

# Propriété de sûreté et vivacité

## Safety and liveness

### Part 1 - Deadlock

[riveill@unice.fr](mailto:riveill@unice.fr)

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



# safety & liveness properties

- Concepts: **properties**: true for every possible execution
  - safety**: nothing bad **never** happens
  - liveness**: something good **eventually** happens
- Models:
  - safety**: no reachable **ERROR/STOP** state
  - progress**: an action is **eventually** executed
    - fair choice and action priority
- Practice:
  - threads and monitors

Aim: property satisfaction.

# Part one : Safety

# Safety

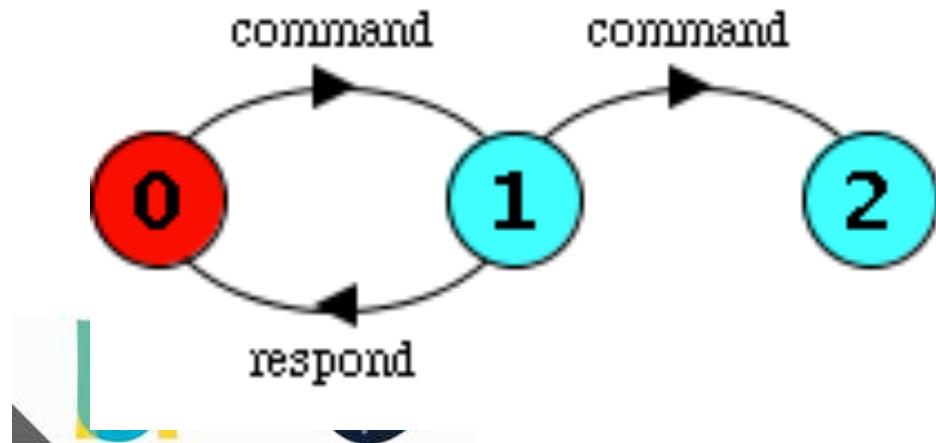
- A **safety** property asserts that nothing **bad** happens.
  - ◆ **STOP** or **deadlocked state** (no outgoing transitions)
  - ◆ **ERROR** process (-1) to detect erroneous behaviour

**ACTUATOR**

= (command->ACTION) ,

**ACTION**

= (respond->ACTUATOR  
| command->**STOP**) .

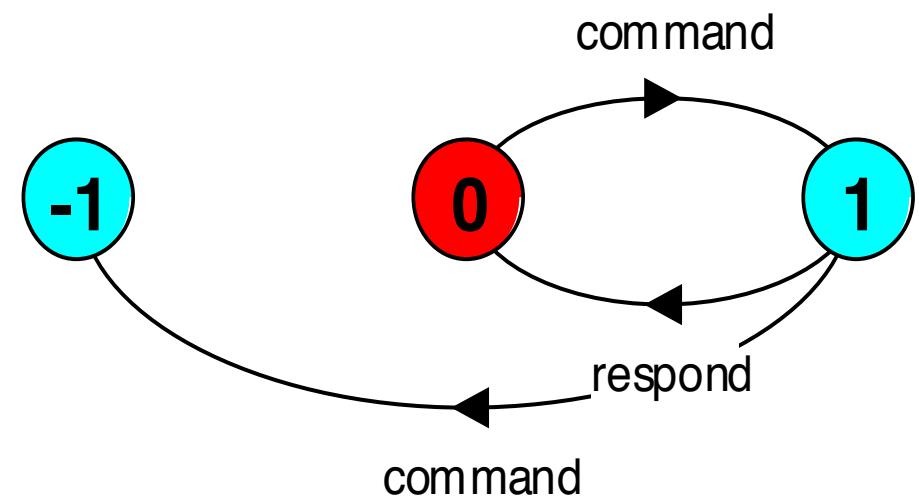


**ACTUATOR**

= (command->ACTION) ,

**ACTION**

= (respond->ACTUATOR  
| command->**ERROR**) .



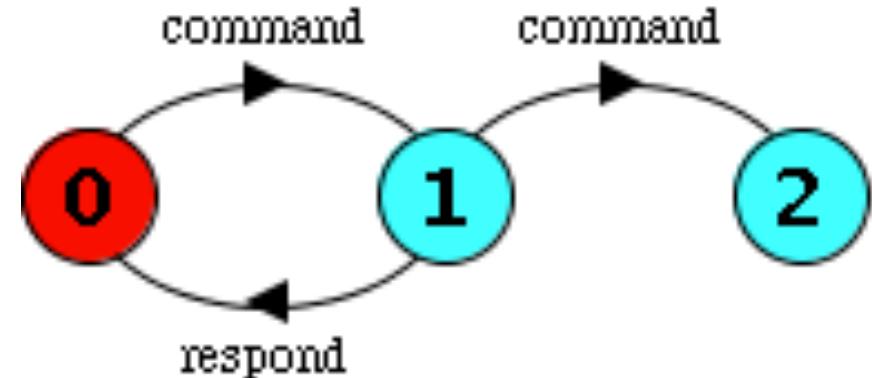
# STOP

**ACTUATOR**

= (command->ACTION) ,

**ACTION**

= (respond->ACTUATOR  
| command->**STOP**) .



- ◆ Analysis using LTSA:
    - ◆ Check -> Safety
- Trace to DEADLOCK:**
- command
- command

Give the shortest path

# ERROR

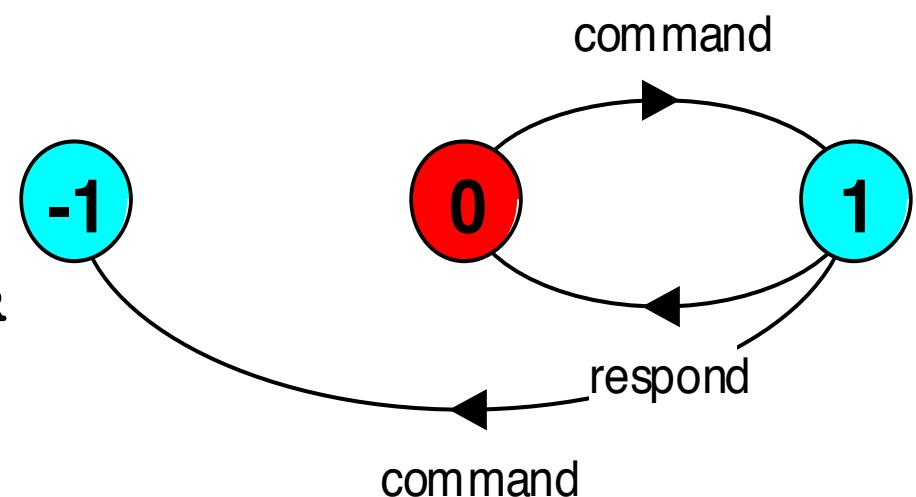
ACTUATOR

= (command->ACTION) ,

ACTION

= (respond->ACTUATOR

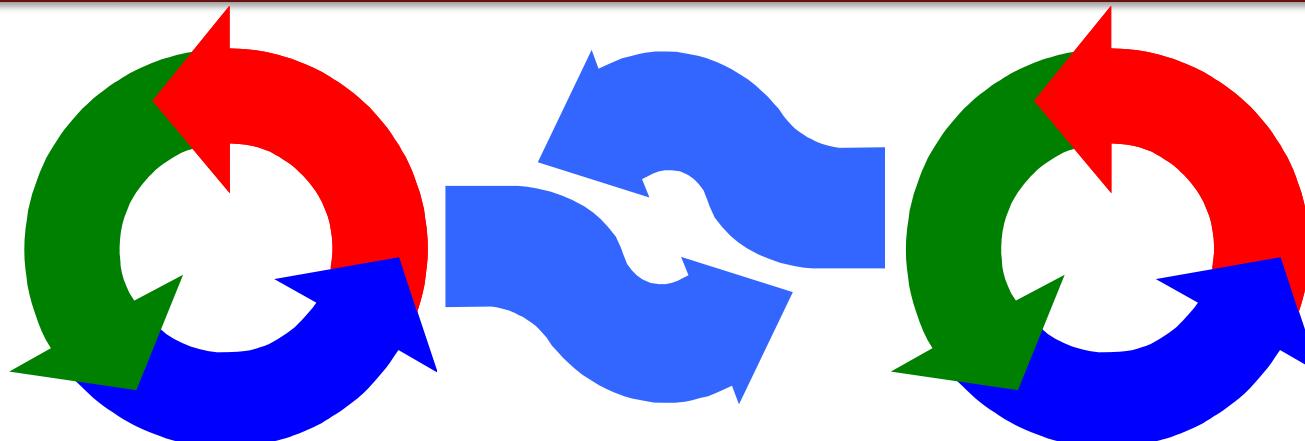
| command->**ERROR**) .



- ◆ Analysis using LTSA:
    - ◆ Check -> Safety
- Trace to ERROR:**
- command
- command

Give the shortest path

# Deadlock - no state STOP Safety property



# Deadlock

**Concepts:**

system **deadlock**: no further progress  
four necessary & sufficient conditions

**Models:**

deadlock - no eligible actions

**Practice:**

blocked threads

**Aim:** deadlock avoidance - to design systems where deadlock cannot occur.

# Deadlock: four necessary and sufficient conditions

- ◆ **Serially reusable resources:**

*the processes involved share resources which they use under mutual exclusion.*

- ◆ **Incremental acquisition:**

*processes hold on to resources already allocated to them while waiting to acquire additional resources.*

- ◆ **No pre-emption:**

*once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.*

- ◆ **Wait-for cycle:**

*a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.*

# deadlock analysis - parallel composition

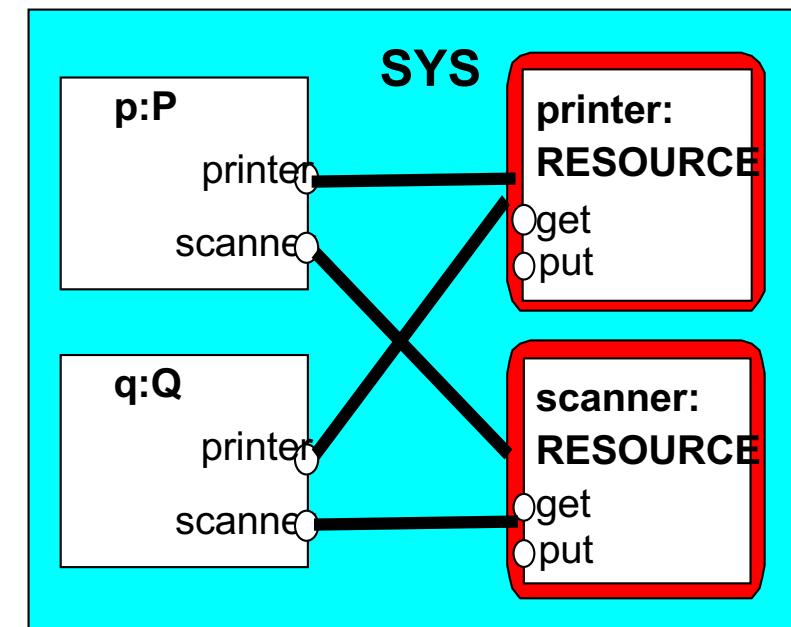
- in systems, deadlock may arise from the **parallel composition** of interacting processes.

```
RESOURCE = (get->put->RESOURCE) .
```

```
P = (printer.get->scanner.get  
      ->copy  
      ->printer.put->scanner.put  
      ->P) .
```

```
Q = (scanner.get->printer.get  
      ->copy  
      ->scanner.put->printer.put  
      ->Q) .
```

```
||SYS = (p:P||q:Q  
         ||{p,q}::printer:RESOURCE  
         ||{p,q}::scanner:RESOURCE  
         ) .
```



- Deadlock Trace?**
- Avoidance?**

# deadlock analysis - parallel composition

Compiled: P

Compiled: Q

Compiled: RESOURCE

Composition:

```
SYS = p:P || q:Q || {p,q}::printer:RESOURCE ||  
{p,q}::scanner:RESOURCE
```

State Space:

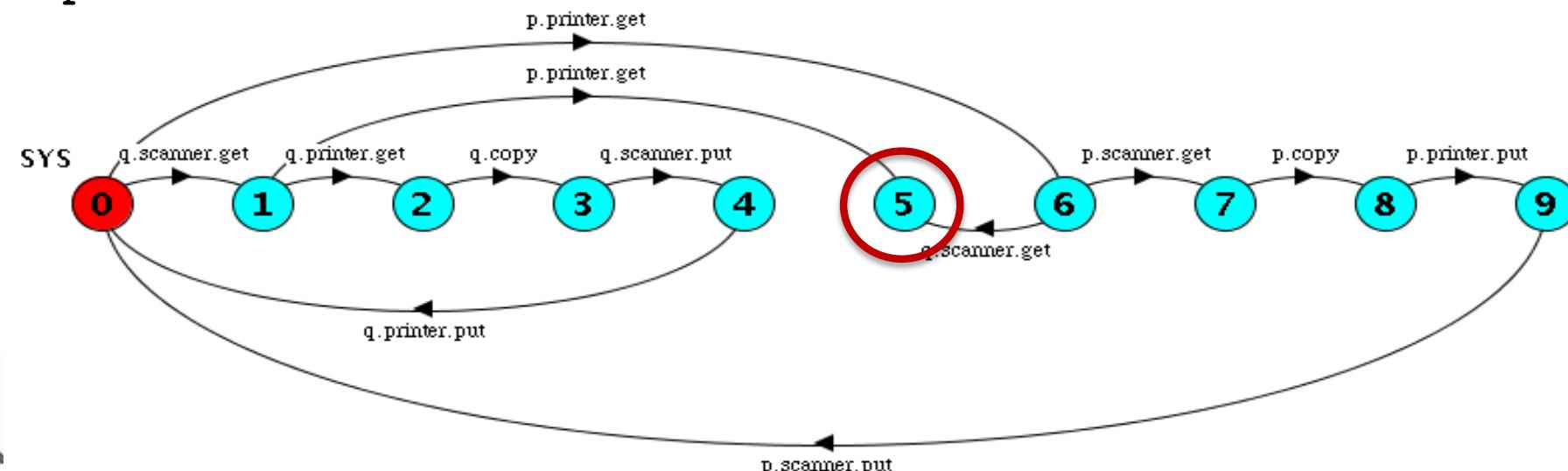
$$5 * 5 * 2 * 2 = 2 ** 8$$

Composing...

**potential DEADLOCK**

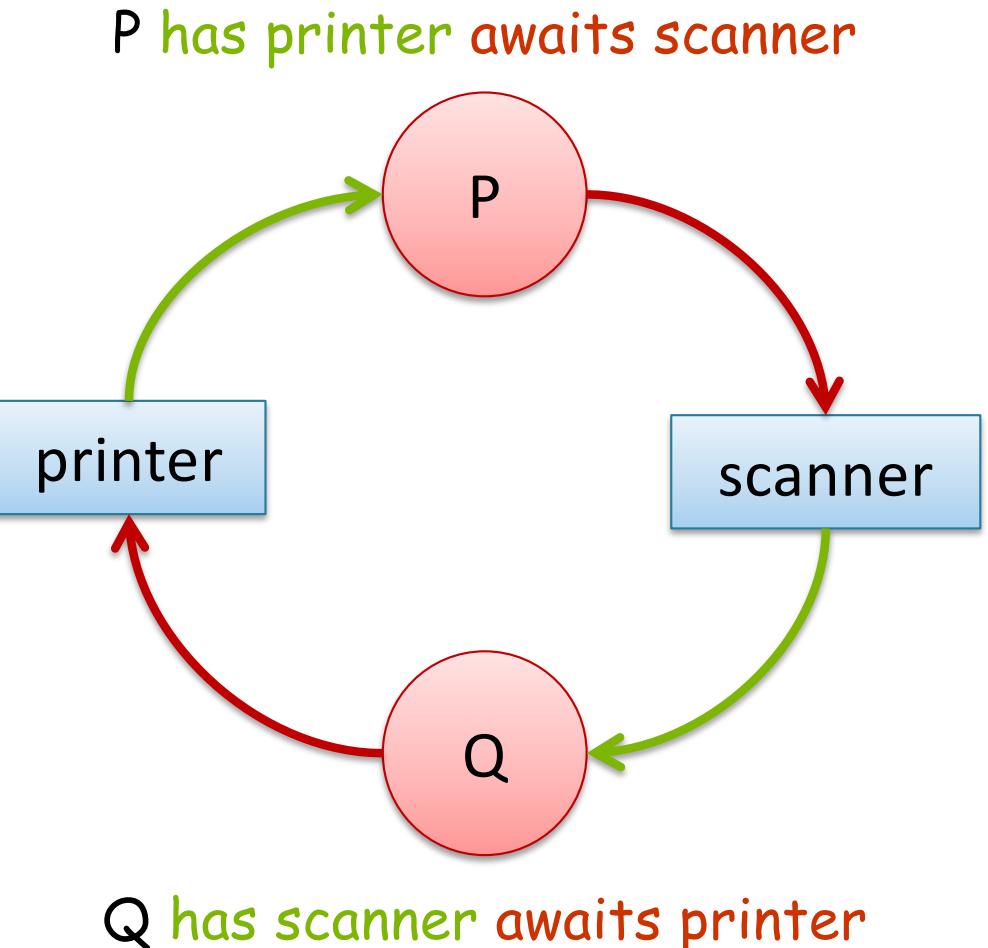
-- States: 10 Transitions: 12 Memory used: 3865K

Composed in 47ms



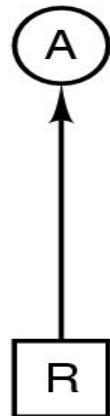
# deadlock analysis - four necessary and sufficient conditions

- ◆ **Serially reusable resources**
  - Printer and scanner are reusable
- ◆ **Incremental acquisition**
  - P takes the printer, then the scanner
  - Q takes the scanner, then the printer
- ◆ **No pre-emption**
  - the printer or the scanner can not be preempted
- ◆ **Wait-for cycle**



# Modélisation des interblocages (1)

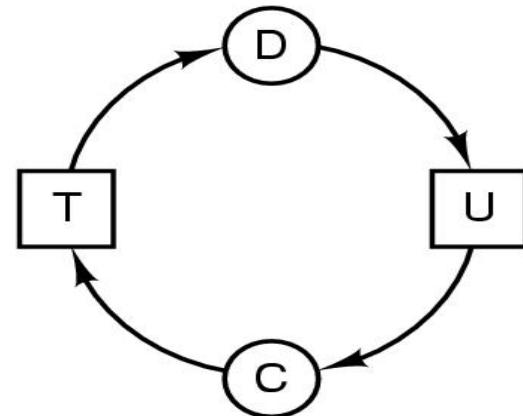
- Modélisation au moyen de graphes dirigés



(a)



(b)



(c)

La ressource R est détenue par le processus A

Le processus B attend après la ressource S

Les processus C and D sont en interblocage

# Interblockage - définition

- Définition formelle:  
*Un ensemble de processus est en interblockage si chacun d'eux attend un événement qui ne peut être provoqué que par un autre processus de l'ensemble.*
- L'événement attendu est habituellement la libération d'une ressource
- Aucun de ces processus ne peut ...
  - s'exécuter
  - libérer de ressources
  - être réveillé
- Peut-on éviter ce scénario catastrophe ?

# Interblocage : les approches possibles

## 1. La politique de l'autruche

- On ne fait rien

## 2. La prévention des interblocages

- On empêche

## 3. Détection des interblocages et leur guérison

- On détecte puis on agit

## 4. Evitement des interblocages

- On agit s'il y a un risque d'interblocage

# La politique de l'autruche

# La politique de l'autruche

- Ignorer le problème
- **Raisonnables si**
  - les interblocages se produisent très rarement ou peuvent être résolu par l'utilisateur
    - Tuer le processus / rebooter le système
  - le coût de la prévention est élevé
- Unix et Windows utilisent cette approche
  - Le système n'offre aucun mécanisme pour empêcher ou résoudre un inter-blocage
    - Unix et Windows considèrent que si le problème est important pour une application donnée
      - Alors il doit être traité par cette application
- Il existe un compromis entre
  - ce qui est pratique
  - ce qui est correct

# La prévention

# When ?

irréaliste

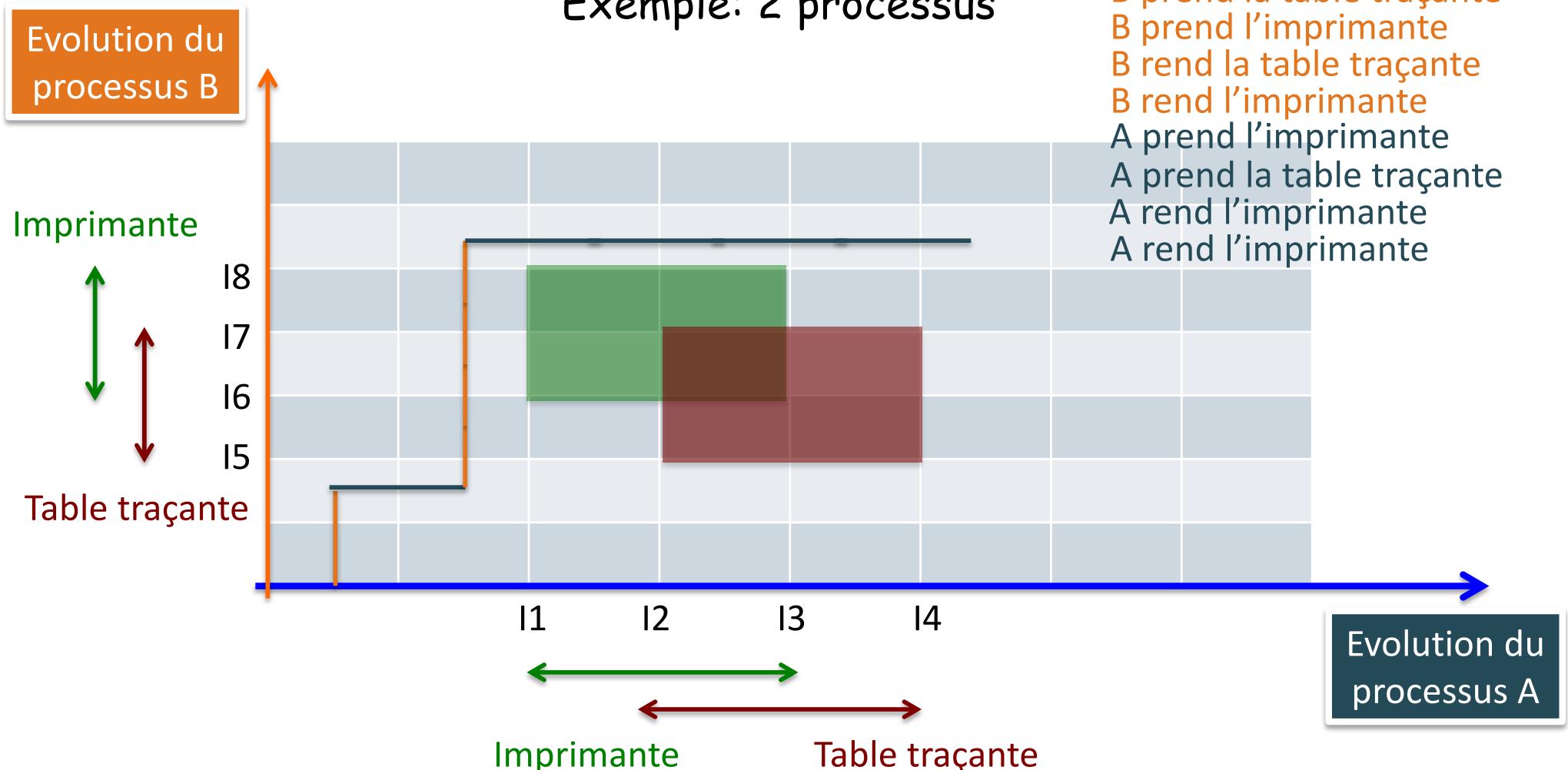
réaliste

- ◆ By preventing one of the four **necessary and sufficient conditions**
  - ◆ **Serially reusable resources:**  
*the processes involved share resources which they use under mutual exclusion.*
  - ◆ **Incremental acquisition:**  
*processes hold on to resources already allocated to them while waiting to acquire additional resources.*
  - ◆ **No pre-emption:**  
*once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.*
  - ◆ **Wait-for cycle:**  
*a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.*



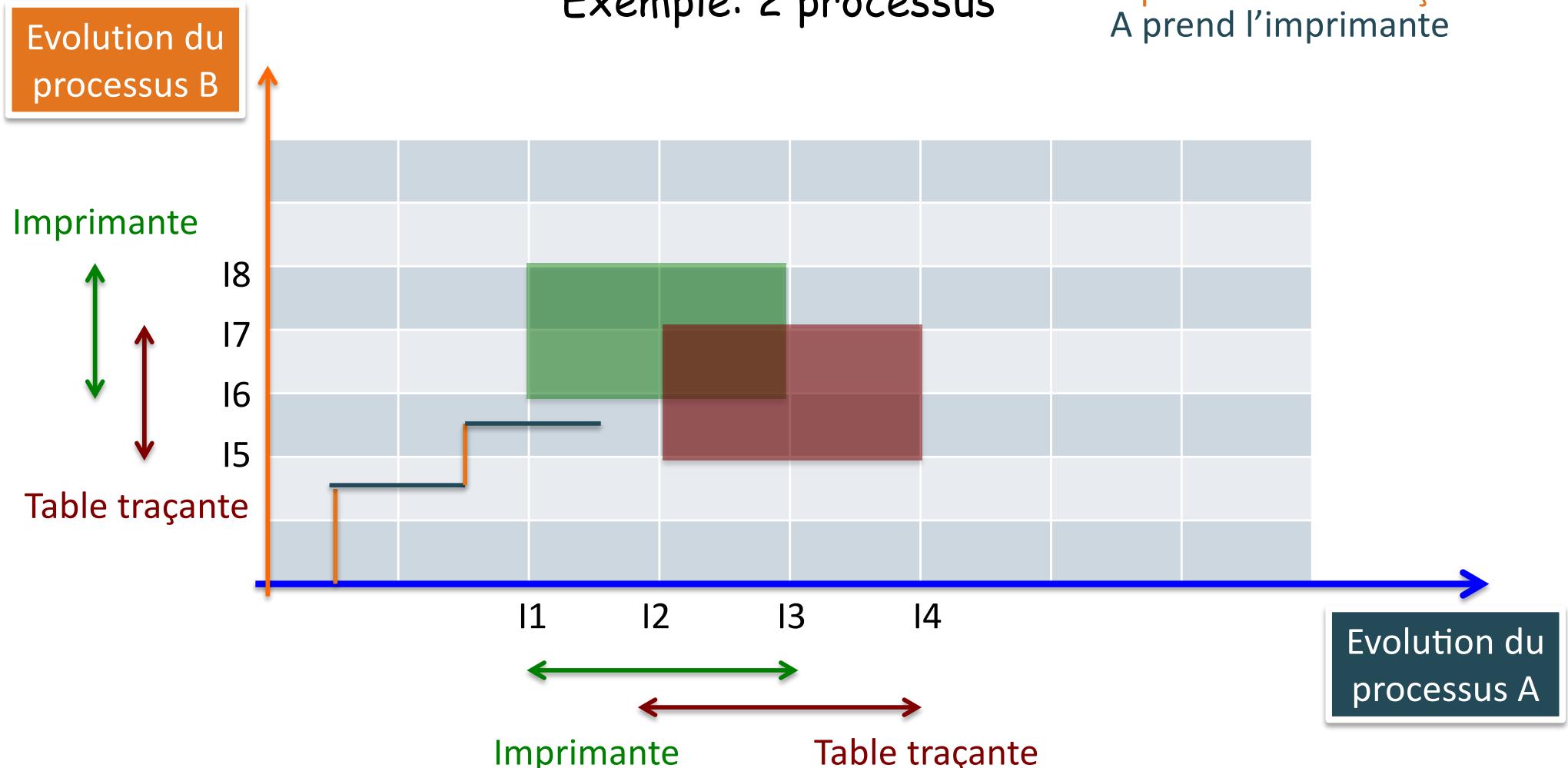
# Évitement des interblocages – Mise en évidence Trajectoires des ressources

Premier  
scénario



# Évitement des interblocages – Mise en évidence Trajectoires des ressources

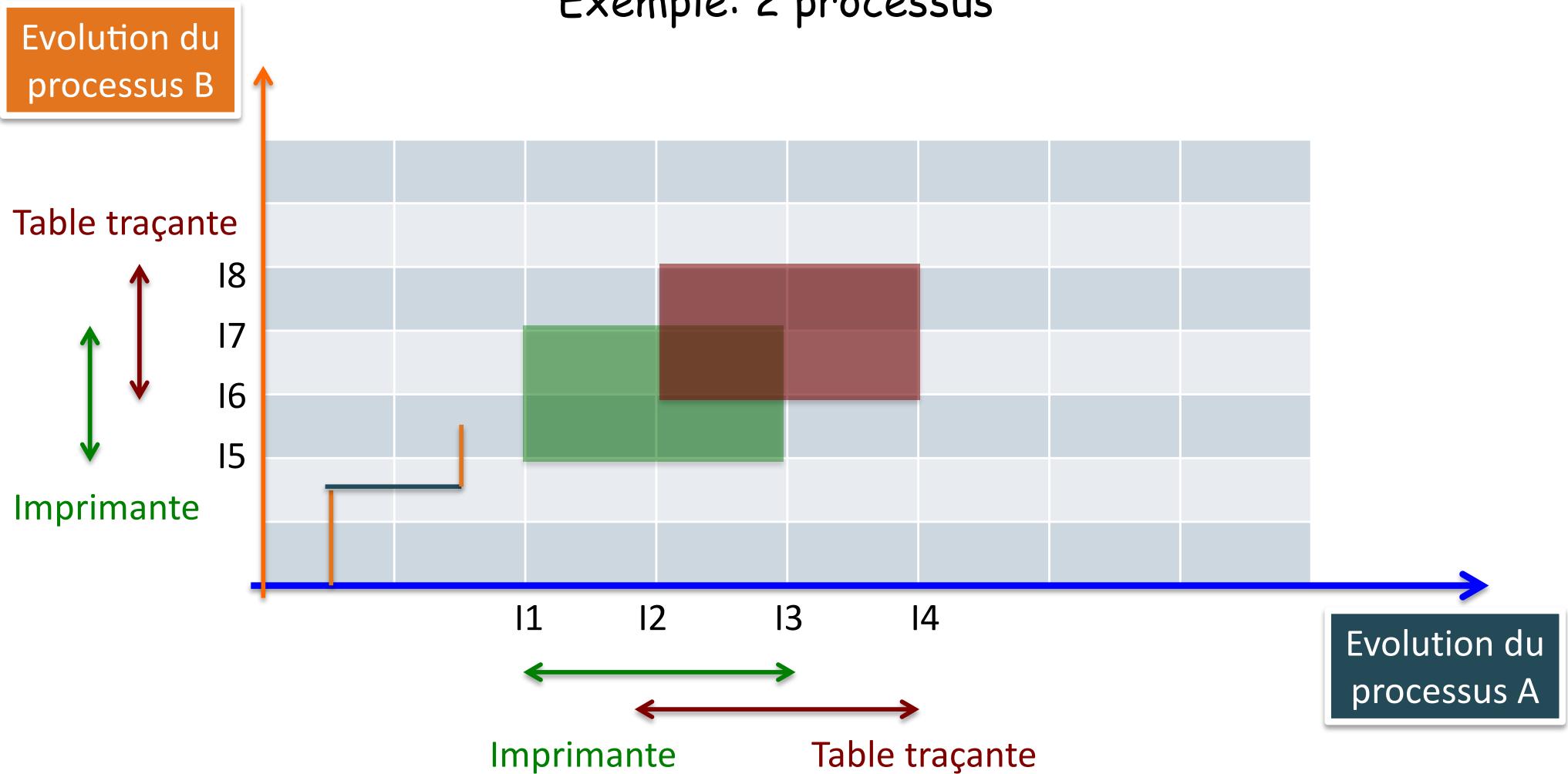
second  
scénario



# Évitement des interblocages

## Solution 1 : allocation ordonnée

Exemple: 2 processus



# deadlock analysis - parallel composition

- ◆ Each process acquire resources in different order?

RESOURCE = (get->put->RESOURCE) .

P = (printer.get->scanner.get  
->copy  
->printer.put->scanner.put  
->P) .

Q = (scanner.get->printer.get  
->copy  
->scanner.put->printer.put  
->Q) .

||SYS = (p:P || q:Q  
|| {p,q} :: printer:RESOURCE  
|| {p,q} :: scanner:RESOURCE  
).

# deadlock analysis - parallel composition

- ◆ Each process acquire order?

# deadlock analysis - avoidance

- Each process acquire resources in the same order?

RESOURCE = (get->put->RESOURCE) .

P = (printer.get->scanner.get  
->copy  
->printer.put->scanner.put  
->P) .

Q = (printer.get->scanner.get  
->copy  
->scanner.put->printer.put  
->Q) .

||SYS = (p:P || q:Q  
|| {p,q} :: printer:RESOURCE  
|| {p,q} :: scanner:RESOURCE  
) .

# deadlock analysis - avoidance

- Each process acquires its resources in the same order?

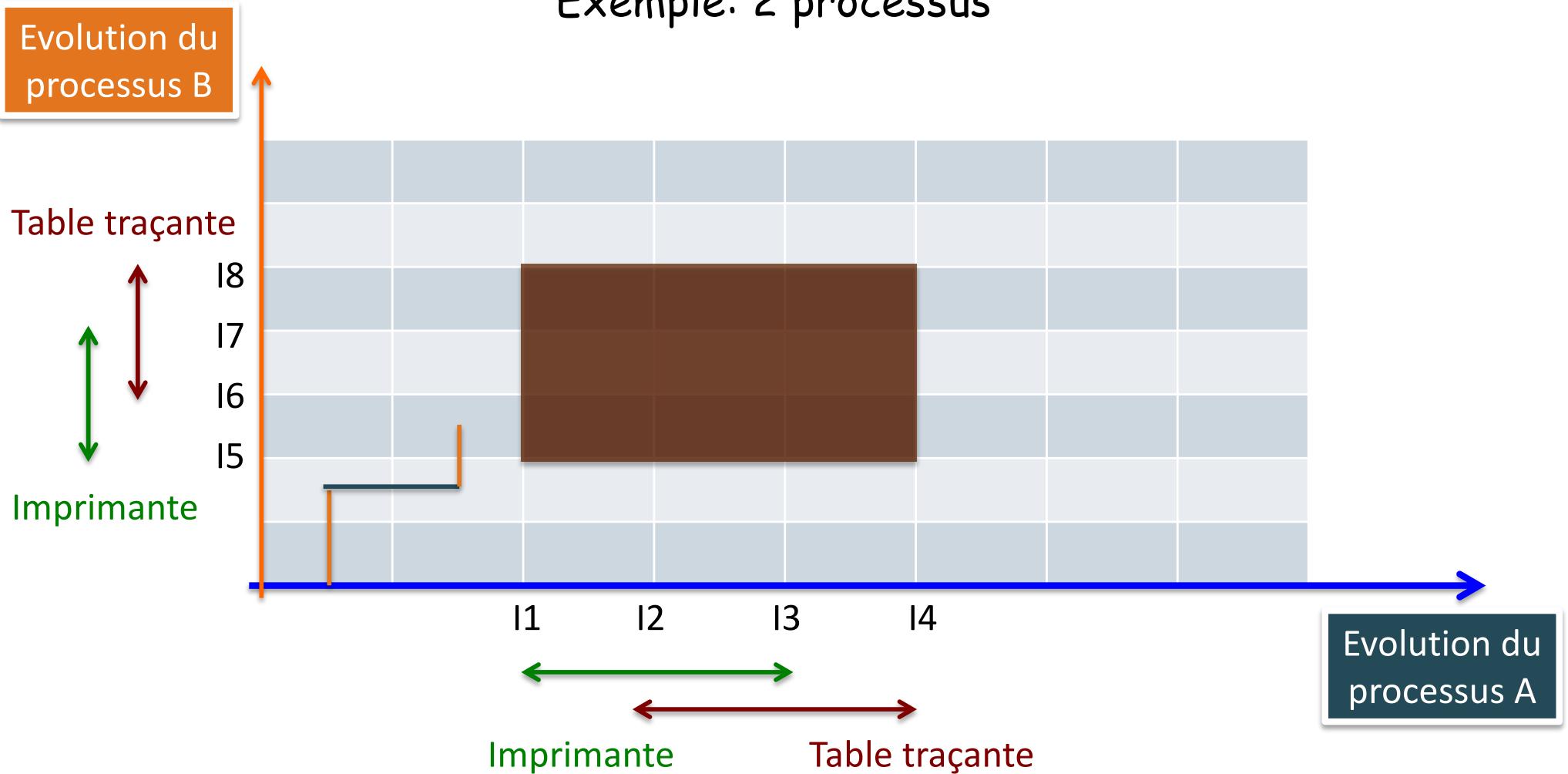
```
RESOURCE
P = Compiled: P
     -> Compiled: Q
     -> Compiled: RESOURCE
          Composition:
          {p,q}::printer:RESOURCE || q:Q ||
->P) SYS = p:P || {p,q}::scanner:RESOURCE ||
          {p,q}::scanner:RESOURCE || printer:RESOURCE ||
          State Space:
          5 * 5 * 2 * 2 = 2 ** 8
Composing...
-- States: 10 Transitions: 12
Memory used: 2739K
Composed in 37ms
||SYS = (p:P
||{p,q}::printer:RESOURCE
||{p,q}::scanner:RESOURCE
).
SOURCE
```

No deadlocks/errors

# Évitement des interblocages

## Solution 2 : allocation globale

Exemple: 2 processus



# Evitement des interblocages

## Solution 3 : algorithme du banquier

- Tout le monde connaît la prudence des banquiers
- Les processus sont comme des clients qui désirent emprunter de l'argent (ressources) à la banque...
- **Principe**
  - Garantir qu'au moins un processus peut toujours poursuivre son exécution
- **Définition**
  - On dit d'un **état** qu'il est **sain** s'il existe un ordonnancement selon lequel chaque processus peut s'exécuter jusqu'au bout,
  - **même si les processus demandent d'un seul coup toutes les ressources qui leurs sont nécessaires**

# Exemples d'états sains

	UTIL	MAX
A	3	9
B	2	4
C	2	7

Libres : 3  
(a)

- **Etat sain :**
  - Il existe  $i$ , tel que  $MAX[i]-UTIL[i] \leq$  libre
- Calcul des «  $MAX[i]-UTIL[i]$  »
  - A :  $9-3 = 6$
  - B :  $4-2 = 2 \leq 3$
  - C :  $7-2 = 5$
- L'état (a) est sain puisque le processus B peut terminer

# Exemples d'états non-sains

	UTIL	MAX
A	3	9
B	2	6
C	2	7

	UTIL	MAX
A	3	7
B	0	4
C	2	7

- Etat (b)
  - A :  $9-3 = 6$
  - B :  $6-2 = 4$
  - C :  $7-2 = 5$
- Etat (c)
  - A :  $7-3 = 4$
  - B :  $4-0 = 4$
  - C :  $7-2 = 5$
- Les états (b) et (c) ne sont pas sains

# Algorithme du banquier

- Cas simple : une seule catégorie de ressource disponible en grand nombre
- Principe : l'ordonnanceur n'accepte que les états sains
- Nécessite une **collaboration** entre **l'application** (qui définit les états sain) et **l'ordonnanceur** (qui passe d'un état sain à un autre).
  - Dans la pratique cela se limite au nombre de ressources nécessaires pour chacun des processus
  - 2 vecteurs + 1 variable + 1 constante
    - MAX[1..N] : description des besoins i.e. nombre de ressources MAX nécessaire à chaque processus
    - UTIL[1..N] : ressources utilisées par chacun des processus
    - libre : la quantité de ressource disponible
    - Max : nombre maximal d'exemplaire de la ressource

# Algorithme du banquier

Demande ( $P_i, N$ )

//  $i$  : numéro du processus

//  $N$  : nombre d'exemplaire demandé

// libre

Repete

    si ( $N \leq \text{libre}$ )

        alors

$\text{UTIL}[i] += N$  // simuler l'allocaiton de ressource

            bool  $\leftarrow$  vérifier si l'état atteind est sain

            si (bool)

                alors allouer ressource ;  $N \leftarrow 0$

                sinon  $\text{UTIL}[i] -= N$

            si (non bool) alors attendre

Tantque ( $N > 0$ )

# Résumé : Solution par Prévention

- Ensemble de solutions qui garantissent que les situations d'interblocage ne peuvent pas se produire.
- **Allocation ordonnée** : les ressources sont numérotés, un processus les acquière selon cet ordre. Dès qu'un processus libère une ressource, alors il ne peut plus en acquérir.
  - Inconvénient : garantir que les ressources soient acquises selon l'ordre établi
- **Allocation globale** : un processus acquiert l'ensemble de ses ressources simultanément et les libère petit à petit, dès qu'il n'en a plus besoin.
  - Avantage : plus besoin d'ordre
  - Inconvénient : minimise le parallélisme, risque accru de famine
- **Algorithme du banquier** : l'ordonnanceur connaît les besoins des processus et garanti à chaque allocation la terminaison d'au moins un processus
  - Avantage : minimise un peu moins le parallélisme que la solution précédente
  - Inconvénient : nécessite une collaboration entre ordonnanceur et application

# La détection / guérison

# When ? → Peut être mis en oeuvre

- ◆ By preventing one of the four **necessary and sufficient conditions**

- ◆ **Serially reusable resources:**

*the processes involved share resources which they use under mutual exclusion.*

- ◆ **Incremental acquisition:**

*processes hold on to resources already allocated to them while waiting to acquire additional resources.*

- ◆ **No pre-emption:**

*once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.*

- ◆ **Wait-for cycle:**

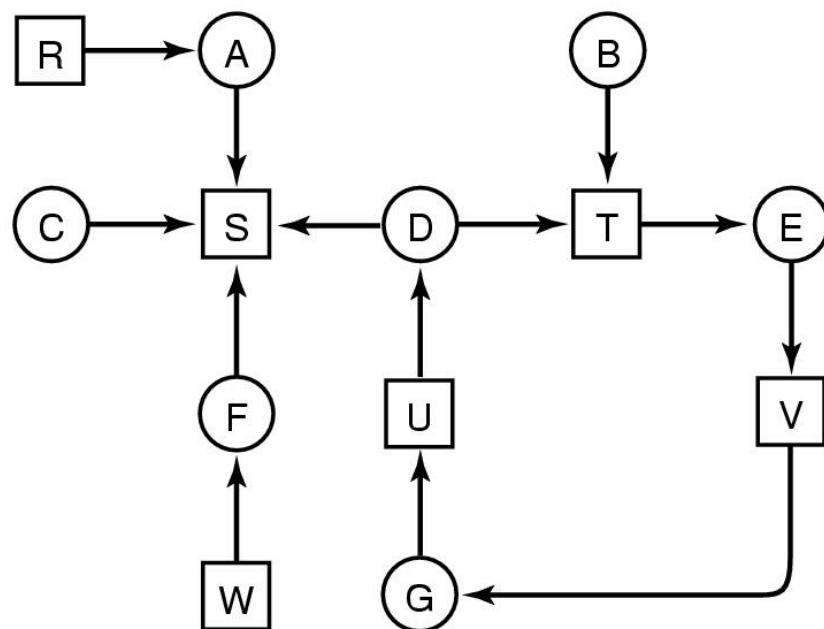
*a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.*



Déetecter le cycle ou empêcher sa construction

# Solution par Détection-Guérison

- Situation d'interblocage = cycle dans le graphe de dépendance



- Algorithme de recherche de cycle (un seul exemplaire de chaque ressource) :
  1. Pour chaque noeud N faire ce qui suit
  2. Initialiser L à une pile vide et désigner tous les arcs comme non marqués
  3. Empiler(N,L) et vérifiez s'il apparaît deux fois. Si oui, le graphe contient un cycle et on termine.
  4. Si N possède un arc sortant (N, M) non marqué alors Goto 5, sinon Goto 6
  5. Marquer l'arc (N,M), Empiler(M,L) et redéfinir N=M; Goto 3
  6. Si Vide(L) alors on arrête; sinon Dépiler(L) et N=Dessus(L); Goto 3

# Solution par Détection-Guérison

- Une fois le cycle détecté, il faut :
  - Choisir un processus impliqué dans l'interblocage
    - lequel ? N'importe lequel résoud le problème mais les principales stratégies choisissent :
      - Celui qui a produit le cycle.
      - Celui qui bloque le maximum de ressources, le plus vieux (libérer le plus de ressources possibles)
      - Celui qui bloque le moins de ressources, le plus récent (défaire le moins possible)
  - Réquisitionner toutes ses ressources
  - Défaire tout ce que le processus à fait
  - Redémarrer le processus au début de son exécution au début
- **Intérêt** : On traite uniquement les cas d'interblocage
- **Inconvénient** : surcoût pour maintenir le graph et détecter cycle

# Solution par Détection-Guérison

- Réquisitionner toutes ses ressources
  - Si les ressources sont préemptibles
    - Si un interblocage se produit : Suspendre un processus et prendre les ressources qu'il a acquises, le mécanisme d'allocation devant lui allouer à nouveau les ressources.
    - Exemple de ressources préemptible : Mémoire (dans le cadre d'un mécanisme de pagination)
  - Si les ressources ne sont pas préemptible mais il existe des points de reprise
    - Enregistrer l'état des processus à chaque point de reprise (rollback)
    - Si un interblocage se produit : Suspendre un processus et libérer les ressources qu'il a acquises depuis le point de reprise, puis reprendre le processus au point de reprise
  - Sinon supprimer le processus
    - Rudimentaire mais simple
    - Si un interblocage se produit : Arrêter un processus pour libérer ses ressources puis le redémarrer depuis le début

# Solution par Détection-Guérison

- Il faut dans la très grande majorité des cas, **défaire tout ce que le processus à fait**, puisqu'il va reprendre son exécution depuis un état antérieur
  - Pourquoi ? Un exemple ?
    - Proc A : réservation une chambre d'hôtel puis d'un avion
    - Proc B : réservation d'un avion puis d'une chambre d'hôtel
- pas même ordre donc interblocage possible...
- Le processus "arrêté" doit rendre la chambre ou le vol réservé

# Mise en oeuvre du rollback

- Un modèle : les transactions
  - Principalement utilisé avec les bases de données
  - Normalement étudié dans le cadre du cours « Base de données »
  - **Nous ferons donc un rapide rappel**
  - Mais peut être utilisé dans d'autres cas (sans base de données)
    - J2EE : certains composants
    - .Net : certains objets
- Les transactions permettent de traiter les problèmes suivants :
  - Panne de l'application
    - erreur en cours d'exécution du programme applicatif
    - nécessité de défaire les mises à jour effectuées
  - Panne système
    - reprise avec perte de la mémoire centrale
    - toutes les transactions en cours doivent être défaits
  - Mais pas la panne disque
    - perte des données

# Les transactions ACID

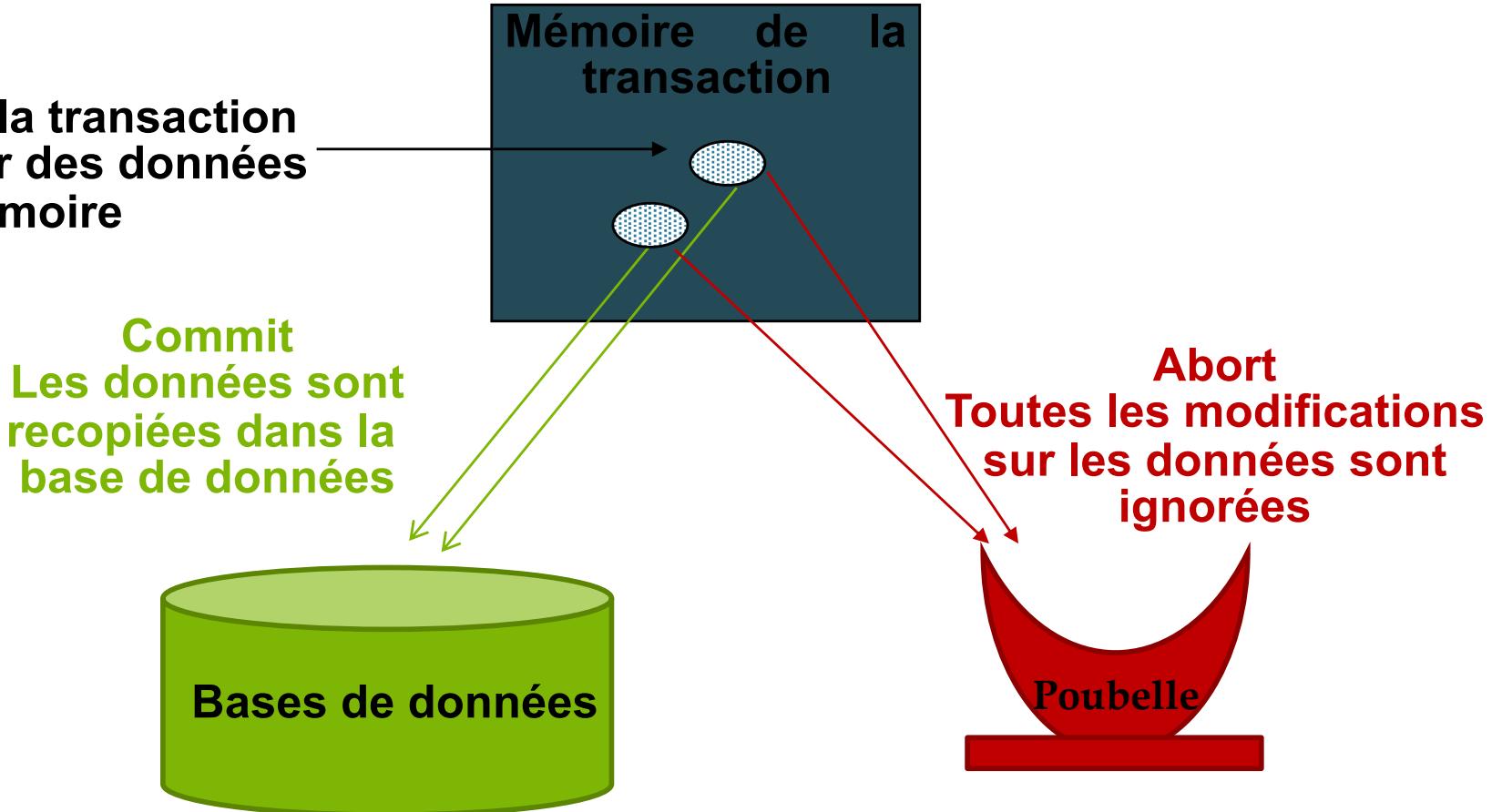
- Une transaction est une suite d'opérations (lectures - écritures) effectuées sur une base de données
- Les caractéristiques souhaitées pour l'exécution d'une transaction sont souvent résumées par l'acronyme **ACID**.
  - **Atomicité**
    - Unité de cohérence : toutes les mises à jour doivent être effectuées ou aucune.
  - **Cohérence**
    - La transaction doit faire passer les données d'un état cohérent à un autre.
    - Généralement : la base de données
  - **Isolation**
    - Les résultats d'une transaction ne sont visibles aux autres transactions qu'une fois la transaction validée.
  - **Durabilité**
    - Les modifications d'une transaction validée ne seront jamais perdue

# Terminaison correcte ou incorrecte d'une transaction

- Une transaction doit être soit:
  - Complètement exécutée : toutes les opérations ont été réalisées avec succès
  - Pas du tout exécutée la transaction ne doit pas avoir eu d'effet ni sur la base de données, ni sur les autres transactions
- Pour cela trois instructions ont été introduites
  - **Begin** (permet de marquer le début de la transaction)
  - **Commit** (marque la terminaison de la transaction avec succès)
    - Validation de la transaction
    - Rend effectives toutes les mises à jour de la transaction
  - **Abort** (fin avec échec)
    - Annulation de la transaction
    - Défait toutes les mises à jour de la transaction

# Effet logique

Au cours de la transaction  
la mise à jour des données  
se fait en mémoire



# Mise en œuvre des transactions

- Normalement vue dans le cours « Base de données »
- Au programme des cours
  - J2EE / JDBC / Spring → cf doc Oracle
  - .Net → cf doc Microsoft .Net

# Evitement et autres approches

# Evitements

- Les processus sont numérotés selon un ordre croissant à l'aide d'une estampille
- Plusieurs types d'algorithmes existent. En voici un.

```
if (Stamp(P1) < Stamp(P2) {  
    /* Le processus demandeur est plus ancien  
       que celui qui possède la ressource.  
       Il est mis en attente */  
    Wait(P1);  
} else {  
    /* le processus demandeur est plus jeune  
       que celui qui possède la ressource.  
       Il est arrêté. */  
    Stop(P1);  
}
```

→ Un processus n'attendra jamais sur un processus plus vieux : pas de cycle, pas d'interblocage

## *Autres solutions : Time Out*

- A chaque processus est associé un temps d'exécution
- Le dépassement de celui-ci est interprété comme une situation d'interblocage
- Le processus est alors arrêté, ses ressources réquisitionnées
- Intérêt : supprime la manipulation du graphe et les surcoûts induits
- Inconvénients : un processus peut être arrêté même s'il n'y a pas interblocage.
  - Un timeout trop court peut conduire à une privation
  - Un timeout trop long dégrade les performances du système
  - Et surtout, il faut savoir arrêter un processus

# deadlock analysis - avoidance

```
// Ajout d'un 'time out'  
RESOURCE = (get-  
(printer.get-  
GETSCANNER  
Compiled: P  
Compiled: Q  
Compiled: RESOURCE  
SYS = p:P || q:Q ||  
{p,q}::printer:RESOURCE ||  
{p,q}::scanner:RESOURCE  
State Space:  
6 * 6 * 2 * 2 * 2 = 2 ** 8  
Composing...  
-- States: 17 Transitions: 31  
Memory used: 3030K  
Composed in 32ms  
|| SYS = (p:P ||  
|| {p,q}::p:  
|| {p,q}::sc  
SOURCE  
put  
put->P  
= P  
put  
put->Q
```

deadlocks/errors

No deadlocks/errors

SOURCE

SOURCE

SOURCE

# Summary

## ◆ Concepts

- deadlock: no further progress
- four necessary and sufficient conditions:
  - ◆ serially reusable resources
  - ◆ incremental acquisition
  - ◆ no preemption
  - ◆ wait-for cycle

## ◆ Models

- no eligible actions (analysis gives shortest path trace)

## ◆ Practice

- blocked threads

 **Aim:** deadlock avoidance -  
to design systems where  
deadlock cannot occur.

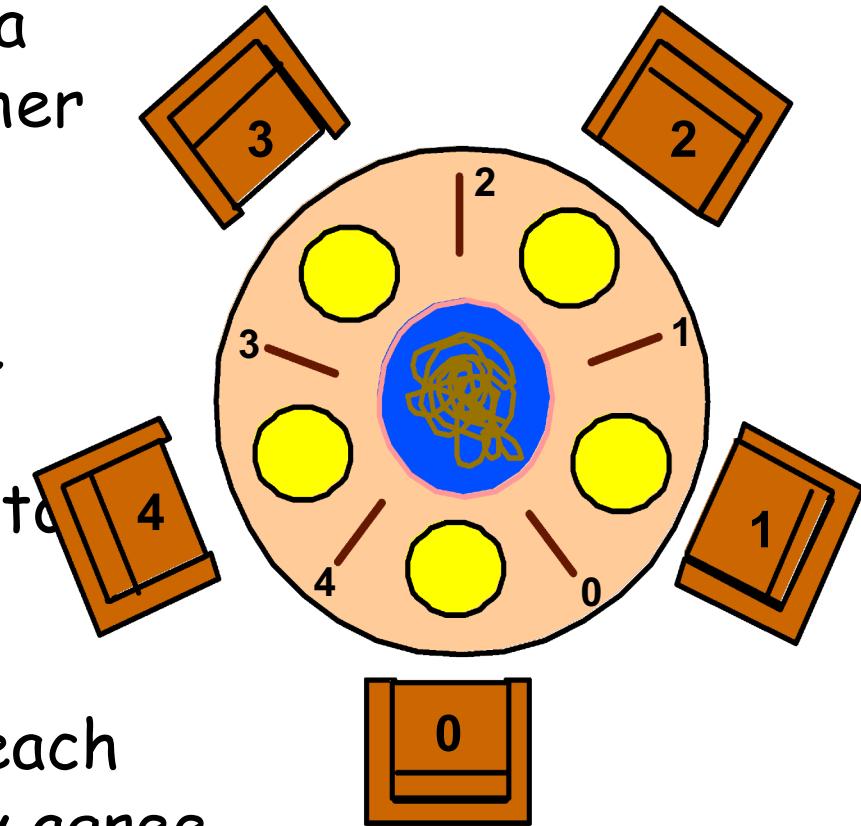
## Q&A

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



## 6.2 Dining Philosophers

Five philosophers sit around a circular table. Each philosopher spends his life alternately **thinking** and **eating**. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.

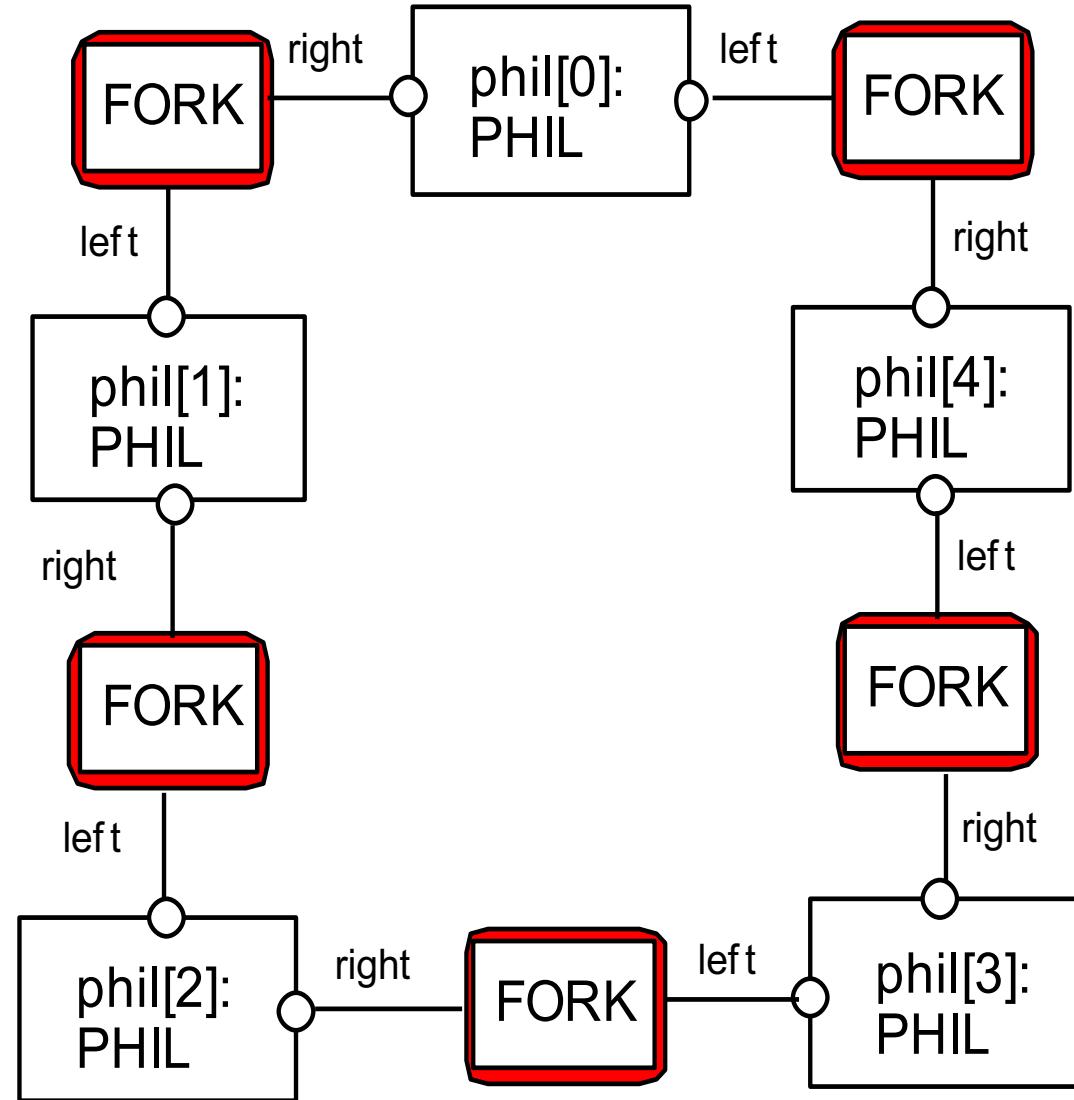


One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

# Dining Philosophers - model structure diagram

Each FORK is a  
**shared  
resource** with  
actions **get** and  
**put**.

When hungry,  
each PHIL  
must first get  
his right and  
left forks  
before he can  
start eating.



# Dining Philosophers - model

```
FORK = (get -> put -> FORK) .  
PHIL = (sitdown      ->right.get->left.get  
        ->eat         ->right.put->left.put  
        ->arise->PHIL) .
```

Table of philosophers:

```
||DINERS(N=5)= forall [i:0..N-1]  
  (phil[i]:PHIL ||  
   {phil[i].left,phil[((i-1)+N)%N].right}::FORK  
  ) .
```

*Can this system deadlock?*

# Dining Philosophers - model analysis

```
Trace to DEADLOCK:  
phil.0.sitdown  
phil.0.right.get  
phil.1.sitdown  
phil.1.right.get  
phil.2.sitdown  
phil.2.right.get  
phil.3.sitdown  
phil.3.right.get  
phil.4.sitdown  
phil.4.right.get
```

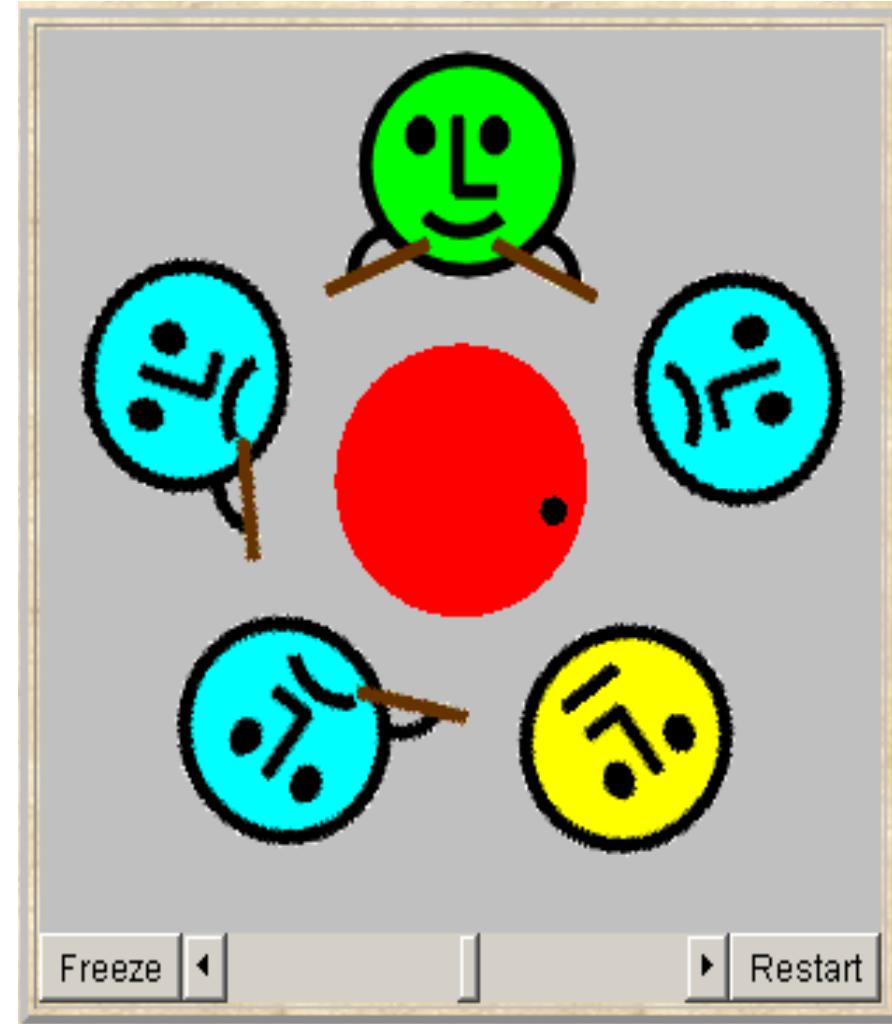
This is the situation where all the philosophers become hungry at the same time, sit down at the table and each philosopher picks up the fork to his **right**.

The system can make no further progress since each philosopher is waiting for a fork held by his neighbor i.e. a **wait-for cycle** exists!

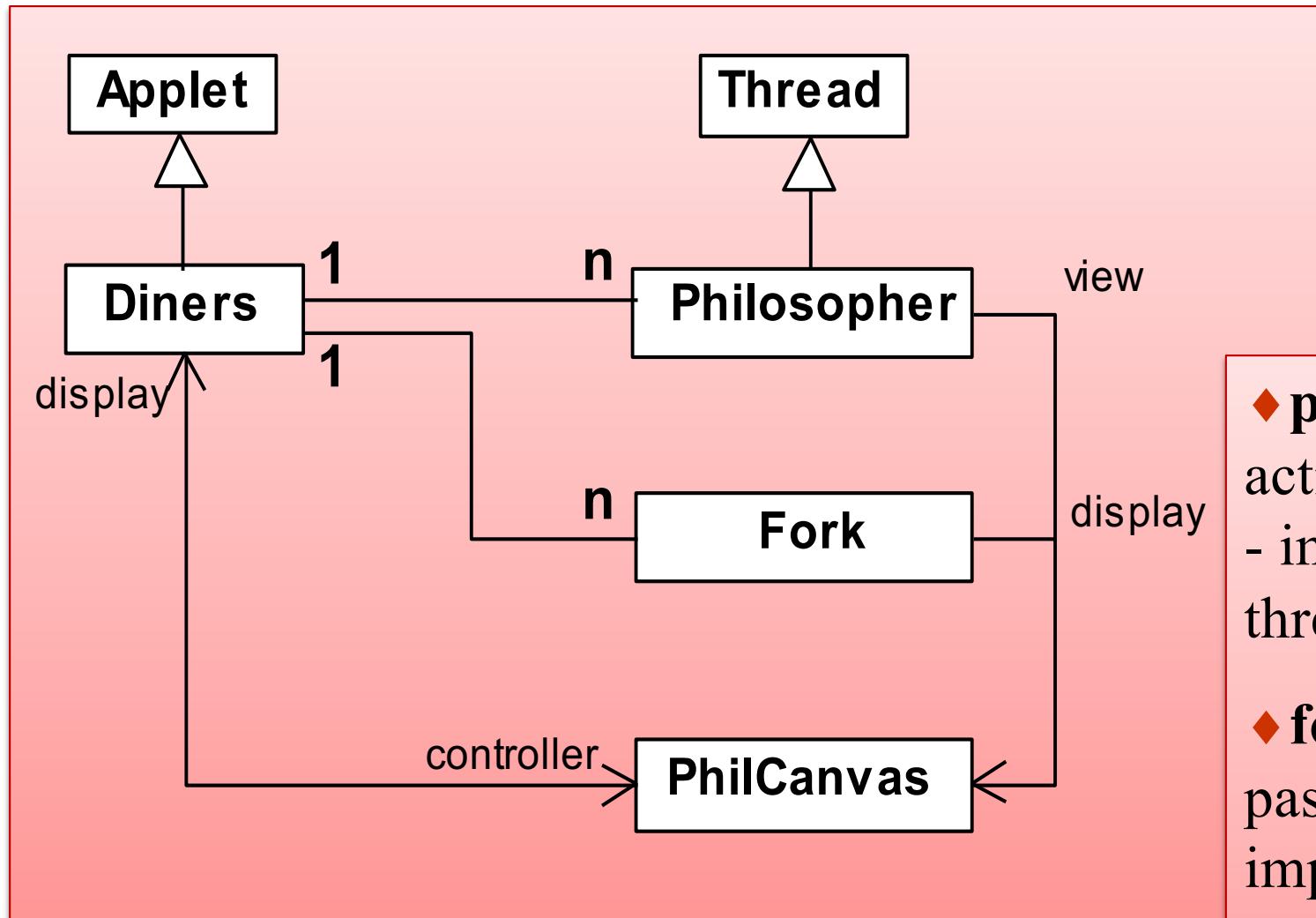
# Dining Philosophers

Deadlock is easily detected in our model.

*How easy is it to detect a potential deadlock in an implementation?*



# Dining Philosophers - implementation in Java



- ◆ **philosophers:** active entities
  - implement as threads
- ◆ **forks:** shared passive entities
  - implement as monitors
- ◆ **display**

# Dining Philosophers - Fork monitor

```
class Fork {  
    private boolean taken=false;  
    private PhilCanvas display;  
    private int identity;  
  
    Fork(PhilCanvas disp, int id)  
        { display = disp; identity = id; }  
  
    synchronized void put() {  
        taken=false;  
        display.setFork(identity,taken);  
        notify();  
    }  
  
    synchronized void get()  
        throws java.lang.InterruptedException {  
        while (taken) wait();  
        taken=true;  
        display.setFork(identity,taken);  
    }  
}
```

taken  
encodes  
the state of  
the fork

# Dining Philosophers - Philosopher implementation

```
class Philosopher extends Thread {  
    ...  
    public void run() {  
        try {  
            while (true) {  
                view.setPhil(identity,view.THINKING);  
                sleep(controller.sleepTime()); // hungry  
                view.setPhil(identity,view.HUNGRY);  
                right.get(); // got right chopstick  
                view.setPhil(identity,view.GOTRIGHT);  
                sleep(500);  
                left.get(); // eating  
                view.setPhil(identity,view.EATING);  
                sleep(controller.eatTime());  
                right.put();  
                left.put();  
            }  
        } catch (java.lang.InterruptedException e) {}  
    }  
}
```

Follows from the model (sitting down and leaving the table have been omitted).

# Dining Philosophers - implementation in Java

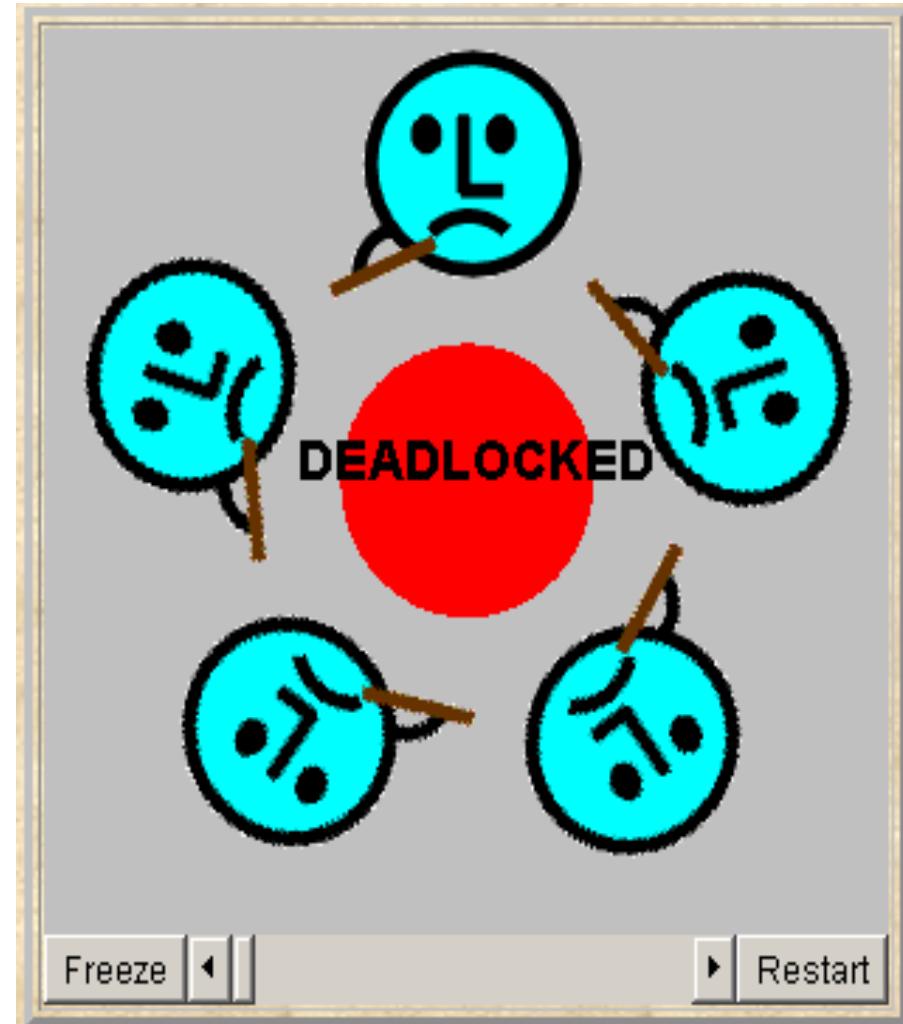
Code to create the philosopher threads and fork monitors:

```
for (int i =0; i<N; ++i)
    fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
    phil[i] =
        new Philosopher
            (this,i,fork[(i-1+N)%N],fork[i]);
    phil[i].start();
}
```

# Dining Philosophers

To ensure deadlock occurs eventually, the slider control may be moved to the left. This reduces the time each philosopher spends thinking and eating.

This "speedup" increases the probability of deadlock occurring.



# Deadlock-free Philosophers

Deadlock can be avoided by ensuring that a wait-for cycle cannot exist. **How?**

Introduce an **asymmetry** (take ressources in same order) into our definition of philosophers.

Use the identity I of a philosopher to make even numbered philosophers get their left forks first, odd their right first.

**Other strategies?**

```
PHIL(I=0)
  = (when (I%2==0) sitdown
      ->left.get->right.get
      ->eat
      ->left.put->right.put
      ->arise->PHIL
    | when (I%2==1) sitdown
      ->right.get->left.get
      ->eat
      ->left.put->right.put
      ->arise->PHIL
    ) .
```

## Q&A

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

