I. Introduction

Les processus ne sont pas toujours indépendants, ils peuvent coopérer et nécessitent donc un moyen de **communication** et **synchronisation**. Parfois, ils se trouvent dans un état de compétition pour les ressources du système (fichiers ou mémoire commune). Les opérations sur ces ressources peuvent provoquer des **incohérences** ou des **interblocages** si l'accès à ces ressources n'est pas contrôlé.

Un interblocage se produit lorsque deux processus concurrents s'attendent mutuellement.
Exemple :

Processus A :	**Processus B :**
Réserve (R1)	Réserve (R2)
Réserve (R2)	Réserve (R1)
{opérations}	{opérations}

Tout se passe bien si Processus B s'exécute après la terminaison du processus A. Mais, considérons maintenant la séquence d'opérations suivante :

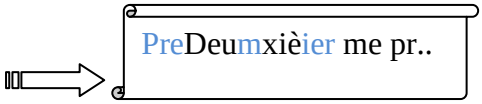
- Le processus A obtient R1.
- Le processus B obtient R2.
- Le processus A attend pour obtenir R2.
- Le processus B attend pour obtenir R1.

⇨ Dans cette situation, les deux processus sont indéfiniment bloqués : **Interblocage**.

Illustration du problème:

Exemple 1 : Imaginons 2 processus voulant imprimer en même temps sur une seule imprimante disponible.

print(« Premier processus »)	print(« Deuxième processus »)



P1 commence à imprimer puis perd le processeur, P2 obtient le processeur et commence à imprimer➔ Les caractères vont se mélanger.

→On est devant une situation d'utilisation d'une **ressource critique** pour laquelle l'accès doit être exprimé en prenant en compte **l'exclusion mutuelle** des processus demandeurs.

→La partie du code des processus utilisant cette ressource critique est dite **section critique**. Ici, « print » est une section critique.

Prenons un autre exemple de section critique ;

Exemple 2 : Mise à jour d'un compte bancaire via 2 processus créditeur et débiteur.

	Processus **Créditeur** (c : réel)		Processus **Débiteur** (d : réel)
	Variables t : réel		Début
	Début	(3)	Si (d > Solde) alors écrire (''découvert'')
(1)	t←Solde	(4)	t←Solde ;
(2)	Solde←t + c ;	(5)	Solde←t - d ;
	Fin		Fin

- ✓ Les instructions (1), (2), (3), (4) et (5) sont indivisibles.
- ✓ L'exécution parallèle de plusieurs processus créditeur ou débiteur est possible et provoque l'entrelacement des instructions.
- ✓ L'exécution parallèle n'est cohérente que si et seulement si le résultat de leur exécution donne le même résultat qu'une exécution séquentielle.

C'est-à-dire, si initialement Solde=10.

Créditeur(5)//débiteur(10)⇔Créditeur(5) puis Débiteur(10)⇔Débiteur(10) puis Créditeur(5)
➔Dans les 2 cas ; Solde=5.

a. Considérons l'exécution parallèle de 2 débits : *débiteur(9).* Initialement Solde=10.

→ Exécution séquentielle : - détection du découvert lors du 2ème débit.
- Impression du découvert
- Solde = -8.

→Exécution parallèle possible : $(3)^1$, $(4)^1$, $(3)^2$, $(4)^2$, $(5)^1$, $(5)^2$.

⇨ Solde = 1 et pas de détection du découvert : une chance pour le client !

☞ Les 2 processus débiteurs entrent en conflit d'accès à la **variable partagée** *Solde*.

b. Considérons l'exécution de 2 crédits : cr*éditeur(5)et créditeur(7)*
*a*vec un Solde initial =10 ➔ Solde =22.

Or, une exécution parallèle possible : $(1)^1$, $(1)^2$, $(2)^1$, $(2)^2$ ➔ Solde =17 !! ☹

→Problème d'accès concurrent à une variable partagée.

Donc, la section de code créditeur (débiteur) est **une section critique** ; au plus, un seul processus peut l'exécuter à la fois.

Pour éviter le conflit, il faut assurer que l'ensemble des opérations sur cette variable (consultation + mise à jour) soit exécuté de manière indivisible (= atomique = non interruptible).

Comment éviter le conflit ?

→*Contrôler l'entrée à la SC.*

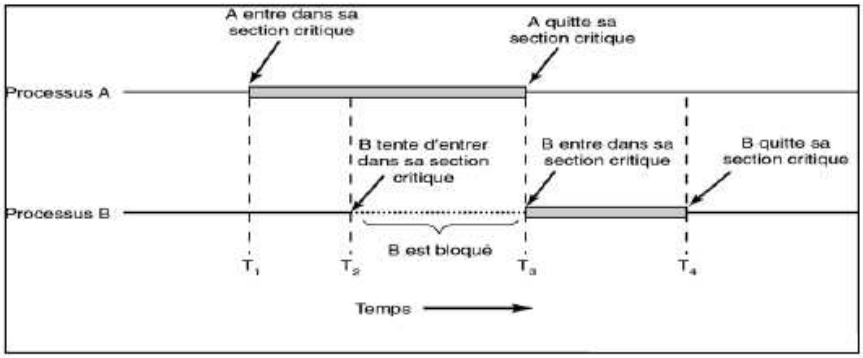
→*Garantir l'exclusion mutuelle dans la SC.*

Exemple3 : Spoule d'impression…..

II. Section critique

Définition : une section critique est une partie d'un programme dont l'exécution ne doit pas *entrelacer* avec d'autres programmes. Une fois qu'une tâche y entre, il faut lui permettre de terminer cette section sans permettre à d'autres tâches utiliser les mêmes données.

L'exécution d'une section critique implique **l'exclusion** de l'exécution de processus coopérants de leur section critique correspondante : **principe d'exclusion mutuelle.**



Pour mettre en œuvre correctement une SC, il faut s'assurer que les propriétés suivantes sont vérifiées :

- ✓ **L'accès exclusif ou l'unicité** : un et un seul processus est en section critique *(exclusion mutuelle).*
- ✓ **Avancement et absence d'interblocage**: un processus en dehors de sa section critique ne doit pas empêcher (=bloquer) un autre processus à entrer en section critique.
  Donc si un processus veut entrer dans la SC à répétition, et les autres ne sont pas intéressés, il doit pouvoir le faire.
- ✓ **Pas de famine** : aucun processus ne doit attendre indéfiniment pour pouvoir entrer dans sa SC *(les demandes doivent être traitées d'une façon équitable).*
  Si plusieurs processus sont en attente d'exécuter leurs SC alors qu'aucun processus n'est dans sa SC, l'un d'eux doit pouvoir y entrer au bout d'un temps fini.
- ✓ **Aucune hypothèse** ne doit être faite sur les vitesses relatives des processus.

➔ 4 conditions nécessaires pour réaliser une exclusion mutuelle.

Il est nécessaire de définir un protocole d'entrée en SC et un protocole de sortie de SC.

```
Algorithme Exclusion_mutuelle (SC)
  ….
  Entrée_SC ; // {inst} qui permet de vérifier si 1 processus n'est pas déjà en SC.
  SC ;
  Sortie_SC ; // {inst} qui permet à 1 processus ayant terminé sa SC d'avertir d'autres
                processus en attente de la libération de la voie.
  ….SNC
Fin
```

Il existe plusieurs méthodes de mise en œuvre de l'exclusion mutuelle dont chacune a ses avantages et inconvénients.

III. Solution matérielle

La commutation de processus se fait par des interruptions : interruption de l'horloge ou de n'importe quel autre périphérique. Ainsi, il est intéressant de **masquer** (désactiver) les **interruptions** quand un processus est entrain d'exécuter sa section critique : *plus d'interruption, donc plus de commutation de processus, donc garantir l'exclusion mutuelle. Le processus peut alors examiner et actualiser la mémoire partagée.*

```
Algorithme Exclusion_mutuelle (SC)
  ….
  Masquer_interruptions ;
  SC ;
  Autoriser_interruptions ;
  ….
Fin
```

Cette solution n'est valable que sur un système monoprocesseur.

En effet, la désactivation des interruptions n'affecte que le processeur en question. Les autres continuent d'exécuter des processus. Donc, il se peut qu'un processus accède à la mémoire partagée !

☞ Il n'est pas judicieux de donner aux processus utilisateur le pouvoir de désactiver les interruptions. Que se passera t-il s'il oublie de les réactiver ?! ➔ Fin du système ☹ Cependant, il est pratique pour le noyau qu'il désactive les Its pendant qu'il actualise des variables ou des listes.

La solution matérielle proposée peut être utilisée si la section critique est courte.

Etudions maintenant quelques solutions logicielles.

IV. Solutions logicielles

On peut envisager des solutions qui utilisent des variables globales. Les 3 premières solutions qui seront présentées ci-dessous sont fausses.

IV.1. Solution 1 : un verrou

Un verrou : une variable unique partagée de valeur initiale 0, par exemple : verrou du disque.

Principe de la méthode : Le processus avant d'entrer en SC teste si un autre est déjà entrain d'exécuter sa SC (SC est verrouillée). Si aucun processus ne se trouve en SC, il pose un verrou et entre en SC.

Si la porte est fermée, le processus reste dehors. Si elle est ouverte, il entre et ferme la porte

Verrou = { 0 : aucun processus ne se trouve en SC : (initialement)
1 : processus exécute sa SC.

Soient 2 processus A et B :

Processus A		Processus B
Tant que (verrou=1) faire	1	Tant que (verrou=1) faire
Ffaire	2	Ffaire
//Si (verrou==0)		//Si (verrou==0)
Verrou←1	3	Verrou←1
SC	4	SC
Verrou←0	5	Verrou←0

Soit la séquence d'exécution suivante: A1, A2, B1, B2, A3, A4, B3, B4 !

2 processus en SC en même temps ! ➔L'exclusion mutuelle n'est pas satisfaite !

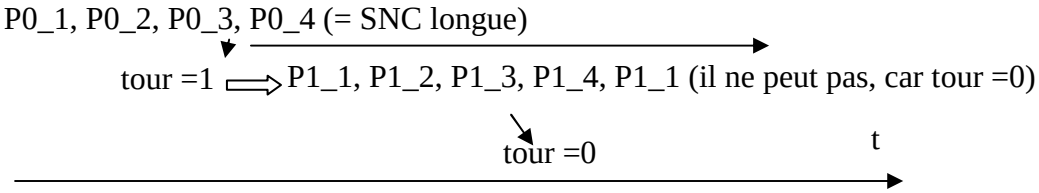
IV.2. Solution 2 : variable « tour »

Le processus ne teste plus si la SC est verrouillée ou non, mais si c'est bien son **tour** pour exécuter la SC : **P0 entre en SC si tour = 0 et P1 entre en SC si tour = 1**

Processus P0		**Processus P1**
Tant que (tour ≠0) faire ; SC tour ← 1 NSC	1 2 3 4	Tant que (tour ≠1) faire ; SC tour ← 0 //une fois terminé, passe le tour à l'autre NSC

Avec une telle solution, si P0 est en SC, P1 teste d'une façon continue la valeur de la variable *tour* jusqu'à ce qu'elle passe à 1. ➔Exclusion mutuelle satisfaite.

Avec cette méthode, on garanti l'exclusion mutuelle mais pas l'avancement des travaux. Prenons le cas suivant :



➔P1 n'est pas autorisé à exécuter sa SC (car tour = 0), alors que P0 n'est pas dans sa SC.

➔P1 est plus rapide que P0 : P1 est bloqué dans P1_1, jusqu'à ce que P0 définisse **tour** à 1.

C'est une alternance stricte !

IV.3. Solution 3 : 2 drapeaux (qui est intéressé)

Un processus P_i signale qu'il veut exécuter sa SC par **intéressé**[i]=vrai. Mais, il n'entre pas si l'autre processus est aussi intéressé.

Processus P0		**Processus P1**
intéressé [0]=vrai ; Tant que (intéressé[1]=vrai) faire ; SC ; intéressé[0] = faux ; NSC ;	1 2 3 4	intéressé [1]=vrai ; Tant que (intéressé[0]=vrai) faire ; SC ; intéressé[1] = faux ; NSC ;

→Exclusion mutuelle est garantie : si P0 est en SC, P1 attend et teste d'une façon continue la valeur de intéressé[0].

→Le critère « avancement des travaux » n'est pas satisfait. En effet, considérons la séquence suivante : P0_1, P1_1 ➔ intéressé[0]=vrai et intéressé[1]=vrai

➔ Les processus attendent indéfiniment pour exécuter leur SC !

➔On a un interblocage !

(chaque processus dit : « après vous » : excès de courtoisie !)

IV.4. Algorithme de Peterson

Il combine les principes des 2 derniers algorithmes précédents (tour et intéressé).

L'idée est : le processus se dit « je ne vais plus attendre que l'autre processus me donne mon tour, j'indique moi-même mon tour. Toutefois, je teste si l'autre n'est pas intéressé ». Là, ça devient question de rapidité ; qui a été le plus rapide à positionner le tour va passer.
Le processus se dit : c'est mon tour, mais je vérifie si l'autre est intéressé, j'attends.

Processus P0		**Processus P1**
intéressé [0]←vrai ; tour←0 ; Tant que (intéressé[1]=vrai && tour=0) faire Finfaire SC ; intéressé[0] ← faux ; NSC ;		intéressé [1]←vrai ; tour ←1 ; Tant que (intéressé[0]=vrai && tour=1) faire Finfaire SC ; intéressé[1] ←faux ; NSC ;

Preuve que cette solution est valide :

- **Exclusion mutuelle :** P0 et P1 peuvent être intéressés en même temps :
  - intéressé[0]=vrai et intéressé[1]=vrai
  - mais le plus rapide entre le premier, exemple : P0 entre bien que *tour*=1 puisque il a modifié *tour* avant P1.
- **Avancement :** P0 a terminé, P1 est dans sa SNC, P0 peut-il continuer ? intéressé[1]=faux, intéressé[0]=vrai, tour=0 ➔Oui, P0 peut exécuter SC puisque la condition est non vérifiée.
- **Pas de famine :** P0 et P1 veulent exécuter Sc, le plus rapide à poser le tour va passer. Les 2 vont exécuter leur SC. L'algorithme est équitable si l'ordonnanceur est équitable.
- On n'a pas posé des hypothèses sur les vitesses (ni sur les caractéristiques) des processus.

Remarque : ce test utilise l'attente active qui consomme beaucoup du temps processeur, puisque le processeur est immobilisé simplement pour attendre➔ on a un gaspillage de la puissance CPU disponible.

IV.5. Solution valide par verrou

Pour que la solution utilisant le verrou soit valide, il faut que les opérations de manipulation du verrou soient atomiques.

```
Algorithme Exclusion_mutuelle (SC)
  ….
  Verrouiller ( );
  SC ;
  Déverrouiller( ) ;
  ….
Fin
```

avec ; type **verrou** = enregistrement
val : entier
FA : LC
Fin

Procédure Verrouiller (var V : verrou) Si (V.val =0) //ouvert alors V.val←1 sinon Insérer (processus, V.FA) état (processus) ← bloqué Fsi Fin	Procédure Déverrouiller (var V : verrou) Si (V.FA ≠ nil) alors extraire (V.FA, processus) Etat(processus)=prêt sinon V.val←0 Fsi Fin

Les différentes solutions présentées ne permettent de gérer qu'une section critique, exemple : utilisation d'une seule ressource non partageable (1 seul exemplaire).

A l'exception du « verrou », les autres solutions présentaient l'inconvénient de l'attente active.

➔Le verrou = sémaphore binaire

= sémaphore d'exclusion mutuelle

Le sémaphore peut être généralisé afin de pouvoir gérer une ressource critique en N exemplaires.

V. Les sémaphores

V.1. Syntaxe et sémantique d'un sémaphore

Un sémaphore est une structure de données composée d' :

- un compteur,
- une file d'attente

```
Sémaphore : enregistrement
            Val : entier
            FA : LC
            Fin
```

Le sémaphore est initialisé comme suit :

```
Procédure Init(S : sémaphore)
Début
  S.val←n  //n : nombre d'autorisations d'accès disponibles
  S.FA ←nil
Fin
```

Initialement : -le compteur est égal à une valeur entière positive (=degré d'accès à la ressource critique),

- la FA est vide : aucun processus n'est en attente.

Nb : Si le sémaphore est initialisé à une valeur n>1, on obtient une solution qui permet à **n** processus d'être simultanément dans leur section critique.

Un sémaphore est manipulé par les deux actions *atomiques* suivantes :

- **P(s)** décrémente le compteur associé au sémaphore. Si sa valeur est négative, le processus appelant se bloque dans la file d'attente.
- **V(s)** incrémente le compteur. Si le compteur est négatif ou nul, un processus est choisi de la file d'attente et est réactivé.

Procédure **P** (S : sémaphore) Début S.val←S.val -1 //décrémentation Si (S.val < 0) alors insérer (processus, S.FA) état (processus) ← bloqué Fsi Fin	Procédure **V** (S : sémaphore) Début S.val←S.val +1 //incrémentation Si (S.val <= 0) alors extraire (processus, S.FA) état (processus) ← prêt Fsi Fin
P(s) correspond à une prise de ressource Puis-je accéder à la ressource	V(s) : une libération de ressource. Vas-y la ressource est disponible

Lorsqu'un processus effectue l'opération P(S) :

- ✓ si la valeur de **S** est supérieure à 0, il y a alors des ressources disponibles
  ➔ P(S) décrémente S.val et le processus poursuit son exécution,
- ✓ sinon ce processus sera mis dans une file d'attente jusqu'à la libération d'une ressource.

Lorsqu'un processus effectue l'opération V(S) :

- ✓ s'il n'y a pas de processus dans la file d'attente, V(S) incrémente la valeur de S,
- ✓ sinon un processus en attente est débloqué.

V.2. Utilisation des sémaphores

On peut utiliser les sémaphores pour protéger les sections critiques et ainsi garantir l'exclusion mutuelle et pour assurer la synchronisation conditionnelle des processus.

a. Exclusion mutuelle

Si on veut par exemple protéger l'accès à une ressource partagée (une variable, une imprimante...), on utilise un sémaphore d'exclusion mutuelle ;

```
Variable partagée mutex : sémaphore // mutex est initialisé à 1
Processus Pi
P(mutex)
  SC
V(mutex)
```

Question : Un des processus provoque un déroutement durant la section critique et s'arrête. Quelle anomalie provoque-t-il ainsi sur la suite de l'exécution des autres processus ?

Réponse : Puisque le processus dérouté n'a pas exécuté la sortie de la section critique par l'appel *V*(m*utex*), la section critique est considérée comme toujours occupée par un processus et par conséquent plus aucun autre processus ne pourra entrer en section critique.

Si **m** processus partagent **n** exemplaires d'une même ressource (m > n)

```
Sémaphore accès=n ;
Processus Pi
P(accès)
  SC //utiliser un des n exemplaires disponibles
V(accès)
```

⇨ Les n premiers appels ne sont pas bloquants

On peut assimiler la valeur positive du compteur au nombre de processus pouvant acquérir librement la ressource et assimiler la valeur négative du compteur au nombre de processus bloqués en attente d'utilisation de la ressource.

Un sémaphore est toujours initialisé à une valeur non-négative mais peut devenir négative après un certain nombre d'opérations P(S) ➔ nombre des processus en attente.

b. Synchronisation des processus

Il se peut qu'un processus doive attendre un autre pour pouvoir continuer son exécution, exemple : P0 doit attendre P1 (P1 puis P0)

Sémaphore **syn** = 0 //initialisé à 0 pour pouvoir bloquer P0	
P_0 P(syn) …	P_1 … V(syn)

Un **sémaphore** est un mécanisme de synchronisation qui fonctionne comme une barrière bloquante dans certaines conditions.

Conclusion : les sémaphores sont des mécanismes de bas niveau, généralement utilisés à l'intérieur des noyaux de systèmes. Pour des séquences brèves d'exclusion mutuelle, les multiprocesseurs peuvent en outre utiliser l'attente active.

En général, toutes les solutions se basent sur l'existence d'instructions atomiques, qui fonctionnent comme SCs de base

VI. Problèmes classiques de synchronisation

VI.1. Rendez-vous

On considère 2 processus, chacun étant constitué de deux phases de calcul AVANT et APRES. Le problème consiste à garantir qu'aucun des processus ne commence sa phase de calcul APRES avant que **l'autre processus** ait terminé leur phase de calcul AVANT. Les 2 processus ont donc un rendez-vous à la fin de leur phase de calcul AVANT.

Sémaphores arrivée1=0, arrivée2=0 ;

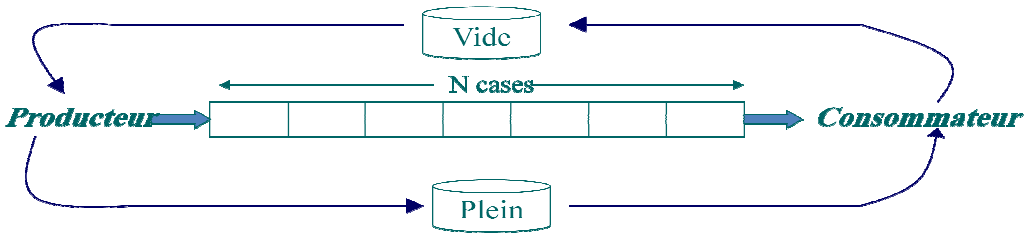
Processus P1		**Processus P2**
{Avant}		{Avant}
V(arrivée1)	//signaler mon arrivée	V(arrivée2)
P(arrivée2)	//attendre l'arrivée de l'autre	P(arrivée1)
{Après}		{Après}

Généralisation du problème de RDV

Soient N processus parallèles ayant un point de rendez-vous. Un processus arrivant au point devrendez-vous se met en attente s'il existe au moins un autre processus qui n'y est pas arrivé. Le dernier arrivé réveillera les processus bloqués. La solution suivante résoud ce problème en utilisant des sémaphores.

```
Semaphore S=0
Nb_arrivée=0 ;
Procédure RDV
Début
P(mutex)
    Nb_arrivée←nb_arrivée+1
    Si (nb_arrivée<N)          //non tous arrivés
        alors V(mutex)         //on libère mutex et
              P(s)             //on se bloque
   Sinon
          V(mutex)                 //le dernier arrivée libère mutex et
          Pour i de 1 à n-1 faire
            V(s)                   //réveille les n-1 bloqués dans l'ordre d'arrivée
          Ffaire
    Fsi
Fin
```

VI.2. Producteur/consommateur (tampon borné)



Contraintes de synchronisation :

- Relation de précédence : Producteur < Consommateur
- Section critique (tampon)
  - tampon plein ⇨ Producteur se bloque
  - tampon vide ⇨ Consommateur se bloque
  - Exclusion mutuelle au tampon

Variables partagées :

#define N 100

Semaphore mutex=1; /* protège l'accès au tampon */

Semaphore nb_plein=0; /* compte le nombre d 'informations produites dans le tampon */

Semaphore nb_vide = N; /* Nb d 'emplacements libres dans le tampon */

Processus Producteur	**Processus Consommateur**
P(nb_vide) P(mutex) Insérer_élément ( ) V(mutex) **V(nb_plein)**	**P(nb_plein)** P(mutex) Retirer_élément( ) V(mutex) **V(nb_vide)**

✍Une mémoire tampon ou un buffer est une zone de mémoire vive ou de disque utilisée pour stocker temporairement des données, notamment entre deux processus ou matériels ne travaillant pas au même rythme.

VI.3. Lecteurs/rédacteurs

Considérons ce problème comme étant un système de réservation de billets d'avions où plusieurs processus tentent de lire et d'écrire des informations:

- On accepte que plusieurs lecteurs utilisent le système➔degré d'accès>=1
- On n'autorise qu'un seul rédacteur à la fois ➔on exclut les lecteurs et les rédacteurs : degré d'accès=1➔exclusion mutuelle → Sémaphore : nb_accès.
- Un rédacteur bloqué doit attendre le dernier des lecteurs pour qu'il puisse entrer en section critique.

Variables partagées :

Sémaphore mutex=1; */* protège le compteur des lecteurs */*
Sémaphore nb_accès=1; */* exclusion mutuelle pour les rédacteurs */*
int nb_lect=0; */* Nombre de lecteurs actifs */*

Lecteur	**Rédacteur**
P(mutex) nb_lect++ Si (nb_lect==1) Alors P(nb_accès) Fsi V(mutex) **Lire** P(mutex) nb_lect - - Si (nb_lect==0) Alors V(nb_accès) Fsi V(mutex)	P(nb_accès) **Ecrire** V(nb_accès)