1. Définition

La synchronisation consiste ***à définir*** et ***à réaliser*** des mécanismes assurant le respect des contraintes liées à la progression des processus, c'est à dire capable de faire attendre un processus qui ferait des actions ne respectant pas ces contraintes.

2. Expression des contraintes de synchronisation

Les contraintes logiques imposées par la coopération de processus peuvent être formulées sous deux formes:

a. On impose un ordre de précédence dans le temps logique sur l'exécution des actions (instructions) des processus.

- **Exemple**

Considérons n processus P1, P2,... , Pn. On définit dans le programme de chaque processus un point dit de «***rendez-vous*** », que le processus ne peut franchir que si tous les autres processus aient atteint leur point de « Rendez-vous ». Le programme d'un processus peut être de la forme :

$Début_i$
« $Rendez\text{-}vous_i$ »
$Suite_i$
La contrainte de synchronisation peut s'exprimer par :
$\forall\ i, j \in [1..n]\ \ fin(Rendez\text{-}vous_i) < début(suite_j)$

b. On impose à certains processus une condition pour le franchissement de certains points de leur trace temporelle. Ces points sont appelés points de synchronisation.

- **Exemple**

Allocateur de ressources. Un processus, qui fait une demande de ressource, ne peut continuer son exécution que lorsque sa requête est satisfaite.

3. Spécification de la synchronisation

La spécification d'un problème de synchronisation consiste à:

1. Définir pour chaque processus ses points de synchronisation.
2. Associer à chaque point de synchronisation une condition de franchissement, exprimée au moyen des variables d'état du système.

Exemples de variables d'état du système

- Variables exprimant le nombre d'unités disponibles pour chaque ressource.
- Variables exprimant le nombre de demandes d'exécution d'une même procédure depuis un temps donné.

- Variables exprimant le nombre d'actions terminées pour un processus depuis un temps donné.
- Variables exprimant le nombre de processus en attente d'une certaine ressource.
- Variables exprimant l'état des ressources : libre, occupée.
- Variables exprimant l'état des processus : actif, prêt, bloqué,…

Exemple : allocation d'une imprimante

Considérons un système informatique ayant 3 imprimantes « banalisées ».

- **Points de synchronisation**

Allouer(une imprimante)
« Imprimer »
Libérer(imprimante)

Deux points de synchronisation *allouer et libérer*. Libérer est point de synchronisation non bloquant : la condition, au niveau de ce point, est toujours vraie.

- **Les contraintes de synchronisation pour l'allocation d'une imprimante.**

nbre_imp_libre = 3; // variable d'état qui exprime le nombre d'imprimantes (ressources) disponibles (libres)//
Allocation d'une imprimante si nbre_imp_libre > 0
Allouer(*une imprimante****)***
Si *nbre_imp_libre=0* ***alors*** *attendre;*
nbre_imp_libre= nbre_imp_libre-1;
fin;

4. Les problèmes types

Les problèmes de synchronisation rencontrés dans la pratique peuvent, dans leur grande majorité, se ramener à des combinaisons d'un petit nombre de situations élémentaires pour lesquelles des schémas de solutions sont connus. Les problèmes types considérés sont les suivants:

1. **Partage d'une ressource commune par plusieurs processus:**
   - Allocateur de ressources banalisées
   - Modèle de lecteurs-rédacteurs.
2. **Communication entre processus:**
   - Modèle des producteurs-consommateurs
   - Rendez-vous

5. Les techniques de synchronisation

Un mécanisme de synchronisation (ou technique de synchronisation) doit être construit indépendamment des vitesses d'exécution des processus et permet à un processus actif :

- De bloquer un autre ou de se bloquer lui-même en attente d'un événement ou d'un signal d'un autre processus.
- D'activer un autre processus en lui transmettant éventuellement de l'information.

Le signal d'activation envoyé par un processus actif peut être mémorisé ou non. Un mécanisme de synchronisation peut offrir deux types d'opérations ou de primitives :

- Le processus actif agit, au moyen de primitives (opérations), sur un autre processus en le désignant par son identificateur (ou par son nom). Dans ce cas on dit que la ***synchronisation est directe***.

- Le processus actionne un mécanisme qui agit sur d'autres processus c'est à dire que le processus ne connaît pas le ou les processus sur lesquels il agit : on dit que la ***synchronisation est indirecte***.

Malgré ses insuffisances, le mécanisme des **sémaphores** permet de programmer la plupart des solutions de synchronisation dans un système centralisé. Par la suite nous verrons d'autres techniques de haut niveau telles que les **régions critiques** et les **moniteurs**.

6. Exemples

6.1. Allocation d'une imprimante

Un système dispose de **n** imprimantes utilisées par plusieurs processus. Le schéma général d'un processus est alors :

Demander imprimante;
Utiliser l'imprimante ;
Libérer l'imprimante;

- **Les points de synchronisation**
    a. Demander imprimante
    b. Libérer l'imprimante

- **Les conditions de franchissement de ces points de synchronisation**
    a. Allocation d'une imprimante *si nbre_imp_libre > 0*
    Ou
    b. Allocation d'une imprimante si nbre_imprimantes_allouées < n (**n** : nombre d'imprimantes)

- **Le programme d'un processus en utilisant les sémaphores**

implibre entier init(n); // variable d'état représentant le nombre d'imprimantes libres//

Demander(imprimander)	*Libérer(imprimante)*
Début *si implibre = 0 alors attendre ;* *implibre = implibre –1;* *fin*	*Début* *Implibre = implibre +1;* *fin*

- **Solution avec les sémaphores**

simplibre semaphore init(n);

Demander(imprimander)	*Libérer(imprimante)*
Début *P(simplibre);* *Fin*	*Début* *V(simplibre);* *Fin*

6.2. Le modèle des lecteurs/rédacteurs (Courtois et al, 1971)

Considérons deux classes de processus appelés *Lecteurs* et *Rédacteurs*. Ces processus se partagent un fichier. Les lecteurs peuvent seulement consulter le fichier et les rédacteurs peuvent seulement écrire sur le fichier. Les processus de ces deux classes doivent respecter les contraintes suivantes:

a. Plusieurs lecteurs peuvent lire simultanément le fichier.
b. Un seul rédacteur à la fois peut écrire sur le fichier.
c. Un lecteur et un rédacteur ne peuvent pas utiliser en même temps le fichier.

- **Les points de synchronisation**

Les processus utilisent le fichier selon le protocole suivant:

Lecteurs	*Rédacteurs*
Demande de lecture *Lecture* *Fin de lecture (libérer le fichier)*	*Demande d'écriture* *Ecriture* *Fin d'écriture (libérer le fichier)*

- **Les conditions de franchissement de ces points de synchronisation**

nl : nombre de lectures en cours,
ne : nombre d'écritures en cours(0 ou 1).

Les deux points de synchronisation bloquants sont demande de lecture et demande d'écriture.

a. Condition de franchissement du point demande de lecture : ne=0;
b. Condition de franchissement du point demande d'écriture : ne=0 et nl=0;

- **Les programmes Lecteurs et Rédacteurs en utilisant les sémaphores**

On suppose que les processus (lecteurs et rédacteurs) ont la même priorité et que l'ordre d'accès au fichier est quelconque (on peut remarquer qu'il y a risque de privation pour les rédacteurs).
On utilise une variable nl qui compte le nombre de lecture en cours (nombre de processus lecteurs qui lisent le fichier). Cette variable n'est utilisée que par les lecteurs. La variable nl doit être utilisée en exclusion mutuelle; soit mutex un sémaphore, initialisé à un, qui protégera cette variable.

On définit un autre sémaphore pour protéger le fichier; ce sémaphore est utilisé par les lecteurs et les rédacteurs ; f : permet l'accès au fichier; f initialisé à 1;

Voici le programme d'un lecteur et d'un rédacteur :

Sémaphore f init(1);

Lecteurs	*Rédacteurs*
Entier nl init(0); *Sémaphore mutex init (1);* *Début* */* demande de lecture */* *p(mutex);* *nl=nl+1;* *si nl=1 alors p(f);* *v(mutex);* *< lire >* */* fin de lecture */* *p(mutex);* *nl=nl-1;* *si nl=0 alors v(f);* *v(mutex);* *fin;*	*Début* */* demande d'écriture */* *p(f);* *<écrire>* */* fin d'écriture */* *v(f);* *fin;*

6.3. Le rendez-vous

6.3.1. Rendez-vous de 2 processus

Soient deux processus p1 et p2; chacun des deux processus doit attendre l'autre processus pour continuer son exécution.

P1	*P2*
Début *Signaler son arrivée à P2;* *Inst1;* *Inst2;* *…* *$Inst_i$;* *Attendre p2;* *$Inst_j$* *…* *Fin*	*Début* *Signaler son arrivée à P1;* *Inst1;* *Inst2;* *…* *$Inst_m$;* *Attendre p1;* *$Inst_n$;* *…* *fin*

- **Solution avec les sémaphores:**

Sémaphore s1,s2 init (0);

P1	*P2*
Début *…* *v(s2);*	*Début* *…* *v(s1);*

p(s1); *…* *fin;*	*p(s2);* *…* *fin;*

- **Exemple de rendez-vous : l'examen**

Un examen ne peut commencer que si l'enseignant responsable de l'examen(R), au moins un surveillant(S) et au moins un étudiant (E) sont présents.
Soit m le nombre de surveillants et n le nombre d'étudiants. La solution suivante utilise 3 sémaphores :

Sémaphore *sresponsable, ssurveillant, sétudiant init (0);*

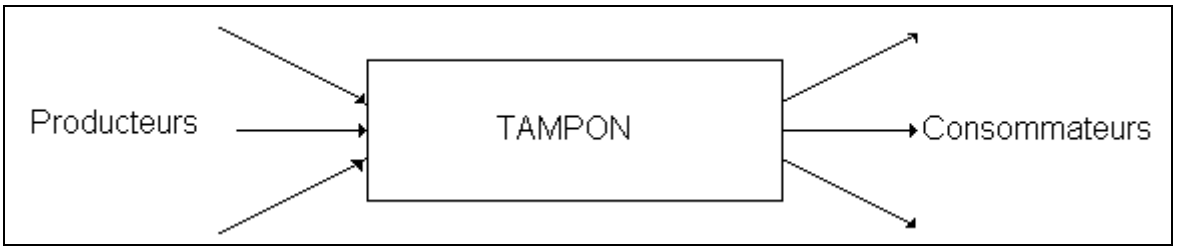
Responsable R	*Surveillant S_i*	*Etudiant E_j*
Début *p(ssurveillant)* *p(sétudiant)* ***Pour*** *i=1 à n+m* *V(sresponsable)* *Finfaire* *Fin*	*Début* *V(ssurveillant)* *p(sresponsable)* *Fin*	*Début* *v(setudiant)* *p(sresponsable)* *Fin*

6.4. Communication par variables communes : le modèle producteur - consommateur

Le Principe de la communication par variable commune est le suivant :
Un tampon, susceptible de contenir la valeur ou la référence (adresse) de l'information à communiquer, est accessible à l'ensemble des processus voulant communiquer.

6.5. Définition

On appelle ***Producteur***, un processus désirant *déposer* de l'information dans un *tampon T* et ***Consommateur*** un processus désirant *retirer* de l'information d'un *tampon T*.



6.5.1. Propriétés de la communication

- La communication est indirecte, asynchrone et multiple.
- La communication est *indirecte* car le producteur et le consommateur ne s'adressent pas l'un à l'autre.
- La communication est *asynchrone* car il n'est pas nécessaire que le producteur et le consommateur entrent en communication en même temps (simultanément). Il suffit, simplement, que le producteur ait déposé l'information dans le tampon avant que le consommateur ne la retire.

- La communication est ***multiple*** car plusieurs producteurs peuvent envoyer(déposer) des messages à plusieurs consommateurs.

Ce mode de communication pose deux types de problèmes :

a) La synchronisation des processus

Cela concerne, notamment les droits d'accès au tampon :

- Un producteur ne peut déposer un message que s'il y a de la place disponible.
- Un consommateur ne peut retirer un message que s'il y en a de disponible.
- Les producteurs et les consommateurs ne peuvent accéder simultanément au tampon que selon certaines règles assurant l'intégrité de l'information.

b) La transmission,

C'est le moyen par lequel l'information est communiquée, concerne la gestion du tampon et la manière d'accéder à l'information une fois le droit d'accès accordé.

6.5.2. Schéma général du producteur-consommateur

Cinq notions apparaissent dans ce schéma :

- *Le message* : l'information à transmettre.
- *L'élément* : le support de mémorisation du message.
- *Le tampon*: ensemble d'éléments.
- *Le mécanisme de synchronisation*: sémaphores, événement, moniteurs, …
- *Le Système de communication (communicateur)* : il est composé du tampon, du mécanisme de synchronisation et des primitives de communication envoyer(message) et recevoir(message).

Soient un producteur désirant envoyer un message M1 dans un système de communication S et un consommateur désirant retirer un message M2 de ce même système de communication :

Producteur	*Consommateur*
Envoyer(M1);	*Recevoir(M2);*

En séparant le problème de la synchronisation et de celui de la transmission, ces primitives se décomposent ainsi:

Producteur	*Consommateur*
Procédure envoyer(M1) *DD; /* demande de dépôt */* *Dépôt(M1);* *SD; /* signal de dépôt */* *Fin;*	*Procédure recevoir(M2)* *DR; /* demande de retrait */* *Retirer(M2);* *SR; /* signal de retrait */* *Fin;*

6.5.3. Solutions avec les sémaphores

On peut distinguer trois types de tampons qui conduisent à des mécanismes différents:

- Tampon à un seul élément (boite aux lettres),

- Tampons à n éléments,
- Tampon à éléments alloués.

a) Producteur-Consommateur avec un tampon à un seul élément

Un "Tampon à un seul élément" ne peut contenir qu'une seule information. Autrement dit, un producteur ne peut y déposer une information que si le consommateur a retiré la précédente. Un "Tampon à un seul élément" possède donc deux états : plein et vide.

Réalisation des procédures " DD, SR, DR et SR " au moyen des sémaphores

On associe deux sémaphores splein et svide au tampon T initialisés respectivement à **zéro** et **un**.

Tampon T;
Splein: sémaphore init (0); Svide : sémaphore init (1);

*Procédure DD; /*demande de dépôt */* *p(svide);* *Fin;* *Procédure SD; /* signal de dépôt */* *v(splein);* *Fin;*	*Procédure DR; /*demande de retrait */* *P(splein);* *Fin;* *Procédure SR; /* signal de retrait*/* *v(svide);* *Fin;*

b) Producteur-Consommateur avec un tampon à n éléments

Dans ce cas les processus producteurs et consommateurs se partagent un Tampon composé de n éléments : c'est une généralisation du cas précédent.

Réalisation des procédures " DD, SR, DR et SR " au moyen des sémaphores

La réalisation est identique au cas précédent sauf que le sémaphore svide est initialisé à **n**. On associe deux sémaphores splein et svide au tampon T initialisés respectivement à **zéro** et **n**.

Tampon T;
splein: sémaphore init (0); svide : sémaphore init (n);

*Procédure DD; /*demande de dépôt */* *p(svide);* *Fin;* *Procédure SD; /* signal de dépôt */* *v(splein);* *Fin;*	*Procédure DR; /*demande de retrait */* *p(splein);* *Fin;* *Procédure SR; /* signal de retrait */* *v(svide);* *Fin;*

c) Producteur-Consommateur avec un tampon à éléments alloués

Dans ce cas, une partie du problème de synchronisation se trouve reporter au niveau de "l'allocateur de mémoire". Celui-ci a pour rôle d'allouer aux producteurs qui lui demandent un élément de tampon de la même manière, les consommateurs doivent signaler la libération des éléments de tampon à l'allocateur.

Réalisation des procédures " DD, SR, DR et SR " au moyen des sémaphores

Considérons deux primitives ***allouer***(élément) et ***libérer***(élément) permettant respectivement l'allocation et la libération d'un élément du tampon.
On utilise un seul sémaphore splein initialisé à zéro. Ce sémaphore permet d'une part, au producteur de signaler le dépôt de messages et d'autre part, il permet au consommateur de vérifier la présence de message dans le tampon.

Tampon T;
splein: sémaphore init (0);

*Procédure DD; /*demande de dépôt */* *Allouer(élément);* *Fin;* *Procédure SD; /* signal de dépôt */* *v(splein);* *Fin;*	*Procédure DR; /*demande de retrait */* *P(splein);* *Fin;* *Procédure SR; /* signal de dépôt */* *Libérer(élément);* *Fin;*

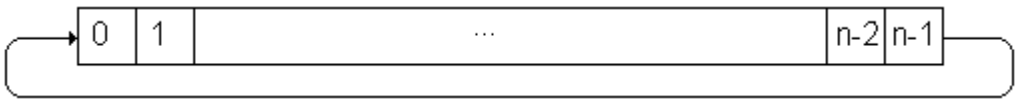
6.5.4. Gestion du tampon (Transmission de message)

a) Tampon à un seul élément

Déposer(Message) *Tampon := Message;* *Fin;*	*Retirer(Message)* *Message := Tampon;* *Fin;*

b) Tampon à un nombre d'éléments fixe

Le tampon sera géré de manière circulaire: on l'appelle Tampon circulaire ou buffer circulaire. Les éléments du tampon sont numérotés de 0 à n-1. Dans un tampon circulaire, on considère l'élément 0 comme le successeur de l'élément n-1.



1. Un producteur et un consommateur

Pour parcourir le tampon, chaque classe de processus (producteurs ou consommateurs) aura son propre pointeur:
In : pointeur (ou index) utilisé par le producteur,
Out : pointeur (ou index) utilisé par le consommateur.
In et out sont deux variables locales respectivement au producteur et consommateur et elles sont initialisées à 0(zéro).

Déposer(Message) *Tampon[in]:= Message;* *In := in +1 modulo n* *Fin;*	*Retirer(Message)* *Message := Tampon[out];* *Out:=out +1 modulo n* *Fin;*

2. Plusieurs producteurs(P) et plusieurs consommateurs(C)

Dans le cas de plusieurs producteurs et plusieurs consommateurs, il y a un problème d'exclusion mutuelle pour l'accès au tampon:

Exclusion mutuelle entre producteurs

Plusieurs producteurs ne doivent pas déposer en même temps de l'information dans un même élément du tampon.

Exclusion mutuelle entre Consommateurs

Plusieurs consommateurs ne doivent pas retirer (consommer) un même message(un message n'est consommé qu'une seule fois par un seul processus).
Pour parcourir le tampon, chaque classe de processus (producteurs ou consommateurs) aura son propre pointeur:
In : pointeur (ou index) utilisé par les producteurs, c'est une variable commune aux producteurs : ils doivent l'utiliser en exclusion mutuelle.
Out : pointeur (ou index) utilisé par les consommateurs. C'est une variable commune aux consommateurs : ils doivent l'utiliser en exclusion mutuelle.
Les deux variables sont initialisées à 0(zéro).
On aura besoin de deux sémaphores d'exclusion mutuelle sin et sout pour protéger les variables communes in et out.
Sin et sout sont initialisés à 1.

Déposer(Message) *p(sin);* *Tampon[in]:= Message;* *in := in +1 modulo n;* *v(sin)* *Fin;*	*Retirer(Message)* *p(sout);* *Message := Tampon[out];* *out:=out +1 modulo n;* *v(sout)* *Fin;*

6.6. Problème du dîner des philosophes

Considérons cinq philosophes qui passent leur temps à penser et à manger. Les philosophes partagent une table entourée de cinq chaises. Au milieu de la table se trouve un bol de riz et cinq baguettes. De temps en temps, un philosophe a faim et essaye de prendre les deux baguettes qui sont le plus près de lui (entre lui et ses voisins de gauche et de droite). Un philosophe peut prendre seulement une baguette à la fois. Il ne peut pas prendre une baguette qui est déjà dans la main d'un voisin. Quand un philosophe affamé possède les deux baguettes à la fois, il mange sans libérer ses baguettes. Quand il a fini de manger, il laisse ses deux baguettes et recommence à penser.

Nous pouvons représenter chaque baguette par un sémaphore. Un philosophe essaie de prendre une baguette en exécutant une opération *wait* sur ce sémaphore. Il la libère en exécutant une opération *signal* sur le sémaphore correspondant.
var *chopstick :* ***array****[0..4]* ***of*** *semaphore ; -- tous les éléments sont initialisé à 1 ---*

Code d'un philosophe i
repeat
wait(chopstick[i]) ;
wait(chopstick[i+1 ***mod*** *5]) ;*
…
manger

```
            …
    signal(chopstick[i]) ;
    signal(chopstick[i+1 mod 5]) ;
    …
        penser
        …
until false ;
```

Quelques remarques s'imposent :

- Cette solution peut conduire à **l'interblocage** : tous les philosophes prennent en même temps toutes les baguettes (de gauche, par exemple).
- La situation de **famine** est possible.
- Cette solution garantit que deux philosophes voisins ne mangent pas en même temps.

Quelques astuces permettent d'éviter l'interblocage : permettre à un philosophe de prendre ses baguettes seulement si les deux sont disponibles (les prendre dans une section critique), utiliser une solution asymétrique (un philosophe impair prend d'abord la baguette de gauche ensuite de droite, un philosophe pair effectue l'inverse).

7. Mécanismes de synchronisation de haut niveau

7.1. Introduction

Les sémaphores sont des outils adaptés aux mises en œuvre où les processus exécutent eux-mêmes les opérations de contrôle puisqu'ils permettent de limiter le nombre d'exécutions simultanées (Exemple: sémaphore d'EM limite à 1 ce nombre). On peut les utiliser aussi pour programmer la communication avec un processus contrôleur (Exemple: accès contrôlé à l'aide d'une boîte aux lettres). Cependant, ils présentent plusieurs inconvénients:

- La lisibilité des programmes : les programmes sont peu lisibles car le nombre d'opérations sur sémaphores est en général important. De plus ces opérations sont éparpillées dans le texte des programmes constituant les processus (Complexité de la maintenance des programmes).
- Difficulté de trouver des solutions correctes à moins de programmer systématiquement à partir de l'expression abstraite ce qui donne des solutions peu efficace du point de vue performance (Exemple: modèle Producteur/Consommateur).

D'autres insuffisances concernent les opérations P et V elles-mêmes :

- La primitive v ne permet d'activer (réveiller) qu'un seul processus de la file d'attente du sémaphore.
- Le processus activé ne connaît pas le processus qui l'a réveillé : ce mécanisme permet une action indirecte sur les processus.

- La primitive v mémorise l'événement : la variable du sémaphore est incrémentée même si le si aucun processus n'attend cet événement.

7.2. Sections ou régions critiques

7.2.1. Définition

Lorsqu'on programme avec des sémaphores, rien n'interdit d'utiliser une variable de synchronisation en dehors d'une séquence d'exclusion mutuelle (EM). L'utilisation correcte des ressources partagées ne repose que sur une bonne programmation qui introduit des opérations adéquates sur sémaphores pour contrôler les accès à ces variables. Le concept de **région critique** qui a été introduit par: **HOARE** (72) et **BRINCH HANSEN** (72) remédie à cet inconvénient.

Une **région critique** définit une suite d'instructions exécutées en EM et relative à l'utilisation d'une variable déclarée explicitement comme partagée. En prenant pour exemple de langage de programmation parallèle le Concurrent PASCAL (extension du PASCAL traditionnel intégrant la gestion des processus) une variable partagée sera déclarée par l'attribut SHARED:

Var V : Shared t;
V : nom de la variable; t: Type de la variable.

Tout accès à la variable V ainsi déclarée sera contrôlé à la compilation. Une instruction spéciale permet l'accès à la variable V à l'aide d'un mot clé Region ou With.

Il existe plusieurs types de régions critiques:

7.2.2. Régions Critiques inconditionnelles

Syntaxe: *Region V Do Begin*
S;
End;

S: suite d'instructions (peut être vide).

Cette notation exprime que la suite d'instructions S doit accéder à la variable partagée V sans condition et ne peut s'exécuter qu'en EM relativement à V (c'est à dire que l'accès à V sera exclusif). En conséquence, lorsque plusieurs processus tentent d'exécuter simultanément des régions associées à une même variable partagée, un seul est autorisé à poursuivre son exécution. Les autres resteront bloqués jusqu'à la sortie du processus courant de la Région critique (après la dernière instruction de la séquence S) ; à ce moment là, un seul processus parmi ceux en attente sera autorisé à entrer en Région critique.

Le compilateur qui va traduire cette instruction gère lui-même les sémaphores nécessaires et placera les P et V nécessaires là où il faut.

Remarque:
Il y a des cas d'interblocage que le compilateur ne pourra jamais détecter et qui correspondent par exemple au cas d'imbrication croisée (ordre différent) de régions critiques correspondantes à des variables partagées distinctes.

- **Exemple**

Processus P1;
Region A Do Begin
Region B Do S1;
End;

Processus P2;
Region B Do Begin
Region A Do S2;
End;

- **Traduction par sémaphores:**

a) Première version:

Le principe est d'associer à chaque variable partagée X un sémaphore d'EM : X_Mutex init 1 qui garantit l'accès exclusif à X. D'où :

Region X Do Begin
S;
End;

se traduira par: P(X_Mutex);
S;
V(X_Mutex);

b) Deuxième version:

La version précédente s'avère insuffisante lorsque la variable X est associée à des conditions d'exécution de S (voir plus loin Régions critiques conditionnelles). Dans ce cas, on verra que la séquence de sortie n'est pas composée d'une seule instruction V(X_Mutex), mais d'une séquence de sortie plus complète:

```
Begin
 If X_Count > 0 then
 Begin
   X_Temp:= 0;
    V(X_Wait);
  End;
 Else V(X_Mutex);
End Sortie;
```

Ceci s'obtient en considérant une condition associée à S toujours vraie.

7.2.3. Régions critiques conditionnelles

Ce type de région, contrairement au précédent, permet de définir, en plus de l'accès exclusif à V, une condition sur l'exécution de S. On peut distinguer deux formes de régions critiques conditionnelles:

Première forme: *Region V When b Do S;*

b: condition (expression booléenne) pouvant contenir des constantes et (ou) la variable V.

Fonctionnement: La suite d'instructions S n'est exécutée que si l'expression booléenne b est vraie.

L'exécution par un processus P est la suivante: P entre en région critique (première porte) et évalue l'expression b(deuxième porte), deux cas sont alors possibles:

1) Soit b est vraie alors P exécute S et sort de la région critique.

2) Soit b est fausse alors P libère la région critique (première porte) et se bloque sur la condition b attendue (deuxième porte). P sera réactivé à chaque fois qu'un processus sort (a terminé complètement) d'une région critique associée à V pour réévaluer b à nouveau.

Pour chaque variable partagée V on aura ainsi 2 classes de processus:
- processus demandant l'entrée en région critique associée (bloqués sur la première porte)
- processus ayant passé la première porte mais ayant trouvé une condition attendue bi fausse (bloqués sur la deuxième porte).

Deuxième forme:

```
Region V  Do Begin
        S1;
        Await(b);
        S2;
        End;
```

Dans cette forme, après entrée en région critique, la séquence S1 est exécutée sans condition, ensuite b est évaluée:
1) Soit b est vraie alors P exécute S2 et sort de la région critique.
2) Soit b est fausse alors P libère la région critique (première porte) et se bloque sur la condition b attendue (deuxième porte). P sera réactivé à chaque fois qu'un processus sort (a terminé complètement) d'une région critique associée à V pour réévaluer b à nouveau.

- **Traduction de ces deux formes par sémaphores:**

Principe: Traduire par sémaphores les deux formes de régions critiques conditionnelles en établissant à chaque sortie d'un processus, la priorité aux processus bloqués sur leur condition bi (deuxième porte).

Sémaphores nécessaires:
Première porte: V_Mutex init 1 permet l'entrée en Region V en assurant l'EM pour l'accès à V.
Deuxième porte: V_Wait init 0 permet de bloquer les processus ayant trouvé leur condition bi fausse.

Compteurs:
V_Count init 0 permet de compter le nombre de processus bloqués dans la file de V_Wait.
V_Temp init 0 permet de compter le nombre de processus qui se sont rebloqués dans la file de V_Wait après chaque réévaluation de leur condition bi

Première forme: Region V When b Do S; se traduit en 3 parties successives:
1_Entrée en Région;
2_S;
3_Sortie de la Région;

```
1_Entrée en Région;
  Begin
    P(V_Mutex)
    If Not(b) then
    Begin
```

```
        V_Count:=V_Count + 1;
        V(V_Mutex);
        P(V_Wait);
        While Not(b) Do
         Begin
           V_Temp:=V_Temp + 1;
            If V_Temp < V_Count then  V(V_Wait);
            else V(V_Mutex);
            P(V_Wait);
          End;
        V_Count:=V_Count - 1;
      End;
End Entrée;

3_Sortie de la Region;
Begin
  If V_Count > 0 then
  Begin
     V_Temp:= 0;
     V(V_Wait);
   End;
  Else V(V_Mutex);
End Sortie;
```

7.2.4. 2.3 Applications:

Reprendre des problèmes classiques d'EM, Synchronisation et Communication déjà traités par sémaphores et les programmer à l'aide des régions critiques. Etablir une comparaison dans la facilité de programmation et l'efficacité à l'exécution.
Problèmes d'EM : Lecteurs/Rédacteurs, Feux de circulation, Pont à voie unique;
Problèmes de Synchronisation/Communication : Producteur/Consommateur, Allocateur de ressources banalisées.

Exemple : Modèle P/C

```
Var V : shared record
                Buf : array[0..N-1] of item;
                Nplein,In,Out : integer;
                end;
Initialisation: Begin
                V.Nplein , V.In , V.Out :=0;
               End;
Procedure P;
Var Nextp: item;
Begin
  Repeat
    Produce(Nextp);
    Region V when (V.Nplein < N) do
    Begin
```

```
      V.Buf[V.In]:= Nextp;
      V.In:=V.In + 1 Mod N;
      V.Nplein:=V.Nplein+1;
    End;
  Until false;
End P;
Procedure C;
Var Nextc: item;
Begin
  Repeat
    Region V when (V.Nplein > 0) do
    Begin
      Nextc:=V.Buf[V.Out];
      V.Out:=V.Out + 1 Mod N;
      V.Nplein:=V.Nplein-1;
    End;
  Until false;
End C;
```

7.2.5. Evaluation des régions critiques:

1) Avantages:

- Les régions critiques permettent à chaque processus coopérant de spécifier le traitement particulier qu'il veut effectuer sur les variables partagées. On a donc une clarté dans la spécification de la synchronisation.
- L'EM est assurée automatiquement:
    - Les variables partagées ne peuvent pas être utilisées en dehors des régions critiques : protection assurée par compilation
    - Une région critique joue le rôle d'un contrôleur assurant l'EM entre les processus (contrôle statique).

3) Certaines techniques de preuves de programmes parallèles utilisent les régions critiques (travaux de **GREES**).Ces techniques peuvent démontrer par exemple qu'un programme ne peut pas se bloquer indéfiniment.

2) Inconvénients:

- L'inconvénient déjà signalé pour les sémaphores persiste: lisibilité et dispersion des Régions dans les textes des programmes.
- Réévaluation trop fréquentes et inutiles des conditions à l'intérieur d'une région. Inutiles car les processus peuvent être réactivés sans que leur condition ait changé.

Certains compilateurs tentent de réduire l'inconvénient 2) en utilisant des réveils explicites sur condition: mais ceci ne peut se faire que si des relations statiques(visibles à la compilation) existent entre les différentes conditions bi associées à une même variable partagée V.

7.3. Moniteurs

La notion de Moniteur a été suggérée par **DIJKSTRA** (notion de secrétaire) et améliorée par **BRINCH HANSEN** [73] et par **HOARE** [74].

7.3.1. Définition

Un moniteur est une entité syntaxique (module) qui regroupe :
- Les variables de synchronisation
- Les ressources partagées
- Les procédures qui les manipulent

Par définition, les variables déclarées dans le moniteur ne sont accessibles que par les procédures du moniteur. Parmi ces procédures certaines sont :
- internes = définies pour les besoins internes du moniteur et non accessibles de l'extérieur.
- externes = appelables de l'extérieur du moniteur, on les nomme services ou points d'entrée du moniteur. Une procédure externe est identifiée par l'attribut ENTRY lors de sa déclaration dans un moniteur:

Procedure Nom Entry(paramètres éventuels);

L'absence de cet attribut définit une procédure interne.

7.3.2. Syntaxe d'un moniteur

Type *nom_du_moniteur* **= *monitor***
Déclarations de variables

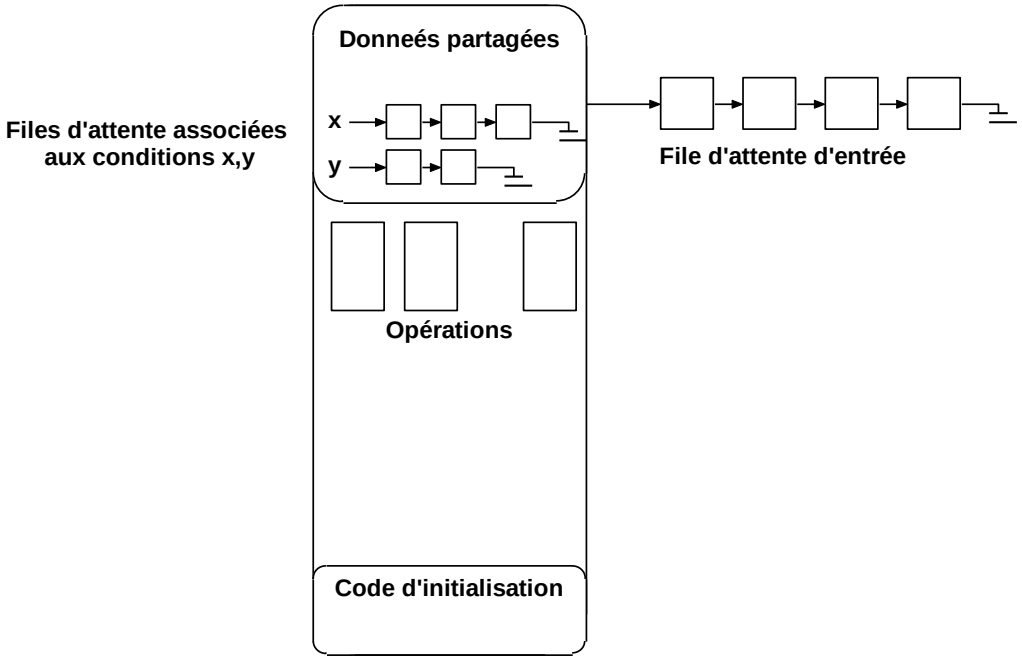
Procedure entry P_1*(..) ;*
begin ..end *;*

Procedure entry P_2*(..) ;*
begin ..end *;*

Procedure entry P_n*(..) ;*
begin ..end *;*

begin
code d'initialisation
end.

Un moniteur peut être schématisé comme suit :



Deux types de contrôle de l'exécution des processus sont définis dans un moniteur:

1) **EM** : tout moniteur réalise l'exclusion mutuelle, de manière implicite, entre les processus qui appellent ses services. Un seul processus à la fois peut exécuter les procédures du moniteur jusqu'à ce qu'il sorte du moniteur (fin de service) ou jusqu'à son blocage intrinsèque.

2) **Synchronisation** : Les variables de synchronisation sont définies explicitement par l'attribut CONDITION. A chaque variable de ce type est associée une file d'attente de processus. Sur ces variables on peut appliquer 3 opérateurs indivisibles:

Syntaxe générale:
Opérateur(Variable de synchronisation) ou Variable de synchronisation.Opérateur

Premier opérateur: Wait
Il permet d'exprimer des attentes sur condition en bloquant systématiquement le processus qui l'exécute dans la file d'attente associée à la variable de synchronisation concernée.

Deuxième opérateur: Signal
Il permet de réveiller un processus bloqué dans la file d'attente de la variable de synchronisation concernée. Si cette file d'attente est vide, le signal n'est pas mémorisé et son action sera sans effet.

Troisième opérateur: Empty
C'est un prédicat indiquant si la file d'attente de la variable de synchronisation est vide ou non.

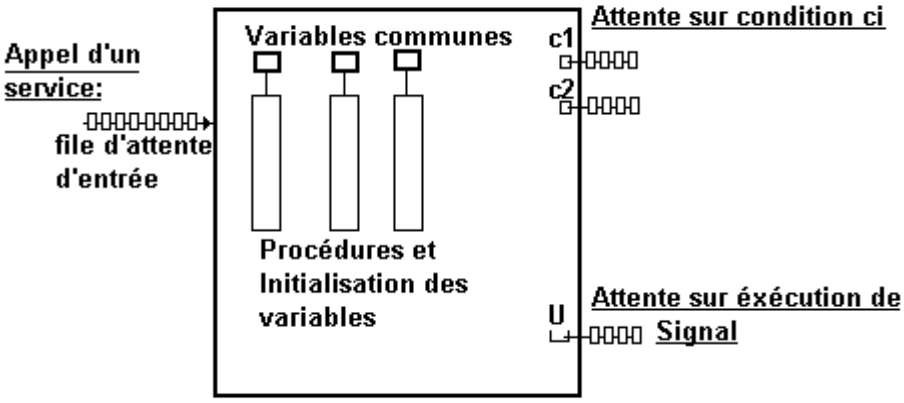
Déclarations des variables de type ***condition*** *:*
var x,y : condition ;

Les opérations **wait**, **signal et empty** sont les seules qui peuvent être appelées sur une variable conditionnelle :

x.wait : le processus qui appelle cette opération est suspendu jusqu'à ce qu'un autre processus appelle **x.signal**.

Problème important: Sémantique de la primitive Signal
La primitive Signal telle qu'elle a été définie risque de provoquer une violation de L'EM dans le moniteur car deux processus vont se retrouver simultanément actifs à l'intérieur du moniteur:
- le processus qui exécute Signal est toujours dans le moniteur
- le processus réveillé par Signal risque de continuer son exécution dans le moniteur.

Historiquement, plusieurs contraintes ont été rajoutées à la sémantique de Signal pour résoudre ce problème. C'est surtout la stratégie de HOARE qui sera adoptée et qui donnera le nom de **moniteur de Hoare** : elle donne la priorité absolue au processus réveillé et consiste à imposer le blocage momentané du processus qui exécute Signal jusqu'à ce que le processus réveillé :
- soit a quitté le moniteur
- soit s'est bloqué à nouveau dans le moniteur.



Vue Schématique d'un Moniteur de HOARE

Dans un moniteur de HOARE la hiérarchie entre les priorités d'exécution des processus sera donc la suivante:
- Basse priorité: classe des processus appelant un service du moniteur (appel de l'extérieur)
-Moyenne priorité: classe des processus exécutent la primitive Signal
- Haute priorité: classe des processus initialement bloqués sur une condition et réveillés par Signal.

7.3.3. Exemples d'application.

Exemple1: Gestion d'une ressource critique.

Protocole d'utilisation:
1)Ressource.Allouer;
2)<accès à la ressource>;
3)Ressource.Libérer;

Moniteur:

```
Ressource : monitor;
Var Libre : boolean; /*ressource libre ou occupée*/
    Roccupée : condition; /*pour la mise en attente des processus si ressource occupée*/
Begin
Procedure entry Allouer;
Begin
  If Not(Libre) then Roccupée.Wait;
  Libre:=false;
End Allouer;
Procedure entry Libérer;
Begin
  Libre:=true;
  Roccupée.Signal;
End Libérer;
Initialisation: Begin
                  Libre:=true;
                End;
End Ressource.
```

Exemple2: Rendez-vous de N processus.
Protocole d'utilisation:

RVi : Rendez-vous.Arriver;

Moniteur:

```
Rendez-vous : monitor;
Var cpt : integer; /*compte le nombre de processus arrivés à leur point de rendez-vous*/
    tous_là : condition; /* mise en attente des processus déjà arrivés*/
Begin
  Procedure entry Arriver;
  Begin
    cpt:=cpt+1;
    if (cpt<N) then tous_là.wait;
    tous_là.signal;
```

```
    cpt:=0;
  End Arriver;
Initialisation: Begin
                  cpt:=0;
                End;
End Rendez-vous.
```

Exemple 3 : Solution au problème du dîner des philosophes

Rappelons que les philosophes représentent les processus dans un système d'exploitation. Ce problème, proposé par **Dijkstra**, sert à tester les noyaux des systèmes modernes.

Nous utiliserons les variables suivantes :

var *Etat :* ***array****[0..4]* ***of*** *(penser, avoir_faim, manger) ;*
var *Cond :* ***array****[0..4]* ***of condition*** *; --- le philosophe i peut se retarder quand il a faim tant qu'il est incapable de prendre les deux baguettes ---*

Le moniteur sera noté **Diner_5**.

Code du philosophe i

```
Diner_5.ramasser(i) ;
        Manger
Diner_5.déposer(i) :
        Penser
```

Le code du moniteur sera :

```
Diner_5 = moniteur ;
Var Etat :tableau[0..4] de (penser, avoir_faim, manger) ;
Var Cond :tableau[0..4] de condition ;

        Procédure ramasser(i :0..4) ;
        Début
                Etat[i] :=avoir_faim ;
                Test(i) ;
                Si Etat[i] # manger Alors Cond[i].attendre ; -- le philosophe i est mis en
attente
        Fin ;

        Procédure déposer(i :0..4) ;
        Début
                Etat[i] :=penser ;
                Test(i + 4 mod5) ;
                Test(i + 1 mod5) ;
        Fin ;
```

```
        Procédure test(k :0..4) ;
        Début
                Si Etat[k + 4 mod5] # manger Et Etat[k] = avoir_faim Et Etat[k + 1 mod5] #
manger Alors
                        Début
                                Etat[k] :=manger ;
                                Cond[k].signaler ; -- le philosophe i est signalé qu'il mange --
                        Fin ;
        Fin ;

-- Initialisation du moniteur --
Début
        Pour i:=0 à 4
        Faire
                Etat[i] :=penser ;
Fin.
```