Université de Strasbourg
UFR Mathématique et Informatique

M1 Informatique
2021/2022

Algorithmes Distribués

TD n° 3 : Interblocages

CORRECTION

Le blocage peut s'avérer être une situation normale : tout processus peut être bloqué à un moment donné : attente d'entrée / sortie, de mémoire ou simplement du processeur. A l'inverse, un interblocage correspond à une situation dans laquelle un ensemble de processus sont bloqués de façon définitive alors que ni le matériel, ni le système ne sont en panne.

Par exemple, la situation suivante est un interblocage :
- P_1 dispose de la ressource R_1 et attend la ressource R_2 pour pouvoir continuer ;
- P_2 dispose de la ressource R_2 et attend la ressource R_1 pour pouvoir continuer ;

L'interblocage est a priori définitif si on ne dispose pas d'autres ressources R_1 et R_2. On parle d'attente circulaire. Trois conditions sont nécessaires pour qu'un interblocage soit possible :

1. **exclusion mutuelle sur les ressources**, i.e., une ressource est soit disponible, soit détenue par un seul et unique processus ;
2. **attente circulaire**, i.e., il existe un cycle d'au moins deux processus attendant chacun une ressource détenue par un autre processus ;
3. **non requisition**, i.e., lorsqu'un processus possède une ressource, on ne peut le forcer à la libérer.

Trois approches sont possible pour résoudre le problème de l'interblocage :
- la **prévention** : on supprime a priori l'une des conditions précédentes (en pratique on supprime généralement les possibilités d'attente circulaire) ;
- l'**évitement** : un processus est capable de détecter si son action entraîne un interblocage. Néanmoins, cette solution nécessite l'utilisation d'un site centralisateur et une connaissance (ou une estimation du maximum) des besoins futurs en ressources de l'ensemble des processus.
- **détection et reprise** : on laisse l'application se dérouler. Lorsqu'on soupçonne un interblocage on réalise les actions suivantes :
  1. détection de la terminaison (pour différencier la fin d'un problème) ;
  2. en cas de non-terminaison, on cherche s'il y a un processus en interblocage *détection* ;
  3. si un processus est en interblocage, on enclenche une phase de *reprise* par une suppression temporaire d'une des conditions. En général on tue un ou plusieurs processus pour qu'ils libèrent leurs ressources (*réquisition*).

Exercice 1 : Caractérisation d'une situation d'interblocage et introduction à sa prévention

On dispose d'un ensemble de processus, et de ressources critiques (1 unité par ressource) nécessaires à l'exécution de ces processus : bloc disque, imprimante, modem, etc. Chaque processus, lors de son exécution, demande l'allocation de ressources. Les ressources sont libérées en fin d'exécution du processus (voir table 1).

Q 1. Montrez qu'il existe un risque d'interblocage avec un tel scénario. Rappelez les 3 conditions nécessaires à l'apparition d'un interblocage.

Il y a bien ici exclusion mutuelle sur les ressources et non réquisition. Concernant la dernière condition nécessaire pour qu'un interblocage soit possible (i.e., attente circulaire), dessinons un graphe biparti avec :
- 3 sommets pour les ressources
- 3 autres pour les processus
- un processus demande une ressource $\rightarrow$ arc orienté du processus vers la ressource
- une ressource est allouée à un processus $\rightarrow$ arc orienté de la ressource vers le processus

P_1	P_2	P_3
demander(bloc disque)	demander(modem)	demander(imprimante)
demander(modem)	demander(imprimante)	demander(bloc disque)
exécution	**exécution**	**exécution**
libérer(modem)	libérer(imprimante)	libérer(bloc disque)
libérer(bloc disque)	libérer(modem)	libérer(imprimante)

TABLE 1 – Exécution de 3 processus utilisant 3 ressources critiques.

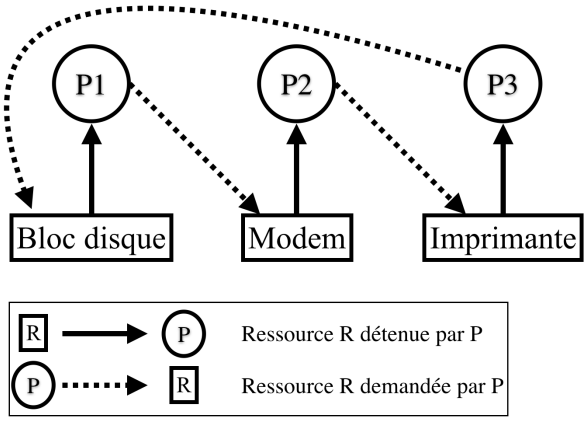


FIGURE 1 – Détection d'un cycle dans le graphe biparti → Risque d'interblocage.

Une fois que les premières demandes respectives de chaque processus ont été satisfaites (i.e., ressources demandées libres et donc allouées aux processus), On obtient la figure 1.

Ici, on observe un cycle alors qu'il n'existe qu'une seule instance par ressource * ; il y a donc attente circulaire et les trois conditions sont réunies pour un possible interblocage.

Q 2. Si l'interblocage se produisait, comment pourrait-on le résoudre ?

- **Préemption de ressource**, i.e., retirer temporairement une ressource à un processus pour l'attribuer à un autre (dépend du type de ressource, demande souvent une intervention manuelle)
- **Destruction de processus** (choix du ou des processus ?)
- **Reprise arrière** (*rollback*) après préemption (quel processus victime choisir ? reprise totale ? risque de famine ?) ; périodiquement enregistrer l'état de chaque processus et restaurer l'état du processus avant l'acquisition de la ressource par le processus et sa préemption (le travail depuis le dernier point de reprise est perdu)

Q 3. Proposez une solution de prévention statique de l'interblocage pour ce cas.

Il faut supprimer une des conditions nécessaires à la survenue d'un interblocage.

1. **Condition d'exclusion mutuelle :** En général impossible à supprimer : exemples partage d'une imprimante, d'une page mémoire. Pour certains périphériques, mécanisme de spoule permettant de différer l'utilisation de la ressource (idée : n'attribuer une ressource que si absolument nécessaire)
2. **Condition de non réquisition :** En général difficile, voire impossible à supprimer.
3. **Condition d'attente circulaire :** Deux solutions générales existent : On peut aussi réaliser une allocation dynamique de ressources et traiter de leur accumulation par les processus.
   - allocation globale : les processus demandent toutes les ressources nécessaires à leur exécution (inconvénients := connaître ses besoins à l'avance, famine possible)
   - allocation par classes ordonnées : (classe := ensemble des instances d'une ressource) les classes sont ordonnées a priori :
     - demande et allocation dans l'ordre
     - demande globale par classe

     → Meilleure utilisation des ressources si l'ordre est bien choisi.

∗. En cas d'instances multiples par ressource, l'existence d'un cycle n'implique pas forcément un interblocage. En revanche, la non-existence de cycle implique qu'il n'y a pas interblocage

Ici, on pourrait déterminer l'ordre $bloc\ disque > modem > imprimante$ afin que P_3 demande le bloc disque avant l'imprimante (voir Fig. 2).

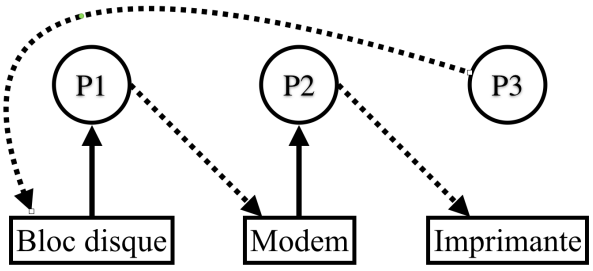


FIGURE 2 – Solution de prévention statique.

Il s'agit bien là d'une solution statique qui permet de supprimer la condition d'attente circulaire.
Inconvénient : souvent difficile de trouver un ordre satisfaisant (nombre de ressources élevé et diversité de leur utilisation).

Exercice 2 : Prévention d'interblocage (modèle basé sur les ressources) – *Wait or Die vs. Wound or Wait*

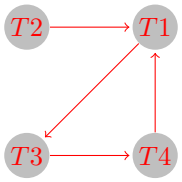
Application de l'algorithme de Rosenkrantz, Stearns et Lewis au scénario détaillé ci-dessous.
Soit quatre transactions avec les séquences d'exécution suivantes :

Transaction	Débute au temps	Séquence à exécuter
T1	1	L(A), R(A), W(A), L(B), W(B), U(A), U(B)
T2	2	L(A), L(C), R(C), W(A), U(C), U(A)
T3	3	L(B), R(B), L(C), W(C), U(B), U(C)
T4	4	L(C), R(C), L(A), W(A), U(C), U(A)

Avec les temps d'exécution de chaque opération (en unité processeur) :

Opération	Temps processeur	Définition
L(X)	1	Demande de verrouillage sur la variable X
R(X)	3	Lecture de la variable X
W(X)	3	Écriture sur la variable X
U(X)	1	Libération de la variable X
Restart()	3	Redémarrage
KilledBy(P)	3	Interruption par le processus P
Wait(X, P)	_	Attente de la variable X détenue par le processus P

Q 1. Représentez le graphe des dépendances de cette exécution. Existe-t-il un interblocage ?



Il existe un cycle dans le graphe des dépendances donc il y a un risque d'interblocage.
Q 2. Déroulez la séquence d'exécution décrite précédemment en utilisant la méthode wait or die, puis la méthode wound or wait. Expliquez comment chaque méthode a évité l'interblocage.

Wound-wait vs. Wait-Die :

Wound-Wait

TS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
T1	L(A)	R(A)			W(A)			L(B)	W(B)			U(A)	U(B)																									
T2		L(A)	Wait(A,T1)										L(A)	L(C)	R(C)			W(A)			U(C)	U(A)																
T3			L(B)	R(B)			L(C)	W(C	KilledBy(T1)			L(B)	Wait	L(B)	R(B)			L(C)	Wait(C, T2)			L(C)	W(C)			U(B)	U(C)										
T4				L(C)	R(C)			Killed by T3			L(C)	R(C)			KilledBy(T2)			L(C)	Wait(C, T2)			L(C)	Killed by(T3)			L(C)	Wait	L(C)	R(C)			L(A)	W(A)			U(C)	U(A)

au temps 22, une autre exécution possible est que T3 obtienne la ressource C en premier et que T4 soit en attente.

Wait-Die

TS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
T1	L(A)	R(A)			W(A)			L(B)	Wait(B, T3)					L(B)	W(B)			U(A)	U(B)																			
T2		L(A)	Restart			L(A)	Restart			L(A)	Restart			L(A)	Restart			L(A)	Restart			L(A)	Wait(A, T4)			L(A)	L(C)	R(C)			W(A)			U(C)	U(A)		
T3			L(B)	R(B)			L(C)	Wait	L(C)	W(C)			U(B)	U(C)																								
T4				L(C)	R(C)			L(A)	Restart			L(C)	Restart			L(C)	R(C)			L(A)	W(A)			U(C)	U(A)												

	Start							
T1	1	L(A)	R(A)	W(A	L(B)	W(B	U(A)	U(B)
T2	2	L(A)	L(C)	R(C)	W(A	U(C)	U(A)	
T3	3	L(B)	R(B)	L(C)	W(C	U(B)	U(C)	
T4	4	L(C)	R(C)	L(A)	W(A	U(C)	U(A)	

L()	1
R()	3
W()	3
U()	1
Restart	3
Kill	3
Wait	

La première technique W-W évite l'interblocage lorsque T1 tue T3 qui aurait sinon bloqué B en attendant que T4 libère C lui même attendant que T1 libère A.
La seconde technique W-D évite l'interblocage par le fait que T4 redémarre lorsqu'il voit que T1 détient A, libérant ainsi C pour T3 qui peut alors libéré B pour T1.

Q 3. Comparez les deux exécutions et dites quels sont les avantages et les inconvénients de l'une par rapport à l'autre en termes de nombre d'interruptions et de temps processeur.

Rappel : les deux méthodes utilisent les estampilles (timestamp) pour ordonner les transactions, donnant toujours priorité aux plus anciennes. Pour W-D cette priorité se traduit par le fait que la transaction ancienne a le droit d'attendre, et c'est lorsqu'une transaction jeune se rend compte a qu'un blocage est possible avec une transaction plus ancienne qu'elle redémarre. Pour W-W la transaction la plus ancienne a une priorité forte, dans le sens ou elle s'exécutera toujours sans interruption, c'est la transaction ancienne qui prend alors l'initiative d'interrompre une plus jeune qui pourrait créer un blocage.

Wait-Die a plus d'interruption que Wound-Wait, 7 contre 4. Pour comprendre ce phénomène il faut voir qu'une transaction ancienne a en général obtenue tout ses lock, or quand une transaction jeune démarre et demande ses lock, il y a de forte chance qu'ils soient déjà pris par une plus ancienne transaction. Pour W-W cela entraîne un wait, donc évite tout interblocage. Or pour W-D cela entraîne une interruption.
W-W a plus de perte de temps processeur que W-D, 10 contre 3 (temps perdu à la réalisation de taches "utiles"). Cela s'explique de la même manière que précédemment. W-D va tendre à interrompre des transactions quand elles sont plus jeunes, or plus une transaction est jeune moins elle a eu de possibilités d'effectuer des actions "utiles". Son interruption a donc moins de chances d'entraîner une perte de temps processeur.

Q 4. (optionnel) Que se passe-t-il si on change la séquence du processus 4 par : L(A), L(D), R(D), W(A), U(D), U(A) ? Qu'est-ce que cela indique sur le comportement des deux méthodes en terme de temps d'exécution ?

Wound-wait :

Wound-Wait																	
TS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T1	L(A)	R(A)			W(A)			L(B)	W(B)			U(A)	U(B)				
T2		L(A)	Wait(A,T1)										L(A)	L(C)	R(C)		
T3			L(B)	R(B)			L(C)	W(C)	KilledBy(T1)			L(B)	Wait	L(B)	R(B)		
T4				L(A)	Wait(A,T1)								L(A)	Wait(A,T1)			

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	W(A)			U(C)	U(A)																	
	L(C)	Wait(C, T2)			L(C)	W(C)			U(B)	U(C)												
						L(A)	L(D)	R(D)			W(A)			U(D)	U(A)							

(Là aussi, au temps 13, un autre scénario est possible si T4 s'accapare A avant T2 – il sera alors blessé par celui-ci...)

Wait-Die :

Wait-Die																			
TS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T1	L(A)	R(A)			W(A)			L(B)	Wait(B, T3)			L(B)	W(B)			U(A)	U(B)		
T2		L(A)	Restart			L(A)	Restart			L(A)	Restart			L(A)	Restart			L(A)	L(C)
T3			L(B)	R(B)			L(C)	W(C)			U(B)	U(C)							
T4				L(A)	Restart			L(A)	Restart			L(A)	Restart			L(A)	Restart		

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
R(C)			W(A)			U(C)	U(A)										
L(A)	Wait(A,T2)							L(A)	L(D)	R(D)			W(A)			U(D)	U(A)

Attention, petite erreur sur le dernier schéma : au temps 21, T4 est redémarré et non mise en attente !
Dans le cas précédent (questions 2 & 3), W-D fait mieux en terme de temps d'exécution car il permet de lancer T1 et T3 en parallèle : en effet il n'y a pas de risque d'interblocage entre ces deux processus. Alors que W-W va interrompre T3 "inutilement", il aurait été plus rentable de le laisser finir. Dans ce dernier cas (question 4), la ressource A est tout de suite utilisée par trois transactions. Dans ce cas T1, T2, T4, n'ont pas le choix de s'exécuter séquentiellement. W-W fonctionne bien car la seule transaction qui n'utilise pas A est une des plus jeunes et donc avec les priorités elle n'interfère pas dans la séquence T1, T2, T4. Alors que W-D T1 attend "inutilement" T3. On peut donc voir qu'en terme de temps d'exécution aucune de deux méthodes ne montre d'avantage systématique (cela dépend du scénario sous-jacent).

Rappel : fonctionnement de base des algorithmes Wait or Die et Wound or Wait
Soit $TS(T_i)$ l'estampille à laquelle la transaction a démarré.
Soit deux transactions T_i et T_j, si T_i essaye d'accéder à la ressource R, mais que celle ci est déjà verrouillée par le processus T_j, la règle à appliquer est la suivante, en fonction de la méthode choisie :

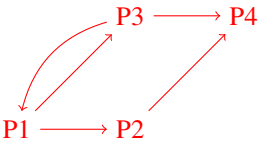
- **wait-die :** si $TS(T_i) < TS(T_j)$ (T_i est plus ancien que T_j) alors T_i peut attendre, sinon T_i est interrompue (tuée) et redémarre plus tard avec la même estampille de démarrage.
- **wound-wait :** si $TS(T_i) < TS(T_j)$ (T_i est plus ancien que T_j) alors T_j est interrompue (T_i blesse T_j) et est redémarrée plus tard avec la même estampille de démarrage, sinon T_i peut attendre.

Exercice 3 : Algorithme de détection d'interblocage (modèle basé sur la communication de messages)

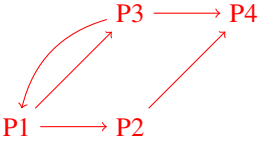
Application de l'algorithme de Chandy, Misra et Haas (dont le code est fourni en annexe 1) au scénario dont l'état initial est le suivant :

- le processus P1 est passif, et attend un message en provenance de P2 ou de P3 ;
- le processus P2 est passif, et attend un message en provenance de P4 ;
- le processus P3 est passif, et attend un message en provenance de P1 ou de P4 ;
- le processus P4 s'exécute.

Q 1. Tracez le graphe des dépendances à l'état initial. Est-ce qu'il inclut une CFTC (composante fortement connexe terminale) ?

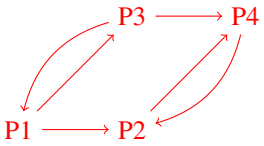


Q 2. Complétez les messages de requête et de réponse pour les calculs diffusants, ainsi que l'évolution des variables locales des processus selon l'algorithme de Chandy, Misra et Haas, dans la séquence d'événements suivante (tracez aussi l'évolution du graphe des dépendances au fur et à mesure du déroulement du scénario) :

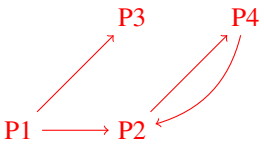


1. P1 initie son premier calcul diffusant :
   dernier1(1) ← 1 ; attendre1(1) ← vrai ; num1(1) ← 2 ;
   P1 envoie requête (1, 1, 1, 2) et requête (1, 1, 1, 3)
2. P2 reçoit requête (1, 1, 1, 2) :
   père2(1) ← 1 ; dernier2(1) ← 1 ; attendre2(1) ← vrai ; num2(1) ← 1 ;
   P2 envoie requête (1, 1, 2, 4)

3. P3 reçoit requête (1, 1, 1, 3) :
   père3(1) ← 1 ; dernier3(1) ← 1 ; attendre3(1) ← vrai ; num3(1) ← 2 ;
   P3 envoie requête (1, 1, 3, 1) et requête (1, 1, 3, 4)
4. P4 envoie un message à P3
5. P1 reçoit requête (1, 1, 3, 1) :
   P1 envoie réponse (1, 1, 1, 3)
6. P4 passe de l'état actif à l'état passif et attend un message de P2



7. P4 reçoit requête (1, 1, 2, 4) envoyée au (2) :
   père4(1) ← 2 ; dernier4(1) ← 1 ; attendre4(1) ← vrai ; num4(1) ← 1 ;
   P4 envoie requête (1, 1, 4, 2)
8. P4 reçoit requête (1, 1, 3, 4) envoyée au (3) :
   P4 envoie réponse (1, 1, 4, 3)
9. P3 reçoit le message de P4 envoyé au (4) :
   $\forall i$ attendre3(i) ← faux ;



10. P4 initie son premier calcul diffusant :
    dernier4(4) ← 1 ; attendre4(4) ← vrai ; num4(4) ← 1 ;
    P4 envoie requête (4, 1, 4, 2)
11. P2 reçoit requête (1, 1, 4, 2) envoyée au (7) :
    P2 envoie réponse (1, 1, 2, 4)
12. P3 reçoit réponse (1, 1, 4, 3) envoyée au (8) :
    ignore car actif.
13. P2 reçoit requête (4, 1, 4, 2) envoyée au (10) :
    père2(4) ← 4 ; dernier2(4) ← 1 ; attendre2(4) ← vrai ; num2(4) ← 1 ;
    P2 envoie requête (4, 1, 2, 4)
14. P4 reçoit réponse (1, 1, 2, 4) envoyée au (11) :
    : num4(1) ← 0
    P4 envoie réponse (1, 1, 4, 2)
15. P4 reçoit requête (4, 1, 2, 4) envoyée au (13) :
    P4 envoie réponse (4, 1, 4, 2)
16. P2 reçoit réponse (1, 1, 4, 2) envoyée au (14) :
    num2(1) ← 0
    P2 envoie réponse (1, 1, 2, 1)
17. P1 reçoit réponse (1, 1, 2, 1) envoyée au (15) :
    num1(1) ← 1
18. P2 reçoit réponse (4, 1, 4, 2) envoyée au (15) :
    num2(4) ← 0
    P2 envoie réponse (4, 1, 2, 4)
19. P4 reçoit réponse (4, 1, 2, 4) :
    num2(4) ← 0
    P4 est déclaré interbloqué.

Q 3. Montrez que si l'initiateur d'un calcul diffusant se déclare interbloqué, alors il appartient à un ensemble de processus interbloqués.

Commençons par une observation : si un processus envoie des requêtes pour un calcul diffusant (i, m) (le m-ième initié par i), puis devient actif, il ignorera tous les messages du calcul (i, m), même après être redevenu passif (voir l'algo). Ainsi, si l'ensemble de dépendance d'un processus change, il ne participe plus au calcul diffusant. Donc toute réponse envoyée au temps t par un processus P_k, signifie bien qu'une réponse a été reçue provenant de chacun des processus de l'ensemble de dépendance, au temps t, du processus P_k.

Soit S l'ensemble des processus, incluant l'initiateur, qui ont reçu les requêtes du dernier calcul diffusant. Supposons par l'absurde que l'initiateur se déclare interbloqué mais que S n'est pas un ensemble de processus interbloqués. Soit Pk le *premier* processus à redevenir actif après avoir envoyé une réponse à son père (ou après s'être déclaré interbloqué si Pk est l'initiateur), au temps t. Comme Pk a envoyé une réponse, c'est qu'il a reçu les réponses de tous les processus de son ensemble de dépendance Dk au temps t (voir observation et propriété (v) dans la preuve détaillée). Or, pour que Pk reprenne effectivement son exécution, il doit avoir reçu un message de l'un des processus Pr de son ensemble de dépendance Dk. Comme Pk a repris son exécution après avoir répondu à son père, il a reçu le message de Pr après la réponse de Pr. Donc Pr a envoyé une réponse (à Pk) puis a repris son exécution au temps $t' < t$. Cela contredit les propriétés attribuées à Pk au début de l'énoncé (premier processus à redevenir actif).

Note sur l'observation précédente : si la propriété de l'algo n'était pas présente, c-à-d si un algo continuait à participer au calcul diffusant (i, m) alors qu'il a changé d'ensemble de dépendance, alors il pourrait envoyer une réponse à son père, au temps t, en pensant qu'il a reçu suffisamment de réponse de son ensemble de dépendance précédent, alors que son nouvel ensemble de dépendance contient un processus actif, qui peut lui envoyer un message au temps $t + 1$, le rendant actif (ce qui montre qu'un processus, peut envoyer une réponse puis être actif sans créer de contradiction).

Seconde version (plus longue et détaillée) de la même preuve :

Pour cela commençons par prouver la propriété suivante (noté (v) dans le cours) :
si un processus Pk répond à son père puis reprend son exécution à un instant t, alors il existe un processus Pr appartenant à l'ensemble de dépendance de Pk (cet ensemble étant déterminé à l'instant où Pk a répondu à son père) qui reçoit une requête puis reprend son exécution à un temps $t' < t$.

Preuve de (v) : si Pk répond à son père, alors il a reçu les réponses de tous les processus de son ensemble de dépendance. Or, pour que Pk reprenne son exécution, il doit avoir reçu un message de l'un des processus Pr de son ensemble de dépendance. Comme Pk a repris son exécution après avoir répondu à son père, il a reçu le message de Pr après la réponse de Pr. Donc Pr a envoyé le message à Pk après lui avoir envoyé la réponse. Ainsi la séquence d'événements est la suivante : Pr reçoit la requête de Pk, lui répond, puis reprend son exécution (instant t'), et envoie un message à Pk qui réactive Pk (instant t) $\Rightarrow t' < t$.

Reprenons maintenant la preuve de la propriété (ii) du cours, celle demandée dans l'énoncé :

Soit S l'ensemble des processus, incluant l'initiateur, qui ont reçu les requêtes pendant le m^e calcul diffusant. Nous allons montrer par l'absurde que S est bien un ensemble de processus interbloqués.

Chaque processus qui répond à son père dans le m^e calcul diffusant, a forcément reçu les réponses de tous les processus de son ensemble de dépendance. On peut donc en déduire (par induction) que si l'initiateur se déclare interbloqué il a reçu toutes les réponses à sa requête concernant le m^e calcul diffusant.

Par conséquent, chaque processus de S répond à son père dans le m^e calcul diffusant. Or d'après la propriété (v) du cours et démontrée ci-dessus, si le processus Pk de S reprend son exécution à l'instant t après avoir reçu la requête, alors il a été réactivé par un processus Pr de S qui a repris son exécution à l'instant $t' < t$ (après avoir reçu la requête). En raisonnant par induction sur l'ensemble S qui décrit nécessairement un circuit dans le graphe de dépendance (plus précisément une CFCT car l'initiateur ne pourrait se déclaré interbloqué car il ne recevrait pas tous les messages en attente sinon), $\exists Pj \in S$ ayant repris son exécution à l'instant $t1$, puis qui a repris son exécution à l'instant $t'1$ (lorsque le circuit reviendra sur Pj) avec $t'1 < t1$, ce qui est impossible (car $t1$ devrait être alors exactement égal à $t'1$).

On a donc prouvé par l'absurde qu'il n'est pas possible qu'un processus de l'ensemble S ait repris son exécution après avoir envoyé sa réponse : par conséquent les processus de l'ensemble S sont bel et bien interbloqués.

Exercice 4 : Introduction à l'évitement d'interblocage – définitions et sûreté

On considère un système comprenant 15 cases de mémoire (ressources banalisées) partagées par 3 processus.

À l'instant $t1$, on a l'état suivant avec $Disponible = (0)$:

Processus	Détenu	Besoin
P_1	3	0
P_2	6	2
P_3	6	2

À l'instant t2, on a l'état suivant avec $Disponible = (0)$:

Processus	Détenu	Besoin
P_1	5	1
P_2	5	2
P_3	5	3

À l'instant t3, on a l'état suivant avec $Disponible = (3)$:

Processus	Détenu	Besoin
P_1	2	1
P_2	7	2
P_3	3	3

Q 1. Formalisez les termes suivants :
— Quand est-ce qu'un processus P_i est dit *bloqué* à l'état l ?
— Quand est ce que le système est dit en *interblocage* à l'état l ?
Un processus P_i est dit bloqué en l ssi $\exists j : Besoin_{i,j}(l) > Disponible_j(l)$.
Le système est en interblocage en l ssi :
— Il existe un ensemble D de processus bloqués ;
— Pour chaque processus $P_i \in D$, il existe au moins une ressource j telle que :
$Besoin_{i,j}(l) > Disponiblej(l) + \sum_{x \notin D} Detenu_{x,j}(l)$

Q 2. Pour les instants $t1$, $t2$ et $t3$, dites s'il y a interblocage. On montrera que l'état du système est sûr ou non.

À l'instant t1, il n'y a pas d'interblocage car :
— P_1 n'est pas bloqué : $Besoin = 0$; il peut donc s'exécuter sans demande supplémentaire de ressources ;
— P_2 et P_3 ne constituent pas l'ensemble D (selon la définition) car :
  — $Besoin_{P_2} < Disponible + Detenu_{P_1}$;
  — Idem pour P_3
L'état correspondant à cet instant est sûr car à la fin de P_0, les 3 ressources libérées pourront être utilisées pour satisfaire les demandes de P_1 et P_2.

À l'instant t_2, il y a interblocage car tous les processus sont bloqués en attente de ressources, alors qu'aucune n'est disponible. L'état est donc non sûr.

À l'instant t_3, il n'y a pas d'interblocage car aucun processus n'est déjà bloqué. Il y a suffisamment de ressources (3) pour satisfaire n'importe quelle demande. L'état correspondant est sûr car il est possible de satisfaire les demandes de P0 et P_1 en même temps ou bien celle de P_2.
Après libération des ressources, d'autres demandes seront satisfaites. Il faut noter qu'à partir d'un tel état on peut s'orienter vers un interblocage si P_3 demande 2 ressources, puis P_2 demande 1 ressource. Dans ce cas, aucune ressource ne devient disponible alors que chacun des trois processus aura besoin d'une ressource.
L'état à l'instant t_3 est néanmoins sûr car il suffit d'avoir une seule exécution évitant l'interblocage. Cette exécution est celle ayant pour chaque demande le besoin maximum.

On suppose maintenant que la demande maximale de chacun des processus est de 10 cases mémoire et le besoin réel est inconnu.
Q 3. Les états du système aux instants $t1$, $t2$ et $t3$ sont-ils sûrs ?
Le besoin n'est désormais pas précis, il va falloir raisonner par rapport à la demande maximale.
À l'instant t1, on a l'état suivant :

Processus	Détenu	Besoin	Disponible
P_0	3	10-3=7	0
P_1	6	10-6=4	
P_2	6	10-6=4	

L'état est non sûr car en raisonnant par rapport à l'annonce, aucune demande ne peut être satisfaite. À l'instant $t2$, on a l'état suivant :

Processus	Détenu	Besoin	Disponible
P_0	5	5	0
P_1	5	5	
P_2	5	5	

L'état n'étant pas sûr déjà avant, il reste non sûr. À l'instant $t3$, on a l'état suivant :

Processus	Détenu	Besoin	Disponible
P_0	2	10-2=8	3
P_1	7	10-7=3	
P_2	3	10-3=7	

L'état est sûr car en raisonnant par rapport à l'annonce, la demande de P_1 (3) peut être satisfaite. La libération des ressources de P_1 permettra de satisfaire les demandes de P_0 et de P_2.

Exercice 5 : Évitement d'interblocage avec l'algorithme du banquier – facultatif (non évalué en CC)

On considère 3 ressources R_1, R_2 et R_3, disposant chacune de 3 unités, et 3 processus devant exécuter les tâches suivantes :

Processus	Tâche	Demande	Rend
P_1	$T_{1,1}$	(1,0,2)	(0,0,0)
	$T_{1,2}$	(0,2,0)	(1,2,2)
P_2	$T_{2,1}$	(1,2,0)	(1,1,0)
	$T_{2,2}$	(1,0,0)	(1,0,0)
	$T_{2,3}$	(0,0,2)	(0,1,2)
P_3	$T_{3,1}$	(1,1,0)	(1,0,0)
	$T_{3,2}$	(1,1,0)	(0,0,0)
	$T_{3,3}$	(1,0,0)	(1,0,0)
	$T_{3,4}$	(0,0,2)	(1,2,2)

Q 1. Montrez que la séquence d'exécution $T_{1,1}$ $T_{2,1}$ $T_{3,1}$ $f_{T_{3,1}}$ $f_{T_{2,1}}$ $T_{2,2}$ $T_{3,2}$ $f_{T_{3,2}}$ $f_{T_{2,2}}$ $T_{3,3}$ $f_{T_{3,3}}$ $f_{T_{1,1}}$ aboutit à un interblocage.

Toutes les tâches sont démarrées au plus tôt sans vérification des pires cas virtuels, i.e. sans évitement (voir Fig. 3).

On obtient les allocations et libérations de ressources suivantes :

Processus	Tâche	Demande	Rend	Disponible
				(3,3,3)
P_1	$T_{1,1}$	(1,0,2)		(2,3,1)
P_2	$T_{2,1}$	(1,2,0)		(1,1,1)
P_3	$T_{3,1}$	(1,1,0)		(0,0,1)
P_3	$f_{T_{3,1}}$		(1,0,0)	(1,0,1)
P_2	$f_{T_{2,1}}$		(1,1,0)	(2,1,1)
P_2	$T_{2,2}$	(1,0,0)		(1,1,1)
P_3	$T_{3,2}$	(1,1,0)		(0,0,1)
P_3	$f_{T_{3,2}}$		(0,0,0)	(0,0,1)
P_2	$f_{T_{2,2}}$		(1,0,0)	(1,0,1)
P_3	$T_{3,3}$	(1,0,0)		(0,0,1)
P_3	$f_{T_{3,3}}$		(1,0,0)	(1,0,1)
P_1	$f_{T_{1,1}}$		(0,0,0)	(1,0,1)

Restent ensuite les tâches suivantes :

Processus	Tâche	Demande	Disponible
P_1	$T_{1,2}$	(0,2,0)	(1,0,1)
P_2	$T_{2,3}$	(0,0,2)	(1,0,1)
P_3	$T_{3,4}$	(0,0,2)	(1,0,1)

On observe que les trois processus sont bloqués et forment un ensemble D en interblocage.

Une autre représentation possible est illustrée en Fig. 3.

Q 2. En appliquant l'algorithme du banquier au cours de cette exécution, quel serait le dernier état sûr ?

Pour calculer le dernier état sûr, il suffit de partir du premier état et d'enchaîner les états en vérifiant systématiquement que l'état fictif t_k résultant de l'état s_k n'est pas un interblocage. Les libérations sont toujours considérées comme des états sûrs. On rappelle que $t_k = [Besoin^t(k), Tenu^t(k)]$ avec :

$$\begin{aligned} Tenu^t(k) &= Tenu(k) \\ Besoin^t(k) &= \begin{cases} MAX_i - Tenu_i(k), \text{ si } Tenu_i(k) + Besoin_i(k) > 0 \\ 0 \text{ sinon} \end{cases} \end{aligned}$$

Notons qu'ici, $Besoin^t(k)$ correspond au besoin maximum (pire cas), i.e., en supposant que les processus ne libèrent jamais des ressources.

$\rightarrow T_{1,1}$?

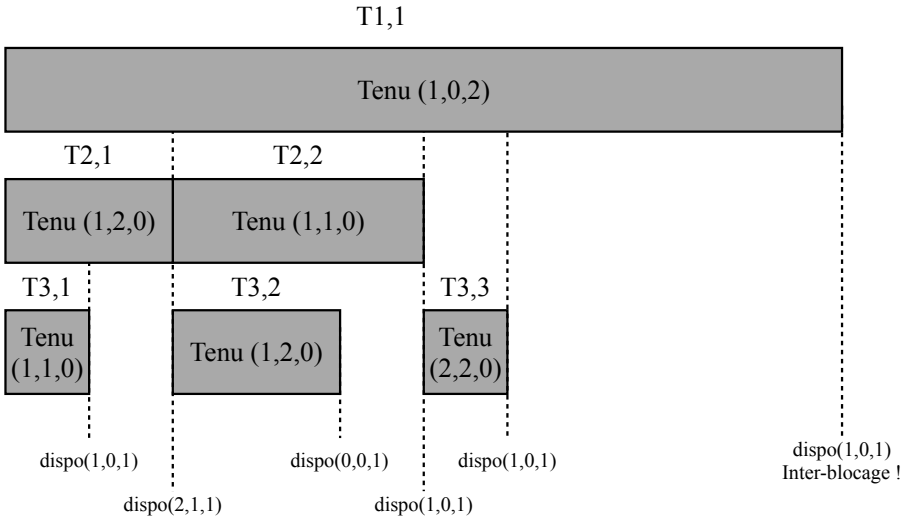


FIGURE 3 – Scénario découlant sur un interblocage

On commence par tenter de démarrer $T_{1,1}$:

$$T_{1,1} = \left[\begin{bmatrix}000\\120\\110\end{bmatrix}, \begin{bmatrix}102\\000\\000\end{bmatrix}\right] \text{ et } t_k = \left[\begin{bmatrix}020\\222\\322\end{bmatrix}, \begin{bmatrix}102\\000\\000\end{bmatrix}\right]$$

On rappelle que l'état s_l est un interblocage s'il existe un sous-ensemble D non vide de processus tel que, pour tout $i \in D$:

— P_i est en attente de ressource
— l'inégalité $Besoin_i(k) \leq Dispo(k) + \sum_{l \notin D} Tenu_l(k)$ est fausse
$\Leftrightarrow$ l'inégalité $Besoin_i(k) \leq N - \sum_{l \in D} Tenu_l(k)$ est fausse.

Ici, en t_k, nous avons $Dispo(k) = (2, 3, 1)$ et :
— $Besoin_1(k) = (0, 2, 0) \leq Dispo(k)$? Oui, donc P_1 n'est pas en attente de ressource
— $Besoin_2(k) = (2, 2, 2) \leq Dispo(k)$? Non, donc P_2 est en attente de ressource
— $Besoin_3(k) = (3, 2, 2) \leq Dispo(k)$? Non, donc P_3 est en attente de ressource

Nous avons donc $D = \{2, 3\}$ et les inégalités suivantes :
$Besoin_2^t(k) = (2, 2, 2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3, 3, 3) - (0, 0, 0) = (3, 3, 3)$
$Besoin_3^t(k) = (3, 2, 2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3, 3, 3) - (0, 0, 0) = (3, 3, 3)$

Donc t_k n'est pas un interblocage, et par conséquent $T_{1,1}$ **est un état sûr**.

$\rightarrow T_{1,1} \rightarrow T_{2,1}$?
On tente ensuite de démarrer $T_{2,1}$:

$$T_{2,1} = \left[\begin{bmatrix}000\\000\\110\end{bmatrix}, \begin{bmatrix}102\\120\\000\end{bmatrix}\right] \text{ et } t_k = \left[\begin{bmatrix}020\\102\\322\end{bmatrix}, \begin{bmatrix}102\\120\\000\end{bmatrix}\right]$$

Ici, en t_k, nous avons $Dispo(k) = (1, 1, 1)$ et :
— $Besoin_1(k) = (0, 2, 0) \leq Dispo(k)$? Non, donc P_1 est en attente de ressource
— $Besoin_2(k) = (1, 0, 2) \leq Dispo(k)$? Non, donc P_2 est en attente de ressource
— $Besoin_3(k) = (3, 2, 2) \leq Dispo(k)$? Non, donc P_3 est en attente de ressource

Nous avons donc $D = \{1, 2, 3\}$ et les inégalités suivantes :
$Besoin_1^t(k) = (0, 2, 0) \leq N - \sum_{l \in D} tenu_l^t(k) = (3, 3, 3) - (2, 2, 2) = (1, 1, 1)$
$Besoin_2^t(k) = (1, 0, 2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3, 3, 3) - (2, 2, 2) = (1, 1, 1)$
$Besoin_3^t(k) = (3, 2, 2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3, 3, 3) - (2, 2, 2) = (1, 1, 1)$

sont fausses ; t_k est donc un interblocage, et par conséquent $T_{2,1}$ **est un état non sûr**.
Le dernier état sûr était donc l'exécution $T_{1,1}$.

Q 3. Utilisez l'algorithme du banquier pour proposer une exécution réalisable.
On vient de voir qu'il était impossible de lancer $T_{2,1}$, essayons avec $T_{3,1}$.

$\rightarrow T_{1,1} \rightarrow T_{3,1}$?
Dès lors, la prochaine tâche pouvant être considérée est $T_{3,1}$:

$$T_{3,1} = \left[\begin{bmatrix}000\\120\\000\end{bmatrix}, \begin{bmatrix}102\\000\\110\end{bmatrix}\right] \text{ et } t_k = \left[\begin{bmatrix}020\\222\\212\end{bmatrix}, \begin{bmatrix}102\\000\\110\end{bmatrix}\right]$$

Ici, en t_k, nous avons $Dispo(k) = (1,2,1)$ et :
— $Besoin_1(k) = (0,2,0) \leq Dispo(k)$? Oui, donc P_1 n'est pas en attente de ressource
— $Besoin_2(k) = (2,2,2) \leq Dispo(k)$? Non, donc P_2 est en attente de ressource
— $Besoin_3(k) = (2,1,2) \leq Dispo(k)$? Non, donc P_3 est en attente de ressource

Nous avons donc $D = \{2,3\}$ et les inégalités suivantes :
$Besoin_2^t(k) = (2,2,2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3,3,3) - (1,1,0) = (2,2,3)$
$Besoin_3^t(k) = (2,1,2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3,3,3) - (1,1,0) = (2,2,3)$
sont vraies ; t_k n'est donc pas un interblocage, et par conséquent $T_{3,1}$ **est un état sûr**.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}}$?
Inutile de tenter à nouveau l'exécution de $T_{2,1}$, ce n'était déjà pas sûr avant celle de $T_{3,1}$. On se contente donc de terminer les tâches en cours.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2}$?
Suivant l'ordre lexicographique, nous pouvons maintenant tenter d'exécuter $T_{1,2}$:

$$T_{1,2} = \left[\begin{bmatrix}000\\120\\110\end{bmatrix}, \begin{bmatrix}122\\000\\010\end{bmatrix}\right] \text{ et } t_k = \left[\begin{bmatrix}000\\222\\212\end{bmatrix}, \begin{bmatrix}122\\000\\010\end{bmatrix}\right]$$

Ici, en t_k, nous avons $Dispo(k) = (2,0,1)$ et :
— $Besoin_1(k) = (0,0,0) \leq Dispo(k)$? Oui, donc P_1 n'est pas en attente de ressource
— $Besoin_2(k) = (2,2,2) \leq Dispo(k)$? Non, donc P_2 est en attente de ressource
— $Besoin_3(k) = (2,1,2) \leq Dispo(k)$? Non, donc P_3 est en attente de ressource

Nous avons donc $D = \{2,3\}$ et les inégalités suivantes :
$Besoin_2^t(k) = (2,2,2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3,3,3) - (0,1,0) = (3,2,3)$
$Besoin_3^t(k) = (2,1,2) \leq N - \sum_{l \in D} tenu_l^t(k) = (3,3,3) - (0,1,0) = (3,2,3)$
sont vraies ; t_k n'est donc pas un interblocage, et par conséquent $T_{1,2}$ **est un état sûr**.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2} \rightarrow T_{2,1}$?
La prochaine tâche à pouvoir être lancée en parallèle serait $T_{2,1}$. Cependant, P_1 détenant encore $(1,2,2)$, on voit directement que $(1,2,0) > (2,0,1)$, i.e., $T_{2,1}$ ne peut pas être exécutée tant que P_1 est en cours.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2} \rightarrow T_{3,2}$?
Suivant l'ordre lexicographique, nous pouvons maintenant tenter d'exécuter $T_{3,2}$ qui demande $(1,1,0)$ et détient encore $(0,1,0)$. Cependant, P_1 détenant encore $(1,2,2)$, on voit directement que $T_{3,2}$ ne peut pas être exécutée tant que $T_{1,2}$ est en cours.
La seule solution est donc de terminer $T_{1,2}$.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2} \rightarrow f_{T_{1,2}} \rightarrow T_{2,1}$?
Suivant l'ordre lexicographique, nous pouvons maintenant tenter d'exécuter $T_{2,1}$:

$$T_{2,1} = \left[\begin{bmatrix}000\\000\\110\end{bmatrix}, \begin{bmatrix}000\\120\\010\end{bmatrix}\right] \text{ et } t_k = \left[\begin{bmatrix}000\\102\\212\end{bmatrix}, \begin{bmatrix}000\\120\\010\end{bmatrix}\right]$$

Ici, en t_k, nous avons $Dispo(k) = (2,0,3)$ et :

— $Besoin_1(k) = (0,0,0) \leq Dispo(k)$? Oui, donc P_1 n'est pas en attente de ressource
— $Besoin_2(k) = (1,0,2) \leq Dispo(k)$? Oui, donc P_2 n'est pas en attente de ressource
— $Besoin_3(k) = (2,1,2) \leq Dispo(k)$? Non, donc P_3 est en attente de ressource
Seul $P3$ est bloqué, i.e., pas d'attente circulaire, $\rightarrow t_k$ n'est donc pas un interblocage. $T_{2,1}$ **est un état sûr**.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2} \rightarrow f_{T_{1,2}} \rightarrow T_{2,1} \rightarrow T_{3,2}$?
Suivant l'ordre lexicographique, nous pouvons maintenant tenter d'exécuter $T_{3,2}$: impossible car 1 unité demandée sur la 2ème ressource épuisée (1 unité monopolisée par P_3 car non rendue à l'issue de $T_{3,1}$ et 2 allouées à $T_{2,1}$ toujours en cours. On ne peut donc que terminer $T_{2,1}$.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2} \rightarrow f_{T_{1,2}} \rightarrow T_{2,1} \rightarrow f_{T_{2,1}} \rightarrow T_{2,2}$?
Aucune tâche n'est en cours, on tente de démarrer $T_{2,2}$:

$$T_{2,2} = \left[\begin{bmatrix}000\\000\\110\end{bmatrix}, \begin{bmatrix}000\\100\\010\end{bmatrix}\right] \text{ et } t_k = \left[\begin{bmatrix}000\\002\\212\end{bmatrix}, \begin{bmatrix}000\\100\\010\end{bmatrix}\right]$$

Ici, en t_k, nous avons $Dispo(k) = (2,0,3)$. Là encore, seul $P3$ est bloqué donc pas d'attente circulaire $\rightarrow t_k$ n'est pas un interblocage, et $T_{2,2}$ **est un état sûr**.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2} \rightarrow f_{T_{1,2}} \rightarrow T_{2,1} \rightarrow f_{T_{2,1}} \rightarrow T_{2,2} \rightarrow T_{3,2}$?
On tente de démarrer $T_{3,2}$ en parallèle :

$$T_{2,2} = \left[\begin{bmatrix}000\\002\\000\end{bmatrix}, \begin{bmatrix}000\\010\\110\end{bmatrix}\right] \text{ et } t_k = \left[\begin{bmatrix}000\\002\\102\end{bmatrix}, \begin{bmatrix}000\\010\\010\end{bmatrix}\right]$$

Ici, en t_k, nous avons $Dispo(k) = (2,1,3)$ et aucun processus n'est bloqué, d'où $D = \{\}$. Par conséquent, t_k n'est pas un interblocage, et $T_{3,2}$ **est un état sûr**.

$\rightarrow T_{1,1} \rightarrow T_{3,1} \rightarrow f_{T_{1,1}} \rightarrow f_{T_{3,1}} \rightarrow T_{1,2} \rightarrow f_{T_{1,2}} \rightarrow T_{2,1} \rightarrow f_{T_{2,1}} \rightarrow T_{2,2} \rightarrow T_{3,2} \rightarrow f_{T_{2,2}} \rightarrow f_{T_{3,2}}$?
On peut terminer ainsi les tâches de P_2 et P_3 avec $T_{2,3}//T_{3,3}$ puis $T_{3,4}$, soit l'exécution suivante :
$T_{1,1}, T_{3,1}, f_{T_{1,1}}, f_{T_{3,1}}, T_{1,2}, f_{T_{1,2}}, T_{2,1}, f_{T_{2,1}}, T_{2,2}, T_{3,2}, f_{T_{2,2}}, f_{T_{3,2}}, T_{2,3}, T_{3,3}, f_{T_{2,3}}, f_{T_{3,3}}, T_{3,4}, f_{T_{3,4}}$

On connait à présent les temps d'exécution de chaque tâche (cf. table 2).

Processus	Tâche	Durée
P_1	$T_{1,1}$	10 sec.
	$T_{1,2}$	2 sec.
P_2	$T_{2,1}$	2 sec.
	$T_{2,2}$	3 sec.
	$T_{2,3}$	1 sec.
P_3	$T_{3,1}$	1 sec.
	$T_{3,2}$	2 sec.
	$T_{3,3}$	1 sec.
	$T_{3,4}$	1 sec.

TABLE 2 – Temps d'exécution des tâches.

Q 4. Quel temps prendrait l'exécution calculée à l'aide de l'algorithme du banquier ?
On reprend l'exécution obtenue précédemment :
$T_{1,1}, T_{3,1}, f_{T_{1,1}}, f_{T_{3,1}}, T_{1,2}, f_{T_{1,2}}, T_{2,1}, f_{T_{2,1}}, T_{2,2}, T_{3,2}, f_{T_{2,2}}, f_{T_{3,2}}, T_{2,3}, T_{3,3}, f_{T_{2,3}}, f_{T_{3,3}}, T_{3,4}, f_{T_{3,4}}$
soit une durée totale d'exécution de 19 secondes.
Q 5. Est-il possible de faire mieux ? Pourquoi ?
Une exécution plus rapide est possible (qui ne dure que 16 secondes) :

$T_{1,1}, T_{3,1}, T_{3,1}, f_{T_{3,1}}, f_{T_{2,1}}, T_{2,2}, f_{T_{2,2}}, f_{T_{1,1}}, T_{1,2}, f_{T_{1,2}}, T_{3,2}, f_{T_{3,2}}, T_{3,3}, f_{T_{3,3}}, T_{3,4}, f_{T_{3,4}}$

Cette dernière exécution ne peut malheureusement pas être trouvée par l'algorithme du banquier (uniquement empiriquement), ce qui montre les limites de ce dernier.