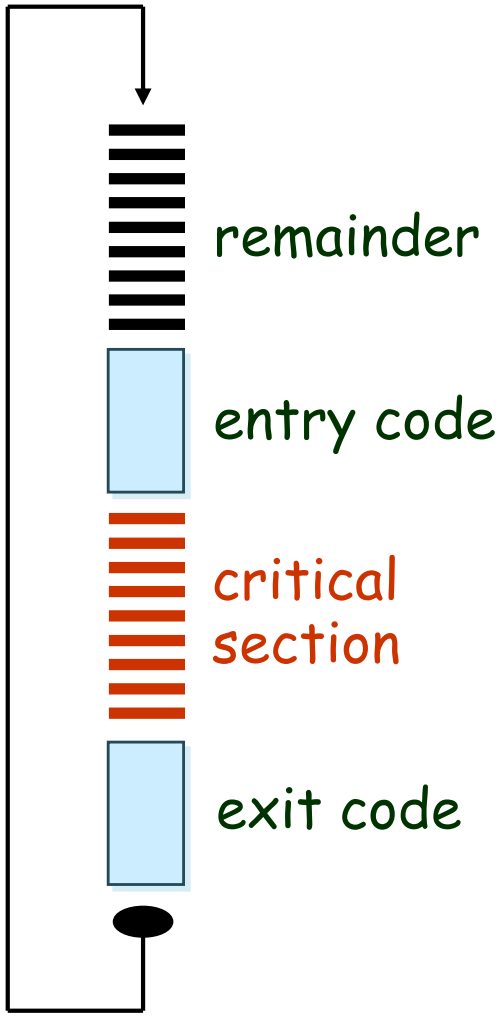
- Mutual Exclu… in their critic… ne.
- Deadl… is trying to ente… some pr… same one, …cal section.

Propriétés garanties par la machine Virtuelle Java

…n: If a process is trying …cal section, then this …eventually enter its critical

…f liveness

…umption time …rocess execute an equivalent algorithms



How do you program “Lock” in Java ?

- Java provides also 3 kind of Lock
- **Lock**
- **ReentrantLock**

```
Lock l = new blabla;
l.lock();
try {     // access the resource
          // protected by this lock
} finally { l.unlock();}
```

- **ReentrantReadWriteLock**

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.readLock().lock(); rwl.readLock().unlock();
rwl.writeLock().lock(); rwl.writeLock().unlock();
```

The mutual exclusion problem

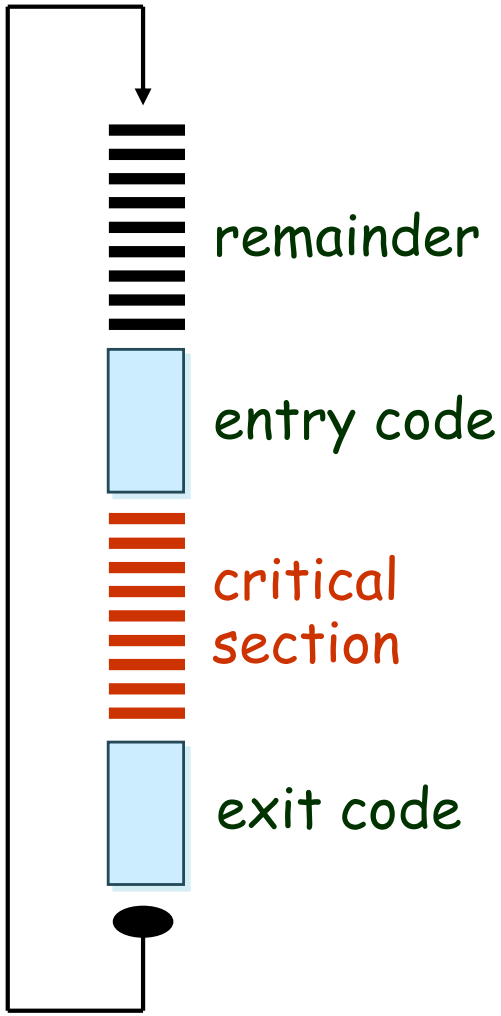
- Mutual Exclu… in their critic… ne.
- Deadl… is trying to ente… some pr… same one, …cal section.

Propriétés garanties par la mise en œuvre des objets 'Lock'

…n: If a process is trying …cal section, then this …eventually enter its critical

…f liveness

…umption time …rocess execute an equivalent algorithms



ATTENTION

Très certainement QCM la semaine prochaine.