Interblocage

Menu

- Conditions pour l'interblocage
- Méthodes de gestion d'interblocage
- Prévention
- Évitement (avoidance)
- Détection
- Récupération (recovery)

Menu

- **Conditions pour l'interblocage**
- Méthodes de gestion d'interblocage
- Prévention
- Évitement (avoidance)
- Détection
- Récupération (recovery)

Modèle

- Un système est constitué de **ressources**
- Ressource types R_i
  - e.g. mémoire, CPU, E/S
- Chaque type de ressource R_i est disponible en quantité W_i
- L'utilisation est divisée en 3 étapes:
  - *demande (request)*
  - *utilisation (use)*
  - *libération (release)*

Interblocage avec verrous

Thread1 fait: `request(L1); request(L2);`
Thread2 fait: `request(L2); request(L1);`

- L'interblocage a lieu lorsque les deux threads bloquent à mi-chemin
- Dépend des choix d'ordonnancement: non-déterministe
- Il peut être très difficile d'identifier et de tester les interblocages

Conditions nécessaires

- Nécessite 4 conditions simultanément:
  - Exclusion mutuelle: Dépend d'une ressource que l'on ne peut partager
  - "Hold & wait": Un thread tient une ressource et en attend une autre
  - Pas de préemption: un ressource n'est libérée que volontairement
  - Circularité: il y a un cycle de threads ou chaque thread attend un ressource tenue par le thread suivant (a besoin d'un "hold & wait")

Graphe d'allocation des ressources

- Un graphe dirigé bipartite

- Les sommets *V* sont partitionné en deux parties:
  - $P = \{ P_1, P_2, \ldots, P_n \}$, tous les processus (threads) dans le système
  - $R = \{ R_1, R_2, \ldots, R_m \}$, tous les ressources dans le système

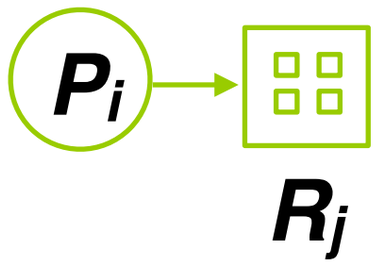
- Chaque **requête** en attente est représentée par un arc $P_i \rightarrow R_j$

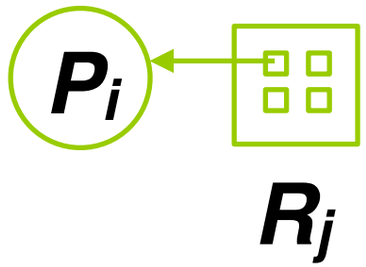
- Chaque ressource **allouée** est représentée par un arc $R_j \rightarrow P_i$

Graphe d'allocation des ressources

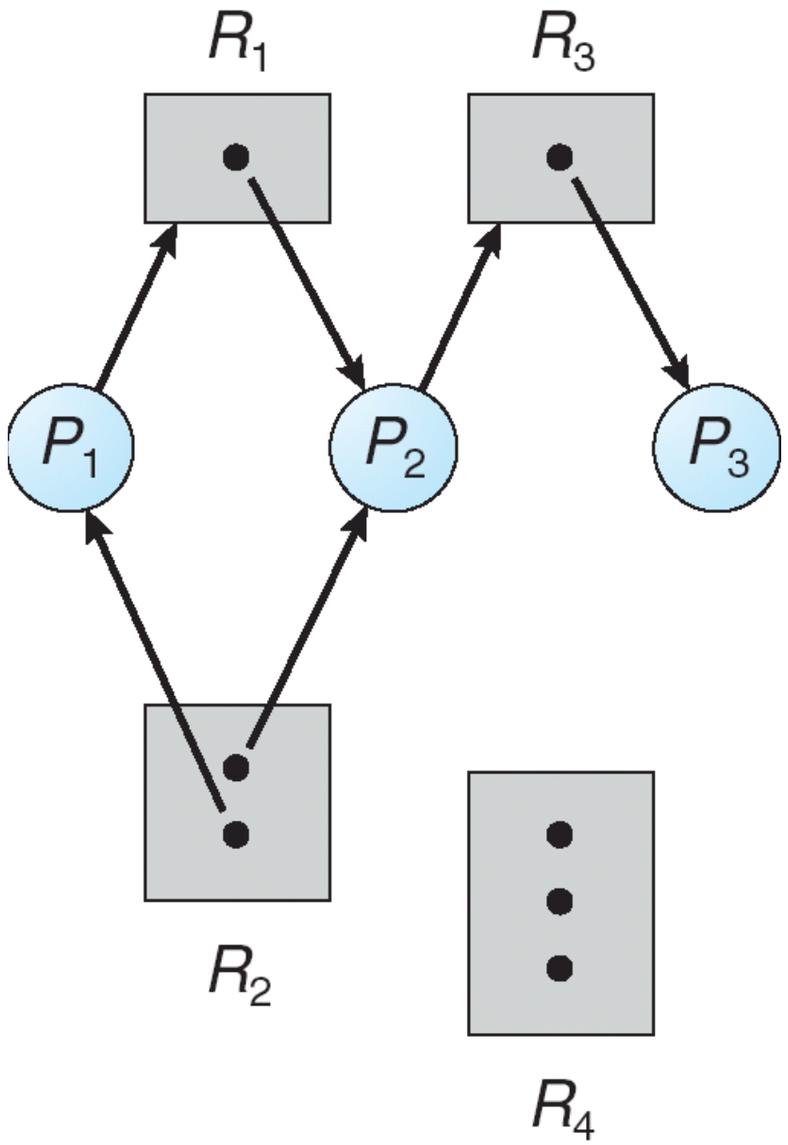
- Process 

- Ressource avec 4 instances 

- P_i requête la ressource R_j 

- P_i tient une instance de R_j 

Exemple de graphe

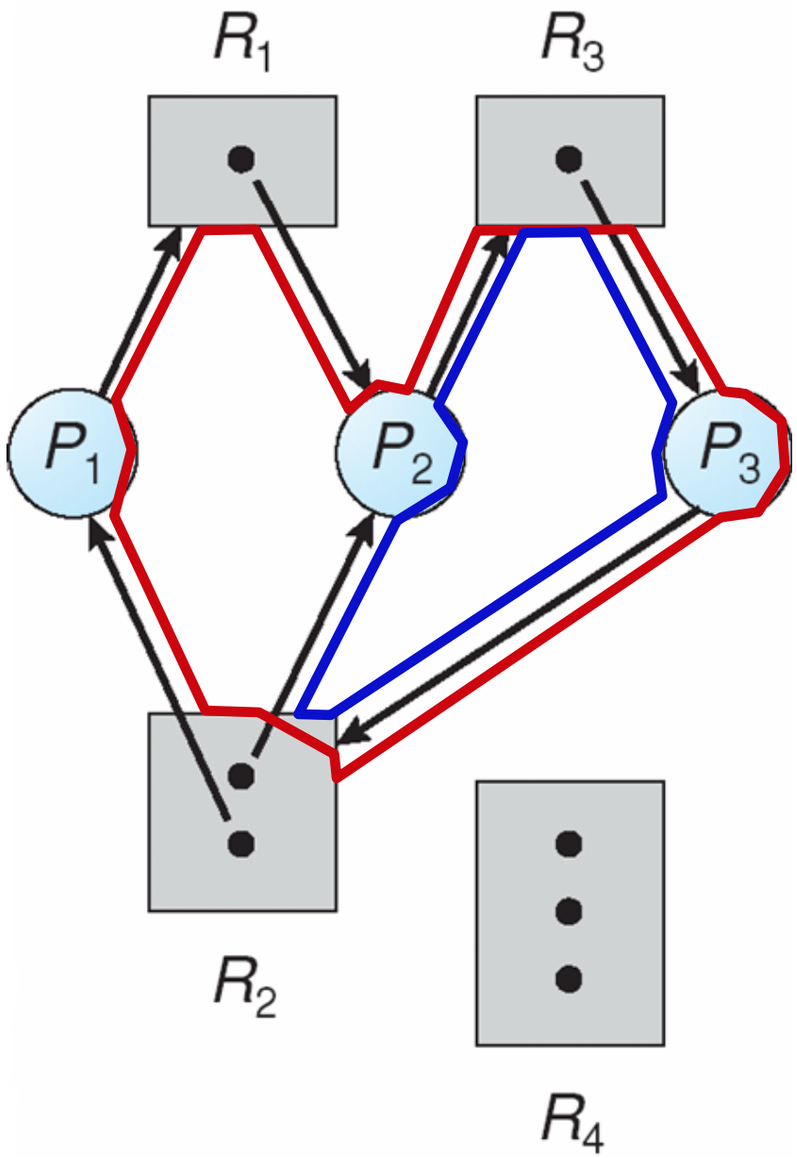


P₁ tiens une ressource ***R₂***
P₁ attend une ressource ***R₁***

...

Pas de cycle
-> pas d'interblocage

Graphe avec interblocage

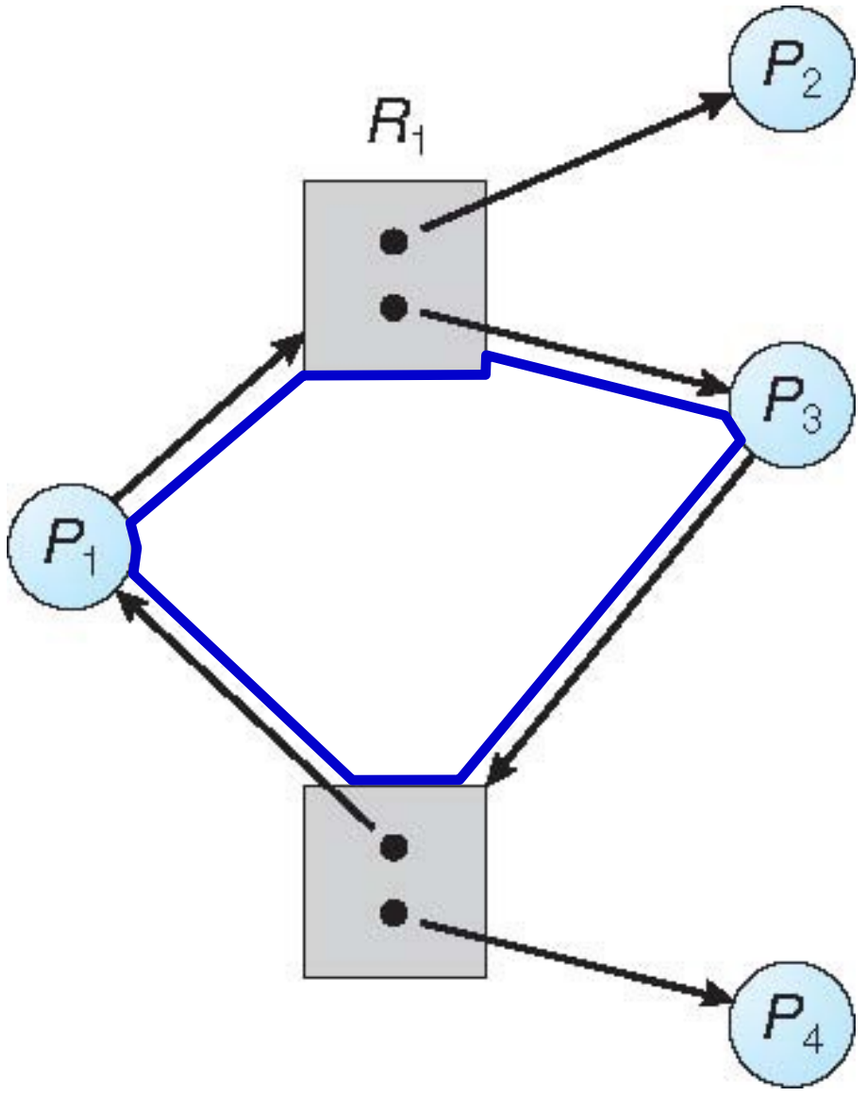


2 cycles minimes

1. bleu
   P2→R3→P3→R2→P2

2. rouge:
   P1→R1→P2→R3→P3→R2
   →P1

Exemple de cycle sans interblocage



un cycle

blue:
P1→R1→P3→R2→P1

P2 and P4 peut libérer leur ressource

pas d'interblocage

Condition d'interblocage

- Pas de cycle => Pas d'interblocage
- Si le graph contient un cycle ⇒ une interblocage peut ou peut ne pas exister
  - si une seule instance par type de ressource, interblocage est garanti
    - ✓ condition nécessaire et suffisante
  - si plusieurs instances par type de ressource, possibilité d'interblocage
    - ✓ condition nécessaire mais pas suffisante

Menu

- Conditions pour l'interblocage
- **Méthodes de gestion d'interblocage**
- Prévention
- Évitement (avoidance)
- Détection
- Récupération (recovery)

Gestion des interbocages

- Option 1: Assurez-vous que le système n'atteindra jamais un état d’ interblocage
  - Prévention (s'assurer qu'une condition nécessaire ne peut pas tenir)
  - Évitement (fournir des informations par processus sur toutes les demandes de ressources)

- Option 2: Autoriser le système à entrer dans un état de blocage, puis récupérer

- Option 3: La méthode de l’autruche
  - Permettre des interblocages
  - Ne même pas essayer de les détecter
  - Méthode la plus populaire

Menu

- Conditions pour l'interblocage
- Méthodes de gestion d'interblocage
- **Prévention**
- Évitement (avoidance)
- Détection
- Récupération (recovery)

Prévention

- Conception du système élimine une des conditions nécessaire:
  - **Exclusion mutuelle** – certaines ressources sont partageables, mais c'est pas le cas tous le temps. *Impossible à éviter en général*.
  - **Hold & wait** – doit garantir que chaque fois qu'un processus demande une ressource, *il ne détient aucune autre ressource*
    1. Exiger que le processus demande et reçoive toutes ses ressources avant de commencer l'exécution
    2. Autoriser le processus à demander des ressources uniquement lorsque le processus n'a aucune
       - il doit libérer ses ressources actuelles avant de les demander à nouveau
  - ✓ Désavantages des deux solutions:
    - moins d'utilisation des ressources
    - famine possible (toujours en attente..)

Prévention

- **Pas de préemption** –
  - Si un processus qui contient des ressources demande une autre ressource qui ne peut pas lui être immédiatement allouée, toutes ses ressources en cours sont libérées
  - Les ressources préemptées sont ajoutées à la liste des ressources pour lesquelles le processus est en attente
  - Le processus ne sera redémarré que lorsqu'il pourra retrouver ses anciennes ressources, ainsi que les nouvelles qu'il demande

- **Circular wait** – imposer un *ordre total* entre tous les types de ressources
  - exiger que chaque processus demande des ressources dans un ordre croissant d'énumération, ou libère des ressources d'ordre supérieur ou égal

$$F : R \mapsto \mathbb{N}$$

$$R_j \text{ après } R_i \implies F(R_j) > F(R_i)$$

Exemple d'interblocage

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /* Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
```

$F(\texttt{first\_mutex}) = 1$

$F(\texttt{second\_mutex}) = 5$

quel thread est en violation?

```
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /* Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Exemple d'interblocage

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

thread 1: `transaction(checking, savings, 25)`

thread 2: `transaction(savings, checking, 40)`

Menu

- Conditions pour l'interblocage
- Méthodes de gestion d'interblocage
- Prévention
- **Évitement (avoidance)**
- Détection
- Récupération (recovery)

Évitement d'interblocage

- Besoin d'information sur le futur
  - E.g., chaque thread annonce son usage maximum de ressources
  - En cours d'exécution, le système vérifie *l'état d'allocation de ressources* pour garantir qu'on ne tombera pas dans un cycle
  - *L'état d'allocation de ressources* est la quantité de ressource allouées et disponibles, ainsi que les usages maximaux annoncés

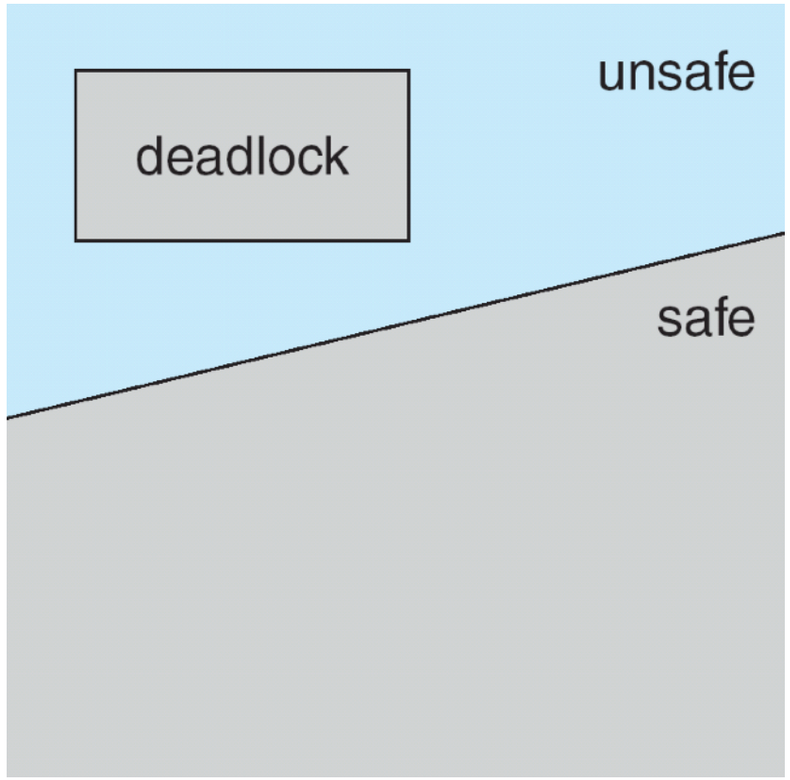
Évitement: État sûr

- Allocation: le système doit décider si une allocation laisse le système dans un état sûr (safe)

- L'état est sûr (safe) s'il existe un séquence d'exécution <P_1, P_2, …,P_n> de tous les processus (threads) telle que pour chaque P_i, les ressources dont il a besoin seront disponible quand tous les P_j, $j < i$ auront terminé leur exécution
  - Quand P_i termine, P_{i+1} peut obtenir ses ressources nécessaires…

Évitement: État sûr

- État sûr ⇒ pas d'interblocage
- État pas sûr (unsafe) ⇒ possibilité d'interblocage
  - interblocage ⇒ état pas sûr
- Évitement ⇒ garantir que le système est toujours dans un état sûr



Examples d'analyse d'état sûr

Ressources: 12 chaises

	Maximum	Actuel
P0	10	5
P1	4	2
P2	9	2

Exécution possible:

P_1 ;	P_0 ;	P_2 :	total
2/4 ;	5/10 ;	2/9 :	9
4/4 ;	5/10 ;	2/9 :	11
end;	**10/10** ;	2/9 :	12
	end ;	9/9 :	9
	;	end :	0

État pas sûr:

P_1 ;	P_0 ;	P_2 :	total
2/4 ;	5/10 ;	3/9 :	10
4/4 ;	5/10 ;	3/9 :	12
end;	9/10 ;	3/9 :	12

interblocage!

Algorithmes d'évitement

- Ressources uniques:
  - Utiliser un algorithme basé sur le graphe d'allocation de ressources
- Pour des ressources multiples
  - Utiliser l'algorithme du banquier

Évitement basé sur le graphe d'allocation

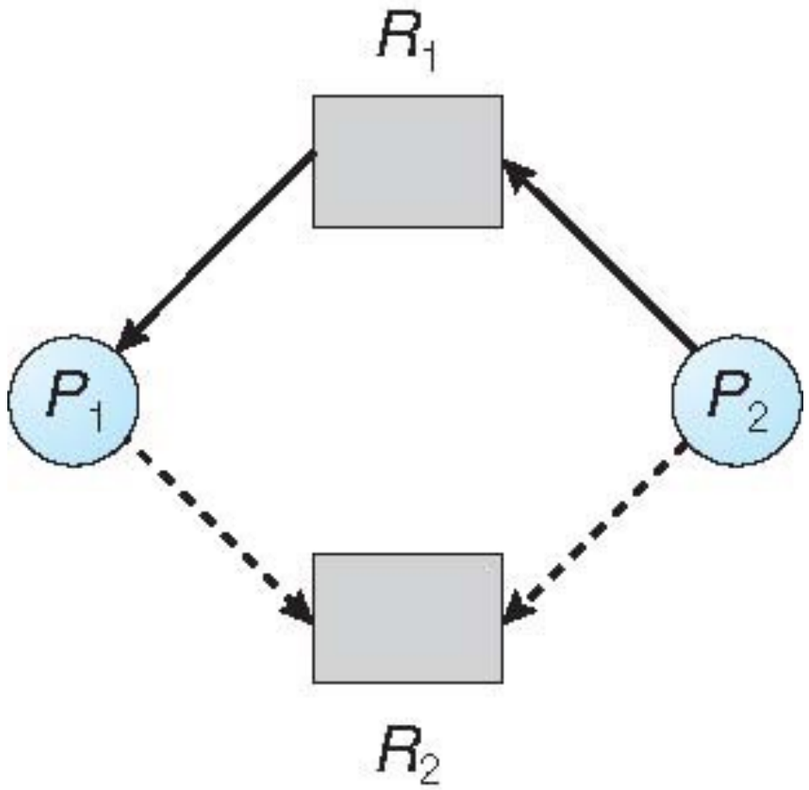
- Deux types d'arcs $P_i \rightarrow R_j$:
  - **Normaux** pour les requêtes en attente
  - **Futurs** pour les requêtes qui pourraient encore arriver (les "claim edge")

- Arc "futur" (claim edge) est converti en arc "requête" (request edge)
- Arc requête est converti en arc "allocation" quand le ressource est alloué au processus

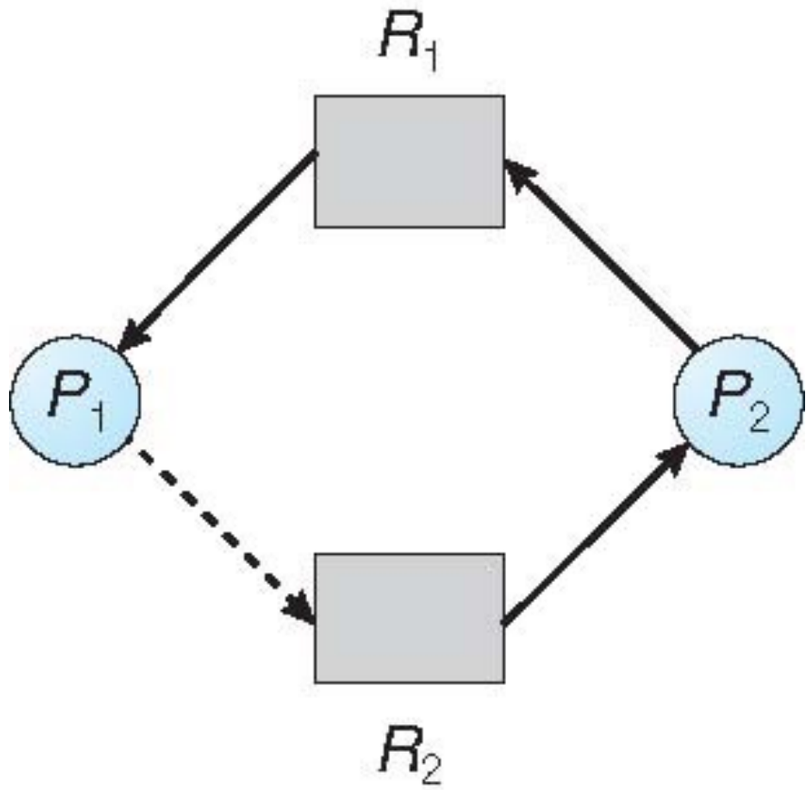
- Il faut connaître tous les arcs futurs dés le départ

Évitement basé sur le graphe d'allocation

- Supposons que le processus P_i demande une ressource R_j
- Une ressource est allouée seulement si cela ne crée pas de cycle



état sûr



Impossible d'attribuer R_2 à P_2
si P_1 demande R_2, état pas sûr

Algorithme du banquier

- Plusieurs instances d'un type de resources
- Moins efficace que l'algorithme du graphe d'allocation des ressources
- Chaque processus doit a priori réclamer le nombre maximum d'instances de chaque type de ressource
- Lorsqu'un processus demande une ressource, il peut devoir attendre
- Quand un processus obtient toutes ses ressources, il doit les renvoyer dans un temps fini

Algorithme du banquier: données

n = nombre de processus, ***m*** = nombre de types de ressources

- *Available[**j**]*: quantité de resource ***j*** disponible. Vecteur de longueur ***m***.
- *Max [**i**,**j**]*: quantité maximum de resource ***j*** utilisé par ***i***. Matrice ***n*** *x* ***m***.
- *Allocated [**i**,**j**]*: quantité de resource ***j*** allouée à ***i***. Matrice ***n*** *x* ***m***.
- *Needed [**i**,**j**]*: quantité de ressource ***j*** dont ***i*** pourrait encore avoir besoin pour terminer son exécution. Matrice ***n*** *x* ***m***.

*Needed[**i**, **j**] = Max[**i**, **j**] – Allocated[**i**, **j**]*

Algorithme du banquier: état sûr

1. Initialize:

   `Work = Available`

   `Finish[ i ] = false  for i = 0, 1, …, n - 1`

2. Trouver un “i” pour que:

   (a) ***`Finish[ i ] == false`***

   (b) ***`Needed[i,j]`*** $\leq$ ***`Work[ j ]`*** (for all ***j***)

   Si aucune “i” procéder a 4.

3.

   `Work[ j ] += Allocated[ i, j]` (for all j)
   `Finish[ i ]` = `true`

   retour à 2

4.

   If ***`Finish[ i ]`*** == `true` for all i

   système dans un état sûr

Algorithme du banquier: allouer des ressources

***Requested**$_i$* = vecteur de requête de processus ***P**$_i$*

Si ***Requested**$_i$ [j]* == ***k*** => ***P**$_i$* veut ***k*** instances de ressource ***j***

***P**$_i$* fait son requête:

1. Si ***Requested**$_i$[j]* > ***Needed[i,j]*** for any ***j*** erreur!
2. Si ***Requested**$_i$[j]* > ***Available[j]*** for any ***j*** doit attendre
3. Essaye d'alloué les ressource à ***P**$_i$* par calculer un nouvel état hypothétique:

   Available[j] -= ***Requested**$_i$[j]* (for all ***j***)

   Allocated[i,j] += ***Requested**$_i$] j]* (for all ***j***)

   Needed[i,j] -= ***Requested**$_i$[j]* (for all ***j***)
4. Testez la sécurité en utilisant la procédure de la diapositive précédente
   - État sûr ⇒ allouer les ressource
   - État pas sûr ⇒ ***P**$_i$* doit attendre

Exemple d'algorithme du banquier

5 processus P0 through P4;

3 types des ressources:

A (10 instances), B (5 instances), and C (7 instances)

au début:

P1 ; P3 ; P4 ; P2 ; P0

	Allocated	*Max*	*Needed*	*Available*
	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

P1 1,2,2 ≤ 3,3,2; libère 2,0,0
Available: 5,3,2
P3 0,1,1 ≤ 5,3,2; libère 2,1,1
Available: 7,4,3
P4 4,3,1 ≤ 7,4,3; libère 0,0,2
Available: 7,4,5
P2 6,0,0 ≤ 7,4,5; libère 3,0,2
Available: 10,4,7
P0 7,4,3 ≤ 10,4,7; libère 0,1,0
Available: 10,5,7

Exemple (cont'd): P_1 requête (1,0,2)

1. Requested ≤ Needed (1,0,2) ≤ (1, 2, 2)
2. Requested ≤ Available (1,0,2) ≤ (3,3,2)

	Allocated	*Needed*	*Available*
	A B C	A B C	A B C
P0	0 1 0	7 4 3	3 3 2 → 2 3 0
P1	2 0 0 → 3 0 2	1 2 2 → 0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

3. État sûr?

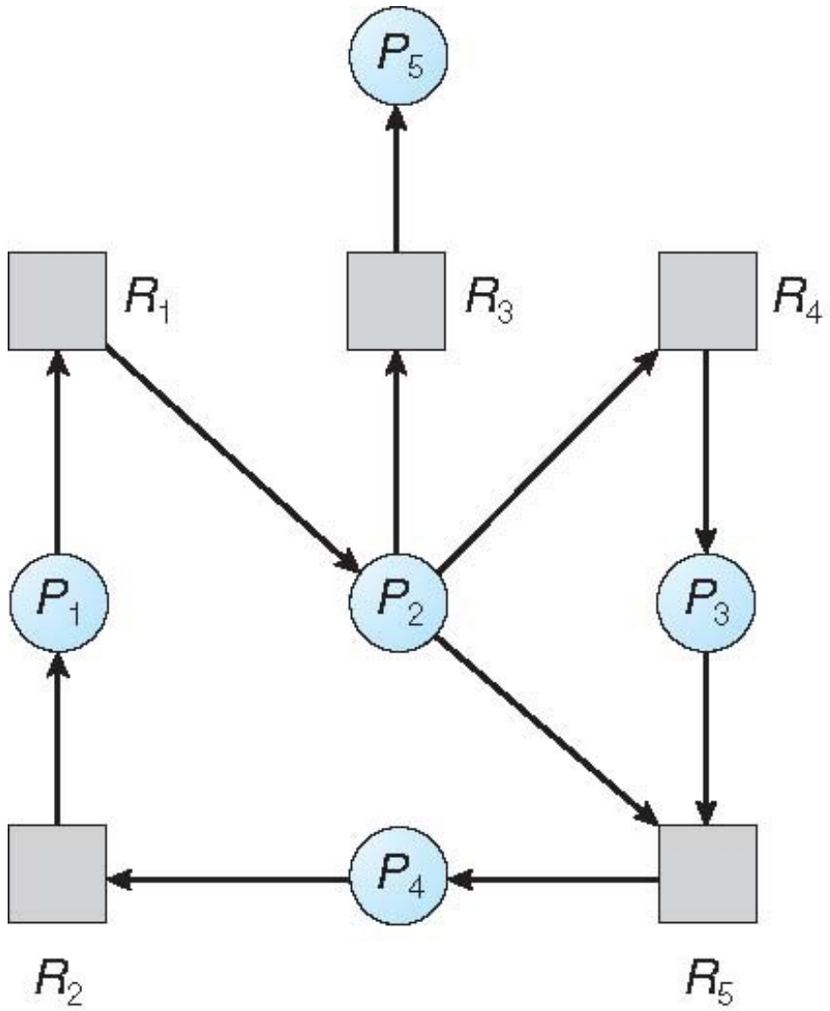
   sequence < P1, P3, P4, P0, P2 > est bon

- Ensuite, requête pour (3,3,0) par P4? non, (2,3,0) < (3,3,0)
- Ensuite, requête pour (0,2,0) by P0? non, état pas sûr

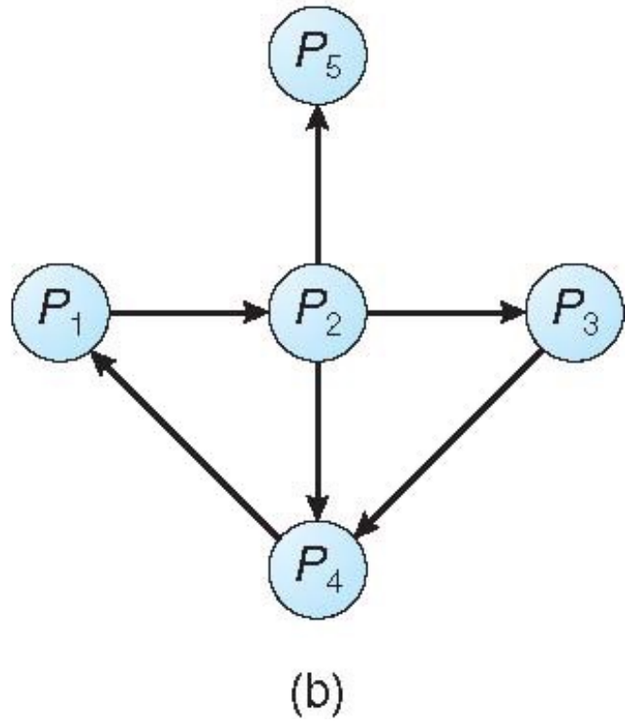
Menu

- Conditions pour l'interblocage
- Méthodes de gestion d'interblocage
- Prévention
- Évitement (avoidance)
- **Détection**
- Récupération (recovery)

Instance unique de chaque type de ressource



Resource-allocation graph



Corresponding wait-for graph

Détection d'interblocage

1. Initialize:

   `Work = Available`

   `Finish[ i ] = false  if Allocation neq 0 else true` `for i = 0, 1, …, n - 1`

2. Trouver un "i" pour que:

   (a) ***`Finish[ i ] == false`***

   (b) ***`Request[i,j]`*** $\leq$ ***`Work[ j ]`*** (for all ***j***)

   Si aucune "i" procéder a 4.

3.

   `Work[ j ] += Allocated[ i, j]` (for all j)
   `Finish[ i ]` = `true`

   retour à 2

4.

   If ***`Finish[ i ]`*** == `true` for all i

   système dans un état sûr

Détection - exemple

Cinq processus P0 through P4;

trois types de ressources A (7 instances), B (2 instances), et C (6 instances)

au début:

	Allocated	*Requesting*	*Available*
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Séquence < P0 , P2 , P3 , P1 , P4 > est bon -> pas d'interblocage

Détection - exemple (cont'd)

P_2 requête un instance additionnelle de type C

au début:

	Allocated	*Requesting*	*Available*
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 **1**	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Interblocage!

Détection d'interblocage: usage

- Quand, et à quelle fréquence, invoquer dépend de:
  - Combien de fois une impasse est susceptible de se produire?
  - Combien de processus devront être annulés?
    - un pour chaque cycle disjoint
- Si l'algorithme de détection est invoqué arbitrairement
  - Longtemps (par exemple, une fois par heure):
    - il peut y avoir plusieurs cycles dans le graphe des ressources et donc nous ne serions pas en mesure de dire lequel des nombreux processus bloqués "a causé" l'impasse
  - Court instant:
    - Après chaque processus d'attente serait très inefficace, mais plus facile d'identifier la «cause», mais aussi le processus peut revenir dans cet état juste après l'avoir renvoyé ...
    - Procédure très coûteuse, où toutes les ressources sont gelées jusqu'à ce qu'il soit terminé
  - Lorsque l'utilisation du processeur passe en dessous de 40% ...

Récupération: préemption

- Supprimer une ressource d'un processus et la donner à un autre processus

- Problèmes:
  - **Choisir une victime** - minimiser les coûts
  - **Rétablissement** - retour à un état sûr, redémarrage du processus pour cet état, mais doit conserver des données sur les états des processus en cours d'exécution
  - **Famine** - le même processus peut toujours être choisi comme victime, inclure le nombre de retour en arrière dans le facteur coût

Récupération: Terminer les processus

- Abandonner tous les processus bloqués
  - grande dépense (les processus devront être réexécutés)
- Demander à l'opérateur de résoudre manuellement le blocage
- Abandonner un processus à la fois jusqu'à l'élimination du cycle de blocage
  - frais généraux considérables (pour vérifier si toujours dans l'impasse)
- Dans quel ordre devrions-nous choisir d'abandonner?
  - Priorité du processus
  - Combien de temps le processus a-t-il calculé et combien de temps il a duré?
  - Ressources utilisées par le processus
  - Le processus de ressources doit être terminé
  - Combien de processus devront être terminés
  - Le processus est-il interactif ou par lots?

Sommaire

- Il y a un interblocage quand les processus attendent indéfiniment des ressources détenues par d'autres
- Les 4 conditions:
  - exclusion mutuelle
  - hold and wait
  - pas de préemption
  - un cycle de dépendance
- Prevention: s’assurer qu'au moins une des quatre conditions nécessaires ne peut pas
- Évitement: graphe des ressources, algorithme du banquier
- Détection: presque la même que l’évitement mais sur l’état actuelle
- Récupération: rollback ou terminaison des processus