3. Systèmes asynchrones à mémoire partagée – Exclusion mutuelle

Rappelons qu'un système asynchrone est un ensemble de processus qui possèdent chacun leur propre horloge. Ces horloges sont indépendantes les unes des autres. Dans ce chapitre, nous supposons que les interactions entre processus se font à travers une mémoire partagée. Comme nous l'avons vu dans le chapitre 1, l'accès aux variables partagées peut poser de problème que ce soit en lecture ou en écriture. L'étude des politiques d'accès est un cas particulier des problèmes de synchronisation. L'exclusion mutuelle est une généralisation de ces problèmes.

3.1 Propriétés d'un système asynchrone

Progression : Un processus d'un système synchrone progresse (normalement) si, à l'issue de l'exécution d'une phase, l'exécution de la phase suivante a lieu au bout d'un temps fini.

Viabilité : Un système asynchrone est viable si tous les processus, qui le composent, progressent.

Interblocage : Un système asynchrone est en interblocage si aucun des processus, qui le composent, ne peut progresser.

Famine : Si, dans un système asynchrone qui n'est pas en interblocage, un processus ne peut plus progresser, alors il est en situation de famine.

Equité : Un système asynchrone viable est équitable si, pour tout processus, la durée entre deux phases consécutives est bornée.

3.2 Exclusion mutuelle

Rappelons qu'un processus peut être considéré comme une suite ordonnée de phases de calcul et d'interactions.

Définition 3.1 : Des processus ont des phases en **exclusion mutuelle** (ou sont en **exclusion mutuelle)** si ces phases d'interactions ne peuvent pas s'exécuter simultanément. Une telle phase est appelée **section critique**.

Pour simplifier, nous allons donc supposer que les processus sont modélisés de la façon suivante :

```
Processus P^i
    Répéter
        Calculer_i
        Protocole_entrée_i
        Section_critique_i
        Protocole_sortie_i
```

Protocole_entrée_i est un ensemble d'opérations élémentaires exécutées par $\mathbf{P^i}$ de manière à garantir son entrée unique en section critique. **Protocole_sortie_i** est un ensemble d'opérations élémentaires exécutées par $\mathbf{P^i}$ de manière à garantir l'entrée d'un autre processus en section critique.

3.3 Algorithme de Dekker et de Dijkstra

Nous considérons tout d'abord le cas de deux processus et d'un protocole basé sur l'utilisation d'une variable partagée, Tour. Nous supposons que l'accès à Tour, en lecture ou en écriture, est réglé par un arbitre de mémoire qui donne accès au processus à tour de rôle. La valeur de Tour est celle du processus autorisé à entrer en section critique. Elle est initialisée arbitrairement à 0 ou à 1.

Algorithme D1 : i = 0 ou 1
Processus P^i
Répéter
Tant que Tour = i⊕1 **alors** attendre / Protocole_entrée_i /
Section_critique_i
Tour := i⊕1 / Protocole_sortie_i /
Calculer_i

Propriété 3.2 : D1 résout le problème de l'exclusion mutuelle. Mais D1 n'est pas équitable.

Démonstration : Tour ne peut prendre que les valeurs 0 ou 1. P^i ne peut rentrer en section critique que si Tour = i et Tour ne prend la valeur i⊕1 qu'après sa sortie de la section critique ce qui garantit l'exclusion mutuelle. Les deux processus ne peuvent être bloqués en attente car Tour ne peut prendre simultanément les valeurs 0 et 1. Donc, il n'y a pas d'interblocage. Aucun processus n'est en situation de famine car ils entrent consécutivement en section critique. Mais D1 n'est pas équitable puisque la durée entre deux entrées consécutives en section critique pour P^i dépend de la durée d'exécution de la section critique de $P^{i⊕1}$ qui peut grandir indéfiniment. %

Pour rendre l'algorithme équitable, on peut définir deux variables Ticket(0) et Ticket(1), initialisées à 1.

Algorithme D2 : i = 0 ou 1
Processus P^i
Répéter
Tant que Ticket(i⊕1) = 0 **alors** attendre
Ticket(i) := 0
Section_critique_i
Ticket(i) := 1
Calculer_i

Propriété 3.3 : D2 ne résout pas le problème de l'exclusion mutuelle.

Démonstration : Le scénario suivant est possible.

Phase	Ticket(0)	Ticket(1)
Initialement	1	1
P0 lit Tour(1)	1	1
P1 lit Tour(0)	1	1
P0 change Tour (0)	0	1
P1 change Tour (1)	0	0
P0 rentre en SC	0	0
P1 rentre en SC	0	0

Algorithme D3 : i = 0 ou 1
Processus P^i

Répéter
Ticket(i) := 0
Tant que Ticket(i⊕1) = 0 **alors** attendre
Section_critique_i
Ticket(i) := 1
Calculer_i

Propriété 3.4 : D3 peut conduire à un interblocage.

Démonstration : Le scénario suivant est possible.

Phase	Tour(0)	Tour(1)
Initialement	1	1
P0 change Tour (0)	0	1
P1 change Tour (1)	0	0
P0 lit Tour(0)	0	0
P1 lit Tour(1)	0	0

Pour assurer l'exclusion mutuelle, il faut combiner les algorithmes D1 et D3. On rajoute donc une variable Tour, initialisée à1.

Algorithme Dekker : i = 0 ou 1
Processus P^i
Répéter
Ticket(i) := 0 / P^i désire rentrer en section critique /
Tant que Ticket(i⊕1) = 0 **alors** / $P^{i⊕1}$ désire aussi rentrer en section critique /
Si Tour = i⊕1 **alors** / c'est le tour de $P^{i⊕1}$ /
Ticket(i) := 1 / P^i laisse la place /
Tant que Tour = i⊕1 **alors** attendre / P^i attend /
Ticket(i) := 0 / P^i redemande à rentrer en section critique /
Section_critique_i
Tour = i⊕1 / passe le tour à $P^{i⊕1}$ /
Ticket(i) := 1
Calculer_i

Théorème 3.5 : L'algorithme Dekker résout le problème de l'exclusion mutuelle, sans interblocage, sans famine et de manière équitable.

Démonstration : Voir le TD3. %

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
Ticket(i) := vrai
Répéter
Si Tour ≠ i **alors** Entrée(i) := faux ; **Tant que** Ticket(Tour) **alors** attendre

```
            Tour := i
            Entrée(i) := vrai
        Jusqu'à ¬(∨j≠i Entrée(j))
        Section_critique_i
        Ticket(i) := faux ; Entrée(i) := faux ; Tour := n
        Calculer_i
```

Théorème 3.6 : L'algorithme Dijkstra résout le problème de l'exclusion mutuelle, sans interblocage, sans famine et de manière équitable.

Démonstration : Voir le TD3. %

3.4 Les sémaphores

Les principaux inconvénients des solutions logicielles au problème de l'exclusion mutuelle sont que la programmation est délicate, que la preuve et la vérification sont difficiles, que la lisibilité est douteuse et qu'elles peuvent être insuffisantes pour certains processeurs modernes : la lecture/écriture d'une variable simple n'est plus nécessairement atomique. Des solutions matérielles sont parfois proposées comme les Test_and_Set. Une solution efficace et élégante est basée sur l'utilisation des sémaphores introduit par Dijkstra.

Définition 3.7 : Un **sémaphore** est une variable entière **s** positive ou nulle. Une fois **S** initialisée, les seules opérations admises sur s sont :

- **Attendre**(S) (ou P(S)) : **si** S>0 **alors** S := S-1 **sinon** suspendre l'exécution du processus appelant
- **Réveiller**(S) (ou V(S)) : **si** un processus a été suspendu lors d'un Attendre(S) antérieur **alors** le réveiller, **sinon** S := S+1

On appelle sémaphore binaire un sémaphore qui ne prend que des valeurs binaires. La stratégie de réveil (gestion de la file d'attente) peut être

- de type FIFO : on dit alors que le sémaphore est « fort » et il est équitable
- de type non FIFO : on dit alors que le sémaphore est « faible » et l'équité n'est pas garantie.

Un sémaphore binaire est un outil puissant pour résoudre le problème de l'exclusion mutuelle. Soit S une variable binaire initialisée à 1.

Algorithme EMS

```
    Processus P^i
        Répéter
            Attendre(S)
            Section_critique_i
            Réveiller(S)
            Calculer_i
```

Lorsqu'un processus réalise un Attendre et que S = 1, alors il met S à 0 et il peut entrer en section critique. Si S = 0, alors il suspend son exécution. Après avoir terminé sa section critique, un processus réveille un autre processus suspendu. S'il n'y en a pas, il remet S à 1. On comprend ici toute la puissance des opérations du sémaphore qui est testé et affecté en une unique opération non préemptable.

Théorème 3.8 : Si dans l'algorithme EMS, le sémaphore S est fort alors EMS résout le problème de l'exclusion mutuelle, sans interblocage, sans famine et de manière équitable.

Démonstration : Pour démontrer le théorème on considère l'invariant suivant :

INV = S + nombre de processus en SC

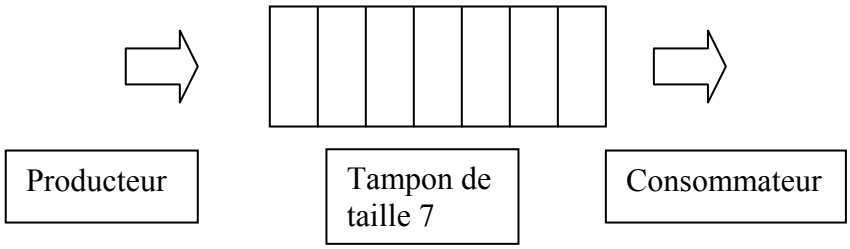
Nous allons montrer par récurrence que INV = 1. La propriété est vraie à l'initialisation de l'algorithme. Supposons la propriété vérifiée et exécution un phase de l'algorithme.

- Si la phase est Attendre(S) et si S=1, il n'y a pas de processus en SC (d'après l'invariant) et dans ce cas le processus met S à 0 et rentre en SC. Il maintient donc l'invariant.
- Si la phase est Attendre(S) et si S=0, il suspend son exécution et maintient donc l'invariant
- Si la phase est Réveiller(S) et qu'il y a des processus suspendu, il réveille un seul processus qui rentre en section critique. Sinon, il remet S à 1. Dans les deux cas, il maintient l'invariant.

Il ne peut y avoir ni interblocage, ni famine puisque la file d'attente est FIFO. Tout processus qui rentre en attente avant un autre processus rentre en SC avant ce processus. Pour la même raison, l'algorithme est équitable. %

3.5 Le problème des producteurs - consommateurs

Le problème des producteurs - consommateurs, comme celui de l'exclusion mutuelle, est une modélisation de certains problèmes de synchronisation informatique. Des processus (producteurs) génèrent des données qui doivent être traitées par des processus (consommateurs). Considérons d'abord le cas simple d'un seul processus producteur et d'un seul consommateur. Une solution immédiate à problème consiste à synchroniser directement les deux processus par rendez-vous : le producteur ne produit que si le consommateur est prêt à consommer. Cette solution est valable si les vitesses des deux processus sont proches ou varient peu dans le temps. Par contre, si les vitesses sont variables (pics de production ou de consommation), la vitesse globale du système sera à chaque instant le minimum des vitesses des processus. Cette solution n'est donc pas efficace.



Si les processus ont des vitesses variables pendant l'exécution du programme, il est nécessaire de disposer d'une mémoire tampon, commune au producteur et au consommateur. Ceci permet aux processus de travailler à leur rythme, mais nous devons veiller à ce qu'il n'y ait pas de débordement de tampon.

Appelons **P(t)** et **C(t)**, le nombre de données produites et consommées à un instant t. Notons :

S(t) = P(t) – C(t)

A chaque instant, **S(t)** doit être positive ou nulle et inférieure ou égale à la taille du tampon. La variable **S** peut donc être utilisée comme sémaphore.

Considérons pour commencer le cas d'un tampon de taille non bornée. Le producteur va donc produire, ajouter au tampon et réveiller le consommateur, alors que le consommateur va attendre le producteur, prendre dans le tampon et traiter la donnée. Nous obtenons la première solution suivante, dans laquelle **S** est initialisée à 0 :

Algorithme PC0

Producteur	**Consommateur**
Répéter	**Répéter**
Produire ;	Attendre(S) ;
Ajouter ;	Prendre ;
Réveiller(S) ;	Traiter ;

Une difficulté de cette solution est qu'elle suppose que les opérations Ajouter et Prendre se déroulent de manière indépendante. Supposons que le tampon soit en accès exclusif. Nous pouvons alors combiner les algorithmes PC0 et EMS pour garantir l'exclusion mutuelle de l'accès au tampon. Pour cela nous introduisons un deuxième sémaphore E, binaire et initialisé à 1.

Algorithme PC1 (tampon non borné)

Producteur	**Consommateur**
Répéter	**Répéter**
Produire ;	Attendre(S) ;
Attendre (E) ;	Attendre (E) ;
Ajouter ;	Prendre ;
Réveiller(E) ;	Réveiller(E) ;
Réveiller(S) ;	Traiter ;

Théorème 3.9 : L'algorithme PC1 résout le problème du producteur - consommateur avec un tampon de taille non bornée, sans interblocage, sans famine et de manière équitable.

Démonstration : Le théorème 3.8 nous garantit le bon fonctionnement de la section critique. S ne peut pas devenir négatif puisque c'est un sémaphore. Comme le consommateur n'accède au tampon qu'après avec avoir exécuté Attendre(S), il est donc impossible qu'il essaie de prendre dans le tampon vide. Par ailleurs après chaque ajout au tampon, le producteur réveille le consommateur s'il est attente. Comme, il n'y a que deux processus, il n'y a donc ni interblocage, ni famine et équité. %

Mais l'algorithme PC1 ne nous donne aucune indication sur la taille du tampon. En fait, si le producteur produit plus vite que le consommateur, la taille va croître indéfiniment. Supposons maintenant que le tampon a une taille c finie. Pour résoudre le problème, il faut suspendre la production lorsque le tampon est plein. Remarquons que cette situation est symétrique de celle du tampon vide : le sémaphore S nous garantit que le consommateur n'accédera jamais au tampon vide. Nous allons donc ajouter un sémaphore C, initialisé à c. Quand C est nul, le tampon est plein et il faut suspendre le producteur. Nous obtenons l'algorithme PC2.

Algorithme PC2 (tampon fini)

Producteur	**Consommateur**
Répéter	**Répéter**
Produire ;	Attendre(S) ;
Attendre(C) ;	Attendre(E) ;
Attendre(E) ;	Prendre ;
Ajouter ;	Réveiller(E) ;
Réveiller(E) ;	Réveiller(C) ;
Réveiller(S) ;	Traiter ;

Théorème 3.10 : L'algorithme PC2 résout le problème du producteur - consommateur avec un tampon de taille finie, sans interblocage, sans famine et de manière équitable.

Démonstration : La démonstration est identique à celle du théorème 3.9. %

Supposons pour terminer ce paragraphe que nous avons n producteurs et m consommateurs produisant et consommant les mêmes données. Si chaque producteur et chaque consommateur exécutent le code de l'algorithme PC2, le problème sera résolu sans interblocage. Par contre, pour garantir l'équité et l'absence de famine, il faut que les files d'attentes le permettent, par exemple en étant FIFO.

Algorithme PC3 (tampon fini)

Producteur^i	**Consommateur^j**
Répéter	**Répéter**
Produire ;	Attendre(S) ; / tampon vide /
Attendre(C) ; / tampon plein /	Attendre(E) ;
Attendre(E) ; / section critique /	Prendre ;
Ajouter ;	Réveiller(E) ;
Réveiller(E) ;	Réveiller(C) ;
Réveiller(S) ;	Traiter ;

Corollaire 3.11 : L'algorithme PC3, avec des sémaphores forts, résout le problème des n producteurs - m consommateurs avec un tampon de taille finie, sans interblocage, sans famine et de manière équitable.

Démonstration : La démonstration est identique à celle du théorème 3.10. %

Voir en TD l'implémentation avec un tampon circulaire.

3.6 Le problème des écrivains et des lecteurs

Le problème des écrivains et des lecteurs correspond à la situation où un seul processus (écrivain) a accès à la mémoire pour modifier des données et où plusieurs lecteurs peuvent accéder simultanément à la mémoire sans la modifier. C'est donc une généralisation de l'exclusion mutuelle dans laquelle on interdit l'accès simultané :

- à un lecteur et à un écrivain (no read-write conflict)
- à deux écrivains (no write-write conflict)

Cette situation formalise les conditions d'utilisation d'une base de données. Les processus exécutent le code suivant.

Ecrivain^i	**Lecteur^j**
Répéter	**Répéter**
Début_écriture ; / protocole d'entrée /	Début_lecture ;/ protocole d'entrée /
Ecriture ;	Lecture ;
Fin_écriture ; / protocole de sortie /	Fin_lecture ; / protocole de sortie /

Il nous faut donc définir les quatre protocoles garantissant les 2 conditions précédentes. Nous allons résoudre ce problème en utilisant des sémaphores et un compteur. Pour l'écriture, l'accès à la base doit se faire en exclusion mutuelle : un sémaphore binaire **Verrou** initialisé à 1. Les protocoles sont donc :

Début_écriture ;	**Fin_écriture** ;

Attendre(Verrou) ; Réveiller(Verrou) ;

Pour la lecture, nous devons d'abord garantir qu'aucun écrivain ne peut accéder à la base. Donc le premier lecteur doit verrouiller la base en utilisant le Verrou. Par contre, les autres lecteurs peuvent accéder sans problème. Pour la fin de lecture, le dernier lecteur doit lever le Verrou. Nous devons donc utiliser un compteur **nb_lect**, initialisé à 0, permettant de connaître le nombre de lecteurs accédant simultanément à la base et un deuxième sémaphore binaire **Prem_Dern_Lect**, initialisé à 1, garantissant que les opérations sur nb_lect s'exécutent en exclusion mutuelle.

Début_lecture ;	**Fin_lecture** ;
Attendre(Prem_Dern_Lect) ;	Attendre(Prem_Dern_Lect) ;
nb_lect++ ;	nb_lect-- ;
Si nb_lect = 1 **alors** Attendre(Verrou) ;	**Si** nb_lect = 0 **alors** Réveiller(Verrou) ;
Réveiller(Prem_Dern_Lect) ;	Réveiller(Prem_Dern_Lect) ;

Théorème 3.12 : Si dans l'algorithme Ecrivains-Lecteurs, le sémaphore Verrou est fort alors EMS résout le problème des écrivains et des lecteurs, sans interblocage, sans famine et de manière équitable.

Démonstration : Soit L(t) le nombre de processus en train de lire et E(t) le nombre de processus en train d'écrire. Nous devons démontrer que la formule logique I suivante est toujours vraie :

$$\forall\, t\,,\ I(t) = (L(t) > 0 \Rightarrow E(t) = 0) \wedge ((E(t) > 0 \Rightarrow (E(t) = 1 \wedge L(t) = 0)) = vrai$$

qui correspond à dire que s'il y a les lecteurs actifs alors aucun écrivain n'est actif et que s'il y a des écrivains actifs alors il n'y en a qu'un et il n'y a pas de lecteur actif. Nous pouvons réécrire I(t) en :

$$I(t) = (E(t) = 0 \vee L(t) = 0) \wedge (E(t) = 0 \vee (E(t) = 1 \wedge L(t) = 0))$$
$$= (E(t) = 0) \vee (E(t) = 1 \wedge L(t) = 0)$$

qui correspond à dire que soit il n'y a aucun écrivain actif, soit il y en a et aucun lecteur actif. I(0) est vraie puisque E(0) = 0. Supposons la formule vraie pour t-1 et démontrons la pour t.

Si E(t-1) = 0 et L(t-1) = 0 alors Verrou = 1 et Prem_Dern_Lect = 1. Si E(t) = 1, alors Verrou = 0. Si un lecteur a exécuté Début_lecture alors nb_lect = 1 et le lecteur est suspendu. Donc L(t) = 0.

Si E(t-1) = 0 et L(t-1) > 0 alors Verrou = 0. Aucun écrivain ne pourra débuter une écriture. Donc E(t) = 0.

Si E(t-1) = 1 et L(t-1) = 0 alors Verrou = 0. Aucun nouvel écrivain ne pourra débuter une écriture et donc E(t) = 0.

Pour démontrer la viabilité de cette solution, il faut montrer qu'un lecteur (écrivain) qui désire lire (écrire) finira par y être autorisé. Un tel lecteur L va exécuter Début_lecture. Supposons d'abord qu'il n'y a aucun écrivain actif. Si Prem_Dern_Lect = 1 alors il n'y a aucun lecteur actif. Donc L progresse, place le verrou à 1 et termine son protocole Début_lecture. Il est donc autorisé à lire. Si Prem_Dern_Lect = 0 alors L va être placé dans la FIFO correspondante. Aucun écrivain ne pourra placer le verrou et donc les lecteurs précédents progressent et finiront par réveiller L. S'il y a un écrivain actif, si Prem_Dern_Lect = 1 alors L va attendre sur le verrou correspondant à l'écrivain qui

va progresser et finira par réveiller L. Si Prem_Dern_Lect = 0 alors L va être placé dans la FIFO correspondante. L'écrivain finira par réveiller un lecteur précédent.

Pour démontrer la progression des écrivains, on utilise un raisonnement similaire. %

3.7 Les moniteurs

Les principaux inconvénients des solutions sémaphores sont leur rigidité et leur complexité d'utilisation en particulier lorsque plusieurs sémaphores sont utilisés de manière imbriquée. Un moniteur est un constructeur de plus haut niveau qui permet de traiter des problèmes de synchronisation complexe : chaque moniteur est chargé d'une tâche précise, a ses propres données et ses propres instructions, l'entrée d'un moniteur par un processus excluant l'entrée par un autre. L'exécution de moniteurs indépendants pourra se faire en parallèle et la modification de l'un n'affectera pas les autres.

Définition 3.13 : Un **moniteur** est un constructeur (synchronized object) composé de variables globales, de variables conditionnelles, de procédures (méthodes) pour les manipuler. Les seules opérations admises sur une variable conditionnelle C sont :

- **Attendre**(C) : suspendre l'exécution du processus appelant et le placer dans la FIFO correspondante
- **Réveiller**(C): **si** la FIFO de C n'est pas vide **alors** réveiller le premier processus suspendu

Nous étudions tout d'abord la simulation d'un sémaphore binaire par un processus. Cette propriété est importante pour justifier l'utilisation des moniteurs en montrant que nous n'avons pas perdu en expressivité ou en puissance. Pour cela nous allons construire un moniteur utilisant une variable booléenne **occupé** pour indiquer si une opération **Attendre** a été exécutée et une variable conditionnelle **Libre** pour attendre (au sens du moniteur) que le sémaphore se libère s'il est occupé.

Algorithme Moniteur Simulateur_Sémaphore

Variable booléenne occupé **initialisée à** faux
Variable conditionnelle Libre

Sém_Attendre	**Sém_Réveiller**
Si occupé **alors Attendre**(Libre) ;	occupé := faux ;
occupé := vrai ;	**Réveiller**(Libre) ;

Montrons que nous pouvons résoudre de manière élégante le problème des producteurs et des consommateurs avec un tampon de taille finie en utilisant un moniteur. Pour cela nous considérons un moniteur correspondant au tampon avec les 2 opérations Ajouter et Prendre. Le code des processus Producteur et Consommateur est alors très simple :

Algorithme PC_moniteur (tampon fini)

Producteur^{i}	**Consommateur**^{j}
Répéter	**Répéter**
Produire(élément) ;	Prendre(élément)
Ajouter(élément) ;	Traiter(élément);

Pour construire le moniteur, nous définissons un tableau de c cases et des variables s, correspondant au nombre d'éléments dans le tampon, dern, le numéro de la première case libre et prem, celui de la première case pleine. Si on souhaite ajouter, il faut attendre si le tampon est plein. Sinon on ajoute dans le première case libre et on signale que le tampon est non vide.

```
Algorithme Moniteur Tampon_fini
Tableau T[0,..,c-1] d'éléments
Variable entière s, prem, dern initialisée à 0
Variable conditionnelle Non_vide, Non_plein
Ajouter(élément)
        Si s=c alors Attendre(Non_plein) ;
        T[dern] := élément ; dern ++ ;
        Si dern =c+1 alors dern:= 0 ;
        s++ ;
        Réveiller(Non_vide) ;
Prendre (élément)
Si s=0 alors Attendre(Non_vide) ;
        élément := T[prem] ; prem ++ ;
        Si prem =c alors prem:= 0 ;
        s-- ;
        Réveiller(Non_plein) ;
```