

2^{ème} partie en parallèle : *LST Info&Miage*

Programmation système et
réseau du point de vue « Multi-
processus »

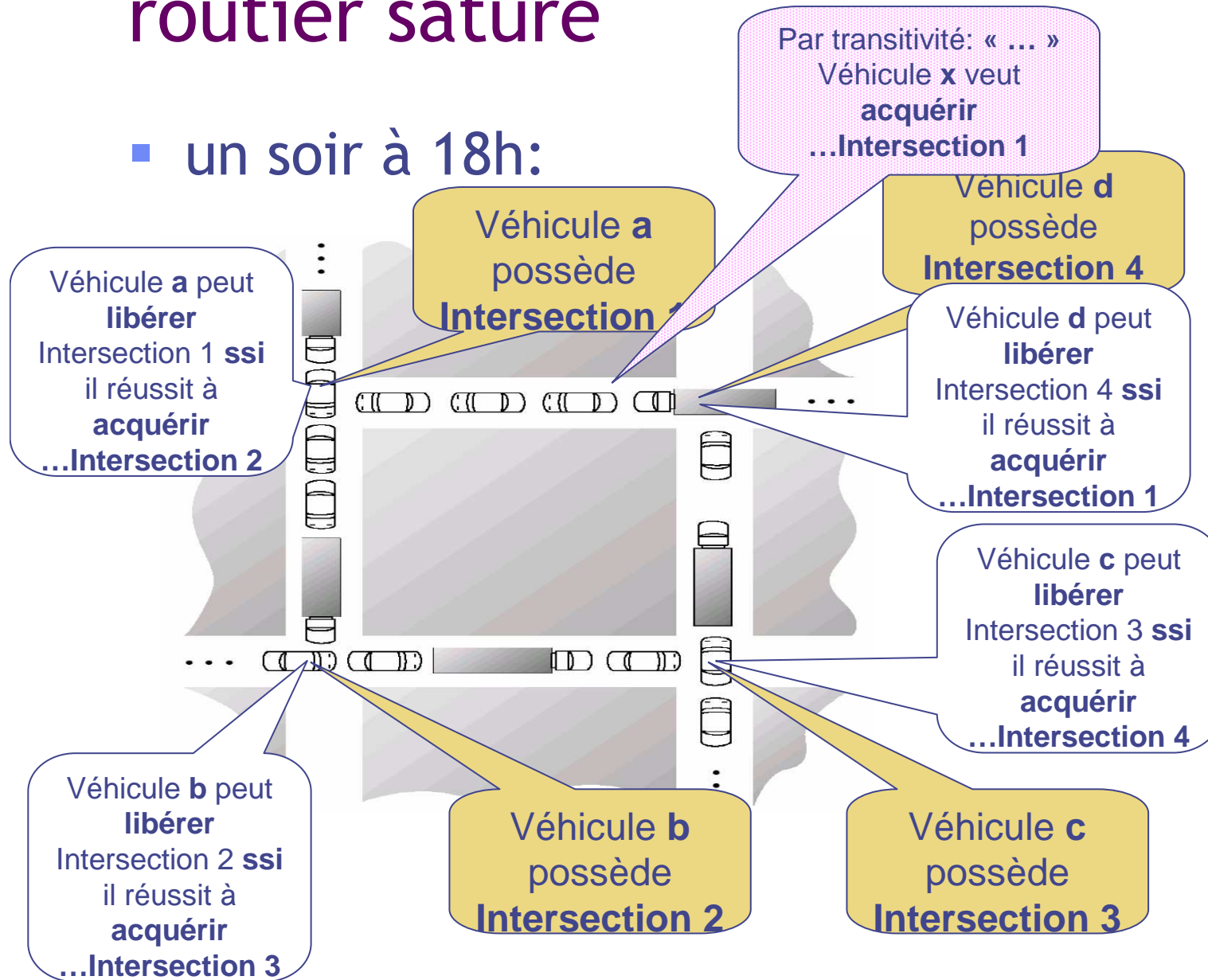
Chapitre 1 : Introduction à la Concurrency
entre processus & Exclusion Mutuelle

Chapitre 2 : Coopération entre processus &
Synchronisation + Communication

Chapitre 3 : Interblocage

Exposé du problème: Carrefour routier saturé

- un soir à 18h:



Inter-blocage:
blocage mutuel
& cyclique

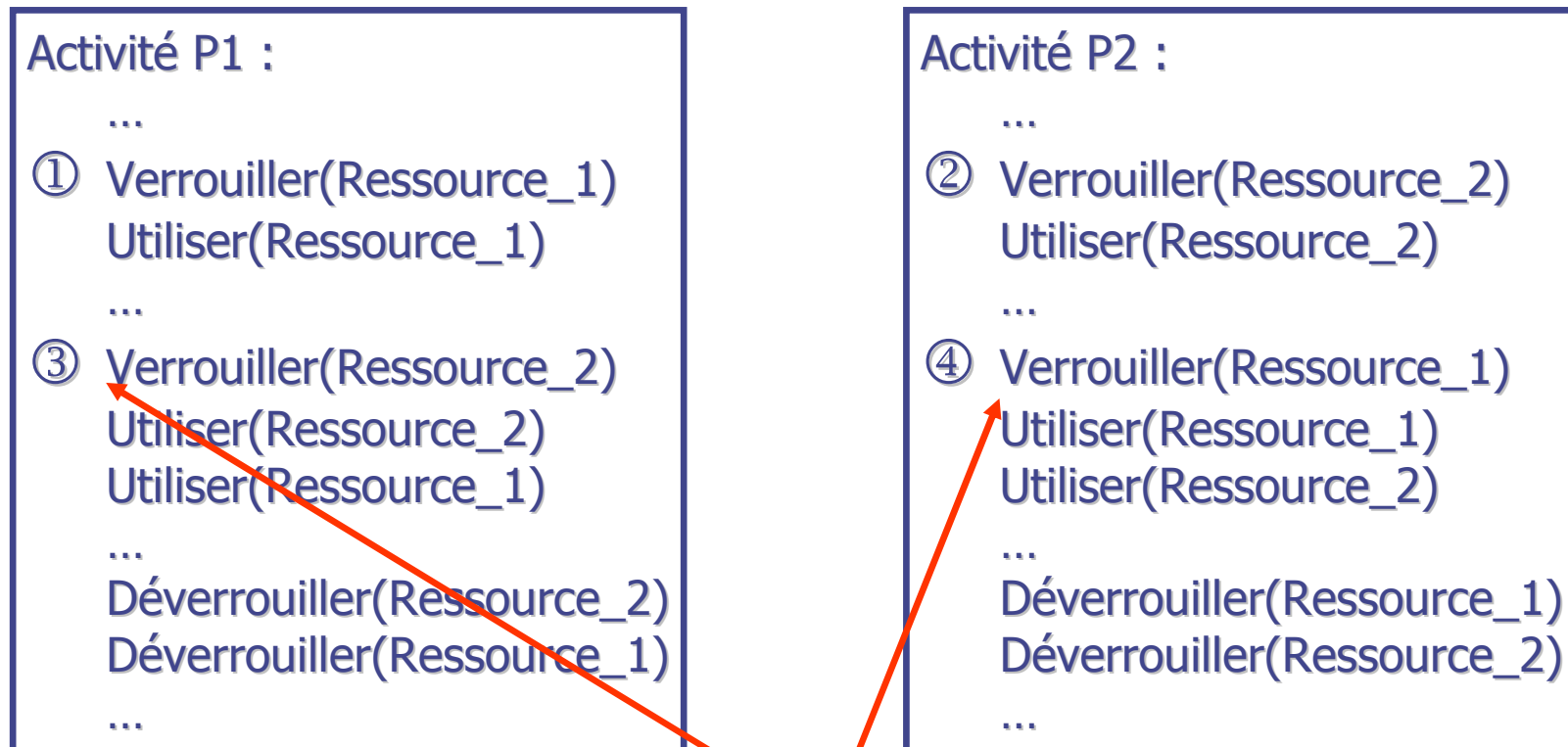
L'attente active,
ou passive, en
espérant que la
situation
s'améliorera est
infinie.

Livelock,
Etreinte
Fatale, (cas att.
Active)
DEADLOCK
(cas att.
Passive)

Définition de l'interblocage

- Un ensemble de processus (thread) est en interblocage si chaque processus de l'ensemble attend qu'un événement advienne, événement que seul un autre processus de l'ensemble peut engendrer.
- L'inter-blocage implique donc au moins 2 processus !
- Souvent lié à l'usage de ressources
 - L'événement est la libération de la ressource convoitée
 - Mais cette ressource est détenue par un autre processus appartenant à l'ensemble
 - Que lui-même ne peut pas rendre tant qu'il en attend d'autres pour pouvoir progresser ...

Exemple d'interblocage potentiel sur acquisition de ressource



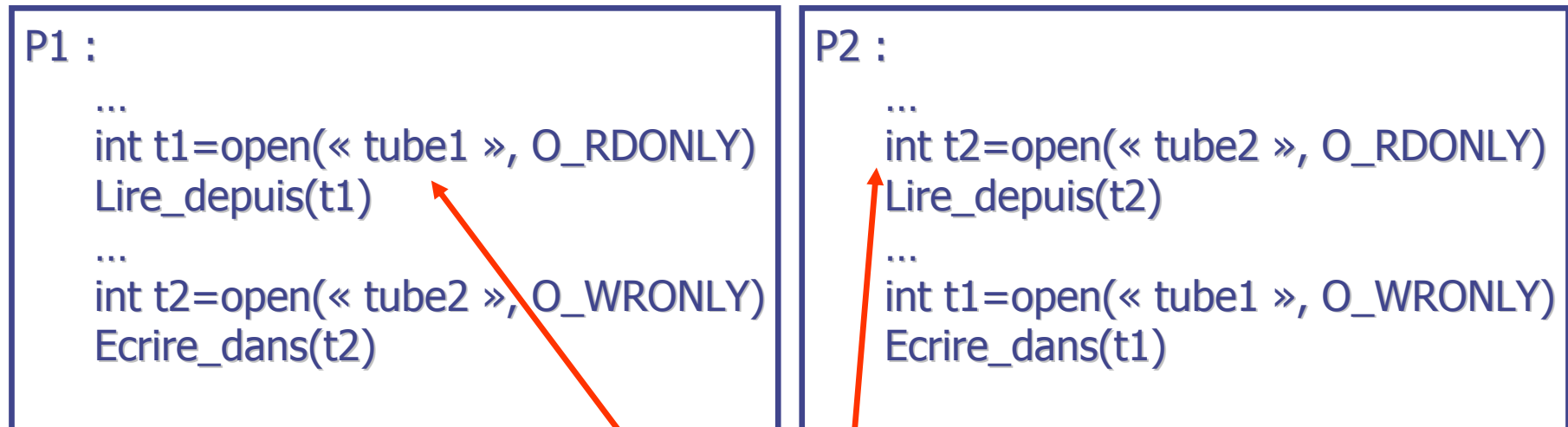
Opérations de Verrouillage pouvant être mortelles (si scénario ① ②) !

(que le verrouillage engendre un endormissement du processus ou un test "actif")

Définition de l'interblocage (suite)

- Il n'est pas forcément lié à une acquisition/libération de ressource
 - Un processus attend qu'une condition soit vérifiée pour continuer, et engendrer une modification d'une autre condition
 - Un autre processus de l'ensemble ne peut pas rendre cette condition vraie, car lui-même, attend que l'autre condition soit vérifiée
- Une telle situation est plutôt dûe à une mauvaise programmation d'un ensemble d'entités coopérantes

Exemple Unix d'interblocage sur ouverture de tube nommé



Opérations d'ouverture menant à un deadlock (car mauvaise programmation) !
(car l'ouverture est bloquante tant que l'événement "un processus a demandé l'ouverture du tube dans le mode symétrique" n'a pas eu lieu)
Dans cet exemple, l'interblocage n'est pas irrémédiable puisque de nouveaux processus pourraient aussi décider d'ouvrir tube1/tube2.

Exemple Unix d'interblocage sur lecture depuis tube volatile

```
int t1[2], t2[2];
...
pipe(t1); pipe(t2);
if fork()==0 { ...
    read(t1[0], ch, 1); /*blocage sur tube1 vide */
    ...
    write(t2[1], ch,1);}
else {...
    read (t2[0], ch, 1); /*blocage sur tube2 vide */
    write(t1[1], ch, 1); }
```

Opérations de lecture menant à un interblocage (mauvaise programmation !) car les 2 boucles de lecture sur tube bloquent (attendant 1 car ou FdF)

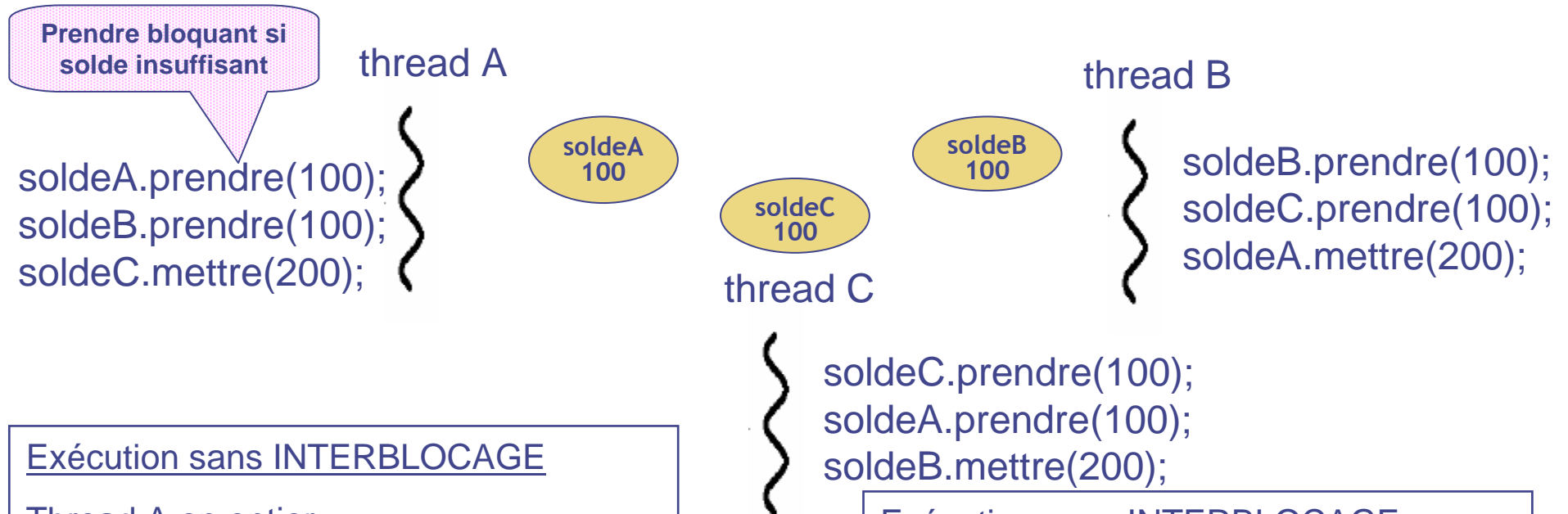
Dans cet exemple, l'interblocage est vraiment irréremédiable puisque seuls ces 2 processus connaissent l'existence du tube.

(Contre-)Exemple Unix d'auto-blocage sur lecture depuis tube volatile

```
int t1[2];
...
pipe(t1);
/* les 2 process. ont conservé desc. en écriture
   t1[1] ouvert */
if fork()==0 { ...
    while (int i=read(t1[0], ch, 1)!=0); /*blocage sur
    tube1 vide tant que pas FdF */ { ... }
    close(t1[0]); ...}
else {... close(t1[0]); /*facultatif, père ne lit pas */
    write(t1[1], ch, 1);
    close(t1[1]); /*important, père n'écrit plus*/
}
```

Opérations de lecture menant
à un blocage irrémédiable
(mauvaise programmation !)
car la boucle de lecture sur
tube bloque (attendant 1 car
ou FdF), mais le process fils a
oublié de fermer son
extrémité en écriture

Exemple d'interblocage potentiel sur condition, attente d'événement



Exécution sans INTERBLOCAGE

Thread A en entier

⇒ soldeA=0, soldeB=0, soldeC=200

⇒ Threads B et C **bloquées**, Thread A non bloquée

⇒ Il y a un espoir que A exécute plus tard mettre(xxx) sur solde B et soldeC

Exécution avec INTERBLOCAGE

Thread A, B, C effectuent leur 1ere op. "prendre(100)"

⇒ soldeA=0, soldeB=0, soldeC=0

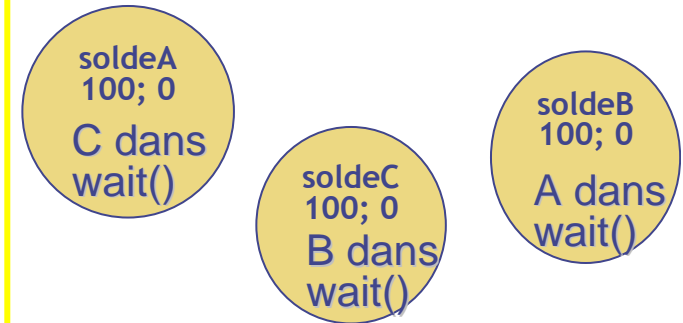
⇒ 2eme op « prendre(100) » de A, B, C : elles sont INTER-bloquées

⇒ Il n'y a plus d'espoir qu'une des 3 threads fasse mettre(xxx) sur les soldes

Suite exemple

```
Moniteur CompteBanque {  
  int solde=0;  
  var condition soldeOK;  
  procedure mettre(int montant){  
    solde = solde + montant;  
    signalAll(soldeOK);  
  }  
  procedure prendre(int montant){  
    while (solde < montant)  
      wait(soldeOK);  
    solde = solde - montant;  
  }  
}
```

soldeA, soldeB, soldeC:
CompteBanque
3 threads cf. transp. avant



Les threads ne sont pas interbloquées alors qu'elles possèdent une ressource et en attendent une autre

Mais, parce qu'elles attendent un événement (augmentation solde compte)

- que seul un thread de l'ensemble peut provoquer
- mais qui n'arrivera jamais car toutes les threads de l'ensemble sont bloquées

Définition de Livelock

- Lorsque le blocage se traduit par une attente active, on utilise le mot **livelock**
 - S'applique aussi parfois avec un seul processus « s'auto-bloquant »
 - Un processus exécute une action, qui engendre un événement
 - Sur occurrence de cet événement, le processus re-exécute la même action
 - Ou si 2+ processus,
 - Un processus exécute une action, qui est perçue par l'autre
 - En réponse de quoi, l'autre exécute une action, perçue par le premier, qui a pour effet de lui faire re-exécuter l'action initiale
 - Exemple: pour se croiser dans un couloir étroit, on change de file
 - Mais, de manière simultanée, les 2 personnes choisissent d'aller sur la même file... cela peut donc continuer sans fin
 - Cf. protocole d'évitement de collision pour accès bus Ethernet
 - C'est donc une sorte de boucle infinie
 - Le(s) processus ne donne(nt) pas l'apparence d'être bloqué(s)
 - Pourtant, son(leur) exécution(s) ne progresse(ent) pas
 - => utiliser un peu d'aléatoire pour casser la similarité

Circonstances de survenue d'interblocage (étudié dans le cadre « allocation de ressource »)

- On constate que sont vérifiées **en même temps**

1 - Exclusion mutuelle

Chaque ressource est soit attribuée à un seul, soit disponible

2- Détention et attente

Les entités qui détiennent des ressources peuvent en demander de nouvelles (sans relâcher celles qu'elles détiennent)

3- Pas de réquisition

Les ressources obtenues par un processus ne peuvent lui être retirées de force

4- Attente circulaire

Il y a un cycle orienté d'au moins 2 entités, chacune en attente d'une ressource détenue par une autre entité (du cycle).

Traitements possibles de l'interblocage lié à l'allocation de ressources

- Crucial pour des entités indépendantes, mais exécutées en concurrence sur le système
- Le prévenir:
 - Faire en sorte (en imposant éventuellement une discipline de programmation) qu'au minimum une des 4 circonstances d'apparition ne soit jamais présente
- L'éviter:
 - Le système insère des contrôles (transparents) lors de chaque opération qui peut interbloquer l'ensemble d'entités:
 - autorise le blocage seulement si pas de risque d'arriver dans une situation d'interblocage : Algorithme du « Banquier »
- Le guérir:
 - Le système laisse les interblocages arriver, mais peut si nécessaire les réparer
 - Par exemple, en terminant de force une des entités qui appartient à l'ensemble interbloqué: les ressources qu'elle détenait sont libérées, cela peut donc débloquent les autres

Nier l'une des 4 conditions : dans la pratique ?

- Interblocage possible d'un ensemble d'entités concurrentes, lorsque les 4 conditions sont vérifiées **en même temps**

1 - Exclusion mutuelle

Chaque ressource est soit attribuée à un seul, soit disponible

2- Détention et attente

Les entités qui détiennent des ressources peuvent en demander de nouvelles (sans relâcher celles qu'elles détiennent)

=> Demander toutes les ressources d'un bloc :

- Plus d'imbrication des SCritiques ou codes synchronisés, mais une seule SC « utilisant » toutes les ressources
- Ou si l'une d'elles est déjà attribuée, relâcher toutes celles déjà acquises

3- Pas de réquisition

- Les ressources obtenues par un processus sont retirées de force: il se termine ou revient en arrière, les ressources reprennent leur état antérieur

4- Attente circulaire

Il y a un cycle orienté d'au moins 2 entités, chacune en attente d'une ressource détenue par une autre entité (du cycle).

=> Toutes les entités doivent demander toutes les ressources toujours dans le même ordre afin de ne jamais « fermer » le cycle

Suppose que les ressources sont ordonnées, et l'ordre respecté

Nier 1 : « Exclusion mutuelle »

- Rarement faisable dans le cas général
- Exemple : deux processus souhaitent imprimer le contenu du même fichier :
 - L'un verrouille l'imprimante puis le fichier
 - L'autre verrouille le fichier puis l'imprimante
- Solution : au lieu d'imprimer directement, mettre la requête d'impression dans une file d'impression
 - Donne l'illusion que l'attribution a été possible alors que ce n'est pas forcément le cas ...

Nier 2 : « Détention et attente »

- Demander toutes les ressources à accès exclusif d'un seul coup
 - Plus d'imbrication des SCs, mais une seule SC « utilisant » toutes les ressources
 - Principe de mise en œuvre : rendre atomique (indivisible) la séquence d'acquisition des ressources
 - On obtient toutes les ressources ou aucune, la phase d'acquisition étant elle-même une section critique

Mises en œuvre possibles de « Nier 2 »

- Sémaphores multiples, *semop* sous Unix

Opération multP(s1, s2, ..., sn : sémaphore) INDIVISIBLE
Si ((s1.Val >=1) ET (s2.Val >=1) ET ... (sn.Val >=1)
Alors
 Pour i := 1 à n **Faire**
 si.Val := si.Val - 1 ;
 FinPour
Sinon
 *Ranger le « contexte » du processus dans sj.File où
 j est le plus petit numéro de sémaphore dont la
 valeur < 1*
 Compteur_Ordinal := adr(multP)
FinSi

Indivisible de par
l'implémentation
de cette fonction
système
=> Assimilable à
une « section de
code critique »

Opération multV(s1, s2, ..., sn : sémaphore) INDIVISIBLE
Pour i := 1 à n **Faire**
 si.Val := si.Val + 1 ;
 *Retirer de si.File le « contexte » de tous les processus
 Mettre ces processus dans l'état prêt*
FinPour

Mises en œuvre possibles de

« Nier 2 »

- Si pas de concept de « sémaphore multiple » disponible ...
- Utiliser un sémaphore d'exclusion mutuelle afin de ne pas entrelacer les séquences d'acquisition de ressources
 - Si une ressource est occupée:
 - Bloque le processus qui l'attend, dans la section critique
 - Mais empêche les autres processus de prendre les ressources encore libres et de risquer de se bloquer attendant des ressources occupées
 - Limite le degré de parallélisme: empêche même l'exécution des processus dont les ressources requises sont libres

Pi pour tout i

Var globale: Mutex_acquisition, Init 1

Répéter

...

P(Mutex_acquisition)

Acquérir toutes les ressources utiles

V(Mutex_acquisition)

Section Critique utilisant les ressources

Libération des ressources

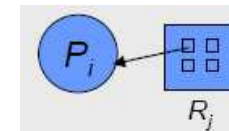
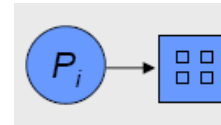
Jusqu'à Faux

Nier 3 : « Pas de réquisition »

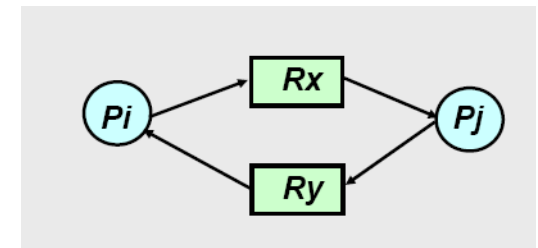
- Suppose de récupérer de force certaines ressources utilisées par un processus
 - Soit en terminant le processus
 - Soit en étant capable de le faire revenir en arrière
 - Nécessite de prévoir des « points de reprise » : photo complète de l'état du processus depuis laquelle il peut reprendre son exécution
 - Cette photo comprend aussi celle des ressources
- Autoriser la réquisition n'est pas toujours souhaitable ou bien est délicate ...
 - Ex: Difficile d'interrompre un processus qui en train d'utiliser une imprimante !
 - Car annuler les effets de l'utilisation de la ressource supposerait d'effacer des lignes déjà imprimées
 - Ex: transaction (concurrente) sur une BD (avec verrouillage d'enregistrements), que le gestionnaire transactionnel doit annuler pour éviter interblocage
 - Faire un retour arrière (« rollback ») nécessite l'usage d'un journal pour enregistrer les modifications effectuées en cas d'annulation de la transaction
 - afin de les défaire, annuler,
 - ou de ne pas les confirmer (« commit »)

Graphe des attentes

- Cas simple: ressource en 1 exemplaire
- Ensemble des Processus $P = \{P_1, P_2, \dots, P_n\}$,
- Ensemble des Ressources $R = \{R_1, R_2, \dots, R_m\}$,
- P_i demande $R_j : P_i \rightarrow R_j$
- P_i détient $R_j : R_j \rightarrow P_i$



- Cycle dans le graphe des attentes :
 - Dénote interblocage d'1 sous-ens de P (sous l'hyp. « Cas simple »)



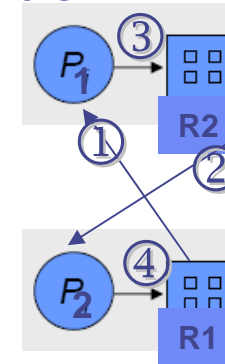
- Ne jamais permettre l'apparition d'un cycle: permet d'éviter tout interblocage

Nier 4 : ordonner les demandes

Activité P1 :	Activité P2 :
...	...
① Verrouiller(Ressource_1) Utiliser(Ressource_1)	② Verrouiller(Ressource_1) pas encore Utiliser(Ressource_1)
...	...
③ Verrouiller(Ressource_2) Utiliser(Ressource_2) Utiliser(Ressource_1)	④ Verrouiller(Ressource_2) Utiliser(Ressource_2) Utiliser(Ressource_1)
...	...
Déverrouiller(Ressource_1) Déverrouiller(Ressource_2)	Déverrouiller(Ressource_1) Déverrouiller(Ressource_2)
...	...

- Contrôle de l'interblocage qui est « statique »
- Suppose un classement connu de toutes les entités concurrentes et une discipline de programmation qui respecte ce classement... Ici : $R1 < R2$
- Peut être contraignant pour le programmeur
- Non efficace, car verrouillage inutile de ressources non (encore) utilisées

Version initiale: interblocage possible



Version sans interblocage

