

Chapitre 4: Moniteurs & Régions critiques

1- Introduction

Les sémaphores sont un outil de synchronisation de bas niveau ; il peut être utilisé pour réaliser de nouveaux outils de synchronisation. L'utilisation des sémaphores permet d'exprimer différentes solutions de synchronisation de manière flexible. Cependant, certains inconvénients apparaissent quant à la mauvaise utilisation de cet outil, notamment l'interblocage ou le blocage indéfini de processus ou de manière générale l'incorrection de la solution. Pour éviter ce problème, la conception d'outils de haut niveau s'impose. Les moniteurs et les régions critiques sont de tels outils.

2- Les moniteurs

2.1- Définition

Un moniteur [Brinch Hansen 73, Hoare 74] est un outil de synchronisation entre processus. Il est constitué principalement d'un ensemble de variables dites de *synchronisation* et des procédures qui manipulent ces variables. Certaines de ces procédures sont *internes* au moniteur et d'autres sont *externes*, donc accessibles par les processus. La synchronisation quant à l'accès aux ressources partagées est faite à l'aide des procédures externes appelées aussi *points d'entrées* ou simplement entrées du moniteur. Les variables de synchronisation ne sont utilisées par les processus que via les entrées. Le mécanisme de synchronisation garantie que ces variables soient utilisées en *exclusion mutuelle*. Une séquence d'initialisation du moniteur permet d'affecter des valeurs initiales avant que les processus les utilisent.

2.2- Structure

La structure générale d'un moniteur est donnée comme suit :

```

<nom du moniteur> : Moniteur ;
<partie déclaration des variables et autres>
[<partie déclaration des procédures internes>]

[entree procedure <nom1> (<paramètres1>) ;
  <corps1> ;
entree procedure <nom2> (<paramètres2>) ;
  <corps2> ;
  - -
entree procedure <nomn> (<paramètresn>) ;
  <corpsn> ;]

initialisation
Début
  <instruction(s)>
Fin
FinMoniteur.

```

La synchronisation s'exprime au moyen de conditions. Une condition est déclarée comme une variable sans "valeur", elle représente une raison de blocage. Le blocage et le réveil des processus s'effectuent à l'aide des primitives *signal()* et *wait()* exécutées au niveau des entrées du moniteur. La déclaration d'une condition est faite comme suit : *var c : condition* ; où c est l'identifiant de la condition.

- Primitive wait (): Elle s'écrit $c.wait()$ où c est une condition. Elle bloque le processus appelant derrière la condition c et en dehors du moniteur.
- Primitive signal (): Elle s'écrit $c.signal()$ où c est une condition. Elle réveille un processus bloqué derrière la condition c de manière FIFO s'il y a lieu sinon elle est sans effet. Pour assurer l'E.M au niveau du moniteur, le processus qui exécute cette primitive attend temporairement jusqu'à ce que le processus réveillé soit se bloque une autre fois sur une condition soit sort du moniteur.
- Primitive empty (): C'est une fonction booléenne, elle s'écrit $c.empty()$. Elle retourne la valeur *faux* s'il y a au moins un processus bloqué sur la condition c .

Afin de préciser davantage la sémantique de la primitive *signal ()*, nous nous proposons l'exemple suivant de trois processus qui utilisent le moniteur décrit ci-dessous:

```

M: Moniteur ();
c1,c2: condition;
entree procedure proc1();
Debut
  -
  c1.wait();
  c2.signal();
  -
Fin;
entree procedure proc2();
Debut
  -
  c2.wait();
  -
Fin;
entree procedure proc3();
Debut
  -
  c1.signal();
  -
Fin;
```

Processus P1;

- $M.proc1();$

Processus P2;

- $M.proc2();$

Processus P3;

- $M.proc3();$

Supposons l'ordre suivant d'appel des entrées du moniteur décrit comme suit. $P1$ commence par exécuter $M.proc1()$, il se bloque sur la condition $c1$ en dehors du moniteur. $P2$ exécute ensuite $M.proc2()$ et se bloque sur $c2$. Finalement, $P3$ exécute $M.proc3()$; il réveille alors $P1$ et se met en attente temporaire. $P1$ ne quitte pas le moniteur et ne se bloque pas sur une primitive *wait ()* mais plutôt réveille $P2$ bloqué sur $c2$, donc $P3$ reste en attente temporaire. $P2$ se met alors en attente temporaire. Quand $P2$ termine l'exécution de l'entrée en cours, $P1$ se réveille automatiquement et quand celui-ci termine l'exécution de l'entrée en cours, $P3$ se réveille pour finir l'exécution de son entrée en cours. Il est alors clair que la file d'attente temporaire est gérée de manière LIFO (*Last In First Out*).

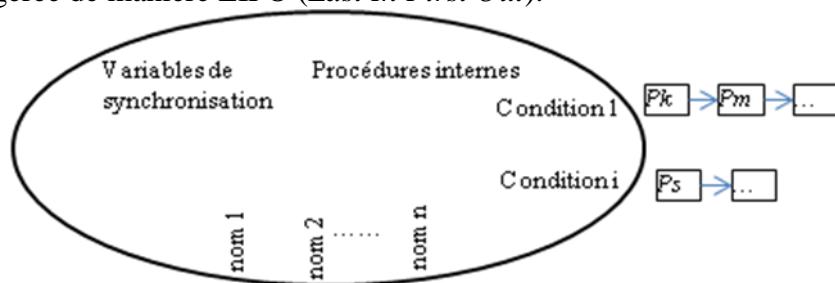


Figure 4.1 : Schéma synoptique d'un moniteur.

Un moniteur définit une clôture (voir Figure 4.1) qui englobe les variables partagées. Les entrées et les conditions font partie de cette clôture. Les processus en attente définissent des files, une par condition. Ils attendent en dehors du moniteur.

Un processus admis à exécuter une entrée du moniteur est considéré conceptuellement comme étant dans le moniteur. Le mécanisme de synchronisation admet alors un seul processus à la fois à l'intérieur du moniteur.

2.3- Illustrations

Exemple1 : Gestion d'une classe de ressources à n points d'accès. Chaque processus demande une seule ressource à la fois.

La solution est donnée comme suit :

```
Allocation : Moniteur ;
const n=... ;
var   nbr : entier ;
      c : condition ;
entree procedure dem () ;
Debut
  Si (nbr=0) Alors c.wait () Fsi ;
  nbr :=nbr-1
Fin ;

entree procedure lib () ;
Debut
  nbr :=nbr+1 ;
  c.signal ()
Fin ;

initialisation
Début
  nbr :=k
Fin
FinMoniteur.
```

Exemple 2 : Producteur et consommateur.

Deux solutions sont données ci-dessous, dont seule la seconde favorise l'accès simultané au tampon.

Solution 1 :

```
ProdCons1 : Moniteur ;
const  n=.... ;
type Tart=... ;
var tampon : tableau[0..n-1] de tart ;
t, q, cpt : entier ;
nonvide, nonplein : condition ;

entry procedure deposer (x : tart) ;
Debut
  Si (cpt=N) Alors nonplein.wait () Fsi ;
  tampon[q] ← x ;
  q :=(q+1) mod n ;
  cpt :=cpt+1 ;
  nonvide.signal ()
Fin ;

entree procedure prelever (Var x : tart) ;
```

```
Processus P ();
Debut
-
allocation.dem () ;
<utiliser ressource>
allocation.lib () ;
-
Fin.
```

Forme d'un processus

Processus Producteur ();

Debut

-
ProdCons1.deposer (x) ;

-
Fin.

Processus Consommateur ();

Debut

-
ProdCons1.prelever x();

-
Fin.

Debut

Si ($cpt=0$) *Alors* *nonvide.wait () fsi* ;
x := tampon[t] ;
t := (t+1) Mod n ;
cpt := cpt - 1 ;
nonplein.signal ()
Fin ;

Debut

cpt := 0 ; *t := 0* ; *q := 0*

Fin

FinMoniteur.

Remarques :

- t peut être remplacé par ($q - cpt$),
- Une seule condition peut suffire étant donné que les deux processus ne se bloquent pas en même temps.
- L'inconvénient majeur de cette solution est l'exclusion mutuelle au niveau du tampon. L'accès au tampon est devenu séquentiel alors qu'on devrait permettre l'accès simultané.

Solution 2 a:

ProdCons2 : Moniteur ;

Const n=.... ;
Var cpt : entier ;
nonvide, nonplein : condition ;

entree procedure dem_dep () ;

Debut

Si ($cpt=N$) *Alors* *nonplein.wait () Fsi*
Fin ;

entree procedure fin_dep () ;

Debut

cpt := cpt + 1 ; *nonvide.signal ()*
Fin ;

entree procedure dem_prel () ;

Debut

Si ($cpt=0$) *Alors* *nonvide.wait () fsi*
Fin ;

entree procedure fin_prel () ;

Debut

cpt := cpt - 1 ; *nonplein.signal ()*
Fin ;

Début

cpt := 0

Fin

FinMoniteur.

Forme d'un processus
Processus Producteur () ;
Debut

-
ProdCons2. dem_dep () ;
< Déposer >
ProdCons2. fin_dep () ;
-
Fin.

Processus Consommateur () ;
Debut

-
-
ProdCons2. dem_prel () ;
< Prélever >
ProdCons2. fin_prel () ;
-
-
Fin.

L'accès simultané au tampon est permis puisque le tampon n'est pas un objet du moniteur.

Solution 2 b:

Dans le cas de plusieurs producteurs et plusieurs consommateurs, l'accès exclusif au sein de chaque famille est nécessaire. Dans ce cas, on ajoute une variable d'état (booléen) pour chaque famille pour indiquer qu'un processus est en cours et une condition pour chaque famille pour l'attente momentanée dans le cas d'occupation du tampon par un membre de la famille. Deux niveaux d'autorisation apparaissent, le premier concerne les cases à utiliser et le second, l'existence de processus en cours. Cependant, si on garde le même compteur cpt, des autorisations excédantes en nombre peuvent être données aux processus. Ce problème est du au fait que la mise à jour de cpt ne peut se faire qu'à la fin du dépôt (resp. prélèvement). De ce fait, cpt est remplacé par deux compteurs nplein et nvide comptant respectivement le nombre de cases pleines et le nombre de cases vides. Ceci va permettre de décrémenter le compteur (nvide/ nplein) juste après le test de la condition qui l'implique afin d'assurer que le prochain processus de la famille prenne une case différente s'il ya lieu et ainsi éviter des autorisations excédantes. La solution complète est donnée comme suit :

```
ProdCons2 : Moniteur ;
const n=.... ;
var nonvide, nonplein, Pcourses, Ccourses : condition ;
      ProdEncours, ConsEncours : boolean ;
      Nvide, nplein : entier ;

entree procedure dem_dep ();
Debut
  Si (nvide=0) Alors nonplein.wait () Fsi ; nvide :=nvide-1 ;
  Si ProdEncours Alors Pcourses.wait Fsi ; ProdEncours :=vrai
Fin ;
entree procedure fin_dep ();
Debut
  nplein :=nplein+1 ; ProdEncours :=faux ; Pcourses.Signal; nonvide.signal ()
Fin ;
entree procedure dem_prel ();
Debut
  Si (nplein=0) Alors nonvide.wait () Fsi ; nplein :=nplein-1 ;
  Si ConsEncours Alors Ccourses.wait Fsi ; ConsEncours :=vrai
Fin ;
entree procedure fin_prel ();
Debut
  nvide :=nvide+1 ; ConsEncours :=faux ; Ccourses.Signal; nonplein.signal ()
Fin ;
Debut
  ProdEncours :=faux ; ConsEncours :=faux ; nvide :=n ; nplein :=0
Fin
FinMoniteur.
```

Exemple 3 : Lecteurs/ Rédacteurs sans priorité explicite.

La solution donnée ci-dessous s'inspire de celle exprimée à l'aide des sémaphores et qui est donnée dans le chapitre 2.

```
Lec_Red : Moniteur ;
var ecriture : booléen ; nl : entier ;
      L, E : condition ;
```

```

entree procedure deb_lecture () ;
  Debut
    nl := nl+1 ;
    Si écriture alors Si (nl=1) Alors E.wait () Sinon L.wait () Fsi; L.signal () Fsi
    Fin ;
  entree procedure fin_lecture ();
  Debut
    nl := nl-1 ; Si (nl=0) Alors E.signal () Fsi
    Fin ;
  entree procedure deb_ecriture () ;
  Debut
    Si écriture ou (nl<>0) Alors E.wait () Fsi ; écriture := vrai
    Fin ;
  entree procedure fin_ecriture () ;
  Debut
    écriture := faux ; E.signal ()
  Fin ;
  Debut
    nl := 0 ; écriture := faux
  Fin .

```

La vérification de la disponibilité de la ressource et sa réservation sont faite à l'aide de deux variables de synchronisation, *écriture*, un booléen qui indique si une écriture est en cours ou non et la variable *nl* qui indique le nombre de lecteurs dans le système.

Remarquons que tous les rédacteurs en attente et le premier lecteur (i.e. représentant de sa famille) en attente se bloquent sur la condition *E* dans l'ordre d'arrivée. Les autres lecteurs se bloquent sur la condition *L*. Ainsi, quand un rédacteur termine une écriture, il réveille le premier processus bloqué, celui-ci peut être un lecteur ou un rédacteur. Ainsi, si un lecteur est réveillé, les autres se réveillent en chaîne. Quand le dernier lecteur termine son opération, un rédacteur éventuel est réveillé.

2.4- Implémentation

- Du moment que l'accès au moniteur est exclusif, on aura donc besoin d'un sémaphore *mutex* d'exclusion mutuel (*mutex := 1*). La primitive *P(mutex)* est exécutée au début de chaque *entry* du moniteur, *V(mutex)* est exécutée à la fin.
- Pour chaque condition *C*, on associe un sémaphore privé *W*, sur lequel un processus désirant se bloquer exécute *P(W)*.
- Quand un processus exécute *signal ()* sur une condition *C*, il attend momentanément. On aura donc besoin d'un sémaphore *S* privé d'attente momentanée (*P(S)*). L'attente se fait jusqu'à ce que le processus réveillé soit quitte le moniteur soit se bloque une autre fois par *wait ()*. Donc en ce moment, ce dernier doit vérifier si un processus est en attente momentanée pour le réveiller par *V(S)*. Sachant qu'un processus réveillé par *signal ()* peut réveiller d'autres à son tour, un compteur *Scount* du nombre de processus bloqués sur *S* est aussi nécessaire.
- Sachant que *signal ()* n'a d'effet que s'il y a au moins un processus bloqué sur *W*, un compteur *Wcount* est donc impératif. Il est testé par rapport à 0 pour éviter l'effet si *Wcount=0*.

La solution est la suivante :

1/ A l'entrée au moniteur (au début de chaque *entry* du moniteur), on exécute :
P(mutex)

2/ *C.wait ()* est implémentée comme suit :

Debut

```

Wcount := Wcount + 1 ;
Si Scount > 0 Alors V(S) Sinon V(mutex) Fsi ;
P(W) ;
Wcount := Wcount + 1 ;

```

Fin;

3/ *C.signal ()* est implémentée comme suit :

Debut

```

Scount := Scount + 1 ;
Si Wcount > 0 Alors V(W) ; P(S) Fsi ;
Scount := Scount - 1 ;

```

Fin ;

4/ A la sortie du moniteur (à la fin de chaque *entry* du moniteur), on exécute :

```
Si (Scount > 0) Alors V(S) Sinon V(mutex) Fsi ;
```

Remarque :

- Les variables de synchronisation *Wcount* et *Scount* sont protégées automatiquement par *mutex*.
- Le réveil des processus en attente momentanée se fait dans l'ordre LIFO.

2.5- Variantes de moniteur

Les Variantes de moniteur sont liées essentiellement à la sémantique de la primitive *signal ()*. Dans la définition classique, la synchronisation se situe sur deux points, l'un est à l'exécution de *signal* et l'autre à l'exécution de *wait ()*. La condition de franchissement n'est pas décrite explicitement, elle n'est représentée que symboliquement et son exécution par un processus le bloque de manière inconditionnelle.

D'autre part, un signal non attendu est perdu ; de plus un processus exécutant *signal ()* est amené à faire une attente momentanée.

2.5.1- Conditions de Kessels :

Une variante du moniteur proposée par [Kessels 77], consiste en la redéfinition de la primitive *wait ()* en lui associant une condition booléenne : *wait(C)*, où *C* est une expression booléenne. *C* fait intervenir les variables de synchronisation du moniteur et des paramètres des entrées du moniteur.

Un processus se bloque sur *wait(C)* si la condition *C* n'est pas vérifiée. A la sortie d'un processus du moniteur ou à son blocage par *wait ()*, toutes les conditions figurant dans les primitives *wait ()* sont réévaluées automatiquement. Si au moins une condition devient *vrai*, un des processus bloqués concerné est activé pour poursuivre son exécution. Le problème de l'exclusion mutuel au niveau du moniteur ne se pose pas dans ce cas. La primitive *signal ()* n'a pas de sens.

Exemple : Un producteur/ Un consommateur.

Une solution est donnée comme suit :

entree procedure dem_dep () ;

Debut

```
wait(cpt < N) ;
```

Fin ;

entree procedure dem_prel () ;

Debut

```
wait(cpt > 0) ;
```

Fin ;

entree procedure fin_dep () ;

Debut

cpt := cpt + 1 ;

Fin ;

entree procedure fin_prel () ;

Debut

cpt := cpt - 1 ;

Fin ;

2.5.2- Condition avec priorité

Certains problèmes nécessitent d'introduire un ordre de réveil qui n'est pas forcément l'ordre chronologique. Pour satisfaire ce besoin, une variante de moniteur avec priorité est introduite.

Un paramètre *entier* est spécifié avec la primitive *wait*, (*c.wait(k)*). Ce paramètre permet d'indiquer une certaine priorité ; le processus avec la valeur minimale de priorité est réveillé. Les processus sont donc rangés dans l'ordre croissant de priorité. La primitive *signal ()* reste nécessaire.

Remarque : Il est possible d'inclure les conditions avec priorité dans une solution avec les moniteurs classiques sans inconvénients.

Exemple 1: Gestion d'une ressource à *n* exemplaires dont l'allocation est faite en priorité à celui qui demande le moins d'instances de la ressource.

Une solution est donnée ci-dessous. Un processus en attendre est associé de la valeur du nombre d'exemplaires demandés. Ainsi, le premier servi est celui dont la valeur est petite.

ress : condition ; dispo : entier ;
entree procedure dem (k : entier) ;
Debut
Tantque (dispo < k) faire
ress.wait(k) ;
Fait ;
dispo := dispo - k ;
ress.signal ()
Fin ;

entry procedure lib (m : entier) ;
Debut
dispo := dispo + m ;
ress.signal () ;
Fin ;
initialisation
Debut dispo := n Fin .

A la libération de ressources, on réveille un seul processus, celui-ci revérifie s'il peut être servi. Le réveil s'arrête, dans le cas échéant, dès qu'un processus ne peut pas être satisfait. Celui-ci regagne sa place dans la file d'attente (s'il n'y a pas de processus bloqués avec la même valeur). Le réveil se fait alors en cascade.

Exemple 2 : Gestion du bras d'un disque

Le disque est constitué de *max* cylindres numérotés de 0 à *max-1*. Le bras se déplace dans le sens haut (0 → *max-1*) et sert toutes les requêtes dans l'ordre de leur rencontre par la tête de lecture/ écriture jusqu'à la dernière requête puis change de sens et sert, de même, les requêtes rencontrée jusqu'à la dernière puis change de sens. Cette méthode est appelée SCAN. Les requêtes sont envoyées de manière dynamique.

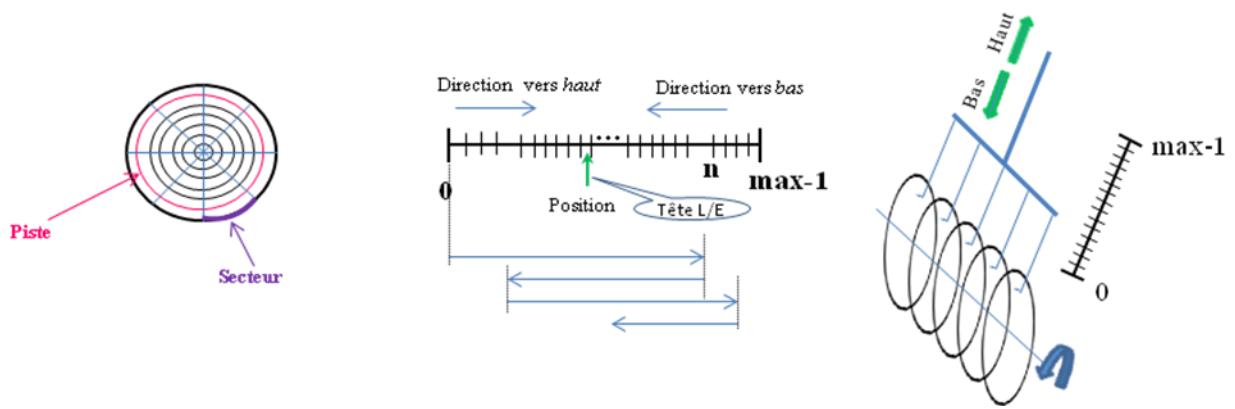


Figure 4.2 : Schéma fonctionnel d'un disque.

Pour solutionner ce problème, un certain nombre de variables sont utilisées comme décrit ci-dessous :

- Deux primitives sont nécessaires :
 - o *dem ()* est exécutée par un processus demandeur de l'E/S.
 - o *lib ()* est déclenchée par une interruption suite à la fin de l'E/S.
- Deux conditions sont utiles :
 - o *chaut*: pour les requêtes en attente de satisfaction si *direction=haut*,
 - o *cbras*: pour les requêtes en attente si *direction=bas*.
- Si la direction du bras est *haut*, chaque requête sur *n* avec *n>position* est servie dans ce sens selon l'ordre de proximité, donc l'attente s'exprime par *chaut.wait(n)*. De même, si la direction du bras est *bas*, les requêtes sur *n* avec *n<position* sont servies dans ce sens et selon l'ordre de la proximité (*n<position*), l'attente s'exprime par *cbras.wait(max-n)*.
- Les requêtes pour *n=position* (si le disque est occupé) seront satisfaites dans la prochaine direction pour éviter la famine.
- Une requête qui arrive doit attendre son tour dans le cas où le disque est occupé, d'où l'utilisation d'une variable booléenne *occupe* qui indique l'état d'occupation du disque.
- Le processus dont la requête est en cours de satisfaction doit attendre temporairement jusqu'à la fin d'E/S, d'où l'utilisation de la condition *encours*.

La solution est alors donnée comme suit :

```

Disque : Moniteur ;
Const max=.... ;
Var position : 0..(max-1) ; direction : (haut, bas) ; occupe : booleen ;
        chaut, cbras : condition avec priorité;
        encours : condition ;

entry procedure dem (n : entier) ;
    Debut
        Si occupe Alors
            Si (n>position) ou (n=position) et (direction =bas)
                Alors chaut.wait(n)
            Sinon Si (n<position) ou (n=position) et (direction= haut)
                Alors cbras.wait(max-n)
            Fsi
        Fsi
        Fsi ;
        position :=n ; occupe :=vrai ; encours.wait ()
    Fin ;

```

```

entree procedure lib () ;
  Debut
    occupe :=faux ; encours.signal ;
    Si (direction=haut) Alors
      Si non chaut.empty () Alors chaut.signal ()
        Sinon direction :=bas; cbas.signal ()
      Fsi
    Sinon
      Si non cbas.empty () Alors cbas.signal ()
        Sinon direction :=haut ; chaut.signal ()
      Fsi
    Fsi
  Fin ;
  Debut position :=0 ; direction :=haut ; occupe :=faux Fin.

```

3- Régions critiques

L'utilisation des sémaphores peut être sujette à différentes erreurs, notamment des erreurs d'ordre temporelles dues à la mal utilisation. Pour surmonter ces erreurs, un autre outil de haut niveau a été introduit par [Brinch Hansen 72] et [Hoare 72], il s'agit des régions critiques. Cet outil est moins contraignant que les moniteurs.

3.1- Régions critiques simples

Il s'agit de déclarer une variable partagée *v* de type *T* :

var v : Shared T ; où *T* est quelconque.

Cette variable *v* peut être accessible uniquement à l'intérieur d'une région dite critique contenant une instruction de la forme :

Region *v do S* ;

Cette structure implique que quand un processus est entrain d'exécuter l'instruction *S*, aucun autre n'est admis pour utiliser la variable *v*, ce qui explique l'exclusion mutuelle au niveau de l'utilisation de *v*. L'instruction *S* traite les éléments de *v* et éventuellement les variables locales à un processus.

- La variable *v* peut être utilisée dans deux régions comme dans l'exemple suivant :

Region *v do S1* ;

Region *v do S2* ;
- L'exécution concurrente est traduite par une séquentialité :

Ex :

Region *v do S1* ;

Region *v do S2* ;

- Les régions critiques peuvent être imbriquées sans inconvénients sauf qu'il faut éviter l'interblocage qui peut se produire.
- Cette construction remédie à certaines erreurs dues à la mal utilisation des sémaphores. Son implémentation est simple, il s'agit de déclarer un sémaphore *v_mutex* :=1 et encadrer l'instruction *S* avec *P(v_mutex)* et *V(v_mutex)*.

3.2- Régions critiques conditionnelles.

La structure précédente ne permet pas de résoudre des problèmes généraux de synchronisation. De ce fait, les régions critiques conditionnelles ont été introduites [Hoare 72]. Il s'agit d'ajouter dans la région une condition booléenne exprimée à l'aide des variables de la région *v* et éventuellement des variables locales au processus qui l'exécute. Selon la disposition de la condition par rapport à l'instruction de la région, on distingue différentes formes de régions critiques :

Forme 1 : **Region v when B do S ;**

Quand un processus exécute la région critique, il évalue la condition en exclusion mutuelle. Si elle est *vraie*, il exécute S et sort de la section critique. Si la condition est fausse, il se bloque hors de la section critique. Un processus bloqué est réactivé lorsqu'un autre processus sort d'une région critique associée à la variable v, il réévalue sa condition. Si la condition est *vraie* pour plusieurs processus, un seul est admis à exécuter l'instruction et les autres se bloquent une autre fois.

Exemple : **var T Shared Enregistrement**

cpt: entier :=0; pencours, cencours : boolean :=faux ;
Fin ;

Prod : **Region T when** (*cpt<n*) et non *pencours do* *pencours :=vrai* ;
< déposer (m)>

Region T do Debut *cpt :=cpt+1 ; pencours :=faux Fin ;*

Cons : **Region T when** (*cpt>0*) et non *cencours Do* *cencours :=vrai* ;
< prélever (m)>

Region T Do Begin *cpt :=cpt-1 ; cencours :=faux End ;*

Implémentation :

Une implémentation de la forme 1 est la suivante :

P(v_mutex)

Si non B alors

Début

v_cpt :=v_cpt+1 ; V(v_mutex) ; P(v_attente) ;

Tant que non B faire

v_temp :=v_temp+1 ;

Si (*v_temp<v_cpt*) **alors** *V(v_attente)*

Sinon *V(v_mutex)*

Fsi ;

P(v_attente) ;

Fait ;

v_cpt :=v_cpt-1 ;

Fin ;

S ;

Si *v-cpt>0 Alors*

v_temp :=0 ;

V(v_attente) ;

Sinon

V(v_mutex)

Fsi ;

- Tous les processus en attente réévaluent la condition à la sortie de la région critique ou lors de l'attente d'un autre processus.

- Le réveil des processus pour lesquels la condition est vérifiée n'est pas tout à fait Fifo, car si un processus se bloque une autre fois, il est mis enfin de file.

Forme 2 : **Region v do Begin S1 ; await(B) ; S2 End ;**

Quand un processus entre la région, il exécute l'instruction *S1* et évalue *B*. Si *B* est *vraie*, *S2* est exécutée en sortant après de la région critique ; dans le cas contraire, le processus libère la

section critique et attend sur la condition B jusqu'à ce que B soit *vraie* et aucun autre processus ne soit dans la région associée à v . Un processus bloqué sur B est réveillé pour réévaluer sa condition, si un autre se bloque sur sa propre condition ou après l'exécution de $S2$.

Exemple 1 : Lecteur/ Rédacteur avec priorité aux lecteurs.

Une solution est donnée comme suit :

Var v : Shared Record

```
nla : entier :=0 ; nlc :entier :=0 ;
ecriture : booleen :=faux;
```

Fin ;

Processus Lecteur () ;

Processus Redacteur () ;

Region v do Begin

```
nla :=nla+1 ;
await(ecriture=faux);
nla :=nla-1 ;
nlc :=nlc+1 ;
```

End ;

< Lecture>

Region v do $nlc :=nlc-1$;

Region v do Begin

```
await((nlc=0) et (nla=0)et (ecriture=faux ))
ecriture :=vrai ;
```

End ;

< Ecriture>

Region v do $ecriture :=faux$;

Exemple 2 : Lecteur/ Rédacteur avec priorité aux rédacteurs.

Une solution est donnée comme suit :

Var v : shared Record

```
nr : entier :=0 ; nlc :entier :=0 ;
ecriture : booleen :=faux ;
```

Fin ;

Processus Lecteur () ;

Processus Redacteur () ;

Region v do Begin

```
await(nr=0) ;
nlc :=nlc+1 ;
```

End ;

< Lecture>

Region v do $nlc :=nlc-1$;

Region v do Begin

```
nr :=nr+1 ;
await(non écriture) et (nlc=0)
écriture :=vrai ;
```

End ;

< Ecriture>

Region v do Begin

```
nr :=nr-1 ;
écriture :=faux ;
```

End ;

3.3. Implémentations

- Considérons l'implémentation de la forme 1 à l'aide des sémaphores. L'exclusion mutuelle est assurée par un sémaphore v_mutex associé à chaque région v . L'attente dans le cas où la condition n'est pas vérifiée est exprimée à l'aide du sémaphore $v_attente$. Afin de permettre à tous les processus en attente de réévaluer la condition B , un compteur v_temp est utilisé. Autrement dit, il compte le nombre de processus qui ont déjà testés leurs conditions sans succès. Il est donc comparé au nombre total de processus bloqués (exprimé à l'aide v_cpt) pour déterminer s'il faut réveiller un autre.

Une solution est donnée comme suit :

```

Var    sémaphore v_mutex := 1 ; v_attente := 0 ;
        Entier v_cpt := 0, v_temp := 0 ;

Début
P(v-mutex)
S1 ;
Si non B Alors
v-cpt := v-cpt + 1 ;
Si (v-cpt>1) Alors v-temp := 1 ; V(v-attente) Sinon V(v-mutex) Fsi ;
P(v-attente) ;
Tant que non B Faire
v-temp := v-temp + 1 ;
Si (v-temp<v-cpt) Alors V(v-attente) Sinon V(v-mutex) Fsi ;
P(v-attente)
Fait ;
v-cpt := v-cpt - 1
Fsi ;
S2 ;
Si (v-cpt>0) Alors v-temp := 0 ; V(v-attente) Sinon V(v-mutex) Fsi ;
Fin.

```