ESSI 2005-2006 **Programmation Concurrente**
Deuxième année

Corrigé du TD2 : Réseaux Asynchrones – Exclusion Mutuelle

1. *Preuve de l'algorithme de Dekker*

Nous allons montrer que l'algorithme de Dekker résout le problème de l'exclusion mutuelle de manière équitable.

1.1 Montrer que les processus sont en exclusion mutuelle.

Rappelons l'algorithme de Dekker :

Algorithme Dekker : i = 0 ou 1

Processus P^i

Répéter

i.0	Ticket(i) := 0	/ P^i désire rentrer en section critique /
i.1	**Tant que** Ticket(i⊕1) = 0 **alors**	/ $P^{i\oplus1}$ désire aussi rentrer en section critique /
i.2	**Si** Tour = i⊕1 **alors**	/ c'est le tour de $P^{i\oplus1}$ /
i.3	Ticket(i) := 1	/ P^i laisse la place /
i.4	**Tant que** Tour = i⊕1 **alors** attendre	/ P^i attend /
i.5	Ticket(i) := 0	/ P^i redemande à rentrer en section critique /
i.6	Section_critique_i	
i.7	Tour = i⊕1	/ passe le tour à $P^{i\oplus1}$ /
i.8	Ticket(i) := 1	
i.9	Calculer_i	

Nous avons numéroté chacune des instructions selon le numéro du processus qui l'exécute. Nous supposons l'existence d'un temps discret initialisé à 0 et nous allons utiliser de manière non formalisée une logique temporelle. La signification de i.j(t) est vraie est qu'au temps t le processus i a déjà exécuté i.(j-1) (avec les bons modulo pour les boucles) et qu'il va exécuter i.j.

Montrer que les processus sont en exclusion mutuelle est donc équivalent à :

$$\mathbf{[\ (\exists\ t\ 0.6(t)) \Rightarrow \neg\ 1.6(t)\ ]\ et\ [\ (\exists\ t\ 1.6(t)) \Rightarrow \neg\ 0.6(t)\ ]\ /\ exclusion\ mutuelle\ /}$$

c'est-à-dire que les processus ne peuvent pas être simultanément dans l'état 6.

Par un raisonnement par l'absurde, soit donc t tel que 0.6(t) et 1.6(t). Nous déduisons de la propriété de progression que 0.0 ou 0.5 ont été exécutées juste avant, au temps t0 pour le processus 0 et t1 pour le processus 1, et donc l'existence de t*= max (t0,t1). Supposons que t*=t0 :

$$\mathbf{(0.6(t) \wedge 1.6(t)) \Rightarrow \forall\ \tau \in [t^*,t],\ Ticket(0)(\tau) = Ticket(1)(\tau) = 0}$$

Mais 1.1 a été exécutée entre t* et t, donc il existe $\mathbf{t^{**} \in [t^*,t]}$

$$\mathbf{Ticket(1)(t^{**}) = 1}$$

Ce qui est contradictoire.

1.2 Montrer qu'il n'y a ni interblocage, ni famine.

Nous devons montrer la viabilité des deux processus, c'est-à-dire qu'aucun processus ne reste bloqué. Comme nous supposons qu'aucun processus ne peut s'arrêter pendant la phase de section critique ou celle de calcul, nous devons montrer la propriété :

$$\exists t\ i.0(t) \Rightarrow \exists\ t'>t\ \ i.6(t')$$

Par l'absurde, nous supposons que

$$0.0(t) \wedge \textbf{jamais(0.6)}$$

Nous montrons d'abord que

$$0.0(t) \wedge \textbf{jamais(0.6)} \Rightarrow \textbf{toujours(Tour = 0)}$$

Montrons qu'il est impossible que :

$$0.0(t) \wedge \textbf{jamais(0.6)} \wedge \textbf{toujours(Tour = 1)}$$

Comme **jamais(0.6),** on a donc **toujours(Ticket(0)=1)**. Mais dans ce cas le processus 1 peut progresser, donc arriver à 1.7 et exécuter l'instruction correspondant ce qui implique que Tour=0 ce qui est contradictoire. Donc **toujours(Tour = 1)** est faux. Mais si Tour=0 à un certain instant, comme **jamais(0.6)**, il est impossible au processus 0 de modifier la valeur de Tour. Nous avons donc démontré que :

$$0.0(t) \wedge \textbf{jamais(0.6)} \wedge \textbf{toujours(Tour = 0)}$$

Ceci implique que le processus 0 boucle sur 0.1. Par conséquent :

$$\textbf{Toujours(Ticket(0) = Ticket(1) = 0)}$$

Mais ceci est impossible puisque le processus 1 peut progresser et donc exécuter 1.8 et mettre Ticket(1) à 1 ce qui conclut la démonstration par l'absurde. L'algorithme est donc viable. Il n'y a donc ni interblocage, ni famine.

1.3 Montrer que le système est équitable.

L'équité est garantie par la variable Tour. En cas de demande continuelle de section critique, l'instruction 1.8 garantit que l'autre processus aura accès à la ressource en un temps fini.

*2. **Algorithme de Dijkstra***

L'algorithme de Dekker a été généralisé à n processus par Dijkstra. Ticket et Entrée sont deux tableaux de n+1 variables booléennes partagées initialisées à faux, Tour est un entier initialisé à n.

Algorithme Dijkstra : i = 0, 1, …, n-1

Processus P^i

Répéter

i.0 Ticket(i) := vrai

Répéter

i.1 **Si** Tour ≠ i **alors** Entrée(i) := faux ;

i.2 **Tant que** Ticket(Tour) **alors** attendre

i.3 Tour := i

i.4 Entrée(i) := vrai

i.5 **Jusqu'à** $\neg(\vee_{j\neq i}$ Entrée(j))

i.6 Section_critique_i

i.7 Ticket(i) := faux ; Entrée(i) := faux ; Tour := n

i.8 Calculer_i

2.1 Montrer que pour n = 2, l'algorithme de Dijkstra est identique à celui de Dekker.

Ecrivons l'algorithme pour n=2, en substituant les valeurs 0 ou 1 dans le code.

Algorithme Dijkstra : i = 0, 1

Processus P^i

Répéter

Ticket(i) := vrai

Répéter

Si Tour ≠ i **alors** Entrée(i) := faux ;

Tour := i
Entrée(i) := vrai
Jusqu'à ¬Entrée(i⊕1)
Section_critique_i
Ticket(i) := faux ; Entrée(i) := faux ;
Tour := 2 ;
Calculer_i ;

Remarquons tout d'abord qu'il est possible de remplacer l'affectation Tour := 2 par Tour := i⊕1 sans modifier l'exécution du code. Il en est de même pour le test Tour ≠ i qui peut être remplacé par le test Tour = i⊕1. On peut donc remplacer le test sur Ticket(Tour) par le test Tour = i⊕1 dans la boucle d'attente. Il est alors possible de remplacer toutes les occurrences de Entrée par Ticket et de supprimer la variable Entrée. Les 2 codes sont alors équivalents.

2.2 Montrer que l'algorithme de Dijkstra résout le problème de l'exclusion mutuelle de manière équitable.

On reprend la méthode de démonstration de l'exercice 1. Montrer que les processus sont en exclusion mutuelle est équivalent à :

$$(\exists\, i\, \exists\, t,\ i.6(t)) \Rightarrow (\forall j \neq i, \neg\, j.6(t)) \quad \text{/ exclusion mutuelle /}$$

Par l'absurde, supposons que la propriété soit vraie simultanément pour i et k :

$$i.6(t) \text{ et } k.6(t)$$

De la progression de i et de k, on déduit :

progression de i : $\exists\, t_1 < t_2 < t\ \forall\, \tau \in [t_1,t]$, **Entrée(i)(τ) = vrai et Entrée(k)(t_2) = faux**

progression de k : $\exists\, t'_1 < t'_2 < t\ \forall\, \tau \in [t'_1,t]$, **Entrée(k)(τ) = vrai et Entrée(i)(t'_2) = faux**

Supposons que $t_1 < t'_1$. On déduit que $t'_2 \in [t_1,t]$ et donc une contradiction pour **Entrée(i)(t'_2)** qui doit être simultanément vrai et faux.

Nous devons montrer la viabilité des processus, c'est-à-dire qu'aucun processus ne reste bloqué. Comme nous supposons qu'aucun processus ne peut s'arrêter pendant la phase de section critique ou celle de calcul, nous devons montrer la propriété :

$$\exists t\ i.0(t) \Rightarrow \exists\, t' > t\ \ i.6(t')$$

Par l'absurde, nous supposons que, pour un certain i,

$$i.0(t) \wedge \text{jamais}(i.6)$$

Remarquons que :

$$\text{toujours}(\text{Ticket}(i) = \text{vrai})$$

Nous montrons d'abord que

$$i.0(t) \wedge \text{jamais}(i.6) \Rightarrow \text{toujours}(\text{Tour} = i)$$

S'il existe t0 > t pour lequel Tour (t0) = i alors, comme Ticket(i) est toujours vrai, nous en déduisons que Tour sera toujours égal à i. Or

$$\text{jamais}(\text{Tour}=i) \Rightarrow \text{toujours}(\text{Ticket}(\text{Tour})=\text{vrai})$$

Ceci implique que Tour ≠ n, et que :

$$\forall\, j \neq \text{Tour}, \text{toujours}(\text{Entrée}(j)=\text{faux})$$

Par conséquent, le processus Tour peut progresser, passer en section critique, en sortir et exécuter Tour.7 ce qui est contradictoire avec **toujours(Ticket(Tour)=vrai).**

Nous avons donc démontré

$$i.0(t) \wedge \text{jamais}(i.6) \Rightarrow \text{toujours}(\text{Tour} = i)$$

Mais ceci implique que :

$$\forall\, j \neq i\ , \text{toujours}(\text{Entrée}(j)=\text{faux})$$

Et donc que i.5 est satisfaite. Le processus peut donc progresser et passer en i.6 ce qui conclut cette partie de la démonstration.

L'équité est conséquence du fait que après passage en section critique, le processus qui possède le Tour, l'affecte à n. Aucun processus n'est donc avantagé.

3. ***Algorithme de Peterson***

3.1 On considère l'algorithme à deux processus suivant :

Algorithme Peterson*: i = 0 ou 1*

Processus P^i

Répéter

i.0 *Ticket(i) := vrai*
i.1 *Dernier := i*
i.2 ***Répéter*** *attendre* ***jusqu'à*** *((¬Ticket(i⊕1) ∨ (dernier = i⊕1))*
i.3 *Section_critique_i*
i.4 *Ticket(i) := faux*
i.5 *Calcul_i*

Montrer que cet algorithme résout le problème de l'exclusion mutuelle de manière équitable.

On reprend la méthode de démonstration de l'exercice 1. Montrons que les processus sont en exclusion mutuelle par l'absurde, en supposant que la propriété soit vraie simultanément :

0.3(t) et 1.3(t)

Nécessairement, Ticket(0)(t) = Ticket(1)(t) = vrai. Supposons que 0 soit arrivé le premier en 0.2. Ceci implique qu'à ce moment dernier = 1. Mais ceci implique que 1 ne peut pas franchir 1.2.

Montrer qu'il y a vivacité, absence de d'interblocage et de famine et équité, se déduit facilement du rôle joué par dernier.

3.2 Généraliser l'algorithme au cas de n processus.

Pour généraliser l'algorithme de Peterson à n processus, nous considérons que Ticket et Dernier sont des tableaux de n+1 variables entières partagées initialisées à -1. Chaque processus fait croître la valeur de son Ticket dans une boucle de 0 à n-1. Pour incrémenter la valeur de son Ticket de j à j+1, le processus i doit vérifier que tous les autres processus ont un Ticket de valeur strictement inférieure. Si un processus k a la même valeur j de Ticket, c'est Dernier(j) qui détermine le processus qui pourra passer à l'étape suivante.

Algorithme Peterson : i = 0, 1, …, n-1

Processus P^i

Répéter

Pour j = 0 **à** n-1

Ticket(i) := j ;

Dernier(j) := i ;

Répéter attendre **jusqu'à** ($\forall k \neq i$ Ticket(k) < Ticket(i) $\vee$ Dernier(j) $\neq$ i);

Section_critique_i ;

Ticket(i) := -1 ;

Calcul_i ;