CHAPITRE 9

MÉMOIRE COMMUNE :
CONTRÔLE DE CONCURRENCE PAR SÉMAPHORES

PROGRAMMATION DES PARADIGMES
exclusion mutuelle
cohorte
lot de ressources banalisées
passage de témoin
producteurs-consommateurs
lecteurs-rédacteurs
repas des philosophes
SCHÉMA DES SÉMAPHORES PRIVÉS
repas des philosophes
allocation de ressources banalisées
MONITEURS
Java
Ada

Manuel 9

Mémoire commune :
Contrôle de concurrence par sémaphores

1. Programmation des paradigmes

L'implantation des sémaphores avec compte positif ou négatif, compte qui représente les autorisations disponibles via le sémaphore, possède quelques propriétés ou invariants utilisables pour valider la concurrence.
R1 : le compte S.E d'un sémaphore S traduit l'histoire de son utilisation ; il commence avec la valeur initiale I, augmente de 1 à chaque début de la primitive V(S), diminue de 1 à chaque début de la primitive P(S).
R2 : le nombre de processus bloqués dans P(S) est égal au nombre des autorisations attendues, c'est donc 0 quand S.E >0 et sinon c'est - S.E
R3 : Les processus franchissent P(S) soit immédiatement lors de son appel, soit après être resté dans la file d'attente de S jusqu'à être réveillé par un V(S). Le nombre de franchissements est borné par le nombre d'autorisations attribuées à S, soit le nombre fourni initialement plus les autorisations fournies à chaque exécution de V(S) ; il ne peut évidemment pas dépasser le nombre d'appel de P(S) par les divers processus.

1.1. Exclusion mutuelle

On a déjà vu au paragraphe 3.8 du chapitre 8 l'exclusion mutuelle par sémaphore et sa programmation modulaire. On en donne en Ada un composant générique et un exemple d'instantiation avec la clause new.

1.2. Cohorte et lot de ressources banalisées

La cohorte peut être constituée de N serveurs, au plus, qui coopèrent pour fournir un service à leurs clients. La cohorte peut être le groupe des clients qui utilisent chacun une ou plusieurs des N ressources banalisées, allouables dynamiquement une à une.

On prend un sémaphore S qu'on initialise à N, ce qui modélise N autorisations, celles qui permettent une concurrence maximale de N. L'entrée dans la cohorte est alors contrôlée par une primitive P(S). Si plusieurs processus concurrents appellent P(S) en même temps, leurs exécutions de P(S) sont faites en séquence, une après l'autre. Le premier reçoit l'autorisation, après que l'exécution de P(S) ait décrémenté S.E et ramené sa valeur à N - 1. L'exécution suivante place S.E à N - 2. Etc... Si S.E atteint 0, c'est que N processus sont entrés dans la cohorte. La prochaine exécution de P(S) bloque bien le processus appelant. Quand un processus quitte la cohorte, il doit rendre l'autorisation et réveiller l'un des processus bloqués. Cela est fait en appelant V(S). Le dernier processus à sortir de la cohorte et à appeler V(S) redonne, par l'exécution de V(S), à S.E la valeur N.

La solution n'est pas toujours équitable. Elle l'est sur un système qui gère les files d'attente à l'ancienneté.

1.3. Passage de témoin

On modélise le passage de témoin, l'envoi d'un signal, l'envoi d'un droit d'accès, l'envoi d'un message, comme l'apport d'une autorisation qui n'est pas disponible initialement. On prend un sémaphore S qu'on initialise à 0, ce qui modélise 0 autorisations. Ce choix permet de bloquer, par P(S), le processus qui attend le signal et cette attente dure jusqu'à ce que le processus émetteur du signal fasse V(S) pour envoyer l'autorisation attendue.

L'avantage du sémaphore est qu'il mémorise une autorisation envoyée. Donc si l'émetteur fait V(S) avant que le récepteur n'ait fait P(S), l'exécution de V(S) a comme résultat S.E = 1 et l'exécution de P(S) n'est pas bloquante pour le récepteur. De même l'émetteur peut envoyer plusieurs signaux par plusieurs V(S) successifs. Ces envois seront mémorisés dans S.E et le récepteur pourra recevoir tous les signaux, un à un, un à chaque exécution de P(S).

Ici aussi, on peut construire un composant générique et l'utiliser dans tous les exemples.

Si on veut programmer un rendez-vous ou un appel procédural entre deux processus, on doit utiliser deux signaux, un par processus. Chaque processus envoie son signal avant d'attendre celui de l'autre processus. (Attention à le faire dans le bon ordre, sinon chacun attend l'autre et on a un interblocage)

Si on veut programmer un signal fugitif, non mémorisé, il faut ajouter une variable booléenne pour indiquer à l'émetteur s'il doit ou non envoyer le signal. Cette variable est mise à Vrai par le récepteur avant de faire P(S), testée et remise à Faux par l'émetteur avant de faire V(S).

Exercice : montrer que la synchronisation n'est pas correcte dans le cas suivant :
- l'émetteur et le récepteur sont cycliques,
- l'émetteur est programmé pour faire V(S) avant de remettre la variable booléenne à Faux,
- le récepteur est suffisamment rapide pour revenir attendre un nouveau signal fugitif avant que la variable booléenne n'ait été remise à Faux.

1.4. Producteurs-consommateurs

1.4.1. Contrôle de flux

La solution pour le contrôle de flux qui doit être traité pour ce paradigme combine l'allocation à chaque début de production d'une des ressources du lot banalisé que sont les N cases du tampon et l'envoi d'un signal à chaque fin de production d'un message. Il faut un

sémaphore de contrôle d'allocation de ces N cases, sémaphore initialisé à N et appelé Nvide, et un sémaphore d'envoi d'un signal, initialisé à 0 et appelé Nplein.

Le producteur se réserve une case par P(Nvide), ce qui le bloque quand il n'y en a plus d'utilisable. Puis le producteur dépose son message dans une case, ce qui prend un certain temps. Quand c'est fini, il envoie un signal au consommateur par V(Nplein).

Le consommateur attend par P(Nplein), le signal qui l'informe qu'il y a un message nouveau. Quand il y en a un, il peut franchir P(Nplein), et il recherche la case qui contient le prochain message et lit ce message, ce qui prend un certain temps. Quand c'est fini, la case est libre pour contenir un autre message et il envoie un signal de retour de case par V(Nvide).

Le schéma présente une symétrie. Le producteur produit des messages pour le consommateur. Le consommateur produit des cases vides pour le producteur.

L'étude du réseau de Petri permet de dégager deux propriétés générales de ce schéma.
P1 : un message du tampon ne peut être concurremment écrit par un producteur et lu par un consommateur. Ceci est dû à la précédence causale entre les opérations :

fin du dépôt du message m -> signal d'un message -> début du retrait du message m

fin du retrait du message m -> signal d'une case vide -> début du dépôt suivant

(la précédence causale est marquée par ->, qu'on lit précède).
P2 : le producteur et le consommateur ne peuvent pas être bloqués simultanément si l'évolution du réseau est faite dès qu'une transition est franchissable. Si le mécanisme de synchronisation assure le blocage sans interblocage, alors la solution est sans interblocage.

1.4.2. Contrôle de cohérence

Même si l'on est sûr qu'un producteur et un consommateur ne peuvent pas chronologiquement accéder au même message, il faut assure la gestion des accès au tampon, pour diverses utilisations de ce tampon.

On examine trois utilisations, de la plus simple à la plus compliquée.

a) Le cas le plus simple est celui où il y a un seul producteur et un seul consommateur (il y a donc exclusion mutuelle structurellement pendant l'accès à chaque message), où le tampon est géré à l'ancienneté comme un tableau circulaire modulo N et où les chemins d'accès au tampon sont deux index, l'un géré entièrement et seulement par le producteur, l'autre géré entièrement et seulement par le consommateur. Aucune donnée, aucun index n'est partagé : on n'a pas d'exclusion mutuelle à gérer.

b) Si, dans les mêmes conditions d'utilisation du tampon, il y a plusieurs producteurs, ils vont devoir partager l'index utilisé pour le dépôt et donc le protéger par un sémaphore d'exclusion mutuelle entre producteurs. Il faut aussi veiller à ce que les dépôts se fassent l'un après l'autre, car c'est l'hypothèse d'utilisation du tampon par le consommateur (voir les règles de précédence, indiquées plus haut) ; tous les messages déposés doivent être consécutifs dans le tampon, il ne doit pas y avoir de trou.

Si, dans les mêmes conditions d'utilisation du tampon, il y a plusieurs consommateurs, on doit introduire un sémaphore d'exclusion mutuelle entre consommateurs.

c) Si maintenant la gestion du tampon, qu'elle soit à l'ancienneté ou toute autre, fait appel à des index ou des variables partagées entre les producteurs et les consommateurs, on doit introduire un sémaphore d'exclusion mutuelle unique pour tous, et inclure dans la section critique, l'accès aux index et l'accès au tampon jusqu'à la fin du dépôt ou du retrait d'un message.

Dans les deux utilisations a) et b), le tampon est accessible en concurrence par un producteur et un consommateur simultanés. Dans l'utilisation c), un seul processus peut faire accès au tampon à un moment donné. Il y a moins de concurrence. On retrouve cette restriction dans le concept de moniteur et ses implantations en Java ou Ada.

La solution n'est pas toujours équitable. Elle l'est sur un système qui gère les files d'attente à l'ancienneté.

Ici aussi, on peut construire un composant générique, qui fournit un objet de synchronisation avec des procédures d'accès, qui gère la concurrence dans ces procédures et qui peut être utilisé par tout processus, qu'il agisse comme producteur ou comme consommateur, ou les deux successivement .

Exercice. Montrer que si on autorise la concurrence des dépôts entre plusieurs producteurs en leur donnant à chacun la valeur de la case suivante à utiliser et cela avant que la case précédente

n'ait été remplie entièrement, le consommateur risque de vouloir lire une case à moitié pleine (avec un message non entièrement écrit). Pour avoir une solution correcte, il faut attendre la fin du dépôt avant de faire un retrait et respecter les règles de précédence vues en 1.4.1. Même chose si on autorise la concurrence des retraits.

1.5. Lecteurs-rédacteurs

Pour contrôler la cohérence des accès, on compte le nombre des lecteurs concurrents. Ce compte est incrémenté à toute entrée d'un lecteur et décrémenté à toute sortie d'un lecteur. Ce compte n'est cohérent que si ces opérations sont faites en exclusion mutuelle entre lecteurs. Le premier lecteur d'un groupe de lecteurs concurrents, celui qui fait passer ce compte à 1, doit demander aussi l'exclusion mutuelle avec toute opération d'écriture par un rédacteur. Le dernier lecteur du groupe, celui qui fait passer le compte de lecteurs à 0, doit libérer la section critique avec les rédacteurs. Les rédacteurs doivent écrire en exclusion mutuelle avec les autres rédacteurs et avec le groupe de lecteurs. L'algorithme utilise un compteur NL et deux sémaphores d'exclusion mutuelle, l'un Mutex_L entre lecteurs, l'autre Mutex_A entre les accès de tout rédacteur et d'un groupe de lecteur.

Cela fonctionne de façon assez subtile :

a) Mutex_L ne bloque que les lecteurs et n'empêche pas un rédacteur de réveiller un lecteur par Mutex_A.

b) Mutex_L ne bloque jamais longtemps les lecteurs qui décrémentent NL pour indiquer la fin de leurs lectures.

c) Un rédacteur bloqué est réveillé par un autre rédacteur qui sort de section critique ou par le dernier lecteur qui finit de lire dans un groupe de lecteurs.

La solution n'est pas équitable. S'il n'y a jamais de dernier lecteur dans un groupe (il suffit pour cela que les entrées dans le groupe soient toujours supérieures aux sorties du groupe), les lecteurs se coalisent et c'est la famine pour les rédacteurs.

On donne une solution qui donne la même priorité à tous. Un sémaphore supplémentaire gère la ressource critique fictive qui est le droit de demander un nouvel accès. Toutes les demandes se font en exclusion mutuelle et dès qu'une demande est mise en attente parce qu'il y a déjà un lecteur (ou un rédacteur), toutes les autres demandes sont aussi bloquées. La solution est équitable si les files d'attente du sémaphore supplémentaire sont gérées à l'ancienneté.

Ici aussi, on peut construire un composant, qui fournit un objet de synchronisation avec des procédures d'accès, qui gère la concurrence dans ces procédures et qui peut être utilisé par tout processus, qu'il agisse comme lecteur ou comme rédacteur, ou les deux successivement .

1.6. Repas des philosophes

La solution simple consistant à considérer chaque baguette comme une ressource critique et à demander à chaque philosophe de prendre ses deux baguettes l'une après l'autre (entrer en section critique pour chacune d'elle) conduit à un interblocage quand chaque philosophe a pris une baguette et demande la seconde sans relâcher la première. Pour éviter ce cas d'interblocage, on ajoute la contrainte supplémentaire qui impose que 4 philosophes seulement puissent prendre des baguettes concurremment. Pour cela on leur demande de manger assis et on ne leur fournit que 4 chaises. Cela introduit une cohorte de 4 philosophes assis.

Cette solution est sans interblocage. En effet, comme il n'y a que 4 philosophes assis, l'un au moins peut recevoir deux baguettes au bout d'un temps fini.

Elle est équitable si l'implantation du sémaphore est telle qu'une opération V() réveille un processus bloqué en lui donnant l'autorisation qu'il demandait par l'opération P(). Il n'y a jamais plus d'un philosophe bloqué par l'un des sémaphores et celui qui est bloqué est réveillé avant que le réveilleur n'ait la possibilité de reformuler sa demande avant lui. (Attention, il y a des implantations de sémaphores, qui diffèrent de celle que nous avons choisie, qui se rapprochent d'une sorte d'attente active en étant telle qu'un V() réveille un processus pour qu'il relance l'exécution d'un P().)

Ici aussi, on peut construire un composant, qui fournit un objet de synchronisation avec des procédures d'accès paramétrées, qui gère la concurrence dans ces procédures et qui peut être utilisé par tout processus, qu'il agisse pour demander ou pour rendre les ressources.

2. Schéma des sémaphores privés

2.1. Le principe du schéma des sémaphores privés

Un sémaphore ne contient qu'un compteur, ce qui peut se montrer insuffisant pour exprimer les contraintes de contrôle d'un problème concurrent. On va introduire un ensemble de données partagées entre processus pour ce contrôle et, pour que ces données restent cohérentes, les processus les utilisent en exclusion mutuelle.

Dans notre gestion de concurrence asynchrone, chaque processus doit pouvoir se bloquer lui-même si c'est nécessaire pour le bon fonctionnement d'ensemble. Il détecte cela en utilisant les données partagées en section critique. S'il se bloque en section critique, il ne libère pas celle-ci et empêche tout autre processus d'accéder à son tour en section critique aux données communes qui lui indiqueraient qu'il y a un processus bloqué à libérer. Bloquer et réveiller en section critique conduit à interblocage.

Pour éviter cet interblocage, tout processus qui doit se bloquer doit libérer la section critique avant de se bloquer. Cette contrainte est assurée automatiquement dans les langages de haut niveau, par construction du moniteur de Hoare ou de ses avatars dans Java ou Ada. Avec les sémaphores, nous devons le programmer explicitement. C'est ce qu'on appelle le schéma des sémaphores privés (Historiquement, il a été inventé avant le moniteur et l'a inspiré).

Pour chaque processus X qui utilise les données partagées, on crée un sémaphore "privé" qui va servir à le bloquer si nécessaire (et lui seulement, d'où le vocable "privé") et à lui envoyer le signal de réveil. Pour une cohorte de N processus à contrôler, on va créer un tableau Sempriv de N sémaphores, chacun initialisé à 0. Le processus X utilisera P(Sempriv(X)) pour se bloquer, et V(Sempriv(Y)) pour envoyer un signal à un processus Y quelconque (ce peut être X lui-même, pourquoi pas ?).

Quand un processus X utilise les données partagées, en exclusion mutuelle, ce peut être :

a) Pour noter une information qui n'entraîne aucune réaction sur un processus.

b) Pour noter une information qui entraîne le réveil d'un processus Y bloqué. Le processus X doit noter les conséquences de cette décision, en particulier que Y n'est plus bloqué, et il réveille Y par l'opération V(Y) qu'il peut faire en section critique. Comme X et Y sont asynchrones, le réveil de Y se fera dans un futur non défini par rapport à la suite de X (en particulier la sortie de X de sa section critique). Le processus X peut aussi réveiller Y (par un V(Y)) après être sorti de section critique.

c) Pour noter une information qui entraîne son propre blocage. Le processus X doit noter les conséquences de cette décision, en particulier qu'il va être bloqué dès que possible. Il doit libérer la section critique avant de se bloquer, hors section critique, sur son sémaphore privé Sempriv(X). On introduit une variable OK locale à X (donc non partagée entre processus) pour indiquer que le processus X peut continuer ou sinon que le blocage a été décidé. Il vient alors deux manières d'obtenir ce blocage, une fois le processus X sorti de sa section critique contrôlée par le sémaphore Mutex :

c1) P(Mutex); if not OK then P(Sempriv(X)); end if;

c2) if OK then V(Sempriv(X)); end if; V(Mutex); P(Sempriv(X));

2.2. Repas des philosophes

Le repas des philosophes fournit un cas d'utilisation du schéma des sémaphores privés. On va programmer la solution où les deux ressources demandées sont allouées globalement ou aucune ne l'est. Les données partagées utilisées pour le contrôle sont les états des philosophes. La condition OK exprime qu'un philosophe peut manger s'il le demande et que les ressources sont toutes disponibles. Donc OK est vraie pour le philosophe X quand il demande à manger et qu'aucun de ses deux voisins ne mange. Cette condition sert à vérifier s'il faut bloquer un philosophe qui veut manger ou s'il faut réveiller un philosophe quand des baguettes sont libérées.

Cette solution est sans interblocage, mais n'est pas équitable.

Bien noter qu'après le test de la condition, il faut indiquer le changement d'état du philosophe qui mange tout de suite ou de celui qui va être réveillé, car une fois réveillé par V(Sempriv(Y)), il va être autorisé à manger sans qu'on lui demande de vérifier qu'il a bien les

baguettes. C'est normal si on les lui attribuées avant son réveil. C'est toujours ainsi que fonctionne le schéma des sémaphores privés.

Ici aussi, on peut construire un composant, qui fournit un objet de synchronisation avec des procédures d'accès paramétrées, qui gère la concurrence dans ces procédures et qui peut être utilisé par tout processus, qu'il agisse pour demander ou pour rendre les ressources.

2.3. Allocation de ressources banalisées

La gestion d'un lot de ressources banalisées fournit un deuxième exemple d'utilisation du schéma des sémaphores privés. On procède en deux étapes : recevabilité de la demande, puis allocation des ressources attribuées.

Les ressources demandées dans une requête sont attribuées toutes ou aucune. Les processus demandeurs non servis sont mis dans une file d'attente L et la valeur de leur demande est conservée dans une autre file d'attente V. Le gestionnaire connaît à tout moment le stock disponible. Les données de contrôle sont le stock et les deux files L et V. Le schéma des sémaphores privés est complété par le sémaphore d'exclusion mutuelle Mutex et par le tableau de sémaphores Sempriv.

Un processus autorisé va prendre le nombre de ressources auxquels il a droit en allant les retirer une à une en exclusion mutuelle car plusieurs processus autorisés peuvent faire ces retraits en concurrence. La disponibilité d'une ressource J donnée est indiquée dans un tableau de booléens appelé Amas. Les données du contrôle de ce retrait sont Amas et Mutex_Amas. Ce contrôle est une simple exclusion mutuelle et n'est pas un schéma.

Lors du retour des ressources, on parcourt la file L à l'ancienneté pour servir autant de processus que possible. D'autres parcours seraient possibles.

Cette allocation peut conduire à un interblocage comme on l'a vu au chapitre 6.

La solution est donnée sous la forme d'un composant, qui fournit un objet de synchronisation avec des procédures Prendre() et Rendre() paramétrées, qui gère la concurrence dans ces procédures et qui peut être utilisé par tout processus, qu'il agisse pour demander ou pour rendre les ressources.

3. Moniteur

On a vu que le concept d'exclusion mutuelle ne suffisait pas à traiter tous les paradigmes de la programmation concurrente. Il faut introduire en plus des mécanismes explicite de blocage et réveil. Hoare et Brinch Hansen ont proposé le concept de moniteur ("Monitor") avec wait() et signal(), et un type "condition". Ce type permet d'associer une file d'attente à une expression booléenne et d'avoir plusieurs conditions de blocage.

Un processus qui se bloque doit libérer le moniteur, car celui-ci est sensé assurer un exclusion mutuelle d'accès aux données qu'il contient. C'est le problème qu'on a déjà rencontré avec le schéma des sémaphores privés et qu'on retrouve partout, en Java comme en Ada.

Un processus qui a accès au moniteur peut y réveiller un autre processus par condition.signal(). Comme un seul processus peut utiliser le moniteur, l'un des deux doit attendre. Selon certains moniteurs, c'est le processus réveillé qui attend. Selon d'autres, c'est le réveilleur qui attend. On a ainsi plusieurs sémantiques possibles.

3.1 Java

En Java, le moniteur est obtenu en déclarant en exclusion mutuelle ("synchronized") certaines méthodes d'une classe. Le blocage et le réveil explicites se font avec "wait()", "notify()" et"notifyAll()". Il n'y a qu'une condition anonyme de blocage avec une seule file d'attente par objet.

Le réveil par notify ou notifyAll envoie une exception et cela conduit à utiliser la clause qui permet d‘attraper une exception : “try{...} catch(InterruptedException e){...}”.

3.2 Ada : type protégé, objet protégé

En Ada 95, on introduit un blocage et un réveil implicites qui dépendent d'expressions booléennes appelées gardes et exprimées avec les variables de l'objet protégé.

Un processus appelant une procédure ou une entrée ("entry") et qui obtient l'objet protégé, s'exécute sans préemption et en exclusion mutuelle. La fin de cette exécution soit s'accompagne d'un retour du processus vers la procédure appelante, soit place le processus en attente sur une entrée si l'instruction "requeue nom_d_entrée" est programmée. L'objet protégé est libéré pour un autre processus et toutes les gardes sont alors réévaluées. La sémantique choisie, celle du modèle de l'oeuf, conduit à servir, un à un pour respecter l'exclusion mutuelle, tous les processus déjà en attente, et dont les gardes sont devenues vraies, et cela avant de servir toute nouvelle requête.

Avec les gardes et le requeue, on peut programmer un objet protégé comme un automate et cela fournit aujourd'hui la programmation de la concurrence qui est la meilleure et la plus fiable.

Les sémaphores peuvent être simulés avec un type protégé en Ada. Dans la version indiquée, l'entier du sémaphore reste toujours positif ou nul. On pourrait avoir un entier négatif en utilisant la cause "requeue".

```
---------- SIMULATIONS DES SEMAPHORE EN ADA95 -------specification : ads----
package Semaphores is
 ----------------------------------------------semaphores are simulated by protected objects ----------
-- this package implements the type SEMAPHORE and the classical primitives P, V and E0
-- each declared semaphore is implemented by a limited private object which is a protected
object

    type semaphore is limited private;
    procedure P (S : in out semaphore);
    procedure V (S : in out semaphore);
    procedure E0 (S : in out semaphore; I : in NATURAL);

private
-------------
    protected type semaphore is
        entry V;
        entry P;
        entry E0(X : in natural);  -- X ≥ 0
    private
        E : natural ;          -- E ≥ 0 dans cette implantation
        INIT : boolean := false;
    end semaphore ;

end Semaphores ; -- fin de l'ads
-------------------------------------------------------------------------------------------adb
package body Semaphores is
---------
    protected body semaphore is

        entry V when INIT is begin E := E + 1; end V;
        entry P when INIT and E> 0 is begin E := E - 1; end P;
        entry E0(X : in natural) when not INIT is begin E := X; INIT := true; end E0;

    end semaphore;

    procedure P (S : in out semaphore) is begin S.P; end P;
    procedure V (S : in out semaphore) is begin S.V; end V;
    procedure E0 (S : in out semaphore; I : in NATURAL) is begin S.E0 (I); end E0;

end Semaphores ; -- fin de l'adb
```