Programmation concurrente

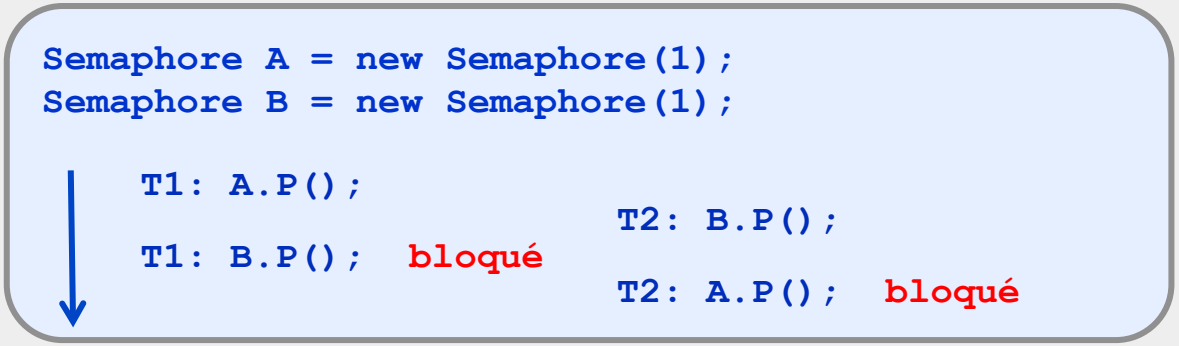
Gestion des Interblocages

Polytech/INFO 4, 2022-2023

Fabienne Boyer
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

Les interblocages

- **Interblocage : situation dans laquelle certains threads (ou processus) d'une application se retrouvent bloqués indéfiniment**



- **4 approches (certaines complémentaires)**
  - Prévention : prévenir (au maximum) les interblocages en contraignant la programmation
  - Evitement : durant l'exécution, vérifier certains invariants avant d'allouer une ressource
  - Détection-Guérison : laisser les interblocages se produirent et les guérir
  - Kill-Restart : laisser les interblocages se produire et supposer que l'administrateur fera un kill-restart de l'application

Un interblocage provient t'il d'un bug de programmation?

- **Parfois oui**

```
void produce(Msg msg) {    Msg consume() {
 mutex.P();                 mutex.P();
 notFull.P();               notEmpty.P();
 buffer[in++]=msg;          msg = buffer[out++];
 notEmpty.V();              notFull.V();
 mutex.V();                 mutex.V();
}                           }
```

→ Interblocage dès qu'un producteur (resp. consommateur) se bloque sur notFull.P() (resp. notEmpty.P())

→ L'interblocage est ici du à une **erreur de programmation** (on ne doit jamais se bloquer dans une section critique)

- **Parfois non**
  - Il faut donc des approches pour les traiter

Caractérisation des interblocages

- **La principale condition *nécessaire* pour qu'un interblocage se produise est la formation d'un cycle dans le graphe des attentes**

  → il existe des threads (ou processus) $\{T_0, T_1, \ldots, T_n\}$ tel que :

  T_0 **attend une ressource possédée par** T_1,

  T_1 " " T_2,

  T_{n-1} " " T_n,

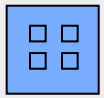
  et T_n " " T_0.

- **Les interblocages résultent donc du fait que des threads partagent plusieurs ressources**

Graphe des attentes

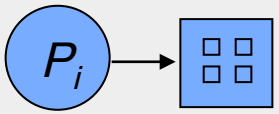
- **Ensemble des Threads/Processus**
  $P = \{P_1, P_2, \ldots, P_n\}$,

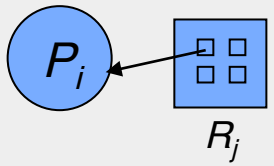
- **Ensemble des Ressources**
  $R = \{R_1, R_2, \ldots, R_m\}$*



- **P_i demande R_j**
  $P_i \rightarrow R_j$

- **P_i détient R_j**
  $R_j \rightarrow P_i$

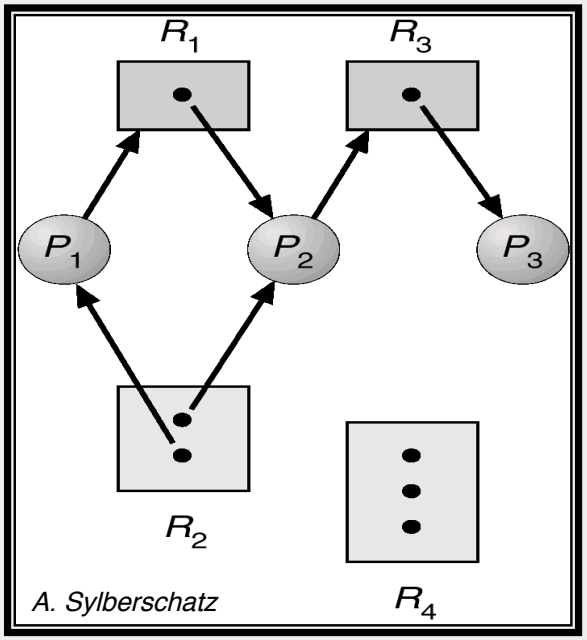




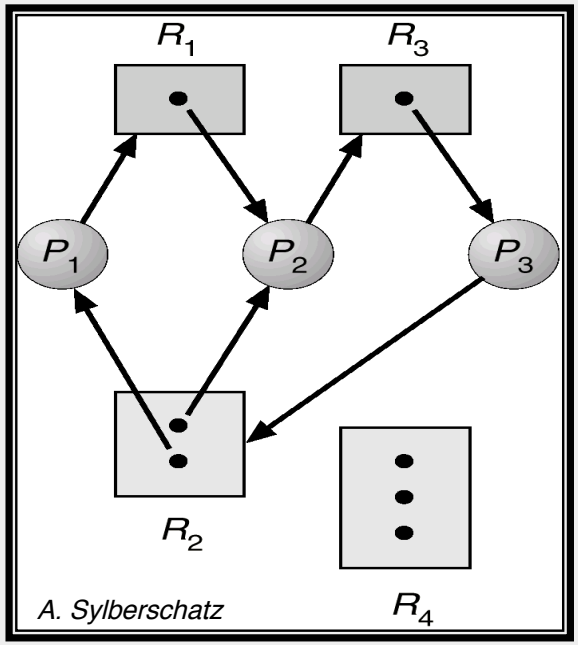
**** Ressources multiples autorisées***

Exemples de graphes des attentes

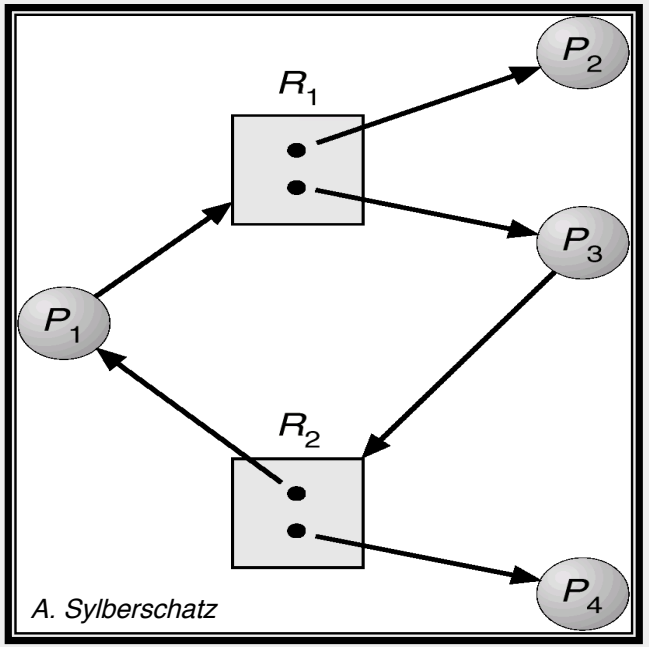
Pas d'interblocage



Interblocage



Pas d'interblocage (malgré le cycle)



Pas de cycle dans le graphe des attentes ⇒ pas d'interblocage

Cycle dans le graphe des attentes

⇒ interblocage (si ressources simples)

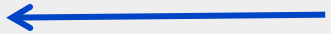
⇒ interblocage possible (si ressources multiples)

Technique 1: prévention des interblocages

- **Prévenir la formation de cycles dans le graphe des attentes**
  - Règles de bonne programmation
  - Allocation de ressources globale ou ordonnée
    → Suppose qu'un thread connaisse à l'avance les ressources qu'il va utiliser

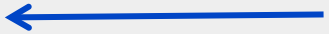
1ère règle de bonne programmation

```
mutex.P();
if …
      S.P();
      …
else
      …
      S.V();
mutex.V();
```



Interblocage probable si un thread se bloque dans le sémaphore S sans libérer la section critique

```
mutex.P();
if …
      mutex.V();
      S.P();
      mutex.P();
      …
else
       …
       S.V();
mutex.V();
```



REGLE : jamais de blocage dans une Section Critique sans libérer la Section Critique

2ème règle de bonne programmation

- **Garantir qu'un verrou est systématiquement relâché**

```
class C {
   private ReentrantLock l = new
ReentrantLock();
    ...

   public void m() {
      …
      l.lock();
      try {
          …
          …
          ...
      }  catch (Exception e) { ... }
      } finally { l.unlock(); }
}
…
```

Un verrou jamais relâché peut mener à un interblocage

La clause finally garantit le relâchement systématique du verrou

Allocation globale des ressources

- Aussi appelé Conservative Two Phase Locking (C2PL)
- Chaque thread acquiert ***simultanément*** toutes les ressources dont il a besoin (tout ou rien)
- Inconvénient: peut mener à une utilisation réduite des ressources

Considérons l'exemple de l'application bancaire

- **On verrouille un compte avant de le manipuler**

```
class Account {
 protected Semaphore acc = new Semaphore(1);

 public void Transfer (Account to, float amount){
    this.acc.P();
    to.acc.P();
    this.bal -= amount;
    to.bal += amount;
    this.acc.V();
    to.acc.V();
```

Interblocage possible si Transfer(C1, C2, ..) // Transfer (C2, C1, ..)

Allocation globale dans le cas du compte bancaire

```
Transfert(account from, account to, amount) {

 mutex.P();
 while (from.locked() || to.locked()) {
  nwaiting++; mutex.V(); waiting.P(); mutex.P();
 }
 from.lock(); to.lock();
 mutex.V();

 from.bal -= amount;
 to.bal += amount;

mutex.P();
from.unlock(); to.unlock();
// wake up all the waiting threads
for ( ;nwaiting>0; nwaiting--){
    waiting.V();
}
mutex.V();
}
```

acquisition globale des ressources

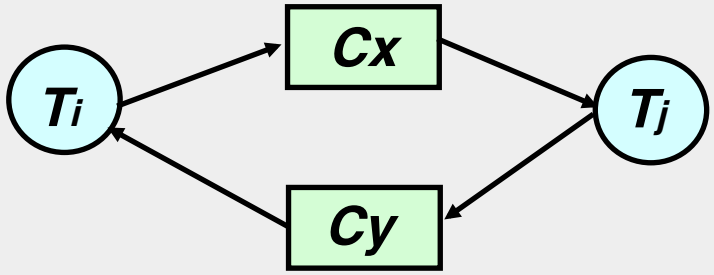
```
Interface Account {
    protected boolean locked();
    protected void lock();
    protected void unlock();
    public void credit(..);
    public void debit((..);
    public void transfer(..);
}
```

Allocation globale des ressources

- **Technique très utilisée dans les bases de données**
- **2 variantes**
  - Conservative 2-phase locking (C2-PL)
    - All locks acquired simultaneously
  - Basic 2-phase locking (B2-PL)
    - Expanding phase (*locks may be acquired but none can be released)*
    - Shrinking phase (*locks can be released but no new lock can be acquired)*
- **C2-PL moins performant que B2-PL mais garantit absence d'interblocages, Basic 2PL garantit seulement la sérialisabilité de l'exécution**

Allocation ordonnée des ressources

- **Chaque thread demande des ressources selon l'ordre croissant de leurs identifiants**
  - Suppose que l'on dispose d'un ordre total sur les identifiants de ressources



cycle impossible si ordre d'allocation = Cx puis Cy (x < y)

Allocation ordonnée dans le cas de l'application bancaire

```
class Account {
 protected Semaphore acc = new Semaphore(1);

 public void Transfer (Account to, float
    amount){
    if smaller(this, to){
       this.acc.P(); to.acc.P();
    } else {
       to.acc.P(); this.acc.P();
    }
    this.bal -= amount;
    to.bal += amount;
    this.acc.V();
    to.acc.V();
```

Interblocage impossible même si Transfer(C1, C2, ..) // Transfer (C2, C1, ..)

Prévention des interblocages

- L'allocation globale ou ordonnée n'est pas toujours utilisable
- En effet, ces types d'allocation impliquent que tout thread connaisse à l'avance les ressources qu'il va utiliser

Technique 2: détection/guérison des interblocages

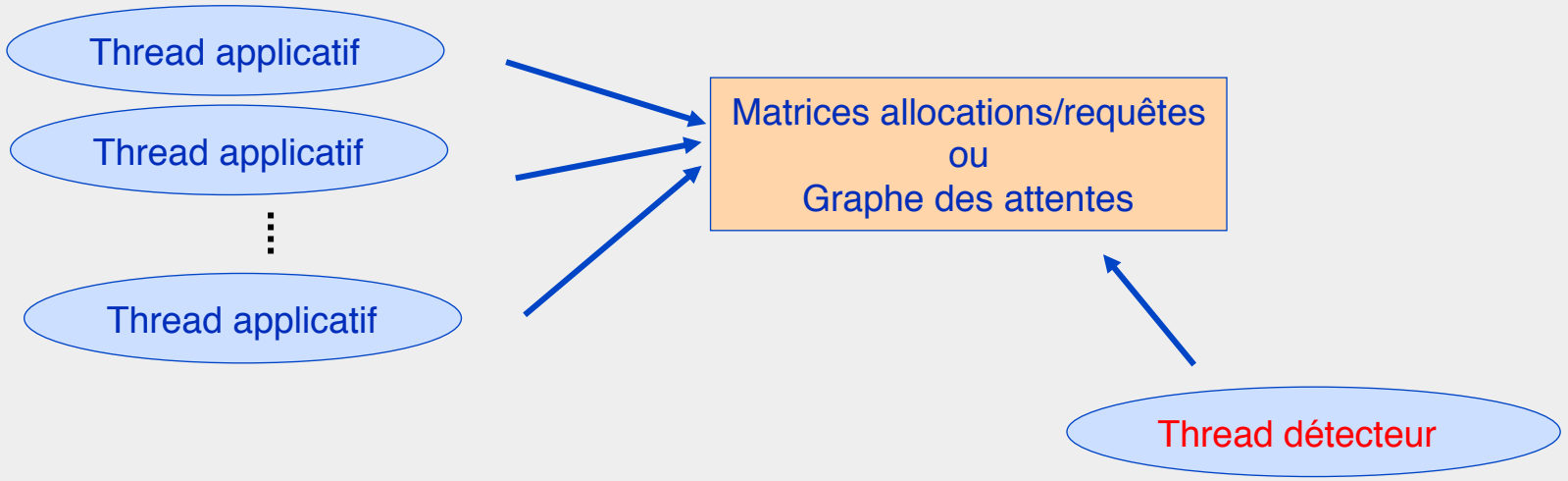
- **Laisse le système entrer en interblocage**
- **Système de détection (thread *démon*)**
  - Ressources simples: recherche un cycle dans le graphe des attentes
  - Ressources multiples: algo de détection plus compliqué, appliqué sur les matrices des allocations/requêtes

```
// daemon thread, in charge of deadlock detection
while (true) {
        < détecter interblocage >
        < traiter interblocage >
        < attendre délai >
}
```

Structure de données pour la détection

- **Structure de données mise à jour à chaque demande, allocation et libération de ressources**
- **Thread détecteur analyse la structure de donnée régulièrement**

→ Coût en CPU et en mémoire



Structure de données dans le cas de ressources multiples

- Hypothèses:
  - m = nombre de types de ressources
  - n = nombre de threads de l'application

- *vecteur Available:* indique le nombre de ressources libres de chaque type.
- *matrice Allocation:* definit le nombre de ressources de chaque type allouées à chaque thread à l'instant courant.
- *matrice Request:* indique les demandes en cours de chaque thread. Si *Request [i,j]= k*, alors le thread T_i demande *k* instances supplémentaires de la ressource de type R_j.

Exemple de matrices de détection

- 5 threads T_0 .. T_4
- 3 types de ressources
  A (7 instances), *B* (2 instances), et *C* (6 instances)
- ***Etat courant:***

	Allocation	***Request***	***Available***
	A B C	***A B C***	***A B C***
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

A l'instant courant, il reste 0 instances des ressources A, B et C

A l'instant courant, T1 possède 2 instances de la ressource A et il demande 2 instances supplémentaires de A et 2 instances de B

Algorithme de détection

1. Initialize two vectors, Work et Finish, as follow:

Work = Available *// ressources disponibles*

for i = 0 to n-1

if Allocation[i] ≠ 0..0 then Finish[i] = false; *// thread i possède encore des ressources*

else Finish[i] = true;

2. Find a thread Ti such that Finish[i] == false and Request[i]≤ Work

If no such thread exists, skip to step 4.

3. Work = Work + Allocation[i]

Finish[i] = true

go to step 2.

4. If Finish[i] == false for some i, then the system is in deadlock state.

→ On cherche une séquence permettant à chaque thread de poursuivre son exécution

Exemple de détection

- 3 types de ressources : A (7 instances), *B* (2 instances), et *C* (6 instances)
- *Etat courant:*

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

→ **Séquence <T_0, T_2, T_3, T_1, T_4> résulte en Finish[i] = true pour tout i, donc pas d'interblocage à l'instant courant**

- → *Work = 000*
- → T_0 : $Request_0$ <= Work → $Work+=Allocation_0$ → Work=010, $Finish_0$ = true
- → T_2 : $Request_2$ <= Work → $Work+=Allocation_2$ → Work=313, $Finish_2$ = true
- → T_3 : $Request_3$ <= Work → $Work+=Allocation_3$ → Work=524, $Finish_3$ = true
- → T_1 : $Request_1$ <= Work → $Work+=Allocation_1$ → Work=724, $Finish_1$ = true
- → T_4 : $Request_4$ <= Work → $Work+=Allocation_4$ → Work=726, $Finish_4$ = true

Exemple de détection

- ***Supposons que T_2* demande une instance supplémentaire de type C**

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 1	
T_2	3 0 3	0 0 1	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

→ **Pas possible de trouver une séquence qui résulte en Finish[i] = true pour tout i, il y a interblocage des threads T1, T2, T3 et T4 à l'instant courant**

- → *Work = 000*
- → T_0 : $Request_0$ <= Work → Work+=$Allocation_0$ → Work=010, $Finish_0$ = true
- → T_2 : $Request_2$ > Work
- → T_3 : $Request_3$ > Work
- → T_1 : $Request_1$ > Work
- → T_4 : $Request_4$ > Work

Guérison

- **Aborter (interrompre l'exécution)**
  - L'application complète (kill-restart)
  - Tous les threads impliqués dans l'interblocage
  - Certains threads seulement
    - On peut tenir compte de leur priorité, âge, etc.

- **Suppose que les thread interrompus sont capables de libérer leurs ressources puis de reprendre leur exécution**
  - C'est ce qu'on appelle un *rollback*
  - Peut reposer sur un journal des opérations

Technique 3: évitement des interblocages

- On garantit une exécution sans interblocage
- Pour cela, chaque thread doit annoncer le nombre maximum de ressources de chaque type qu'il peut utiliser durant son exécution
- A chaque demande de ressource, si la ressource est disponible, un *algorithme d'évitement* examine l'état du système et donne son accord seulement si l'allocation de la ressource laisse le système dans un état sûr
- Un état sûr est un état dans lequel on est sûr que l'application pourra terminer son exécution sans interblocage (même si chaque thread demande le maximum de son annonce)

Structure de données dans le cas de ressources multiples

- Hypothèses:
  - m = nombre de types de ressources
  - n = nombre de threads de l'application

- *vecteur Available:* indique le nombre de ressources libres de chaque type.
- *matrice Max:* définit le maximum de ressources de chaque type que chaque thread est susceptible de demander durant son exécution.
- *matrice Allocation:* définit le nombre de ressources de chaque type allouées à chaque thread à l'instant courant.
- *matrice Need:* définit le maximum de ressources de chaque type que chaque thread est susceptible de demander *durant le reste* de son exécution.

(Need[i,j] = Max[I,j] – Allocation[i,j])

Evitement des interblocages

Lorsque T_i demande k instances de la ressource de type R_j :

1. Si $Request_i \leq Need_i$ aller en 2
   Sinon, erreur (Pi excède sa demande max)

2. Si $Request_i \leq Available$, aller en 3.
   Sinon P_i attend (ressources non disponibles)

3. Essaie d'allouer les ressources à Pi en modifiant l'état comme suit :

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

   - Si état sûr $\Rightarrow$ ressources allouées à P_i.
   - Sinon $\Rightarrow$ P_i attend, et l'ancien état (Available, $Allocation_i$ et $Need_i$) est restauré

Algorithme qui détermine si le système est dans un état sûr

1. Initialize two vectors, Work et Finish, as follow:

Work = Available

for i = 0 to n-1: Finish[i] = false;

2. Find a thread Ti such that Finish[i] == false and Need[i]≤ Work

If no such thread exists, skip to step 4.

3. Work = Work + Allocation[i]

Finish[i] = true

go to step 2.

4. If Finish[i] == true for some i, then the system is in a safe state.

Exemple : T1 demande (1, 2 ,1)

- 5 threads $T_{0...}$ T_4;
- 3 ressources de type *A* (10 instances), *B* (5instances), and *C* (7 instances).
- *Etat courant:*

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
T_0	0 1 0	7 5 5	3 3 2	7 4 5
T_1	2 0 0	3 2 2		1 2 2
T_2	3 0 2	9 0 2		6 0 0
T_3	2 1 1	2 2 2		0 1 1
T_4	0 0 2	4 3 3		4 3 1

A l'instant courant, il reste 3 instances de A et B et 2 instances de C

T3 est suceptible de demander encore 1 instance de A et 1 instance de B

→ Le système reste t'il dans un état sûr si on alloue (1A, 2B, 1C) à T1?

Exemple : T1 demande (1, 2 ,1)

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
T_0	0 1 0	7 5 5	3 3 2	7 4 5
T_1	2 0 0	3 2 2		1 2 2
T_2	3 0 2	9 0 2		6 0 0
T_3	2 1 1	2 2 2		0 1 1
T_4	0 0 2	4 3 3		4 3 1

- **Available <- 211, Allocation1 <- 321, Need1 <- 001**
  - *Work <- 211*
  - T_1 : $Need_1$ <= Work → Work+=$Allocation_1$ → Work=532, $Finish_1$ = true
  - T_3 : $Need_3$ <= Work → Work+=$Allocation_3$ → Work=743, $Finish_3$ = true
  - T_4 : $Need_4$ <= Work → Work+=$Allocation_4$ → Work=745, $Finish_4$ = true
  - T_0 : $Need_0$ <= Work → Work+=$Allocation_0$ → Work=755, $Finish_0$ = true
  - T_2 : $Need_2$ <= Work → Work+=$Allocation_2$ → Work=(10)57, $Finish_2$ = true

  - **Le système reste dans un état sûr si on alloue (1A,2B,1C) à T1, donc l'allocation est autorisée**

Exemple : T_0 demande (0,2,0)

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 5	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 1	
P_3	2 1 1	0 1 0	
P_4	0 0 2	4 3 1	

→ Si on alloue 2 ressources B à T0 : état non sûr donc l'allocation est refusée, même si il y a à l'instant courant 3 ressources B disponibles.

Conclusions sur les IB

- **Dans les applications relativement simples, on peut les prévenir par programmation**
  - Vérification possible via des outils spécifiques (ex: model checking)
- **Sinon, il n'y a pas de technique idéale**
  - Evitement: ralentit l'exécution (toute allocation de ressource doit être analysée)
  - Détection-guérison: ralentit moins l'exécution (par rapport à l'évitement) mais lourd à gérer en cas d'interblocage
  - Kill-Restart: pas toujours facile de mettre en œuvre des applications capables d'être interrompues/redémarrées à tout moment