UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 3

Synchronisation

Gabriel Girard[1]

Sherbrooke

31 janvier 2023

[1] Gabriel.Girard@usherooke.ca

Table des matières

3 Synchronisation **5**
3.1 Exclusion mutuelle et section critique . . . 13
3.2 Solutions (algorithmes) avec attente active (spinlock) . . . 14
3.2.1 Algorithme # 1 . . . 14
3.2.2 Algorithme #2 . . . 16
3.2.3 Algorithme #3 . . . 16
3.2.4 Algorithme #4 . . . 17
3.2.5 Algorithme #5 . . . 19
3.2.6 Algorithme #6 : algorithme de Dekker . . . 19
3.2.7 Algorithme #7 : algorithme de Dijkstra . . . 22
3.2.8 Algorithme #8 : algorithme de Eisenberg et McGuire . . . 24
3.2.9 Algorithmes #9, #10 et #11 : les algorithmes de Lamport . . . 27
3.2.10 Algorithme #12 : algorithme de Peterson (deux processus) . . . 29
3.2.11 Algorithme #13 : algorithme de Peterson (N processus) . . . 31
3.2.12 Algorithme avec attente active utilisant des instructions machines . . . 33
3.2.13 Problématiques dues aux algorithmes d'attente active . . . 33
3.3 Sémaphore . . . 36
3.3.1 Aspects critiques pour l'implantation . . . 38
3.3.2 Utilité des sémaphores . . . 39
3.3.3 Avantages et inconvénients des sémaphores . . . 41
3.3.4 Sémaphores dans la littérature, les systèmes et les langages . . . 43
3.4 Exemples classiques . . . 43
3.4.1 Problème du tampon fini (producteurs/consommateurs) . . . 43
3.4.2 Problème des lecteurs/écrivains . . . 44
3.4.3 Problème des philosophes . . . 47
3.5 Conclusion . . . 48

Appendices **53**

Chapitre 3

Synchronisation

Ce chapitre est inspiré de [2, 3, 4].

Au chapitre précédent, nous avons présenté la condition nécessaire pour exécuter des processus en parallèle : l'obligation qu'ils soient disjoints. Malheureusement (mais aussi heureusement) cela n'est pas toujours possible. Dans bien des cas, les processus partagent des ressources (coopèrent et communiquent volontairement ou non). Être disjoints constitue donc un attribut pas très commode vu l'objectif, développer un programme parallèle afin d'accomplir plus rapidement une certaine tâche. Pour que plusieurs processus puissent coopérer sur une tâche commune, ils doivent généralement communiquer ou se partager de l'information.

Le comportement anormal que nous avons fait ressortir dans les exemples du chapitre précédent est dû aux **interférences non contrôlées**. Il est possible d'éliminer ce comportement anormal en empêchant le chevauchement d'exécutions d'énoncés (ou groupe d'énoncés) non-disjoints dans le temps. Il suffit de contrôler l'ordonnancement des événements ou les interférences. On appelle cet ordonnancement, la synchronisation.

Définition : Synchronisation

La synchronisation est un terme général qui s'applique à toutes les contraintes sur l'ordonnancement des opérations dans le temps.

Elle permet de contrôler les interférences de deux façons :

1. par synchronisation conditionnelle :

   Dans ce cas, la synchronisation sert à remettre à plus tard l'exécution d'un processus jusqu'à ce qu'une certaine condition soit vérifiée ou jusqu'à ce qu'un événement se produise.

   Quelques exemples de synchronisation conditionnelle (hors informatique) :

   - Une personne attend son autobus. L'arrivée de l'autobus est l'événement qui met fin à l'attente.
   - Une personne attend son repas au restaurant. L'arrivée du repas est l'événement qui met fin à l'attente et démarre les actions suivantes (manger !).

   Des exemples en informatique :

- Un processus attend la fin d'une E/S (écriture sur disque, impression, arrivée d'un message sur le réseau...).
- Un processus attend qu'un autre processus ait produit une information pour poursuivre son exécution.

2. par exclusion mutuelle :

   Dans ce second cas, la synchronisation sert à rendre l'exécution d'un bloc d'opérations indivisible (atomique).

   Quelques exemples de situation hors informatique exigeant une forme d'exclusion mutuelle :

   - Une personne attend pour charger sa voiture électrique que le chargeur soit disponible, ce dernier ne pouvant être partagé (idem pour une pompe à essence pour faire le plein d'une voiture à combustion).
   - Un personne attend que le téléphone public d'une cabine téléphonique soit disponible pour l'utiliser, celui-ci étant généralement une ressource qui ne se partage pas.

   Nous présentons plus loin dans ce chapitre plusieurs exemples d'exclusion mutuelle reliés au domaine de l'informatique.

La synchronisation permet donc à des processus non disjoints de s'exécuter concurremment et de produire des résultats valides. Il devient alors possible de partager des ressources (une variable est une ressource). Ce partage est nécessaire afin que les processus coopèrent et communiquent. En effet, la coopération inter-processus implique une certaine forme de communication.

Il est important de ne pas confondre synchronisation et communication. Cette dernière implique un échange d'information qui permet à l'exécution d'un processus d'influencer celle d'un autre processus. Elle se fait par des variables communes ou par messages.

Définition : Communication

La communication implique un échange d'information qui permet à l'exécution d'un processus d'influencer l'exécution d'un autre processus.

Attention :

La synchronisation n'implique pas nécessairement la communication.

Pour le moment, nous présentons la synchronisation sur des variables partagées (nos ressources communes). Celles-ci peuvent servir (ou non) à établir une communication directe ou indirecte. Nous abordons la communication au chapitre 5.

Exemple 1 : Partage d'une variable `x`

Le programme 3.1 reprend notre exemple de la section **??** dans lequel nous avions deux processus partageant une variable `x`. Pour que ce programme produise toujours le résultat attendu, les exécutions des deux énoncés ne doivent pas se chevaucher. Ils doivent donc se synchroniser afin de s'exécuter en exclusion mutuelle.

```
1 cobegin
2    P1: x := x + 1;
3    P2: x := x + 2;
4 coend;
```

Programme 3.1 – Deux énoncés modifiant la variable x en parallèle.

Exemple 2 : Les producteurs/consommateurs

Un autre exemple de partage fréquemment exploité dans un système est celui dit «de type producteur/consommateur».

Dans ce type de relation, un processus producteur produit de l'information (appelons cela un message) qui sera ensuite traitée par un processus consommateur. Par exemple :

- un pilote d'imprimante produit des lignes qui sont consommées par l'imprimante ;
- un compilateur produit du code d'assemblage qui sera consommé par l'assembleur. À son tour, l'assembleur produira du code objet éventuellement consommé par le chargeur ;
- un transfert de fichier dans lequel le producteur lit des enregistrements et produit les blocs de données Ces derniers sont repris par le consommateur et écrit dans le nouveau fichier ;
- deux processus se servant du concept de pipe («|») de Unix pour effectuer une tâche. Exemples : «`ps -ax | grep firefox`» ou «`ls | grep tp1`»

Pour permettre l'exécution parallèle des processus impliqués dans ce type de relation, ceux-ci doivent impérativement se synchroniser.

Une implantation de producteur/consommateur nécessite l'usage d'un ensemble de tampons. Les tampons sont remplis par le producteur et vidés par le consommateur. Ces derniers doivent toutefois se synchroniser de façon à ce que :

- les consommateurs ne tentent pas de traiter un tampon pour lequel aucune information (message) n'a été produite (**synchronisation conditionnelle)** ;
- les producteurs ne tentent pas de produire un message dans un tampon contenant déjà de l'information. i.e. information qui n'a pas encore été consommée (**synchronisation conditionnelle**) ;

  Ce dernier cas ne se produit pas si l'ensemble contient un nombre illimité de tampons. Cette approche n'est généralement pas recommandée car elle apporte avec elle certains risques (remplir la mémoire). Il est d'usage de limiter le nombre de tampons à un certain seuil, disons `N` (facile à implanter via une liste circulaire). Lorsqu'un nombre maximum de tampons est imposé, il est nécessaire de vérifier s'ils ne sont pas tous utilisés avant de produire de l'information.
- les producteurs et les consommateurs ne doivent pas accéder simultanément à la structure de données dans le but de la modifier (**exclusion mutuelle**).

Le programme 3.2 présente une première implantation d'une relation producteur/consommateur. L'ensemble de tampons est construit comme une liste circulaire bâtie sur un tableau primitif. Cette solution exige de la synchronisation conditionnelle (lignes 13 et 20) mais ne requiert aucune exclusion mutuelle puisqu'aucun processus n'accède le même élément de la structure en même temps. Étant donné le programme utilise un tableau primitif, l'accès aux tampons se fait à l'aide des indices `in` et `out`. La synchronisation conditionnelle est alors simple. La figure 3.1 présente le cas $in = out$ qui signifie que le tampon est vide. La figure 3.2 présente une séquence pendant laquelle le producteur

```
1  type item = ...
2  var  tampon: array [0..n-1] of item;
3       in, out : 0..n;
4       nextp, nextc : item;
5  in := 0; out := 0;
6  parbegin
7  //  Producteur
8     while (true)
9     begin
10         ...
11          produire un item
12          ...
13         while((in + 1) % n = out); // on attend un tampon vide
14         tampon[in] := nextp;
15         in := (in + 1) % n;
16    end
17 //  Consommateur
18    while (true)
19    begin
20         while(in = out) ;          // un attend qu'un tampon soit produit
21         nextc := tampon[out]
22         out := (out + 1) % n;
23         ...
24         traiter un item
25         ...
26    end
27 parend
```

Programme 3.2 – Producteur/consomateur : version 1.

produit un élément que le consommateur consomme. La figure 3.3 illustre, quant à elle, une séquence de productions menant au remplissage complet des tampons ($(in + 1)\%n = out$).

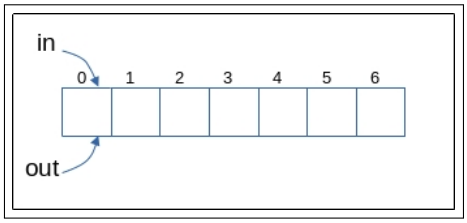


Figure 3.1 – Tableau vide

Le programme 3.3 propose une seconde implantation se servant cette fois d'une liste chaînée. Le producteur n'effectue aucune synchronisation conditionnelle car la liste a la possibilité de croître à l'infinie. Le consommateur, quant à lui, doit attendre qu'un élément soit produit pour le consommer (synchronisation conditionnelle ligne 24).

Il est important de noter que ce programme ne fournit aucune exclusion mutuelle. Dans ce cas particulier, il y a risque de corruption dans le cas où le producteur et le consommateur accèdent simultanément à la liste.

Comme nous pouvons le constater, les étapes numérotées de 1 à 8 au programme 3.3 représentent la séquence d'exécution qui cause le problème. Les figures 3.4 a à 3.4 h illustrent cette séquence et la corruption qui s'en suit. La figure 3.4 a décrit la liste originale. La figure 3.4 b présente la liste après que le consommateur ait débuté son accès à l'étape 1 (ligne 26). Toutefois, avant de terminer, il se fait interrompre et le contrôle passe au producteur. Aux figures 3.4 c à 3.4 e, celui-ci exécute

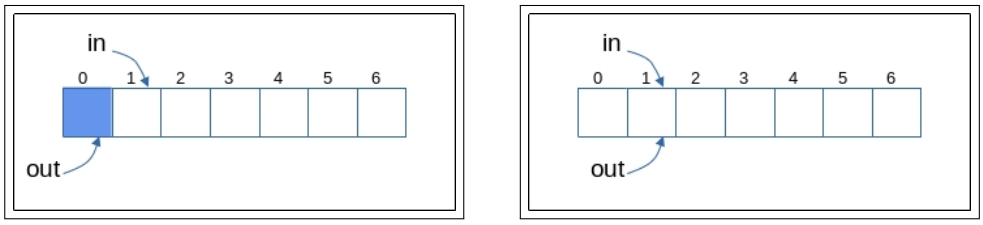


Figure 3.2 – Séquence «production ;consommation» pour retrouver un tableau vide.

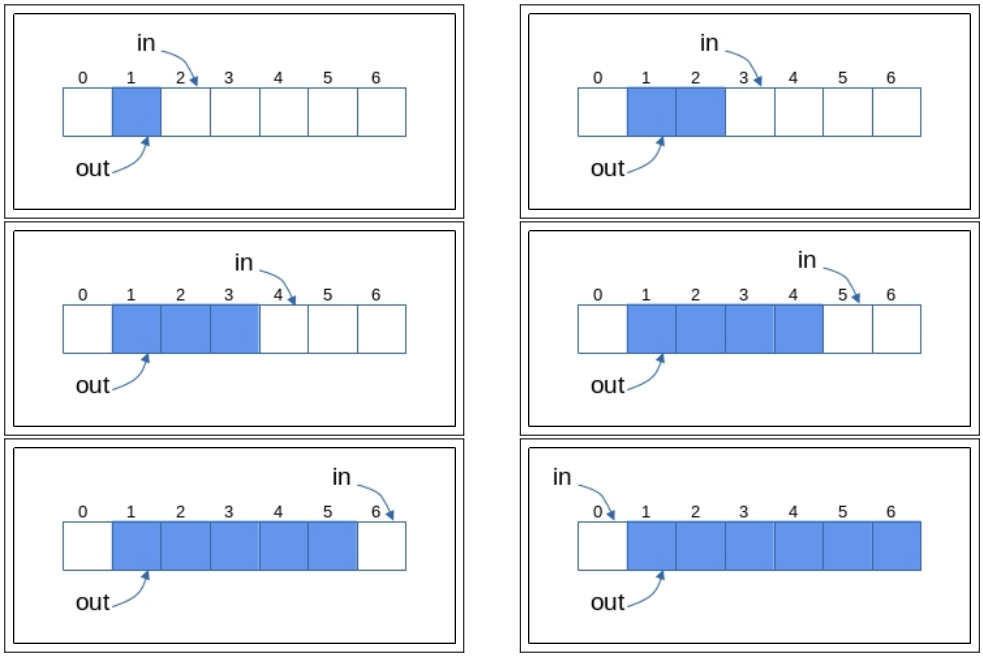


Figure 3.3 – Séquence de production remplissant le tableau

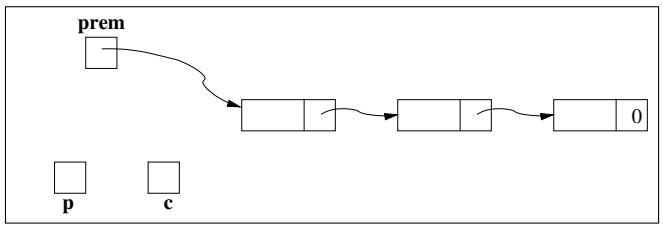
```
1  type item = ...
2  type tampon: record
3              element : item;
4              suiv : pointer to tampon;
5       end;
6  var prem, p, c: pointer to tampon;
7  nextp, nextc : item;
8  prem := nil;
9  parbegin
10 //      Producteur
11    while (true)
12    begin
13         ...
14         produire item nextp
15         ...
16         new(p);              // 2
17         p.elem := nextp;     // 3
18         p.suiv := prem;      // 4
19         prem := p;           // 5
20    end
21 //       Consommateur
22    while (true)
23    begin
24       while (prem=nil);     // On attend qu'un élément soit produit
25       c := prem;            // 1
26       prem:=prem.suiv;      // 6
27       nextc := c.elem;      // 7
28       dispose(c);           // 8
29       ...
30       traiter item nextc
31       ...
32    end
33 parend
```

Programme 3.3 – Producteur/consomateur : version 2.

les étapes 2 à 5 sans se faire interrompre (ligne 17 à 20). Finalement, aux figures 3.4 f à 3.4 h, le consommateur reprend le contrôle et termine ses étapes 6 à 8 (lignes 27 à 29). Notons que suite à cette séquence,deux événements indésirables se sont produits :

- L'élément ajouté à la liste par le producteur est perdu ;
- Le pointeur `prem` pointe vers un tampon qui a été détruit, la liste est donc perdue.

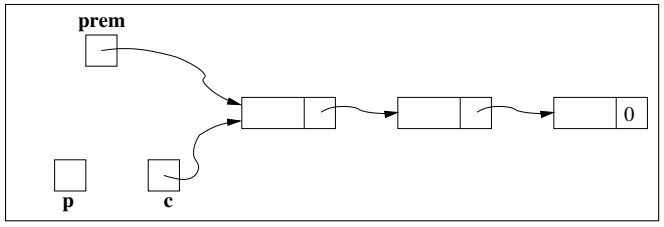
La figure 3.4 i présente la liste espérée lorsqu'une exécution se déroule correctement.



a Liste chaînée originale

Comsommateur (1)

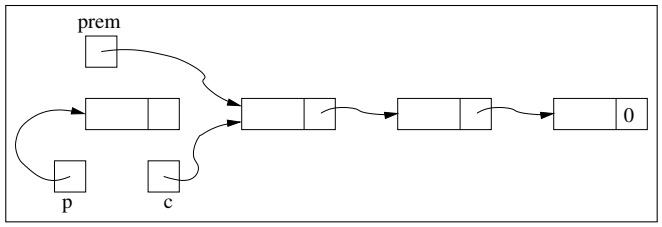
```
c:=prem;
```



b Liste après exécution de l'étape 1 du consommateur

Producteur (2-3)

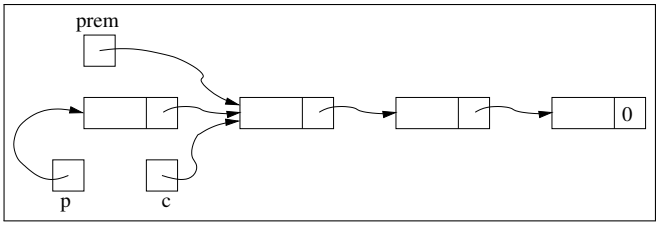
```
new(p);
p.elem:=nextp;
```



c Liste après exécution des étapes 2-3 du producteur

Producteur (4)

```
p.suiv:=prem;
```

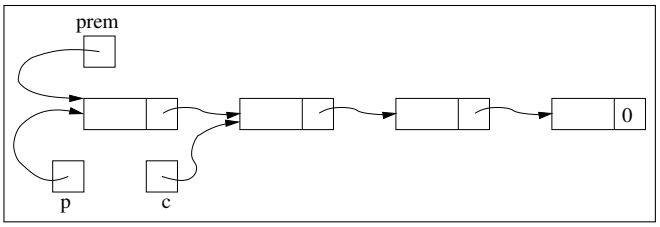


d Liste après exécution de l'étape 4 du producteur

Cette corruption est le fait d'une manipulation simultanée sans aucun contrôle par le producteur et le consommateur. Pour remédier à ce problème, il faut regrouper, dans chacun des processus

Producteur (5)

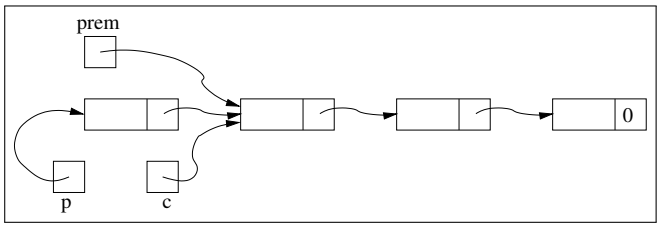
```
prem := p;
```



e Liste après exécution de l'étape 5 du producteur

Consommateur (6-7)

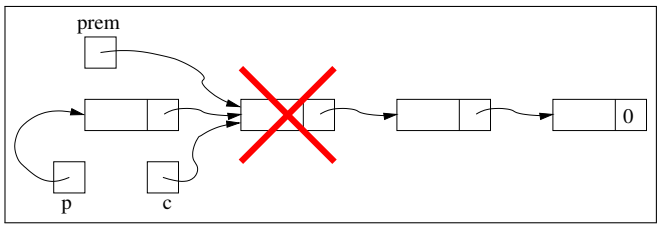
```
p.suiv:=prem;
nextc:=c.elem;
```



f Liste après exécution des étapes 6-7 par le consommateur

Consommateur (8)

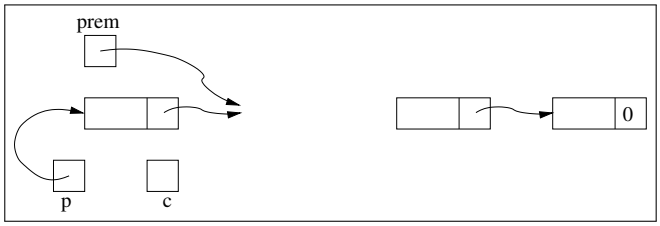
```
dispose(c)
```



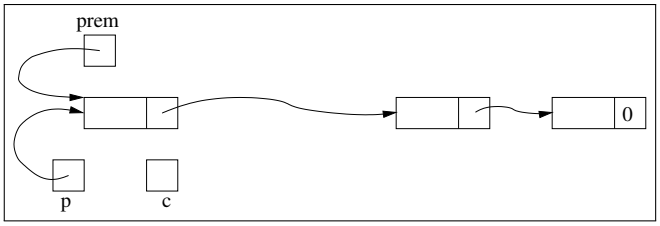
g Liste après exécution de l'étape 8 par le consommateur

Consommateur (-)

```
...
```



h Liste après la fin de l'exécution



i Liste espérée après une exécution correcte

Figure 3.4 – Comportement du programme producteur/consommateur

(producteur et consommateur), les énoncés qui manipulent la liste et, là, s'assurer qu'ils s'exécutent en exclusion mutuelle.

Considérant toujours la même situation, quels sont donc les énoncés qu'on se doit de regrouper ?

1. producteur : lignes 17-20 et consommateur : lignes 26-29 ?
2. producteur : lignes 18-20 et consommateur : lignes 26-28 ?
3. producteur : lignes 19-20 et consommateur : lignes 26-27 ?

Notons aussi qu'il y a une autre raison pour laquelle ce problème survient. Elle réside dans l'utilisation d'une liste LIFO (pile). Qu'en serait-il avec une liste de type FIFO ?

3.1 Exclusion mutuelle et section critique

Toutes les séquences d'instructions devant être regroupées et exécutées comme une seule opération (de façon atomique) sont appelées des sections critiques. Toutes exécutions d'une même section critique doivent l'être en exclusion mutuelle afin d'éviter les problèmes présentés précédemment.

Définition : Section critique

Une **section critique (SC)** est définie comme une séquence d'instructions qui doit être exécutée en exclusion mutuelle (de façon atomique).

La difficulté avec les sections critiques est justement d'assurer l'exclusion mutuelle. Pour y parvenir, on ajoute deux protocoles à exécuter par tous les processus, l'un à l'entrée, l'autre à la sortie, pour chacune des sections critiques.

La programme 3.4 illustre cette structure. Ainsi chaque processus vérifie, grâce au protocole d'entrée, s'il peut exécuter la section critique. De même, lorsqu'il termine la section critique, il exécute le protocole de sortie pour indiquer que la section critique est de nouveau disponible.

```
1  process Pi(i=1..n)
2     loop
3        ... section non-critique ...
4
5        protocole d'entrée
6        section critique (SC)
7        protocole de sortie
8
9        ...
10    endloop
```

Programme 3.4 – Structure d'une section critique avec les protocoles d'entrée et de sortie.

Pour être acceptables, les différents protocoles d'entrée et de sortie doivent respecter les contraintes suivantes :

1. aucune supposition sur le matériel (sauf atomicité des instructions) ;

   Il ne faut faire aucune supposition concernant les instructions matérielles ou sur le nombre de processeurs que la machine supporte. On suppose toutefois que les instructions de base

de la machine (les instructions de type `load`, `store`, `move`, `add`, `sub`, ...) sont atomiques. Cela signifie que si deux instructions sont exécutées simultanément (sur deux processeurs), le résultat sera le même que si elles l'avaient été de façon séquentielle.

Ne faire aucune supposition sur les instructions machines implique des solutions indépendantes du matériel.

2. aucune supposition sur les vitesses d'exécution relatives des processus ;

   Ainsi, une solution utilisant des instructions de type «`sleep`» pour ralentir l'exécution d'un processus est inacceptable.

3. un processus qui n'est pas lui-même en section critique (SC) ne doit pas pouvoir empêcher les autres processus d'entrer dans leur SC ;
4. il ne faut pas remettre indéfiniment à plus tard l'admission d'un processus en section critique (quand il y a plusieurs candidats possibles).

Les contraintes 3 et 4 offrent une protection contre les phénomènes d'interblocage et de famine. Cela signifie que, s'il n'y a aucun processus dans une section critique particulière et qu'un unique processus veut y accéder, alors ce dernier doit éventuellement réussir à y être admis.

3.2 Solutions (algorithmes) avec attente active (spinlock)

Dans cette section nous présentons les premiers algorithmes qui ont permis d'assurer l'exclusion mutuelle. Ceux-ci étaient basés sur l'attente active. Selon cette approche, un processus boucle sur une condition fausse (ou vraie) jusqu'à ce qu'elle devienne vraie (ou fausse). On emploie l'expression **attente active** car les processus testent une condition de façon répétitive sans aucune pause.

Cependant, assurer l'exclusion mutuelle avec une telle approche tout en respectant nos quatre contraintes n'est pas vraiment chose facile. Le problème général étant très complexe, nous commençons par développer des solutions qui tentent de synchroniser seulement deux processus. La figure 3.5 décrit ce cas simplifié. Dans cet exemple, les processus P_1 et P_2 ont exactement la même structure, chacun incluant un protocole d'entrée et un protocole de sortie.

```
1 process Pi(i=1..n)
2    loop
3       ... section non-critique ...
4       protocole d'entrée
5       section critique (SC)
6       protocole de sortie
7       ...
8    endloop
```

Programme 3.5 – Structure d'une section critique avec deux processus et leurs protocoles.

3.2.1 Algorithme # 1

Pour le premier algorithme, visons une approche simple. Soit une variable booléenne appelée `libre` qui indique si un processus peut entrer en section critique. Chaque processus exécute le cycle suivant :

1. il réserve la ressource (`libre = faux`) *(Protocole d'entrée)* ;
2. il exécute la section critique ;

3. il libère la section critique (`libre = vrai`) *(Protocole de sortie)*;

Avant de réserver la ressource, un processus attend que celle-ci soit disponible (boucle jusqu'à ce que `libre = vrai`). Le programme 3.6 implante de cet algorithme.

```
1  var  libre: boolean;
2  begin
3    libre := vrai;
4    parbegin
5     P1: repeat
6            while not libre;
7            libre := faux;
8            section critique
9            libre := vrai;
10           section non-critique
11       forever;
12    P2: repeat
13           while not libre;
14           libre := faux;
15           section critique
16           libre := vrai;
17           section non-critique
18       forever;
19   parend
20 end.
```

Programme 3.6 – Algorihme #1

Ce programme est-il fonctionnel ?

Suivons son exécution pour constater ce qu'il pourrait s'y produire. Il en résulte la séquence d'événements suivants :

1. Initialement, `libre = vrai`.
   À ce moment, les deux processus ont la possibilité d'y référer simultanément et trouver `libre` à `vrai`.
2. Comme `libre` est à `vrai`, les deux processus positionnent simultanément `libre` à `faux` et entrent en section critique.

La figure 3.5 expose cette séquence et met en évidence que cet algorithme ne satisfait pas la condition d'exclusion mutuelle sur la section critique (i.e. accès exclusif à un seul processus à la fois).

P_1	P_2	`libre`
_____	_____	vrai
`while not libre`	_____	vrai
_____	`while not libre`	vrai
_____	`libre = faux`	faux
_____	... section critique	faux
`libre = faux`	_____	faux
... section critique	_____	faux
... section critique	... section critique	faux

Figure 3.5 – Séquence menant au non respect de l'exclusion mutuelle

3.2.2 Algorithme #2

Pour ce second essai, nous introduisons une variable entière «`tour`». Si `tour = i`, alors c'est au tour du processus i d'entrer en section critique. Le programme 3.7 implante cette solution. Celle-ci assure qu'un seul processus à la fois entre en section critique. En effet, car dans le cas contraire, cela signifierait que la variable «`tour`» contient deux valeurs distinctes simultanément, ce qui est impossible.

```
1  var tour: integer;
2  begin
3    tour := 1;  (ou 2)
4    parbegin
5      P1 : repeat
6               while tour = 2 do /* rien */ ;
7               section critique
8               tour := 2;
9               section non-critique
10           forever;
11     P2 : repeat
12              while tour = 1 do /* rien */;
13              section critique
14              tour := 1;
15              section non-critique
16           forever;
17   parend
18 end.
```

Programme 3.7 – Algorihme #2

Cependant, cet algorithme implique une alternance stricte des processus qui exécutent la section critique. Un processus i peut attendre pour la ressource tandis que, celui à qui c'est le tour ne s'en prévaut pas (ou n'a même pas l'intention de s'en prévaloir avant longtemps, sinon jamais). Le processus le plus lent impose donc son rythme à l'autre. Si le processus lent veut accéder à la ressource une seule fois par jour et l'autre une fois par heure, alors là il y a un réel problème ! De plus, imaginer si un processus disparaît, l'autre risque d'attendre vraiment longtemps.

Le bon point en faveur de cet algorithme, c'est d'assurer l'exclusion mutuelle et d'éviter tout interblocage. Il y a cependant le risque de famine. Cela va donc à l'encontre d'une de nos contraintes. L'algorithme #2 n'est donc pas acceptable.

3.2.3 Algorithme #3

Le problème des algorithmes précédents est de ne pas considérer individuellement l'état de chaque processus. Comme ils utilisent une unique variable, ils ne se souviennent que de l'état du processus autorisé à entrer en section critique. Il serait intéressant de connaître l'état de tous les processus voulant entrer en section critique. Pour y parvenir, remplaçons la variable entière (`tour`) et la variable booléenne (`libre`) par deux variables booléennes, `enSC1` et `enSC2`. Ces variables représentent l'état des processus (s'ils sont en section critique ou non) et sont initialisées à «`faux`». Si la variable d'un processus est à «`vraie`» alors celui-ci est en section critique. Le programme 3.8 implante cette solution.

```
1  var  enSC1, enSC2: boolean;
2  begin
3    enSC1 := enSC2 := faux;
4    parbegin
5      P1 : repeat
6               while enSC2 do /*rien*/;
7               enSC1  := vrai;
8               section critique
9               enSC1 := false;
10              section non-critique
11           forever;
12     P2 : repeat
13             while enSC1 do /*rien*/;
14             enSC2 := vrai;
15             section critique
16             enSC2 := faux;
17             section non-critique
18           forever;
19   parend
20 end.
```

Programme 3.8 – Algorihme #3

Cette nouvelle approche ne garantit toutefois pas la présence d'un seul processus à la fois en section critique. La séquence d'exécution de la figure 3.6 illustre une violation de l'exclusion mutuelle. Cet algorithme fonctionne donc (ou non) selon les vitesses relatives d'exécution des deux processus.

P_1	P_2	enSC1	enSC1
_____	_____	faux	faux
`while enSC2 do;`	_____	faux	faux
_____	`while enSC1 do;`	faux	faux
_____	`enSC2 := vrai`	faux	vrai
_____	... section critique	faux	vrai
`enSC1 := vrai`	_____	vrai	vrai
... section critique	_____	vrai	vrai
... section critique	... section critique	vrai	vrai

Figure 3.6 – Séquence menant au non respect de l'exclusion mutuelle

3.2.4 Algorithme #4

L'échec de l'algorithme #3 est dû au fait que chacun des processus (P1 ou P2) prend une décision sur l'état de l'autre avant même d'avoir l'opportunité de changer son propre état. Corrigeons cette situation en déplaçant l'énoncé d'affectation tel que le propose le programme 3.9. Ainsi,

«`enSC1:=vrai`» indique que le processus P1 veut entrer en section critique, et ce avant qu'il ne vérifie l'état de P2 (`enSC2`).

```
1  var  enSC1, enSC2: boolean;
2  begin
3    enSC1 := enSC2 := faux;
4    parbegin
5       P1 : repeat
6                enSC1  := vrai;
7                while enSC2 do /*rien*/;
8                section critique
9                enSC1 := false;
10               section non-critique
11           forever;
12       P2 : repeat
13               enSC2 := vrai;
14               while enSC1 do /*rien*/;
15               section critique
16               enSC2 := faux;
17               section non-critique
18           forever;
19   parend
20 end.
```

Programme 3.9 – Algorihme #4

Cette solution assure effectivement l'exclusion mutuelle, i.e. la présence d'un seul processus à la fois en section critique. Cependant la figure 3.7 met en évidence une séquence hasardeuse car elle mène à un interblocage sous forme de boucle infinie, i.e. chaque processus vérifie indéfiniment l'état de l'autre processus. L'algorithme #4 est donc inacceptable car il autorise des séquences d'exécutions entraînant une forme d'interblocage, appelée «**livelock**».

P_1	P_2	enSC1	enSC1
_____	_____	faux	faux
`enSC1 := vrai`	_____	vrai	faux
_____	`enSC2 := vrai`	vrai	vrai
_____	`while enSC1 do;`	vrai	vrai
_____	... `while enSC1 do;`	vrai	vrai
`while enSC2 do;`	_____	vrai	vrai
... `while enSC2 do;`	_____	vrai	vrai
... `while enSC2 do;`	... `while enSC1 do;`	vrai	vrai

Figure 3.7 – Séquence menant au non respect de l'exclusion mutuelle

Définition : Interblocage

Situation dans laquelle au moins deux processus sont incapables de poursuivre leur exécution car chacun attend que les autres produisent quelque chose. Les processus sont généralement inactifs (en attente) et l'utilisation de l'UCT demeure faible.

Définition : Famine

Situation dans laquelle la décision de poursuivre l'exécution d'un processus est continuellement remise à plus tard.

Définition : Livelock

Situation dans laquelle au moins deux processus changent continuellement leur état en réponse à des changements d'état dans les autres processus, et ce, sans faire aucun travail utile. Les processus poursuivent leur exécution mais ne sont pas productifs (et consomment beaucoup d'UCT). Un livelock [a] ressemble à une famine et à un interblocage en même temps.

a. Malheureusement je n'ai pas trouvé de terme français satisfaisant. Certains traduisent livelock par famine, ce qui est inexact.

3.2.5 Algorithme #5

Dans l'algorithme #4, chaque processus modifie son état sans connaître l'état de l'autre et ne le change plus jamais par la suite. Corrigeons cette situation en obligeant l'un d'eux à renoncer temporairement à son intention d'entrer en section critique, lorsque tous deux souhaitent y entrer en même temps, et ce afin de permettre l'accès à l'autre. Le programme 3.10 présente cette solution.

Cet algorithme est **quasi** correct. Il assure bien l'exclusion mutuelle mais la possibilité de boucle infinie demeure réelle (les chances sont excessivement faibles mais c'est possible). La séquence d'exécution fautive est indiquée à la figure 3.8. Celle-ci survient lorsque les processus s'exécutent simultanément et à la même vitesse sur deux processeurs distincts.

Même si cette erreur est peu probable, celle-ci fait en sorte que nos exigences ne sont pas respectées.

3.2.6 Algorithme #6 : algorithme de Dekker

Au fur et à mesure que nous progressons vers une solution fonctionnelle (du moins on l'espère), celle-ci semble se complexifier.

Heureusement, l'algorithme de Dekker est correct [1]. Pour y parvenir, son créateur (Dekker – on s'en doute !!!) a combiné les approches des algorithmes 2 et 5.

L'algorithme de Dekker est basé sur la solution précédente mais élimine les risques de boucles infinies en donnant la priorité à l'un des deux processus (seulement dans le cas d'une condition de course lors de l'entrée, i.e. lors de tests simultanés).

À la fin de l'exécution de la section critique, le processus doit libérer cette dernière et transférer la priorité à l'autre processus. Le programme 3.11 implante cette solution.

1. Seulement sur un ordinateur dont la mémoire respecte la cohérence séquentielle (que nous verrons plus tard). Cela signifie dans les faits qu'il ne fonctionne plus sur la plupart des ordinateurs modernes.

```
1  var enSC1, enSC2: boolean;
2  enSC1 := enSC2 := faux;
3  parbegin
4  P1: while(1)
5      {
6         enSC1 := vrai;
7         while(enSC2)
8         {
9            enSC1 := faux;
10           while (enSC2) /*rien*/;
11           enSC1  := vrai;
12        }
13        ...section critique
14        enSC1 := false;
15        ... section non-critique ...
16    }
17 P2: while(1)
18    {
19        enSC2 := vrai;
20        while(enSC1)
21        {
22           enSC2 := faux;
23           while (enSC1) /*rien*/;
24           enSC2  := vrai;
25        }
26        ... section critique
27        enSC2 := false;
28        ... section non-critique...
29   }
30 parend
```

Programme 3.10 – Algorihme #5

P_1	P_2	enSC1	enSC1
_____	_____	faux	faux
enSC1 := vrai	enSC2 := vrai	vrai	vrai
while enSC2	while enSC1	vrai	vrai
enSC1 := faux	enSC2 := faux	faux	faux
while enSC2	while enSC1	faux	faux
enSC1 := vrai	enSC2 := vrai	vrai	vrai
while enSC2	while enSC1	vrai	vrai
enSC1 := faux	enSC2 := faux	faux	faux
while enSC2	while enSC1	faux	faux
enSC1 := vrai	enSC2 := vrai	vrai	vrai
...	...	vrai	vrai

Figure 3.8 – Séquence menant au non respect de l'exclusion mutuelle

```
1  var enSC1, enSC2: boolean;  tour : integer;
2  enSC1 := enSC2 := faux; tour := 1;
3  parbegin
4  P1:  while(1)
5       {
6          enSC1:=vrai;
7          while (enSC2) if tour=2
8                      {
9                           enSC1 := faux;
10                          while (tour=2);
11                          enSC1  := vrai;
12                     }
13         ... section critique
14         enSC1 := false;  tour := 2;
15         ... section non-critique...
16      }
17
18 P2:  while(1)
19      {
20         enSC2  := vrai;
21         while (enSC1) if tour=1
22                     {
23                          enSC2:=faux;
24                          while (tour=1);
25                          enSC2 :=vrai;
26                     }
27         ...section critique
28         enSC2 := false; tour := 1;
29         ... section non-critique...
30      }
31 parend
```

Programme 3.11 – Algorithme de Dekker

L'algorithme de Dekker satisfait donc nos critères, et il est possible de le prouver. On doit montrer que l'exclusion mutuelle est assurée et qu'aucun interblocage ne peut se produire.

1. L'algorithme assure l'exclusion mutuelle ;

   Un processus P_i entre en section critique seulement si `enSC`j `= faux` ($j \neq i$). Comme seul P_i peut mettre à jour `enSC`i et qu'il inspecte `enSC`j seulement lorsque `enSC`i `= vrai`, alors l'exclusion mutuelle est assurée.

2. L'algorithme ne génère aucun interblocage ou livelock ;

   Il est important de noter que la variable `tour` n'est modifiée que par le protocole de sortie. Supposons que le processus P_i veut entrer en section critique. Deux situations sont possibles :

   (a) P_j ne veut pas entrer en section critique (`enSC`j `= faux`) ;
   P_i trouvera `enSC`j `= faux` ($j \neq i$) et passera directement en section critique et ce, peu importe la valeur de `tour`.

   (b) P_j veut entrer en section critique (`enSC`j `= vrai`) et `tour = i` ;
   Deux situations sont envisageables :

      i. P_j trouve `enSC`i `= faux` (P_j est plus rapide que P_i) et passe en section critique et cela même si `tour = i`.

      ii. P_j trouve `enSC`i `= vrai` alors P_i entrera en section critique avant P_j car, soit P_i a été plus rapide, soit tous deux sont arrivés en même temps dans la boucle interne et, qu'alors `tour`, étant égal à i, a favorisé P_i et a forcé P_j à remettre `enSC`i à faux.

Notons que si tour avait été égal à j, l'inverse se serait produit.

L'algorithme de Dekker assure donc bien l'exclusion mutuelle et ne génère aucune interblocage. Il satisfait donc toutes nos contraintes.

3.2.7 Algorithme #7 : algorithme de Dijkstra

La solution de Dekker résout le problème de l'exclusion mutuelle dans le seul cas où deux processus sont en jeu. La première solution fonctionnelle pour le cas de N processus fut proposée par Dijkstra. Ce dernier a généralisé la solution de Dekker pour produire l'algorithme décrit par le programme 3.12.

```
1  begin /* programme */
2  var   enCS: array[0..N-1] of (idle, want-in, in-cs);
3        tour : 0..N-1;
4    parbegin
5         P(1); P(2); P(3); P(4); ... P(N-1);
6    parend;
7  end. /* programme */
8
9  Procedure P(i : integer)
10    var   j : integer;
11 begin /* procedure */
12   while (1)
13   {
14      do
15      {
16         enCS[i] := want-in;
17         while (tour!=i)
18         {
19             if (enCS[tour]=idle) tour := i;
20         }
21         enCS[i] := in-cs;  j:= 0;
22         while (j< N) and (j=i or enCS[j] != in-cs)
23               j:= j+1;
24
25      } while (j<N);
26
27      ...section critique
28
29      enCS[i]:=idle;
30
31      ...section non-critique
32   }
33 end;  /* procedure */
```

Programme 3.12 – Algorithme de Dijkstra

La preuve du bon fonctionnement de cet algorithme est similaire à celle faite pour l'algorithme de Dekker. Pour se faire, il faut prouver que la solution garantie bien l'exclusion mutuelle et qu'elle ne provoque pas d'interblocage (ou livelock).

1. L'algorithme assure l'exclusion mutuelle ;

   Cette affirmation est vraie car P_i entre en section critique seulement lorsque pour tous les processus P_j, `enSC[j]` $\neq$ `in-cs` $\forall j \neq i$. Comme seul le processus P_i peut modifier la valeur

Edsger Wybe Dijkstra : «Prix Turing 1972»

Edsger Wybe Dijkstra (tiré de [10, 11]) est un des pionniers du domaine de l'informatique. Il est reconnu pour ses multiples contributions dans les domaines de la construction de compilateurs, des systèmes d'exploitation, des systèmes distribués, de la programmation séquentielle et concurrente, de la méthodologie et des concepts de programmation, de la recherche sur les langages de programmation, de la conception et du développement de programmes, de la vérification de programmes, des principes du génie logiciel, des algorithmes sur les graphes et les fondements philosophiques de l'informatique. Il est particuliement reconnu pour avoir introduit le terme «programmation structurée», pour ses travaux en parallélisme (exclusion mutuelle, sémaphore, interblocage), sur la théorie des graphes (algorithme du plus court chemin) et sur la tolérance aux fautes (autostabilisation).
Il s'est d'abord fait connaître grâce au système d'exploitation THE, un système construit en couches d'abstraction successives. Suite à cette expérience, il formalise le concept de sémaphore puis introduit le concept de « section critique » avec deux exemples devenus classiques : «le problème des lecteurs/écrivains» et «le dîner des philosophes».
Il rédige en 1968, un article qu'il nomme « A Case against the GOTO Statement » (rebaptisé « Go To Statement Considered Harmful »). Suite à cette parution, l'instruction `goto` est rapidement marginalisée (presque éliminée) et remplacé par la programmation structurée (concept de Wirth et Dijkstra). En programmation structurée, le `goto` est remplacé par des instructions telles «`if ... then ... else ...`, `while ... do`, `repeat ... until`» qui furent introduites par Wirth dans Algol W : chaque instruction contient une seule entrée et une seule sortie, ce qui rend enfin possible des tests systématiques exhaustifs, impossibles avec le « code spaghetti ».
Dijkstra a joué un rôle important dans le développement du langage Algol à la fin des années 1950 et a grandement contribué à la compréhension de la structure, de la représentation et de l'implémentation des langages de programmation.
Il est également à l'origine de l'algorithme éponyme, l'algorithme de Dijkstra, permettant de calculer le plus court chemin dans un graphe orienté.
En 1974, Dijkstra publie l'article fondateur de l'autostabilisation, une propriété d'un système réparti qui lui permet de retrouver automatiquement un comportement correct après toute défaillance transitoire.
Il a reçu le prix Turing en 1972. Le discours qu'il prononce en cette occasion, The Humble Programmer, est célèbre. Juste avant sa mort, en 2002, il remporte le prix PoDC de l'article le plus influent, pour ses travaux sur l'autostabilisation. L'année suivant sa mort, ce prix sera renommé en son honneur, il devient le prix Dijkstra.
Quelques citations célèbres de Dijkstra :

- « La programmation par objets est une idée exceptionnellement mauvaise qui ne pouvait naître qu'en Californie. »
- « Le plus court chemin d'un graphe n'est jamais celui que l'on croit, il peut surgir de nulle part, et la plupart du temps, il n'existe pas. »
- « Le test de programmes peut être une façon très efficace de montrer la présence de bugs mais est désespérément inadéquat pour prouver leur absence »

de `enSC[i]` et qu'il inspecte tous les `enSC[j]` seulement lorsque `enSC[i] = in-cs`, alors l'exclusion mutuelle est garantie.

2. L'algorithme ne génère aucun interblocage ou livelock ;

   Pour prouver cette affirmation, observons le comportement de l'algorithme :

   (a) Lorsque P_i exécute le protocole d'entrée, (`enSC[i]` $\neq$ `idle`) ;

   (b) (`enSC[i]` $\neq$ `in-cs`) n'implique pas que `tour = i`. Plusieurs processus peuvent vérifier l'état de `enSC[tour]` simultanément et trouver `enSC[tour] = idle`. Ils exécuteront tous `tour:=i` et passeront à l'état `in-cs`.

   Cependant, une fois qu'un processus P_k arrive à cette étape du protocole d'entrée, `enSC(k)` $\neq$ `idle`. Une fois la variable `tour` modifiée, aucun autre processus, n'ayant pas encore testé l'état de `inCS[tour]`, ne pourra exécuter `tour := i`. À partir de ce moment, plus aucun processus ne passera à l'état `enSC = in-cs`

   (c) Soit $\{P_1, P_2, ..., P_m\}$, l'ensemble des processus à l'état «`in-cs`» (`enSC[i] = in-cs`) et soit `tour = k` ou $1 \leq k \leq m$. Tous ces processus sortiront de la seconde boucle (ligne 22) avec `j< n` (car il existe un $j \neq i$ tel que `enCS[j] = in-cs`) et retourneront au début de l'énoncé (ligne 16). Ils remettront alors leur état à `want-in`. À partir de ce moment, tous les processus (sauf P_k) bloqueront à la première itération (ligne 17 : `while (tour!=i)`, car `tour!=i` et `enCS[k]= want-in` .

   Le processus P_k trouvera alors tous les `enCS[j]` $\neq$ `in-cs` $\forall j \neq k$, et entrera en section critique.

Ainsi, cet algorithme respecte toutes les contraintes déjà énoncées pour être qualifié «acceptable». Cependant, il n'assure pas l'équité entre les processus. Il est effectivement de l'ordre du possible pour un processus, de rester bloqué indéfiniment en attente d'entrer en section critique pendant que d'autres processus le précéderont toujours. Le risque de famine est donc réel, d'où la nécessité d'ajouter une contrainte additionnelle. Celle-ci s'énonce comme suit :

5. Un nombre fini de processus doit être autorisé à entrer en section critique entre le moment où un processus quelconque fait une demande d'entrée et celui où cette entrée est autorisée.

3.2.8 Algorithme #8 : algorithme de Eisenberg et McGuire

Le premier algorithme satisfaisant les cinq contraintes imposées fut développé par Knuth en 1966. Son algorithme assure qu'il n'y aura pas plus de 2^n processus qui s'insèreront entre une demande et son autorisation. DeBrujin a amélioré cet algorithme pour réduire l'attente à n^2. Finalement Eisenberg et McGuire, en 1972, ont proposé l'algorithme 3.13 qui réduit ce temps à $n - 1$. Cet algorithme est une variante de celui de Dijkstra. Il utilise la même structure et les mêmes données. Le programme 3.13 ne reproduit donc que la boucle pour un processus P_i.

La preuve du bon fonctionnement de cet algorithme est similaire à celle faite pour l'algorithme de Dijkstra. Encore là, il faut prouver que la solution garantie bien l'exclusion mutuelle, qu'elle ne provoque pas d'interblocage (ou livelock) et qu'elle assure l'équité (que chaque processus entre en section critique en moins de N-1 tours).

1. L'algorithme assure l'exclusion mutuelle ;

Niklaus Wirth : «Prix Turing 1984»

Niklaus Emil Wirth (tiré de [13, 14]) est l'inventeur de plusieurs langages de programmation. Il a introduit plusieurs innovations reprises dans les langages modernes et a notamment travaillé au développement des langages de programmation Euler (1965), ALGOL-60, Algol 68, ALGOL-W (1966), PL360 (un assembleur de haut niveau, en 1966), Pascal (le plus connu, en 1970), le PCode (dont les principes sont repris dans Matlab et Java), Modula-1 (1975), Modula-2 (1978), Oberon (1987) et Oberon-07 (1987). Il a aussi contribué au développement de systèmes d'exploitation.
Son livre intitulé «*Program Development by Stepwise Refinement*» est considéré comme un classique dans le domaine du développement de logiciel. Un second livre, «*Algorithms + Data Structures = Programs*», a aussi connu un grand succès.
Il a reçu de multiples prix dont le prix Turing (1984), le prix Max Petitpierre (1990) et le prix d'excellence du SIGPLAN de l'ACM (2007).
En 1995, Niklaus Wirth a popularisé un adage, qui a ensuite porté le nom de la loi de Wirth, indiquant que « les programmes ralentissent plus vite que le matériel n'accélère » (parfois appelé Obésiciel ou infobésité) [12].

```
1  Procedure P(i : integer)
2    var  j : integer;
3  begin /* procedure */
4    repeat
5    {
6        do
7        {
8            enSC[i] := want-in;
9            j := tour;
10           while (j!=i) if enSC[j]!=idle
11               then j:=tour;
12               else j:=j+1 mod N;
13           enSC[i] := in-cs;  j:= 0;
14           while (j<N) and (j = i or enSC[j]!=in-cs) do
15                          j:= j+1;
16       } while  (j<N) or (tour=i or enSC(tour)=idle);
17       tour := i;
18
19       ...section critique ...
20
21       j:=tour+1 mod N;
22       while ((j!=tour) and (enSC[j]=idle)) j:=j+1 mod N;
23       tour := j;  enSC[i] := idle;
24
25       ... section non-critique ...
26
27   } forever;
28 end;  /* procedure */
```

Programme 3.13 – Algorithme de Eisenberg et McGuire

La preuve de l'exclusion mutuelle est la même que celle de l'algorithme de Dijkstra. Nous ne la répéterons pas ici.

2. L'algorithme ne génère aucun interblocage ou livelock ;

   Pour garantir ce fait, on observe que la valeur de `tour` ne peut être modifiée que lorsqu'un processus entre (après en avoir obtenu l'accès) ou quitte la section critique. Donc, si aucun processus n'est en section critique ou la quitte, la valeur de `tour` reste constante. Avec cet algorithme, le premier processus en attente dans l'ordre (tour, tour + 1, ..., n-1, 0, ..., tour-1) entrera en section critique.

3. L'algorithme assure l'équité

   Lorsqu'un processus quitte la section critique, il désigne comme son unique successeur le premier processus en attente dans l'ordre cyclique. Si un processus veut entrer en section critique, il le fera donc en moins de N-1 tours.

Donald Knuth : «Prix Turing 1974»

Donald Ervin Knuth (Tirée de [18, 19]) est un des pionners de l'informatique ayant apporté de nombreuses contributions dans le domaine de l'algorithmique et dans plusieurs branches de l'informatique théorique. Il est l'auteur de multiples publications (articles et livres), notamment ses manuels intitulés «The Art of Computer Programming (TAOCP)» devenus une référence du domaine.
Parmi ses contributions, mentionnons un algorithme de recherche de sous-chaîne (Knuth-Morris-Pratt), un algorithme de parcours en profondeur, les analyseurs de grammaires formelles LR(k), la méthode des attributs sémantiques en compilation et le concept de programmation lettrée (literate programming). Il est aussi connu pour avoir créer deux logiciels libres, TeX et Metafont, ainsi que la police Computer Modern, police par défaut de TeX.
Knuth a reçu de nombreux prix dont le prix Grace Murray Hopper de l'Association for Computing Machinery (ACM) en 1971, le prix Turing (1974), la National Medal of Science (États-Unis) (1979), la médaille John von Neumann de l'IEEE (1995), le prix de Kyoto (1996) et la médaille Franklin (1988). Il est élu membre associé de l'Académie des sciences française en 1992 et membre de la Royal Society en 2003.
Knuth est aussi connu pour son humour. Ainsi, il offre une prime de 2,56$ pour chaque faute typographique ou erreur découverte dans ses livres, expliquant que « 256 cents font un dollar hexadécimal ». Les numéros de version de TeX convergent vers π, c'est-à-dire que les versions se suivent de la sorte : « 3 », « 3,1 », « 3,14 », etc., les numéros de version de Metafont convergent, eux, vers le nombre « e ». Il a également mis en garde ainsi les personnes utilisant un de ses logiciels : « Faites attention aux bogues dans ce code ; je n'ai fait que démontrer qu'il était correct, je ne l'ai pas essayé». Knuth a cessé de recourir au courrier électronique, disant qu'il s'en était servi entre 1975 et le 1er janvier 1990, et que cela suffisait pour toute une vie. Il trouve plus efficace de tenir une correspondance en « mode batch » et y consacrer une journée tous les six mois, en répondant par courrier « classique».

3.2.9 Algorithmes #9, #10 et #11 : les algorithmes de Lamport

Les programmes 3.14 et 3.15 présentent l'algorithme de «La boulangerie», développé par Leslie Lamport pour être appliqué en général dans un environnement distribué.

Leslie Lamport : «Prix Turing 2013»

Leslie Lamport (tiré de [8, 9]) est un chercheur en informatique américain, spécialiste de l'algorithmique répartie. Il est reconnu pour avoir imposé une vision claire et cohérente sur le comportement des systèmes distribués (qui semblait auparavant chaotique).
Ses contributions sont nombreuses dont :

- l'algorithme de la boulangerie qui apporte une solution « remarquablement intuitive et naturelle »[8] au problème d'exclusion mutuelle formulé par Dijkstra ;
- la relation « arrivé-avant » et les horloges logiques (qui portent son nom) ;
- la notion de cohérence séquentielle pour la mémoire partagée ;
- le problème des généraux byzantins (tolérance aux fautes) ;
- l'algorithme «Chandy-Lamport» qui permet d'obtenir des états globaux cohérents ;
- l'outil LaTeX (basé lui-même sur TeX, de Donald Knuth), un système de mise en page de documents particulièrement populaire parmi les scientifiques de nombreuses disciplines.
- les signatures de Lamport, un prototype de signature digitale (cryptographie) ;
- la logique temporelle des actions, pour spécifier et raisonner sur des systèmes distribués.

Leslie Lamport a reçu le prix Dijkstra à trois reprises en 2000, 2005 et 2014 et la médaille John von Neumann en 2008. Le 18 mars 2014, il reçoit le prix Turing pour «ses contributions fondamentales théoriques et appliquées dans les systèmes distribués et concurrents, notamment en inventant des concepts tels que la causalité, les horloges logiques et la cohérence séquentielle».

L'algorithme de «La boulangerie» est basé sur la méthode usuelle pour ordonnancer l'arrivée des clients (premier arrivé-premier servi), soit celle pratiquée dans les commerces et autres types de salle d'attente. Ainsi, à l'entrée, chaque client reçoit un numéro. Le client avec le plus petit numéro sera le prochain servi.

Cela fonctionne bien dans les commerces car les numéros sont uniques. Toutefois dans les ordinateurs, il n'y a aucune garantie que deux processus ne recevront pas le même numéro. Lorsque cette situation se produit, le processus avec le plus petit identificateur est servi en premier. Cela signifie que si P_i et P_j reçoivent le même numéro, alors P_i est servi en premier si $i < j$. Comme ces identificateurs sont uniques (sur un système centralisé évidemment) et complètement ordonnés, l'algorithme fonctionne. Sur un système distribué, les identificateurs ne sont pas nécessairement uniques d'un système à l'autre. Dans ce cas particulier, l'adresse de la machine (qui elle est unique) sert à établir la priorité.

Le programme 3.14 implante la première version proposée par Lamport pour deux processus. Le programme 3.15 présente une seconde version du même algorithme pour deux processus, appelée «Dutch Beer»

Le programme 3.16 implante la solution pour N processus encore une fois proposée par Lamport. L'algorithme adopte les notations suivantes :

- $(a, b) < (c, d)$ si $a < c$ ou si $a = c$ et $b < d$.
- $max(a_0, a_1, ...a_n)$ est un nombre k tel que $\forall i : k \geq a_i$.

```
1  var  c1, c2, n1, n2: boolean;
2  begin
3    c1 := c2 := n1 := n2 := 0;
4    parbegin
5    P1 : repeat
6            c1 := 1; n1 := n2 + 1; c1 := 0;
7            while c2!=0 do /* rien */;
8            while (n2 != 0) and (n2<n1) do /*rien*/;
9            .... section critique
10           n1 := 0;
11           .... section non-critique
12         forever;
13   P2 : repeat
14           c2:=1; n2 := n1 + 1; c2 :=0;
15           while c1!=0  do /*rien*/;
16           while (n1!=0) and (n1 <= n2) do /*rien*/;
17           .... section critique
18           n2 := 0;
19           .... section non-critique
20         forever;
21   parend
22 end.
23 end;  /* procedure */
```

Programme 3.14 – Algorithme de la Boulangerie pour deux processus (version 1)

```
1  var n1, n2:  boolean;
2  begin
3    n1 := n2 := 0;
4    parbegin
5    P1 :  repeat
6             n1 := 1; n1 := n2 + 1;
7             while (n2!=0) and (n2<n1) do /*rien*/;
8          .... section critique
9             n1 := 0;
10         .... section non-critique
11         forever;
12   P2 :  repeat
13            n2 := 1; n2 := n1 + 1;
14            while (n1!=0) and (n1<=n2)  do /*rien*/;
15         .... section critique
16            n2 := 0;
17         .... section non-critique
18         forever;
19   parend
20 end. /* procedure */
```

Programme 3.15 – Algorithme de la Boulangerie pour deux processus (version 2)

```
1  var  choosing : array[0..n-1] of boolean;
2       number   : array[0..n-1] of integer;
3  begin
4    choosing[0..n-1]:=faux;
5    number[0..n-1] := 0;
6    process Pi(1..n)
7    { repeat
8      {  choosing[i] := vrai;
9         number[i] := max(number[0], ..., number[n-1])+1;
10        choosing[i] := faux;
11        for (j:=0 to n-1)
12        { while choosing[j] do /*rien*/;
13          while (number[j]!=0) and
14                ((number[j],j)<(number[i],i)) do /*rien*/;
15        }
16   .... section critique
17        number[i] := 0;
18   .... section non-critique
19     } forever;
20   }
21 end
```

Programme 3.16 – Algorithme de la Boulangerie pour N processus

Pour prouver le bon fonctionnement de cet algorithme, il faut établir que si P_i est en section critique et que $P_k (k \neq i)$ a déjà choisi son numéro (`number[k]` $\neq$ `0`), alors `(number[i], i) < (number[k], k)`.

À partir de ce résultat, on démontre que l'exclusion mutuelle est assurée. Pour cela, considérons que P_i est en section critique et que P_k essaie d'y entrer. Lorsque P_k exécute la seconde `while` (ligne 13), pour $j = i$, il trouve :

1. `number[j]` $\neq 0$
2. `(number[j], j) < (number[k], k)`

P_k restera donc inactif tant que P_i ne sortira pas de la section critique.

Il subsiste toutefois une situation potentiellement nuisible au bon fonctionnement de cet algorithme (ainsi qu'à tous les autres algorithmes déjà présentés). Supposons un mauvais fonctionnement du matériel ou une mauvaise conception du matériel qui permet une lecture et une écriture simultanées sur une variable N_i. Dans ce cas, la lecture de la variable N_i pourrait retourner n'importe quelle valeur. L'écriture quant à elle fonctionnera toujours correctement. Les lectures qui ne chevauchent pas l'écriture, retourneront toujours la bonne valeur.

Exercice : Dans cette situation, montrez comment les deux algorithmes de Lamport (pour deux processus) échoueront.

3.2.10 Algorithme #12 : algorithme de Peterson (deux processus)

Force est de constater que la plupart des algorithmes présentés sont relativement complexes. Peterson a proposé un nouvel algorithme beaucoup plus simple. Le programme 3.17 décrit la solution de Peterson pour deux processus.

La preuve du bon fonctionnement consiste toujours à montrer que l'exclusion mutuelle est garantie, qu'il n'y a pas d'interblocage (livelock) et que l'équité est assurée.

1. L'algorithme assure l'exclusion mutuelle ;

```
1  Program Peterson()
2  begin /* programme */
3  var enSC : array[0..1] of boolean;
4      tour : 0..1;
5      enSC[0] := enSC[1] := faux;
6      tour := ?;  /*une valeur entre 0 et 1 */
7     parbegin
8        P(0); P(1);
9     parend;
10 end. /*programme*/
11
12 Procedure P(i : integer); /* i=0 ou 1 et j=i+1 mod 2*/
13     var j : integer;
14 begin /* procedure */
15    j := i + 1 mod 2;
16    repeat
17       enSC[i] := vrai;
18       tour := j;
19       while (enSC[j] and tour = j) /*rien*/;
20
21       .... section critique
22
23       enSC[i] := faux;
24
25       .... section non-critique
26
27    forever;
28 end; /*procedure*/
```

Programme 3.17 – Algorithme de Peterson (2 processus)

Supposons que P_0 et P_1 sont tous les deux en section critique en même temps. Cela implique que les variables `enSC[0]` et `enSC[1]` sont toutes les deux à vrai.
Cette situation est impossible car leurs tests respectifs pour entrer en section critique ne peuvent être vrais simultanément. En effet, la variable partagée `tour` sera favorable à un ou à l'autre des processus de la façon suivante :

- Un des processus, P_i par exemple, est entré en section critique car il a trouvé `tour = j`.
- P_j n'aura pas accès à la section critique car pour cela, il devrait trouver `tour = i` et la seule affectation à la variable `tour` qui lui est permise lui est défavorable.
- L'exclusion mutuelle est assurée.

2. L'algorithme ne provoque pas d'interblocage ;
   Considérons que P_0 est retenu dans sa boucle. Au bout d'un temps fini, P_1 sera :
   (a) soit non intéressé par la section critique ;
   Dans ce cas, `enSC[1] = faux` et P_0 passe.
   (b) soit en attente dans la boucle avec P_0 ;
   Cela est impossible car `tour` valant 0 ou 1 validera l'une des deux conditions. Les deux processus ne peuvent donc pas être tous les deux bloqués sur cette condition.
   (c) soit itère sur l'utilisation de la section critique, la monopolisant.
   Dans ce cas, P_1, en revenant dans le protocole d'entrée pour accéder à la section critique, devra d'abord repositionner `tour` à 1, ce qui validera plutôt la condition de P_0, qui lui, P_0, accédera alors à la section critique.

3. L'algorithme assure l'équité ;
   Il est impossible qu'un processus qui revient faire une demande pour entrer en section critique passe devant un processus en attente, car la seule affectation qu'il fera à la variable `tour` lui sera défavorable (ce qui donnera automatiquement la priorité au processus en attente). Ce positionnement défavorable assure l'équité.

3.2.11 Algorithme #13 : algorithme de Peterson (N processus)

L'extension à N processus de l'algorithme de Dekker n'est pas aussi triviale que certains ont voulu le faire croire. Peterson, en revanche, a montré que son algorithme se généralise facilement (c'est relatif !).

Le programme 3.18 présente cette généralisation. Le principe est simple : la solution du cas de deux processus est utilisée de manière répétitive ($n-1$ fois) pour éliminer au moins un processus à chaque tour jusqu'à ce qu'il n'en reste plus qu'un.

```
Program Peterson
begin /* programme */
var   enSC : array[0..n-1] of -1..n-2;
      tour : array[0..n-2] of 0..n-1;
      enSC[0..n-1] := -1;
      tour[0..n-2] := 0;
parbegin
      P(0); P(1); P(2); ...; P(n-1);
parend;
end. /* programme */
Procedure P(i : integer);
var j : integer;
begin /*procedure*/
  repeat
  {
     for (j:=0 to n-2)
     {
       enSC[i] := j;
       tour[j] := i;
       Repeat  /*rien*/
       until ((forall k!=i : enSC[k] < j) or (tour[j] != i));
     }
     ..... section critique
     enSC[i] := -1;
     ..... section non-critique

  } forever;
end;  /* procedure */
```

Programme 3.18 – Algorithme de Peterson (N processus)

La preuve du bon fonctionnement consiste toujours à démontrer que l'exclusion mutuelle est garantie, qu'il n'y a pas d'interblocage (livelock) et que l'équité est assurée.

1. L'algorithme assure l'exclusion mutuelle ;

   La variable `enSC` a été généralisée. Si elle contient -1, cela signifie le non-engagement du processus envers la section critique (avant représenté par faux). Les valeurs 0 à n-2 servent à représenter la progression du processus dans son itération par rapport aux autres processus.

De même, la variable `tour`, qui permet de gérer les conflits entre deux processus, est généralisée en un tableau identifiant le processus bloqué à chacune des itérations de la boucle. Ainsi, un processus P_i est bloqué à la $j^{ième}$ itération (dans la boucle) lorsque `tour[j]=i`.

Lorsqu'un processus pénètre en section critique, on a :

$$(\forall k \neq i : enSC[k] < n-2) \textbf{ ou } (tour[n-2] \neq i)$$

En d'autres termes, soit P_i la plus grande valeur de `enSC` , soit `enSC[i] = n-2` ou soit il n'est pas retardé par la $n-2^{ième}$ et dernière itération possible.

Il est aisé de voir que la supposition de deux processus en section critique conduit à une contradiction (il suffit de généraliser la preuve faite pour deux processus).

2. L'algorithme ne provoque pas d'interblocage ;

   De même, il est facile de montrer qu'il ne peut y avoir d'interblocage en généralisant la précédente preuve pour deux processus.

3. L'algorithme assure l'équité ;

   Ce protocole est aussi équitable. Considérons le cas extrême où tous les processus veulent constamment accéder à la section critique. Suivons l'évolution du processus P_i, le plus malchanceux des processus.

   À la première itération (`j=0`), $\forall k \neq i : enSC[k] < j$ étant faux pour tous les processus, un seul d'entre eux, P_i (le dernier à avoir positionné `tour[0]` à i), se bloquera. Au pire, il était premier et s'est fait doublé par les $n-1$ autres processus (il est tout de même le dernier à avoir positionné `tour[0]`).

   Toujours dans l'hypothèse la plus défavorable, ces $n-1$ processus reviennent demander l'exclusion mutuelle en même temps, avant que P_i ne puissent avancer dans la boucle. P_i sera automatiquement débloqué car un autre processus changera la valeur de `tour[0]` et se bloquera à sa place. P_i avancera à la seconde itération (`j=1`) et compétitionnera avec n-2 autres processus (un des processus est resté bloqué à la première itération). À cette deuxième itération, on suppose que P_i sera le dernier à positionner `tour[1]` (le chanceux!!!). Il se bloquera donc et se fera devancer par n-2 processus.

   Encore une fois dans l'hypothèse la plus défavorable, $n-2$ processus se présenteront à la seconde itération avant que P_i puisse avancer. P_i sera automatiquement débloqué car un autre processus changera la valeur de `tour[1]` et se bloquera à sa place. P_i avancera à la troisième itération (`j=2`) et compétitionnera avec n-3 autres processus (un des processus est resté bloqué à la première itération et un autre à la seconde). À cette troisième itération, on suppose que P_i sera le dernier à positionner `tour[2]` (encore plus chanceux!!!). Il se bloquera donc et se fera devancer par n-3 processus.

   Si on poursuit ainsi on se rend compte que :

   (a) à la première itération (`j=0`), P_i se fait devancer par n-1 processus ;

   (b) à la seconde itération (`j=0`), P_i se fait devancer par n-2 processus ;

   (c) à la troisième itération (`j=0`), P_i se fait devancer par n-3 processus ;

   (d) ...

   (e) à la $n-2^{ième}$ itération (`j=n-3`), P_i se fait devancer par 2 processus ;

(f) à la dernière itération (`j=n-2`), P_i se fait devancer par 1 processus ;

Nous obtenons alors, si $a(n)$ défini le nombre d'itérations d'attente lorsqu'il y a n processus :

$$a(n) = n - 1 + a(n-1) = n(n-1)/2$$

Ce protocole est caractérisé par la même borne que le protocole de De Brujin, i.e. n^2.

Pour mieux visualiser, la figure 3.9 illustre le déroulement de l'algorithme avec quatre processus. Les figures 3.9 a, 3.9 b et 3.9 c décrivent un premier passage des processus dans la boucle. Lors de ce passage le processus P_1 est le premier à se bloquer. Les figures 3.9 d, 3.9 e et 3.9 f présentent le second passage. Après quatre passages, on remarque que le processus P_1 atteint enfin la section critique, et ce malgré le pire cheminement possible (blocage à chaque étape).

3.2.12 Algorithme avec attente active utilisant des instructions machines

Dans cette section, nous introduisons d'autres solutions basées sur l'attente active mais qui, cette fois, font usage d'instructions spécifiques fournies par le matériel. Ces solutions ne respectent pas le concept d'indépendance envers le matériel mais respectent les autres contraintes.

Certaines machines fournissent des instructions spéciales permettant à un processus de tester (ou lire) et modifier le contenu de la mémoire ou d'échanger le contenu de deux zones mémoire en une seule opération et de façon atomique. Des exemples de telles opérations : `tst` (test-and-set), `swap`, `fadd` (fetch-and-add), `rmw` (read-modify-write), etc. Les programmes 3.19 et 3.20 implantent les opérations `tst` et `swap`.

Ces deux opérations sont caractérisées par l'atomicité de leur exécution. Ainsi si deux instructions `tst` ou `swap` sont exécutées simultanément, le résultat sera le même que si elles l'étaient séquentiellement dans un ordre quelconque.

Les programmes 3.21 et 3.22 implantent les protocoles d'entrée et de sortie par le biais de ces opérations. Notons qu'il est très simple d'assurer l'exclusion mutuelle grâce à celles-ci.

Il est aussi important de noter que ces instructions permettent d'implanter l'exclusion mutuelle simplement et sans presqu'aucun risque de famine ou d'interblocage.

On dit bien «presque» car cela peut se produire dans le cas où des processus hautement prioritaires monopolisent l'UCT en bouclant sur la condition pendant qu'un processus de basse priorité détient la section critique. Ce dernier ne pourra possiblement jamais reprendre le contrôle de l'UCT et les processus en attente seront bloqués (livelock). Un système de vieillissement ou de hausse temporaire de priorité peut régler ce rare problème.

De telles instructions sont maintenant fournies dans certains langages, notamment dans le langage C++ qui fournit les instructions suivantes :

- `atomic_compare_exchange`
- `atomic_fetch_and_add`

3.2.13 Problématiques dues aux algorithmes d'attente active

Les algorithmes de synchronisation basés sur l'attente active :

- sont difficiles à concevoir et à prouver ;
  Nous le constatons bien dans nos preuves plutôt «informelles».

		j=0		
Flag	P1	0 X		
	P2	0		
	P3	0		
	P4	0		
	tour	1		

a P1 bloque

		j=0	j=1	
Flag	P1	0 X		
	P2	0	1 X	
	P3	0	1	
	P4	0	1	
	tour	1	2	

b P2 bloque

		j=0	j=1	j=2
Flag	P1	0 X		
	P2	0	1 X	
	P3	0	1	2 X
	P4	0	1	2
	tour	1	2	3

c P3 bloque

P4 entre en section critique

P2 ,P3, P4 reviennent avant que P1 puisse avancer

		j=0		
Flag	P1	0		
	P2	0		
	P3	0		
	P4	0 X		
	tour	1→4	2	3

d P4 bloque (P1 avance)

		j=0	j=1	
Flag	P1	0	1 X	
	P2	0	1	
	P3	0	1	
	P4	0 X		
	tour	4	1	3

e P1 bloque

		j=0	j=1	j=2
Flag	P1	0	1 X	
	P2	0	1	2
	P3	0	1	2 X
	P4	0 X		
	tour	4	1	3

f P3 bloque

P2 en section critique

P2 et P3 reviennent avant que P1 et P4 puissent avancer

		j=0	*j=1*	
Flag	P1	0	*1 X*	
	P2	0		
	P3	0 X		
	P4	0		
	tour	4→3	1	3

g P3 bloque (P4 avance)

		j=0	j=1	
Flag	P1	0	1	
	P2	0	1 X	
	P3	0 X		
	P4	0	1	
	tour	3	1→2	3

h P2 bloque (P1 avance)

		j=0	j=1	j=2
Flag	P1	0	1	2 X
	P2	0	1 X	
	P3	0 X		
	P4	0	1	2
	tour	3	2	1

i P1 bloque

P4 entre en section critique

P4 revient avant que P1, P2 et P3 puissent avancer

		J=0	*j=1*	*j=2*
Flag	P1	0		*2 X*
	P2	0	*1 X*	
	P3	0		
	P4	0 X		
	tour	3→4	2	1

j P4 bloque (P3 avance)

		J=0	j=1	*j=2*
Flag	P1	0		*2 X*
	P2	0	1	
	P3	0	1 X	
	P4	0 X		
	tour	4	2→3	1

k P3 bloque (P2 avance)

		j=0	j=1	j=2
Flag	P1	0	1	2
	P2	0	1	2 X
	P3	0 X	1 X	
	P4	0 X		
	tour	4	3	1→2

l P2 bloque (P1 avance)

P1 en section critique

Figure 3.9 – Exemple de fonctionnement de l'algorithme de Peterson

```
1 procedure tst(var a,b : boolean)
2 begin
3    a:=b;
4    b:=true;
5 end
```

Programme 3.19 – Implantation de la fonction `tst` (test-and-set)

```
1 procedure swap(var a,b : boolean)
2 begin
3    var temp : boolean;
4    temp := a;
5    a := b;
6    b := temp;
7 end
```

Programme 3.20 – Implantation de la fonction `swap`

```
1 var active : boolean := false;
2
3 process P(i:=1 to n)
4   var libre : boolean;
5
6   repeat
7      libre := true;
8      while libre do tst(libre, active);
9
10     ... section critique
11
12     active := false;
13  forever;
```

Programme 3.21 – Exclusion mutuelle avec la fonction `tst`

```
1 var active : boolean := false;
2
3 process P(i := 1 to n)
4    var cle : boolean;
5
6    repeat
7       cle := true;
8       while (cle = TRUE)
9            swap(active,cle);
10      endwhile
11
12      ..... section critique
13
14      active := false;
15  forever;
```

Programme 3.22 – Exclusion mutuelle avec la fonction `swap`

- amènent des risques d'interblocage ;

  Ces risques sont toujours présents dus aux politiques et ce, même en recourant aux des instructions spéciales. En effet, des processus prioritaires «peuvent attendre» (en monopolisant le processeur) qu'un processus moins prioritaire libère la section critique.
- accaparent inutilement du temps de la machine ;

  Les processus bouclent en attente de la section critique. Ces boucles consomment énormément de temps UCT. Le processeur deviendrait plus productif s'il ne perdait pas de temps à boucler sur une condition quelconque.
- laissent toute la responsabilité de la synchronisation à l'utilisateur ;

  En effet, avec l'attente active, toute la synchronisation dépend de l'usager et il n'est jamais évident si une variable sert à implanter la synchronisation ou à tout autre chose.
- offrent des solutions peu lisibles ;
- sont difficiles à généraliser.

  Nous avons vu que la généralisation de l'algorithme de Dekker est très complexe et que celle de Peterson, même plus simple, n'en demeure pas moins ardue.

3.3 Sémaphore

Pour résoudre les problèmes engendrés par l'attente active, dans les années 60, Dijkstra a développé la notion de «sémaphores» [17, 5, 16, 1].

Un sémaphore est une variable entière sur laquelle on définit deux opérations, `P` (`signal`) et `V` (`wait`). Ce sont les deux seules opérations permises sur un sémaphore et celles-ci se doivent d'être atomiques.

Définition : Sémaphore

Un sémaphore est une variable entière sur laquelle on définit deux opérations, `P` (`signal`) et `V` (`wait`). Ce sont les deux seules opérations permises sur un sémaphore et celles-ci se doivent d'être atomiques.

Un sémaphore prend initialement une valeur $n > 0$.

Soit un sémaphore `S`, alors l'opération `P(S)` bloque le processus émetteur jusqu'à ce que $S > 0$. On associe à chaque sémaphore une file d'attente qui contiendra tous les processus bloqués (sur ce sémaphore). Lorsqu'un processus se bloque sur un sémaphore, on l'ajoute à la file d'attente de ce dernier.

L'opération V(S) débloque le premier processus en attente, s'il y en a un. S'il n'y a aucun processus en attente, les signaux s'accumulent.

Les programmes 3.23 et 3.24 fournissent deux implantations possibles pour un sémaphore. La première présente un sémaphore sous la forme d'un entier signé, acceptant des valeurs négatives. La valeur absolue de cet entier indique le nombre de processus en attente dans la file. La seconde implantation propose un sémaphore basé sur un entier strictement positif.

```
1  class semaphore
2  {
3        int valeur;
4        listeDePcs liste;
5    public:
6        void P();
7        void V();
8  }
9  semaphore::void P()
10 {  valeur--;
11    if (valeur < 0)
12    {    état du processus courant := bloqué;
13         liste.ajoute(processus courant);
14    }
15 }
16 semaphore::void V()
17 {  process processus;
18    valeur++;
19    if (valeur <= 0)
20    {   processus := liste.retire();
21        processus.etat := prêt;
22    }
23 }
```

Programme 3.23 – Définition d'un sémaphore

```
1  class semaphore
2  {
3        unsigned int valeur;
4        listeDePcs liste;
5    public:
6        void P();
7        void V();
8  }
9  semaphore::void P()
10 {  if (valeur = 0)
11    {    état du processus courant := bloqué;
12         liste.ajoute(processus courant);
13    }
14    else  valeur--;
15 }
16 semaphore::void V()
17 {  process processus;
18    if (!liste.vide())
19    {   processus := liste.retire();
20        processus.etat := prêt;
21    }
22    else valeur++;
23 }
```

Programme 3.24 – Définition d'un sémaphore

3.3.1 Aspects critiques pour l'implantation

Si les sémaphores sont utilisés correctement, ils assurent l'exclusion mutuelle, évitent les interblocages et selon l'implantation, assurent aussi l'équité. Toutefois leur implantation fait apparaître un certain nombre de défis.

La file d'attente

La contrainte d'équité d'un sémaphore dépend en grande partie de l'implantation de la file d'attente sous-jacente. Cette contrainte est satisfaite ou non par la gestion des insertions et des retraits de cette file. Si la liste est gérée avec une politique de type LIFO (Last-In-First-Out ou Pile), la famine de certains processus peut survenir. Par contre, sous la politique de type FIFO (Fisrt-in-First-Out), tous les processus sont à l'abri de la famine. Cependant, cette dernière politique est aussi considérée inéquitable dans le cas où l'on souhaite que les processus soient servis dans l'ordre de leur priorité. Il serait en effet peu judicieux de servir un processus de très basse priorité avant celui qui en possède une très haute. Il devient alors possible de gérer la file d'attente par priorité. Cependant, celle-ci ramène le risque de famine.

Un autre point à considérer : Où implanter la file d'attente ?

La file d'attente d'un sémaphore (lorsque celui-ci est fourni par le système d'exploitation) s'implante facilement avec une liste chaînée de descripteurs (PCBs) dans le noyau du système. Le sémaphore contient un pointeur vers le premier descripteur de la liste et chaque descripteur en contient un vers le suivant pour assurer la continuité. Quand un processus (son descripteur en fait) est ajouté à une liste d'attente associée à un sémaphore, son état devient «bloqué». La figure 3.10 illustre cette structure.

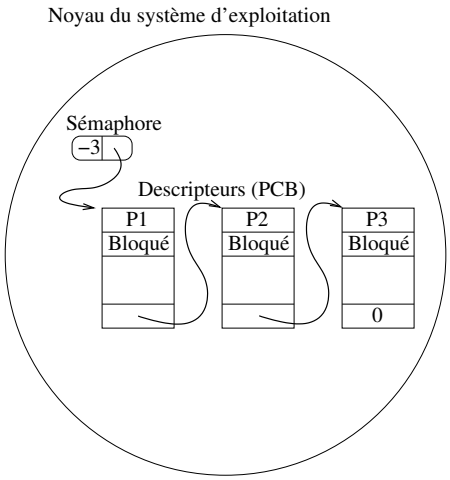


Figure 3.10 – File d'attente d'un sémaphore dans le noyau du système d'exploitation

La gestion de la liste de descripteurs est, soit sous la responsabilité d'un seul processeur, soit elle est partagée entre plusieurs. Si plusieurs processeurs y ont accès simultanément, elle devient sujette à l'accès concurrent ce qui requiert de l'exclusion mutuelle pour assurer l'atomicité des accès.

L'attente active est fréquemment utilisée à ce niveau comme nous le verrons dans la prochaine sous-section.

Atomicité

Comme déjà mentionné, les opérations `P` et `V` sur un sémaphore se doivent d'être atomique. L'implantation de cette atomicité présente toutefois un défi. Rappelons que l'atomicité garantie la validité des résultats même dans le cas où deux opérations s'exécutent simultanément sur un même sémaphore.

Définition : Atomicité

L'atomicité [6] est une propriété pour désigner une opération ou un ensemble d'opérations d'un programme qui s'exécutent entièrement sans pouvoir être interrompues avant la fin de leur déroulement. Une opération qui vérifie cette propriété est qualifiée d'« **atomique** », ce terme dérive du grec $\alpha\tau o\mu o\varsigma$ (atomos) qui signifie « que l'on ne peut diviser ».

Pour assurer l'atomicité, il faut donc exécuter les opérations en exclusion mutuelle. Les codes des procédures `P` et `V` sont alors des sections critiques. Il est important de noter l'impossibilité d'employer des sémaphores pour assurer l'exclusion mutuelle à ce niveau.

Si les sémaphores sont implantés dans le noyau du système d'exploitation, il est possible d'assurer l'atomicité sur un système mono-processeur (et mono-cœur), et ce, en rendant les opérations `P` et `V` non interruptibles. Il suffit pour cela de masquer les interruptions au niveau du processeur. Comme l'exécution de ces opérations est très courte, cela ne causera aucun problème.

Sur un système multi-processeurs (ou multi-cœurs), cette solution ne fonctionne plus. En effet, le masquage des interruptions se fait en positionnant des informations dans un registre associé à un seul processeur. Les autres processeurs accéderont toujours aux mêmes informations en même temps. L'exclusion mutuelle ne sera plus garantie.

La solution consiste à se tourner de nouveau vers l'attente active. Si le matériel fournit des instructions spéciales, celles-ci serviront alors à rendre l'exécution des `P` et `V` atomique, sinon, un algorithme comme celui de Peterson sera envisagé. Cette solution n'élimine cependant pas totalement l'attente active. Nous retirons plutôt cette dernière du programme usager pour la localiser à un endroit particulier (dans le noyau). Toutefois, même si l'attente active est présente, dû à la taille très limitée du code des instructions `P` et `V` (une dizaine de lignes de code), celle-ci est alors réduite à une très courte période de temps. Ce cas se distingue des algorithmes par attente active que l'on retrouve directement dans les applications et pour lesquelles les sections critiques ont généralement une durée significative.

Pour les cas où des sémaphores sont implantés à l'extérieur du noyau du système d'exploitation, soit l'attente active sera requise, soit on aura recours à un outil fournit par l'API du système.

3.3.2 Utilité des sémaphores

Le rôle des sémaphores est d'implanter :

1. l'exclusion mutuelle ;

   Pour assurer l'exclusion mutuelle, la valeur du sémaphore est initialisée à 1.

S'il existe plusieurs sections critiques à l'intérieur d'un même programme, il est possible d'associer des sémaphores différents à chacune des sections critiques. Ils assurent ainsi facilement et fiablement l'exclusion mutuelle.

Soit une section critique accédée par N processus. Dans ce cas, les processus se partagent un sémaphore, appelons-le `mutex`, initialisé à 1. La programme 3.25 présente la structure générale des différents processus.

```
1 semaphore mutex;
2 mutex.init(1);
3 ------------------------------------------
4    repeat
5       P(mutex)    // protocole d'entrée
6       .....section critique
7       V(mutex)    // protocole de sortie
8       .....section non-critique
9    forever
```

Programme 3.25 – Sémaphore et exclusion mutuelle

2. la synchronisation conditionnelle ;

   Les sémaphores permettent aussi d'implanter la synchronisation conditionnelle. Dans ce cas, le sémaphore, appelons-le `cond`, est initialisé à 0. Supposons deux processus, P_1 et P_2 exécutant les énoncés `E1` et `E2` respectivement. Supposons aussi que l'énoncé `E1` doit absolument s'exécuter avant l'énoncé `E2`. Les processus P_1 et P_2 partagent alors un sémaphore `cond` initialisé à 0. Le programme 3.26 expose la structure de ces deux processus.

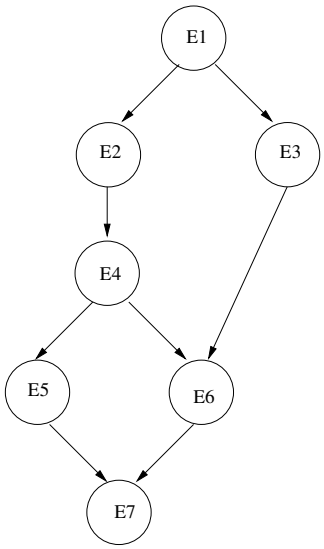
```
1  semaphore cond;
2  cond.init(0);
3  ------------------------------------------------
4  // processus P1          |      // processus P2
5  P1 : .                   |      P2 : .
6       .                   |           .
7       .                   |           .
8       S1;                 |           P(cond);
9       V(cond);            |           S2;
10      .                   |           .
11      .                   |           .
12      .                   |           .
```

Programme 3.26 – Sémaphore et synchronisation conditionnelle

   Le graphe de précédence présenté à la figure **??** de la section **??** constituait un cas de parallélisme non représentable par les énoncés `parbegin` et `parend`. La figure 3.11 propose une solution à ce cas de parallélisme en combinant les `parbegin/parend` et des sémaphores pour assurer l'ordonnancement des énoncés (synchronisation conditionnelle). Ce programme explicite le fait que l'utilisation des sémaphores rend les énoncés `parbegin/parend` aussi puissants que les énoncés `fork/join`.

3. un contrôle des accès concurrents ;

   Les sémaphores limitent aussi les accès à une ressource. Reprenons notre exemple de producteur/consommateur. Il s'agissait alors d'implanter un tampon contenant n espaces (tous




```
Var a,b,c,d,e,f,g : semaphores (=0)
begin
 parbegin
   begin E1; V(a); V(b); end;
   begin P(a); E2; E4; V(c); V(d); end;
   begin P(b); E3; V(e); end;
   begin P(c); E5; V(f); end;
   begin p(d); P(e); E6; V(g); end;
   begin P(f); P(g); E7; end;
 parend;
end;
```

Figure 3.11 – Graphe de précédence et son programme

disponibles au départ). Pour limiter les accès à ce tampon, on emploie un sémaphore, appelons le `sem1`, initialisé à n. Celui-ci évitera au producteur de générer plus d'éléments qu'il y a d'espaces disponibles dans le tampon. Avant d'ajouter un élément, un producteur devra toujours exécuter l'énoncé «`P(sem1);`». Nous reviendrons sur cet exemple ultérieurement.

3.3.3 Avantages et inconvénients des sémaphores

Les principaux avantages des sémaphores sont :

1. leur puissance ;
2. leur facilité d'utilisation ;
3. leur capacité à éviter les interblocages ;
4. leur capacité à assurer l'équité.

Toutefois ils possèdent les défauts suivants :

1. dans certains cas, un processus doit connaître le nom de ses compétiteurs afin de les réactiver ;
2. les opérations sur les sémaphores doivent se manipuler avec soin. Les programmes qui y font appel sont difficiles à vérifier et une mauvaise programmation mène aisément à un interblocage. Par exemple, on peut :
   - omettre un `P` au début d'une section critique ;
   - omettre un `V` à la fin d'une section critique ;
   - inverser le `P` et le `V` ;
   - mettre un `V` à la place du `P` ou l'inverse ;
   - faire un `P` sur un sémaphore et un `V` sur un autre.
3. Il est aisé d'oublier de placer en section critique tous les énoncés qui réfèrent à l'objet partagé.

4. L'exclusion mutuelle et la synchronisation conditionnelle sont assurées avec les mêmes primitives, rendant ainsi complexe l'identification du rôle précis d'une opération `P` ou `V` particulière sans référer aux opérations complémentaires correspondantes.
5. Les opérations P et V ne donnent aucune indication sur la ressource visée

Autres problèmes ou inconvénients associés aux sémaphores sont :

- la libération accidentelle d'un sémaphore ;
- les interblocages récursifs ;

  Cette situation est susceptible d'apparaître quand une tâche tente de faire deux `P` consécutifs sur le même sémaphore (sans faire de V). Un sémaphore normal bloquerait le processus (interblocage récursif). Cependant, des sémaphores «plus intelligents» existent, capables de détecter les cas où le processus détient déjà le sémaphore, et ainsi éviter qu'il ne bloque (sémaphore ré-entrant ou récursif [7, 15]).

Définition : Interblocage récursif

Un interblocage récursif (*self deadlock* ou *recursive deadlock*) se produit lorsqu'un processus (ou un fil d'exécution) tente d'acquérir un verrou (sémahore, mutex, ...) qu'il détient déjà.

Définition : Sémaphore ré-entrant ou récursif

Un mécanisme de verrouillage ré-entrant ou récursif (mutex, sémaphore, ...) permet à un même processus de faire appel à ce mécanisme (de le verrouiller) de façon répétitive sans causer d'interblocage.

- L'inversion de priorité ;

  Cette situation survient lorsqu'un processus de basse priorité monopolise une section critique pour laquelle un autre processus, de plus haute priorité, attend. Pour permettre à ce dernier de s'exécuter rapidement, on devra augmenter temporairement la priorité du processus détenant la ressource visée.
- Interblocage par la mort d'un processus ;

  Cette situation survient lorsqu'un processus détenant une section critique est détruit avant de la libérer. Il faudra prévoir libérer tous les sémaphores qu'il détient.
- Le temps pour les changements de contexte.

  Lorsqu'un processus se bloque sur un sémaphore, il se produit au moins deux changements de contexte : un pour lui retirer le contrôle du processeur et un autre pour lui redonner. Le problème réside dans le fait que le temps d'attente pour la section critique est parfois beaucoup moins long que celui des changements de contexte eux-mêmes. C'est du temps «perdu» pour l'UCT et une attente trop longue pour le processus. Certaines implantations de sémaphores prévoient donc une courte attente active avant de bloquer le processus. Ainsi, si la section critique se libère rapidement, il ne se produira pas de coûteux changements de contexte.

Les sémaphores constituent un très bon outil pour la synchronisation mais c'est avant tout un

outil de bas niveau (équivalent au langage d'assemblage comparé à un langage évolué tel Java ou Python). En conséquence, leur emploi devrait être principalement restreint à certains contextes précis (le noyau, les appels générés par le compilateur, etc.). Les sémaphores servent aussi à implanter des techniques de synchronisation plus évoluées, lesquelles sont abordées dans les prochains chapitres. Notons toutefois qu'actuellement, les sémaphores ne sont pas réservés à des environnements de «bas niveaux», ils sont directement accessibles dans la plupart des langages modernes.

3.3.4 Sémaphores dans la littérature, les systèmes et les langages

Les **sémaphores généraux**, pouvant prendre plusieurs valeurs, sont disponibles dans la plupart des langages de programmation parallèle. Dans la littérature, ils sont souvent appelés des «counting semaphores».

Les **sémaphores binaires** sont fréquemment abordés dans la littérature. Ceux-ci, comme leur nom l'indique, n'acceptent que les valeurs 0 ou 1. Ces sémaphores ont l'avantage d'offrir une implantation très optimale. Ils ne peuvent toutefois pas accumuler les signaux.

Un «type» de sémaphores binaires fréquemment rencontré dans les langages de haut niveau ou dans les bibliothèques de fils d'exécution (tel Posix) est le «**mutex**». Les «**mutex**» sont des sémaphores binaires dont l'unique fonction consiste à implanter l'exclusion mutuelle. On leur associe toujours une contrainte supplémentaire à savoir que seul le processus ayant fait le `P` sur le sémaphore peut émettre le `V` correspondant. Ils servent aussi à implanter une forme de moniteur (notion que nous approfondirons plus tard).

Un dernier «type» de sémaphore, le **futex** (Fast Userspace Mutex), apparaît dans le noyau de Linux. Un **futex** est un appel système qui fournit une forme primitive de sémaphore en mode utilisateur permettant également de construire des éléments plus élaborés tels que les sémaphores ou les mutex POSIX sous Linux.

3.4 Exemples classiques

3.4.1 Problème du tampon fini (producteurs/consommateurs)

Nous avons déjà abordé ce cas dans les sections précédentes sous le nom de «producteurs/consommateurs». Considérons un tampon contenant n espaces, chacun pouvant contenir un «item».

Pour implanter le tampon, on utilise trois sémaphores :

- un sémaphore, `mutex`, d'exclusion mutuelle ;
- un sémaphore, (`est_plein`), pour indiquer que le tampon est plein (synchronisation conditionnelle) ;
- un sémaphore, (`est_vide`), pour indiquer que le tampon est vide (synchronisation conditionnelle).

Le programme 3.27 implante un tampon à n éléments.

Pour assurer une exclusion mutuelle sélective, on définit un sémaphore distinct pour chaque section critique (avec un nom significatif). Le programme 3.28 propose un exemple d'utilisation sélective des sémaphores pour implanter un mini système d'exploitation à traitement par lots. Les sémaphores définis par le programme sont :

- in_mutex : pour assurer l'exclusion mutuelle sur un tampon d'entrée ;
- out_mutex : pour assurer l'exclusion mutuelle sur un tampon de sortie ;

```
1  type  item : ...
2  var   est_plein, est_vide, mutex : semaphore};
3        tampon  array[0..n-1] of item;
4        nextp, nextc : item;
5  begin
6      est_plein := n; est_vide := 0; mutex := 1;
7      parbegin
8      producteur    : repeat
9                           ... produire un "item" dans  nextp;
10                          P(est_plein);
11                          P(mutex);
12                          ... dépose nextp dans tampon;
13                          V(mutex);
14                          V(est_vide);
15                    forever;
16     consommateur :  repeat
17                          P(est_vide);
18                          P(mutex);
19                          ... lire nextc  de tampon;
20                          V(mutex);
21                          V(est_plein);
22                          ... traite nextc;
23                   forever;
24     parend;
25 end;
```

Programme 3.27 – Tampon à n éléments

- num_in : pour assurer la synchronisation conditionnelle sur le tampon d'entrée (pour vérifier qu'il n'est pas vide) ;
- num_out : pour assurer la synchronisation conditionnelle sur le tampon de sortie (pour vérifier qu'il n'est pas vide) ;
- free_in : pour assurer la synchronisation conditionnelle sur le tampon d'entrée (pour vérifier qu'il n'est pas plein) ;
- free_out : pour assurer la synchronisation conditionnelle sur le tampon de sortie (pour vérifier qu'il n'est pas plein) ;

3.4.2 Problème des lecteurs/écrivains

Soit un objet de données (fichiers ou autres) partageable entre plusieurs processus concurrents. Certains d'entre eux ne font que des lectures (lecteurs) et d'autres n'effectuent que des mises à jour (lecture et écriture : ce sont les écrivains).

La distinction entre eux est importante car le résultat de plusieurs lectures simultanées sur cet objet sera toujours correct (nécessairement valide). En revanche, si plusieurs écrivains, ou même un seul mais avec plusieurs lecteurs, accèdent simultanément au même objet, alors des incohérences peuvent survenir.

Reprenons notre exemple du chapitre précédent sur l'accès simultané à un compte en banque pour lequel un processus veut retirer 10$ et un autre ajouter 1000$. La figure 3.12 met en lumière une incohérence générée par des accès simultanés.

Pour éviter cette situation, il est impératif de fournir un accès exclusif à un écrivain. Il est toutefois correct de fournir l'accès à plusieurs lecteurs simultanément. Plusieurs variantes sont

```
1  Program OPSYS;
2  var  in_mutex, out_mutex :  semaphore initial (1,1);
3       nun_in, num_out : semaphore initial (0,0);
4       free_in, free_out : semaphore initial (n,n);
5       tampon_in :  array[0..n-1]  of entree;
6       tampon_out : array[0..n-1]  of sortie;
7  process lecteur;
8    var ligne : entree;
9    loop
10       lecture ligne;
11       P(free_in);
12       P(in_mutex);
13       ... dépose ligne  dans tampon_in;
14       V(in_mutex);
15       V(num_in);
16   end;
17 end process;
18
19 process traitement;
20   var ligne : entree; resultat : sortie;
21   loop
22      P(num_in);
23      P(in_mutex);
24      ... lecture ligne de tampon_in;
25      V(in_mutex);
26      V(free_in);
27      ... traitement de ligne et génération de resultat;
28      P(free_out);
29      P(out_mutex);
30      ... dépose} resultat dans} tampon_out;
31      V(out_mutex);
32      V(num_out);
33   end;
34 end process;
35
36 process imprimante;
37   var resultat : sortie;
38   loop
39      P(num_out);
40      P(out_mutex);
41      ... lecture} resultat de tampon_out;
42      V(out_mutex);
43      V(free_out);
44      ... impression de resultat;
45      end;
46  end process;
```

Programme 3.28 – Mini système à traitement par lots.

Compte A contient 500$		
Processus P1	Processus P2	Argent dans le compte
—	—	500
`Lecture` A `(500$)`		500
	`Lecture` A `(500$)`	500
	`Ajout de 1000$ (1500$)`	500
	`Écriture` A `(1500$)`	1500
`Retrait de 10$ (490$)`		1500
`Écriture` A `(490$)`		490$
Problème : les deux processus poursuivent leur exécution		

Figure 3.12 – Deux processus qui modifient un compte simultanément

possibles pour solutionner le problème des lecteurs/écrivains :

1. **priorité aux lecteurs**

   Dans ce cas, aucun lecteur n'attend, à moins qu'un écrivain n'occupe déjà la ressource.

   En d'autres termes, tant qu'un lecteur occupe la ressource, tous les autres lecteurs passent. Les écrivains devront attendre qu'il n'y ait plus aucun lecteur dans la file d'attente. Il risque donc d'y avoir famine pour les écrivains. De plus, les lecteurs n'auront pas une information à jour vu l'impossibilité d'effectuer des mises à jour.
2. **priorité aux écrivains**

   On donne accès à un écrivain aussitôt qu'il en fait la demande.

   Ainsi, si un écrivain attend, aucune lecture ne peut débuter (même si des lectures sont déjà en cours). Seules celles déjà commencées pourront se terminer. Il est possible que cette solution cause une famine chez les lecteurs.
3. **aucune priorité**

   On donne accès aux lecteurs et aux écrivains dans l'ordre d'arrivée.

   Dans ce cas, les demandes de lectures pourront se faire simultanément si seulement aucune autre demande d'écriture ne survient entre elles.
4. des variations qui éviteraient la famine.

Le programme 3.29 propose une solution au problème des lecteurs/écrivains, dans laquelle on fait appel à :

- deux sémaphores d'exclusion mutuelle, `mutex` et `wrt`, initialisés à 1.

  Le sémaphore `mutex` assure l'exclusion mutuelle pour l'accès à la variable `nbLecture`, tandis que le sémaphore `wrt` assure l'exclusion mutuelle pour l'accès exclusif à la donnée.

  Lorsqu'un écrivain entre en section critique et qu'il y a plusieurs lecteurs en attente, l'un de ces derniers est bloqué sur le sémaphore `wrt` et les autres le sont sur le sémaphore `mutex`.
- un entier `nbLecture` initialisé à 0.

  Cette variable compte le nombre de lectures simultanées sur la ressource.

Cette solution accorde-t-elle une priorité aux écrivains ? Aux lecteurs ? Ou n'en accorde-t-elle aucune ?

```
1  ...
2  parbegin
3     Lecteur : P(mutex)
4                  nblecteur++
5                  if (nblecteur=1) P(wrt)
6               V(mutex)
7               .... lecture.....
8               P(mutex)
9                  nblecteur--
10                 if (nblecteur=0) V(wrt)
11              V(mutex)
12
13    Écrivain: P(wrt)
14              .... écriture .....
15              V(wrt)
16 parend
17 ...
```

Programme 3.29 – Solution au problème des lecteurs/écrivains.

3.4.3 Problème des philosophes

Ce problème fut énoncé par Dijkstra en 1965.

Cinq philosophes passent leur vie à penser et à manger. Comme illustré à la figure 3.13, ces philosophes partagent une table circulaire ayant cinq chaises, chacune appartenant à l'un des philosophes. Sur la table, on retrouve cinq baguettes et cinq plats de riz [2].

Lorsqu'un philosophe pense, il n'interagit pas avec ses collègues. À un moment donné, l'un des philosophes a faim. Il essaie alors de se procurer deux baguettes qui lui sont nécessaires pour se nourrir. Un philosophe ne peut prendre qu'une seule baguette à la fois. Il doit obligatoirement prendre les baguettes les plus rapprochées de la place qui lui est assignée, soient celles immédiatement à sa droite et à sa gauche. Ainsi le philosophe assis à la position 1 prendra les baguettes 1 et 2. À vous de constater qu'alors deux philosophes voisins ne pourront jamais manger en même temps. Un philosophe ne peut évidemment pas prendre une baguette qui est déjà dans la main d'un autre philosophe. Si l'une des baguettes n'est pas disponible, il attend.

Lorsqu'un philosophe a deux baguettes, il mange. Lorsqu'il termine, il libère ses baguettes et recommence à penser. Le cycle se poursuit ainsi à l'infini.

Le programme 3.30 implante une solution au problème des philosophes. Dans celle-ci, chaque baguette est représentée par un sémaphore. Lorsqu'un philosophe essaie de prendre une baguette, il exécute un `P` et lorsqu'il la libère, il exécute un `V`. Cette solution garantie que deux philosophes voisins ne mangeront jamais simultanément.

Cette solution n'est toutefois pas valide ! En effet, un interblocage est possible. **Trouver la séquence d'exécution qui mène à cet interblocage.**

Énumérons ici quelques solutions fonctionnelles ne menant pas à un interblocage :

- autoriser seulement quatre philosophes à s'asseoir à cette table en même temps ;
- ne permettre à un philosophe de prendre ses baguettes seulement lorsqu'elles sont toutes les deux libres ;
- utiliser une solution asymétrique forçant certains philosophes à prendre leurs baguettes dans un ordre différent. Par exemple, les philosophes pairs accéderaient à leur baguette de droite,

2. Une autre version du problème utilise plutôt du spaghetti.

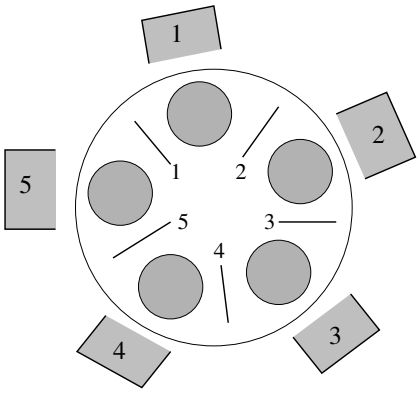


Figure 3.13 – Le problème des philosophes

```
1  var baguette : array[0..4] of semaphore;
2  procedure phil(i:integer)
3  begin
4     repeat
5        P(baguette[i])
6        P(baguette[i+1mod5])
7       ...mange ...
8        V(baguette[i]
9        V(baguette[i+1mod5])
10      ... pense ...
11    forever
12 end
13 begin
14    baguette[0..4] :=1
15    cobegin
16       phil(0); phil(1); phil(2); phil(3); phil(4)
17    coend
18 end
```

Programme 3.30 – Solution au problème des philosophes.

puis celle de gauche, tandis que les philosophes impairs à leur baguette de gauche, puis à celle de droite.

Une bonne solution doit aussi éviter la famine. Une solution sans interblocage ne garantit pas que la famine ne surviendra pas.

3.5 Conclusion

Dans ce chapitre, plusieurs solutions ont été présentées pour assurer la synchronisation, soient les algorithmes basés sur l'attente active et les sémaphores. Tout semble donc beau.... En théorie.

Le problème réside dans le fait que la plupart des algorithmes basés sur l'attente active ne fonctionnent plus sur les ordinateurs modernes. En effet, ces algorithmes supposent que la mémoire de l'ordinateur fournit une cohérence stricte des informations en mémoire (la cohérence séquentielle). Cependant, cette forme de cohérence est absente sur bon nombre (sinon tous) des ordinateurs

modernes. Ceux-ci en fournissent une, mais plus faible (Nous abordons ces différentes formes de cohérence mémoire dans un prochain chapitre.).

Autre inconvénient : le ré-ordonnancement des instructions (quand elles sont indépendantes) effectué par certains compilateurs (optimisation) ou certains ordinateurs (super-scalaire ou autres optimisations) est susceptible de nuire au bon fonctionnement de ces algorithmes.

En somme, les algorithmes de Dekker et de Peterson ne fonctionnent plus. Illustrons ce fait par le programme 3.31, dans lequel deux fils d'exécution incrémentent une même variable `cpt`. L'algorithme de Peterson, implanté en C++, assure l'accès exclusif à la variable. À la fin, on s'attend à ce que «`cpt = 60 000`». La figure 3.14 reproduit plusieurs exécutions de ce programme. Force est d'admettre que le programme ne produit pas le résultat escompté. L'algorithme de Peterson n'assure donc pas l'exclusion mutuelle sur l'accès au compteur.

```
#include <thread>
#include <iostream>
#include <vector>
#include <chrono>
#include <atomic>

using namespace std;
bool interet[2];
int victime = 0;
int val;
int cpt = 0;

void fil(int no)
{
    for(int i = 0; i < 30000; ++i)
    {
        int autre = 1 - no;
        interet[no] = true;
        victime = no;
        while (interet[autre] && victime == no) ;
           // Section critique
           cpt++;
           // fin de la section critique
        interet[no] = false;  //protocole de sortie
    }
    cout << "Fin de l'exécution du fil "<< no << endl;
}

int main(){
    interet[0] = false;
    interet[1] = false;

    vector<thread> threads;
    for(int i = 0; i < 2; ++i)   { threads.push_back(thread(fil, i)); }

    for(auto& thread : threads)  {  thread.join(); }

    std::cout << "fin du programme avec cpt = " << cpt << std::endl;
    return 0;
}
```

Programme 3.31 – Algorithme de Peterson en C++.

Les programmes 3.32 et 3.33 proposent des solutions fonctionnelles permettant de passer outre les optimisations fournies pas les compilateurs et les ordinateurs (cache et autres). Le premier

```
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59998
-------------
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59999
-------------
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59994
-------------
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59997
-------------
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59999
-------------
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59998
-------------
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59993
-------------
Test\$ ./peterson
Fin du fil 0
Fin du fil 1
Fin de l'exécution du programme avec cpt = 59997
```

Figure 3.14 – Résultats de plusieurs exécutions du programme

programme fait appel à une instruction de type barrière pour assurer la cohérence des données. Le second utilise des variables de types «`atomic`» fournies par le langage C++. Le résultat produit par chacune des exécutions de ces programmes est `60 000`.

```cpp
#include <thread>
#include <iostream>
#include <vector>
#include <chrono>
#include <atomic>

using namespace std;
bool interet[2];
int victime = 0;
int val;
int cpt = 0;

void fil(int no)
{
    for(int i = 0; i < 100000; ++i)
    {
        int autre = 1 - no;
        interet[no] = true;
        victime = no;

        // Utilisation d'une barrière pour assurer la cohérence des données
        asm ("mfence");

        while (interet[autre] && victime == no) {continue;}
        // Section critique
        cpt++;

        // Fin de la section critique
        interet[no] = false;
    }
    cout << "Fin de l'exécution du fil "<< no << endl;
}

int main(){
    interet[0] = false;
    interet[1] = false;

    vector<thread> threads;
    for(int i = 0; i < 2; ++i)   {   threads.push_back(thread(fil, i));   }

    for(auto& thread : threads)  {   thread.join();  }

    std::cout << "fin du programme avec cpt = " << cpt << std::endl;
    return 0;
}
```

Programme 3.32 – Algorithme de Peterson en C++.

```cpp
#include <thread>
#include <iostream>
#include <vector>
#include <chrono>
#include <atomic>

using namespace std;

atomic<bool> interet[2];    // utilisation de variable atomique
atomic<int> victime(0);     // utilisation de variable atomique
int val;
int cpt = 0;

void fil(int no)
{
    for(int i = 0; i < 100000; ++i)
    {
        int autre = 1 - no;
        interet[no] = true;
        victime = no;
        while (interet[autre] && victime == no) {continue;}
        // Section critique
        cpt++;

        // Fin de la section critique
        interet[no] = false;
    }
    std::cout << "fin du programme avec cpt = " << cpt << std::endl;
}

int main(){
    interet[0] = false;
    interet[1] = false;

    vector<thread> threads;
    for(int i = 0; i < 2; ++i)    {     threads.push_back(thread(fil, i));  }

    for(auto& thread : threads)   {     thread.join();  }

    std::cout << "fin..... " << cpt << std::endl;
    return 0;
}
```

Programme 3.33 – Algorithme de Peterson en C++.

Bibliographie

[1] GeeksForGeeks : Difference between counting and binary semaphores. https://www.geeksforgeeks.org/difference-between-counting-and-binary-semaphores/, 2021.

[2] James L. Peterson et Abraham. Silberschatz : *Operating system concepts / James L. Peterson, Abraham Silberschatz*. Addison-Wesley Pub. Co Reading, Mass, 1983.

[3] Abraham Silberschatz, Greg Gagne et Peter Baer Galvin : *Operating System Concepts*. Wiley, 6 édition, 2002.

[4] Abraham Silberschatz, Peter Baer Galvin et Greg Gagne : *Operating System Concepts, 10th Edition*. Wiley, 2018.

[5] Techno-Science.net : Sémaphore (informatique) - définition et explications. https://www.techno-science.net/glossaire-definition/Semaphore-informatique.html, 2021.

[6] Wikipedia : Atomicité (informatique). https://fr.wikipedia.org/wiki/Atomicit%C3%A9_(informatique), 2018.

[7] Wikipedia : Sémaphore réentrant. https://fr.wikipedia.org/wiki/Mutex_rÃ©entrant, 2018.

[8] Wikipedia : Leslie amport. https://fr.wikipedia.org/wiki/Leslie_Lamport, 2020.

[9] Wikipedia : Leslie amport. https://en.wikipedia.org/wiki/Leslie_Lamport, 2020.

[10] Wikipedia : Edsger dijkstra. https://fr.wikipedia.org/wiki/Edsger_Dijkstra, 2021.

[11] Wikipedia : Edsger wybe dijkstra. https://en.wikipedia.org/wiki/Edsger_W._Dijkstra, 2021.

[12] Wikipedia : Loi de wirth. https://fr.wikipedia.org/wiki/Loi_de_Wirth, 2021.

[13] Wikipedia : Niklaus wirth. https://fr.wikipedia.org/wiki/Niklaus_Wirth, 2021.

[14] Wikipedia : Niklaus wirth. https://en.wikipedia.org/wiki/Niklaus_Wirth, 2021.

[15] Wikipedia : Reentrant mutex. https://en.wikipedia.org/wiki/Reentrant_mutex, 2021.

[16] Wikipedia : Semaphore (programming). https://en.wikipedia.org/wiki/Semaphore_(programming), 2021.

[17] Wikipedia : Sémaphore (informatique). https://fr.wikipedia.org/wiki/S%C3%A9maphore_(informatique), 2021.

[18] Wikipedia : Donald knuth. https://fr.wikipedia.org/wiki/Donald_Knuth, 2022.

[19] Wikipedia : Donald knuth. https://en.wikipedia.org/wiki/Donald_Knuth, 2022.