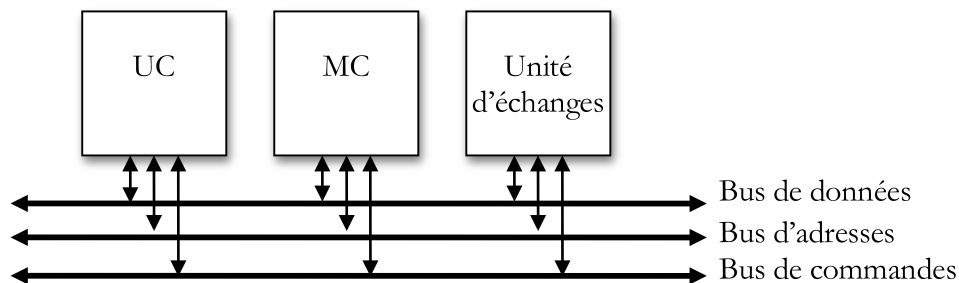
Les Processus

Notions de Coopération, de Compétition et de Parallélisme

I. Rappels

1. L'architecture de Von Neumann

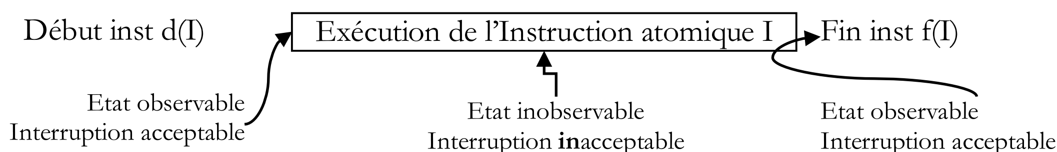


- Programme : une suite ordonnée d'instructions. L'instruction est l'élément indivisible du programme constituant la plus petite unité atomique du langage.
- Processeur : c'est un dispositif physique ayant trois caractéristiques :
  - Il peut prendre un ensemble pratiquement fini d'états distincts.
  - Il communique avec un environnement caractérisé lui aussi par son état.
  - Il peut modifier son propre état et celui de son environnement en exécutant un programme.

L'état du processeur est caractérisé par :

- Le registre d'état de l'UAL.
- Le contenu des registres programmables et Les registres internes

- Point Observable (Interruptible) : l'état du processeur ne peut être observable qu'au début ou à la fin de l'exécution d'une instruction. Autrement dit, le processeur ne peut être arrêté qu'après avoir exécuté complètement l'instruction courante (principe d'une instruction **atomique**). Par conséquent, aucune interruption n'est tolérée par le processeur au milieu d'une instruction.



2. Ressources

On appelle une ressource tout élément qui contribue à la progression des process. Certains relève du matériels : le CPU, la mémoire, les périphs d'E/S, … d'autres logicielles : des procédures, des applications, des processus systèmes, …

➔ Un process utilise une ressource suivant l'algorithme suivant :

```
Demander( type_ressource, qté_demandée) ;
  <Utiliser(ressource) >;
Libérer(type_ressource, qté_allouée) ;
```

➔ Selon le degré de partageabilité d'une ressource on distingue des ressources **N_ partageable** (mémoire, lecture disque, …) et d'autres **critiques** impartageable (le processeur, imprimante, fichier en écriture, variable globale, procédure non réentrante, ..)

➔ On distingue aussi des ressources **banalisée**, existant en plusieurs copies identiques (espace mémoire ou espace disque, …), et d'autres **non banalisées** (un fichier nommé, terminal, …)

3. Processus

Définition : c'est un programme en cours d'exécution auquel on associe :

⇒ ***Un Contexte du processeur (CPU) : l'ensemble des registres (CO, SW, SP, Rx, ...)***

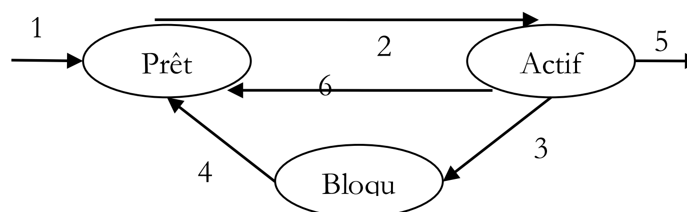
⇒ ***Un Contexte de la Mémoire Centrale : segments de code, segments de données, ...***

Un processus est dit **séquentiel** si l'exécution de l'ensemble de ses instructions s'établie dans un ordre strict et bien déterminé.

3.1. Etats d'un processus

L'aspect dynamique que présente un process lors de son exécution n'implique pas uniquement l'évolution de son contexte, il s'étend aux différents états d'exécution qu'un process connaît dans SE multiprogrammé. En fait, dans de tel environnement, un process peut :

- Obtenir le processeur et s'exécuter, on dit qu'il est ***actif*** ou **élu**. Le processus alors dispose de toutes les ressources nécessaires à son exécution.
- Demander une ressource et il ne peut pas poursuivre son exécution tant qu'il ne l'a pas obtenue. Donc le processeur **bloque** le process qui doit attendre la disponibilité de cette ressource. On dit qu'il est en état ***bloqué*** ou ***endormi***
- Lorsqu'un process vient d'être chargé dans la MC, ou il vient d'obtenir une ressource qu'il a demandée, et il trouve que le processeur est alloué à un autre process, il passe à l'état ***prêt*** , on dit qu'il est en ***attente du CPU***



3.2. Représentation interne des processus

➔ Au moment du chargement d'un programme exécutable, le SE lui crée une structure représentant le process associé. Cette structure devrait contenir toutes les informations nécessaires permettant l'évolution dynamique du process au sein du SE, entre autre son contexte, son état, …cette structure est appelée le ***Bloc de Contrôle du Process (PCB).***

PCB
ID process, IDpropriétaire
état du process
Contexte processeur
Contexte mémoire
Informations ressources (fichiers ouverts, E/S,..
Infos : Pages, Segments,
infos sur les fils
…

➔ Pour manipuler tous les process, le SE détient d'une **table de process** dont chaque entrée contient un pointeur vers un PCB d'un processus.

3.3. Opérations sur les processus

L'ensemble minimum d'opérations exercées par le SE sur les process comprend :

* Création – Destruction

* Blocage – réveil

* Suspension – Reprise

* Modification de la priorité

➔ Certaines opérations sont **invocables** par le processus lui-même (création, destruction, …) d'autres sont réservées au SE (scheduler, dispatcher, chargeur, …)

3.4. Notion de tâche (thread)

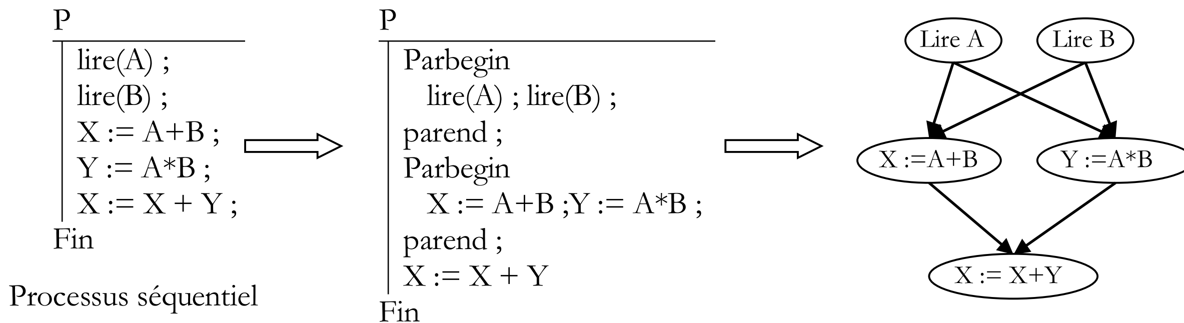
Lorsqu'un processus présente un code un petit peu long ou compliqué, il peut être décortiqué en un ensemble d'unités de traitements élémentaires ayant une cohérence logique dont la fonction est bien déterminée. Cette unité est appelée une **tâche.**

Ainsi un process séquentiel de N tâches P peut être écrit comme suit : $P = T_1\ T_2\ T_3 \dots T_N$

➔ Certaines tâches doivent s'exécuter en série d'autres peuvent s'exécuter en **parallèle**.

➔ On peut représenter l'ensemble de tâches constituant un processus par un graphe de précédence

➔ On résolvant un problème, on peut transformer un processus séquentiel (une série de tâches séquentielles) en un système équivalent où certaines tâches peuvent s'exécuter en parallèle. Ainsi on augmente le degré de multiprogrammation.



4. Interactions de processus

En s'exécutant en parallèle, les processus ne s'exécutent pas tous de manière isolée ; ils peuvent agir les un sur les autres. Certains se coopèrent, d'autres rentrent en conflit entre eux et bien d'autres sont totalement indépendants.

4.1. Processus indépendants

Un processus est dit indépendant s'il ne peut pas affecter ni être affecté par le déroulement d'un autre processus. Autrement dit, si l'ensemble de données qu'il manipule (statique ou automatiques) et l'ensemble de ressources qu'il détient sont privés et ne sont pas partagés avec d'autres process.

Exemple **:** deux processus s'exécutant sur deux processeurs indépendants

4.2. Processus concurrents (compétitifs)

Les ressources que dispose un SIQ sont généralement limitées en nombre et en quantités et d'autres sont critiques. L'exécution des processus implique la consommation de telles ressources qui peuvent être sollicitées simultanément par plusieurs process. Ainsi, la coopération conduit à une relation de conflit et de compétition pour l'acquisition de ressources.

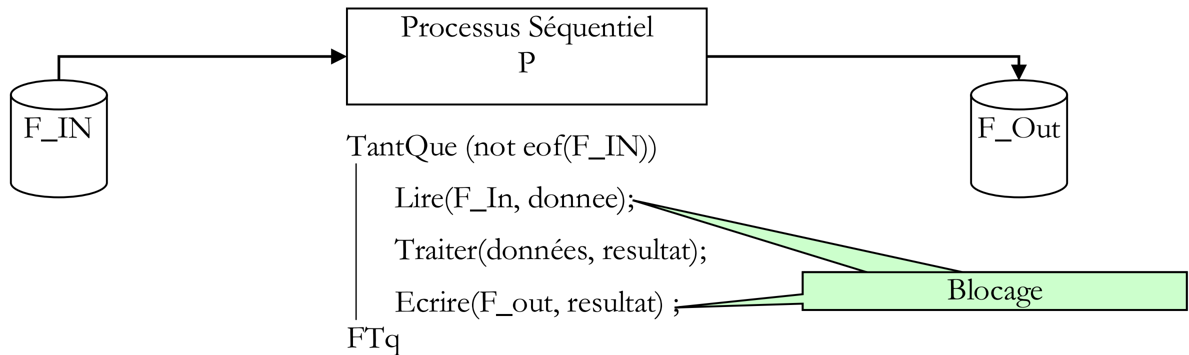
4.3. Processus coopérants

Un processus est dit coopérant, avec d'autres, s'il peut affecter ou être affecté par le déroulement d'un autre process. Ainsi un ensemble de données et ressources manipulés par ce process est partagé avec d'autres comme moyen d'interaction ou de communication (variable globales, fichiers, messages, …).
Plusieurs raisons favorisent la coopération entre processus :

- Partage de données : plusieurs utilisateurs peuvent s'être intéressés à la même information (serveur de fichiers, base de données, site web,…) ou ressources. le SE doit fournir un moyen de partage **efficace** de ces données.
- Accélération de l'exécution (calcul) : des processus se coopèrent pour la contribution à l'exécution d'une tâche globale en découpant le processus global en plusieurs processus fils (ou tâches), chacun a sa propre tâche, qui peuvent s'exécuter en parallèle.
- Modularité

Exemple **:**

Soit un processus qui lit des données à partir d'un fichier F_In, les traite puis les sauvegarde dans un autre fichier F_Out.



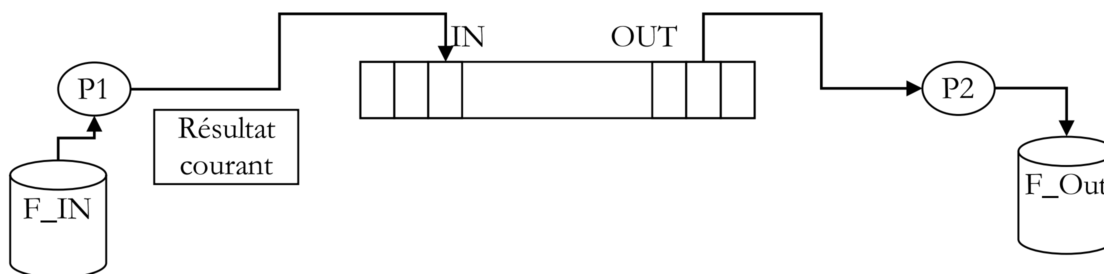
Le processus P peut être découpé en deux processus :

P1 (dépôt): lecture et traitement puis dépôt de données.

P2 (retrait) : Lire le résultat courant et l'écrire dans un fichier de sortie.

Pour que les deux process puissent se communiquer, il leur faut un objet intermédiaire entre eux qui sert d'un outil de transmission de données.

Ce moyen peut se réaliser en utilisant une variable globale consistant en un tableau, BUF, de N entrées.



Variable globale : BUF : array[0..N-1] d'Element

P1

```
Var IN : integer
TantQue (not eof(F_IN))
    Lire(F_In, donnee);
    Traiter(données, resultat);
    Buf[IN] := resultat ;
    IN := IN+1 Mod N ;
FTq
```

P2

```
Var OUT : integer
TantQue (il y a message)
    Mess := BUF[OUT] ;
    OUT := OUT +1  Mod N ;
    Ecrire(F_out, resultat) ;
FTq
```

☹ **Problèmes** :

- Possibilité d'écrasement de données si le rythme des dépôts est plus rapide que celui des retraits.
- Possibilité de lire un contenu non valide si le rythme des retraits est plus rapide que celui des dépôts.

➔ Les deux process doivent se synchroniser !

II. Problématiques

1. Le problème de la Section Critique & L'exclusion Mutuelle

Position du problème : soit deux processus qui s'exécutent en parallèle pour mettre à jour un compte bancaire.

P1 : Process Crédit

lire(compte) ;	(a)
compte := compte + 1000 ;	(b)
réécrire (compte)	(c)

Fin

P2 : Process Débit

lire(compte) ;	(a')
compte := compte - 500 ;	(b')
réécrire (compte)	(c')

Fin

Supposons que *compte* contient initialement 3000.

1er cas : exécution séquentielle : le compte a une valeur cohérente

- exécution suivant (P1 puis P2) ou (P2 puis P1) : le résultat est *compte = 3000 + 1000-500 = 3500.*

2ème cas : exécution concurrentielle (imbriquée): le compte pourrait avoir une valeur non cohérente

- Soit la suite d'exécution suivante : *a, a', b, b', c, c'* ; la valeur finale de compte est : *2500* ! → erreur !

➔ L'incohérence est due en fait qu'on a autorisé l'accès simultané à la variable partagée *compte* pour que chaque process en fasse une copie chez lui. La dernière mise à jour de P2 a annulé la première mise à jour de P1 qui n'a pas été signalée chez P2 !

1.1. Définition de la Section critique

On appelle **section critique** la partie d'un programme où la ressource partagée (variable, fichier, etc…) est seulement accessible par un seul processus à un moment donné. Autrement dit, l'accès simultané à la section critique sur une même ressource critique est interdit. On dit que la ressource est en accès *exclusif* et les processus sont en exclusion mutuelle (EM).

- On étend cette définition à une ressource N_partageable où le nombre de processus autorisés à entrer en section critique est au maximum N.

2. Problème de la Privation (Starvation)

La situation où quelques processus progressent normalement en bloquant indéfiniment d'autres processus.

Exemple **:** Processus dont la vitesse d'exécution est rapide, processus privilégié,

2.1. Problème de l'Interblocage (deadlock)

La situation où un ensemble de processus sont bloqués indéfiniment. Chacun en attente de ressources pour sa progression détenue par un autre process dans l'ensemble.

Exemple **:** le process P1 déteint la ressource R1 et attend la ressource R2 détenue par P2 qui attend la ressource R1.

III. Eléments Généraux sur les outils de Coopération : Réalisation de l'EM

La solution du problème de l'EM est d'imposer aux processus de suivre un protocole d'utilisation de ressources critiques bien déterminé :

- Toute manipulation d'une ressource critique doit se faire dans une section critique <SC>.
- Tout processus désirant l'accès à la <SC> doit demander une permission d'accès sous le modèle suivant :

Demande d'accès à la SC
< Accès à la ressource critique RC>
Sortie de la SC : libérer la RC

- Ainsi, si un process P demande l'accès à la SC et si celle-ci est libre, le protocole l'autorise d'y accéder. A ce moment, si un process Q demandant l'accès à la SC alors que P1 n'en a pas encore terminé, le protocole le bloquera jusqu'à sa libération par P1

* Hypothèses de travail : toute solution apportée au problème de l'EM doit prendre en compte les points suivants :

i) Aucune supposition n'est faite sur les vitesses relatives d'exécution des process. Autrement dit, la solution ne doit pas dépendre de la vitesse du processeur.
ii) La durée de la section critique est finie.
iii) L'ordre d'accès à la section critique n'est pas imposé.

* Propriétés de Dijkstra : toute solution apportée au problème de l'EM doit respecter les quatre propriétés suivantes :

1- *Exclusion Mutuelle* : si un process se trouve en sa section critique, aucun autre process ne peut se trouver dans la sienne. ➔ A tout instant : nb_process_dans_SC ≤ 1.

2- *Absence d'interblocage* : si la RC est libre et que plusieurs processus la demandent alors l'un d'eux doit l'obtenir au bout d'un temps fini.

3- *Progression* : le blocage d'un process hors de sa SC ne doit pas empêcher les autres process d'entrer en SC.

4- *Absence de processus privilégiés* : la solution doit être la même pour tous les process.

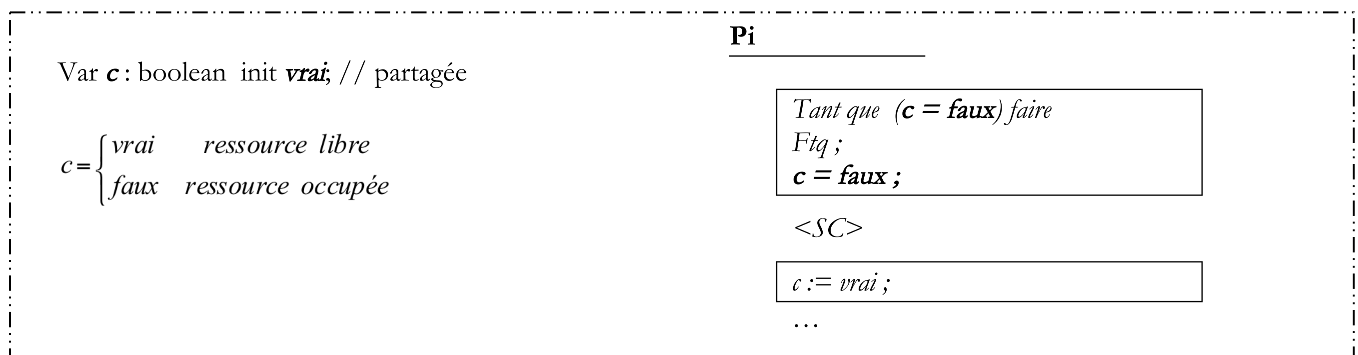
1. Solution par attente active

Lorsqu'un processus est dans sa section critique, les autres process exécutent, d'une manière continue, une boucle qui tente l'accès à la section critique.

1.1. Solution de l'EM entre deux processus

1ére solution : On utilise une variable booléenne partagée *c* indiquant la disponibilité de la ressource critique.

La structure de chaque processus $P_{i=1,2}$ est la suivante :



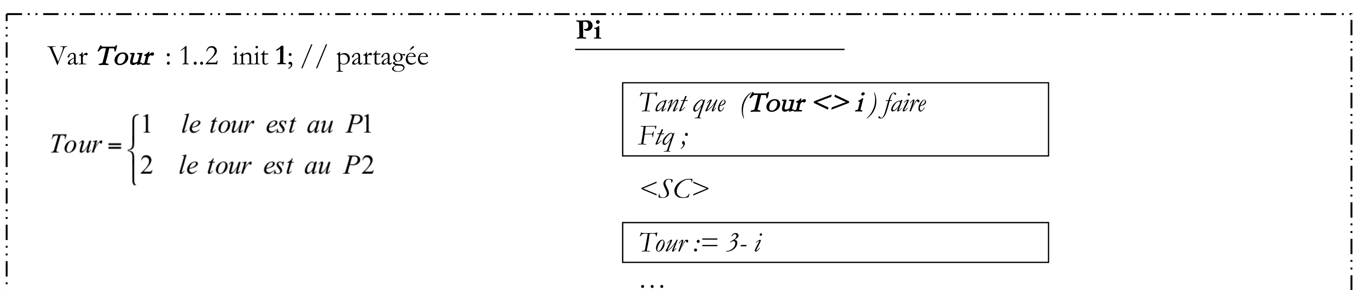
☹ La première propriété de Dijkstra (propriété de l'EM) n'est pas respectée. En fait le bloc de test sur **C = vrai** est traduit en assembleur par le code suivant :

```
Tq :    Mov    reg, C    -I1-
        Cmp    reg, 0    -I2-
        Jg     Tq        -I3-
```

Si P2 s'exécute plus rapidement que P1 : P1 exécute I1 alors que P2 exécute I1 et I2 ➔ les deux process se trouvent simultanément dans la SC !

☹ La variable partagée C est elle-même une variable critique nécessitant la protection

➔ Etablir un ordre d'accès entre les deux process



2ème solution : Pour établir un ordre d'accès, on utilise une variable entière partagée *Tour* indiquant à qui le tour d'accès à la section critique

☺ La première propriété de Dijkstra (propriété de l'EM) est respectée.

- Preuve : Démonstration par l'absurde

Supposons que l'EM n'est pas respecté, alors les deux process ont réussi de terminer leurs boucles de test. Ceci veut dire :

(P1 dans sa SC ==> Tour = 1) **et (**P2 dans sa SC ==> Tour = 2)

Contradiction : Tour ne peut pas prendre deux valeurs différentes en même temps.

☹ La Troisième propriété de Dijkstra (propriété de progression) n'est pas respectée.

- Preuve : si P2 s'exécute puis sort, donnant la main ainsi à P1 puis il se bloque. P1 à son tour entre à la SC puit sort donnant la main à P2 qui est bloqué. Si P1 veut entrer une deuxième fois il ne pourra pas faire alors que la SC est libre !!

☹ Le précédent problème est du au fait que Cette solution impose une alternance stricte sur l'ordre des exécutions : P1, P2, P1, P2, P1,…. Si la chaîne est brisée à un endroit toute la suite sera condamnée !

➔ Le problème de cette solution est qu'elle ne mémorise pas des informations suffisantes concernant l'état de chaque process, elle retient uniquement le process qui est **autorisé** d'accéder à la SC.

3ème solution : pour mémoriser l'état de chaque process on utilise une variable partagée ***Désire*** indiquant que le process désire entrer à la section critique ou il est déjà dedans.

Var ***desire*** : Array [1..2] of Boolean; init [**false, false**]

$$desire[i] = \begin{cases} vrai & Pi \text{ veut entrer à la SC ou il est déjà dedans} \\ faux & Pi \text{ est hors de sa SC et il n'a pas l'envie d'y accéder} \end{cases}$$

Pi

```
Désire[i] := true                    // Pi veut entrer
Tant que  (desire[3-i]) faire
Ftq ;

<SC>

Desire[i] := faux ;
…
```

☺ La première propriété de Dijkstra (propriété de l'EM) est respectée.

- Preuve : Démonstration par l'absurde

Supposons que l'EM n'est pas respecté, alors les deux process ont réussi de terminer leurs boucles de test. Ceci veut dire :

P1 dans sa SC ==> *(Désire [1] = true* **et** *Désire [2] = false)*

et

P2 dans sa SC ==> *(Désire [1] = false* **et** *Désire [2] = true)*

Contradiction : *Désire[i]* ne peut pas prendre deux valeurs différentes en même temps.

☹ La deuxième propriété de Dijkstra (propriété de l'EM) n'est respectée.

- <u>Preuve :</u> prenons le cas où les deux process positionnent en même temps leurs *Desire* à *true*, alors on aura *(Desire [1] = true)* *et (Desire[2] = true)*

Les deux process rentrent en un blocage infini ➔ ils s'interbloquent infiniment!!!

4ème solution : Solution de DEKKER

La solution de Dekker est une combinaison entre les deux solutions précédentes.

Var **désire** : Array [1..2] of boolean init [false,false]
Tour : 1..2 init 1 (ou 2)

$$desire[i] = \begin{cases} vrai & Pi \text{ veut entrer à la SC ou il est déjà dedans} \\ faux & Pi \text{ est hors de sa SC et il n'a pas l'envie d'y accéder} \end{cases}$$

$$Tour = \begin{cases} 1 & \text{le tour est au } P1 \\ 2 & \text{le tour est au } P2 \end{cases}$$

Pi

```
Désire[i] := true                        // Pi veut entrer
Si (Desire[3-i] ) alors                  //Pi suppose que l'autre process veut rentrer ou il est dans la SC
    Si (tour = 3-i) alors                // Situation de conflit : Est-ce que P3-i à le tour d'y accéder
        Désire[i] := false ;             // oui, Pi renonce
        Tant que (tour <> i) faire       // Pi attend son tour
        FTq
        Désire[i] := true ;              // Pi reprend sa décision d'accéder
    FSi
    Tant que Desire[3-i] faire           // P3-i est il sorti de sa SC ? ou désire t- il encore d'y accéder ?
    FTq
Fsi
<SC>
Tour := 3-i ;
Desire[i] := faux ;
…
```

☺ Cette solution respecte les quatre conditions de Dijkstra.

- <u>Preuve de la première propriété : l'EM</u>

Supposons que l'EM n'est pas respecté et les deux process sont simultanément dans leurs SC. Ceci veut dire qu'ils ont réussi de franchir les deux boucles avec succès:

P1 dans sa SC ==> *(Désire [2] = false)* **et** **(***Désire [1] = true) // tour quelconque*

<u>et</u>

P2 dans sa SC ==> **(***Désire [1] = false)* **et (***Désire [2] = true) // tour quelconque*

Contradiction : dans les quatre cas possibles. ➔ L'EM est respectée

- <u>Preuve de la deuxième propriété : Absence d'interblocage</u>

Supposons qu'il y ait un interblocage entre P1 et P2 à l'entrée de la SC alors que la ressource est libre. Ceci veut dire que chacun des deux process est bloqué au niveau de l'une des deux boucles : « *Tant que Tour <>i* » ou « *Tant que Desire[3-i]* » :

- (P1 Bloqué) ==> $\begin{cases} Tour \neq 1 \\ ou \\ Desire[2] = true \end{cases}$ **et** (P2 Bloqué) ==> $\begin{cases} Tour \neq 2 \\ ou \\ Desire[1] = true \end{cases}$

Quatre cas sont possibles résumés dans le tableau suivant :

ET		P1 bloqué sur	
		While Tour <> 1	*While Désire [2] = true*
P2 bloqué sur	*While Tour <> 2*	**Contradiction** : tour ne peut prendre que deux valeurs 1 ou 2	**Contradiction** : si P2 est bloqué sur Tour <> 2 alors sûrement il a exécuté l'instruction *Desire[2] := false*
	While Désire [1] = true	**Contradiction** : si P1 est bloqué sur Tour <> 1 alors sûrement il a exécuté l'instruction *Desire[1] = false*	Donc les deux process ont pu passer le test sur tour (ou éventuellement la boucle sur tour) donc : (Tour = 1 pour P1) et (Tour =2 pour P2) ➔ **Contradiction**

- <u>Preuve de la troisième propriété :</u> supposons que P2 s'est bloqué hors de sa SC, après avoir exécuté les deux dernières instructions de libération de la SC, alors que P1 est bloqué à l'entrée de la SC.

P1 bloqué ➔ $\begin{cases} Tour \neq 1 \\ ou \\ Desire[2] = true \end{cases}$ ET (P2 est sorti de sa SC) ➔ $\begin{cases} Tour = 1 \\ ou \\ Desire[2] = false \end{cases}$

Contradiction : P1 peut accéder à la SC dans un temps fini ➔la propriété de progression est satisfaite

- <u>Preuve de la troisième propriété :</u> évidente

Remarque : La solution de Dekker respecte les propriétés de Dijkstra mais engendre une Privation (starvation)!

1.2. Solution de l'EM entre plusieurs processus

<u>5ème solution :</u> Solution de Peterson

La solution de Peterson est une variante de Dekker légère et plus lisible, elle est extensible à N processus.

Var *desire* : Array [1..2] of Boolean init [false, false]
Tour : 1..2 init 1 (ou 2)

Desire : idem que pour la solution de DEKKER

Tour : en cas de conflit, tour désigne le numéro de process qui doit <u>prochainement attendre</u>

<u>Pi</u>

```
Désire[i] := true
Tour := j
Tant que (Desire [3-i] et Tour = 3-i ) faire
FTQ
<SC>
Desire[i] := faux ;
…
```

Exercice : prouver que l'algorithme de Peterson respecte les propriétés de Dijkstra.

6ème solution : Solution en utilisant une instruction spéciale

Reprenons l'algorithme de la première solution, le problème résidait dans le fait qu'un process teste une variable au moment où un autre est entrain de la modifier. Si ce process arrive à exécuter le test et la modification d'une manière indivisible sans qu'un autre process ne puisse l'interrompre le problème sera résolu.

Exemple: l'instruction « **Test And Set »** TAS **(P)**

Var P : 0..1 init 0 // P = 0 : Ressource critique libre ; P =1 : Ressource critique Occupée

TAS (P)

```
- bloquer l'accès à la variable P
Si P = 0 alors
        P = 1 ;
        CO := CO +2 -------> (1)
Sinon
CO := CO + 1 ---------> (2)
Fsi
Libérer l'accès à la variable P
```

- L'ensemble de ses instructions (microinstrutions) doivent s'exécuter d'une manière indivisible
- Le premier saut du Compteur ordinal pointe vers la première instruction de la SC.
- Le deuxième saut repointe sur la boucle du test

Ainsi une solution à l'EM peut être donnée comme suit :

Var P : 0..1 init 0 // Variable partagée P = 0 : Ressource critique libre ; P =1 : Ressource critique Occupée

Pi

```
Boucle : TAS(P)
            Goto boucle
        < SC>
        P := 0 ;
```

L'ensemble de ses instructions (microinstrutions) doivent s'exécuter d'une manière indivisible

- De la même façon que précédemment, on démontre que cette solution respecte les propriétés de Dijkstra.
- La variable P est aussi une variable critique, l'EM sur P est assurée par câblage (physiquement).

1.3. Critiques des l'approche par attente active

- Mise au point et généralisation pour plusieurs process est difficile
- La boucle d'attente active est une perte de temps du processeur et peut nuire aux performances de la multiprogrammation

2. Solution par attente Passive

Le principe est d'éviter la perte de temps engendrée par la boucle d'attente active. Si la condition d'accès à la SC est fausse pour un processus quelconque, il doit être bloqué. L'attente passive utilise deux outils : les verrous et les sémaphores

2.1. Les verrous (Lock)

Un verrou *P* est une variable structurée en un couple consistant en une variable booléenne « *p* » indiquant la disponibilité de la ressource critique et une file d'attente « *f(p)* » contenant les PCB des processus bloqués sur l'entrée de la SC.

```
Type   Verrou  =     Record
                          p      : boolean ;
                          f(p) : File de PCB ;
                     End ;
```

- À l'état initial : - la ressource RC est libre : *p init true ;*
  - Aucun process n'est en attente : *f(p) init Nil.*

- Deux procédures <u>indivisibles</u> (Primitives) sont associées à chaque variable P de type verrou : Lock(P) et Unlock(P)

```
Var P : Verrou   init [true, nil] //

Lock(P) // Verrouiller (p)
    Si (p = true)
        P = false ; // RC occupée
    Sinon
        // bloquer le process appelant
        - Process.Etat := BLOQUE ;
        - Enfiler (f(p), Proces.PCB)      // Insérer le PCB du process appelant dans la file « f(p ) »
        - Appeler Scheduler()             // Choisir un autre process et faire une commutation de contexte
    FSi

UnLock(P) // Déverrouiller (p)
    Si (non filevide(f(p)) )
        // débloquer un process de la file
        - Choisir un processus de f(p)     // en appliquant une stratégie de sélection Fifo, par priorité,
        - PCB_Choisi := défiler(f(p))      // Retirer son PCB de la file ;
      - PCB_Choisi.Etat := PRET ;          // Selon la stratégie du scheduler :Insérer le PCB du process choisi dans la file
                                           // des prêts ou éventuellement le mettre directement à l'état ACTIF
    Sinon
        P := true ;                       // La Ressource critique est libre
    FSi
```

2.2. Application des verrous pour résoudre le problème de l'EM

Une solution de l'EM consiste à ce que tous les process, au nombre de N, partagent une variable P de type verrou, l'accès de chaque process Pi à la section critique est soumis au modèle suivant

Demande d'accès ============> **Lock (p)** // *Accéder, éventuellement, et bloquer l'accès*

< Accès à la SC >

Libérer l'accès ============> **UnLock (p)** // *Libérer l'accès et éventuellement réveiller un processus*

- On démontre facilement que cette solution respecte les conditions de Dijkstra
- La variable P {donc la structure p et f(p)}est une variable globale partagée en mise à jour, par conséquent elle doit être protégée contre les accès simultanés :

  - Les seules actions autorisées sur P sont Lock et UnLock. Elles doivent être indivisibles pour protéger le verrou contre les accès simultanés.

  - les deux actions Lock(P) et UnLock(P) doivent s'exécuter en exclusion (ne peuvent pas s'exécuter sur P en même temps)

2.3. Les Sémaphores de Dijkstra

Un Sémaphore *S* est une généralisation de la notion de verrou, il consiste en une structure contenant une variable entière « *e(s)*» et une file d'attente «*f(s)* » contenant les PCB des processus bloqués sur l'entrée de la SC.

```
Type   Semaphore   =   Record
                              e(s)    : entier ;
                              f(s) : File de PCB ;
                        End ;
```

- *e(s)* est initialisé à une valeur non négative indiquant le nombre de process pouvant simultanément accéder à la section critique, ce nombre indique aussi le degré de partageabilité de la RC.
- Deux fonctions indivisibles (Primitives) sont associées à chaque variable S de type sémaphore :

    - P(S) : Puis-je passer par S =La barrière du sémaphore permettant le franchissement de S.

    - V(S) : Vas y = l'autorisation (ou laissez-passer) de S de déblocage.

```
Var S : Semaphore   init [e0, nil] // e0: valeur entière positive

P(S)
    e(s) := e(s) -1
    Si ( e(s) < 0)   //
        // fermer la barrière et bloquer le process appelant
        - Process.Etat := BLOQUE ;
        - Enfiler (f(S), Proces.PCB)        // Insérer le PCB du process appelant dans la file « f(s ) »
        - Appeler Scheduler()               // Choisir un autre process et faire une commutation de contexte
    FSi

V(S)
    e(s) := e(s) +1
    Si (e(s) <= 0 )                         // il y a des process bloqués
        // débloquer un process de la file
        - Choisir un processus de f(s)      // en appliquant une stratégie de sélection Fifo, par priorité,
        - PCB_Choisi := défiler(f(s))       // Retirer son PCB de la file ;
      - PCB_Choisi.Etat := PRET  ;          // Selon la stratégie du scheduler :Insérer le PCB du process choisi  dans la file
                                            // des prêts ou  éventuellement le mettre directement à l'état ACTIF
    FSi
```

- La valeur initiale de e(s) détermine le rôle du sémaphore (sémaphore d'EM, sémaphore de synchronisation, …).

- La valeur initiale de e(s) est positive mais elle peut devenir négative au fur et à mesure des demandes d'accès.

- A un moment donné, e(s) en valeurs positives indique le nombre de process pouvant encore entrer à la SC, et en nombre négatif indique en valeur absolue le nombre de process bloqués dans la file f(s).

- Soit : - *nP(s)* le nombre d'instructions *P* exécutées sur le sémaphore *S*.

    - *nV(s)* le nombre d'instructions *V* exécutées sur le sémaphore *S*.

    - *e0(s)* la valeur initiale du sémaphore *S*.

- *nf(s)* le nombre de process qui ont pu franchir la primitive *P(s),* c.à.d. les process qui, ou bien n'ont pas été bloqués par celle-ci, ou bien ont été bloqués mais débloqués après.

A tout instant on a les relations suivante vérifiées :

$$e(s) = e0(s) - nP(s) + nV(s)$$

$$nf(s) = \min[\, nP(s) \,,\, e0(s) + nV(s)\,]$$

2.4. Application des sémaphores pour résoudre le problème de la SC : Sémaphore d'Exclusion Mutuelle

Les sémaphores permettent de résoudre un grand nombre de problèmes lies a la programmation prallèle, notamment le problème de l'exclusion mutuelle.

En utilisant la notion du sémaphore, le problème de l'EM se résout comme suit : on introduit un sémaphore mutex initialisé à la valeur e0(mutex) = 1 et chaque process s'exécute selon le protocole :

mutex : sémaphore init 1 // partagé

….

Demande d'accès ========> **P(mutex)** // *Accéder, éventuellement, et bloquer l'accès*

< Accès à la SC >

Libérer l'accès ========> **V(mutex)** // *Libérer l'accès et éventuellement réveiller un processus*

…

mutex est appelé un sémaphore d'Exclusion mutuelle

Démontrons qu'un sémaphore d'EM vérifie les propriétés de Dijkstra

- Preuve de la première propriété : l'EM

nP
P(mutex)
nf
SC
nPSC
nV
V(mutex)

Soit nPSC : le nombre de process qui sont simultanément dans la SC, montrons qu'à tout instant « *nPSC ≤ 1* »

On a : nPSC = nf – nV

aussi : nf = min[nP, e0 + nV)] et e0 =1

- Si nf = 1 + nV : alors nPSC = 1 + nV – nV ➔ nPSC = 1

- Si nf = nP : ceci veut dire que : nP < 1 + nV

➔ nf < 1+nV

➔ nf-nV <1

➔ nPSC <1

Donc la contrainte d'EM est respectée.

- Preuve de la deuxième propriété : Absence d'Interblocage

Supposons que la SC est libre et qu'il y a des process qui sont bloqués sur P(mutex) alors :

- SC est libre : nPSC =0 ➔ nf = nV

- des process bloqués sur P(mutex) ➔ nP – nf > 0 donc nP > nf

Mais nf = min[nP, 1 + nV)]

Donc nf = 1+ nv

Contradiction ➔ la deuxième propriété est vérifiée

- Preuve de la troisième propriété : Progression : Supposons qu'un process s'est bloqué hors de sa SC alors qu'il l'a libérée (la SC est libre) et qu'il y a des process bloqués en entrée.

  - la SC est libérée ➔ le process a exécuté V (mutex) par conséquent il va nécessairement débloqué un process parmi les bloqués et le faire passer en exécution ➔ la progression est garantie

- Preuve de la quatrième propriété : Pas de privilège : il est évident que la notion de sémaphore, telle que présentée, n'introduit aucune supposition de privilège sur l'exécution des process.

- La privation : si la file f(s) est gérée par une stratégie FIFO, tout processus entre en SC au bout d'un temps fini (pas de privation). Cependant, si la file applique une stratégie par priorité quelconque, on peut tomber sur des cas de privation.

2.5. Mise en ouvre de l'Exclusion Mutuelle par sémaphores

L'avantage des primitives présentées pour les sémaphores est qu'elles facilitent un petit peu de programmer l'exclusion mutuelle. Cependant l'exécution de ces primitives ne suffit pas pour garantir l'EM, il faut imposer certaines conditions :

- Seules les primitives P et V peuvent modifier le sémaphore associé à elles. Il est impératif de protéger la table des sémaphores contre toute autre écriture.

- Les primitives P et V sont indivisibles et ne peuvent pas être interrompues.

- Les deux primitives P et V sont exclusives entre elles, elles ne doivent pas s'exécuter sur le même sémaphore simultanément.

La première condition est à la charge du SE par son module de protection. Pour mettre en ouvre la deuxième et la troisième condition, deux cas se posent :

1/ Cas monoprocesseur : l'indivisibilité des primitives est assurée de deux façons :

- soit par microprogrammation : ainsi chaque primitive devient une instruction assembleur à laquelle correspond un micropgramme.

- soit par masquage des interruptions : chaque primitive aura l'allure suivante :

```
Debut Primitive
        Masquer les interruptions
        < Corps de la primitive >
        Démasquer les interruptions
Fin de la primitive
```

1/ Cas multiprocesseur : l'indivisibilité des primitives par masquage des interruptions ne suffit pas car l'effet de masquage et démasquage s'applique au niveau de chaque processuer indépendamment de l'autre, ce qui n'empêche pas que deux processus s'exécutant sur deux processuers différents d'exécuter les primitives simultanément. La soluttion dans ce cas est d'utilser, en plus le masquage des IT, une solution par attente active en utilisant par exemple l'instruction TAS de verouillage d'un emlplaceemnt mémoire. chaque primitive aura l'allure suivante :

soit la variable booléenne C indiquant la disponibilité des deux primitives en même temps:

C : booleen init true

C = true : aucune primitive n'est en train d'être exécutée

C = 1 : une, et une seule, primitive parmi les deux est en train d'être exécutée

```
Debut Primitive
    Masquer les interruptions
    Boucle  : TAS(C)                    // Verouillage
                Aller à Boucle
    < Corps de la primitive >
            C := true ;
    Démasquer les interruptions         // DéVerouillage
Fin de la primitive
```

Remarques :

- Dans cette solution, l'inconvénient de l'attente active revient ; mais dans ce cas l'attente ne dure que le temps d'exécution de la primitive qui est censé être très très court.
- Dans le cas de monoprocesseur, et en dehors des sémaphores, l'EM peut être résolue uniquement en utilisant le masquage des interruptions, ainsi on peut s'assurer q'aucun autre proces ne peut interrompre le processus en cours d'utilisation d'une variable critique.

IV. Conclusion

- En s'exécutant en parallèle, les processus peuvent être indépendants, ils peuvent se coopérer comme ils peuvent rentrer en conflit pour la consommation de ressources.
- Les situations de conflit (concurrence) peuvent conduire le SIQ à des situations graves et peuvent même nuire à la cohérence de données.
- La préservation du SIQ dans un état cohérent oblige les process de s'exécuter dans une section critique en exclusion mutuelle selon le protocole :

Demande d'accès

< Accès à la SC >

Libérer l'accès

- La algorithmes par attente active de mise ne ouvre de l'EM mettent en évidence la difficulté de programmer une solution correcte selon les critères de Dijkstra.
- Les verrous et les sémaphores sont des outils facilitant un petit peu la programmation de la SC mais leur mise en ouvre nécessite l'indivisibilité des primitives, leur protection et leur exécution en exclusion mutuelle.
- Les sémaphores sont des outils à utilisation générale surtout pour la synchronisation et la communication entre processus.