1. Introduction

Dans un système donné, plusieurs processus peuvent entrer en compétition pour l'accès à un nombre fini de ressources. Généralement la demande de ressources est bloquante : lorsque les ressources demandées par un processus ne sont pas disponibles à l'instant de la demande, ce processus passe à l'état bloqué et doit attendre jusqu'à leur libération effective.

Il peut arriver qu'un sous-ensemble E1 de processus bloqués en attente de ressources, ne sortent jamais de leur situation d'attente parce que les ressources qu'ils demandent sont détenues par un autre sous-ensemble E2 de processus eux mêmes bloqués en attente d'autres ressources détenues par les processus de E1. Cette situation est appelée **interblocage** ou **deadlock**.

Exemple 1 : Utilisation croisée de 2 ressources critiques.
Lorsque deux processus p1 et p2 demandent chacun deux ressources critiques distinctes r1 et r2 mais dans un ordre différent, il y a un risque de blocage mutuel infini entre les deux processus, chacun détenant une ressource et attendant celle occupée par l'autre sans libérer celle déjà détenue.

P1	P2
Demander(r1);	*Demander(r2);*
<utiliser r1>	*<utiliser r2>*
Demander(r2);	*Demander(r1);*
<utiliser r2>	*<utiliser r1>*
<utiliser r1>	*<utiliser r2>*
Libérer(r1);	*Libérer(r2);*
Libérer(r2);	*Libérer(r1);*

Exemple 2 : Allocation partielle d'une ressource banalisée existant en (N) plusieurs exemplaires. Nombre d'unités de ressource disponibles: Rlibre = 4.

P1	P2
Demander(3);	*Demander(3);*
obtient 2 unités	*obtient 2 unités*
attend 1 unité	*attend 1 unité*

On arrive à: Rlibre = 0 et donc à un interblocage.

2. Conditions d'un interblocage

Quatre conditions sont indispensables pour qu'un interblocage survienne:

1. **Exclusion mutuelle :** chaque ressource peut être allouée à un seul processus à la fois.
2. **Occupation et attente (hold and wait):** les processus ne libèrent pas les ressources précédemment allouées quand ils attendent les réponses à leurs requêtes.
3. **Pas de réquisition :** les ressources précédemment allouées ne peuvent être retirées des processus les détenant.
4. **Attente circulaire :** il existe un ensemble $\{P_0,P_1,..,P_n\}$ de processus en attente tel que P_0 attend une ressource détenue par P_1, P_1 attend une ressource détenue par P_2, …, P_{n-1} attend une ressource détenue par P_n et P_n attend une ressource détenue par P_0.

3. Méthodes de traitement de l'interblocage

Le problème de l'interblocage peut être traité selon deux approches:

1. **Approche pessimiste**. Elle concerne toutes les méthodes préventives dont le principe général est d'instaurer un mécanisme de contrôle avant l'allocation effective des ressources garantissant qu'une situation d'interblocage ne peut se produire. La prévention de l'interblocage peut être réalisée sous deux formes:

   - Statique (application d'une technique préventive avant l'exécution du processus)
   - Dynamique (application du mécanisme préventif pendant l'exécution du processus)

2. **Approche optimiste**. Elle concerne toutes les méthodes de détection et guérison de l'interblocage. Dans ce cas aucune mesure préventive n'est prise: on laisse les processus évoluer librement. Le contrôle se fait à posteriori: si une situation d'interblocage est détectée alors une méthode de guérison est appliquée pour remettre le système dans un état cohérent.

3.1. Détecter les interblocages

3.1.1. Méthode graphique

On peut permettre au système d'entrer dans une situation d'interblocage et le corriger ensuite. Il existe des algorithmes qui *détectent* les interblocages en utilisant un **graphe d'allocation des ressources ou graphe d'état**, qui est une modélisation de l'utilisation courante des ressources du système. Les systèmes laissent se produire les interblocages pour les corriger ensuite.

Un graphe d'allocation des ressources est un graphe orienté constitué d'un ensemble de nœuds N et d'un ensemble d'arcs A. L'ensemble des nœuds est partitionné en deux types :

- $P=\{P_1, P_2, .., P_N\}$ensemble des processus du système
- $R=\{R_1,R_2, .., R_N\}$ ensemble des ressources du système.

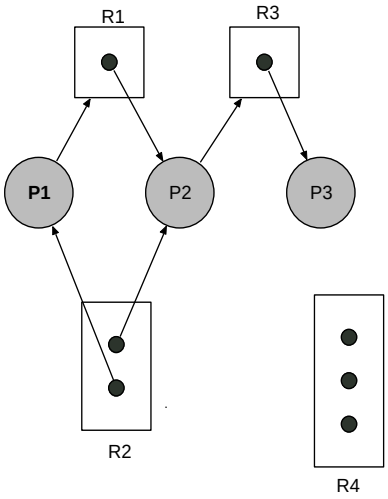
Un arc du processus P_i vers la ressource R_i est noté $P_i \rightarrow R_j$; il signifie que le processus P_i a demandé une instance de la ressource R_j et il attend cette ressource. Cet arc est dit arc de requête. Un arc de la ressource R_i vers le processus P_i est noté $R_j \rightarrow P_i$; il signifie que la ressource R_j a été allouée au processus P_i. Cet arc est dit arc d'affectation.

- **Exemple**

$P=\{P_1,P_2,P_3\}$
$R=\{R_1,R_2,R_3,R_4\}$
$A=\{P_1 \rightarrow R_1,\ P_2 \rightarrow R_3,\ R_1 \rightarrow P_2,\ R_2 \rightarrow P_2,\ R_2 \rightarrow P_1,\ R_3 \rightarrow P_3\}$

Les instances des ressources sont :
R1(1), R2(2), R3(1), R4(3)
Les états des processus :

- P_1 détient une instance de la ressource R_2 et attend une instance de la ressource R_1
- P_2 détient une instance de la ressource R_1 et attend une instance de la ressource R_3
- P_3 détient une instance de la ressource R_3



On peut facilement démontrer que si le graphe ne contient aucun circuit alors aucun processus ne se trouvera en situation d'interblocage. En revanche, la présence d'un circuit dans le graphe n'implique pas toujours l'existence d'interblocages. Dans le cas où chaque ressource possède une seule instance, la présence d'un circuit est une condition nécessaire et suffisante d'interblocages. Dans le cas où chaque ressource possède plusieurs instances, la présence d'un circuit dans le graphe est une condition nécessaire mais non suffisante d'interblocages.

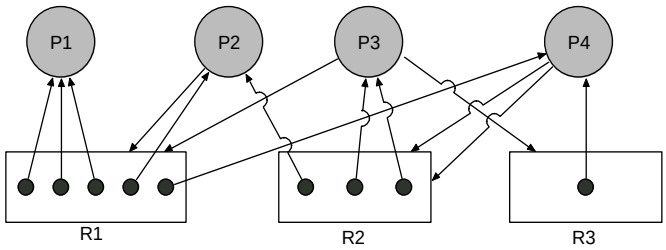
Exemple : si on ajoute au graphe précédent l'arc de requête suivant : $P_3 \rightarrow R_2$ on aura un circuit dans le graphe. On vérifie aisément que les processus P_1, P_2 et P_3 sont en situation d'interblocage.

Une méthode de détection des interblocages dans tous les cas consiste à produire un **graphe réduit d'allocation des ressources** à partir du graphe d'allocation des ressources. Pour cela, il convient de vérifier les flèches associées à chaque processus et à chaque ressource :

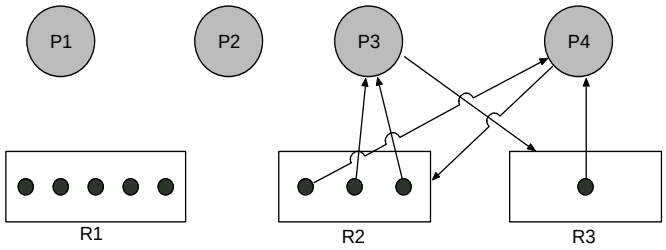
i. Si une ressource ne possède que des départs de flèches (elle n'a aucune requête imminente), il faut effacer toutes ses flèches.
ii. Si un processus n'a que des arrivées de flèches (toutes ses requêtes ont été satisfaites), il faut effacer toutes ses flèches.
iii. Si un processus a des flèches au départ, mais qu'un point de ressource est disponible pour chacune d'elles (un point sans flèche qui s'éloigne du processus) dans la ressource dans laquelle pointe la flèche, il faut supprimer toutes les flèches du processus.

Il faut continuer jusqu'à ce qu'il n'y ait plus de flèche restante ou jusqu'à ce qu'aucune flèche ne puisse être effacée. Le graphe réduit permet donc de détecter plus facilement l'existence de circuits. S**'il ne reste plus de flèche, le système n'est pas en interblocage.**

Exemple : soit le graphe d'allocation de ressources suivant :



On arrive au graphe réduit suivant :



On conclut donc que le système est bien dans un état d'interblocage car il existe un circuit :

$P_3 \rightarrow R_3 \rightarrow P_4 \rightarrow R_2 \rightarrow P_3$

3.1.2. Méthode algorithmique : représentation matricielle

On représente l'état d'un système par une représentation matricielle. Pour un système comportant n processus et m classes de ressources, on peut définir les variables représentatives suivantes:

- Vecteur des ressources disponibles : **Dispo** de dimension m représenté en vecteur ligne, tel que: Dispo[i] = nombre d'unités disponibles de la classe de ressources Ci.

- Matrice des unités de ressources allouées: **Alloc** de dimension n x m, telle que: Alloc [i,j] = nombre d'unités de ressources de la classe j détenues par le processus Pi. La ligne i de Alloc qu'on notera Alloci ou Alloc[i,*] représente l'ensemble des ressources détenues par le processus Pi.

- Matrice des demandes en attente (requêtes non satisfaites): **Requête** de dimension n x m, telle que: Requête [i,j] = nombre d'unités de ressources de la classe Cj demandées et non encore obtenues par le processus Pi. La ligne i de Requête qu'on notera Requêtei ou Requêtet[i,*] représente l'ensemble des ressources demandées et non encore obtenues par le processus Pi (requêtes pendantes de Pi).

- **Equivalence avec la méthode graphique**

Il y a équivalence entre la représentation matricielle et la représentation par graphe car :

i. l'ensemble des arcs d'allocation correspond à la matrice d'allocation **Alloc**.
ii. l'ensemble des arcs de requêtes en attente correspond à la matrice des demandes en attente **Requête**.
iii. l'ensemble des unités de ressources non reliées par des arcs aux processus correspond au vecteur des ressources disponibles **Dispo**.

3.1.2.1.Notion d'inégalité vectorielle

Soient U et V deux vecteurs de même dimension k on écrira par convention que:

a) $U \leq V$ si et seulement si $U[i] \leq V[i]$ pour tout i de 1 à k.
b) $U < V$ si et seulement si $U \leq V$ et $U \neq V$.

Par convention, on peut comparer un vecteur U au vecteur nul de même dimension qu'on notera [[0]].

3.1.2.2.Notion d'état réalisable

L'état d'un système est dit réalisable si les contraintes de cohérence suivantes sont respectées:

i. Contrainte 1 : Le nombre d'unités disponibles est toujours positif ou nul, et un processus ne peut demander plus de ressources qu'il n'en existe dans le système. Soit **Max** le vecteur des ressources maximales disponibles, on aura à l'initialisation (t=0) :

$$(Dispo) = Max = [r1\max r2\max \dots rmmax]$$

Dans ce cas la contrainte 1 s'écrira:

$$\boldsymbol{Dispo} \geq [[\mathbf{0}]] \boldsymbol{et\ Requête}[\boldsymbol{i},*] \leq \boldsymbol{Max}$$

ii. Contrainte 2 : L'ensemble des ressources détenues par un processus ne peut dépasser l'ensemble des ressources maximales disponibles, et un processus ne peut détenir plus de ressources qu'il n'en a demandées. Dans ce cas la contrainte 2 s'écrira:

$$\boldsymbol{Alloc}[\boldsymbol{i},*] \leq \boldsymbol{Max\ et\ Max} \geq \boldsymbol{Alloc}[\boldsymbol{i},*] + \boldsymbol{Requête}[\boldsymbol{i},*] \geq \boldsymbol{Alloc}[\boldsymbol{i},*]$$

iii. Contrainte 3 : A tout instant, la somme des acquisitions des processus ne peut dépasser la totalité des ressources du système:

$$\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{Alloc}[\boldsymbol{i},*] \leq \boldsymbol{Max}$$

3.1.2.3.Notion d'état sain (safe)

Le système est dit dans l'état sain si on peut allouer les ressources dans un certain ordre tel que tous les processus du système peuvent s'exécuter jusqu'à leur fin sans interblocage. D'une manière plus formelle ceci revient à trouver une suite S ordonnée dite saine et complète de processus, *S={Pi, Pj, ... Ps}*, contenant les n processus du système et telle que pour tout processus Pi de cette suite les requêtes de Pi en attente peuvent être totalement satisfaites en prenant l'ensemble des ressources disponibles et des ressources détenues par l'ensemble des processus qui précèdent Pl dans la suite (c'est à dire de rang inférieur à Pi). Donc cette propriété s'écrira: pour tout processus Pi de rang k dans la suite S alors:

$$Dispo + \sum_{rang\ r=1}^{k-1} Alloc[r,*] \geq Requête[1,*]\ avec\ Card(S) = n$$

- **Propriétés des suites saines**

Lorsque l'état du système est sain (non interbloqué) alors on peut construire au moins une suite saine complète de processus.

Lorsque l'état du système est sain (non interbloqué) alors toute suite saine partielle (incomplète) peut être prolongée en une suite saine complète de processus.

Pour détecter un interblocage dans un système il suffit de réaliser un algorithme permettant de construire une suite saine de processus S. Si la suite S est complète donc de cardinal n alors le système est dans un état sain et il n'y a pas d'interblocage. Si S est incomplète et ne peut plus être prolongée en une suite saine complète alors l'état n'est pas sain et il y a interblocage: dans ce cas si Card(S)=k <n alors il y a (n-k) processus interbloqués.

- **Equivalence dans le graphe d'état:**

Un processus est dit réducteur du graphe si toutes ses requêtes peuvent être satisfaites, c'est à dire si tous ses arcs de requêtes en attente peuvent être transformés en arcs d'allocation. Dans le graphe d'état, une suite saine de processus est une suite de processus réducteurs du graphe.

L'état sera sain si la suite processus réducteurs est complète (contient tous les processus du graphe) : le graphe est totalement réduit. Dans le cas contraire (graphe partiellement réduit) on n'obtient que k processus réducteurs (k<n) alors les (n-k) processus restants (non réducteurs du graphe) sont donc interbloqués.

- **Algorithme de Détection de l'interblocage**

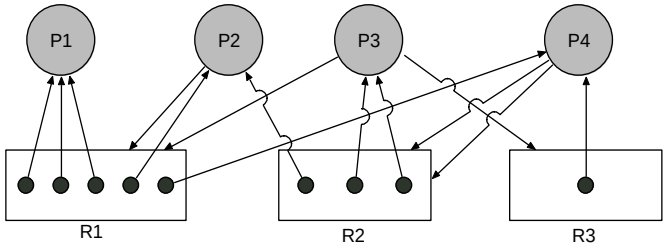
```
Rmax : array [1..m] of constant init [r1max r2max        rm max];
Dispo: array [1..m] of integer;
Work     : array [1..m] of integer;/* vecteur de travail */
Alloc: array [1..n,1..m] of integer;
Requête: array [1..n,1..m] of integer;
Reste : set of processus;
Possible : boolean;
/* Initialisation */
 Dispo:=Rmax;

Procedure SafeTest();
Begin
  Possible := true;
  Reste:={P1,P2, ...., Pn};
  Work:=Dispo;
   While (Possible) and Reste <> {} do
    Begin
      Chercher p dans Reste tel que Requête[p,*] <= Work;
       If trouvé then Begin
                        Reste:= Reste-{p};
                        Work:=Work+Alloc[p,*];
                       End;
        Else Possible:=false;
     End;
  End SafeTest;
```

Après exécution de SafeTest, si Reste={} (ensemble vide) alors l'état est sain (pas d'interblocage); sinon il y a interblocage et Reste contient les numéros des processus interbloqués. La complexité de cet algorithme est polynomiale: O (mxn²).

3.1.2.4.Application

Si on reprend l'exemple précédent :



On arrive à la table suivante :

Processus	**Ressources allouées (Alloc)**			**Ressources demandées (Requête)**			**Ressources disponibles (Dispo)**		
	R_1	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**	**R_1**	**R_2**	**R_3**
P_1	3	0	0	0	0	0	0	0	0
P_2	1	1	0	1	0	0			
P_3	0	2	0	1	0	1			
P_4	1	0	1	0	2	0			

En déroulant l'algorithme, on arrive bien à Reste={P3,P4}.

3.1.3. Corriger les interblocages

Après détection d'un interblocage, on peut envisager de les *corriger* en utilisant trois méthodes :

- la préemption : interrompre un processus.
- La terminaison : terminer un processus.
- Intervention manuelle de l'opérateur système.

Dans tous les cas, la **sélection** du *processus victime* pose problème car on ne connaît pas la *conséquence* d'une telle sélection sur le système. En particulier, un processus victime sans cesse de préemption connaîtra la **famine**.

3.2. Prévenir les interblocages

3.2.1. Méthodes de prévention statique

En s'assurant qu'*au moins une des quatre conditions précédentes ne peut se vérifier*, nous pouvons *prévenir* l'occurrence d'interblocages. Cependant, une telle démarche peut introduire davantage de problèmes qu'elle n'en résout.

3.2.1.1.Méthode de l'allocation globale

Cette méthode vise à éliminer la deuxième condition nécessaire d'interblocage (allocation partielle). On impose à chaque processus de demander globalement toutes les ressources dont il aura besoin avant de commencer son exécution. Il peut libérer une ressource lorsqu'elle ne lui est plus nécessaire. Les inconvénients de cette méthode sont:

- Contraignante car oblige tout processus à demander des ressources à un moment où il n'en a pas besoin.
- Immobilisation des ressources de façon non productive (thésaurisation).
- Dans la pratique, elle diminue et même supprime tout parallélisme d'exécution entre processus.

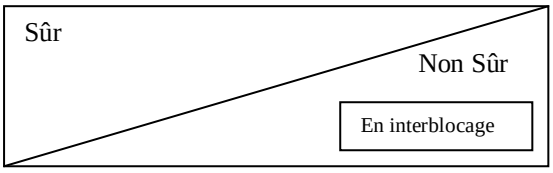
3.2.1.2.Méthode des classes ordonnées

Cette méthode vise à éliminer la quatrième condition nécessaire d'interblocage (attente circulaire). On ordonne les classes de ressources par une relation d'ordre simple, en général en associant à chaque classe une valeur de priorité, et on impose à tout processus de demander ses ressources dans l'ordre croissant des priorités des classes. En conséquence un processus ne peut acquérir les ressources d'une classe donnée que s'il a acquis toutes ses ressources nécessaires dans les classes de priorité inférieure.

Cette méthode évite les inconvénients de la méthode d'allocation globale et améliore la gestion des ressources. En revanche, elle manque de souplesse car la priorité est fixée statiquement entre les classes et impose une rigidité dans la programmation des demandes des processus.

3.2.2. Méthode de prévention dynamique

Il faut utiliser un *protocole* afin de s'assurer que le système ne se retrouvera jamais dans une situation d'interblocage. Une solution consiste à maintenir le système dans un **état fiable (ou sûr)**. On dit qu'un système est dans un état fiable s'il y a une séquence d'exécution sûre. Une séquence d'exécution est un ordre d'exécution des processus dans laquelle tous les processus s'exécutent entièrement. Un état non sûr n'est pas nécessairement un état d'interblocage, mais il y a au moins une séquence de requêtes des processus susceptible de placer le système dans cet état.



La solution pourrait s'appliquer à un système bancaire pour s'assurer que la banque ne prête jamais son liquide disponible de telle sorte qu'elle ne puisse plus satisfaire tous ses clients. Quand un nouveau processus entre dans le système, il doit *annoncer* le nombre *maximal* d'instances de chaque ressource dont il a besoin (appelé **annonce**). Le système doit déterminer ensuite si l'allocation de ces ressources laissera le système dans un état fiable. Si c'est le cas, les ressources sont allouées ; sinon, le processus devra attendre que d'autres processus libèrent suffisamment de ressources.

On aura donc besoin de deux matrices supplémentaires:

- **Annonce**=Matrice des demandes maximales de ressources, de dimension n x m , telle que:

Annonce[i,j] = nombre maximum d'unités de ressources de la classe j que le processus Pi peut demander avant son exécution.

- **Need**=Matrice des besoins maximales de chaque processus, de dimension n x m, telle que:

Need[i,j] = nombre maximum d'unités de ressources de la classe j que le processus Pi peut demander à chaque instant. On pourra ainsi écrire :

Need=Annonce - Allocate

3.2.2.1.Notion d'état fiable:

Le système est dit dans l'état fiable si on peut allouer les ressources dans un certain ordre tel que tous les processus du système peuvent s'exécuter jusqu'à leur fin sans interblocage. D'une manière plus formelle ceci revient à trouver une suite S ordonnée dite fiable et complète de processus, *S={Pi, Pj, ... Ps}*, contenant les n processus du système et telle que pour tout processus Pi de cette suite les besoins maximaux de Pi en attente peuvent être totalement satisfaits en prenant l'ensemble des ressources disponibles et des ressources détenues par l'ensemble des processus qui précèdent Pi dans la suite (c'est à dire de rang inférieur à Pi). Donc cette propriété s'écrira: pour tout processus Pi de rang k dans la suite S alors :

$$Dispo + \sum_{rang\ r=1}^{k-1} Alloc[r,*] \geq Need[1,*]\ avec\ Card(S) = n$$

- **Propriétés des suites fiables:**

Lorsque l'état du système est fiable (non interbloqué) alors on peut construire au moins une suite fiable complète de processus.

Lorsque l'état du système est fiable (non interbloqué) alors toute suite fiable partielle (incomplète) peut être prolongée en une suite fiable complète de processus.

Pour prévenir dynamiquement un interblocage dans un système par test de l'état fiable, il suffit de réaliser un algorithme permettant de construire une suite fiable de processus S. Si la suite S est complète donc de cardinal n alors le système est dans un état fiable et il n'y a pas de risque d'interblocage. Si S est incomplète et ne peut plus être prolongée en une suite fiable complète alors l'état n'est pas fiable et il y a risque d'interblocage.

3.2.2.2.Algorithme du banquier

L'algorithme a été proposé par Dijkstra (67) et Haberman (69).

```
/* Algorithme du banquier */
Rmax : array [1..m] of constant init [r1max r2max        rm max];
Dispo : array [1..m] of integer;
Work     : array [1..m] of integer;/* vecteur de travail */
Alloc : array  [1..n,1..m] of integer;
Requête  : array  [1..n,1..m] of integer;
Annonce : array  [1..n,1..m] of integer;
Need     : array  [1..n,1..m] of integer;
Reste    : set of processus;
Possible : boolean;
```

```
/* A chaque requête d'un processus Pi cet algorithme est lancé: */
Begin
Need:=Annonce - Alloc ;
If Requête[i,*] > Need[i,*] then "erreur demande rejetée car annonce dépassée"
Else If Requête[i,*] > Dispo then /*ressources insuffisantes */ < mettre Pi en attente> ;
      Else Begin /*simuler l'allocation */
             Dispo:=Dispo - Requête[i,*] ;
             Alloc[i,*]:=Alloc[i,*] + Requête[i,*] ;
             Need[i,*]:=Need[i,*] - Requête[i,*] ;
            End;
     /* Tester si état fiable */
        SafeTest();    /*procedure SafeTest mais en remplaçant
                        Requête[p,*] par Need[p,*]              */
       If état fiable then "allocation acceptée " <valider l'allocation>
       Else Begin /* état non fiable */ "allocation refusée"
              <restaurer état antérieur du système : Dispo ; Alloc[i,*]; Need[i,*]; >
              < mettre Pi en attente> ;
             End;
End Banquier;
```

L'algorithme du banquier a besoin de connaître le nombre maximal de chaque classe de ressources pouvant être requis par chaque processus. En utilisant cette information nous pouvons éviter un interblocage. Or, ce type d'information étant rarement disponible dans les systèmes, très peu de systèmes utilisent cet algorithme.

4. Conclusion

Nous avons présenté quelques approches pour traiter le problème de l'interblocage. Nous avons déduit que la plupart des approches sont difficiles à mettre en œuvre dans la réalité. De plus, les chercheurs ont démontré qu'aucunes de ces approches ne résout le problème complètement. Au meilleur des cas, ces méthodes ajoutent à la complexité du système d'exploitation. En revanche, les approches discutées peuvent être combinées, en permettant de sélectionner séparément une approche optimale pour chaque classe de ressources dans le système. Enfin, on peut envisager d'ignorer le problème et prétendre que les interblocages ne se produiront jamais. C'est la solution choisie par la plupart des systèmes, dont Unix. Cette méthode est connue sous le nom de l'**algorithme de l'autruche**.