6 Synchronisation de processus

- Section critique et problèmes de synchronisation et d'exclusion mutuelle

- Solutions logicielles : algorithmes

- Solutions matérielles : Tas, interruptions, etc.

- Sémaphoes

- Moniteurs

- Messages

☒ Outil : la relation de précédence sur un ensemble de tâches dans un système S pour établir un ordre d'accès aux ressources.

⇒ Mais cette approche est insuffisante / indésirable pour certains problèmes, e.g. gestion d'événements **asynchrones**.

☒ **Exemple** : une ressource, 2 processus :

Deux gardiens placés aux deux entrées d'un magasin comptent les clients qui entrent dans dans le magasine par une variable unique (la ressource partagée) N.

Un autre exemple peut être le cas des tickets au rayon poissonnerie d'un magasin avec des vendeurs différents qui incrémentent le même et unique compteur.

Algorithme :

```
Tantque "le magasin est ouvert" faire
    Si entrée alors N := N+1 ; Finsi ;
FinTq ;
```

Le **programme parallèle** de cette activité :

```
Début
    N := 0 ;
    Parbegin Compteur1 ; Compteur2 ; Parend
Fin
```

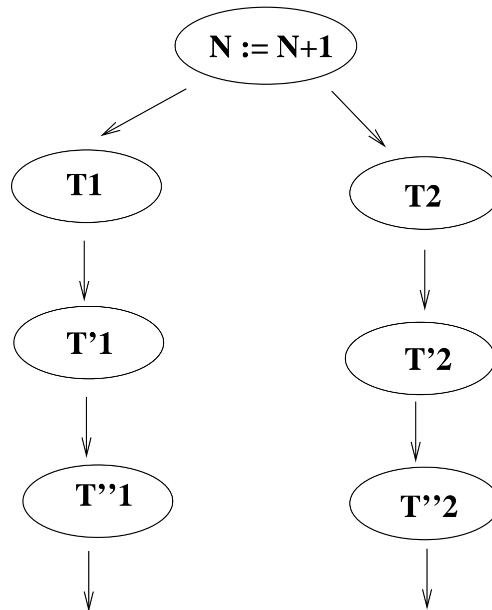
N.B : Le comptage doit avoir lieu en parallèle car l'ordre des entrées des clients est imprévisible.

On montre le problème d'accès à N par le système de tâches :

Soit $T1 = d1f1$; $T'1 = d'1f'1$..........

la représentation des incrémentations successives (la boucle 'Tanque') pour *compteur1*. Idem pour *Compteur2*

Un graphe de précédence :



⇒ La condition de **BERNSTEIN** n'est pas réalisée :

⊡ On peut avoir des comportements *correctes* :

e.g. $\boxed{d1\ f1\ d2\ f2 \text{ et } d1\ f2\ d1\ f1}$

mais aussi $\boxed{d1\ d2\ f1\ f2\ ,\ d1d2f2f1\ ,\ d2d1f1f2}$, etc.

⇒ Solution : T1 et T2 doivent être **indivisibles** (*atomiques*)

⇒ Ou bien il faut avoir ($T1\ T2$) ou ($T2\ T1$).

⊡ Les solutions acceptables transforment un graphe G pour obtenir un graphe équivalent en ajoutent des tâches à G pour rendre certaines tâches atomiques (du moins indivisibles).

⇒ Problème de **section critique** avec des solutions *logicielles* et *matérielles*.

⇒ Primitives de synchronisation pour rendre certaines exécutions indivisibles (en les enfermant dans des sections protégées).

1 Problème de section critique

⊡ N processus parallèles : $P1 \mid\mid P2 \mid\mid P3 \mid\mid$ .... avec pour chacun la structure :

Répéter   < Section restante > % les 2 sections   < Section critique > % exécutées en séquence Jusqu'à faux

⊡ Problème : pour cetrains comportements du système, plus d'un processus peut être à un instant donnée dans sa section critique.

⊡ Une solution : Transformer les graphes (processus) pour qu'à tout instant, **seul** un processus soit dans une section critique.

⇒ On dit : l'entrée et l'exécution du code de la section critique (SC) s'effectue en **exclusion mutuelle**

⊡ Technique : réaliser l'exclusion mutuelle sur les ressources lorsque plusieurs processus demandent l'accès à une ressource *non partageable.*

⊡ Dans l'exemple des gardiens, la variable N est cette ressource (de même pour une imprimante partagée où on risque d'imprimer des fichiers entrelacés !).

⊡ Principe : on transforme le code de chaque processus en :

Répéter % code exécuté en séquence

  < Section restante (SR) >

  < Section **d'entrée** >

    < SC >

  < Section **de sortie** >

Jusqu'à faux

⇒ Remarques importantes :

- Toutes les solutions proposées concernent les sections **entrée** (SE) et **sortie** (SS).

- A priori, rien ne permet de dire que les sections SE et SS sont exécutées de manière indivisible (2 processus peuvent être en même temps dans leur SE / SS).

⊡ **Un autre exemple** : imprimante et *race condition* (problème d'accès concurrent) :

- Pour imprimer, un processus place le nom d'un fichier dans un répertoire spécial (SPOOL).
- Un autre processus (démon d'impression) vérifie périodiquement ce répertoire et imprime les fichiers.
- Le répertoire Spool a un certain nombre de places (même infini) numérotés de 1 à n.
- 2 variables in pour désigner la première place libre et out pour désigner le prochain fichier à imprimer. Ces deux variables sont accessibles à tout processus.

⇒ Loi de *Murphy* et problème d'accès concurrent aux variables in et out :

- P1 lit in, il est coupé, $p2$ lit la même valeur et les deux processus placent leur fichier au même endroit.

→ L'un des deux fichiers ne sera jamais imprimé.

⊡ Solutions au problème de SC : solutions logicielles et matérielles.

2 Solutions logicielles

⊡ Une première solution naïve.

⊡ Deux processus $P1$ et $P2$ (pour plus de processus, solution plus compliquée).

- P1 et P2 se partagent une variable $Tour \in$ [1,2].
- Chaque processus consulte $Tour$; Pi entre dans SC si $Tour = i$; en sortant, il passe (donne) le tour à l'autre.

⊡ Programme principal :

```
Tour := 1
Parbegin
    P1 ; P2
Parend
```

⊡ Code de $P1$ (symétrique au code de $P2$) :

```
Répéter
    < SR1 >
    Tantque Tour=2 faire rien finTq
    < SC1 >
    Tour := 2 ;
Jusqu'à faux ;
```

⊡ Les problèmes de cet algorithme :

- P1 dans son SC, Tour=1, P2 ne peut pas entrer dans SC (⇒ exclusion mutuelle souhaitée).

- Mais, cette solution impose une **alternance** ; e.g. l'algorithme permet le comportement partiel

$$\boxed{SE1\ SC1\ SS1\ SR1\ SE2\ SC2\ SS2\ SR2}$$

- De plus, si l'un des processus s'arrête, l'autre processus ne peut s'exécuter au qu'une fois .

2.1 Conditions de comportement correct

1. Deux processus ne peuvent pas être en même temps dans leur SC (**mutex**)
2. Aucune hypothèse sur la vitesse relatives et sur le nombre de processus
3. Aucun processus suspendu à l'extérieur de sa SC ne doit bloquer les autres ou les empêcher d'entrer dans leur SC ($\Rightarrow$ absence de **blocage** $\equiv$ condition de **progression**)
4. Aucun processus ne doit attendre longtemps avant d'entrer en SC (attente **bornée**)

$\Longrightarrow$ Dans l'exemple ci-dessus, la condition de **progression** n'est pas respectée.

⊡ **Une deuxième solution**

- $P1$ et $P2$ se partagent deux variables booléennes $D1$ et $D2$ initialisées à faux.

- Elles correspondent aux demandes de P1 et P2 pour entrer en SC.

⊡ Code de $P1$ (symétrique au code de $P2$) :

```
Répéter
    < SR1 >
    D1 := vrai
    Tantque D2 faire rien finTq
    < SC1 >
    D1 := faux;
Jusqu'à faux;
```

⊡ Le problème de cet algorithme : possibilité de blocage (Morphy) :

si les deux processus ont fait $Di := vrai$, chaque processus attendra dans sa SR.

⊡ Une troisième solution avec attente bornée (équité) :

équité : si un processus est en attente de SC, borner le nombre de fois où les autres entrent dans la SC.

⇒ Algorithme de **Peterson**

Peterson : cet algorithme garantie les 4 conditions : mutex, absence de blocage, progression, attente bornée.

$\boxdot$ $P1$ et $P2$ se partagent une variable $Tour$ (init 1) et deux variables booléennes $D1$ et $D2$ initialisées à faux.

$\boxdot$ Code de $P1$ (symétrique au code de $P2$) :

```
Répéter
        < SR1 >
        D1 := vrai
        Tour := 2
        Tantque D2 && Tour=2 faire rien finTq
        < SC1 >
        D1 := faux;
Jusqu'à faux;
```

$\Rightarrow$ Le seul cas où $P1$ ne peut pas entrer en SC1 se produit lorsque (idem pour $P2$) :

- P2 a demandé à entrer dans SC2 (D2=vrai)
- C'est le tour de P2 (Tour=2)
- Si cette écriture vient de P1, c'est P2 qui entre sinon P2 sera bloqué et P1 entre.

Rappel : une variable accédée (par 2 processus et en même temps) aura la valeur de la dernière écriture.

$\boxdot$ Chaque processus qui demande l'entrée en SC attendra au plus un tour.

⊡ On peut établir la table de vérité de cet algorithme et constater son fonctionnement correct :

D1	D2	Tour	Comportement
V	V	1	P2 (hors SC) attend, P1 en SC
V	V	2	P1 (hors SC) attend, P2 en SC
V	F	1	P2 (hors SC) attend, P1 ?
V	F	2	P1 (hors SC) attend, P2 ?
F	V	1	P1 (hors SC), P2 ?
F	V	2	P1 (hors SC), P2 ?
F	F	1	P1 et P2 hors SC (car Di=faux)
F	F	2	P1 et P2 hors SC (car Di=faux)

⊡ Les solutions logicielle sont utilisées dans les multi processeurs (un peu plus compliqué).

Leur adaptation est plus compliquée dans les systèmes répartis (en général sans mémoire commune mais avec des messages) .

⊡ Autres solutions logicielles : algorithmes

- de "Decker"

- de "Eisenberg"

- de "Dijkstra",

-etc. (Page 22-2 R/V du manuscrit)

⊡ Page SE 22-1 du manuscrit : solution pour N processus.

⊡ Le problème **d'attente active** de tous ces algorithmes : voir plus loin.

3 Solutions matérielles

⊡ Utilisées plutôt dans les système mono processeurs.

→ (1) masquage d'it°, (2) TAS, (3) verrouillage de bus.

⊡ **1- Masquage/désarmément d'Interruption :**

- Le moyen le plus simple : on masque les interruptions pendant la SC (y compris l'interruption Horloge par le SE).

→ Particularité : cette solution n'est pas utilisable dans les multi processeurs.

↘ Une solution : spécialiser un des processeurs.

Problèmes de masquage :

- Danger de laisser un utilisateur masquer tout. S'il oublie de démasquer ?

- Si un processus prioritaire doit passer ?

Remarque : Néanmoins, le principe de *masquage total* est utilisé par le noyau pour e.g. mettre à jour la table des processus (PCB) et pour la gestion des "Prêts".

Les autres solutions matérielles :

⊡ 2- TAS .

⊡ 3- Verrouillage et réquisition du bus.

3.1 TAS : Test and Set

- Avec (éventuellement) le verrouillage du bus.

- **Tas** : instruction élémentaire (insécable, atomique, exécutée en 1 cycle) pour lire et écrire un mot mémoire.

- Deux opérandes : un registre R et un mot mémoire B .

- On copie B dans R et on place 1 dans B.

- Le code suivant simule le comportement du *TAS* :

```
Procédure TAS (var a,b : entier)
    Début
        a := b ; b :=1 ;
    Fin TAS ;
```

TAS résout le problème de SC de la manière suivante :

- Les processus partagent une variable *Verrou*.

- Chaque processus Pi possède une variable locale $Testi$.

⊡ Code de Pi :

```
Testi : entier;
Répéter
    <SR>
    TAS(testi, Verrou);
    Tantque Testi=1 faire
        TAS(testi, Verrou);
    finTq
    < SC >
    Verrou := 0;
Jusqu'à Faux;
```

⊡ Le premier processus qui exécute TAS trouve $Verrou = 0$ et entre en SC.

Les autres trouveront $Verrou = 1$ et attendront.

⊡ Cette solution garantie *l'exclusion mutuelle* mais pas l'attente bornée (assurée par d'autres moyens).

⊡ IBM360 était la première machine à proposer TAS (TST). Par la suite, Motorola a implanté TAS puis Intel iAPX86 a proposé XCHG qui échange le contenu d'un registre avec un mot mémoire.

Code de XCHG ($\simeq$ TAS) :

$$Reg = 1; \quad XCHG \;\; Reg, Verrou$$

$$\Rightarrow Reg := Verrou \text{ et } Verrou := Reg = 1$$

⊡ On peut se servir de TAS dans les multiprocesseurs mais $Verrou$ est en mémoire $\Rightarrow$ différents processus mettent Verrou à 1 mais d'autres (sortie de SC) le mettent à 0.
$\Rightarrow$ Solution : verrouillage du bus pendant l'exécution de TAS (même si TAS est insécable) ou trouver une gestion de la variable Verrou (cf. Motorola 680xx).

3.2 Les inconvénients de TAS

⊡ **Problème1** : Attente active (on tourne en rond) et on utilise l'UC.

⊡ **Problème2** : Inversion des priorités (les deux problèmes peuvent être liés) :
$\Rightarrow$ Un processus $P1$ de faible priorité prend l'accès ; $P2$ avec une forte priorité prend l'UC $\Rightarrow$ on ne s'en sort plus ! !
$\Rightarrow$ Blocage d'un type particulier difficile à détecter/éviter (garder une trace de toute instruction coûte cher).

3.3 Solutions aux problèmes de TAS

⊡ Pour le Problème-1 (pour Problème2, voir plus loin) :

- Remplacer l'attente active par une attente passive.

- Quand le Verrou sera libéré, on réveillera un processus en attente

```
Testi : entier ;
Répéter
    <SR>
    TAS(testi, Verrou) ;
    Si Testi=1 alors <se mettre en attente> FinSi ;
    < SC >
    S'il existe < Réveiller un Processus >
    Sinon Verrou := 0 ;
    Finsi
Jusqu'à Faux ;
```

⇒ Mais cette solution peut poser problème :

Supposons les codes de processus P1, P2 et P3.

P1	P2	P3
TAS(test1, Verrou) ;	$\downarrow$	*% P_3 en <SC>*
Si Test1 = 1 alors	$\downarrow$	$\downarrow$
<attente>$\rightarrow$	*% P_1 est interrompu*	$\downarrow$
FinSi	TAS(Test2, Verrou) ;	$\downarrow$
$\downarrow$	Si Test2 = 1 alors $\rightarrow$	*% P_2 est interrompu*
$\downarrow$		*% P_3 sort de la $< SC >$ et*
$\downarrow$		*% réveille P_1 (seul en attente)*
% Ici, P1 est réveillé par P3	$\longleftarrow \longleftarrow \longleftarrow$	<Réveiller P_1>
$< SC >$		
Verrou := 0 ;		$\downarrow$
<Réveiller ?>		$\downarrow$
$\downarrow$	**<attente> infinie !**	
	FinSi	

⇒ SC libérée mais un processus (P2) attend

⇒ Attente perpétuelle pour $P2$ si P1 reste dans sa SR et P3 ne demande pas SC ! ! (il peut la demander)

⇒ Le problème vient du fait que <Réveiller> s'exécute trop tôt.

⇒ Pour régler ce Problème, il faut garder la trace de plusieurs réveils.

⇒ C'est l'objet d'autres mécanismes (e.g. *Sémaphores*).

⇒ D'autres mécanismes : SLEEP (wait, attendre) et WAKEUP (signaler, réveiller).

- SLEEP : suspendre l'appelant (par un SVC) en attendant un réveil par un autre processus .

⊡ Exemple du modèle Producteur-Consommateur avec SLEEP/WAKEUP (V. manuscrit SE-24.1 Verso.)

4 Sémaphores

Un sémaphore S est un entier à laquelle on n'accède que par deux primitives P et V.

4.1 Version 1 : simulation code avec attente active sans file d'attente

P(S) : Tanque S $\preceq$ 0 faire **rien** FinTq ;

S := S -1 ;

V(S) : S := S +1 ;

NB : avec éventuellement *init(S,val)* où val=nombre de ressources (val=1 pour SC dans le problème mutex simple).

⇒ Important :

- dans les opération P et V, le test et la modification de S sont exécutés de manière **indivisible** (à l'aide d'un TAS par exemple).
- les procédures P et V n'ont pas besoin d'être exécutées en mutex (Exclusion Mutuelle) même si c'est souvent le cas car plus simple à traiter.

⊡ Pour implanter P et V dans un mono processeur, on peut désarmer les interruptions. Ce qui ne suffit pas dans un multi processeur / système réparti sauf si les Interruption sont gérées par l'un des machine/processeurs.

⊡ Dans un multi processeurs avec la mémoire commune, on utilise un TAS ou bien l'on verrouille le bus pendant l'accès (ou même pendant P et V dont les codes sont courts).

⊡ Pour assurer l'exclusion mutuelle pour N processus, on utilise un *mutex init 1*.

⊡ Exemple de code de Pi :

```
Répéter
    <SR>
    P(mutex)
    <SC>
    V(mutex)
Jusqu'à faux
```

⇒ Explication et fonctionnement.

⇒ La logique des Sémaphores (le principe) est implantée de diverses manières.

Problème d'attente active réglé avec wait / signal.

```
Type Sémaphore : enregistrement
     val : entier ;
     File-d-attente : File de PCB
Fin Enregistrement
```

⊡ Init : *Sémaphore S(File-d-attent => vide, val => val_init) ;*

⊡ **Codes de P et V modifiés :**

```
P(S) : Si (S.val ⪯ 0) alors
          <Ajouter PCB de l'appelant à S.File-d-attente>
          <mettre l'appelant en attente>
       Sinon S.val := S.val-1 ;
       Finsi
```

```
V(S) : Si (S.File-d-attente ≠ vide) alors
          <Choisir et enlever un PCB de S.File-d-attente>
          <mettre ce processus dans Prêts>
       Sinon S.val := S.val+1 ;
       Finsi
```

⊡ Le choix du PCB dans la File d'attente :

$\rightarrow$ problème de famine $\Rightarrow$ gestion Fifo.

N.B. : les sémaphores d'OS/2 :

- OS2 possède deux types de sémaphores : binaires et système avec des files séparées
- On peut attendre sur plusieurs sémaphores (Linux aussi).

4.2 Autres réalisations de Sémaphore

⊡ **Avec masquage / démasquage (attente active)** :

```
P(S) :  masquer
        Tantque (S.val ⩽ 0) faire
                démasquer ; NOP ; masquer ;
        FinTq
        S.val := S.val -1 ;
        démasquer
```

```
V(S) :  masquer
        S.val := S.val + 1 ;
        démasquer
```

De facto, P et V sont ici non interruptibles (insécables).

⊡ **Avec TAS (attente passive)** :

```
P(S) : % accès à S.val en mutex à l'aide de TAS
        TAS(Test, Verrou) ;
        Tantque (Test = 1) faire
                TAS(Test, Verrou) ;
        FinTq ;
        Si (S.val ⩽ 0) alors
            Verrou=0 ;
            <Attendre dans S.File-d-attente >
        Sinon S.val := S.val -1 ; Verrou=0 ;
        FinSi
```

```
V(S) :
      % <Ici, le même code pour accéder à S.val par un TAS>
        Si (S.File-d-attent ≠ vide) alors
            < Réveiller un processus, PCB → Prêts >
        Sinon S.val := S.val + 1 ;
        Finsi
        Verrou=0 ;
```

⇒ Ici, on a une attente active pour passer le TAS.

⊡ **Une autre version de code simplifiée** :

Il nous faut accéder à P et V en mutex.

```
P(S) :   S.val := S.val -1 ;
         Si (S.val ≺ 0) alors
             <Attendre dans S.File-d-attente >
         FinSi
```

```
V(S) :   S.val := S.val +1 ;
         Si (S.val ⪯ 0) alors            % ou si File ≠ vide
             < Réveiller un processus, PCB → Prêts >
         Finsi
```

⊡ Pour garantir l'indivisibilité, on peut placer *solo()* et *Tuti()* au début et à la fin de ces codes (au moins, pour les accès aux variables) avec :

- **solo()** : seul : masquage ou prise de Verrou ;
- **tuti()** : tous : l'inverse

5 Utilisation des sémaphores

⊡ Pour les problèmes de synchronisation

⊡ Pour les problèmes d'exclusion mutuelle

5.1 Problèmes de synchronisation

⊡ synchronisation et contraintes de précédence

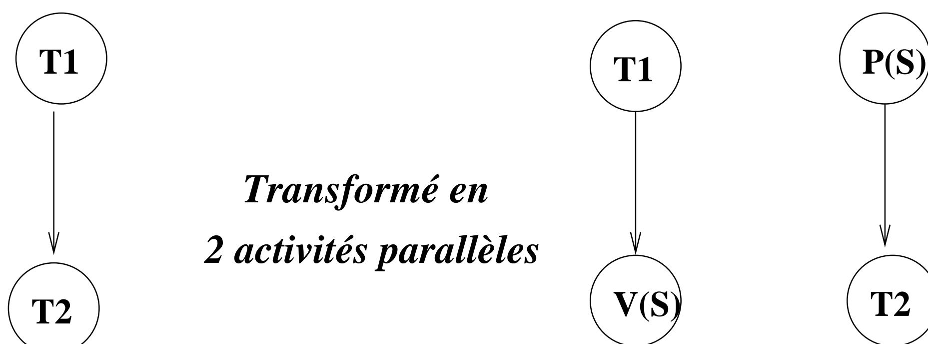
Exemple 1

- Soient deux tâches $T1$ et $T2$ avec la contrainte $T1 < T2$.

- Elles peuvent se présenter dans n'importe quel ordre.

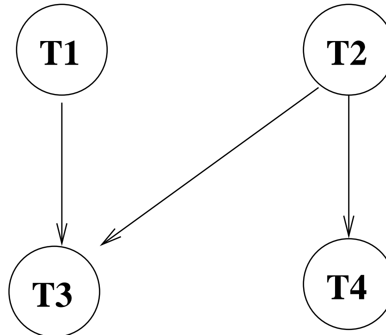
```
S : Sémaphore init 0 ;
parbegin
        begin T1 ; V(S) ; end ;
        begin P(S) ; T2 ; end ;
parend
```

⇒ Le graphe de précédences des tâches est remplacé par un graphe équivalent.



Exemple 2

- Soient le graphe de précédences suivant :



- Une solution qui minimise le nombre de sémaphores :

```
S : Sémaphore init 0 ;
parbegin
        begin T1 ; P(S) ; T3 ; end ;
        begin T2 ; V(S) ; T4 ; end ;
parend
```

⊡ Les sémaphores servent à traiter des problèmes plus complexes que la contrainte de précédence.

5.2 Problèmes d'exclusion mutuelle

⊡ Exemple de distribution de ressources :

- Soit K exemplaires d'une ressource ($K \geqslant 2$)

- Pour un (parmi N) processus, l'accès à X exemplaires de la resource est une SC.

- Pour $X = 1$, on peut avoir au plus K processus en SC.

- Une solution (le code de chaque processus) :

```
S : Sémaphore init K;
Répéter
        <SR>
        P(x);
        <SC>
        V(x);
jusqu'à faux;
```

⊡ Sémaphores **privés** : les P et V ne sont pas forcément faits par le même processus (cf. les exemples de synchronisations précédents)

⊡ Inconvénient des sémaphores : l'utilisateur doit bien placer P et V → risque de blocage en particulier lorsque le code est compliqué avec des sémaphores privés.

⇒ Outil de plus haut niveau : **Moniteurs.**

5.3 Suite exercices

(P. manuscrit SE30, ...)

1- producteur consommateur

1-1 : producteur consommateur simple synchronoisés (taille=1)

1-2 : tampon non borné

1-3 : tampon borné (taille=N)

1-4 : N producteur / M consommateurs (M=1, M>1)

1-5 : contraintes de consommation : tout doit être consommé (par tous)

2- Lecteur Rédacteur

2-1 : Un / N lecteurs , un /M rédateurs

2-2 : Priorité aux lecteurs / rédacteurs

3- Les 5 philosophes

4- La discribution de ressources (billes)

5- Coiffeur

6- etc.

5.4 Exemples

1-1 : producteur consommateur simple (*size(tampon) = 1*)

I- Schéma de base (insuffisante) :

Producteur()	Consommateur()
Répéter	Répéter
Produire un object **M**	Retirer un object **M**
Ajouter M au Tampon	Consommer M
Jusqu'à faux	Jusqu'à faux

Problème : les ajouts et les retraits s'entremèlent !

⇒ Première solution : protéger le tampon par un sémaphore *mutex* (pour l'exclusion mutuelle).

II- Schéma amélioré (mutex : Sémaphore init 1) :

Producteur()	Consommateur()
Répéter	Répéter
Produire un object **M**	*P(mutex)*
P(mutex)	Retirer un object **M**
Ajouter M au Tampon	*V(mutex)*
V(mutex)	Consommer M
Jusqu'à faux	Jusqu'à faux

Problème : Le consommateur risque de retirer un objet M non encore créé !

⇒ Deuxième solution : synchronisaer avec un sémaphore

III- Schéma amélioré (+ synchronisation) :

mutex : Sémaphore init 1

pret : Sémaphore init 0 – signaler une production

Producteur()	Consommateur()
Répéter	Répéter
Produire un object **M**	P(pret)
P(mutex)	*P(mutex)*
Ajouter M au Tampon	Retirer un object **M**
V(mutex)	*V(mutex)*
V(pret)	Consommer M
Jusqu'à faux	Jusqu'à faux

Remarques :

- Cet exemple montre l'utilisation des sémaphores en exclusion mutuelle et en synchronisation.

- Ce schéma fonctionne bien pour un Producteur et un Consommateur (quid de P conso et Q prod).

- L'exemple ne donne pas de détail sur la gestion du tampon qui peut être borné ;

- Si deux consommateurs, on ne peut pas garantir que tous les consommateurs retire chaque production

Solutions : Gestion d'un tampon borné de taille N :

- On peut prévoir un sémaphore (*Plein : sémaphore init N*) qui laissera passer N fois le producteur.

- Ajouter une gestion circulaire du tampon pour les ajouts et les retraits.

- La gestion du *Tampon[1..N]* :

-Deux entiers **In** et **Out** initialisés à 1 ;

-Le Producteur utilise **In** et Tampon**[In]** est la case où il place la prochaine production ;

- Le Consommateur utilise **Out** et Tampon**[Out]** est la case d'où il retire une production ;

IV- Schéma Tampon borné de taille N :

- Hypothèse : 1 Producteur et 1 Consommateur

- Plus besoin de sémaphore pour gérer l'accès au tampon ;

vide : Sémaphore init N

plein : Sémaphore init 0

Producteur()	Consommateur()
Répéter	Répéter
Produire un object **M**	P(plein)
P(vide)	M ← T[Out] ;
T[In] ← M	Out ← (Out +1) % N
In ← (In +1) % N	*V(vide)*
V(plein)	Consommer M
Jusqu'à faux	Jusqu'à faux

Remarques :

- Ce schéma fonction pour un Prod et un conso.

- Il garantit que le conso récupère toutes les productions

- Dans ce dernier cas, toute production est consommée par chaque conso ?

- Quid de P producteurs et Q consommateurs (si 2 producteurs font *P(vide)* puis lisent la même valeur de *In*) ?

- Solution : protéger l'accès à In (Out).

V- Schéma Tampon borné de taille N et P producteurs et Q consommateurs :

vide : Sémaphore init N ; *plein : Sémaphore init 0*

Producteur()	Consommateur()
Répéter	Répéter
Produire un object M	P(plein)
P(vide)	**Get(M)**
Put(M)	*V(vide)*
V(plein)	Consommer M
Jusqu'à faux	Jusqu'à faux

mutexIn, mutexOut : Sémaphore init 1

Fonction Put(+ Objet M)	Fonction Get(- Objet M)
int l ;	int l ;
P(mutexIn)	*P(mutexOut)*
l ← In ;	l ← Out ;
In = (In +1) % N ;	Out = (Out +1) % N ;
V(mutexIn)	*V(mutexOut)*
T[l] ← M	M ← T[l]

Remarques :

- Protection des variables In et Out.

- Le rôle de la variable locale "l"

VI - Schéma P producteur et Q consommateurs.

- les consomateurs lisent tous chaque production (mais pas les anciennes !)
- les producteurs doivent attendre la consommation par tout producteur (le rôle de Vide).
- Pour simplifier, supposons un tampon de taille 1

*vide, **mutex_cpt** : Sémaphore init 1 ; plein : Sémaphore init 0 ;*

Producteur()	Consommateur()
Répéter	Répéter
Produire un object M	*P(mutex_cpt)*
P(vide)	cpt := cpt +1 ;
Mettre(M) dans Tampon	Si (cpt = 1) – je suis le 1er
V(plein)	Alors *P(plein)* ;
Jusqu'à faux	V(mutex_cpt) ;
	Prendre M ;
NB : pour un Tampon de taille N,	P(mutex_cpt)
on utilise **put(M)** dans Producteur ;	cpt := cpt -1 ;
mais dans Consommateur,	Si (cpt = 0) – je suis le der
on ne fera +1 sur **Out** que	Alors V(vide) ;
si l'on est le dernier (cpt=0)	V(mutex_cpt) ;
	Consommer M
	Jusqu'à faux ;

Attention : schéma de principe ; lire les remarques.

Remarques : le shcéma précédent n'empêche pas un consommateur de lire deux fois la même production.

Pour éviter ce cas, on peut marquer la récupération d'un objet par un consommateur. Pour ce faire, on peut procéder de plusieurs manières :

1. définir un tableau de booléens où les indices représentent les processus consommateurs (avec un hashage éventuel Pid → indice)
2. gérer une *liste* de Pid de consommateur par objet **m**.
3. faire attendre les consommateurs qui ont retiré un objet **m** sur un (autre) sémaphore, en attendant que les autres retirent le même objet.
4. Le consommateur lui même sauvegarde des traces de l'objet **m** qu'il a déjà retiré/consommé.
5. le premier consommateur qui aura retiré un objet **m** libère le (les) producteurs(s) puis permet aux autres consommateurs d'avoir accès à cet objet.

5.5 Application des sémaphores

Problème : soit K exemplaires d'une ressource R et N processus ($N > K$).
Chaque processus peut demander X ($X \geqslant 1$) exemplaires de R.

```
K : shmem init N;     // K=nb. de R restant
mutex : semaphore init 1;  // protection de K
Procedure prendre(+X : entier)
{Attente : bool init true;
 Tant que(Attente)
   {P(mutex);   // accès à K
    Si (K >= X)  Attente := false;
    Sinon {V(mutex); sleep(1);}
   } // on peut prendre X exemplaires de R
 K := K - X;
 V(mutex);
}
Procedure rendre(+X : entier)
{P(mutex);   // accès à K
 K := K + X;
 V(mutex);
}
```

6 Moniteurs

⊡ Motivation : palier les problèmes des Sémaphores et dispenser l'utilisateur des détails de synchronisation et d'exclusion mutuelle (e.g. P et V des sémaphores).

⊡ Mécanisme proposé par *Brinvh-Hansen, Dijkstra et Hoare*

⊡ **Un Moniteur** : une strcuture contenant des variables (d'état) partagées et du code de manipulation de ces variables.

⇒ Existe dans certains langages comme Concurrent Pascal sous forme d'une strcture de données.

⊡ La strcuture Moniteur contient :

- des variables partagées uniquement accessibles dans le Moniteur
- des procédures / fonctions internes seules à pouvoir manipuler les variables
- ces fonction et leurs paramètres = l'interface avec l'extérieur.
- une partie initialisation des variables

⊡ Exemple des gardiens :

```
Type nb_clients = Moniteur
var N : entier;
Proc incrémenter
begin
        N++;
end;
begin % initialisation
        N := 0;
end;
```

⇒ Les processus $compteur1$ et $compteur2$ (les gardiens) :

```
Tantque <le magasin est ouvert> faire
        Si <entrée> alors nb_clients.incrémenter; Finsi;
FinTq;
```

⊡ **Avantages des Moniteurs :**

- La SC est mise dans un Moniteur
- Les SC sont transformées en procédures / fonctions de Moniteur au lieu d'être dispersées
- La gestion de la SC n'est plus à la charge de l'utilisateur
- Le Moniteur garantie qu'au plus un processus à la fois peut accéder à cette structure...

- Un sémaphore binaire (mutex) suffira pour garantir cette exclusion mutuelle.

- Le Moniteur tout entier est implanté comme une SC.

⊡ Fonctionnement :

- un processus appelle une routine d'un Moniteur

- il y entre si le Moniteur est libre

- sinon, son PCB est mis en file d'attente associée à ce Moniteur (e.g. file du sémaphore).

- Quand le Moniteur deviendra libre, un processus est sorti de cette file et entre dans le Moniteur.

- Ce fonctionnement assure une exclusion mutuelle.

- Pour le synchronisation, les Moniteurs utilisent des **conditions**

- L'utilisteur ne demande que l'entrée, pas de demande de sortie nécessaire → plus de risuque d'oubli !

⊡ Variables de type **condition** :

- Donnent aux Moniteurs la possibilité de synchronisation des sémaphores.

- Déclarartion dans le Moniteur sous forme *X : condition*

→ X désigne une file d'attente (dans le Moniteur) manipulée par *wait* et *signal*.

→ une condition n'est pas un compteur (comme les sémaphores) : on ne mémorise pas les signaux pour un traitement ultérieur.

- un processus (dans le Moniteur) exécute *wait* sur une condition, se met en attente de la réalisation de la condition → le Moniteur devient libre.

- l'exécution d'un *signal* sur une condition par un processus déclenche le réveil (s'il y en a) d'un processus en attente dans la file de cette condition .

⇒ L'implantation de *signal* doit régler deux problèmes :

1- le choix du processus à réveiller et à sortir de l'attente

2- le choix du processus qui doit continuer dans le Moniteur (le signaluer ou le réveillé ?)

→ risque d'avoir deux processus dans le Moniteur !

- Il y a aussi ceux qui demandent le Moniteur !

→ 3 processus à gérer.

⊡ Plusieurs solutions (les files gérées en FIFO) :

- Hoare : priorité au signaleur : on attend qu'il quitte le Moniteur (ou qu'il se mette en attente sur une autre condition) pour passer la main au processus réveillé.

- Concurrent Pascal : Hansen : le signaleur quitte immédiatement le Moniteur (le processus réveillé continue).

→ dans le code, il ne faut rien mettre après *signal*.

⊡ Exemple : N processus, une ressource

```
Type allocateur = Moniteur
var occupé : bool ;
var libre : condition ;
Proc acquisition
begin
        Si occupé alors libre.wait Finsi ;
        Occupé := true ;
end ;
Proc libération
begin
        Occupé := false ;
        libre.signal ; % on peut sortir de Moniteur.
end ;
begin           % initialisation
        Occupé := false ;
end ;
```

⇒ On remarque l'analogie entre les deux procédures et P et V d'un sémaphore.

⊡ Un exemple : lecteurs rédacteur

- N processus : 1 rédacteur, N-1 lecteurs (critiques).
- une masion d'édition : le Moniteur
- Les lecteurs attendent le livre.
- Il faut garantir qu'ils auront tous le livre.

```
Type Edition = Moniteur
var livre_dispo : bool ;  livre : string ;
var dépot_fait : condition ;
Proc Dépot(l : string)
begin
      livre_dispo := true ; livre := l ; % dépôt physique du livre
      depot_fait.signal ;
end ;
fonction Retrait () → string
begin
      Si not livre_dispo alors
            depot_fait.attendre ;
            depot_fait.signaler ;       % signal en chaîne
      Finsi
      return livre ;
end ;
begin      livre_dispo := false ;       end ;


Code de l'écrivain :                     Code d'un lecteur :
livre : string ;                         livre : string ;
<élaborer le livre>                      livre := Edition.Retrait() ;
Edition.Depot(livre) ;                   <Critiquer le livre>
```

⇒ discuter des détails.

Implantation des Moniteurs

- Avec des sémaphores (voir BEs SE)

- etc.

Inconvénients des Moniteurs

- peu de langages de support

- utiles dans les systèmes à mémoire partagée (pas distribués)

$\Rightarrow$ d'où la solution : passage de messages.

7 Messages

- Utiles dans les systèmes distribués pour régler les problèmes de synchronisation et d'exclusion mutuelle.

- Existent aussi dans les systèmes centralisés.

- Primitives (appels SVC) :

  – **send(dest, message)**

    bloquant / non bloquant

  – **receive(source, message)**

    Si source=Any,on reçoit de tous

    bloquant / non bloquant

Propriétés des systèmes d'échange de messages :

- aussi intéressant que les mécanismes Sémaphore / Moniteur
- les messages peuvent se perdre !
  ⇒ acquittement dans un délai fixe, sinon, on recommence.
- si l'acquittement se perd ! on risque d'emettre 2 fois
  ⇒ numéroter les messages
- savoir nommer les processus dans send et receive ?
  ⇒ format : processus@machine.domaine
- Authentification : à qui a-t-on réellement affaire :
  - comment authentifier le (vrai) serveur de fichiers
  - comment le serveur sait que c'est un (vrai) client
  ⇒ codage des messages avec clefs (connues des utilisateurs). → cf. Cryptographie
- problème de performances : c'est plus lent qu'un Sémaphore / Moniteur
  -dépend de la vitesse de traitement et de transmission
  -on utilise des registres pour traiter des messages courts

⊡ Un exemple :

Producteur - consommateur avec une Boîte de taille N

Send non bloquant, receive bloquant

```
Producteur :
    M : message
    while (true)
        <produire DATA>
        receive(Consommateur, message vide M)
        <créer message M avec DATA >
        send(Consommateur, M)
    end while

Consommateur :
    M : message := vide ;
    for i : 1..N
        send(Producteur, M)
    while (true)
        receive(Producteur, message M)
        <retirer Data de M>
        send(Producteur, message vide M) % débloque le prod
    end while
```

⊡ La communication inter processus d'Unix peut se faire aussi avec des tubes (pipes) de communication

⊡ La différence avec une boîtes aux lettres (messages) : les messages ne sont pas délimités dans un tube.

⇒ une lecture de tube peut donner K messages à la fois (qu'il faudra traiter)

⇒ on peut lire des messages de taille fixe .

⊡ On peut démonter l'équivalence entre sémaphores, Moniteurs et les messages.

⇒ chacun peut être implanté à l'aide des autres

⇒ Voir aussi BE-SE : boîte aux lettres.