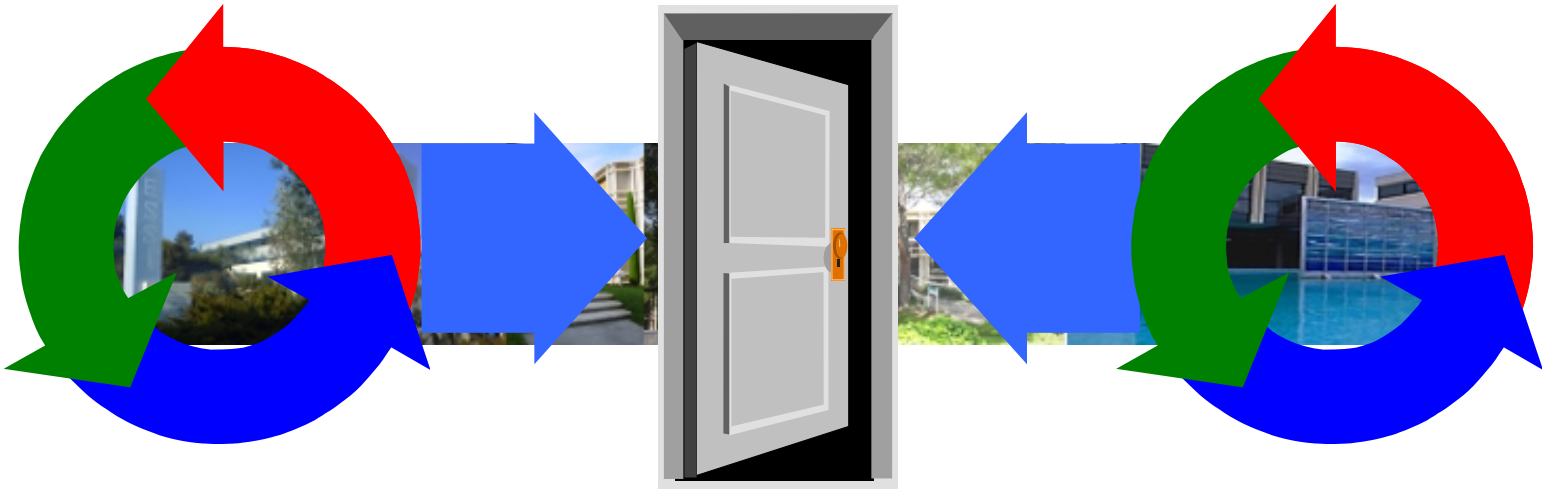
Mise en oeuvre de section critique

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



Université Nice Sophia Antipolis

Contrôle d'une section critique par sémaphore

Semaphores

- Concept inventé par Edsger Dijkstra en 1962
- Implémenté dans de très nombreux systèmes d'exploitation

- A chaque sémaphore est associé :
  - Un compteur
  - Deux procédures qui s'exécutent en mutuelle exclusion
    - Down
    - Up

Semaphores

- Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore ***s*** is an 'integer variable' that can take only positive or null values.
- The only operations permitted on ***s*** are *up(s)* and *down(s)*. Blocked processes are held in a FIFO queue.

```
down(s): if s >0 then                                  // originally P(s)
             decrement s
         else
             block execution of the calling process

up(s):   if processes blocked on s then                // originally V(s)
             awaken one of them
         else
             increment s
```

Modeling semaphores

- To ensure analyzability, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an ERROR. N is the initial value.

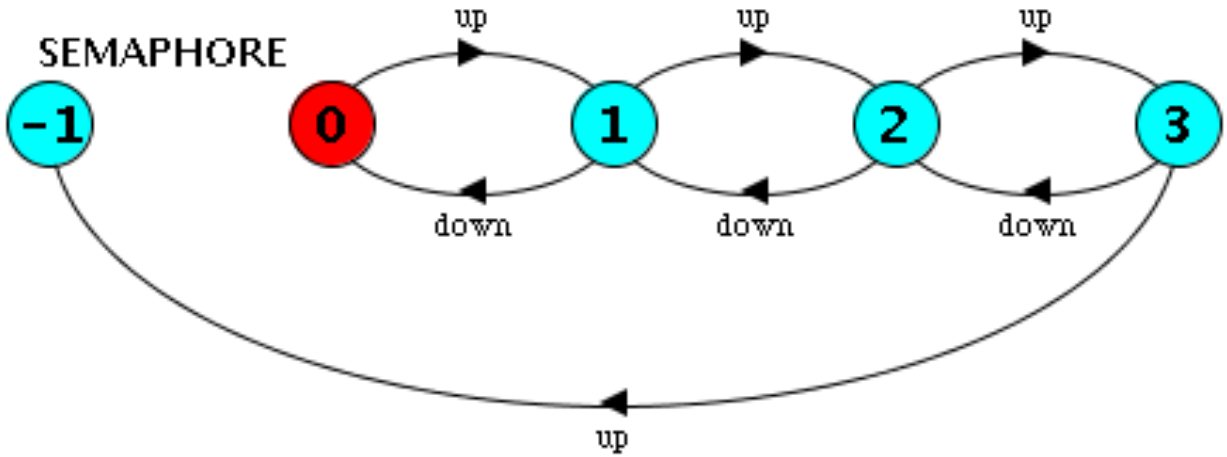
```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int]    = (up->SEMA[v+1]
                  |when(v>0) down->SEMA[v-1]
                           ),
SEMA[Max+1]    = ERROR.
```

- *LTS?*

modeling semaphores

- Action down is only accepted when value v of the semaphore is greater than 0.
- Action up is not guarded.



- Trace to a violation:
  - **up → up → up → up**

Critical section with semaphore

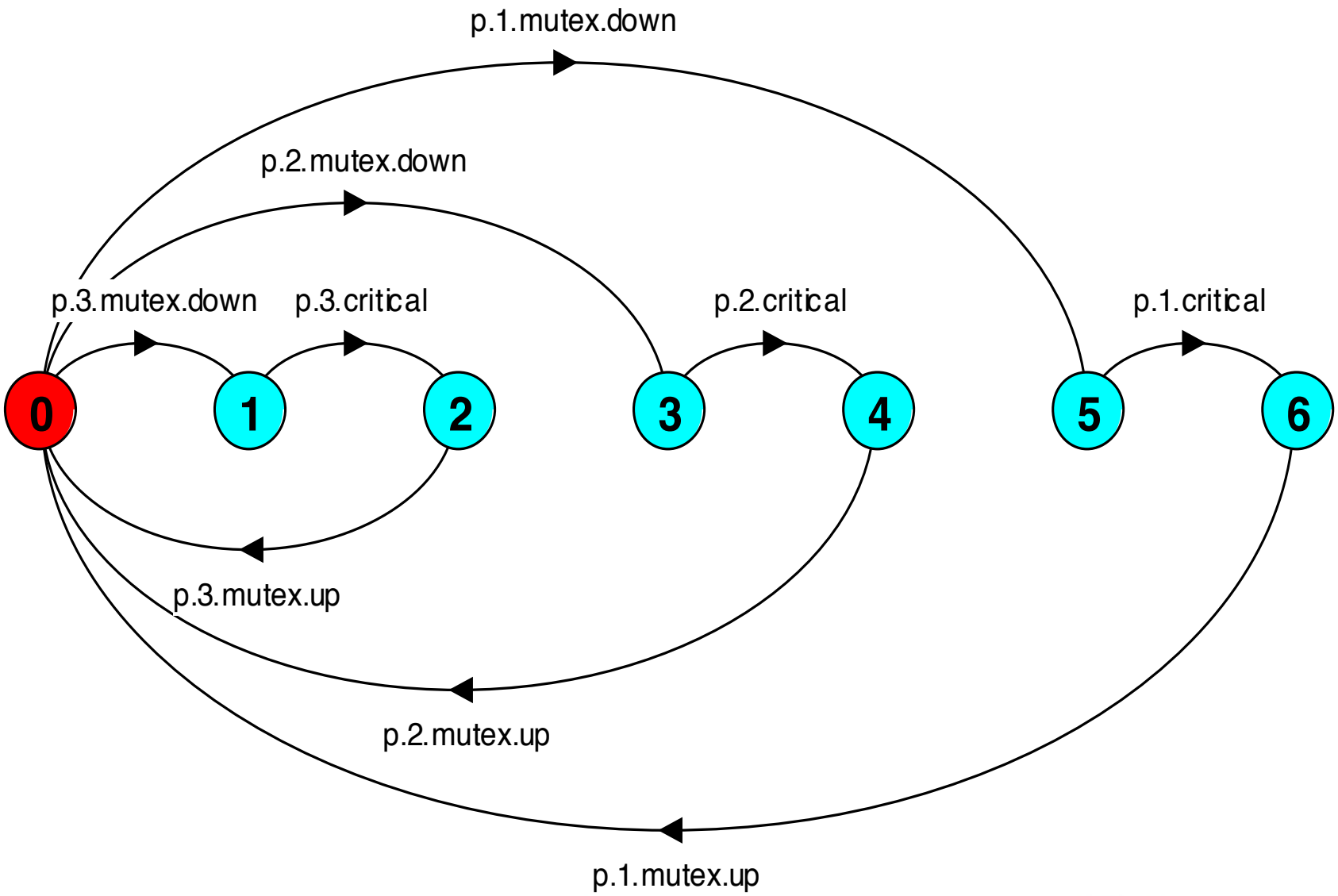
- Three processes `p[1..3]` use a shared semaphore **`mutex`** to ensure mutually exclusive access (action **`critical`**) to some resource.

```
LOOP      =   (mutex.down->critical->mutex.up->LOOP).
SEMAPHORE = (…).

||SEMADEMO = (p[1..3]:LOOP
          ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

- For mutual exclusion, the semaphore initial value is 1.
  - *Why?*
- *Is the* **`ERROR`** *state reachable for* **`SEMADEMO`***?*
- *Is a binary semaphore sufficient (i.e.* **`Max=1`***)?*
- *LTS?*

Critical section with semaphore



Preuve en FSP

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int] = (up->SEMA[v+1]
              |when(v>0) down->SEMA[v-1]),
SEMA[Max+1] = ERROR.

PROCESSUS = PROLOGUE,
PROLOGUE = (mutex.down -> SC),
SC = (entre_SC -> sort_SC -> EPILOGUE),
EPILOGUE = (mutex.up -> PROCESSUS).

||SYSTEM = ({a,b}:PROCESSUS || {a,b}::mutex:SEMAPHORE(1)).

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

Avec sémaphore initialisé à 0

```
||SYSTEM = ({a,b}:PROCESSUS
            || {a,b}::mutex:SEMAPHORE(0)).

||TEST = (SYSTEM || PREUVE).

Composing...  potential DEADLOCK

Check -> Safety
      Trace to DEADLOCK:
```

Avec sémaphore initialisé à 2

```
||SYSTEM = ({a,b}:PROCESSUS
            || {a,b}::mutex:SEMAPHORE(2)).

||TEST = (SYSTEM || PREUVE).

Composing...  property PREUVE_Mutual_Exclusion violation.

Check -> Safety
      Trace to property violation in PREUVE_Mutual_Exclusion:
            a.mutex.down  a.entre_SC
            b.mutex.down  b.entre_SC
```

Sémaphore

- Java possède une classe Sémaphore

```
Semaphore sample = new Semaphore(1, true);
    // if true, fair semaphore
sample.acquire();   // equivalent of down
sample.release();   // equivalent to up
```

- Généralement, les valeurs d'initialisation des sémaphores sont les suivantes :
  - 0 : sémaphore privé permettant de bloquer un processus
  - 1 : mutex permettant de contrôle l'accès à une section critique
  - N : sémaphore permettant de contrôle l'accès à une ressource disponible en N exemplaires

The mutual exclusion problem

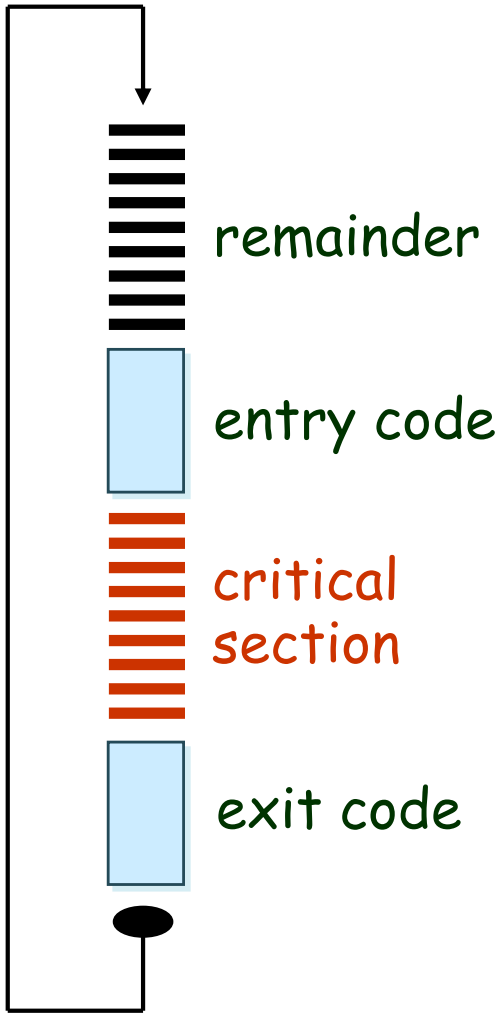
- Mutual Exclu… …e in their critic… …ne.
- Deadl… …s is trying to ente… …n some pr… …e same one, …cal section.

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

…f liveness

…umption time …rocess execute an equivalent algorithms

Propriétés
Garantie par la mise en œuvre des objets 'sémaphores'



Contrôle d'une section critique par attente active

Accès aux sections critiques

- Accès bloquant
  - Verrou (`lock` ou `synchronized`)
  - Sémaphore (initialisé à 1)
  - Avantage : simple d'utilisation
  - Inconvénient : opération couteuse si inutile

- Accès par attente active
  - Dekker, Peterson → fonctionne pour 2 processus
  - Diskstra → a publié une solution pour N processus

  - Attention, ces approches font l'hypothèse que les lectures et les écritures mémoires sont atomiques
    - l'ordonnanceur ne peut pas changer de thread tant que les effets de l'écriture ne sont pas complets

  - A réserver aux programmeurs avertis et uniquement si le risque d'attente est faible

Dekker (fonctionne pour 2 processus)

- Initialisation des variables partagées

```
tour = 0
actif = {faux, faux}
```

- Prologue pour le processus i (i = 0 ou 1)

```
actif[i] := vrai
tantque (actif[i-1]) faire
    si (tour = (i-1) alors
        actif[i] := faux
        tantque (tour ! = i) faire
            nothing
        fintantque
        actif[i] := vrai
    finsi
fintantque
```

- Epilogue pour le processus i

```
tour := 1-i
actif[i] := faux
```

Peterson (fonctionne pour 2 processus)

- Initialisation des variables partagées

```
Tour = 0
EnAccès = {faux, faux}
```

- Prologue pour le processus i (i = 0 ou 1)

```
EnAccès[i] = vrai
Tour = 1-i
tantque (EnAccès[1-i]) && ((1-i) = Tour )) faire
    nothing
fintantque
```

- Epilogue pour le processus i

```
EnAccès[i] = faux
```

Peterson en FSP

```
range T = 0..1
set VarAlpha = {tour.{read[T], write[T]}, [0].{read[T], write[T]},
[1].{read[T], write[T]}}

VAR = VAR[0],
VAR[u:T] = (read[u] -> VAR[u]
           | write[v:T] -> VAR[v]).

PROCESSUS(I=0) = PROLOGUE,
PROLOGUE = ([I].write[1] -> tour.write[1-I] -> READ),
READ = ([1-I].read[u:T] -> tour.read[v:T]-> TEST[u][v]),
TEST[u:T][v:T] = (when (u != 1 || (1-I) != v) [I].entre_SC -> SC
                 |when (u == 1 && (1-I) == v) [I].skip -> READ),
SC = ([I].sort_SC -> EPILOGUE),
EPILOGUE = ([I].write[0] -> PROCESSUS)+VarAlpha.

||SYSTEM = (a:PROCESSUS(0) || b:PROCESSUS(1)
          || {a, b}::tour:VAR
          || {a, b}::[0]:VAR
          || {a, b}::[1]:VAR).
```

Peterson en FSP

```
PREUVE_Mutual_Exclusion = (a.[0].entre_SC -> A.   // A entre en SC
                         | b.[1].entre_SC -> B),          // B entre en SC
A = (a.[0].sort_SC -> PREUVE_Mutual_Exclusion.   // A sort de la SC
                         | b.[1].entre_SC -> ERROR),      // B entre en SC
B = (b.[1].sort_SC -> PREUVE_Mutual_Exclusion.   // B sort de la SC
                         | a.[0].entre_SC -> ERROR).      // A entre en SC

||TEST_Mutual_Exclusion = (SYSTEM || PREUVE_Mutual_Exclusion).
```

Check -> Safety

No deadlocks/errors

The mutual exclusion problem

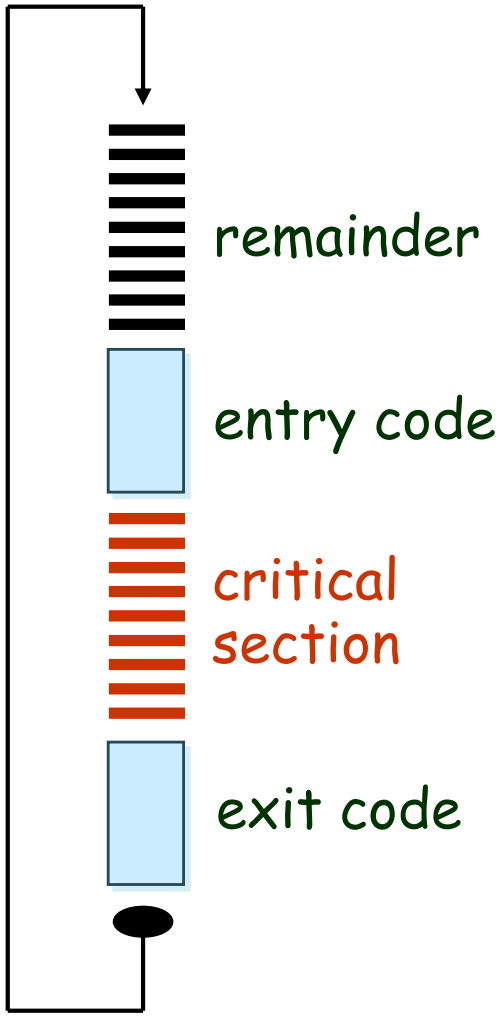
- Mutual Exclu… in their critic… ne.
- Deadl… is trying to ente… some pr… same one, …cal section.

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

…f liveness

…umption time
…rocess execute an equivalent algorithms

Pour que ce soit correct…
Il faut respecter les hypothèses :
Lectures + écritures atomiques



Hypothèses : lecture et écriture atomique
Est-ce vrai en Java ?

- Soit `v` une variable partagée par plusieurs threads
- Si la thread `T` modifie la valeur de `v`, cette modification peut ne pas être connue immédiatement par les autres threads
  - Le compilateur a put utiliser un registre pour conserver la valeur de v pour T
  - La spécification de Java n'impose la connaissance de cette modification par les autres threads que lors de l'acquisition ou le relâchement du moniteur d'un objet (`synchronized`)
- Si la variable `v` est déclarée comme **`volatile`**
  - Les différents threads partageront une même zone mémoire commune pour ranger la valeur de la variable v
  - Les opérations de lecture et d'écriture sont garanties atomiques pour tous les types simple (`long`, `double` compris)
    - Mais pas pour les tableaux…

Petit test…

1. **Implémentation de Dijkstra, Dekker et Peterson sans utiliser les volatile**
   - ***C'est pas la joie*** : on « perd » régulièrement quelques entrées et parfois, une thread se bloque…
2. Déclaration des variables 'volatiles' y compris tableaux
   - Pour le jardin pas d'erreur détectée… au bout de quelques heures
   - Mais sur une architecture multi-cœur en stressant un peu le système,
     - Avec Dijkstra 0,03 % d'erreur
     - avec Dekker 0,004 % d'erreur
     - avec Peterson pas d'erreur détectée après 10^{10} exécutions
3. En remplacant les tableaux par des variables

→ Plus d'erreur, après 24 heures d'éxecution pour les 3 algorithmes

→ **Attention :** pas d'erreur visible n'est pas synonyme à « code correct »

→ Le code est correct parce que

→ Respect de l'algorithme

→ Respect des hypothèses

Monsieur, est-ce que mon code est correct ?

- Code utilisé par deux threads T1 et T2
  - T1 appelle la méthode work()
  - T2 appelle la méthode stopWork()

```
public class BouclePotentiellementInfinie {
    private boolean termine = false;
    public void work() {
        while (!termine) { /* do stuff */ }
    }
    public void stopWork() {
        termine = true;
    }
}
```

→ On souhaite que l'appel de stopWork() par T2 arrête l'exécution de la thread T2 qui appellee la méthode work()

Réponse : NON

- Pourquoi ?
  - En l'absence de mécanisme de synchronisation ou de déclaration de la variable comme volatile

  → la JVM a aucune obligation de mettre en cohérence les caches mémoire

- Par conséquent, il est possible que la valeur de termine soit en cache et que le cache ne soit jamais mis à jour.

  → la boucle se transforme alors en boucle infinie.

- Comment résoudre le problème ?
  - Il faut forcer la "mise en cohérence des caches" après chaque écriture

```
public class BouclePotentiellementInfinie {
    private volatile boolean termine = false;
    public void work() { while (!termine)
                                { /* do stuff */ } }
    public void stopWork() { termine = true; }
}
```

Safety and Liveness
(sureté et vivacité)

safety & liveness properties

Concepts: properties: true for every possible execution

safety: nothing bad happens

liveness: something good *eventually* happens

Models: safety: no reachable ERROR/STOP state

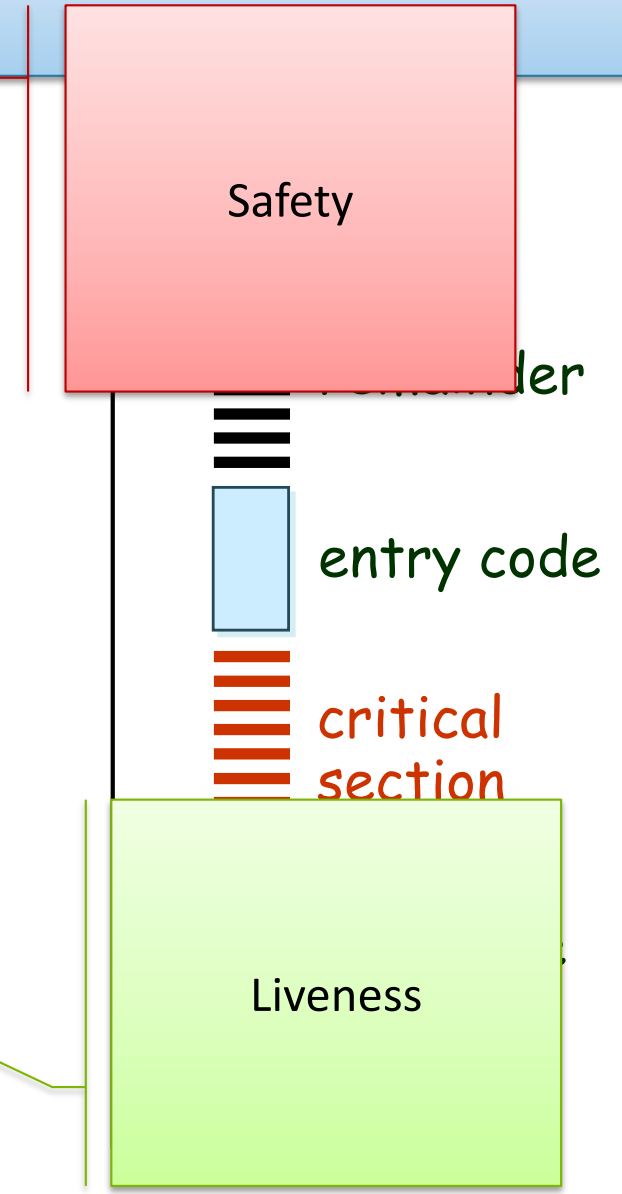
progress: an action is *eventually* executed

fair choice and action priority

The mutual exclusion problem

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

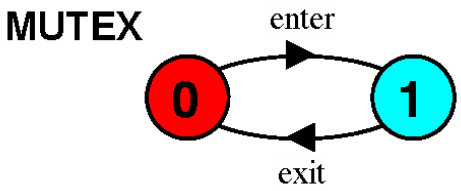
- No assumption time
- All process execute an equivalent algorithms



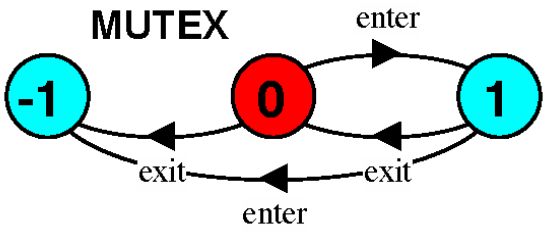
Safety property specification

- A **safety** property asserts that **nothing** bad happens.
  - **ERROR** conditions state what is **not** required (cf. exceptions).
  - In complex systems, it is usually better to specify safety **properties** by stating directly what **is** required.
    - It's the goal of 'property' directive

```
MUTEX = (enter
        -> exit
        -> MUTEX).
```




```
property MUTEX = (enter
        -> exit
        -> MUTEX).
```



Liveness property specification

- A **liveness** property asserts that something good ***eventually*** happens.
- Critical section: *does every process eventually get an opportunity to enter in the critical section?*
  - ie. make **PROGRESS?**
- A progress property asserts that it is *always* the case that an action is *eventually* executed. Progress is the opposite of *starvation*, the name given to a concurrent programming situation in which an action is never executed.

```
progress P = {enter}
```

Tout ce que l'on vient de voir…
et que l'on approfondira en TD…
doit être connu la semaine prochaine

Q&A

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