VI. Accès Concurrent : Synchronisation avec Sémaphores

1. Principe de l'Accès concurrents

Les processus ne sont pas tous indépendants les uns des autres. Ils peuvent être en **interaction** s'ils accèdent à des **ressources communes**. Ils ont besoin de communiquer des données malgré leurs états général qui asynchrone. Cependant, pour faire **communiquer** deux ou plusieurs processus, il faut les **synchroniser.** Dans cette partie nous étudions les différentes méthodes de synchronisation. Nous nous intéressons plus particulièrement à la synchronisation par les sémaphores.

1.1 Définitions :

On appelle ressource **critique** une ressource qui est à **accès unique**. Un processus peut accéder à une ressource critique à différents moments

On appelle **section critique** l'ensemble d'instructions permettant à un processus d'accéder à une ressource critique

On dit que deux ou plusieurs processus sont en **exclusion mutuelle** lorsqu'ils **exécutent** *en même temps* leur **section critique** (pour une même ressource critique).

Exemple :

L'UC est considérée comme une ressource critique**.** En effet plusieurs processus partagent l'UC. Pour l'UC, tous les processus sont exclusion mutuelle. Dans ce cas particulier c'est l'allocateur (Ordonnanceur, *scheduler*) qui résout ce problème par des algorithmes de type FIFO (First Job In First Job out) , SJF (Small First Job) , ou le Tourniquer.

Dans les autres cas, c'est aux processeurs eux-mêmes de faire appel à des techniques de synchronisation

Méthode de résolution de l'exclusion mutuelle

- par **action direct** sur l'état d'un processus
- par **attente active et scrutation** de variables commune
- par **mise en file d'attente** avec partage de variables communes
- par **communication** par des **messages**
- par utilisation de **sémaphores**

1.2 Résoudre le problème de l'exclusion mutuelle

Soient *n* processus en exclusion mutuelle pour une ressource critique, pour résoudre le problème de l'accès concurrent à la section critique :

- Chaque fois qu'un processus veut exécuter sa section critique, il doit le signaler aux autres processus (**Prologue**)
- Chaque fois qu'un processus sort de sa section critique, il doit le signaler aux autres processus (**Epilogue**)
- Tous les processus exécutent le même prologue et le même épilogue, s'ils ont les mêmes droits d'accès à cette ressource critique

Pour chaque processus il faut écrire son *Prologue* et son *Epilogue.*

```
Début

      Prologue ;
      <Section Critique> ;
      Epilogue ;

Fin
```

Règles d'écriture du Prologue et de l'Epilogue

- **Règle 1 :** une **seule** exécution de **SC** à la fois
- **Règle 2 :** seuls les processus qui exécutent leur **prologue** ou **épilogue** peuvent décider de **laisser** ou pas un **processus d'entrer** dans sa **SC**
- **Règle 3 :** un processus doit **entrer** dans sa **SC** en un **temps fini**

2. Synchronisation par Sémaphores

A l'aide de la technique par évènements ou par verrou, il n'est pas possible de connaître le nombre de processus bloqués dans la file d'attente en attente de la ressource, ou bien de mémoriser plusieurs demandes etc.

Dijkstra a introduit en 1968, dans le système THE, le concept de **sémaphore.**

Un **sémaphore *s*** est un objet abstrait composé d'une **file d'attente** et d'une **valeur**

- La **file d'attente** du sémaphore, notée ***File(s)***, contient les **processus bloqués** sur ***s***
- La **valeur** du sémaphore est notée ***Val(s)***

- ❖ Initialement la file d'attente *File(s)* est vide, et le sémaphore possède une valeur $Val(s) \geq 0$
- ❖ *File(s)* et *Val(s)* ne peuvent être manipulées que par les deux primitives **indivisibles** et **non interruptibles *P(s)*** et ***V(s).***

Primitive ***P(s)*** :

Début
Val(s) := *Val(s)* - *1* ;
Si *Val(s)* < *0* **Alors**
Mettre le processus actif dans la file *File(s)* ;
Fin

Primitive ***V(s)*** à la sortie de la section critique :

Début
Val(s) := *Val(s)* + *1* ;
Si *Val(s)* ≤ *0* **Alors**
{il y a au moins un processus bloquée dans la File(s)}
Sortir un processus de la file *File(s)* ;
Fin

Utilisation :

P(s)
<Section Critique>
V(s)

La primitive P(s) est à exécuter avant d'entrer en section critique
La primitive V(s) est à exécuter à la sortie de la section critique

Remarques :

- un processus qui se bloque en exécutant *P(s)* termine l'exécution de cette instruction. Lorsqu'il est à nouveau actif, il exécutera l'instruction suivant *P(s)*
- un **seul processus** peut **exécuter *P(s)*** ou ***V(s)*** à un instant donner
- *Val(s)* peut devenir négative après une plusieurs exécution de *P(s)*
- si on veut **autorisée *N* exécutions** de *P(s)* (par un seul ou plusieurs processus), ***Val(s)*** doit être initialisée à ***N***

3. Exclusion Mutuelle

But : Protéger l'accès à une ressource unique (exemple : variable, imprimante, etc.) par plusieurs processus P1, P2, P3, ...Pi

Contexte commun : var *mutex* : sémaphore initialisé à *1*

pour chaque processus Pi :

Processus P_i

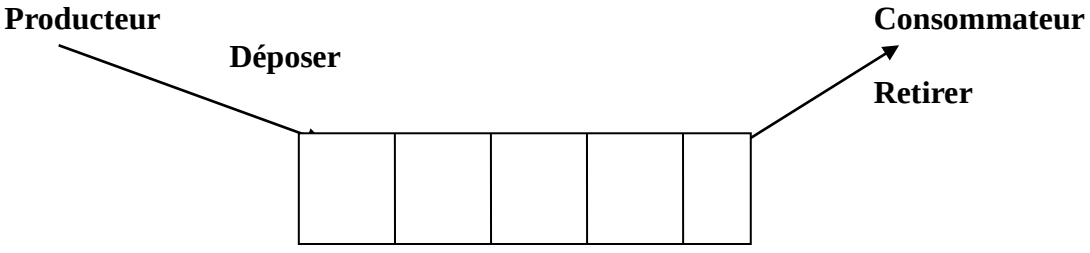
Début

……
P(mutex)
{section critique}
V(mutex)
……

Fin

4. Synchronisation Producteur/ Consommateur

A présent nous allons utiliser le concept des sémaphores pour la résolution du problème Producteur Consommateur. Deux processus partagent une mémoire fixe. Producteur met des information dans la mémoire tampon, le consommateur les retirent.



Problèmes :

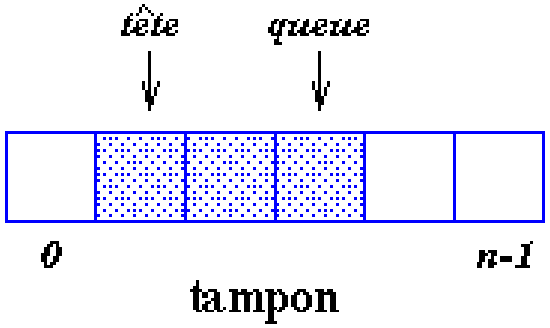
1. Producteur veut déposer un message alors que la mémoire est pleine.
2. Consommateur veut retirer un message alors que la mémoire est vide.
3. Producteur et consommateurs ne peuvent pas s'exécutées en même temps.

Nous traitons en premier le cas où la mémoire tampon contient une case. En suite nous traitons la solution du problème dans le cas d'une mémoire tampon à N cases.

Sémaphore solution à *1* case :
On utilise 2 sémaphores *plein* et *vide* initialisé à *0* et *1*.
plein indique si la case est pleine et *vide* si la case est vide

Producteur	Consommateur
Produire (Info); *P (vide)* Déposer (case, Info); *V (plein)*	*P (plein)* Retirer (case, Info); *V (vide)* Consommer (Info);

Solution à *n* cases :
Hypothèse : tampon à *n* cases avec une structure de file (FIFO : First In, First Out)



Il faut gérer le tampon. C'est-à-dire :

- si le tampon est vide, le consommateur ne peut rien retirer ;
- si le tampon est plein, le producteur ne peut rien déposer ;
- le tampon est circulaire, il faut empêcher que les indices tête et queue se chevauchent.

On utilise 2 sémaphores *plein* et *vide* initialisé à : *val(plein)= 0 et val(vide)= n*.

Les indices *tête* et *queue* sont initialisés à *0*. *Plein* indique le nombre de cases pleines et *Vide* le nombre de case vide.

Producteur	Consommateur
Produire_Information (Info) ; *P(vide)* ; tampon [*tête*] = Info ; *tête* = (*tête* + *1*) mod *n* ; *V(plein)* ;	*P(plein)* ; Info = tampon [queue] ; *queue* = (*queue* + *1*) mod *n* ; *V(vide)* ; Consommer (Info)

Dans ce qui suit nous allons présenter les appels systèmes permettant de manipuler les sémaphores sous UNIX en C.

Exercice : Ajouter un sémaphore Mutex pour traiter le problème de l'exclusion mutuelle.

6. Synchronisation avec Sémaphores

Processus 1

.
.
.

/* Traitement A1*/

.
.
.

Processus 2

.
.
.

/* Traitement A2*/

.
.
.

Deux Processus P1 et P2 s'exécutent en parallèle on veut réaliser la synchronisation suivante :

P2 ne peut exécuter son traitement A2 que si P1 a terminé l'exécution de son traitement A1

Solution : Synchronisation avec Sémaphore

- Bloquer le processus 2 avant d'entrer dans le traitement A2
- Réveiller le processus P2 après la fin de l'exécution du traitement A1 par le processus P1

Le processus1 et le processus 2 partagent un sémaphore S avec val(S) = 0

Processus 1	**Processus 2**
.	.
.	
.	P(S)
/* Traitement A1*/	/* Traitement A2*/
V(S)	.
.	.
.	.