PUBLICATIONS MATHÉMATIQUES ET INFORMATIQUES DE RENNES

C. KAISER

Quelques problèmes de parallélisme et leurs solutions par sémaphore

Publications des séminaires de mathématiques et informatique de Rennes, 1972, fascicule 3
« Informatique », , exp. n° 1, p. 1-23

<http://www.numdam.org/item?id=PSMIR_1972___3_A1_0>

22/11/71

ESOPE/A/019

Quelques problèmes de parallélisme et leurs solutions par sémaphore.

C. KAISER *

* ingénieur de Recherche à l'Institut de Recherche d'Informatique et d'Automatique (IRIA)

P L A N

QUELQUES PROBLEMES DE PARALLELISME ET LEURS SOLUTIONS PAR SEMAPHORES.

Préambule : Cette note a été écrite dans le but de recenser quelques problèmes de parallélisme et d'utiliser à fond un mécanisme connu, celui des sémaphores, pour en analyser les avantages et les limites.

Elle a été élaborée avec la collaboration fructueuse et critique de J. FERRIE et de J. MOSSIERE.

1. Définitions

On appelle processus séquentiel le déroulement d 'un programme séquentiel [4]. Cette notion est indispensable pour exprimer des activités parallèles et leur coopération.

La coopération des processus peut s'exprimer à l'aide de sémaphores [5, 6].

Un sémaphore S est constitué par l'association d'une variable entière E(S) et d'une file d'attente F(S) appelée file d'attente du sémaphore. La variable E(S) peut prendre des valeurs quelconques. Les éléments de la file d'attente F(S), lorsque celle-ci n'est pas vide, sont des processus.

Les opérations primitives sur les sémaphores sont :

```
P(S) : begin
       E(S) := E(S) - 1 ;
       if E(S) < 0 then le processus qui exécute cette primitive se bloque
                        et rejoint la file F(S)
       end ;
V(S) : begin
       E(S) := E(S) + 1 ;
       if E(S) ≤ 0 then un processus de F(S) est ôté de cette file et est
                        remis en activité
       end ;
```

Ces opérations primitives doivent être indivisibles, c'est à dire que si deux processus tentent d'exécuter simultanément deux de ces primitives (ou la même) les deux opérations ne sont exécutées que l'une après l'autre, dans un ordre qui est d'ailleurs indifférent.

[ ] les numéros renvoient à la bibliographie

Le choix du processus à activer dans la primitive V n'est pas imposé par la primitive. Il dépend uniquement de la politique de gestion des files d'attente qui peut varier d'une réalisation à une autre. L'utilisation des sémaphores doit être indépendante de cette politique de gestion et de toute politique de priorité sur les processus.

La définition de ces primitives a comme corollaires :

1. Si E(S) est négative, sa valeur absolue est égale au nombre de processus bloqués dans la file F(S)
2. Si E(S) est positive et si le sémaphore sert à règler l'accès à une ressource, E(S) donne le nombre de voies d'accès encore disponibles.

Par la suite, on assimilera E(S) à S. Ceci est légitime car F(S) est inconnue du programmeur.

2. Problèmes fondamentaux du parallélisme

2.1 Exclusion mutuelle [5]

Lorsque plusieurs processus ont la possibilité de modifier l'état d'une ressource commune, il faut s'assurer que cette ressource n'est modifiée ou consultée que par un processus à la fois, sinon les renseignements sur cette ressource risqueraient d'être incohérents. On appelle souvent section critique la phase du processus pendant laquelle il modifie ou consulte la ressource. Le problème consiste à s'assurer qu'un processus au plus progresse dans sa section critique. La solution, pour être générale, ne doit pas faire d'hypothèse sur les vitesses relatives des divers processus. Elle ne doit pas non plus synchroniser les processus, ni faire d'hypothèse sur les priorités relatives.

La solution est la suivante (en pseudo algol) :

sémaphore mutex ; (initial value = 1)

Processus 1	Processus n
.....	.....
P(mutex);	P(mutex);
section critique;	section critique;
V(mutex);	V(mutex);
.....	.....

2.2 Synchronisation sur un évènement

On veut synchroniser deux processus coopérant à une certaine tâche. Par exemple le processus A prépare une certaine information qui lui est nécessaire ainsi qu'à un processus B, pour continuer leur progression. Si cette information n'est pas prête, le processus B doit l'attendre. Sinon il peut l'utiliser sans avoir à appeler A. Il en résulte que A ne doit pas attendre B lorsque l'information est prête.

La solution est la suivante :

sémaphore S ; (initial value = 0)

Processus A	Processus B
Préparation de l'information;	.....
V(S);	P(S);
utilisation de l'information;	utilisation de l'information;

Si on veut que A et B s'attendent, il faut écrire :

sémaphore S1, S2 ; (initial value = 0)

Processus A	Processus B
.....	.....
V(S1);	P(S1);
P(S2);	V(S2);
.....	.....

Si l'évènement attendu s'exprime en fonction de certaines variables d'état qui peuvent être modifiées par d'autres processus, on met en jeu un sémaphore privé. On appelle ainsi un sémaphore associé à un processus particulier qui est le seul à effectuer sur lui la primitive P. Il vient alors :

sémaphore mutex, (initial value = 1), privé (initial value = 0) ;
comment : événement est une expression booléenne

Processus A	Processus B
.....	.....
P(mutex);	P(mutex);
modification des variables d'état;	if évènement then V(privé);
if évènement then V(privé);	V(mutex);
V(mutex);	P(privé);
.....	.....

2.3 Compositon d'évènements

2.3.1 Conjonction d'évènements

L'activation d'un processus C subordonnée à l'émission de signal par les processus A et B se programme :

sémaphore S;(initial value = 0)

Processus A	Processus B	Processus C
.....	.....	.....
V(S);	V(S);	P(S);
.....	.....	P(S);
		.....

2.3.2 Disjonction d'évènements

L'activation d'un processus C subordonnée à l'émission de signal par l'un des processus A ou B s'écrit :

sémaphore S;(initial value = 0)

Processus A	Processus B	Processus C
.....	.....	.....
V(S);	V(S);	P(S);
.....	.....	.....

3. Producteur-Consommateur

3.1 Problème 1 [5]

Soit deux processus cycliques appelés producteur et consommateur. A chaque cycle de leur activité, le producteur génère des informations tandis que le consommateur les exploite. Les informations produites ou consommées sont situées dans une zone de tampons accessibles aux deux processus. Toutefois ceux-ci ne peuvent travailler simultanément sur le même tampon.

Le producteur remplit des tampons, tant qu'il y en a de disponibles. Il se bloque lorsqu'il n'en reste plus. Le consommateur prélève les tampons pleins qu'il vide jusqu'à ce qu'ils soient tous vides. Dans ce cas, il attend. Le problème consiste à programmer ces attentes et à prévoir le réveil du producteur lorsqu'un tampon vient d'être vidé ou du consommateur lorsqu'un tampon vient d'être rempli.

Par hypothèse :

- les tampons sont de taille fixe
- les tampons sont consommés dans l'ordre de leur production
- les vitesses du producteur et du consommateur sont quelconques.

Soit une zone de n tampons. Le problème se résoud comme suit :

sémaphore vide,(initial value = n), plein (initial value = 0);

```
Producteur
    begin
    integer i;i:=0;
E1: phase de production
    P(vide);
    remplissage du tampon i;
    i:=i+1 modulo n;
    V(plein);
    goto E1
    end
```

```
Consommateur
    begin
    integer j;j:=0;
E2: P(plein);
    vidage du tampon j;
    j:=j+1 modulo n;
    V(vide);
    exploitation des informations;
    goto E2
    end
```

3.2 Problème 2 [10]

Soit une zone de tampons communs utilisés par des paires de processus producteurs consommateurs (P1,C1),...(Pn,Cn), Chaque couple de processus est affecté à une tâche bien particulière différente des autres. Il prend les tampons nécessaires dans la zone commune et les y restitue lorsqu'il n'en a plus besoin.

La coopération de ces processus se résoud comme suit :

```
sémaphore disponible (initial M), vide 1 (initial M1),...,
          vide n (initial Mn),
          mutex, mutex 1,..., mutex n (initial 1), plein 1,...
          plein n (initial 0);
          /* au total 2 + 3 n sémaphores */
```

Producteur i	Consommateur i
Pi : phase de production;	Ci : P(plein i);
P(vide i);	P(mutex i);
P(disponible);	vidage d'un tampon de la zone locale à (Pi,Ci);
P(mutex);	V(mutex i);
Obtention d'un tampon;	P(mutex);
V(mutex);	restitution du tampon à la
remplissage d'un tampon;	zone commune;
P(mutex i);	V(mutex);
pose du tampon dans la zone	V(disponible);
locale à (Pi,Ci);	V(vide i);
V(mutex i);	goto Ci;
V(plein i);	
goto Pi;	

Remarques

1) vide i et plein i sont utilisés comme vide et plein du problème précédent et controlent la zone locale de tampons, de taille maximum Mi.

2) disponible controle l'emploi de l'ensemble des tampons car on peut très bien avoir $\sum_i Mi > M$

3) mutex et mutex i sont là pour éviter que plus d'un processus ne manipule, à un instant donné, les files de tampons vides et de tampons pleins.

4. Lecteurs et rédacteurs (accès aux fichiers)

Il s'agit de l'étude d'une série de problèmes posés par le partage d'une ressource entre deux catégories de processus. Les "rédacteurs" doivent avoir l'accès exclusif à la ressource, tandis que les "lecteurs", peuvent partager la ressource avec un nombre illimité d'autre lecteurs.

4.1 Problème 1 [3]

Les lecteurs peuvent se partager la ressource tant que celle ci n'est pas occupée par un rédacteur. Ils ne doivent pas attendre si un rédacteur attend, ils ont le droit de se coaliser pour occuper la ressource au détriment des rédacteurs. Les lecteurs ont priorité pour utiliser la ressource dès que l'un d'eux l'occupe.

Une solution donne :

Integer NL ; (initial = 0)
sémaphore mutex, w; (initial value = 1)

lecteur	rédacteur
P(mutex);	
NL := NL + 1 ;	
if NL = 1 then P(w);	
V(mutex);	P(w);
.....	.....
lecture	écriture
.....	.....
P(mutex);	V(w);
NL := NL - 1;	
if NL = 0 then V(w);	
V(mutex);	

Remarques

1) w est un sémaphore d'exclusion mutuelle pour tous les rédacteurs. Il sert aussi au premier lecteur qui occupe la ressource et au dernier qui la libère.

2) mutex est un sémaphore d'exclusion mutuelle pour les lecteurs seuls. Il protège le compte de lecteurs, NL.

4.2 Problème 2 [3]

Dès qu'un rédacteur réclame l'accès à la ressource, elle doit lui être attribuée le plus vite possible au détriment des lecteurs nouveaux qui arrivent. Donc tout lecteur arrivé après que la ressource a été demandée par un rédacteur doit attendre, même si la ressource est encore occupée par des lecteurs. Cette fois on donne la priorité aux rédacteurs et les lecteurs peuvent attendre indéfiniment si les rédacteurs se coalisent.

solution

integer NL, NR ; (initial value = 0)

sémaphore mutex 1, mutex 2, mutex 3, w, r ; (initial value = 1) ;

Lecteur

```
P(mutex 3);
    P(r);
        P(mutex 1);
        NL := NL + 1;
        if NL = 1 then P(w);
        V(mutex 1);
    V(r);
V(mutex 3);
.........
lecture
.........
P(mutex 1);
NL := NL - 1;
if NL = o then V(w);
V(mutex 1);
```

Rédacteur

```
P(mutex 2);
NR := NR + 1;
if NR = 1 then P(r);
V(mutex 2);
P(w);
.............
écriture
.............
V(w);
P(mutex 2);
NR := NR - 1;
if NR = 0 then V(r);
V(mutex 2);
```

Remarques :

1) mutex 1 et w jouent le même rôle que mutex et w du problème 1.

2) r est utilisé par les rédacteurs pour se réserver l'accès à la ressource, tout comme les lecteurs le faisaient avec w dans le problème 1. Le premier rédacteur en actionnant P(r) bloque les lecteurs avant leur entrée dans la section critique encadrée par mutex 1. Il est important qu'il le fasse en dehors de cette section car mutex 1 encadre une autre section critique dans laquelle les lecteurs libèrent la ressource.

3) Sans mutex 3, on pourrait trouver un rédacteur et plusieurs lecteurs dans la file d'attente de r. On n'assurerait pas la priorité absolue aux rédacteurs. Mutex 3 garantit qu'un lecteur au plus utilise r. Donc la file d'attente de r ne peut plus contenir qu'un rédacteur, (pendant qu'un lecteur incrémente NL), ou qu'un lecteur.

4.3 Problème 3

Aucune catégorie n'a priorité sur l'autre. Si un lecteur a la ressource, tous les lecteurs nouveaux qui arrivent y accèdent jusqu'à ce qu'un rédacteur arrive. Dans ce cas les lecteurs nouveaux attendent, comme les rédacteurs nouveaux. Si un rédacteur a la ressource tout nouvel arrivant attend sans distinction de catégorie. Quand le rédacteur a fini, il réveille le premier processus en attente selon l'ordre, inconnu ici, des files d'attente. Si plusieurs lecteurs se suivent dans la file d'attente, on les fait accéder ensemble à la ressource.

solution

sémaphore r, w, mutex 1; (initial value = 1)

integer NL ; (initial = 0)

lecteur

```
P(r);
    P(mutex 1);
    NL := NL + 1;
    if NL = 1 then P(w);
    V(mutex 1);
V(r);
......
lecture
......
P(mutex 1);
NL := NL - 1;
if NL = 0 then V(w);
P(mutex 1);
```

rédacteur

```
P(r);
P(w);
.......
écriture
........
V(w);
V(r);
```

Remarques

1) r bloque tous les lecteurs nouveaux dès qu'un rédacteur est arrivé.

2) w bloque toute écriture tant que les lectures ne sont pas finies. Il y a au plus, un rédacteur bloqué dans la file associée a w.

3) dès qu'un lecteur passe ou est réveillé, il incrémente le compte des lecteurs, bloque éventuellement la ressource et réveille le processus suivant de la file associée à r.

4) comme l'ordre P(r), P(w) est respecté dans chaque processus, il ne peut y avoir de blocage par étreinte fatale. [9]

5) pour rendre la solution plus symétrique, le rédacteur peut faire la primitive V(r) immédiatement après P(w).

6) cette solution n'assure de service selon l'ordre d'arrivée (FIFO, "first in-first out"), que si toutes les files sont gérées selon ce principe. Par hypothèse on n'en sait rien.

4.4 Problème 4

Comme précédemment, aucune catégorie n'a priorité sur l'autre. Toutefois lorsqu'un lecteur est libéré et peut accéder à la ressource, il entraine avec lui tous les autres lecteurs bloqués. Mais tout nouveau lecteur qui arrive pendant ce passage en groupe n'a pas le droit de se joindre à eux s'il y a encore des rédacteurs en attente, et doit attendre. On enlève ainsi le risque d'attente indéfinie pour les rédacteurs.

Ce problème se pose dans l'allocation de mémoire lorsqu'on fait du regroupement pour tenir compte de la présence de traducteur réentrant [1]. Dans ce cas, on regroupe entre eux tous les usagers qui emploient le même traducteur réentrant. Toutefois, pour éviter que ceux ci ne se coalisent et ne fassent attendre indéfiniment les autres usagers, le regroupement ne se fait que lorsqu'aucun usager n'emploie le traducteur réentrant.

```
solution
sémaphore w, t, mutex 1, mutex 2 (initial = 1);
integer  NL, NR,(initial = 0), état, (initial = 1);
```

Lecteur

```
P(t); V(t);
P(mutex 1);
NL := NL + 1 ;
if NL = 1 then
    begin
    P(w);
        P(mutex 2);
        if NR > 0 then
           begin
           P(t);
           état := 2;
           end
        else état := 3;
        V(mutex 2) ;
    end;
V(mutex 1);
.........
lecture
.........
P(mutex 1);
NL := NL - 1;
if NL =0  then
    begin
    P(mutex 2);
    if NR > 0 then  V(t);
    état := 1;
    V(mutex 2);
    V(w);
    end ;
V(mutex 1);
```

Rédacteur

```
P(mutex 2);
NR := NR + 1 ;
if (NR=1) & (état=3) then
     begin
     P(t);
     état := 2;
     end ;
 V(mutex 2) ;

P(w) ;
.........
~~lecture~~ écriture
.........

P(mutex 2) ;
NR := NR - 1
V(mutex 2);
V(w);
```

Remarques

1) w et mutex 1 ont toujours le même rôle. Un lecteur au plus est bloqué par w. Dans ce cas, mutex 1 est bloquant.

2) la variable état prend 3 valeurs :
   1. aucun lecteur ne lit
   2. un lecteur, au moins, lit et un rédacteur, au moins, attend
   3. un lecteur, au moins, lit et aucun rédacteur n 'attend

La distinction entre les états 1 et 2 n'est faite que pour la clarté de l'exposé ; elle pourrait disparaître de l'algorithme.

3) le sémaphore t sert à bloquer tout rédacteur dès qu'on peut passer dans l'état 2. On y passe depuis l'état 1, ce qui est détecté chez les lecteurs, ou depuis l'état 3, ce dont les rédacteurs s'aperçoivent.

4) pour que le test sur NR > 0 soit toujours correct, il faut prendre garde à bien respecter l'ordre des opérations suivantes :
   1. les rédacteurs doivent incrémenter NR avant de faire P(w).
   2. les rédacteurs doivent décrémenter NR avant de faire V(w).
   3. les lecteurs ne doivent tester NR qu'après avoir fait P(w) et avant d'avoir fait V(w).

C'est un ordre nécessaire pour être certain que NR ne diminue pas pendant que les lecteurs occupent la ressource.

5. Feux de circulation à un carrefour.

La circulation, à un carrefour, est réglée par des feux, vert et rouge. Quand le feu est vert pour une voie, les voitures qui s'y trouvent peuvent traverser le carrefour. Les feux ne sontjamais verts pour les deux voies à la fois. De temps en temps la couleur des feux des voies change. Ce changement se fait avec précaution pour éviter qu'une voiture d'une voie ne soit carambolée par une voiture d'une autre voie.

5.1 Problème 1

Régler la circulation en faisant alterner les feux. Traiter le cas d'une file de voiture par voie et supposer que le carrefour ne peut contenir qu'une voiture à la fois.

Solution

sémaphore mutex 1, mutex 2, feu 1, (initial value = 1),
feu 2 , (initial value = 0);

```
   voie 1                    Changement                    voie 2

P(mutex 1);                   boolean a; a:=true          P(mutex 2);
   P(feu 1);                 C:wait(m):/*délai d'attente*/     P(feu 2);
   traversée du carrefour;     if a then                       traversée
   V(feu 1);                        begin                      du carrefour
V(mutex 1);                         P(feu 1);                  V(feu 2);
                                    V(feu 2);               V(mutex 2);
                                    a:=false;
                                    end
                               else
                                    begin
                                    P(feu 2);
                                    V(feu 1);
                                    a:=true
                                    end ;

                               goto C ;
```

Remarques

1) feu 1 et feu 2 règlent chaque file. Ils évitent aussi toute modification de feu tant qu'une voiture est dans le carrefour.

2) mutex 1 et mutex 2 sont là pour prévenir toute coalition de voiture voulant bloquer le feu. En conséquence il y a au plus une voiture bloquée par feu 1 ou feu 2.

5.2 Problème 2

Régler la circulation de la même façon, en faisant alterner les feux. Traiter cette fois le cas où k voitures peuvent entrer dans le carrefour. Se protéger contre toute coalition de h voitures qui occuperaient et bloqueraient le feu.

Solution

```
sémaphore mutex 1, mutex 2, (initial = k), feu1, mutex, w (initial=1),
          feu 2, (initial = 0);
integer n (initial = 0)
```

```
   voie 1                    changement                    voie 2
P(mutex 1);                  boolean a; a:=true       P(mutex 2);
   P(feu 1);                 C: wait(m);                   P(feu 2);
      P(mutex);                 if a then                     P(mutex);
      n:=n+1                       begin                    n:=n+1;
      if n=1 then P(w);            P(feu 1);                if n=1
      V(mutex);                    P(w);                    then P(w);
   V(feu 1);                       V(feu 2);                V(mutex);
.........                          V(w);                 V(feu 2);
traversée du                       a:=false              ..............
carrefour                          end                   traversée du car-
.........                     else                       refour
   P(mutex);                       begin                 ..............
   n:=n-1;                         P(feu 2);                P(mutex);
   if n=0 then V(w);               P(w);                    n:=n-1;
   V(mutex);                       V(feu 1);                if n=0 then V(w);
V(mutex 1);                        V(w);                    V(mutex);
                                   a:=true               V(mutex 2);
                                   end ;
                                goto C;
```

Remarques

1) mutex 1 et mutex 2 ne laissent passer que k voitures au plus

2) feu 1 et feu 2 servent à arrêter les voitures lorsque les feux vont changer. Le processus changement n'est jamais bloqué longtemps derrière feu 1 (ou feu 2) car s'il est bloqué aucune voiture ne peut se bloquer par P(w).

3) w sert à bloquer le changement de feu tant qu'il y a des voitures dans le carrefour. A cause des deux primitives P(feu), P(w), il n'y a que le processus changement qui puisse être bloqué par w.

5.3 Problème 3

Cette fois les feux sont controlés par radar. Ils ne changent, au bout d'un temps minimum, que s'il y a des voitures sur la voie bloquée par le feu rouge. Traiter le cas où le carrefour ne contient qu'un véhicule à la fois.

Solution

sémaphore mutex 1, mutex 2, feu 1 ; (initial = 1)

sémaphore feu 2, présence 1, présence 2; (initial = 0)

```
voie 1
V(présence 1);
P(mutex 1);
   P(feu 1);
.............
traversée du carrefour
.............
   P(présence 1) ;
   V(feu 1) ;
V(mutex 1) ;
```

```
changement
boolean a; a:= true
C : wait(m);
  if a then
     begin
     P(présence 2);
     V(présence 2);
     P(feu 1);
     V(feu 2);
     a:=false
     end
  else
     begin
     P(présence 1);
     V(présence 1);
     P(feu 2);
     V(feu 1);
     a:=true
     end ;
  goto C;
```

```
voie 2
 V(présence 2);
 P(mutex 2);
     P(feu 2);
 ...........
 traversée du car-
 refour
 .............
     V(feu 2);
 P(présence 2);
 V(mutex 2);
```

Remarques

1) feu 1, feu 2, mutex 1 et mutex 2 sont employés comme dans le problème 1

2) présence 1 et présence 2 servent à compter le nombre de voitures arrivées sur chaque voie. Aussi quand une voiture s'en va, elle fait P(présence 1) ou P(présence 2). Le feu teste aussi ce compteur et s'arrête éventuellement. Comme la primitive P dont il se sert pour cela décrémente la valeur du sémaphore, le feu rétablit l'ancienne valeur par une primitive V.

3) l'ordre des deux dernières opérations d'une voie : P(présence 1); V(mutex 1); est indifférent.

4) présence 1 et présence 2 peuvent prendre des valeurs positives quelconques, mais il n'y a au plus qu'un seul processus bloqué par l'un deux.

5) un véhicule arrêté devant le feu, (c'est à dire, par exemple, après V(présence 1) et avant P(feu 1)) ne bloque pas le changement de feu.

6. Les philosophes aux spaghettis [7]

Cinq philosophes sont réunis pour philosopher. En plus des problèmes philosophiques, un problème pratique leur est posé au moment du repas. En effet celui est composé de spaghettis qui selon le savoir vivre de ces philosophes se mangent avec deux fourchettes. Or la table n'est dressée qu'avec une fourchette par couvert et aucun philosophe ne veut transgresser les règles du savoir vivre. Après quelques instants de réflexion, les philosophes décident d'adopter le rituel suivant :

1. Tous les cinq philosophes s'assoient à table, ce qui règle le problème de politesse entre eux.
2. A tout instant chaque philosophe se trouve dans l'un des état suivants : ou bien il pense, ou bien il mange.
3. Tout philosophe qui mange utilise sa fourchette et celle de son voisin de droite. Il ne peut pas en emprunter d'autre (ce serait contraire au savoir vivre). Deux philosophes voisins ne peuvent donc manger en même temps.
4. Tout philosophe qui pense a la politesse de n'utiliser aucune fourchette.
5. Initialement, tous les philosophes pensent.

Le comportement de chaque philosophe se réduit à une succession d'intervalles quelconques de réflexion et de ripaille.

Le rituel choisi satisfait les règles de savoir vivre de ces philosophes, mais il doit être complêté car si tous les philosophes décident de manger et saisissent au même moment leur propre fourchette, la docte assemblée se trouve bloquée en étreinte fatale. De même un philosophe pourrait attendre indéfiniment si lorsqu'il a décidé de manger, il n'était pas averti de la libération des fourchettes dont il a besoin.

Le problème consiste à compléter le rituel choisi en faisant les hypothèses suivantes :

1. les activités des philosophes penser, manger sont strictement séquentielles et n'ont d'interaction avec les activités des autres philosophes qu'au début et à la fin de leur exécution.

2. ces actions se déroulent à des vitesses quelconques, non nulles.

3. le comportement de chaque convive peut être assimilé à un processus cyclique.

solution [7]

l'addition et la soustraction sont exécutées modulo 5 dans ce qui suit.

```
integer array C [0:4] ; (initial values = 0)
sémaphore array sempriv [0:4] ; (initial values = 0)
sémaphore mutex ; (initial = 1)
procédure test(k); value k ; integer k ;
   if (C [k] = 1) & (C [k + 1] ≠ 2) & (C [k - 1] ≠ 2 then
                  begin
                  C [k] := 2 ;
                  V(sempriv [k]) ;
                  end ;
```

```
Philosophe i
Li : begin
   pense ;
       P(mutex) ;
       C [i] := 1 ;
       test(i) ;
       V(mutex) ;
       P(sempriv [i] ) ;
   mange ;
       P(mutex) ;
       C [i]:= 0 ;
       test(i-1) ;
       test(i+1) ;
       V(mutex) ;
   goto Li ;
    end ;
```

les autres philosophes utilisent le même rituel.

7. Conclusion

7.1 Sur les sémaphores

Les solutions proposées utilisent les sémaphores, à l'exclusion de tout autre mécanisme, pour gérer

- d'une part les conflits dûs au parallélisme, grâce à l'instauration d'un ordre là où cela est nécessaire à la cohérence du système et là seulement,
- d'autre part, les phénomènes d'attente et l'allocation des ressources, selon un ordre imposé par les objectifs du système.

Le mécanisme des sémaphores est bien adapté pour controler la synchronisation d'actions, mais non pour répartir les ressources.

Ainsi les problèmes de lecteurs et rédacteurs (§4) sont bien mieux résolus en séparant nettement les deux fonctions de synchronisation et d'allocation. Dans ce cas, la solution se présente comme celle du problème des philosophes et s'écrit :

```
P(mutex) ;
demande d'allocation de ressource (lire ou écrire) ;
if action possible then V(sempriv) ;
V(mutex) ;
P(sempriv) ;
action utilisant la ressource allouée ;
P(mutex) ;
fin d'allocation ;
if action possible pour d'autres then V(sempriv autre) ;
V(mutex) ;
```

Remarques

1) chaque philosophe i utilise un sémaphore privé sempriv [i] derrière lequel il se bloque éventuellement

2) on associe 3 états à chaque philosophe i :

C [i] = 0 lorsqu'il pense

C [i] = 1 lorsqu'il voudrait bien manger, mais ne le peut par manque de fourchette

C [i] = 2 lorsqu'il mange

Le passage de C [i] de 1 à 2 n'est possible que si (C [i+1] ≠2)&.C [i-1] ≠2). C'est ce qu'on vérifie dans la procédure test.
Le passage de C [i] de 2 à 0 libère deux fourchettes. Si les voisins immédiats de i sont dans l'état 1, il faut alors les réveiller. C'est pourquoi on appelle test(i-1) et test(i+1).

3) mutex protège le tableau C.

4) Ce rituel n'interdit pas à certains philosophes de se coaliser au détriment d'un autre : par exemple i-1 et i+1 peuvent empêcher indéfiniment i de manger.

5) cette solution oblige les philosophes à gérer eux-même la pénurie de fourchettes.

7.2 Sur la validité des solutions

Dans cette note, on s'est contenté de donner des problèmes et certaines solutions supposées exactes. L'élaboration des solutions et la vérification de leur validité se sont faites empiriquement en utilisant une série d'essais et de test. Pour être sûr des solutions, il faut avoir fait tous les tests possibles, ou un sous ensemble de ceux-ci si on sait trouver des relations d'équivalence entre tests. Dès que les problèmes deviennent un peu complexes, le nombre de tests à faire rend la méthode inutilisable.

D'autres démarches sont nécessaires. Elles relèvent encore du domaine de la recherche [7,8] . La première consiste à prouver, par une analyse de l'algorithme que la solution est correcte. C'est une preuve à postériori, une fois l'algorithme trouvé. Une démarche plus élaborée serait de partir de la définition du problème et de construire correctement, avec preuves à l'appui, l'algorithme qui répond au problème La découverte de l'algorithme se ferait alors par étape, chacune d'elle étant correcte. Ceci permettrait d'éliminer plus rapidement les fausses solutions.

BIBLIOGRAPHIE

[1] BETOURNE, BOULENGER, FERRIE, KAISER, KOTT, KRAKOWIAK, MOSSIERE. Process management and resource sharing in the multiaccess system ESOPE CACM 13, 12 (dec. 70)

[2] BETOURNE, BOULENGER, FERRIE, KAISER, KRAKOWIAK, MOSSIERE. Présentation du système ESOPE congrès AFCET (sept. 70)

[3] COURTOIS,P.J,HEYMANS,F. and PARNAS, D.L. Concurrent control with readers and writers CACM 14,10 (oct. 71)

[4] DENNIS,J.B. and VAN HORN,E.C. Programming semantics for multiprogrammed Computations CACM 9,3 (March 1966)

[5] DIJKSTRA,E.W. Cooperating Sequential Processes, Programming Languages (F. Genuys, ed.) Academic Press (1968)

[6] DIJKSTRA,E.W. The structure of THE multiprogramming System CACM 11,5 May 1968)

[7] DIJKSTRA,E.W. Hierarchical ordering of sequential processes. EWD 310, (1971)

[8] FLOYD,R.W. Assigning meanings to programs. Proc. Symposia in applied Mathematics. vol. American Mathematical Society (1967)

[9] HABERMANN,A.N. Prevention of system deadlocks CACM 12,7 (july 69)

[10] RIDDLE,W.E and SAAL,H.S Communicating semaphores CGRM 117, Stanford 1970 (note interne)