Djamel Eddine Saïdouni Mourad Bouzenada

Synchronisation des processus dans environnement centralisé

Support de cours avec exercices solutionnés destiné aux étudiants de la licence L3-SCI et du master M1-STIC

janvier 5, 2020

Département IFA, Faculté des NTIC
Université Constantine 2 - Abdelhamid Mehri

Preface

Ce support de cours est le fruit de plusieurs années d'enseignement de la matière par ses auteurs. Il traite le problème de la synchronisation des processus dans un environnement centralisé. L'objectif étant d'offrir un document pédagogique pour les étudiants de graduation en informatique, toutes spécialités confondues, qui ont dans leur cursus un module sur les processus concurrents et coopératifs. Le manuscrit est composé de cinq chapitres. Le premier chapitre introduit la notion de concurrence entre les processus tout en expliquant le problème du partage de ressources dont l'utilisation doit se faire en exclusion mutuelle par les processus. Ainsi, le problème de la section critique est mis en évidence. Les propriétés des solutions résolvant ce problème sont alors définies. Comme première solution à ce problème, ce premier chapitre présente les solutions algorithmiques pour deux processus et généralisées par la suite à un nombre quelconque de processus.

Le deuxième chapitre introduit le mécanisme de synchronisation des sémaphores offert par les systèmes d'exploitation. La sémantique fonctionnelle des sémaphores est d'abord expliquée, puis les techniques d'implémentation de ce mécanisme sont présentées. Les différents types de sémaphores sont exposés ainsi que leurs intérêts respectifs dans la synchronisation des processus concurrents et coopératifs.

Le troisième chapitre est consacré au mécanisme de synchronisation de haut niveaux offert par les langages de programmation, à savoir les moniteurs.

Comme problème qui s'impose dans les applications concurrentes, il y a le problème de l'interblocage induit par l'allocation des ressources disponibles dans le système. Le quatrième chapitre définit ce problème et présente les différentes approches visant à le résoudre.

Le cinquième chapitre présente les différents paradigmes universels de synchronisation, à savoir le paradigme des philosophes, celui des lecteurs et des rédacteurs et celui des producteurs et des consommateurs. Le regroupement de ces paradigmes dans un chapitre à part est fait dans l'objectif de comparer les différentes solutions utilisant différents mécanismes de synchronisation.

Pour faciliter la compréhension de ces concepts par les étudiants, chaque chapitre est divisé en deux parties. La première partie donne un résumé du cours et la deuxième partie donne une série d'exercices solutionnés. L'étudiant pourra ainsi

s'introduire graduellement dans la matière en assimilant aussi bien les concepts théoriques que les problèmes types pouvant être utilisés comme source d'inspiration pour résoudre des problèmes pratiques.

Il est à noter que ce support de cours représente une entrée pédagogique à la matière. Les étudiants sont invités à approfondir leurs connaissances en consultant d'autres ouvrages plus riches tels que Silberschatz et al (2018), Tanenbaum (2009).

Constantine, *Saidouni Djamel Eddine*
juin 2020, *Bouzenada Mourad*

Table des matières

Chapter 1
Synchronisation par attente active

Rappel de cours

1.1 Problème de la section critique

1.1.1 Exemple illustratif

Soit l'exemple du programme suivant dans lequel les deux processus concurrents *Processus 1* et *Processus 2* partagent en lecture écriture la variable x initialisée à 0.

Programme

Var x: integer init 0 // variable partagée

Processus 1	**Processus 2**
Begin	**Begin**
.	.
.	.
$x := x+1$	$x := x+1$
.	.
.	.
End	**End**

Quel est la valeur attendue de x ?

Nous pouvons penser que la valeur de la variable x sera égale à 2. Cela est vraie si les deux exécutions concurrentes de l'instruction $x := x+1$ sont faites de manière séquentielle. Cependant, le programme ci-dessous, écrit dans un langage de haut niveau, sera exécuté après sa compilation. Dans le programme résultant de la compilation, l'instruction $x := x+1$ sera replacée par trois instructions machines qui ressemblera au code suivant dans lequel x désignera l'adresse de l'espace mémoire qui contiendra la valeur de cette variable:

LDA x Qui signifie: Charger le contenu de l'espace mémoire d'adresse x dans l'accumulateur

INC Qui signifie: Incrémenter l'accumulateur de 1

STA x Qui signifie: stocker le contenu l'accumulateur dans l'espace mémoire d'adresse x

Ainsi, le programme résultant est comme suit dans lequel nous avons numéroté les instructions machines.

Programme

Var x: integer init 0 // variable partagée

Processus 1	**Processus 2**
Begin	**Begin**
.	.
.	.
(1)*LDA x*	(1') *LDA x*
(2)*INC*	(2') *INC*
(3) *STA x*	(3') *STA x*
.	.
.	.
End	**End**

Soit le scénario suivant d'exécution des processus 1 et 2. Supposons que l'état du processus 1 est *en exécution* alors que l'état du processus 2 est *prêt*. Supposons qu'après l'exécution de l'instruction (1) le processus 1 est interrompu suite à la fin du quantum de temps processeur. Après commutation du contexte, la valeur de l'accumulateur stockée dans le PCB (Process Control Block) est égale à 0. Après le chargement du processus 2 au niveau du processeur, l'exécution des instructions (1'), (2') et (3') utilise donc la valeur initiale de la variable x qui est égale à 0. La variable x sera égale à 1 après la fin d'exécution du processus 2. Après le chargement du processus 1 au niveau du processus, la valeur de l'accumulateur récupérée est égale à 0. L'exécution des instructions (2) et (3) finira par l'attribution de la valeur 1 à la variable x, ce qui contredit la valeur attendue de cette variable sensée être égale à 2.

àtant donné que les deux instructions $x := x+1$ manipulant la même variable x en lecture/écriture peuvent être exécutées concurremment par les processus *Processus 1* et *Processus 2*, chacune d'elle constitue une section critique pour le processus auquel elle appartient.

Une solution de gestion des sections critiques consiste à imposer que ces dernières soient exécutées de manière entrelacée dans le temps (une après une). On dit aussi qu'une section critique doit être indivisible ou atomique.

Definition 1.1. Section (de code) Critique : Code devant pouvoir faire l'hypothèse qu'il utilise la ressource de manière exclusive.

Si aucune précaution particulière n'est prise, rien n'empêche plusieurs processus à la fois d'utiliser la ressource. Ce qui implique:

- Empêcher les processus qui sont en compétition (pour une ressource donnée) d'entrer simultanément dans leur section (de code) critique.
- Les sections critiques s'exécutent donc en **exclusion mutuelle**.

1.1.2 Structure générale des solutions

La structure générale de chaque processus en concurrence pour l'accès à sa section critique est la suivante:

Repeat
Section de début (SD)
section d'entrée
section critique (SC)
section de sortie
section restante (SR)
Until Condition

La section de début **(SD)**, la section critique **(SC)** et la section restante **(SR)** font partie du code utilisateur. Chacune de ces section est propre au processus auquel elle appartient.

La solution au problème de la section critique consiste donc en la définition de la section d'entrée (section de code qui effectue une demande d'entrée dans la section critique) et la section de sortie (code qui suit normalement la section critique).

1.2 Solutions algorithmiques

1.2.1 Critères de validité d'une solution au problème de la section critique

Hypothèse: Le temps de résidence d'un processus dans sa section critique est fini.

Toute solution au problème de la section critique doit vérifier les propriétés suivantes:

1. **Exclusion mutuelle:** Si le processus P_i exécute sa section critique SC_i alors aucun autre processus P_j ne peut exécuter sa section critique SC_j. Cette propriété s'exprime par:
   - Ã $tout\ instant\ t, Not(P_i\ en\ SC_i\ and\ P_j\ en\ SC_j)$
2. **Progrès:** Si aucun processus n'est dans sa section critique et si certains processus désirent entrer dans les leurs, alors seulement les processus qui ne se trouvent pas dans leurs sections restantes peuvent décider qui rentrera prochainement à sa section critique. Cette sélection ne peut pas être reportée indéfiniment. Cette propriété reflète aussi :
   - l'absence de l'interblocage.
   - La non interférence: Si un processus s'arrête dans sa section restante, les autres processus devront pouvoir accéder à leurs sections critiques.

3. **Attente limitée:** Il doit exister une limite au nombre de fois que l'on permet à d'autres processus à entrer dans leurs sections critiques après qu'un processus a effectué une requête pour entrer dans la sienne et avant que la requête ne soit accordée. Cette propriété reflète:
   - l'absence de famine pour chaque processus.

1.2.2 Classification des propriétés d'une solution

Les propriétés définies précédemment peuvent aussi être classées selon la classification universelle de toute solution d'un problème donné de la manière suivante:

- **Propriété de sûreté:** Quelque chose de mauvais arriverai jamais.
  - Pour le problème de la section critique la propriété de l'exclusion mutuelle représente une propriété de sûreté.
- **Propriété de vivacité:** Quelque chose de bon arrivera sûrement
  - Pour le problème de la section critique on distingue deux propriétés:
    - **Vivacité globale du système:** Si des processus sont demandeurs d'accès à leurs sections critiques et que la section critique est libre, alors un parmi ces demandeurs finira par accéder à sa section critique.
    - **Vivacité locale ou individuelle de chaque processus:** Tout processus demandeur d'accès à sa section critique finira par y accéder au bout d'un temps fini.

1.2.3 Validation d'une solution

Pour la vérification de la validité d'une solution au problème de la section critique on procède comme suit:

1. Vérifier la propriété de sûreté, dans notre cas la propriété d'exclusion mutuelle. Si la solution ne vérifie pas la sûreté, ce n'est pas la peine de vérifier les deux autres propriétés.
2. Vérifier la propriété de progrès (vivacité globale). Si le système n'est pas vivant alors la famine peut toucher tous les processus. De ce fait on ne vérifie pas la propriété de l'attente limitée.
3. Vérifier la propriété de l'attente limitée.

1.2.4 Solution pour deux processus: Algorithme de Peterson

Pour aboutir à l'algorithme de Dekker vérifiant toutes les propriétés requises nous procédons par un développement graduel de ce dernier.

1.2.4.1 Premier algorithme

Les hypothèses que nous considérons sont:

- Les processus considérés sont P_0 et P_1.
- *turn* est une variable partagée qui indique à qui le tour. Elle est initialisée à 0 ou à 1, pas de différence.
- Le processus P_i peut entrer dans sa section critique si et seulement si $turn = i$.
- Ã la sortie de la section critique, *turn* est affectée par l'autre valeur, afin de permettre à l'autre processus d'accéder à sa section critique.
- Le processus P_i peut être occupé à attendre si le processus P_j est dans sa section critique.

Processus P_0	**Processus** P_1
While (true)	While (true)
{	{
$while(turn \neq 0)\{\}$	$while(turn \neq 1)\{\}$
Section critique	Section critique
$turn := 1$;	$turn := 0$;
Section restante	Section restante
}	}

Discussion:

- *Assure-t-il l'exclusion mutuelle ?*
  Oui, à tout moment la variable *turn* n'a qu'une valeur, et si le processus P_i est dans sa section critique alors $turn = i$.
- *Assure-t-il le progrès ?*
  Non, car l'alternance entre les deux processus est stricte. Si un processus se termine suite à l'exécution de sa section restante alors que l'autre processus a besoin de faire plusieurs cycles, il se retrouvera bloquée dans sa section d'entrée.

1.2.4.2 Deuxième algorithme

Les hypothèses que nous considérons sont:

- On introduit deux variables partagées $flag[0]$ et $flag[1]$ pour indiquer si le processus $P-0$ et le processus P_1 veulent respectivement entrer dans leurs sections critiques respectives.

- Le processus P_i met à *vrai* $flag[i]$ avant d'essayer d'entrer dans sa section critique et à $faux$ $flag[i]$ en sortant de sa section critique.
- Le processus P_i n'entre pas dans sa section critique tant que $flag[j]$ est *vrai*.
- Si un processus veut entrer dans sa section critique à plusieurs reprises, elle n'aura aucun problème, car le drapeau de l'autre processus sera toujours $faux$.

Processus P_0 :	Processus P_1 :
{	{
$flag[0] := true$;	$flag[1] := true$;
$while(flag[1])\{\}$	$while(flag[0])\{\}$
Section critique	Section critique
$flag[0] := false$;	$flag[1] := false$;
Section restante	Section restante
}	}

Discussion:

- *Assure-t-il l'exclusion mutuelle* ?
  Oui, un processus dans sa section critique a vrai dans son $flag$, mais n'entre pas si le $flag$ de l'autre est vrai.
- *Assure-t-il le progrès* ?
  Qu'arrive-t-il suite à la séquence d'instructions:
  - Processus P_0: $flag[0] = vrai$;
  - Processus P_1: $flag[1] = vrai$;

  Pas de progrès car il y a interblocage entre les deux processus.

1.2.4.3 Troisième algorithme

Les hypothèses que nous considérons sont:

- Utilisation de $flag[i]$ pour indiquer le désire d'entrer à la section critique du processus P_i.
- Utilisation de la variable *turn* afin de permettre à l'autre processus d'entrer à sa section critique.

Processus P_0:	Processus P_1 :
{	{
$flag[0] := vrai$;	$flag[1] := vrai$;
$/* P_0$ veut entrer	$/* P_1$ veut entrer
$turn := 1$; $/* P_0$ donne une chance à P_1	$turn := 0$;
while ($flag[1] = vrai$ and $turn = 1$) {}	$/* P_1$ donne une chance à P_0
Section critique	while ($flag[0] = vrai$ and $turn = 0$) {}
$flag[0] := faux$; $/* P_0$ sort	Section critique
Section restant	$flag[1] := faux$; $/* P_1$ sort
}	Section restant
	}

Discussion: La preuve des propriétés de cet algorithme est laissée comme exercice (Voir Exercice 1.2).

1.2.5 Solution générale: Algorithme du boulanger

Origine de l'idée

Durant les crises économiques et la pénurie de blé, le problème de l'approvisionnement en pain apparaît, et des files de personnes s'installent devant les boulangeries pour s'approvisionner de pain. Afin de satisfaire toutes les personnes, une solution consiste à :

- Donner un ticket (numéro) à toute personne qui se présente à la boulangerie. Cela créera un ordre total entre les clients. La satisfaction des clients se fera donc selon cet ordre.
- Limiter le nombre de baguettes de pain que peut prendre un client. Cette limitation peut conduire un client à faire la queue une deuxième fois. Cela est autorisé car ce client aura un numéro de ticket supérieur aux numéros de tous les clients qui l'ont précédé.

Comparaison des deux problèmes

- **Points similaires**
  - Un processus est similaire à un client demandant du pain.
  - Un processus exécutant sa section critique est similaire à un client qui est entrain d'être servi. Cela est dû au fait que le boulanger sert un et un seul client à la fois.
- **Différences**
  - Le nombre de clients est indéfini, alors que le nombre de processus considérés est fixé à n.
  - L'identité d'un client est inconnue, alors que chaque processus est identifié par son pid.

1.2.5.1 Algorithme du boulanger: Version naïve

Les hypothèses que nous considérons sont:

- Utilisation d'une variable globale de type tableau de dimension 1 (vecteur) *Number* de taille n de type entier initialisée à $[0,..,0]$ ($Number : Array[0..n-1] of integer initialisé à [0,..,0]$).
- Un processus P_i demandeur d'accès à sa section critique prendra comme valeur de son ticket $(Max(Number[j] : j \in \{0,..,n-1\}) + 1$.

Le comportement d'un processus P_i quelconque peut être défini par le code suivant:

Var $j : 0..n-1$; Variable locale
Repeat
$Number[i] := (max(Number[0],..,Number[n-1])) + 1$;
For $j := 0$ **to** $n-1$ **do**
Begin
While ($Number[j] \neq 0 and (Number[j] < Number[i]$) **do** $no-op$;
End;
SECTION CRITIQUE
$Number[i] := 0$;
SECTION RESTANTE
Until false;

Discussion:

- La variable *Number* étant partagée entre tous les processus, rien n'empêche que plusieurs processus lisent la même copie de la variable *Number* en même temps. Ils obtiennent donc le même numéro, ce qui peut conduire à l'accès simultané de plusieurs processus dans leurs sections critiques respectives.
- Pour mieux comprendre, considérons le scénario suivant dans lequel $i \neq j$, P_i et P_j ont lu le même vecteur *Number* dont toutes les valeurs sont égales à 0. Supposons que P_j est en exécution (détient le processeur). Dans la boule *while* il trouvera $Number[i] = 0$, il accèdera donc à sa section critique. Cependant si durant cette période P_i se voit affecté le processeur, $Number[j] = 1$ mais $not(Number[j] < Number[i])$. Donc P_i accédera à son tour à sa section critique. Ce qui implique la violation de la propriété d'exclusion mutuelle.

1.2.5.2 Solution correcte

Ingrédients de la solution

- On rajoute un vecteur $Choosing : array[0..n-1] of boolean$ tel que:
  - $Choosing[i] = true$ signifie que le processus P_i est entrain de choisir son numéro de ticket.
- Étant donné que deux processus distincts P_i et P_j peuvent obtenir le même numéro de ticket, la comparaison de ces derniers sera faite par la relation $<$ étendue à l'ensemble des couples $(Number[i], i)$ définie comme suit:
  - $(Number[j], j) < (Number[i], i)$ si et seulement si
    - Soit $Number[j] < Number[i]$ selon la relation $<$ opérant sur les entiers naturels
    - Soit $Number[j] = Number[i]$ et $j < i$

Algorithme final

Ci-dessous l'algorithme du boulanger.

```
Var  Choosing : array[0..n−1] of boolean ;  /* Choosing[i] = true signifie que P_i est
     Number : array[0..n−1] of integer ;    /* entrain de choisir son numéro de ticket
For  k := 0..n−1 do
     Begin
          Choosing[k] := false ; Number[k] := 0 ;  /* Aucun processus n'est en phase
                                                   /* de choix de son numéro de ticket
     End
```

Comportement d'un processus P_i :

```
Var     J : 0..n−1; * Variable locale à P_i
Repeat
        Choosing[i] := true ; /* P_i commence à choisir son numéro
        Number[i] := (max(Number[0], â, Number[n−1])) + 1;
        Choosing[i] := false ; /* P_i a terminé de choisir son numéro
        For j := 0 to n−1 do
        Begin
          While Choosing[j] do { } ;  /* Attendre que P_j termine le choix de son
                                      /* numéro, il se pourrait qu'il soit prioritaire.
          While (Number[j] ≠ 0 and (Number[j], j) < (Number[i], i)) do { } ;
        End;
        SECTION CRITIQUE
        Number[i] := 0;
        SECTION RESTANTE
Until   false;
```

1.3 Conception d'un algorithme pour le problème de la section critique

Tel que nous l'avons souligné dans la section 1.2.2, tous solution au problème de la section critique doit vérifier la propriété d'exclusion mutuelle, la propriété de progrès et la propriété d'attente limitée. Cela passe inévitablement par l'intégration d'une asymétrie départageant les processus demandeurs d'entrée en section critique. La vérification des propriétés requises est conditionnée par la réalisation des propriétés suivantes sur cette asymétrie.

1.3.1 Propriété d'exclusion mutuelle

Pour assurer la propriété de l'exclusion mutuelle on introduit une asymétrie qui partitionne les processus demandeurs en deux parties. Une partie qui contient un singleton (le processus qui sera élu), et l'autre partie contenant les autres processus demandeurs.

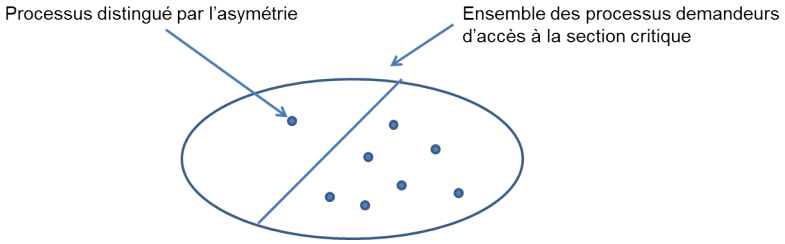

Fig. 1.1 Asymétrie départageant les processus demandeurs

Example 1.1. La relation d'ordre $<$ opérant sur les couples $(Number[i], i)$ dans l'algorithme du boulanger constitue une exemple d'une telle asymétrie.

1.3.2 Propriété de progrès

Pour assurer la propriété de progrès on introduire un mécanisme de perception de l'asymétrie. Le processus distingué par l'asymétrie doit se rendre compte que c'est à lui le tour pour accéder à la section critique.

Example 1.2. A titre d'illustration, considérons un bureau de service pour les malvoyants. L'accès au bureau des clients se fait via une porte par laquelle un seul client peut rentrer à la fois. L'agent de service n'a aucune visibilité sur l'ordre d'arrivée des clients. Dès qu'il termine avec un client il appelle le prochain en disant **au suivant** (Voir Figure 1.2).

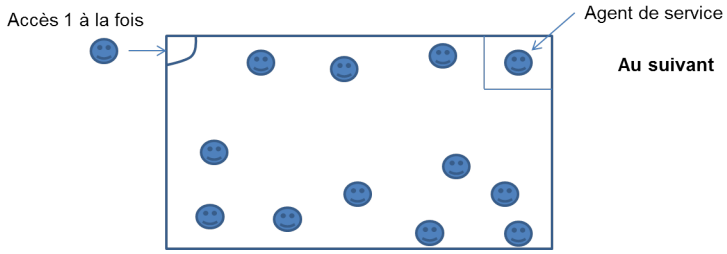

Fig. 1.2 Bureau de services pour les malvoyants

On considérant la propriété consistant à ordonner les clients selon leur ordre d'arrivée, il est claire que cela constitue une asymétrie vérifiant la propriété précédente. En effet parmi les clients il existe un qui est premier. Cependant, ce dernier ne dispose pas de facultés qui lui permettent de s'apercevoir qu'il est le client concerné par le passage à l'agent de service.

L'assurance de la propriété de progrès passera inévitablement l'intégration d'un mécanisme de perception de cette asymétrie.

1.3.3 Propriété d'attente limitée

Pour assurer l'attente limitée on introduire un mécanisme qui rend l'asymétrie dynamique afin qu'elle touche dans le temps tous les processus demandeurs d'accès à la section critique.

Example 1.3. Dans un processus d'élection, le critère du plus àgé parmi les candidats ne vérifie pas la dynamicité de l'asymétrie. Car après le mandat d'élection, si l'ancien élu se présente aux élections une autre fois, il vérifiera toujours le critère du plus âgé.

1.4 Solution système au problème de la section critique

1.4.1 Masquage des interruptions

Nous pouvons solutionner le problème de la section critique en autorisant le masquage des interruptions avant l'accès à la section critique, et leur démasquage après l'exécution de cette dernière. Le schéma de solution pour un processus $P-i$ est le suivant:

Processus P_i
Begin
    DisableInterrupt()
    Section critique
    EnableInterrupt()
End

1.4.2 Inconvénients de la solution

Cette solution est impraticable pour les raisons suivantes:

- Une mauvaise programmation du code de la section critique par le programmeur (boucle infinie dans cette section), aura des conséquences néfaste sur tous le système.
- Ne fonctionne que dans un environnement mono-processeur.
- Les interruptions peuvent être perdues si elles ne reçoivent pas de traitement rapide.
- Un processus en attente de la section critique peut souffrir de famine (Fonction de la politique d'allocation du processeur).

1.5 Solutions matérielles au problème de la section critique

Les solutions matérielles se base sur l'idée d'offrir un mécanisme hardware qui permet de verrouiller l'accès à une variable. Deux instructions, dont l'exécution est atomique, sont disponibles sur les ordinateurs modernes, ce sont :

- $TestAndSet()$

- $Swap()$

1.5.1 Instruction TestAndSet

L'instruction *TestAndSet* est implémentée de manière câblée (Hardware), nous donnons ci-après sa sémantique sous forme d'une fonction comme suit:

1.5.1.1 Sémantique de l'instruction *TestAndSet*

Function *TestAndSet*(**var** *target* : *boolean*):*boolean*;
{
TestAndSet := *target*;
target := *true*;
}

De manière atomique, la fonction *TestAndSet* rend comme résultat la valeur de la variable *target* et positionne la valeur de cette dernière à $true$.

1.5.1.2 Application au problème de la section critique

Pour résoudre le problème de la section critique entre n processus, une variable booléenne $lock$ initialisée à $false$ est utilisée. Comme première tentative on propose la solution suivante:

Comportement d'un processus P_i
Repeat
while $TestAndSet(lock)$ **do** {}
Section critique
$lock := faux$;
Section restante
Until $false$

1.5.1.3 Analyse de la solution

- **Propriété de l'exclusion mutuelle**: Elle est assurée grâce à l'atomicité de l'instruction *TestAndSet*.
- **Propriété de progrès**: Supposons que la section critique est libre, dans ce cas $lock = false$, et qu'il y a des demandeurs d'accès à la section critique. Tous les processus demandeurs tentent d'exécuter l'instruction $While$. Le premier qui appelle la fonction $TestAndSet(lock)$ rentrera à la section critique. A sa sortie il mettra la variable $lock$ à $false$, ce qui libère réellement la section critique. Donc la propriété de progrès est assurée.

- **Propriété de l'attente limitée**: Cette propriété n'est pas vérifiée car rien n'empêche un processus sortant de la section critique de la demander et d'y accéder immédiatement. Ce cycle peut se répéter un nombre indéterminé de fois.

1.5.1.4 Correction de la solution

Nous renvoyons le lecteur à l'exercice 1.5 pour une solution vérifiant toutes les propriétés requises au problème de la section critique.

1.5.2 Instruction Swap

Pareil à l'instruction *TestAndSet*, l'instruction *Swap* est aussi implémentée de manière cÃblée (Hardware), nous donnons ci-après sa sémantique sous forme d'une fonction comme suit:

1.5.2.1 Sémantique de l'instruction *Swap*

```
Function Swap(var a, b:boolean);
var      temp: boolean;
{
         temp := a;
         a := b;
         b := temp;
}
```

De manière atomique, la fonction *Swap* interverti les contenus des paramètres a et b passés comme arguments à cette fonction.

1.5.2.2 Application au problème de la section critique

Pour résoudre le problème de la section critique entre n processus, une variable booléenne partagée $lock$ initialisée à $false$ est utilisée. Comme première tentative on propose la solution suivante:

Comportement d'un processus P_i
Var *key*: *boolean*
Repeat
$key := true$;
Repeat
$Swap(lock, key)$;
Until $key = false$;
Section critique
$lock := false$;
Section restante
Until $false$;

1.5.2.3 Analyse de la solution

- **Propriété de l'exclusion mutuelle**: Elle est assurée grâce à l'atomicité de l'instruction *Swap*.
- **Propriété de progrès**: Supposons que la section critique est libre, dans ce cas $lock = false$, et qu'il y a des demandeurs d'accès à la section critique. La variable *key* de chacun est positionnée à $true$. Tous les processus demandeurs tentent d'exécuter l'instruction $Repeat$. Le premier qui appelle la fonction $swap(lock, key)$ rentrera à la section critique. A sa sortie il mettra $lock$ à $false$, ce qui libère réellement la section critique.
- **Propriété de l'attente limitée**: Cette propriété n'est pas vérifiée car rien n'empêche un processus sortant de la section critique de la demander et d'y accéder immédiatement. Ce cycle peut se répéter un nombre indéterminé de fois.

1.5.2.4 Correction de la solution

Nous renvoyons le lecteur à l'exercice 1.6 pour une solution vérifiant toutes les propriétés requises au problème de la section critique.

1.6 Exercices

Exercice 1.1 *Les chemins de fer dans les hautes montagnes des Andes entre l'état du Pérou et l'état de la Bolivie, sous la forme circulaire, et ayant une section commune située en zone internationale. Malheureusement les deux trains des deux pays entrent occasionnellement en collision, quand ils empruntent simultanément la section commune parce que les conducteurs de trains sont* ***sourds*** *et* ***aveugles****. Afin d'éviter une telle situation les deux conducteurs ont élaboré une méthode d'entrer dans la section commune. Ils ont mis un bol à l'entrée de cette section. A chaque fois qu'un conducteur arrive à l'entrée de la section commune, il arrà*^{a}*te son train et vérifie si le bol contient un caillou. Dans le cas où le bol est vide, le conducteur placera un caillou dans le bol pour indiquer qu'il conduit son train dans la section commune. Une fois sorti de la section commune le conducteur reviendra pour retirer le caillou du bol. Ceci indiquera que la section commune est à présent libre. Ensuite il continue le reste de son circuit. Dans le cas contraire,* ***i.e. le bol contient un caillou****, le conducteur de train fait une sieste suivie d'une nouvelle vérification, jusqu'à ce que le bol devienne vide. Chercher un caillou et le placer dans le bol, conduira le train dans la section commune.*

1. *Expliquez qu'une mauvaise programmation volontaire des autorités péruviennes bloquerait le train bolivien à jamais ? Justifiez votre réponse. Quelle est la propriété qui n'est pas assurée ?*
2. *Le conducteur bolivien n'a pas cru en cette déclaration en indiquant qu'elle est complètement fausse car une telle situation n'est jamais arrivée. Expliquez ? Justifiez votre réponse.*
3. *Malheureusement un jour les deux trains entrent en collision. Expliquez ? Justifiez votre réponse.*
4. *Quelle propriété est non assurée ?*
5. *Après cet accident, un étudiant ayant suivi des cours sur la synchronisation a été consulté pour éviter les accidents définitivement. L'étudiant expliquera que le bol était utilisé de manière erronée. Le conducteur bolivien doit attendre à l'entrée de la section commune jusqu'à ce que le bol devienne vide, conduira son train dans la section commune et à sa sortie, ce conducteur reviendra pour placer un caillou dans le bol. Par contre le conducteur péruvien attendra jusqu'au moment où le bol contiendra un caillou, conduira son train dans la section commune et à sa sortie, ce conducteur reviendra pour retirer le caillou du bol. Cette proposition a évité les accidents. Cependant, avant cette arrangement, le train péruvien roulait deux fois par jour alors que le train bolivien une seule fois par jour. Les autorités péruviennes n'étaient pas contentes. Dites pourquoi ? Justifiez votre réponse.*
6. *Quelle propriété est non assurée ?*
7. *L'étudiant est de nouveau consulté pour résoudre ce problème en épargnant les situations des accidents. Il suggéra l'utilisation de deux bols, où chacun est assigné à un conducteur. A l'arrivée de chaque conducteur à l'entrée de la section commune, il placera dans son propre bol un caillou et puis il vérifie si le bol de*

son partenaire est vide. Dans ce cas il conduira son train dans la section commune et à sa sortie il revient pour retirer le caillou de son propre bol. Cependant, dans la situation inverse il vide son bol et prend une sieste pour ensuite recommencer le processus de nouveau. Cette méthode a bien fonctionné jusqu'au mois de Mai où les deux trains étaient bloqués simultanément à l'entrée de la section commune pour plusieurs siestes. Dites pourquoi ? Justifiez votre réponse.

8. *Quelle propriété est non assurée ?*

Exercice 1.2 *Démontrez que l'algorithme de Deckker vérifie les trois propriétés du problème de la section critique.*

Exercice 1.3 *En admettant une machine Mono-processeur, soient les deux processus P_1 et P_2 de priorité différente. Ces deux processus partagent des variables communes. La régulation d'accès à ces variables est basée sur une approche d'attente active implémentée par l'algorithme de Deckker.*

1. *Que se passe-t-il si le processus P_2 de plus faible priorité est à présent en accès exclusive aux variables partagées alors que le processus P_1 de plus grande priorité tente d'accéder à ces mà^{a}mes variables.*
2. *Proposez un remède efficace à ce problème.*

Exercice 1.4 *Dans cet exercice on se propose d'étudier une solution résolvant le problème de la section critique pour n processus basée sur l'attente active.*

```
Var  Choosing : array[0..n−1] of boolean ;   /∗ Choosing[i] = true signifie que P_i est
     Number : array[0..n−1] of integer ;     /∗ entrain de choisir son numéro de ticket
For  k := 0..n−1 do
     Begin
          Choosing[k] := false ; Number[k] := 0 ;   /∗ Aucun processus n'est en phase
                                                    /∗ de choix de son numéro de ticket
     End
```

Comportement d'un processus P_i :

```
Var      j : 0..n−1; ∗ Variable locale à P_i
Repeat
         L1: Choosing[i] := true ;
         L2: Number[i] := (max(Number[0],â,Number[n−1])) + 1;
         L3: Choosing[i] := false ;
         L4: For j := 0 to n−1 do
         Begin
          L5:  While Choosing[j] do { } ;
          L6:  While (Number[j] ≠ 0 and (Number[j], j) < (Number[i], i)) do { } ;
         End;
         SECTION CRITIQUE
         L7: Number[i] := 0;
         SECTION RESTANTE
Until    false;
```

***Questions**:*

1. *Les différentes solutions résolvant le problème de la section critique se basent principalement sur la création d'une asymétrie permettant de départager les processus demandeurs d'entrée en section critique. Quelles sont les caractéristiques que doit avoir cette asymétrie afin que la solution vérifie les propriétés d'exclusion mutuelle, de déroulement (progrès) et d'attente limitée ?*
2. *On remarque qu'au niveau de la ligne L2, la variable globale Number est partagée. Quelle est la conséquence du partage de cette variable ? Peut-on assurer l'atomicité de cette instruction tout en restant aligné sur le principe de l'attente active ? Justifier votre réponse ?*
3. *Quelle est l'asymétrie adoptée dans cette solution ? Expliquez son principe ?*

4. *Donnez un exemple montrant la nécessité de la ligne L5 (NB : en supprimant cette ligne montrez l'incohérence pouvant survenir lors de l'exécution concurrentes des processus ?).*
5. *Quelle est la sémantique de la ligne L6 ?*
6. *Montrez que l'asymétrie adoptée dans cette solution répond aux besoins attendus ?*

Exercice 1.5 *Proposez une solution au problème de la section critique dans le cas de n processus par l'utilisation de l'instruction TestAndSet.*

Exercice 1.6 *Proposez une solution au problème de la section critique dans le cas de n processus par l'utilisation de l'instruction Swap.*

1.7 Solutions des exercices

Solution exercice 1.1:

Le problème posé peut à[a]tre reporté au problème de la section critique dans lequel les processus sont le parcours de chaque train du circuit avec la traversée de la section critique qui constitue une section critique pour chaque processus. Nous pouvons voir ce système comme l'exécution concurrente des deux processus **Péruvien** et **Bolivien** comme suit:

```
Processus Péruvien                  Processus Bolivien
While (true)                        While (true)
{                                   {
   Traversée circuit Péruvienne        Traversée circuit Bolivien
   Traversée partie commune            Traversée partie commune
}                                   }
```

1. Dans le protocole adopté par les deux partie, le comportement des deux processus est comme suit:

   ```
   Processus Péruvien                  Processus Bolivien
   While (true)                        While (true)
   {                                   {
      Traversée circuit Péruvienne        Traversée circuit Bolivien
      While (not bol vide) { petite sieste }   While (not bol vide) { petite sieste }
      Mettre un caillou dans le bol       Mettre un caillou dans le bol
      Traversée partie commune            Traversée partie commune
      Retirer le caillou du bol           Retirer le caillou du bol
   }                                   }
   ```

   La mauvaise programmation du protocole par les péruviens ressemblerait au code suivant:

   ```
   Processus Péruvien                  Processus Bolivien
   Traversée circuit Péruvienne
   While (not bol vide) { petite sieste }
   Mettre un caillou dans le bol
   While (true)                        While (true)
                                       {
                                          Traversée circuit Bolivien
   {                                      While (not bol vide) { petite sieste }
      Traversée partie commune            Mettre un caillou dans le bol
      Traversée circuit Péruvienne        Traversée partie commune
   }                                      Retirer le caillou du bol
                                       }
   ```

   Dans ce cas, le processus bolivien pourra traverser la partie commune tant que le processus péruvien ne la pas encore traversé. Autrement il restera bloquer éternellement. La propriété qui n'est pas vérifiée est celle de l'attente limitée qui implique la famine pour le processus bolivien.
2. Le conducteur bolivien a considéré que cette situation est absurde car le temps de résidence du train dans la partie commune est fini de temps très court. De ce fait il a changé son comportement selon sa nouvelle affirmation. Cela conduirait au comportement suivant:

   ```
   Processus Péruvien                  Processus Bolivien
   Traversée circuit Péruvienne
   While (not bol vide) { petite sieste }
   Mettre un caillou dans le bol
   While (true)                        While (true)
   {                                   {
      Traversée partie commune            Traversée circuit Bolivien
      Traversée circuit Péruvienne        Traversée partie commune
   }                                   }
   ```

3. Évidement, la collision s'est produite car aucune protection n'a été considérée quant à l'accès à la section commune. Remarquons que la collision peut ne jamais se produire.

4. La propriété qui n'est pas assurée est celle de l'exclusion mutuelle. En d'autres termes, la sûreté qui stipule que quelque chose de mauvais n'arrivera jamais.
5. Le nouveau protocole est comme suit: Initialement on peut admettre que le bol contient un caillou.

```
Processus Péruvien                  Processus Bolivien
While (true)                        While (true)
{                                   {
   Traversée circuit Péruvienne        Traversée circuit Bolivien
   While (bol vide) { petite sieste }  While (not bol vide) { petite sieste }
   Traversée partie commune            Traversée partie commune
   Retirer le caillou du bol           Mettre un caillou dans le bol
}                                   }
```

   Il est claire que cette solution fait rentrer les trains bolivien et péruvien avec une alternance stricte. Ce qui implique que l'accès à la section commune se fera selon la plus petite fréquence d'accès des deux trains. Dans ce cas le train bolivien.
6. La propriété qui n'est pas vérifiée est celle du progrès. Car l'accès à la section critique d'un train dépend de du train qui est dans sa section restante.
7. Le nouveau protocole est comme suit: A l'état initial les deux bols sont vides.

```
Processus Péruvien                  Processus Bolivien
While (true)                        While (true)
{                                   {
   Traversée circuit Péruvienne        Traversée circuit Bolivien
α  mettre un caillou dans bolpérou  β  mettre un caillou dans bolbolivie
   If (not bolbolivie vide) {          If (not bolpérou vide) {
   petite sieste; Goto α }             petite sieste; Goto β }
   Traversée partie commune            Traversée partie commune
   Retirer le caillou du bolpérou      Retirer le caillou du bolbolivie
}                                   }
```

   Ce cas d'attente peut arriver si les deux conducteurs arrivent à l'instruction if en mà^{a}me temps et prennent une sieste de mà^{a}me durée à chaque fois qu'ils trouvent les bols non vides. Ce cas est très improbable.
8. La propriété qui n'est pas vérifiée est celle du progrès. Celle dernière fait partie des propriétés dites de vivacité qui stipulent que quelque chose de bon arrivera sûrement.

Solution exercice 1.2:

Rappelons l'algorithme de Dekker et les hypothèses considérées:

- Utilisation de $flag[i]$ pour indiquer le désire d'entrer à la section critique du processus P_i.
- Utilisation de la variable *turn* afin de permettre à l'autre processus d'entrer à sa section critique.

```
Processus P0:                       Processus P1:
{                                   {
   flag[0] := vrai;                    flag[1] := vrai;
    /* P0 veut entrer                   /* P1 veut entrer
   turn := 1; /* P0 donne une chance à P1  turn := 0;
                                        /* P1 donne une chance à P0
   while (flag[1] = vrai and turn = 1) {}  while (flag[0] = vrai and turn = 0) {}
   Section critique                    Section critique
   flag[0] := faux; /* P0 sort        flag[1] := faux; /* P1 sort
   Section restant                     Section restant
}                                   }
```

Discussion:

- *Assure-t-il l'exclusion mutuelle* ?
  On fait une preuve par l'absurde. Supposons que les deux processus réussissent à accéder leurs sections critiques en mà^{a}me temps.

- **Cas 1:** P_i trouve la condition $flag[1-i] = vrai\, and\, turn = (1-i)$ fausse avant que P_{1-i} ne positionne sont $flag$ à vrai. Dans ce cas lorsque P_{1-i} exécute la boucle while il a déjà positionné $turn$ à i et P_i en section critique (donc $flag[i] = vrai$). Ce qui est absurde.
- **Cas 2:** Les deux processus ont positionné leur $flag$ à vrai avant qu'elles arrivent à la boucle while. Dans ce cas $turn \neq 1$ et $turn \neq 0$, ce qui est absurde.

- *Assure-t-il le progrès* ?
  Soit l'hypothèse que la section critique est libre. Deux cas se présentent.
  - **Cas 1:** Seule le processus $P-i$ demande l'accès à sa section critique. Dans ce cas $turn = 1-i$ et $Flag[1-i] = faux$. $P-i$ réussira donc à franchir la boucle while et entre dans sa section critique.
  - **Cas 2:** Les deux processus sont demandeurs d'accès à la section critique. Donc $flag[i] = flag[1-i] = vrai$. Quelque soit l'ordre dans lequel les instructions $turn := 1-i$ et $turn := i$ sont exécutées, la variable $turn$ prendra la valeur 0 ou 1. Donc l'une des deux conditions se trouvera fausse et l'un des processus réussira à accéder à sa section critique.

- **Assure-t-il l'attente limitée?**
  Nous montrons que si P_i est demandeur (bloquée au niveau de la boucle $While$) alors que P_{1-i} est dans sa section critique. On montre que mà[a]me si P_{1-i} détient le processeur et qu'il quitte sa section critique et redemande une autre fois l'accès à sa section critique, elle ne réussira pas à y accéder. Dans ce cas elle positionne son $flag$ à vrai et $turn$ à i alors que $flag[i]$ est vrai. Dans ce cas P_{1-i} restera bloqué dans l'instruction *while*, et dès que P_i se voit affecté le processeur, sa condition d'attente étant fausse, il accèdera donc à sa section critique.

Solution exercice 1.3:

1. Dans le cas oà le processus P_2 de plus faible priorité est en section critique et le processus P_1 de plus grande priorité tente d'accéder à la sienne, ce dernier passera à l'état **En exécution** et monopolisera le processeur. Les deux processus resterons bloqués pour toujours.
2. Comme solution, on peut changer la priorité du processus qui accède sa section critique pour la rendre la plus élevée. De ce fait, tout processus accédant sa section l'exécutera en totalité avant qu'il ne soit interrompu, évitant ainsi des blocages par des attentes actives.

Solution exercice 1.4:

1. Les caractéristiques que doit avoir cette asymétrie pour l'assurance des propriétés requises sont:
   - Pour assurer la propriété de l'exclusion mutuelle l'asymétrie doit partitionner les processus demandeurs en deux parties. Une partie qui contient un singleton

(le processus qui sera élu), et l'autre partie contenant les autres processus demandeurs.
- Pour assurer la propriété de progrès on introduire un mécanisme de perception de l'asymétrie. Le processus distingué par l'asymétrie doit se rendre compte que c'est à lui le tour pour accéder à la section critique.
- Pour assurer l'attente limitée on introduire un mécanisme qui rend l'asymétrie dynamique afin qu'elle touche dans le temps tous les processus demandeurs d'accès à la section critique.

2. Le partage de la variable *Number* implique la possibilité que plusieurs processus concurrent exécutant la ligne $L2$ obtiennent la mà[a]me valeur. L'atomicité de cette instruction ne peut pas à[a]tre assurée tout en restant aligné au principe de l'attente active car cela impliquera l'utilisation d'une autre variable partagée qui doit à[a]tre protégée à son tour.
3. La relation d'ordre $<$ opérant sur les couples $(Number[i], i)$ dans l'algorithme du boulanger constitue l'asymétrie requise.
4. En considérant l'algorithme du boulanger dans lequel la ligne $L5$ est supprimée, supposons que que deux processus P_i et P_j sont demandeurs d'accès à la section critique et qu'ils ont obtenu le mà[a]me numéro avec l'hypothèse que $i < j$ et que P_i n'a pas encore écrasé son ancienne valeur ($Number[i]$ est égale à 0). Supposons aussi que le processus P_j exécute en premier la ligne $L6$. Il est claire que le processus P_j accède à sa section critique. Si à ce moment le processus P_i exécute la ligne $L6$, ce dernier se voit prioritaire par rapport au processus P_j et accède à sa section critique. D'oà la violation de la propriété d'exclusion mutuelle.
5. Dans la ligne $L6$, le processus P_i trouvant le processus P_j est demandeur et qu'il est plus prioritaire que le lui rendre dans une attente active jusqu'à ce que ce dernier exécute et sorte de sa section critique. Mà[a]me s'il deviendra encore demandeur, son numéro sera supérieur à celui de P_i. Le processus P_i arrivera donc à explorer l'état des autres processus.
6. Pour l'asymétrie adoptée dans l'algorithme du boulanger nous pouvons remarquer que:

   - Pour les processus demandeurs d'accès à la section critique, le plus petite élément donné par la relation $<$ est unique.
   - Dans le cas oà plusieurs processus sont demandeurs d'accès à la section critique, tous ces processus seront bloqués au niveau de la ligne $L6$ sauf le processus qui à la plus petite valeur $(Number[i], i)$. Ce qui implémente la perception de l'asymétrie.
   - Un processus sortant de la section critique et redemandant encore une fois l'accès à sa section critique obtiendra forcément un numéro plus grand que ceux des processus demandeurs. Ce qui implémente la dynamicité de l'asymétrie.

Solution exercice 1.5:

Voici une solution au problème de la section critique pour n processus utilisant l'instruction *TestAndSet* que nous expliquerons ci-après.

Déclaration des variables globales

Var Waiting: Array[1 .. n] of Boolean; initialisée à [false, ... , false]
lock: Boolean; initialisée à false

Comportement d'un processus P_i

Var j : $integer$;

Do {	L1
Waiting[i] = true;	L2
Key = true;	L3
While (waiting[i] *and* key) { key = TestAndSet(lock); };	L4
Waiting[i] = false;	L5
Section Critique	
j = (i+1) mod n;	L6
While (($j \neq i$) *and* (waiting[j]=false)) $\{j := (j+1) \, mod \, n;\}$;	L7
If ($j = i$) $\{lock := false;\}$;	L8
Else $\{Waiting[j] := false\}$;	L9
Section restante	
} **While** (true)	

L'idée de la solution et de rajouter une autre variable d'état partagée, nommée $Wating$, qui est un vecteur tel que $Waiting[i]$ signifie le processus P_i désire accéder à sa section critique (Ligne L2). Cette fois-ci la compétition pour l'accès à la section critique peut se présenter selon deux cas:

- **Cas 1:** La section critique est libre. Dans ce cas l'accès à la section critique sera pour le processus qui exécute en premier l'instruction atomique $TesAndSet$. Pour ce dernier, sa variable locale Key sera positionnée à $false$ et rentrera en conséquence à sa section critique (Ligne L4).
- **Cas 2:** La section critique est occupée par un processus, soit P_i ce dernier. Dans ce cas, ce dernier parcourra le vecteur $Waiting$ à partir de la position $(i+1) \, mod \, n$ (ligne L6), et positionnera la première case j rencontrée dont $Waiting[j] = true$ à $false$ (ligne L9). Le processus P_j pourra ainsi accéder à sa section critique. De cette manière tous les processus demandeurs accèderons à la section critique. Dans le cas oà le processus P_i est le dernier demandeur, le parcoure du vecteur $Waiting$ se terminera par critère $i = j$. Dans ce cas le processus P_i positionnera la variable globale $lock$ à false (ligne L8).

Solution exercice 1.6:

L'idée de la solution présentée ci-après est similaire à celle de l'exercice 1.5.

Déclaration des variables globales

Var Waiting: Array[1 .. n] of Boolean; initialisée à [false, ... , false]
lock: Boolean; initialisée à false

Comportement d'un processus P_i
Var j : $integer$;
Do {
$Waiting[i] := true$;
$Key := true$;
While (waiting[i] *and* key) { $Swap(lock, key)$; };
$Waiting[i] := false$;

Section Critique

$j := (i+1) \, mod \, n$;
While (($j \neq i$) *and* ($waiting[j] = false$)) {$j := (j+1) mod n$;};
If ($j = i$) {$lock := false$;};
Else {$Waiting[j] := false$} ;

Section restante

} **While (true)**

Chapter 2
Synchronisation par les sémaphores

2.1 Introduction

Les solutions hardware sont difficiles à utiliser par les programmeurs. Comme alternative on peut envisager de donner la possibilité au programmeur de masquer les interruptions avant l'accès à la section critique et de les autoriser à la sortie de la section critique. Le comportement d'un processus P_i ressemblerait au code suivant:

Comportement d'un processus P_i
Repeat
disable interrupt
Section critique
enable interrupt
Until (*False*)

Une telle solution, malgré sa simplicité, présente plusieurs inconvénients à savoir:

- Ne fonctionne que dans un environnement mono-processeur.
- Un processus en attente de la section critique peut souffrir de famine (Fonction de la politique d'allocation du processeur).
- Une mauvaise programmation de l'application peut avoir des conséquences graves sur tout le système. A titre d'exemple, la présence d'une boucle infinie dans une section critique peut bloquer le système en entier.

Les raisons ci-dessus on conduit en la proposition d'une solution système implémentant l'outil sémaphore.

Definition 2.1. Un sémaphore **S** est une variable entière qui est, sauf à l'initialisation, accédée via les opérations standards **wait()** et **signal()**. Ces opérations sont invoquées à travers des appels système.

A l'origine l'opération *wait()* s'appelait *P* (abréviation du mot holondais *proberen*, pour tester); *signal()* qui à l'origine s'appelait *V* (abréviation de *verhogen*, pour incrémenter). Dans la suite, on utilisera indifféremment les deux appellations.

Definition 2.2. L'opération **wait()** sur un sémaphore *S* Peut être définie comme suit:

```
wait(S) {
        while S ≤ 0 { } ;
        S − −;
        }
```

Definition 2.3. L'opération **signal()** sur un sémaphore *S* Peut être définie comme suit:

```
signal(S) {
          S + +;
          }
```

L'atomicité des opérations d'incrémentation, de décrémentation et de test sur la variable *S* est prise en charge par le système d'exploitation.

Les sémaphores implémentés avec l'attente actives sont appelés **les spining sémaphores**. Il est clair que l'utilisation des spining sémaphores sur une machine mono-processeur aurait des conséquences néfastes quant aux performances du système.

2.2 Implémentation des sémaphores sans attente active

Afin d'éviter une perte du temps processeur dans des attentes actives inutiles, un sémaphore *S* peut être vu comme un objet derrière lequel des processus ayant appelé une opération *wait(S)* bloquante sont bloqués. Le réveil d'un de ces processus sera fait par l'appel de l'opération *signal(S)* par un autre processus. Une implémentation d'un tel sémaphore peut être comme suit:

```
Structure semaphore
                    {
                      int count;
                      ProcessQueue queue ;
                    }
```

L'opération *wait()* sur le sémaphore *S* est définie comme suit:

```
Void wait(semaphore S)
    {
    disable interrupt ;
    S.count − −
     if (s.count < 0) /* La condition de blocage est satisfaite
        {
        S.queue.insert; /* Le processus est inséré dans la file
        Block() /* Le processus est bloqué derrière le sémaphore S
        }
    enable interrupt
    }
```

L'opération *signal()* sur le sémaphore *S* est définie comme suit:

```
Void signal(semaphore S)
    {
    disable interrupt ;
    S.count++
    if (s.count ≤ 0) /* Il y a au moins un processus bloqué
       {
       S.queue.remove(P); /* Un des processus dans la file
       wakeup(P) /* Le processus P est réveillé
       }
    enable interrupt
    }
```

On remarque que dans cette solution le masquage et le démasquage des interruptions sont encapsulés dans les opérations systèmes dont les codes ne présentent aucun risque de dysfonctionnement du système.

2.3 Types de sémaphores

2.3.1 Sémaphore binaire

Son compteur est initialisé par la valeur 1. Ce type de sémaphore est utilisé pour la résolution du problème de la section critique. Une solution à ce problème pour n processus peut être comme suit:

```
Var mutex: semaphore init(1) /* mutex pour mutual exclusion
Comportement d'un processus P_i
 Repeat
                Section restante
                wait(mutex); /* Section d'entrée
                Section critique
                signal(mutex); /* Section de sortie
 Until (False)
```

2.3.2 Sémaphore de comptage

Son compteur est initialisé à une valeur n. En général n est le nombre d'instances d'une ressource.

- Chaque processus qui a besoin d'une instance exécute l'opération $Wai()$ sur le sémaphore.
- Lorsque toutes les instances sont occupées, le processus se bloque derrière le sémaphore. Un processus qui libère une instance de la ressource exécute

l'opération *Signal*() sur le sémaphore. Il débloquera éventuellement un processus bloqué derrière ce sémaphore.

2.4 Exercices

Exercice 2.1 *Percevoir le wait et signal des spining sémaphore comme section critique, nécessitent une application automatique de mutexbegin et mutexend, protocole d'entrer $< entry-section >$ et de sortie $< exit-section >$ d'une section critique. Cette application directe peut s'avérer désastreuse.*

1. *Donnez la solution appliquant directement mutexbegin et mutexend rendant wait et signal indivisibles en les considérant comme une section critique.*
2. *Que se passe t-il lors de l'utilisation d'un sémaphore binaire de valeur initiale égale à zéro, sachant qu'un appel à wait précède l'appel de signal.*
3. *corrigez la proposition défectueuse précédente en utilisant toujours mutexbegin et mutexend de l'attente active.*

Exercice 2.2 *Proposez une solution utilisant exclusivement des sémaphores binaires au problème d'allocation/libération d'une ressource partagée à plusieurs unités identiques. Les unités de cette ressource sont allouées et libérées une par une. Les processus en attente de la disponibilité d'une unité doivent être servis selon l'ordre de leurs demandes. On suppose qu'il y a n processus dans le système identifié chacun par son indice i tel que* $0 \leq i \leq n-1$.

Exercice 2.3 *La solution d'une paire de producteur/consommateur à l'aide des sémaphores, déjà étudiés auparavant, ressemble au code suivant :*

Var *nplein: Sémaphore initialisé à* 0
nvide: Sémaphore initialisé à n

Processus *Producteur*	***Processus*** *Consommateur*
Repeat	***Repeat***
Produire un message ;	*wait(nplein);*
Wait(nvide) ;	*Prelever un message;*
Deposer le message;	*signal(nvide);*
Signal(nplein) ;	*Consommer message;*
Until *false;*	***Until*** *false;*

Soient les propriétés suivantes des sémaphores:

1. *Un sémaphore ne peut être initialisé à une valeur négative, mais sa valeur peut devenir négative après un certain nombre d'opérations wait.*
2. *Soit $np(s)$ le nombre d'appels de la primitives $wait(s)$ par les processus utilisant le sémaphore s.*
3. *$nv(s)$ le nombre d'appels de la primitives $signal(s)$ par les processus utilisant le sémaphore s.*
4. *$e0(s)$ et $e(s)$ sont respectivement la valeur initiale et la valeur courante du sémaphore s.*
5. *$e(s) = e0(s) – np(s) + nv(s)$.*
6. *Soit $nf(s)$ le nombre de processus qui ont franchi la primitive $wait(s)$, c'est-à-dire qui, ou bien ils n'ont pas été bloqués par celle-ci, ou bien ils ont été bloqués mais débloqués depuis ; à tout instant on a $nf(s) \leq np(s)$.*

Il faut remarquer que l'exécution des primitives wait et signal laisse invariant la relation $nf(s) = min(np(s), e0(s) + nv(s))$. A l'aide de ces propriétés démontrez que le code du producteur/consommateur assure:

- *L'absence de l'interblocage.*
- *On n'opère jamais simultanément sur le même message, lorsque les messages sont consommés dans l'ordre de leur production et que le buffer est un tableau linéaire circulaire.*

Exercice 2.4 *La circulation au carrefour de deux voies est réglée par des signaux lumineux (feu Vert/Rouge). On suppose que les voitures traversent le carrefour en ligne droite et que le carrefour peut contenir au plus une voiture à la fois. On impose les conditions suivantes :*

- *Toute voiture se présentant au carrefour le franchit en un temps fini.*
- *Les feux de chaque voie passent alternativement du vert au rouge, chaque couleur étant maintenue pendant un temps fini.*
- *Les arrivées sur les deux voies sont réparties de façon quelconque. Le fonctionnement de ce système peut être modélisé par un ensemble de processus parallèles :*
  - *Un processus P qui exécute la procédure changement de commande des feux.*
  - *Un processus est associé à chaque voiture.*
  - *La traversée du carrefour par une voiture qui circule sur la voie i (i = 1,2) correspond à l'exécution d'une procédure Traverséei par le processus associé à cette voiture.*

1. *Donnez les différents types de processus, leur nombre, leur mode de synchronisation (inter-processus de même type et inter-processus de types distincts).*
2. *A quel paradigme de synchronisation peut-on rapprocher ce problème ?*
3. *On demande décrire le programme changement ainsi que les procédures Traversée1 et Traversée2.*
   Remarque: *Le processus P doit attendre que la voiture engagée dans le carrefour en soit sortie avant d'ouvrir le passage sur l'autre voie.*

Exercice 2.5 *Écrire les procédures d'accès à un objet de type barrière dont le fonctionnement est le suivant :*

1. *La barrière est fermée à son initialisation.*
2. *La barrière s'ouvre lorsque N véhicules sont bloqués sur elle.*

Définition de la barrière:

```
Type  barrière = record
                   Sema1 : Sémaphore, /*Sémaphore de section critique
                   Sema2 : Sémaphore, /*Sémaphore de blocage sur la barrière
                   Count : Integer, /*Compteur
                   Maximum : Integer, /*Valeur déclenchant l'ouverture
End ;
```

1. *Complétez le code de la procédure d'initialisation :*

```
Var B : Barrière
Procédure Initialisation
Begin
        Init(B.Sema1, . . . ) ;
        Init(B.Sema2, . . . ) ;
        B.Count := . . . ;
        B.Maximum := . . . ;
End ;
```

2. *Complétez le code de la procédure d'attente donné ci-dessous :*

```
Procédure Wait(B :Barrière)
Var     Dowait : boolean ;
        I : Integer;
Begin
        Dowait := true;
        . . . . . ;
        B.Count := B.Count +1;
        If B.Count = B.maximum then
            Begin
                For (I = 1 to B.Maximum – 1) do ( . . . . . );
                B.Count := 0;
                Dowait := false;
            End;
        . . . . . ;
        If Dowait then . . . . . ;
End;
```

2.5 Solutions des exercices

Exercice 2.1
MutexBegin et *MutexEnd* associées aux sémaphores de l'attente active peuvent êtres réalisées entre autres grâce à l'algorithme du boulanger dans le cas de n processus ou celui de Deckker lorsque le système se compose de deux processus. Rappelons que les deux algorithmes vérifient les trois propriétés de la section critiques. Un sémaphore de l'attente active est défini par:

S **: entier ;**

La primitive wait(S)	**La primitive signal(S)**
while($S \leq 0$) do no-op ; $S := S - 1$;	$S := S + 1$;

1. Puisque le sémaphore est variable partagée en lecture/écriture, les opérations wait et signal sont donc des sections critiques. Ces deux primitives peuvent donc être rendues atomiques en appliquant d'une manière naïve **MutexBegin** et **MutexEnd** comme $< entry - section >$ et $< exit - section >$ respectivement. Le code qui en résulte est donné ci-après :

   S **: entier ;**

La primitive wait(S)	**La primitive signal(S)**
MutexBegin while($S \leq 0$) do no-op ; $S := S - 1$; **MutexEnd**	**MutexBegin** $S := S + 1$; **MutexEnd**

2. On observe alors une boucle sans fin de la primitive wait lorsque $s = 0$ car toute activation de signal est impossible.
3. Corriger la version précédente consiste à remodeler la primitive wait de sorte à ce que l'opportunité d'exécuter un signal en soit possible en libérant systématiquement l'accès au sémaphore dont la valeur actuelle est 0. Ci-après est la version corrigée correspondante. On ne modifie que le code de la primitive wait, le reste est inchangé :

   S **: entier ;**
   La primitive wait(S)
   Var Blocked: Booléen
   Begin
   blocked := true ;
   Repeat
   MutexBegin
   If(S0) Then S := S-1 ; Blocked := false ;
   MutexEnd
   Until (not Blocked)
   End

On remarque la variable Blocked est locale à la procédure wait, de ce fait, chaque appel à cette procédure utilise sa propre instance de cette variable.

Exercice 2.2

Soient les vaiables suivantes:

- **file** : file d'attente des identités des processus en attente. Cette file est dotée de la procédure **put(id : identité d'un processus)** et **get()** qui renvoie l'identité du premier processus de la file ou -1 lorsque la file est vide.
- **Mutex** : semaphore ; initial(0) ;
- **Sempriv** : array[0..n-1] of semaphore ; initial(0....0) ;
- **available** : entier ; initial(N) ;

La variable **Sempriv** est un tableau de sémaphores privés. En effet pour un indice $0 \leq i \leq n-1$, seul le processus P_i peut exécuter l'opération **wait(Sempriv[i])** à travers l'appel de la procédure **allouer()** par le processus P_i.

```
Procedure allouer()
Var blocked : boolean ;
Begin
        Wait(Mutex) ;
        blocked := (available==0) ;
        if (blocked) Then put(me) ;
        Else available-1 ;
        Signal(Mutex) ;
        If (blocked) Then Wait(Sempriv[me]) ;
End ;
Procedure liberer()
Var id: entier
Begin
        Wait(Mutex) ;
        available := available+1 ;
        id := get() ;
        if (id ≠ −1) Then
        Begin
                avaialble := available-1 ;
                Signal(Sempriv[id]) ;
        End ;
        Signal(Mutex) ;
End ;
```

Exercice 2.3

```
Processus Producteur
Repeat
        Produire un message ;
        CheckPoint1: Wait(nvide) ;
        Deposer le message;
        Signal(nplein) ;
Until   false;
```

```
Processus Consommateur
Repeat
        CheckPoint2: wait(nplein);
        Prelever un message;
        signal(nvide);
        Consommer message;
Until   false;
```

1. Procédons par absurde. Intuitivement parlant l'état d'interblocage est possible seulement si l'état du buffer complètement vide est confondu avec son état complètement plein. D'une manière analytique : les deux processus sont bloqués respectivement dans *CheckPoint1* et *CheckPoint2* d'où les équations suivantes : L'aboutissement au résultat peut se faire de deux méthodes:

   - **Première méthode:** Seul le producteur exécute les opérations $wait(nvide)$ et $signal(nplein)$. Son blocage à *CheckPoint1* implique:

     $$np(nvide) = nv(nplein) + 1 \; et \; e(nvide) = -1$$

     De manière similaire, seul le consommateur exécute les opérations $wait(nplein)$ et $signal(nvide)$.Son blocage à *CheckPoint2* implique:

     $$np(nplein) = nv(nvide) + 1 \; et \; e(nplein) = -1$$

     La propriété $e(s) = e0(s) - np(s) + nv(s)$ en jonction avec l'hypothèse de blocage des deux processus implique:

     $$-1 = nv(nplein) - np(nplein) \; which implies \; nv(nplein) = np(nplein) - 1$$

     $$-1 = n + nv(nvide) - np(nvide) \; which implies \; nv(nvide) = np(nvide) - n - 1$$

     Par remplacement de $nv(nplein)$ et $nv(nvide)$ par leurs expressions respectives on déduit que:

     $$np(nplein) = nv(nvide) + 1 \; et \; np(nvide) = np(nplein)$$

     Donc $np(nvide) = np(nplein) + 1 - 1$ et $np(nplein) = np(nvide) - n - 1 + 1$, de ce fait $np(nplein) = np(nplein) - n$ cela implique $n = 0$, ce qui est absurde avec l'hypothèse que $n > 0$. L'hypothèse de l'interblocage est donc fausse.
   - **Deuxième méthode:** Nous pouvons utiliser l'invariant

     $$nf(s) = min(np(s), e0(s) + nv(s))$$

     D'où les équations:

     $$\begin{array}{ll} nf(nvide) = nv(nplein) & nf(nplein) = nv(nvide) \\ nf(nvide) = n + nv(nplein) & nf(nplein) = 0 + nv(nplein) = nv(nplein) \end{array}$$

     Donc $nv(nplein) = nf(nvide) = n + nf(nplein) = n + nv(nvide) = nv(nvide)$, ce qui est impossible car n est non nulle.

2. Montrons que les deux processus n'opèrent jamais sur la même case. Soient les deux codes suivants:

Processus Producteur	**Processus** Consommateur
Repeat	**Repeat**
Produire un message ;	wait(nplein);
Wait(nvide) ;	CheckPoint4: Prelever un message;
CheckPoint3: Deposer le message;	signal(nvide);
Signal(nplein) ;	Consommer message;
Until false;	**Until** false;

On procède par l'absurde. On suppose donc que les deux processus sont aux points d'exécution respectifs CheckPoint3 et CheckPoint4 simultanément. On a donc:

$$nf(nvide) = 1 + nv(nplein) \; et \; nf(nplein) = 1 + nv(nvide)$$

De l'invariant des sémaphores nous déduisons les inégalités suivantes:

$$nf(nvide) \leq n + nv(nvide) \; et \; nf(nplein) \leq nv(nplein)$$

D'où:

$$1 + nv(nplein) \leq n + nv(nvide) \; et \; 1 + nv(nvide) \leq nv(nplein)$$

D'où:

$$nv(nplein) - nv(nvide) \leq n - 1 \; et \; 1 \leq nv(nplein) - nv(nvide)$$

D'où:

$$1 \leq nv(nplein) - nv(nvide) \leq n - 1$$

Supposons maintenant que les deux processus opèrent sur la même case. On a $tête = (tête_0 + nf(nplein)) \, mod \, n$ et $queue = (queue_0 + nf(nvide)) \, mod \, n$. Or $tête_0 = queue_0 = 0$ et selon la supposition que les deux processus opèrent sur la même case, on déduit que $tête = queue$, ce qui revient à $nf(nplein) = nf(nvide)$. Or substituant $nf(nvide) = 1 + nv(nplein)$ et $nf(nplein) = 1 + nv(nvide)$. Donc $nv(nplein) = nv(nvide)$, ce qui revient à écrire $nv(nplein) - nv(nvide) = 0$. Ce qui contredit l'inégalité précédente. On conclue donc que l'exclusion mutuelle est garantie.

Exercice 2.4

1. Éléments conceptuels du problème posé:

   - **Types de processus:** On distingue trois types de processus.
     - Type 1: Correspond aux processus exécutant la procédure **Traversée1**. Leur nombre est indéterminé car chaque processus correspond à une voiture arrivant sur la voie 1.
     - Type2 : Correspond aux processus exécutant la procédure **Traversée2**. Leur nombre est indéterminé car chaque processus correspond à une voiture arrivant sur la voie 2.

  - Type3 : C'est le type du processus *P* correspondant à l'exécution de la procédure **Changement**. On a un seul processus de ce type.
- **Synchronisation des processus:**
  - **Synchronisation entre les processus de type 1:** Deux règles de synchronisation sont à assurer. Les processus de type 1 traversent le carrefour à tour de rôle d'une part, et la traversée du carrefour se fait en exclusion mutuelle entre ces processus. Nous pouvons donc rapprocher ces processus à un ensemble de rédacteurs qui utilisent un fichier commun.
  - **Synchronisation entre les processus de type 2:** Même chose que pour les processus de type 1.
  - **Synchronisation des processus de type 1 avec le processus de type 2:** L'exclusion mutuelle est imposée entre les processus de chacun des types (type 1 et type 2).
  - **Synchronisation entre le processus *P* avec les autres processus:** Le processus *P* travail en arrière plan en assurant l'alternance de la priorité entre les processus de type 1 et ceux de type 2.

2. Après la décomposition précédente, on peut rapprocher le problème à celui de rédacteurs/rédacteurs avec priorité alternée.
3. On va procéder par étape:

   - Pour assurer l'exclusion mutuelle entre les processus de type 1 (respectivement type 2), on utilise un sémaphore d'exclusion mutuelle **CarrefourLibrePourVoie1** (respectivement **CarrefourLibrePourVoie2**). Les codes respectives des procédures **Traversée1** et **Traversée2** sont:

Procédure Traversée1	**Procédure Traversée2**
Begin	**Begin**
wait(CarrefourLibrePourVoie1)	wait(CarrefourLibrePourVoie2)
Traverser le carrefour	Traverser le carrefour
signal(CarrefourLibrePourVoie1)	signal(CarrefourLibrePourVoie2)
End	**End**

   - Nous allons maintenant assurer l'accès concurrents des processus des deux types chacun sur sa voie. Nous devant choisir une hypothèse d'initialisation des feux pour qu'une seule voie soit ouverte. Soit par exemple la voie 1. Donc, à l'initialisation la voie 2 est fermée. De ce fait on associe à la voie 2 un sémaphore de blocage (initialisé à 0), nommé **Voie2Ouverte** et un sémaphore d'exclusion mutuelle (initialisé à 1) pour la voie 1, nommé **Voie1Ouverte**. En effet, la sémantique d'un tel sémaphore est que un et un seul véhicule doit s'apprêter à traverser le carrefour. Les procédures précédentes sont changées

comme suit:

Procédure Traversée1	**Procédure Traversée2**
Begin	**Begin**
wait(CarrefourLibrePourVoie1)	wait(CarrefourLibrePourVoie2)
wait(Voie1Ouverte)	wait(Voie2Ouverte)
Traverser le carrefour	Traverser le carrefour
signal(Voie1Ouverte)	signal(Voie2Ouverte)
signal(CarrefourLibrePourVoie1)	signal(CarrefourLibrePourVoie2)
End	**End**

- Revenons maintenant à la procédure **Changement**. C'est une procédure cyclique qui à pour rôle de changer périodiquement l'état des voies. Fermer la voie ouverte et ouvrir la voie fermée. A cet effet on a besoin de connaître la voie qui est actuellement ouverte. Pour cela on utilise la variable booléenne **Voie1libre** initialisée à **true** selon l'hypothèse d'initialisation considérée. La période d'attente est définie par la constante **Période** initialisée par une valeur adéquate. Ainsi, le code de la procédure **Changement** est comme suit:

```
Procédure Changement
Begin
    Repeat
        Sleep(Période);
        If  Voie1Libre Then
            Begin
                wait(Voie1Ouverte);   /* Fermeture de la voie 1
                signal(Voie2Ouverte); /* Ouverture de la voie 2
                Voie1libre := false;
            End
        Else
            Begin
                wait(Voie2Ouverte);   /* Fermeture de la voie 2
                signal(Voie1Ouverte); /* Ouverture de la voie 1
                Voie1libre := true;
            End
    Until   false
End
```

Exercice 2.5

1. Procédure d'initialisation :

```
Var B : Barrière
Procédure Initialisation
Begin
        Init(B.Sema1, 1 ) ;
        Init(B.Sema2, 0 ) ;
        B.Count := 0 ;
        B.Maximum := N ;
End ;
```

2. Procédure d'attente :

```
Procédure Wait(B :Barrière)
Var     Dowait : boolean ;
        I : Integer;
Begin
        Dowait := true;
        wait(sema1) ;
        B.Count := B.Count +1;
        If B.Count = B.maximum then
            Begin
                For (I = 1 to B.Maximum – 1) do ( signal(sema2) );
                B.Count := 0;
                Dowait := false;
            End;
        signal(sema1) ;
        If Dowait then wait(sema2) ;
End;
```

Chapter 3
Synchronisation par les moniteurs

3.1 Rappel de cours

3.1.1 Avantages et inconvénients des sémaphores

La synchronisation par des sémaphores, vue dans la section précédente, permet à l'utilisateur de ne manipuler qu'une seule variable partagée par section critique en utilisant deux seules opérations: wait et signal.En effet, le blocage des processus et la gestion des files d'‘attente est un service offert par le système d'exploitation.

En plus de leur simplicité d'utilisation, les sémaphores donne la possibilité de faire entrer plusieurs processus à la fois dans leurs sections de code respectives, ce qui permet de les appliquer à la synchronisation dans le cas général.

Néanmoins, la dispersion de wait et signal parmi plusieurs processus, d'un côté, doit être correcte dans tous les processus puisqu'un seul "mauvais" processus peut faire échouer l'ensemble. D'un autre côté, il n'est pas toujours facile de comprendre ce qui se passe dans de telle structure.

Cet inconvénient a motivé Hoare en 1974 à définir un nouveau concept de synchronisation appelé Moniteurs.

3.1.2 Définitions

Les moniteurs sont des constructions,en langage de haut-niveau, qui procurent une fonctionnalité équivalente aux sémaphores mais plus facile à contrôler. Ils sont disponibles dans beaucoup de langages de programmation. Nous citons à tire d'exemples: Concurrent Pascal, Modula-3, C++, C sharp, Java. Ils peuvent être réalisés avec des sémaphores.

Plus formellement, un moniteur est un module (type de données abstrait – ADT) contenant: une ou plusieurs procédures, une séquence d'initialisation et des variables locales. Ces variables locales ne sont accessibles que par les procédures du moniteur.

Un processus entre dans le moniteur en invoquant une de ses procédures. A tout instant, un seul processus peut être actif dans le moniteur mais plusieurs processus peuvent être en attente dans ce dernier.

Le moniteur assure à lui seul l'exclusion mutuelle des procédures du moniteur. l'utilisateur n'a pas besoins de la programmer explicitement. Ainsi, la protection des données partagées est simplement assurée en les plaçant dans le moniteur.

La synchronisation de processus est effectuée en utilisant des variables conditionnelles qui représentent des conditions après lesquelles un processus pourrait attendre avant d'exécuter dans le moniteur.

3.1.3 Structure générale du moniteur

Voici la structure générale d'un moniteur:

Monitor nom-de-moniteur
DEBUT
/* déclarations de variables locales partagées */
Procédure p1(. . .)
    debut
        {code de p1}
    fin
Procédure p2(. . .)
    debut
        {code de p2}
    fin
. . . . . .
debut
    {code d'initialisation}
fin
FIN

Et voici un schéma (voir figure 3.1) représentant la structure générale d'un moniteur.

3.1.4 Variables conditionnelles

Les variables conditionnelles sont accessibles seulement dans le moniteur comme toutes les autres variables locales. Elles ne peuvent être invoquées qu'à travers deux fonctions:

x.wait() qui bloque l'exécution du processus exécutant sur la condition x. Le processus pourra reprendre son exécution seulement si un autre processus exécute x.signal.

x.signal() reprend l'exécution d'un processus bloqué sur la condition x. Si plusieurs processus sont bloqués sur cette condition, un seul processus est choisi parmi l'ensemble. Si, par contre, aucun processus n'est bloqué sur la condition x, la fonction x.signal() n'aura aucun effet. Nous présentons ci-dessous une représentation graphique des variables conditionnelles (voir figure 3.3)dans la structure générale d'un moniteur.

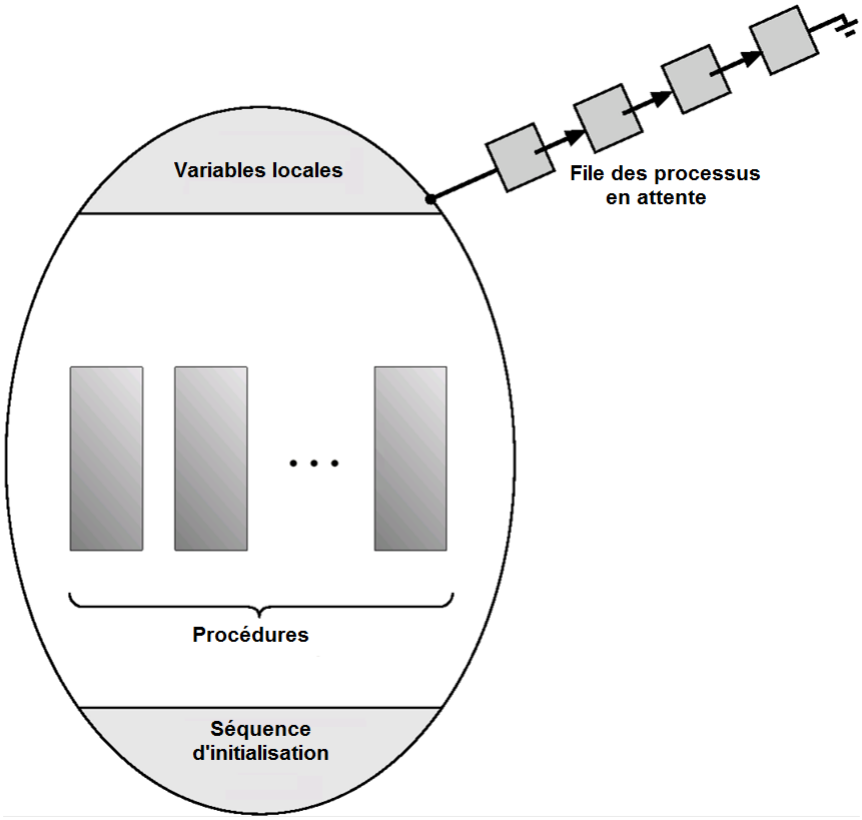

Fig. 3.1 Structure générale d'un moniteur Silberschatz et al (2018)

3.1.5 Stratégie de réveil de threads suite à l'exécution d'un signal

Si un processus P exécute x.signal() et libère un processus Q. Il y aura donc deux processus (P et Q) qui peuvent s'exécuter dans le moniteur. D'après la définition d'un moniteur ceci est interdit.

Deux stratégies sont, alors, possibles:

- Signaler et attendre: P attend jusqu'à ce que Q sorte du moniteur. La motivation de cette première stratégie est que: "Si on ne réveille pas tout de suite Q, la condition pour laquelle il a été réveillé pourrait perdre sa validité".

- Signaler et continuer: P continu son exécution et Q pourrait attendre jusqu'à ce que P sorte du moniteur. La motivation de cette deuxième stratégie est de: "Éviter une commutation de contexte."

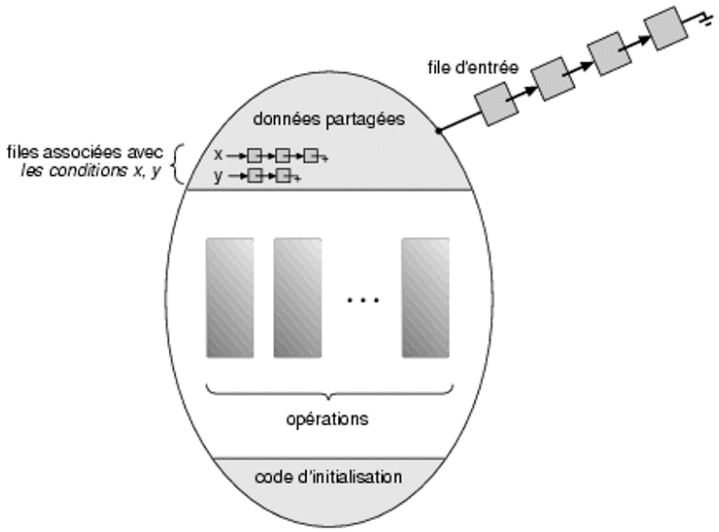

Fig. 3.2 Représentation graphique des variables conditionnelles Silberschatz et al (2018)

3.1.6 Exemple Illustratif

Pour conclure cette partie, nous allons présenter une solution, au problème célèbre de synchronisation connu sous le nom du problème des philosophes, en utilisant les moniteurs.

Pour cela, nous allons adopté, le protocole suivant:

Un philosophe prend la baguette qui est sur sa droite puis celle qui est sur sa gauche (pour philosophe i, baguette(i) puis baguette((i+1)

Et voici la solution:

DP1: est le nom du moniteur

prendrebaguette() et déposerbaguettes(): se sont des procédures du moniteur DP1.

Comportement du philosophe i

```
While (true) {
    penser
    DP1.Prendrebaguette(i)
    DP1.Prendrebaguette((i+1)%5)
    manger
    DP1.Déposerbaguettes(i)
}
```

Monitor DP1 {

```
Var Baguettelibre: array[0..4] of boolean;
Attendrebaguette: array[0..4] of Condition;
```

```
    Procédure Prendrebaguette(int i) {
        if not Baguettelibre[i] {
            Attendrebaguette[i].wait };
        Baguettelibre[i] = false
    }
    Procédure Déposerbaguettes(int i) {
        /*philosophe i dépose ses baguettes
        Baguettelibre[i] = true;
        Baguettelibre[(i+1)%5] = true;
        Attendrebaguette[i].signal; /*réveil éventuel du philosophe (i+4)%5
        Attendrebaguette[(i+1)%5].signal; /*réveil éventuel du philosophe (i+1)%5
    }
    Initialisation {
        for (int i = 0; i ¡ 5; i++)
            Baguettelibre[i] = true;
    }
}
```

3.2 Exercices

Exercice 3.1 *Nous considérons le problème de plusieurs processus concurrents manipulant un seul fichier en lecture et écriture. Le scénario de synchronisation considéré dans cet exercice donne la priorité aux rédacteurs. Ecrire le moniteur qui réalise ce scénario. Les processus lecteurs et rédacteur sont décrits comme suit :*

Processus lecteur

.....
StartRead ;
....reading
EndRead ;

Processus rédacteur

.....
StartWrite ;
.....Writing ;
EndWrite ;

Exercice 3.2 *Pour traverser une rivière dans une ville, la mairie assure aux citoyens un transport par barque. Ainsi, pour manque de moyens financiers, elle ne met à leur disposition qu'une seule barque pouvant transporter au maximum quatre (04) personnes à la fois. Si on considère, dans ce cas, que chaque citoyen est un processus qui exécute le code suivant:*

Citoyen (rive : entier) *// rive désigne les deux bords de la rivière (0 ou 1)*
Debut
Arrive(rive)
Embarquer-et-Traverser-rivière
Quitte(rive)
Fin

Compléter le moniteur, ci-dessous, qui permet le bon fonctionnement de ce système de transport. Pour cela, le moniteur dispose de deux procédures (Arrive et Quitte). La procédure arrive doit s'assurer que la barque est sur la rive où se trouve le citoyen voulant traverser et qu'elle n'est pas complet. La procédure quitte est invoquée par le citoyen quand il atteint sa destination. Pour simplifier la solution, on ne s'intéresse pas à Embarquer-et-Traverser-rivière.

type rivière = moniteur
localisation, nbce : entier;
//localisation: position de la barque(0 ou 1)
//nbce: nombre de citoyens dans la barque
c: condition;

```
procedure entry arrive(rive:entier)
debut
    Tantque ((rive ¡¿ localisation) ou (..... )) faire
        ..........;
    ........;
fin
procedure entry quitte(rive:entier)
debut
    .........;
    si (nbce=1) alors
        debut
            .........;
            .........;
        fin
fin
Initialisation
debut
    .........;
    .........;
fin
Fin rivière
```

Exercice 3.3 *Considérons une ressource dont il existe un nombre fixe d'exemplaires, ou unités. Un processus peut acquérir, sur sa demande, un nombre n de ces unités, les utiliser, puis les libérer. Entre son acquisition et sa libération, une unité est dite allouée au processus qui l'utilise. Toutes les unités sont équivalentes du point de vue de leur utilisation ; on dit aussi que la ressource est banalisée. Nous admettons les règles d'utilisation suivantes :*

1. *une unité ne peut être allouée qu'a un processus à la fois.*
2. *une unité ne peut être allouée à un processus demandeur que si elle est libre.*
3. *une opération de libération s'applique toujours aux dernières ressources acquises par le processus qui l'exécute.*
4. *une demande d'allocation est bloquante en cas d'échec (i.e., nombre insuffisant)*

Ecrire le moniteur qui réalise la gestion de cette ressource banalisée.

Exercice 3.4 *Un salon de coiffure est doté d'un seul fauteuil de coiffure et de N chaises en salle d'attente. Quand il n'y a pas de clients, le coiffeur se repose dans le fauteuil de coiffure. Lorsqu'un client arrive et trouve le coiffeur endormi, il le réveille. Un client qui arrive et trouve le coiffeur occupé s'assoit sur l'une des N chaises et attend son tour. S'il n'y a plus de chaises libres il n'attend pas et s'en va. En assimilant le coiffeur et les clients à des processus, donner une solution à ce problème en utilisant les moniteurs.*

Exercice 3.5 *Soit la solution, à base de moniteur au problème classique des philosophes, présenté ci-dessous. Cette solution exploite l'idée triviale où chaque philosophe*

commencera par prendre la baguette qui est sur sa gauche, puis la baguette qui est sur sa droite. Le pseudo code de cette solution utilise le moniteur Dinner-philosophes décrit ci-après :

```
Type Dinner-philosophes = Monitor
    Var baguette-disponible : array[0..4] of condition ;
    Var baguette-libre : array[0..4] of 0..1;
    Procedure Entry prendre(i: 0..4)
    Begin
        If ( baguette-libre[i] ≠ 1) then
            wait(baguette-disponible[i]);
        baguette-libre[i]:= baguette-libre[i] – 1;
    End;
    Procedure Entry poser(i:0..4)
    Begin
        baguette-libre[i]:= baguette-libre[i] + 1;
        signal(baguette-disponible[i]) ;
    End;
    Begin
        For i := 0 to 4 do baguette-libre[i] := 1;
    End;
End Dinner-philosophes.
```

1. *Donner le pseudo code d'un processus philosophe.*
2. *Existe-t-il un risque d'interblocage dans cette solution ? Justifiez par un scénario.*

3.3 Solutions des exercices

Exercice 3.1

```
Type LRPR = Monitor
    Var nl, nlc, nred : integer ;
        ecr:Boolean;
        clect, cecr: condition;

    Procedure entry StartRead
    Begin
        nl:=nl+1;
        if nred ¿ 0 then
        Begin
            clect.wait;
            clect.signal;
        End;
        nlc:=nlc+1;
    End;

    Procedure entry EndRead
    Begin
        nl:=nl-1;
        nlc:=nlc-1;
        if nlc = 0 then
            cecr.signal;
    End;

    Procedure entry StartWrite
    Begin
        nred:=nred + 1;
        if ecr ou nlc ¿ 0 then
            cecr.wait;
        ecr:=true;
    End;

    Procedure entry EndWrite
    Begin
        ecr:=false;
        nred:=nred - 1;
        if nred ¿ 0 then
            cecr.signal
        else
            clect.signal;
    End;
```

```
    Begin
        ecr:=false;
        nl:=0;
        nlc:=0;
        nred:=0;
    End;
End monitor LRPR.
```

Exercice 3.2

```
type rivière = moniteur
    localisation, nbce : entier;
    //localisation: position de la barque(0 ou 1)
    //nbce: nombre de citoyens dans la barque
    c: condition;

    procedure entry arrive(rive:entier)
    debut
        Tantque ((rive ¡¿ localisation) ou (nbce = 4)) faire
            c.wait;
        nbce++;
    fin

    procedure entry quitte(rive:entier)
    debut
        nbce–;
        si (nbce=1) alors
        debut
            localisation = 1-localisation;
            c.signal;
        fin
    fin

    Initialisation
    debut
        nbce = 0;
        localisation = 0;
    fin
Fin rivière
```

Exercice 3.3

```
Type ressource = monitor
    Var nlibre : integer ; disp : condition ;
```

```
    Procedure entry demander(n)
    Begin
        While n ¿ nlibre do
        Begin
            disp.wait;
            disp.signal;
        End;
        nlibre:=nlibre – n;
    End;

    Procedure entry libérer(n)
    Begin
        nlibre:=nlibre + n ;
        disp.signal;
    End;

    Begin
        nlibre:= N;
    End;
End ressource.
```

Exercice 3.4

```
Type CoiffeurDormeur = Moniteur
    Var attendre, dormir : condition;
        NbClientsAttente : entier;

    Procedure entry Veutcoiffer()
    Debut
        Si NbClientsAttente ¿ 0 alors
            attendre.signal;
        Sinon
            dormir.wait;
            attendre.signal;
        Finsi
    Fin

    Procedure entry Veutsecoiffer(var B: Boolean)
    Debut
        Si NbClientsAttente ¡ N alors
            NbClientsAttente++;
            dormir.signal;
            attendre.wait;
            NbClientsAttente–;
```

```
        Sinon
            B:=Vrai;
        Finsi
    Fin
    Debut
        NbClientsAttente:=0;
    Fin
Fin CoiffeurDormeur

Processus Client i quelconque
    Boolean complet initialisé faux
    Debut
        CD.VeutseCoiffer(complet);
        Si non(complet) alors
            Se faire coiffer;
        Finsi
    Fin

Processus Coiffeur
    Debut
        Tanque vrai faire
            CD.VeutCoiffer();
            Travaille;
        Fintantque
    Fin
```

Exercice 3.5

1. **Processus Philosophe i**

```
    Tantque (vrai) faire
        Penser;
        Dinner-philosophes.prendre((i+1)mod5);
        Dinner-philosophes.prendre(i);
        Manger;
        Dinner-philosophes.poser((i+1)mod5);
        Dinner-philosophes.poser(i);
    Ftantque
```

2. Nous pouvons atteindre dans cette solution une situation d'interblocage. En effet, si tous les philosophes prennent en même temps la baguette de gauche. Ils vont se bloquer mutuellement, puisque chaque philosophe pour pouvoir manger attend la baguette de droite détenu par son voisin.

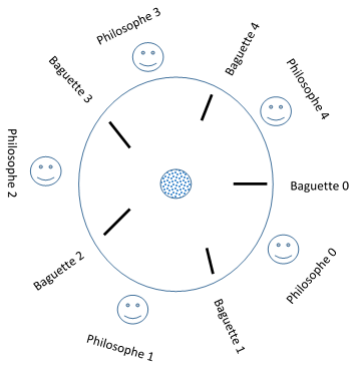

Fig. 3.3 Dîner des cinq philosophes

Chapter 4
Problème de l'interblocage et ses solutions

4.1 Rappel de cours

4.1.1 Définition de l'interblocage

Un ensemble de processus est en interblocage si chaque processus attend un événement que seul un autre processus de l'ensemble peut provoquer.

4.1.2 Caractérisation de l'interblocage

Une situation d'interblocage peut survenir si les quatre conditions suivantes se produisent simultanément (conditions nécessaires mais non suffisantes):

- Exclusion mutuelle : Le système a des ressources non partageables (un seul processus à la fois peut s'en servir). Comme exemple : Processeur, une zone de mémoire, un périphérique, mais aussi un sémaphore, un moniteur, une section critique.
- Occupation et attente (hold and wait) : Un processus détient au moins une ressource non partageable et qui attend d'acquérir des ressources supplémentaires détenus par d'autres processus.
- Pas de réquisition (préemption) : Un processus qui a saisi une ressource non partageable la garde jusqu'à ce qu'il aura complété sa tâche.
- Attente circulaire : Il y a un cycle de processus tel que chaque processus pour compléter doit utiliser une ressource non partageable qui est utilisée par le suivant, et que le suivant gardera jusqu'à sa terminaison

4.1.3 Méthodes de traitement

Trois méthodes principales sont possibles pour traiter le problème de l'interblocage:

- Employer un protocole pour assurer que le système ne se trouvera jamais dans une situation d'interblocage.
- Permettre que le système se trouve en situation d'interblocage et le corriger ensuite.
- Ignorer complètement le problème d'interblocage et supposer que le système ne se trouvera jamais dans de telles situations.

4.1.3.1 Prévenir les interblocages

Pour prévenir un interblocage, il faut éviter la réalisation d'au moins une des conditions citées ci-dessous.

Exclusion mutuelle

Réduire le plus possible l'utilisation des ressources partagées et Sections Critiques. En général, ceci n'est pas possible puisque certaines ressources sont intrinsèquement non partageables.

Occupation et attente

Un processus qui demande de nouvelles ressources ne devrait pas en retenir d'autres (les demander toutes ensemble). Dans ce cas, Il y aura possibilité de famine.

Pas de réquisition

Si un processus qui demande d'autres ressources ne peut pas les avoir, il doit être suspendu, ses ressources doivent être rendues disponibles. Cela Peut s'appliquer aux ressources dont l'état peut être facilement sauvegardé et restauré (registre UC ou zones mémoires)

Attente circulaire

Imposer un ordre total sur les ressources. Un processus doit demander les ressources dans cet ordre (comme exemple : tout processus doit toujours demander une imprimante avant de demander une unité de disque)

- F: R vers N
- Un Processus détient Ri et ne peut avoir Rj que si F(Rj)¿F(Ri)
- Pour avoir Rj, le processus doit libérer les Ri / F(Ri) ¿= F(Rj)

Le fait d'éviter d'assurer l'une au moins des quatre conditions ci-dessus pour prévenir un interblocage, engendre:

- Une faible utilisation des ressources
- Une capacité de traitement réduite du système.

D'où l'utilisation d'une autre méthode, à savoir éviter les inter-blocages en demandant des informations supplémentaires sur la façon dont les ressources vont être requises. Cette méthode est présentée dans la section suivante.

4.1.3.2 Eviter les interblocages

Pour éviter un éventuel futur interblocage, il faut savoir quelles sont les ressources disponibles, les ressources allouées à chaque processus et les futures requêtes de chaque processus. Sur la base de ces informations, un examen dynamique est effectué sur l'état d'allocation des ressources afin d'assurer qu'il ne puisse jamais exister une condition d'attente circulaire. Ainsi, Il faut éviter d'accepter toute requête

qui fait passer un système d'un état sain vers un état malsain. Puisque l'ensemble des états d'interblocage est un sous ensemble des états malsains (voir figure ci-dessous).

Un état est dit sain s'il existe à partir de cet état une séquence saine. Sinon, l'état est dit malsain. Si à partir d'un état donné, l'ensemble des processus du système peuvent terminer normalement dans un ordre donné leur exécution, alors cette succession de terminaisons est dite séquence saine.

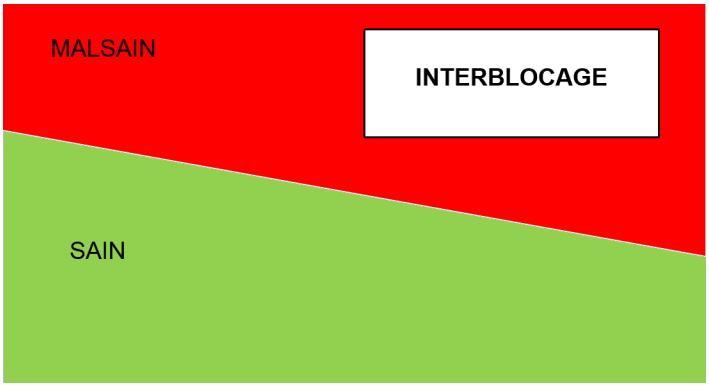

Fig. 4.1 Etats d'une système

Exemple illustratif:

Soit un système avec 12 imprimantes, 3 processus (P0, P1, P2) et qui se trouve dans l'état suivant:

Processus	Besoins maximaux	Allocations courantes
P0	10	5
P1	4	2
P2	9	2

Cet état représente un état sain pour le système puisqu'il y a une possibilité de terminaison de P1 ensuite P0 et enfin P2. En d'autres termes, il existe une séquence saine ¡P1, P0, P2¿ pour cet état.

Tandis que, l'état suivant est un état malsain puisqu'il n'existe pas de séquence saine.

Processus	Besoins maximaux	Allocations courantes
P0	10	5
P1	4	2
P2	9	3

Pour traiter le problème d'interblocage d'une façon effective, nous allons définir, dans la section suivante, la notion de graphe d'allocation de ressources.

Graphe d'allocation de ressources

Définitions

G = (V, E) où V : ensemble de sommets et E : ensemble d'arêtes.

- V est partitionné en deux sous-ensembles :
- P = P1, P2, . . . , Pn, l'ensemble de tous les processus (voir figure 4.2 -a-).
- R = R1, R2, . . . , Rm, l'ensemble de tous les types de ressources(voir figure 4.2 -b-).
- E contient :
- Arête requête : arête dirigée de Pi vers Rk (Pi a besoin d'un exemplaire de Rk)(voir figure 4.2 -c-)
- Arête affectation : arête dirigée de Ri vers Pk (Pi détient un exemplaire de Rk)(voir figure 4.2 -d-)
- Arête de réclamation : arête dirigée de Pi vers Rk (Pi peut demander dans le futur un exemplaire de Rk)(voir figure 4.2 -e-)

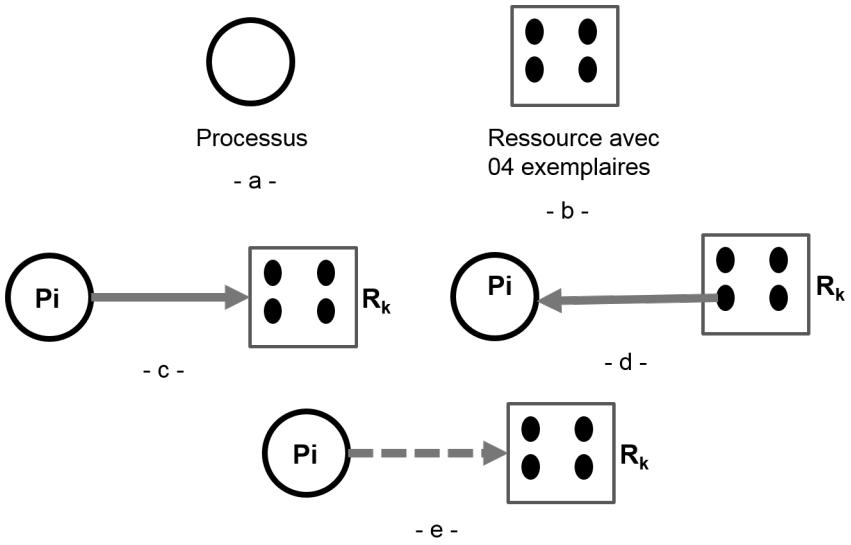

Fig. 4.2 Représentation graphique d'un GAR

Algorithme pour éviter l'interblocage

Il s'agit de construire un graphe d'allocation de ressources et voir s'il y a une manière dont tous les processus peuvent terminer. Dans le cas d'une ressource par type, l'algorithme cherche des circuits dans le graphe (algorithme d'ordre n2 , avec n = nombre de processus)(voir figures 4.3 et 4.4). Par contre, cet algorithme n'est

pas applicable dans le cas de plusieurs ressources par type. Un autre algorithme sera présenté ultérieurement.

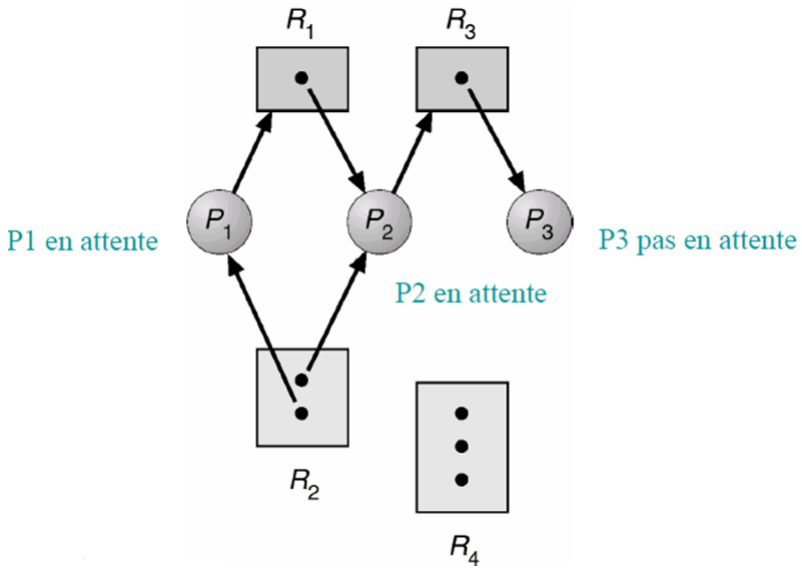

Fig. 4.3 Absence de circuit - Pas d'interblocage

Algorithme du banquier

Cet algorithme pourrait s'appliquer dans un système bancaire pour s'assurer que la banque ne prête jamais son liquide disponible de telle sorte qu'elle ne puisse plus satisfaire tous ses clients. Il est applicable à un système d'allocation de ressources avec possibilité de plusieurs instances pour chaque type de ressource. En entrant dans le système, tout processus doit déclarer le nombre maximal d'instances de chaque type de ressource dont il a besoin (il ne doit pas excéder le nombre total de ressource). Quand un processus requiert un ensemble de ressources, le système doit déterminer si cette allocation laissera le système dans un état sain.

Détermination d'un état sain

1- Soient Work et Finish : deux vecteurs de longueur m et n respectivement.
Work := Available et Finish[i] := false (i= 1 à n)
2- Trouver i tel que :
Finish[i]=false et Need[i,-] ¡= Work
Si i n'existe pas aller à 4.

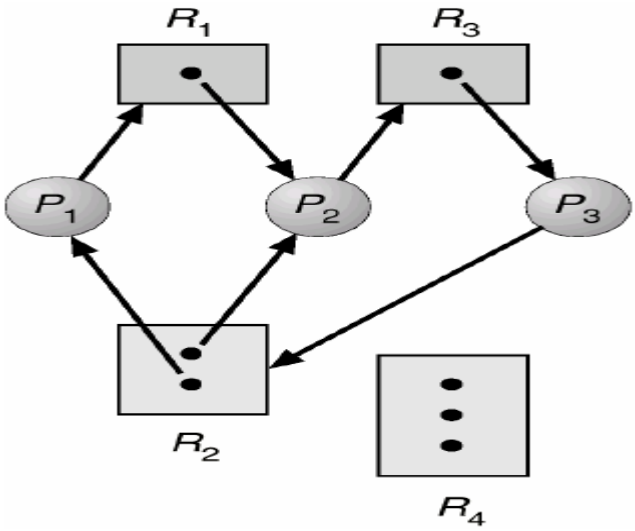

Fig. 4.4 Existence d'un circuit - Interblocage

3- Work := Work + Allocation[i, -]
Finish[i] := True //on simule la fin d'exécution de pi Aller à 2
4- Si Finish[i] = true pour tout i, alors le système est dans un état sain.
Sinon l'état est malsain.
Sachant que :
n : nombre de processus
m : nombre de type de ressources
Available : vecteur de longueur m Available[j]=k (k instances de Rj sont disponibles)
Max : matrice n x m, Max[i, j]=k (Pi peut acquérir au plus k instances de Rj)
Allocation : matrice n x m, Allocation[i, j]=k (k instances de Rj sont allouées à Pi)
Need: matrice n x m, Need[i, j]=k (Pi peut avoir besoin de k instances de Rj) Need[i, j]=Max[i, j] - Allocation[i, j]

Requête de ressources

Soit Requesti le vecteur de requête pour le processus Pi. Requesti[j] = k (Pi désire k instances de Rj)
1- Si Requesti ¡= Need[i,-] aller à 2
Sinon erreur (le processus i a excédé sa réclamation maximale)
2- Si Requesti ¡= Available aller à 3
Sinon Pi doit attendre (ressources non disponibles)
3- On suppose que : le système alloue les ressources demandées par Pi en modifiant l'état ainsi:
Available := Available - Requesti

Allocation[i,-]:=Allocation [i,-] + Requesti
Need[i,-]:=Need [i,-] – Requesti
4- Si le nouvel état est sain alors la requête est accordée
Sinon l'état est restitué et le processus ayant fait la requête est mis en attente.

4.1.3.3 Détecter et corriger les interblocages

Si un système n'emploie pas d'algorithme pour prévenir ou pour éviter les interblocages, il doit en cas de situation d'interblocage, détecter cette interblocage ensuite le corriger.

Détecter les interblocages

Instance unique pour chaque type de ressource

Dans ce cas, le système doit assurer la maintenance du graphe d'attente et appeler périodiquement un algorithme qui cherche un circuit dans ce graphe.
Le graphe d'attente est obtenu à partir du graphe d'allocation de ressources en éliminant les nœuds représentant les ressources. Ainsi, le graphe d'attente représente un sous graphe du graphe d'allocation de ressources.

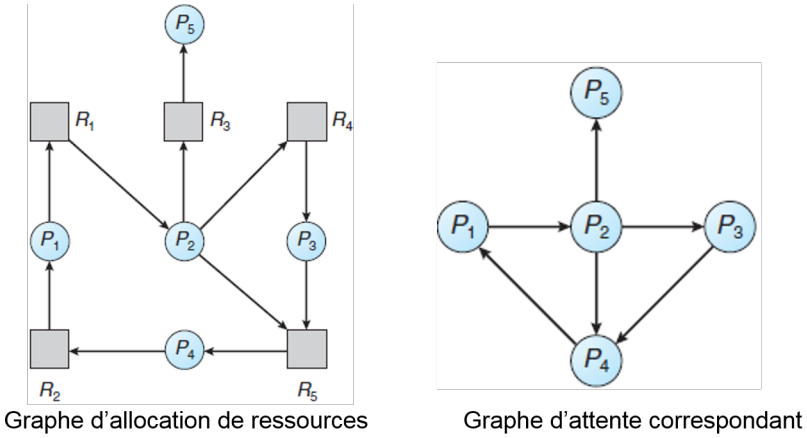

Fig. 4.5 Exemple d'un graphe d'attente

Plusieurs instance pour un type de ressource

Request mat(n x m): Request[i,j]=k (Pi demande k instances de plus de Rj)

1- Soient Work et Finish: deux vecteurs de longueur m et n respectivement
Work:=Available
Pour (i= 1 à n)
Si Allocation[i,-] ¡¿ 0 alors Finish[i]:=false
sinon Finish[i]:=True

2- Trouver i tel que: Finish[i]=false et Request[i,-]¡=Work
Si i n'existe pas alors aller à 4

3- Work:=Work + Allocation[i,-]
Finish[i]:=True
aller à 2

4- Si Finish[i] = false pour un certain i, alors le système est en situation d'interblocage.

Corriger les interblocages

Deux alternatives sont possibles pour corriger un interblocage détecté:

1. Manuellement en informant l'opérateur.
2. Automatiquement par le système.

Dans le cas d'une correction automatique, le système peut procéder par l'une des deux possibilités suivantes:

1. Terminaison de processus: Dans ce cas, il est possible de:
   a. Avorter tous les processus en interblocage, ce qui va rompre le circuit d'interblocage mais à un cout très élevé. En effet, tous les calculs partiels effectués par ces processus seront perdus et recalculés plus tard.
   b. Avorter un processus à la fois jusqu'à ce que le circuit d'interblocage soit éliminé. Ceci provoque une surcharge considérable puisqu'après chaque avortement, on doit appeler l'algorithme de détection de circuit.
2. Réquisition de ressources: Enlever des ressources à un processus et les affecter à un autre pour défaire le circuit d'interblocage. Cette requisition nous oblige à réfléchir sur:
   a. La sélection d'une victime en précisant quelles ressources seront réquisitionnés et à partir de quels processus. On peut prendre comme facteur de coût le nombre de ressources détenues et le temps d'exécution.
   b. Le retour en arrière puisqu'une perte de ressources implique forcément un redémarrage du processus à partir d'un état sain (état initial).

c. Comment garantir que les ressources ne seront pas réquisitionnées à partir du même processus, en d'autres termes comment éviter la famine.

4.1.4 Conclusion

Une situation d'interblocage se produit si un ou plusieurs processus attendent indéfiniment un événement qui ne peut être provoqué que par l'un des processus en attente. Le concepteur du système d'exploitation peut adopter l'une des approches suivantes vis à vis du problème d'interblocage:

1. Prévenir ou éviter
2. Détecter et corriger
3. Ignorer

Aucune de ces approches n'est adéquate à elle seule pour tous les types de problème d'allocation de ressources dans les SE. Ainsi, ces approches peuvent être combinées, en permettant d'utiliser séparément une approche optimale pour chaque classe de ressources dans le SE.

4.2 Exercices

Exercice 4.1 *On considère un système composé de quatre processus P1, P2, P3 et P4 qui partagent un même fichier. Ce fichier est composé de* 6 *blocs de données notés B1, B2, B3, B4, B5 et B6. L'exécution de chacun des processus nécessite la pose de verrous exclusifs sur certains blocs. Un verrou posé par un processus sur un bloc réserve exclusivement ce bloc à d'éventuelles utilisations ultérieures par ce dernier. Étant donnée que la pose de verrous identifie explicitement les blocs de données concernés, les blocs ne sont ni anonymes ni interchangeables. Par conséquent, le nombre de classes de ressources dans ce système est égal au nombre de blocs distincts de ce fichier partagé (i.e.* 6*). Les annonces de poses de verrous exclusifs sur les blocs de chaque processus sont données ci-après :*

*P*1 : *B*1 *B*3 *B*2
*P*2 : *B*4 *B*2 *B*3
*P*3 : *B*5 *B*2 *B*1
*P*4 : *B*6 *B*2 *B*4 *B*3

On suppose qu'à l'état courant, les blocs suivants sont verrouillés : B1 et B3 par P1, B4 par P2, B5 par P3, B6 par P4.

1. *Donner le vecteur Available de cet état* ?
2. *Donner la matrice Allocation de cet état* ?
3. *Donner la matrice Max du système* ?
4. *Donner la matrice Need de cet état* ?
5. *Cet état est-il sain* ?
6. *Le système reçoit dans l'ordre les requêtes suivantes : P3 demande B2, P1 demande B2, P2 demande B2. Le système répond favorablement à une requête si cela conduit vers un état sain. Indiquer, en utilisant l'algorithme du banquier, lequel des processus parviendra à verrouiller le bloc B2. Remarque : Indiquer pour qu'elle raison le processus qui ne parvient pas à verrouiller le bloc B2 est mis en attente.*

Exercice 4.2 *Soit l'état suivant d'un système à plusieurs classes de ressources:*

	Allocation					*Max*					*Available*			
	R1	*R2*	*R3*	*R4*			*R1*	*R2*	*R3*	*R4*				
P0	*0*	*0*	*1*	*2*		*P0*	*0*	*0*	*1*	*2*				
P1	*1*	*0*	*0*	*0*		*P1*	*1*	*7*	*5*	*0*	*R1*	*R2*	*R3*	*R4*
P2	*1*	*3*	*5*	*4*		*P2*	*2*	*3*	*5*	*6*	*1*	*5*	*2*	*0*
P3	*0*	*6*	*3*	*2*		*P3*	*0*	*6*	*5*	*2*				
P4	*0*	*0*	*1*	*4*		*P4*	*0*	*6*	*5*	*4*				

1. *Donner le contenu de la matrice Need ?*
2. *Montrer que cet état est un état sain ?*
3. *Si une demande du processus p1 (0,4,2,0) arrive, doit on la satisfaire immédiatement?*

Exercice 4.3 *Soit l'état suivant d'un système à plusieurs classes de ressources:*

Allocation					*Max*					*Available*			
	R1	*R2*	*R3*	*R4*		*R1*	*R2*	*R3*	*R4*				
P0	*0*	*0*	*1*	*1*	*P0*	*0*	*0*	*1*	*1*				
P1	*1*	*0*	*0*	*0*	*P1*	*1*	*7*	*5*	*0*	*R1*	*R2*	*R3*	*R4*
P2	*1*	*3*	*5*	*4*	*P2*	*2*	*3*	*5*	*6*	*1*	*5*	*1*	*0*
P3	*0*	*6*	*2*	*2*	*P3*	*0*	*6*	*5*	*3*				
P4	*0*	*0*	*1*	*4*	*P4*	*0*	*6*	*5*	*6*				

1. *Montrez que cet état est un état malsain ?*
2. *Montrez que pour chaque classe isolée l'état est sain ?*
3. *Que peut-on conclure ?*

Exercice 4.4 *Soit l'état suivant d'un système à une classe de ressources.*

Allocation		*Max*		*Available*
	R		*R*	
*P*0	5	*P*0	10	*R*
*P*1	2	*P*1	4	2
*P*2	1	*P*2	6	

1. *Vérifier que cet état est malsain ?*
2. *Montrer à l'aide d'un scénario qu'il est différent d'un état d'interblocage ? Justifier votre réponse.*

Exercice 4.5 *Soit un système à une classe de ressources dont l'état courant est donné par les matrices suivantes:*

Allocation		*Max*	
	R		*R*
*P*1	1	*P*1	3
*P*2	1	*P*2	2
*P*3	3	*P*3	9
*P*4	2	*P*4	7

1. *Calculer la matrice Need.*
2. *Combien de ressources au minimum doivent être disponibles pour que l'état du système soit sain ? (**Indication:** Ordonner les processus par ordre croissant de leur besoins restants).*
3. *Quel est donc le nombre de ressources existantes dans le système ?*

Exercice 4.6 *Même questions que pour l'exercice 4.5 avec l'état courant suivant:*

Allocation		*Max*		*Available*
	R		*R*	
P1	*1*	*P1*	*2*	*R*
P2	*1*	*P2*	*5*	*min*
P3	*3*	*P3*	*11*	
P4	*2*	*P4*	*7*	

4.3 Solutions des exercices

Exercice 4.1

1. Vecteur Available = (0, 1, 0, 0, 0, 0)

2. Matrice Allocation =

	B1	B2	B3	B4	B5	B6
P1	1	0	1	0	0	0
P2	0	0	0	1	0	0
P3	0	0	0	0	1	0
P4	0	0	0	0	0	1

3. Matrice Max =

	B1	B2	B3	B4	B5	B6
P1	1	1	1	0	0	0
P2	0	1	1	1	0	0
P3	1	1	0	0	1	0
P4	0	1	0	1	0	1

4. Matrice Need = Max - Allocation =

	B1	B2	B3	B4	B5	B6
P1	0	1	0	0	0	0
P2	0	1	1	0	0	0
P3	1	1	0	0	0	0
P4	0	1	0	1	0	0

5. En appliquant l'algorithme du banquier, nous montrons que $P1.P2.P3.P4$ est une séquence saine, donc cet état est sain.

6. P3 demande B2: Cette requête ne peut être satisfaite immédiatement puisque l'état obtenu en appliquant l'algorithme du banquier est un état malsain.
   P1 demande B2: Cette requête peut être satisfaite immédiatement puisque l'état obtenu en appliquant l'algorithme du banquier est un état sain.
   P2 demande B2: Cette requête ne peut être satisfaite immédiatement puisque B2 est déjà alloué à P1.

Exercice 4.2

1. La matrice Need = Max - Allocation

Need =

	R1	R2	R3	R4
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	0

2. En appliquant l'algorithme du banquier, Nous montrons que $P0.P3.P1.P2.P4$ est une séquence saine donc l'état à l'instant t0 est sain.

3. Sachant que: $(0,4,2,0) \leq (0,7,5,0)$ et $(0,4,2,0) \leq (1,5,2,0)$. En plus l'état(présenté ci-dessous) obtenu à l'instant t1, après satisfaction de cette demande est sain. Donc, cette requête peut être satisfaite immédiatement.

	Allocation					Need					Available			
	R1	R2	R3	R4		R1	R2	R3	R4					
P0	0	0	1	2	P0	0	0	0	0					
P1	1	4	2	0	P1	0	3	3	0		R1	R2	R3	R4
P2	1	3	5	4	P2	1	0	0	2		1	1	0	0
P3	0	6	3	2	P3	0	0	2	0					
P4	0	0	1	4	P4	0	6	4	2					

Exercice 4.3

1. Matrice Need = Max - Allocation =

	R1	R2	R3	R4
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	3	1
P4	0	6	4	2

En appliquant l'algorithme du banquier, initialement work = (1, 5, 1, 0), la première ligne de Need $\leq$ Work. Donc P0 peut terminer son exécution et rendre les ressources qu'il détient.
La deuxième itération work = (1, 5, 2, 1), les lignes 2, 3, 4 et 5 de la matrice Need ne sont pas $\leq$ au vecteur work donc aucun processus ne peut terminer. Ainsi il n'existe pas de séquences saines, donc cet état est malsain.

2. En appliquant l'algorithme du banquier premièrement pour R1, deuxièmement pour R2, ensuite pour R3 et enfin pour R4; nous remarquons qu'à chaque fois il existe des séquences saines donc par conséquent les états sont sain

3. Le fait que l'état est sain pour chaque classe de ressources n'implique pas forcément qu'il est sain pour l'ensemble des ressources.

Exercice 4.4

L'état des besoins courants est:

Need		$Available_c$
p0	5	2
P1	2	
p2	5	

La séquence fiable est composée d'un seul processus $P1$ et elle n'est pas prolongeable donc elle est partielle. Par conséquent l'état est malsain. Toute fois, puisque il y a de l'activité dans le système, les processus n'ont pas encore formulé des demandes. Le deuxième processus par exemple tente d'accéder une zone de non droit sera avorté par le système tout en réquisitionnant ses ressources et ainsi le processus $P0$

termine son exécution. Pas d'interblocage car pas d'attente circulaire.

Exercice 4.5

1. Matrice Need (Pour ce cas il s'agit d'un vecteur).

	Need	Available
P1	2	min
P2	1	
P3	6	
P4	5	

2. Remarquons que l'état est sain si nous pouvons trouver une séquence d'exécution de tous les processus. La valeur minimale de ressources disponibles correspond à la séquence qui fait terminer les processus dans l'ordre croissant de leur besoin restant. De ce fait la séquence susceptible de définir la valeur min correspond à la séquence P2.P1.P4.P3.
   Soit Work := Available = X. Le processus de terminaisons séquentielle de processus peut être résumé dans le tableau suivant:

Processus à faire terminer	Condition de terminaison	Instances rendues	Condition sur X
P2	$Work = X \geq 1$	1	$X \geq 1$
P1	$Work = X + 1 \geq 2$	1	$X \geq 1$
P4	$Work = X + 1 + 1 \geq 5$	2	$X \geq 3$
P3	$Work = X + 1 + 1 + 2 \geq 6$	3	$X \geq 2$

   La valeur de *min* est la valeur minimale de X qui satisfait toutes les inégalités. En d'autre termes $min = 3$.
3. Le nombre de ressources dans le système est égale aux ressources allouées plus les ressources disponibles initialement, donc $3+1+1+2+3 = 10$

Exercice 4.6

1. Matrice Need (Pour ce cas il s'agit d'un vecteur).

Allocation	Max	Need	Available
R	R	R	R
P1 1	P1 2	P1 1	X
P2 1	P2 5	P2 4	
P3 3	P3 11	P3 8	
P4 2	P4 7	P4 5	

2. Remarquons que l'état est sain si nous pouvons trouver une séquence d'exécution de tous les processus. La valeur minimale de ressources disponibles correspond à la séquence qui fait terminer les processus dans l'ordre croissant de leur besoin restant. De ce fait la séquence susceptible de définir la valeur min correspond à

la séquence P1.P2.P4.P3.
Soit Work := Available = X. Le processus de terminaisons séquentielle de processus peut être résumé dans le tableau suivant:

Processus à faire terminer	Condition de terminaison	Instances rendues	Condition sur X
P1	$Work = X \geq 1$	1	$X \geq 1$
P2	$Work = X + 1 \geq 4$	1	$X \geq 3$
P4	$Work = X + 1 + 1 \geq 5$	2	$X \geq 3$
P3	$Work = X + 1 + 1 + 2 \geq 8$	3	$X \geq 4$

La valeur de *min* est la valeur minimale de X qui satisfait toutes les inégalités.
En d'autre termes $min = 4$.

3. Le nombre de ressources dans le système est égale aux ressources allouées plus les ressources disponibles initialement, donc $4 + 1 + 1 + 2 + 3 = 11$

Chapter 5
Paradigmes universels de synchronisation

5.1 Paradigme des philosophes

5 philosophes passent leur temps à penser et à manger. Au centre de la table, un bol de riz. Seulement 5 baguettes sont disponibles. Pour manger il faut 2 baguettes, celle de droite et celle de gauche.
En effet le problème des philosophes est un problème classique de synchronisation qui illustre la difficulté d'allouer des ressources aux processus tout en évitant l'interblocage et la famine. Les ressources sont utilisées en exclusion mutuelle.

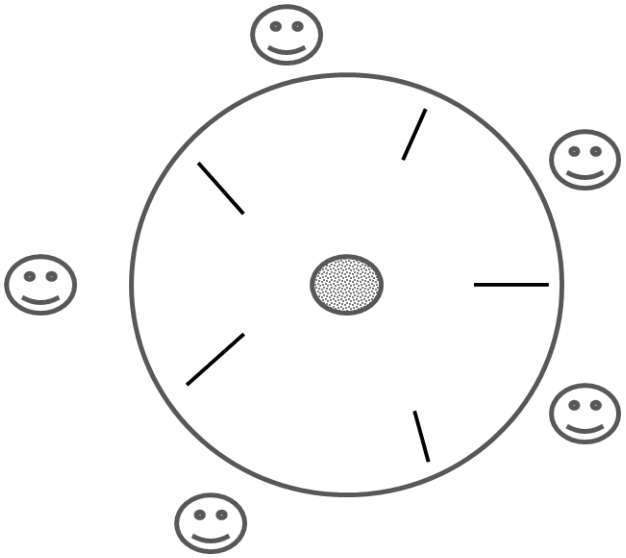

Fig. 5.1 Problème des philosophes

Hypothèses générales:

- Un philosophe qui a faim tente d'acquérir les baguettes. Si une baguette est occupée par son voisin il attend qu'il la dépose sur la table (critère de politesse). Point de vue informatique, on dit qu'il n'y a pas de réquisition de ressources. Le philosophe attend sans se faire remarquer qu'il a faim.
- Un philosophe qui a les deux baguettes commence à manger. Il garde les baguettes dans ses mains jusqu'à ce qu'il termine de manger.
- Un philosophe qui termine de manger dépose les baguettes et fait un signe pour se faire remarquer par ses voisins.

Rapprochement du problème à celui des processus:

- Un philosophe est similaire à un processus.
- Une baguette est une ressource partagée entre deux philosophes.

- L'utilisation d'une baquette se fait en exclusion mutuelle.

5.1.1 Premier protocole:

Afin d'élaborer une solution à ce problème, le philosophe 1 propose le protocole qui consiste à: Tout philosophe qui a faim tente de prendre les baguettes une à la fois, celle qui est sur sa droite puis celle qui est sur sa gauche.

Le pseudo code régissant le comportement d'un philosophe est le suivant:

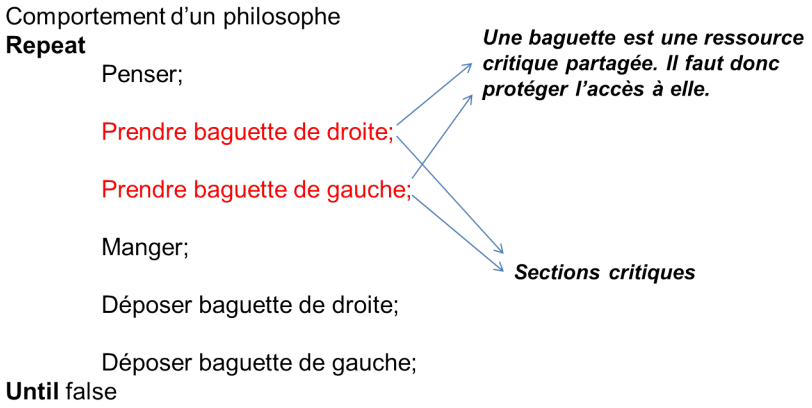

Fig. 5.2 Premier protocole: Comportement d'un philosophe

Afin d'aboutir à une implémentation de ce protocole, nous formalisons le problème selon la figure 5.3.

5.1.1.1 Solution à base des sémaphores

1. Déclaration des variables:

   Var Baguette:array[0..4] **of** Sémaphore initialisée à [1,1,1,1,1]

2. Solution

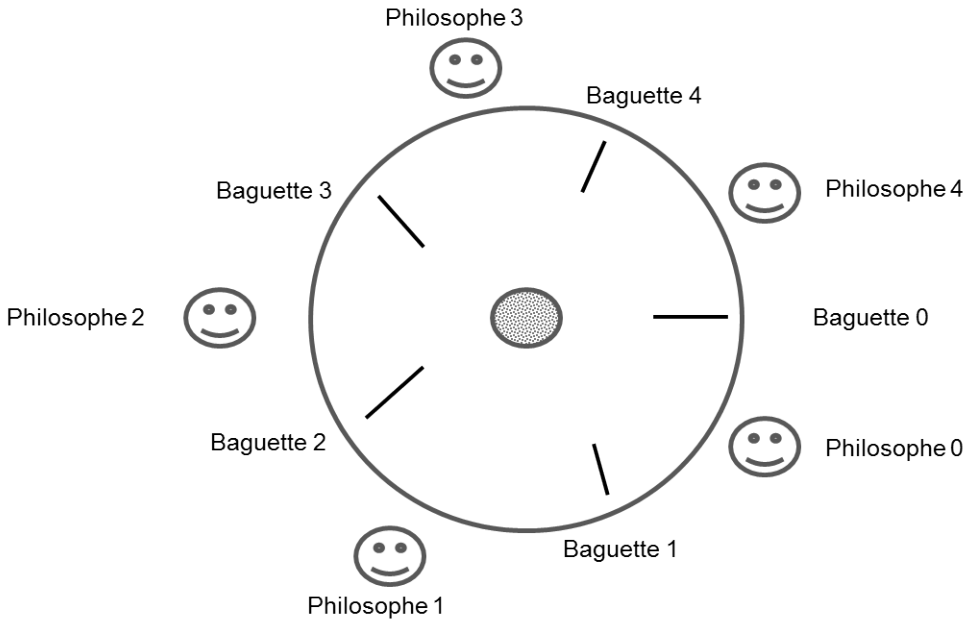

Fig. 5.3 Formalisation du problème des philosophes

Processus Philosophe i
Repeat
Penser;
Wait(Baguette[i]);
Prendre baguette de droite;
Wait(Baguette[(i + 1) mod 5]);
Prendre baguette de gauche;
Manger;
Déposer baguette de droite;
Signal(Baguette[i]);
Déposer baguette de gauche;
Signal(Baguette[(i + 1) mod 5]);
Until false

3. Les quatre conditions nécessaires pour l'occurrence de l'interblocage sont réunies dans cette solution à savoir:

- Exclusion mutuelle: Cette propriété concerne chaque baguette. Elle doit être utilisée en exclusion mutuelle.
- Occupation et attente: Chaque philosophe peut se retrouver dans la situation où il occupe une baguette et demande l'acquisition de la deuxième baguette.

- Pas de réquisition: On a pas les moyens pour réquisitionner une baguette occupée par un philosophe même s'il est en attente.
- Attente circulaire: L'allocation des baguettes aux philosophes peut mener à la situation où chaque philosophe i attend que le philosophe ((i+1) mod 5) libère sa baguette pour qu'il puisse la prendre. Ce qui forme une attente circulaire de tous les philosophes.

4. Nous pouvons atteindre dans cette solution une situation d'interblocage. En effet, si tous les philosophes prennent en même temps la baguette de droite. Ils vont se bloquer mutuellement, puisque chaque philosophe, pour pouvoir manger, attend la baguette de gauche détenue par son voisin.

5.1.1.2 Solution à base des moniteurs

- Dinner-philosophes1: est le nom du moniteur.
- Prendre() et Déposer(): sont des procédures du moniteur Dinner-philosophes1.

1. Déclaration du moniteur **Dinner-philosophes1**

```
Type Dinner-philosophes = Monitor
        Var baguette-disponible : array[0..4] of condition ;
        Var baguette-libre : array[0..4] of 0..1;
Procedure Entry prendre(i: 0..4)
Begin
      If ( baguette-libre[i] ≠ 1) Then
         wait(baguette-disponible[i]);
      baguette-libre[i]:= baguette-libre[i] – 1;
End;
Procedure Entry poser(i:0..4)
Begin
      baguette-libre[i]:= baguette-libre[i] + 1;
      signal(baguette-disponible[i]) ;
End;
Begin
   For i := 0 to 4 Do baguette-libre[i] := 1;
End;
End Dinner-philosophes1.
```

2. Comportement d'un processus Philosophe

Processus Philosophe i
While (vrai) **Do**

Penser;
Dinner-philosophes1.prendre(i);
Dinner-philosophes1.prendre((i+1) mod 5);
Manger;
Dinner-philosophes1.poser(i);
Dinner-philosophes1.poser((i+1) mod 5);
EndWhile

Remarque: Dans cette solution, nous aurions pu intégrer dans la procédure *poser* la libération des deux baguettes. Dans ce cas le paramètre de la procédure fera référence au philosophe appelant.

5.1.2 Deuxième protocole:

Les philosophes se sont rendus compte du problème de l'interblocage. Pour éviter ce problème, le philosophe 2 propose le protocole suivant:

Comportement d'un philosophe
Repeat
Penser;
Vérifier si les deux baguettes sont disponibles;
Si c'est le cas **alors** les prendre immédiatement
Sinon attendre qu'elles soient toutes les deux disponibles;
Manger;
Déposer les baguettes et faire un signe aux voisins en attente des baguettes;
Until false

5.1.2.1 Solution à base des sémaphores

Var BaguetteDisponible : Array[0..4] **of** Boolean Initialisée à [True,True,True,True,True]
AttendreBaguettes : Array[0..4] **of** Boolean Initialisée à [False,False,False,False,False]
mutex : Sémaphore initialisée à 1 /* Pour protéger la manipulation des variables
semphilosophe : Array[0..4] of sémaphore init. [0,0,0,0,0] /* sémaphores de blocage

Sémantique des variables:

- BaguetteDisponible[i] = True signifie que baguette i est disponible
- Pour philosophe i, AttendreBaguette[i] = true signifie que philosophe i est en attente que les deux baguettes soient disponibles

- semphilosophe[i] est un sémaphore de blocage derrière lequel le philosophe i se bloquera s'il trouve que les baguettes ne sont pas toutes les deux disponibles.

Remarque: Seul philosophe i exécute un wait sur semphilosophe[i], on dit que semphilosophe[i] est un sémaphore privé.

```
Comportement du Philosophe i
Repeat
        Penser;
        wait(mutex);
            If not (BaguetteDisponible[i] and BaguetteDisponible[(i+1) mod 5] Then
              AttendreBaguettes[i] := True;
          Else
              BaguetteDisponible[i] := False;
              BaguetteDisponible[(i+1) mod 5] := False
        EndIf
        signal(mutex);
        If AttendreBaguettes[i] Then wait(semphilosophe[i]) EndIf;
        Prendre baguette[i];
        Prendre baguette[(i+1) mod 5];
        Manger;
        wait(mutex);
        BaguetteDisponible[i] := True;
        BaguetteDisponible[(i+1) mod 5] := True;
        Déposer baguette[i];
        Déposer baguette[(i+1) mod 5];
            if AttendreBaguettes[(i+1) mod 5] and BaguetteDisponible[(i+2) mod 5] then
              BaguetteDisponible[(i+1) mod 5] = False;
              BaguetteDisponible[(i+2) mod 5] = False;
              signal(semphilosophe[(i+1) mod 5]);
        EndIf;
            if AttendreBaguettes[(i-1) mod 5] and BaguetteDisponible[(i-1) mod 5] then
              BaguetteDisponible[i] = False;
              BaguetteDisponible[(i-1) mod 5] = False;
              signal(semphilosophe[(i-1) mod 5]);
        EndIf; //
        signal(mutex);
  Until False ;
```

5.1.2.2 Solution à base des moniteurs

- Dinner-philosophes2: est le nom du moniteur.
- Prendrebaguettes() et Déposerbaguettes(): sont des procédures du moniteur Dinner-philosophes2.

Comportement de philosophe i
While (true) **Do**
penser;
Dinner-philosophes2.Prendrebaguettes(i);
manger;
Dinner-philosophes2.Déposerbaguettes(i);
EndWhile
Déclaration du moniteur **Dinner-philosophes2**

Type Dinner-philosophes2 = **Monitor**
Type état = {pense, faim, mange};
Var phétat:array[0..4] of état;
Var phcondition:array[0..4] of condition;
Procedure Entry prendrebaguettes(i:0..4)
Begin
phétat[i] := faim;
test(i);
If (phétat[i]$\neq$ *mange*)**Then***phcondition*[*i*].*wait*;
End;
Procedure Entry Déposerbaguettes(i:0..4)
Begin
phétat[i :]= pense;
test((i+4) mod 5);
test((i+1) mod 5);
End;
Procedure test(i:0..4)
Begin
If ((phétat[(i+4) mod 5] $\neq$ *mange*)**and**
(phétat[(i+1) mod 5] $\neq$ *mange*)**and**
(phétat[i] = faim))
Then
phétat[i] := mange;
phcondition[i].signal;
EndIf
End;
Begin
For i := 0 to 4 **Do** phétat[i] := pense;
End;
End Dinner-philosophes2.

Sur le plan fonctionnel, un philosophe qui cherche à manger appelle la procédure **test**, s'il sort de **test** sans que son état ne devienne **mange**, il doit attendre jusqu'à l'appel de **phcondition[i].signal()** par l'un de ses voisins.

Remarque:
Après une longue durée, le philosophe 4 s'est rendu compte qu'il n'est pas parvenu

à manger par manque de baguettes alors que les autres philosophes mangeaient régulièrement. En effet il a constaté qu'ils mangeaient selon le cycle représenté dans la figure 5.4. Il a conclu que les autres philosophe ont établi une stratégie de libération des baguettes pour le faire mourir de faim !

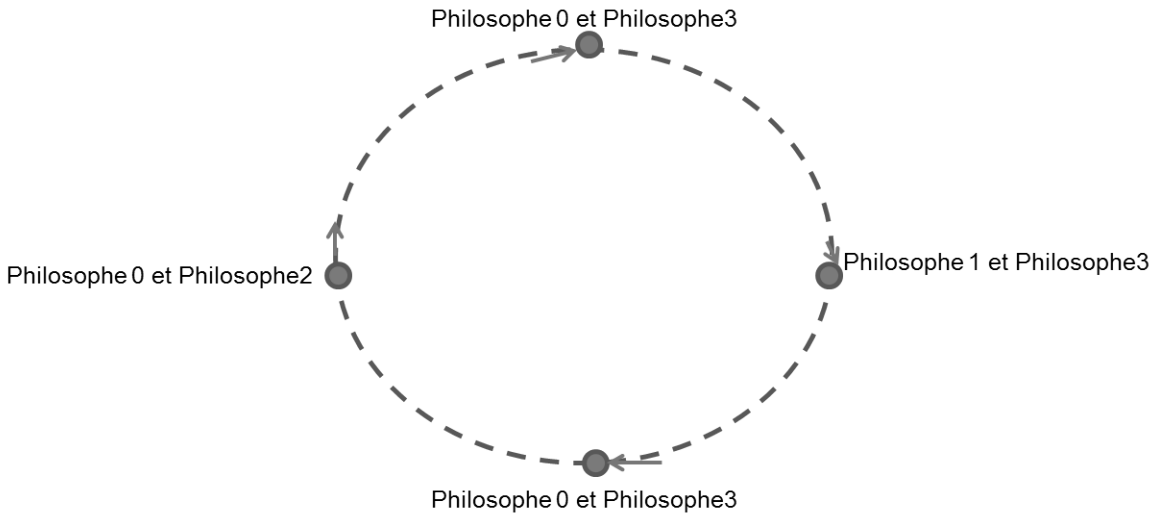

Fig. 5.4 Cycle provoquant la famine de Philosophe 4

5.1.3 Troisième protocole:

Le philosophe 3 a proposé de changer le protocole et de laisser la prise des baguettes comme dans la première solution tout en limitant le nombre de philosophes en compétition pour l'acquisition des baguettes à 4.

5.1.3.1 Solution à base des sémaphores

- Ajout d'un sémaphore de comptage **Comp** qui limite à 4 le nombre de philosophes qui rentrent en compétition pour l'acquisition des baguettes.
- **Var** Comp: Sémaphore initialisé à 4

Comportement de Philosophe i
While (true) **Do**
penser();
wait(Comp);
wait(baguette[i]);
Prendre baguette de droite;
wait(baguette[(i+1) mod 5]);
Prendre baguette de gauche;
manger();
Déposer baguette de droite;
signal(baguette[i]);
Déposer baguette de gauche;
signal(baguette[(i+1) mod 5]);
signal(Comp)
EndWhile

5.1.3.2 Solution à base des moniteurs

- Dinner-philosophes3 est le nom du moniteur.
- Prendrebaguetted(), prendrebaguetteg() et Déposerbaguettes() sont des procédures du moniteur Dinner-philosophes3.

Comportement de philosophe i
While (true) **Do**
penser
Dinner-philosophes3.Prendrebaguetted(i)
Dinner-philosophes3.Prendrebaguetteg(i)
manger;
Dinner-philosophes3.Déposerbaguettes(i)
EndWhile

Remarque: La valeur du parametre *i* dans l'appel des procédures du moniteur fait référence au numéro du philosophe appelant et non au numéro de la baguette.

Déclaration du moniteur **Dinner-philosophes3**

Type Dinner-philosophes3 = **Monitor**
Var Baguettelibre: array[0..4] of boolean;
Nbrencomp: int;
Attendrebaguette: array[0..4] of condition;
Compfaible: condition;

```
Procedure Entry prendrebaguetted(i:0..4)
Begin
        if Nbrencomp = 4 Then Compfaible.wait ;
        Nbrencomp++;
        if not Baguettelibre[i] Then Attendrebaguette[i].wait;
        Baguettelibre[i] := false ;
End;
Procedure Entry prendrebaguetteg(i:0..4)
Begin
        if not Baguettelibre[(i+1) mod 5] Then Attendrebaguette[(i+1) mod 5].wait;
        Baguettelibre[(i+1) mod 5] := false ;
End;
Procedure Entry Déposerbaguettes(i:0..4)
Begin
        Baguettelibre[i] := true;
        Baguettelibre[(i+1) mod 5] := true;
        Nbrencomp–;
        Attendrebaguette[i].signal;              /* réveil éventuel du philosophe (i+4) mod 5
        Attendrebaguette[(i+1) mod 5].signal; /* réveil éventuel du philosophe (i+1) mod 5
        Compfaible.signal;                       /* réveil du cinquième philosophe s'il désire
                                                    rentrer en compétition
End;
Begin
    Nbrencomp = 0;
    For i := 0 to 4 Do Baguettelibre[i] = true;
End;
End Dinner-philosophes3.
```

Cette Solution ne souffre ni de blocage ni de famine, cependant elle réduit la concurrence entre les processus philosophes.

5.1.4 Quatrième protocole:

Malgré que le protocole proposé par le troisième philosophe résout le problème tout en assurant l'exclusion mutuelle pour l'utilisation des baguettes, l'absence de l'interblocage et l'absence de famine, malgré cela le philosophe 4 a commenté son désagrément comme suit:

- En général, nous ne rentrons pas tous en compétition pour l'acquisition des baguettes. Et même si cela arrive, il est rare que chacun de nous prenne la baguette qui est sur sa droite en même temps que son voisin.
- Autoriser la compétition de tous les philosophes augmentera l'exploitation des baguettes et nous permettra de réduire notre attente pour manger.

Ainsi, le philosophe 4 proposa le protocole suivant:

- Vous tous, vous prenez les baguettes dans l'ordre droite puis gauche, sauf moi, je prendrai la baguette qui est sur ma gauche puis celle qui est sur ma droite.

5.1.4.1 Solution à base des sémaphores

Comportement du philosophe i i=0..3

```
Repeat
        Penser;
        Wait(Baguette[i]);
        Prendre baguette de droite;
        wait(Baguette[i + 1]);
        Prendre baguette de gauche;
        Manger;
        Déposer baguette de droite;
        Signal(Baguette[i]);
        Déposer baguette de gauche;
        Signal(Baguette[i + 1]);
  Until false
```

Comportement du philosophe 4

```
Repeat
        Penser;
        Wait(Baguette[0]);
        Prendre baguette de gauche;
        wait(Baguette[4]);
        Prendre baguette de droite;
        Manger;
        Déposer baguette de gauche;
        Signal(Baguette[0]);
        Déposer baguette de droite;
        Signal(Baguette[4]);
  Until false
```

5.1.4.2 Solution à base des moniteurs

Le moniteur qui implémente ce protocole est identique au moniteur Dinner-philosophes1. La différence réside dans les codes des processus philosophes qui sont comme suit:

Comportement du philosophe i tel que i=0..3

```
Repeat
        Penser;
        Dinner-philosophes1.Prendrebaguette(i);
        Dinner-philosophes1.Prendrebaguette(i+1);
        manger;
        Dinner-philosophes1.Déposerbaguettes(i);
  Until false
```

Comportement du philosophe 4

```
Repeat
        Penser;
        Dinner-philosophes1.Prendrebaguette(0);
        Dinner-philosophes1.Prendrebaguette(4);
        manger;
        Dinner-philosophes1.Déposerbaguettes(4);
  Until false
```

Remarque: La valeur du paramètre de la procédure Prendrebaguette fait référence au numéro de la baguette alors que celui de la procédure Déposerbaguettes fait référence au numéro du philosophe.

5.1.5 Cinquième protocole:

La solution proposée par le philosophe 4 n'a pas fait l'unanimité. En effet, le philosophe 0 a eu le sentiment que le philosophe 4 a proposé cette solution pour se distinguer des autres philosophes, ce qui pourrait ouvrir son appétit pour d'autres ambitions, se proclamer le futur chef par exemple. Afin de remettre les choses à leurs places tout en assurant la continuité dans les cycles d'évolution de chaque philosophe, le philosophe 0 proposa le protocole suivant:

- Adopter le premier protocole pour l'acquisition des baguettes, c-à-d chaque philosophe prendra la baguette qui est sur sa droite puis celle qui est sur sa gauche.
- Désigner une tierce personne du restaurant qui supervisera l'état des philosophes. Si un interblocage se produit, cette personne désignera le philosophe qui doit déposer sa baguette pour débloquer la situation. Le philosophe concerné reprendra la tentative d'acquisition de la baguette.
- La personne qui arbitre la scène, doit choisir un philosophe tout en assurant l'équité du choix.

Point de vue implémentation, cette personne est un processus que nous appelons **Processus arbitre**.

5.1.5.1 Solution à base des sémaphores

Les variables utilisée dans

- BaguetteDroitePh: array[0..4] of Boolean, initialisée à [false, false, false, false, false, false]. BaguetteDroitePh[i]=True signifie que le philosophe i détient la baguette qui est sur sa droite, c'est à dire la baguette i. Le processus Arbitre saura qu'il y a interblocage s'il trouve que BaguetteDroitePh[i]=True pour i=0..4.
- BaguetteRequisitionnée: array[0..4] of Boolean, initialisée à [false, false, false, false, false, false]. Cette variable est mise à jour par le processus arbitre lorsqu'il détecte un interblocage. La sémantique de cette variable est que si le philosophe i trouve que BaguetteRequisitionnée[i]=true, il doit déposer sa baguette est réveiller le philisophe (i-1) mod 5, c'est à dire (i+4) mod 5.
- mutex: sémaphore d'exclusion mutuelle (initialisée à 1). Permet de protéger les variables partagées.
- Baguette: array[o..4] of sémaphore initailisée à [1,1,1,1,1]. Chaque sémaphore Baguette[i] est associé à la baguette i.

```
Comportement de philosophe i
    Var faim: Boolean init. à false
Repeat
        wait(mutex);
        If      BaguetteRequisitionnée[i] Then
                BaguetteRequisitionnée[i] := false;
                Déposer baguette de droite;
                BaguetteDroitePh[i] := false;
                Signal(Baguette[i]);            /*Le philosophe (i+4) mod 5 est réveillé.
        EndIf;
        signal(mutex);
        If not faim Then Penser; faim := true EndIf;
        Wait(Baguette[i]);
        wait(mutex);
        BaguetteDroitePh[i] := true;
        signal(mutex);
        Prendre baguette de droite;
        Wait(Baguette[(i + 1) mod 5]);
        If      not BaguetteRequisitionnée[i] Then
                Prendre baguette de gauche;
                Manger;
                Déposer baguette de droite;
                wait(mutex);
                BaguetteDroitePh[i] := false;
                signal(mutex);
                Signal(Baguette[i]);
                Déposer baguette de gauche;
                Signal(Baguette[(i + 1) mod 5]);
                faim := false;
        EndIf;
Until false
```

Processus arbitre

Var interblocage: Boolean;
tour: 0..4, initialisée aléatoirement par une valeur du domaine [0..4].
Repeat
Delay(15 mn);
wait(mutex);
interblocage := (BaguetteDroitePh[0] and
BaguetteDroitePh[1] and
BaguetteDroitePh[2] and
BaguetteDroitePh[3] and
BaguetteDroitePh[4])
signal(mutex);
If interblocage **Then**
wait(mutex);
BaguetteRequisitionnée[tour] := true; /*Le philosophe tour saura qu'il doit déposer sa baguette
signal(mutex);
signal(Baguette[(tour + 1) mod 5]); /*Le philosophe tour est réveillé
tour = (tour + 1) mod 5;
EndIf
Until false

5.1.5.2 Solution à base des moniteurs

Comportement du philosophe i tel que i=0..4
Repeat
Penser;
Dinner-philosophes5.Prendrebaguetted(i);
Dinner-philosophes5.Prendrebaguetteg(i);
Dinner-philosophes5.Libérerbaguetteréquisitionnée(i);
Dinner-philosophes5.Récupérerbaguettes(i);
manger;
Dinner-philosophes5.Déposerbaguettes(i);
Until false

Comportement du processus Arbitre
Repeat
Delay(15 min);
Dinner-philosophes5.TesterInterblocage
Until false

Remarque: La valeur du paramètre des procédures fait référence au numéro du philosophe.

Déclaration du moniteur **Dinner-philosophes5**

```
Type Dinner-philosophes5 = Monitor
        Var BaguetteRéquisitionnée: array[0..4] of boolean;
            Tour: int;
            interblocage : Boolean ;
            Attendrebaguette: array[0..4] of condition;
            Baguettelibre: array[0..4] of Boolean;
            Phoccupebagetted: array[0..4] of Boolean;
Procedure Entry prendrebaguetted(i:0..4)
Begin
        If not Baguettelibre[i] Then Attendrebaguette[i].wait;
        Baguettelibre[i] := false ;
        Phoccupebagetted[i] := true;
End;
Procedure Entry prendrebaguetteg(i:0..4)
Begin
        If not Baguettelibre[(i+1) mod 5] Then Attendrebaguette[(i+1) mod 5].wait;
        Baguettelibre[(i+1) mod 5] := false ;
End;
Procedure Entry Libérerbaguetteréquisitionnée(i:0..4)
Begin
        If      baguetteréquisitionnée[i] Then
                Phoccupebagetted[i] := false;
                Attendrebaguette[i].signal ;
        EndIf
End;
Procedure Entry Récupérerbaguettes(i:0..4)
Begin
        If      baguetteréquisitionnée[i] Then
                baguetteréquisitionnée[i] := false;
                Attendrebaguette[i].wait;
                Phoccupebagetted[i] := true;
                Attendrebaguette[(i+1) mod 5].wait;
        EndIf
End;
Procedure Entry Déposerbaguettes(i:0..4)
Begin
        Baguettelibre[i] := true;
        Baguettelibre[(i+1) mod 5] := true;
        Attendrebaguette[i].signal;              /* réveil éventuel du philosophe (i+4) mod 5
        Attendrebaguette[(i+1) mod 5].signal; /* réveil éventuel du philosophe (i+1) mod 5
End;
```

```
Procedure Entry Testerinterblocage
Begin
        interblocage := ( Phoccupebagetted[0] and
                          Phoccupebagetted[1] and
                          Phoccupebagetted[2] and
                          Phoccupebagetted[3] and
                          Phoccupebagetted[4] )
            If interblocage then
              baguetteréquisitionnée[tour] := true ;
              Attendrebaguette[tour].signal ;
              tour := (tour + 1) mod 5 ;          /* Prochain philosophe qui sera touché
                                                     par la réquisition de sa baguette droite
        EndIf
End;
Begin
        interblocage := false ;
              For i := 0..4 Do
                  baguetteréquisitionnée[i] := false ;
                  Phoccupebagetted[i] := false ;
                  Baguettelibre[i] := true;
        EndFor ;
  End Dinner-philosophes5.
```

Remarques

- Nous pouvons remarquer que dans ce cinquième protocole, le processus philosophe concerné par la réquisition de sa baguette de droite ne sera concerné de nouveau par la réquisition qu'après l'acquisition des deux baguettes. En effet, le cas extrême est que ce processus restera bloqué après l'acquisition de sa baguette de droite et durant l'occurrence d'une succession d'interblocages. Avec la politique choisie, le philosophe qui détient sa baguette de gauche sera concerné par la réquisition de sa baguette et réveillera ce philosophe. Ce dernier obtiendra donc la baguette manquante et passera à l'opération **manger**.
- Nous parrellons que les quatres conditions nécessaires à l'occurrence de l'interblocage sont:

  1. Exclusion mutuelle;
  2. Occupation et attente;
  3. Attente circulaire;
  4. Pas de réquisition.

  Pour prévenir l'occurrence de l'interblocage, il suffit de supprimer l'une des conditions 2, 3 et 4.
  Dans l'exemple des philosophes, les protocoles 2, 3, 4 et 5 ne souffrent pas de ce problème. Une lecture minutieuse des solutions proposées montre que:

  – Le protocole 2 supprime la condition d'occupation et attente;
  – Le protocole 3 et le protocole 4 suppriment l'attente circulaire;

– Le protocole 5 autorise la réquisition des ressources.

5.2 Paradigmes de lecteurs rédacteurs

5.2.1 Description du problème

Plusieurs processus doivent accéder à une ressource partagée. Ces processus sont regroupés en deux catégories. La première catégorie représente les processus qui ne peuvent que lire les données à partir de cette ressource. Ils sont appelés les lecteurs. La seconde catégorie représente les processus qui peuvent modifier les données de cette ressource. Ils sont nommés les rédacteurs.
En effet, le problème des lecteurs-rédacteurs est un problème classique de synchronisation qui autorise à plusieurs lecteurs d'accéder simultanément aux données de la ressource. Par contre, il interdit l'accès simultané des rédacteurs entre eux et entre les lecteurs et les rédacteurs.
Ainsi, Que faire si plusieurs lecteurs sont en train de lire et un rédacteur veut modifier. La réponse à cette question fait apparaître trois variantes possibles au problème des lecteurs-rédacteurs:

- **Priorité aux lecteurs:** Ne pas faire attendre les nouveaux lecteurs.
- **Priorité aux rédacteurs:** Faire attendre les nouveaux lecteurs et donné l'accès au rédacteur en attente.
- **Priorité égale:** l'accès est donné selon l'ordre d'arrivée.

5.2.2 Solution avec priorité aux lecteurs

5.2.2.1 A base de sémaphores

- **BD:** Pour base de Données et qui représente la ressource partagée.
- **Sémaphore binaire ecrire:** Permet l'accès exclusif d'un rédacteur pour modifier la BD.
- premier lecteur exécute wait(ecrire) pour empêcher à tout rédacteur d'entrer dans sa SC et le dernier lecteur qui quitte la BD doit relâcher le sémaphore ecrire avec signal(ecrire).
- Besoin d'un compteur lecteurs pour déterminer le premier et le dernier lecteur.
- Besoin d'un 2ième sémaphore mutex pour protéger le compteur lecteurs.

1. Déclaration des variables:

Var ecrire, mutex: sémaphore init. 1;
Lecteurs: integer init.0;

2. Solution

Processus Lecteur
while(true)
{
wait(mutex);
lecteurs++;
if (lecteurs == 1) wait(ecrire);
signal(mutex);
lire base de données;
wait(mutex);
lecteurs–;
if (lecteurs == 0) signal(ecrire);
signal(mutex);
}

Processus Rédacteur
while(true)
{
wait(ecrire);
Accès base de données(SC);
signal(ecrire);
}

3. **Remarque:** Dans cette solution si des rédacteurs sont en attente, le premier lecteur qui arrivera ainsi que les lecteurs qui arriveront par la suite attendent que ces rédacteurs accèdent à la BD.

5.2.2.2 A base de moniteurs

Processus Lecteur
{
Lectredpriolect.Debutlire();
Accès en lecture à BD;
Lectredpriolect.Finlire();
}

Processus Rédacteur
{
Lectredpriolect.Debutecrire();
Accès en écriture à BD;
Lectredpriolect.Finecrire();
}

Le code du moniteur

```
Type lectredpriolect = Monitor
   Var flect , fecr: Condition;
        nblec: Integer // nbre lecteurs dans la base
        ecr: Boolean // True si une écriture est en cours
   Procedure entry debutlire( )
   {
     nblec ++;
     if (ecr) flect.wait( ); //s'il y a un écrivain on attend
     flect.signal( ); // si on peut lire alors tous ceux qui sont en attente aussi
   }
   Procedure entry finlire( )
   {
     nblec – ;
     if (nblec ==0) fecr.signal( );
   }
   Procedure entry debutecrire( )
   {
     if (nblec ¿ 0 or ecr) fecr.wait( );
     ecr = true;
   }
   Procedure entry finecrire( )
   {
     ecr = false;
     if ( nblec ¿ 0) flect.signal( );
     else fecr.signal( );
   }
   Initialisation( )
   {
     nblec = 0;
   }
```

5.2.3 Solution avec priorité aux rédacteurs

5.2.3.1 A base de sémaphores

1. Déclaration des variables:
   Var lire, mutexl, mutexr: sémaphore init. 1;
        lecteurs, redacteurs: integer init.0;

2. Solution
   Processus Lecteur

```
    while(true)
    {
            wait(lire);
            wait(mutexl);
            lecteurs++;
            if (lecteurs == 1) wait(ecrire);
            signal(mutexl);
            signal(lire);
            lire base de données;
            wait(mutexl);
            lecteurs–;
            if (lecteurs == 0) signal(ecrire);
            signal(mutexl);
    }
```

Processus Rédacteur

```
    while(true)
    {
            wait(mutexr);
            rédacteurs++;
            if (rédacteurs == 1) wait(lire);
            signal(mutexr);
            wait(ecrire);
            Accès base de données(SC);
            signal(ecrire);
            wait(mutexr);
            rédacteurs–;
            if (rédacteurs == 0) signal(lire);
            signal(mutexr);
    }
```

5.2.3.2 A base de moniteurs

Processus Lecteur

```
    {
      Lectredpriored.Debutlire();
      Accès en lecture à BD;
      Lectredpriored.Finlire();
    }
```

Processus Rédacteur

```
    {
      Lectredpriored.Debutecrire();
      Accès en écriture à BD;
      Lectredpriored.Finecrire();
    }
```

Le code du moniteur

```
Type lectredpriored = Monitor
    Var flect , fecr: Condition;
        nblecbase: integer //compte le nombre de lecteurs dans la base de données
        nblecatt: integer //compte le nombre de lecteurs en attente d'une lecture dans la BD
        nbecratt: integer //compte le nombre d'écrivains en attente d'écriture
        ecrbase: boolean //indique si on est en train d'écrire dans la base
    Procedure entry debutlire( )
    {
      nblecatt++;
      if (ecrbase or nbecratt ¿ 0) flect.wait( );
      nblecatt–;
      flect.signal( ); //Réveil en chaine des lecteurs
      nblecbase++;
    }
    Procedure entry finlire( )
    {
                                nblecbase – ;
                                if (nblecbase ==0) fecr.signal( );
    //on réveille un rédacteur }
    Procedure entry debutecrire( )
    {
      nbecratt++;
      if (nblecbase ¿ 0 or ecrbase) fecr.wait( );
      ecrbase = true;
      nbecratt–;
    }
    Procedure entry finecrire( )
    {
      ecrbase = false;
      if ( nbecratt ¿ 0) fecr.signal( );
      else flect.signal( );
    }
    Initialisation( )
    {
      nblecbase = 0;
      nblecatt = 0;
      nbecratt = 0;
      ecrbase = false;
    }
```

5.2.4 Solution avec priorité égale

5.2.4.1 A base de sémaphores

1. Déclaration des variables:
   Var fifo, mutexG, mutexL: sémaphore init. 1;
   NL: integer init.0;

2. Solution

Processus Lecteur
while(true)
{
wait(fifo);
wait(mutexL);
NL++;
if (NL == 1) wait(mutexG);
signal(mutexL);
signal(fifo);
lire base de données;
wait(mutexL);
NL–;
if (NL == 0) signal(mutexG);
signal(mutexL);
}

Processus Rédacteur
while(true)
{
wait(fifo);
wait(mutexG);
signal(fifo);
Accès base de données(SC);
signal(mutexG);
}

5.2.4.2 A base de moniteurs

Processus Lecteur
{
Lectredprioegal.Debutlire();
Accès en lecture à BD;
Lectredprioegal.Finlire();
}

Processus Rédacteur

```
{
  Lectredprioegal.Debutecrire();
  Accès en écriture à BD;
  Lectredprioegal.Finecrire();
}
```

Le code du moniteur

Type lectredprioegal = Monitor
Var flect , fecr: Condition;
nblecbase: integer //compte le nombre de lecteurs dans la base de données
nblecatt: integer //compte le nombre de lecteurs en attente d'une lecture dans la BD
nbecratt: integer //compte le nombre d'écrivains en attente d'écriture
ecrbase: boolean //indique si on est en train d'écrire dans la base
tourlect: boolean //vraie si le tour est aux lecteurs

```
Procedure entry debutlire( )
{
  nblecatt++;
  if (ecrbase or (nbecratt ¿ 0 and not tourlect)) flect.wait( );
  nblecatt–;
  flect.signal( ); //Réveil en chaine des lecteurs
  nblecbase++;
}
Procedure entry finlire( )
{
  nblecbase – ;
  if (nblecbase ==0) {tourlect = false; fecr.signal( );} //on réveille un rédacteur
}
Procedure entry debutecrire( )
{
  nbecratt++;
  if (nblecbase ¿ 0 or ecrbase) fecr.wait( );
  ecrbase = true;
  nbecratt–;
}
Procedure entry finecrire( )
{
  ecrbase = false;
  tourlect = true;
  if ( nbecratt ¿ 0) flect.signal( );
  else fecr.signal( );
}
Initialisation( )
```

```
{
  nblecbase = 0;
  nblecatt = 0;
  nbecratt = 0;
  ecrbase = false;
  tourlect = true; // indifféremment
}
```

5.3 Paradigme des producteurs et des consommateurs

Un processus producteur produit des données consommées par un processus consommateur. Par exemple, en compilation, un assembleur produit des modules objets consommés par un éditeur de lien.
Pour cela, nous avons besoin d'un tampon pour contenir les items (données) produits et qui seront ensuite pris pour se faire ensuite consommer.
Ainsi, il est important de distinguer les cas où le tampon est de taille limitée ou illimitée.

5.3.1 Cas du tampon de taille illimitée

- Considérons d'abords un tampon (buffer) de taille illimitée qui comprend un tableau linéaire d'éléments de type item.
- in pointe au prochaine emplacement pour recueillir le prochain item produit.
- out pointe au prochain item à être consommé.

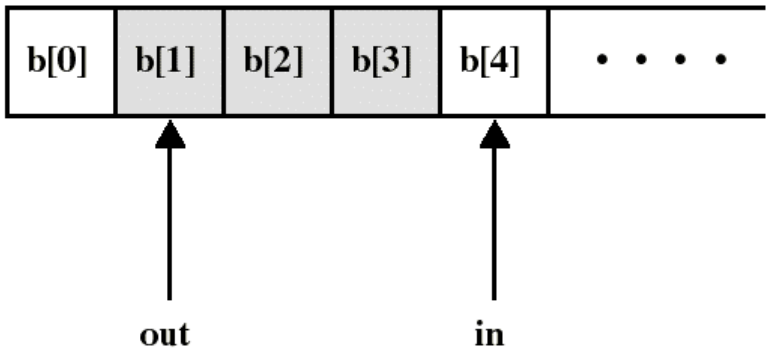

Fig. 5.5 Tampon de taille illimitée

- Besoin d'assurer une exclusion mutuelle pour accéder aux cases du tampon.
- Comment assurer que le consommateur ne consomme pas ce qui n'est pas encore produit?

5.3.1.1 Solution à base de sémaphores

- Comment assurer que le consommateur ne consomme pas ce qui n'est pas encore produit?
  - Un sémaphore nbr, qui renseigne sur le nombre d'items produits et pas encore consommés. A priori, le consommateur se bloquera initialement, donc nbr est un sémaphore de blocage (initialisé à 0).
  - Lorsque le producteur produit un item, il appelle signal(nbr).
  - Quand un consommateur veut consommer un item, il appelle wait(nbr).
- Le schéma de synchronisation proposé fait que le producteur et le consommateur n'accèdent jamais à la même case du tampon. De ce fait on ne sera pas obligé d'assurer une exclusion mutuelle pour l'accès au tampon.

1. **Les déclarations:**
   nbr: Sémaphore init à 0;
   in, out: integer init à 0;

2. **Processus Producteur**
   while (true)
   {
   v = produit();
   ajouter(v);
   signal(nbr);
   }
   Processus Consommateur
   while (true)
   {
   wait(nbr);
   w = **prendre();**
   consomme(w);
   }
3. **Les sections critiques:**
   ajouter(v)
   {
   b[in] = v;
   in++;
   }
   prendre()
   {
   w = b[out];
   out++;
   return w;
   }

5.3.1.2 Solution à base de moniteurs

Processus Producteur

```
while (true)
  {
    Produire(v);
    PC.ajouter(v);
  }
```

Processus Consommateur

```
while (true)
  {
    w = PC.prendre();
    Consomme(w);
  }
```

Le code du moniteur

```
Type PC = Monitor
  Type item;
  Var nbrcasespleines, in, out: integer;
      buffervide: condition;
      b:array[0..N] of item;
  Procedure entry ajouter(v:item)
  {
    b[in] = v;
    in++;
    nbrcasepleines++;
    buffervide.signal;
  }
  Procedure entry prendre()
  {
    if nbrcasepleines = 0 buffervide.wait;
    w = b[out];
    out++;
    nbrcasepleines–;
    return w;
  }
  Initialisation( )
  {
    nbrcasespleines = 0;
    in = 0;
    out = 0;
  }
```

5.3.2 Cas du tampon borné de taille k

- Le consommateur peut consommer seulement lorsque le nombre d'items (consommables) est au moins 1.
- Le producteur peut produire seulement lorsque le nombre d'espaces libres est au moins 1.

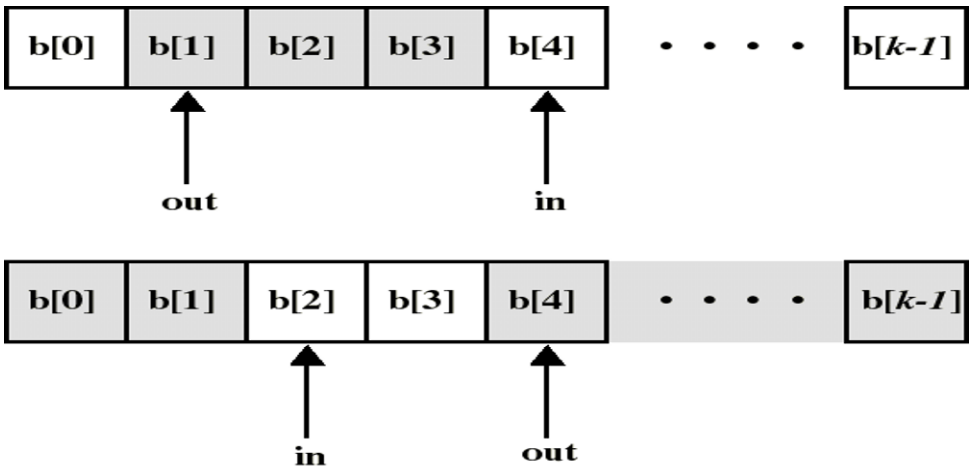

Fig. 5.6 Tampon borné de taille k

5.3.2.1 Solution à base de sémaphores

- Comme pour le tampon illimité, nous avons besoin d'un sémaphore de blocage nbr pour bloqué le consommateur au cas où y a pas d'items à consommer.
- En plus, dans la cas du tampon borné le producteur se bloque si le tampon est plein. Nous utilisons donc un sémaphore appelé vide initialisé à k pour synchroniser le producteur et le consommateur au niveau du nombre d'espaces libres. Vide est donc un sémaphore de comptage dont la valeur courante compte le nombre de cases vides.

1. **Les déclarations:**
   nbr: Sémaphore init à 0;
   vide: Sémaphore init à k;
   in, out: integer init à 0;

2. **Processus Producteur**

```
while (true)
{
  v = produit();
  wait(vide);
  ajouter(v);
  signal(nbr);
}
```

Processus Consommateur

```
while (true)
{
  wait(nbr);
  w = prendre();
  signal(vide);
  consomme(w);
}
```

3. **Les sections critiques:**

ajouter(v)

```
{
  b[in] = v;
  in = (in + 1)%k;
}
```

prendre()

```
{
  w = b[out];
  out = (out + 1)%k;
  return w;
}
```

5.3.2.2 Solution à base de moniteurs

Processus Producteur

```
while (true)
  {
    Produire(v);
    PC.ajouter(v);
  }
```

Processus Consommateur

```
while (true)
  {
    w = PC.prendre();
    Consomme(w);
  }
```

Le code du moniteur

```
Type PC = Monitor
    Type item;
    Var nbrcasespleines, in, out: integer;
        bufferplein, buffervide: condition;
        b:array[0..k-1] of item;
    Procedure entry ajouter(v:item)
    {
      if nbrcasepleines = k bufferplein.wait;
      b[in] = v;
      In = (in+1)%k;
      nbrcasepleines++;
      buffervide.signal;
    }
    Procedure entry prendre()
    {
      if nbrcasepleines = 0 buffervide.wait;
      w = b[out];
      out = (out+1)%k;
      nbrcasepleines–;
      bufferplein.signal;
      return w;
    }
    Initialisation( )
    {
      nbrcasespleines = 0;
      in = 0;
      out = 0;
    }
```

5.3.3 Généralisation à P producteurs et C consommateurs

Nous allons généraliser la solution à P producteurs et C consommateurs dans le cas d'un tampon borné de taille k.

5.3.3.1 Solution à base de sémaphores

- Tampon à k cases.
- Il faut protéger l'utilisation des indices. Pour celà, nous utilisons deux sémaphores d'exclusion mutuelle mutexprod et mutexcons.

1. **Les déclarations:**
   nbr: Sémaphore init à 0;
   vide: Sémaphore init à k;
   mutexprod, mutexcons: Sémaphore init à 1;
   in, out: integer init à 0;

2. **Processus Producteur**
   while (true)
   {
   v = produit();
   wait(vide);
   wait(mutexprod);
   ajouter(v);
   signal(mutexprod);
   signal(nbr);
   }
   Processus Consommateur
   while (true)
   {
   wait(nbr);
   wait(mutexcons);
   w = **prendre();**
   signal(mutexcons);
   signal(vide);
   consomme(w);
   }
3. **Les sections critiques:**
   ajouter(v)
   {
   b[in] = v;
   in = (in + 1)%k;
   }
   prendre()
   {
   w = b[out];
   out = (out + 1)%k;
   return w;
   }

5.3.3.2 Solution à base de moniteurs

La solution est la même pour un producteur et un consommateur puisque le tampon ainsi que les variables sont déclarés à l'intérieur du moniteur, ce qui assure et impose

une exclusion mutuelle pour l'accès aux procédures du moniteur. Mais cela induit moins de concurrence que pour la solution à base des sémaphores.

References

Silberschatz A, Galvin PB, Gagne G (2018) Operating System Concepts, 10th Edition, vol ISBN 978-1-118-06333-0. Wiley

Tanenbaum AS (2009) Modern operating systems, 3rd Edition, vol ISBN 0138134596. Pearson Prentice-Hall