Outils de Synchronisation et de communication

I. Introduction

Les processus s'exécutant dans un système d'exploitation ne sont pas toujours en compétition sur la consommation de ressources, ils peuvent définir entre eux des relations ou des contraintes de coopération, qui dépendent de la logique de la tâche à accomplir, fixant leur déroulement dans le temps pour consommer des ressources. Le terme de synchronisation désigne l'ensemble de ses relations et le mécanisme de leur mise au point.

Exemple **:** le problème du Producteur/Consommateur

⇒ Producteur : Dépôt s'il y a au moins une case vide

⇒ Consommateur : Consommation s'il y a au moins une case pleine.

II. Définition

Le problème de la synchronisation consiste à construire un mécanisme, indépendant de la vitesse d'exécution des process, permettant à un processus actif P de :

⇒ D'en **bloquer** un autre process Q ou de se bloquer lui-même en attendant un signal d'un autre process

⇒ D'activer ou **d'éveiller** le process Q en lui transmettant un signal d'activation

III. Formulation et spécification des contraintes de synchronisation

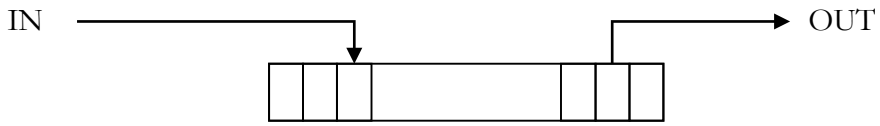
Les contraintes de coopération entre deux process peuvent être formulées selon deux formes :

1. Imposer un ordre de précédence logique dans le temps sur la trace d'exécution de certains points du code.
2. Imposer aux processus une condition de franchissement de certains points de leurs traces temporelles.
3. Ses points, sur lesquels on va imposer des conditions ou un ordre de précédence, sont appelés des points de synchronisation.

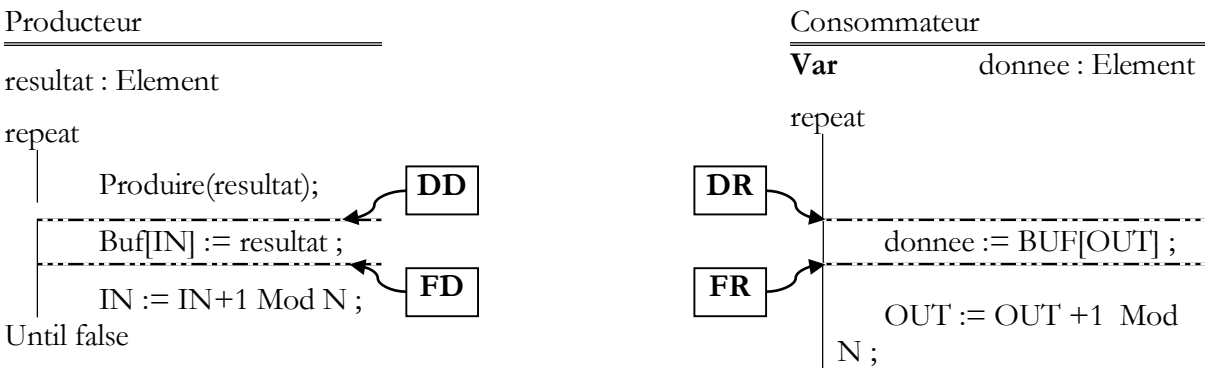
Donc, les processus voulant de se synchroniser doivent :

- ❖ Définir, chacun, ses points de synchronisation.
- ❖ Associer à chaque point de synchronisation les conditions de franchissement exprimés au moyen de variables d'états de synchronisation.

Exemple **:** le problème du Producteur/Consommateur



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



- Quatre points de synchronisation doivent être présents :

1- Début de dépôt	(DD)	3- Début de Retrait	(DR)
2- Fin de dépôt	(FD)	4- Fin de Retrait	(FR)

- Choix des variables d'état : deux approches sont possibles :
  - Approche basée sur l'état de la ressource en commun :

    Var **Nplein** : integer init 0 ; Nombre de cases pleines à un moment donné.

    Nvide : integer init N ; Nombre de cases vides à un moment donné.

    Condition dépôt : (Nplein < N) équivalente à (Nvide > 0)

    Condition retrait : (Nplein > 0) équivalente à (Nvide < N)

  - Approche basée sur l'état des processus en synchronisation :

    Var ND : integer init 0 ; Nombre de dépôts effectués

    NR : integer init N ; Nombre retraits effectués.

    Condition dépôt : (ND-NR < N)

    Condition retrait : (ND-NR > 0)

IV. Réalisation de la synchronisation

Le problème de la réalisation de la synchronisation réside dans la manière d'implémenter les deux actions évoquées dans la définition : bloquer un processus et l'éveiller. Il existe deux types de mécanismes de synchronisation :

⇒ Mécanisme d'action directe (MAD)

⇒ Mécanisme d'action indirecte (MAID)

1. Mécanisme d'action directe (MAD)

Le principe est d'agir directement sur les processus concernés en les mettant directement à l'état bloqué ou à l'état prêt (éventuellement actif) après réveil.

Exemple : les fonctions su SE fourni pour actionner un processus

- Suspendre(process_id) [C++ SuspendThread (HANDEL HTread), Htread le thread principal du processus]
- Reveiller(process_id) [C++ ResumeThread (HANDEL HTread)]

☹ Nécessite de connaître l'identité du process cible (son identificateur) ainsi que le nombre de process à actionner.

☹ L'instant où intervient l'action de blocage n'est pas toujours indifférente, surtout lorsque le process est dans sa section critique. Les MAD doivent prendre en considération cette remarque.

2. Mécanisme d'action indirecte (MAID)

Le principe est d'agir indirectement sur les processus concernés par le bais d'objets intermédiaires communs entre les process.

Deux principaux outils sont disponibles : les Evènements et les Sémaphores

2.1. Synchronisation par évènements

Un évènement est une structure de données commune à des processus permettant une synchronisation passive entre des process qui peuvent l'actionner par des opérations spécifiques. Ainsi, un évènement peut être attendu par des process et déclenché par d'autres.

On distingue deux types d'évènements : évènements mémorisées et évènements non mémorisés

2.1.1. Evènement mémorisé :

L'objet intermédiaire est une structure de données contenant les champs suivants :

```
Type  event  =  Record
                    arrivé   : boolean ;
                    attendu  : boolean ;
                    file     : File de PCB ;
               End ;
```

⇒ *arrivé* : un booléen indiquant si l'évènement est arrivé ou non.

⇒ *attendu* : un booléen indiquant si l'évènement est attendu par des process.

⇒ *file* : une file d'attente contenant les process bloqués en attente de l'évènement.

Pour actionner un évènement, deux primitives sont associées à chaque type d'évènement :

E : event init (false, false, nil) ;

⇒ *Wait(e)*: permet d'indiquer au SE que le process appelant est en attente de l'évènement « e ». Si ce dernier n'est pas encore arrivé, le process appelant sera bloqué et mis dans la file « file », sinon, i.e. si l'évènement « e » est déjà arrivé avant l'appel à *wait*, le processus ne sera pas bloqué.

⇒ *Signal(e)* : permet de signaler l'arrivé de l'évènement « e ».

Selon les systèmes, l'arrivé d'un évènement par le biais de la primitive *signal* libère un ou tous les process qui sont en attente de « e ».

L'évènement mémorisé peut être remis à zéro (l'état initial) :

- Soit explicitement par le biais d'une troisième primitive : *reset(e)*.
- Soit implicitement dès qu'il est pris en compte par le process qui l'attend et qui est rendu actif après l'appel à la primitive signal.

```
Var          e : event   init [false, false, nil] //

Wait(e) // Attendre (e)
    Si (e.arrivé = false)
        e.attendu := true ;
        // 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

Signal(e) // Signaler (e)
    e.arrivé := true ;
    Tq (non filevide(f(p)) )
        // débloquer le process en tête de la file
        - PCB_Choisi := défiler(f(p))       // Retirer son PCB de la file ;
      - PCB_Choisi.Etat := PRET ;
    FTq
    e.attendu = false ;
```

Exemple : E/S

Fin_E/S_event : event

P1	P2
....	
Demande E/S	Exécution de l'E/S
Wait(Fin_E/S_event) ;	…
<ins_suite>	**Signal(Fin_E/S_event)**

2.1.2. Evènement non mémorisé :

Un évènement non mémorisé n'est pris en charge que s'il y a un process qui l'attend. Sinon il sera perdu. Ce type d'évènements est largement utilisé dans les systèmes temps réel où si un évènement est retardé, l'information qu'il apporte peut être considérée périmée et invalide pour prendre une décision .

Exemple : l'évènement horloge dépendant d'une heure bien déterminée.

La structure de données représentant l'objet évènement non mémorisé est identique à celle de l'évènement mémorisé privée du champ « arrivé ».

```
Type  event  =  Record
                    attendu  : boolean ;
                    file     : File de PCB ;
              End ;
```

La notion de l'évènement non mémorisé est délicate à manipuler car elle met en jeu la vitesse d'exécution du process.

2.2. Synchronisation par Sémaphore

La notion du sémaphore étudiée pour résoudre le problème de l'EM peut jouer le rôle d'un évènement de synchronisation mémorisé.

Sémaphore de synchronisation : un sémaphore « synch » initialisé à la valeur « 0 » est dit sémaphore de synchronisation. Il permet de :

⇒ Attendre un signal de synchronisation en bloquant le process appelant en exécutant un P(synch) ;

⇒ Signaler l'arrivé d'un signal de synchronisation en débloquer un process déjà bloqué en exécutant un V(synch)

***Exemple* : Attente de la fin d'une E/S**

Var *Synch sémaphore init 0*

P1	P2
....	
Demande E/S	Exécution de l'E/S
P(synch)	…
<ins_suite>	**V(synch)**

Sémaphore privé : un sémaphore « sempriv » de synchronisation est dit un sémaphore privé d'un processus P si seul ce process peut exécuter l'opération P(sempriv). Les autres process ne peuvent agir sur s que par V(s).

Ainsi, un process P peut se bloquer derrière son sémaphore privé pour attendre un signal de synchronisation. Son réveil sera assuré par d'autres process en faisant exécuter une opération V sur le sémaphore privé de P.

***Exemple* :** relation d'ordre entre deux process P1 et P2

Le process P1 a une instruction A qui ne doit pas être exécutée que si P2 a exécuté l'instruction B. Ecrire le programme de synchronisation entre P1 et P2

Var *sempriv sémaphore init 0*

P1	P2
…..	….
P(*sempriv* **)**	B
A	**V(***sempriv* **)**
….	….

Deux cas peuvent se présenter :

⇒ P1 est déjà bloqué sur P, l'exécution de V par P2 le débloque ➔ la contrainte est satisfaite

⇒ P1 n'est pas bloqué sur P, il n'est pas encore arrivé à cette instruction et P2 a exécuté V, par conséquent le compteur de *sempriv* vaut 1, et dès l'arrivé de P1, il exécute un P sur sempriv qui dans ce cas n'est pas bloquante. ➔ La contrainte est satisfaite.

Notons que P2 peut exécuté « n » fois V donnant la possibilité à P1 d'exécuté « n » fois P sans être bloqué

Le sémaphore sempriv apparaît comme un mécanisme de synchronisation suffisamment général pour permettre non pas uniquement de mémoriser l'arrivée de l'évènement mais il a aussi le potentiel, à la différence des évènements, de mémoriser le nombre d'activations.

Il est possible de combiner les sémaphores d'EM et les sémaphores de synchronisation pour élaborer un modèle de synchronisation plus complexe. Cependant, à tout moment un process doit consulter des variables d'état pour

savoir s'il a le droit de poursuivre son exécution ou se bloquer. La consultation de ces variables ne peut se faire que dans une section critique protégée par un sémaphore d'EM. Le schéma général alors est le suivant :

```
Var        sempriv  sémaphore init 0 ; mutex : sémaphore init 1 ;

P                                                    Un processus activateur
P(mutex)                                             P(mutex)
Modification et test des variables d'état            Modification et test des variables d'état
Si (condition bloquante)                             V(mutex)        // Réveil
   V(mutex) ;      // Libérer le sémaphore d'EM pou les autres pour
                   // qu'ils puissent tester leurs conditions
   P(sempriv) ;    // blocage                        ……..
Sinon
   Actions ;
   V(mutex)
Fsi
```

Exemples :

1) Rendez-vous de N processus : Soit N processus : P_1, P_2, …, P_N, on défini dans chaque process un point particulier de synchronisation, **RVi ,** appelé point de rendez-vous qu'un process ne peut franchir que si tous les autres process aient atteint leurs points de synchronisation.

On veut synchroniser ces process en utilisant les sémaphores.

```
Pi
…..
<Attente des autres process>  --- RVi
….
```

- Les points de synchronisation sont indiqués par des rectangles noirs.
- La condition de franchissement est : cpt = N

```
Var        cpt : integer init 0          // compteur, partagé, du nombre de process qui sont déjà arrivés
           synch  sémaphore init 0 ;     // sémaphore de synchronisation
           mutex : sémaphore init 1 ;    // sémaphore d'EM pour protéger la variable d'état critique « cpt »

Pi
P(mutex)
cpt++ ;
Si (cpt < N)
   V(mutex) ;
      P(synch) ;
Sinon                  // réveiller tous les process  bloqués
      Tant que (cpt <> 1) faire
            V(synch) ;
            cpt --- ;
      FTq
      Cpt := 0 ;       // pour que la procédure soit réutilisable
      V(mutex) ;
Fsi
```

2) Lecteurs / Rédacteurs : Priorité absolue au lecteurs

Dans ce cas il y a une coalition des lecteurs. Autrement dit, s'il y a une lecture en cours tout lecteur venant accède directement au fichier même s'il y a des rédacteurs en attente =➔ possibilité de privation.

Protocole	Lecteurs		Rédacteurs	
	Demande Fichier	**DL**	Demande Fichier	**DE**
	<Lecture>		<Lecture>	
	Libérer fichier	**FL**	Libérer fichier	**FE**

Points de synchronisation

DL : un lecteur ne peut accéder au fichier s'il y a une écriture en cours

DE : un rédacteur ne peut accéder au fichier s'il y a une lecture ou une écriture en cours.

FL : le dernier lecteur doit réveiller un rédacteur en attente.

FE : un rédacteur lorsqu'il termine doit rendre la main à un lecteur en attente, sinon la rendre à un rédacteur en attente

Variable d'Etat

```
nl :        integer init 0      // nombre de lecteurs en cours de lecture
nlatt  :    integer init 0      // nombre de lecteurs en attente
nratt:  integer init 0      // nombre de rédacteurs en attente
E :     booleen init false      // E = true : écriture en cours
```

Conditions de franchissement des points de synchronisation

- DL : (E = false)
- DE : (E = false) and (nl =0)

```
Var        mutex : sémaphore init 1 ;      // sémaphore d'EM
           Sl    : sémaphore init 0 ;      // sémaphore de synchronisation pour bloquer les lecteurs
           Sr    : sémaphore init 0 ;      // sémaphore de synchronisation pour bloquer les rédacteurs
           nl , nlatt, nratt   : integer init 0
           E     : boolean init false :
```

DL

```
P(mutex)
Si (E = true)
    nlatt++
    V(mutex)
    P(Sl);
Sinon
    nl++;
    V(mutex);
FSi
```

<Lecture>

FL

```
P(mutex)
nl - -
Si (nl =0) et (nratt
>0)
    E := true ;
    Nratt - - ;
    V(Sr) ;
FSi
V(mutex);
```

DL

```
P(mutex)
Si (E = true) ou (nl >0)
    nratt++
    V(mutex)
    P(Sr);
Sinon
    E := true
    V(mutex);
FSi
```

<Ecriture>

FL

```
E :=false  ;
P(mutex)
Si (nlatt >0)
    Tant que (nlatt > 0)
        nalatt--;
        nl++;
        V(Sl);
    FTq
Sinon
    Si (nratt >0)
        E :=true  ;
        nratt - - ;
        V(Sr) ;
    FSi
FSi
V(mutex);
```

Remarque : l'utilisation non attentive des sémaphores d'EM avec des sémaphores de synchronisation peut conduire à une situation d'interblocage. Cette situation pourrait avoir lieu lorsqu'un process se bloque en attente d'un signal d'activation émanant uniquement d'un process qui est déjà bloqué. L'exemple suivant en donne une illustration :

Var S1 , S2 : sémaphore init 1

P1	P2
Write(..)	
….	
P(*S1*)	**P(S2)**
P(S2)	P(S1)
…	….
V(*S1*)	**V(*S2*)**
V(S2)	V(S1)

V. Communication entre processus (***IPC : interprocess communication***)

La coopération de plusieurs process à l'exécution d'une tâche commune nécessite en général une communication d'information entre ces process. La synchronisation que nous avons vue dans la section précédente est un cas particulier de communication où l'information transmise est réduite à une action d'autorisation ou d'interdiction l'exécution au-delà de points de synchronisation bien déterminés.

La communication interprocess (IPC) désigne cette notion d'échange <Emission – Réception> de message représentant l'information communiquée, amis elle désigne aussi le mécanisme permettant cet échange.

La communication d'information entre process implique l'accès de ces derniers à un ensemble de variables globales constituant un univers commun entre eux. De cette façon, il existe deux schémas complémentaires de communication :

1. IPC en utilisant des variables communes : dans ce schéma, la communication est à la charge des programmeurs qui doivent définir les fonctions et les conventions de communication, autrement dit, ils doivent définir clairement les règles de manipulations des variables pour permettre l'émission et la réception de messages.

Avantages & inconvénients :

☺ Les fonctions sont spécifiques aux besoins du programmeur

☺ Le programmeur doit implémenter lui-même le système de communication et doit supporter ses erreurs de conception et réalisation (EM, sécurité, …).

2. IPC en utilisant un système de communication : dans ce schéma, la communication est entièrement à la charge du système (en mode noyau) qui doit assumer tout les aspects d'implémentation, de protection (variables partagées) et de manipulation du système de communication.

1. Réalisation de la communication

La communication entre process se résume en un échange de messages entre eux, par conséquent un système d'IPC doit fournir un mécanisme de base permettant au moins l'émission et la réception de messages entre les process communiquant.

Deux fonctions sont nécessaires pour établir la communication entre deux process :

⇒ SEND(destinataire, message) : envoi un message à un process destinataire éventuellement via un objet intermédiaire.

⇒ RECEIVE(émetteur, message) : réception d'un message venant d'un process émetteur éventuellement via un objet intermédiaire.

Par conséquent, la réalisation d'un système d'IPC se réduit à deux problèmes :

⇒ La définition du comportement logique de ces deux fonctions vis-à-vis la réception et l'émission des messages entre process.

⇒ La manière d'implémenter physiquement les deux actions évoquées précédemment (les SDD et la mise au point des algorithmes);

1.1. Le comportement logique peut être défini de deux manières différentes :

⇒ Communication directe

⇒ Communication Indirecte

1.1.1. Communication Directe

Dans cette discipline de communication, tout process voulant communiquer doit nommer son interlocuteur.

- SEND(proccess_destinataire, message)
- RECEIVE(proccess_emetteur, message) ;

Propriétés :

1. Liaison automatique entre chaque paire de process
2. Le process a besoin de savoir, uniquement, l'identité de son interlocuteur
3. Entre deux process, il existe une et une seule liaison
4. La liaison est bidirectionnelle (P←→ Q)

Le modèle de communication tel qu'il vient d'être décrit est dit symétrique (l'identité du process est connue), lorsque le process attend un message de n'importe quel process la communication est dite asymétrique

Exemple : le modèle Producteur / Consommateur

```
Prod                                    Cons
Var     resultat : Element;             Var     donnee : Element ;
repeat                                  repeat
    Produire (resultat);                    RECEIVE(Prod, donnee)
    SEND(Cons, resultat)                    Consommer(donnee) ;
Until false
```

Avantages et Inconvénients

☺ Ce schéma est très utile lorsque un process ne veut communiquer qu'avec des process bien définis, pour but de sécurité, qu'il doit nommer.

☺ La contrainte de « nommer » est aussi un inconvénient majeur limite la souplesse d'utilisation de ce schéma. Si l'on renomme un des process, on doit modifier toutes les instruction de tous les process mentionnant l'ancien nom puis recompiler tous les process. Le même problème se pose lorsque un nouveau process vient joindre d'autres en communication.

1.1.2. Communication Indirecte

L'identité de l'interlocuteur n'est pas obligatoire. Les messages sont reçus et émis à partir d'un objet intermédiaire appelé ***Boite aux lettre*** *(mailbox* ou *port de communication)*. La BAL sert de zone de transit des messages dans laquelle le process peut déposer un message et depuis laquelle il peut en retirer un.

Dans ce type, deux process en communication doivent se partager un objet de type BAL deux opérations alors sont offertes :

⇒ SEND(id_BAL, mess) : envoi un message « mess » à l'objet intermédiaire BAL.

⇒ RECEIVE(id_BAL, mess) : retirer un message de l'objet intermédiaire BAL vers la variable locale «mess ».

Propriétés du modèle indirect de communication:

1. Liaison établie entre deux process s'ils partagent la même BAL.
2. Entre deux process, il pourrait y avoir plusieurs liaisons.
3. La liaison peut être uni ou bidirectionnelle.

Propriétés de la BAL

1. Une BAL peut être crée par un processus qui sera dit le Propriétaire de la BAL.
2. Une BAL est dite privée si seul son propriétaire peut y retirer des messages. Les autres process peuvent uniquement y déposer.
3. La création d'une BAL implique les opérations suivantes :
4. - Affectation d'un identificateur unique à la BAL
5. - Réservation d'un espace mémoire suffisant pour les messages
6. - un descripteur contenant toutes les informations de la BAL est crée et associé à cette BAL

7. Une BAL peut être détruit par son propriétaire ou par le système.
8. Une BAL a une capacité qui peut être :
   - Nulle : la BAL ne peut contenir ni recevoir aucun message
   - Limitée : de taille N messages chacun de M octets
   - infinie : taille théorique, ainsi l'émetteur ne sera jamais bloqué

1.2. Implémentation physique du système IPC

La mise en œuvre du mécanisme de communication dans les deux schémas peut se faire tout simplement selon le modèle Producteur / consommateur. L'opération d'émission [SEND(BAL, mess)] est un dépôt d'un message par le producteur (processus émetteur) et l'opération de réception [RECEIVE (BAL, mess)] est un retrait d'un message par le consommateur (processus récepteur).

Une BAL de capacité N est un buffer de taille N partagé entre les différents process en communication. Ce buffer doit être protégé contre les accès simultanés d'où la nécessité d'un sémaphore d'EM. De plus, tout process voulant déposer un message, en appelant SEND, alors que le buffer est plein doit attendre la libération d'une case ; et tout process voulant retirer un message, en appelant RECEIVE, alors que le buffer est vide doit attendre le remplissage d'une case d'où la nécessité d'un sémaphore de synchronisation.

Une autre solution possible consiste à utiliser deux sémaphores généraux, l'un compte le nombre de cases vides pour déposer un message alors que l'autre compte le nombre de cases pleines pour en retirer un.

```
Var     Npelin : sémaphore init 0 ;     // sémaphore contrôlant les dépôts
        Nvide : sémaphore init N ;      // sémaphore contrôlant les retraits

Producteur                                  Consommateur
Var     resultat : Element;                 Var     donnee : Element ;
        IN : integer init 0;                        OUT : integer init 0 ;
repeat                                      repeat
    Produire (resultat);                        P(Nplein);
    P(Nvide);                                   donnee := BUF[OUT] ;
    Buf [IN]:= resultat;                        OUT := OUT +1  Mod N ;
    IN := IN+1 Mod N;                       V(Nvide);
    V(Nplein);                              Consommer(donée) ;
Until false                                 Until false
```

Remarque

Le principe de la boite aux lettre est largement implémenté dans les systèmes modernes surtout pour la communication réseau. C'est le même principe qu'utilise les process communiquant à travers les ports de communication virtuels, c'est le principe aussi des communications Client Serveur sous les sockets TCP IP, bien sur avec un traitement encore plus délicat pour le protocole TCP IP et les problèmes de sécurité et contrôle qui viennent avec.

C'est aussi le principe de pipe sous Unix.

il existe des sémaphores spéciaux de communication appelés Sémaphores de messages. L'envoi d'un message consiste tout simplement en l'exécution d'une opération V_M(s,message) et la réception se résume en l'exécution d'une opération P_M(s,message)

2. Processus – thread et Exclusion mutuelle sous windows

Un processus sous Windows est une instance d'application qui possède un espace d'adressage, de la mémoire, des fichiers et des fenêtres ouvertes. On crée un nouveau processus par la fonction CreateProcess(…). Ce processus comprendra un thread principal. On peut lui en ajouter d'autres avec la fonction :

HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpsa, DWORD cbStack,
LPTHREAD_START_ROUTINES, lpSStartAddr, LPVOID lpThreadParm,
DWORD fdwCreate, LPDWORD lpIDThread).

Un process fils (ou thread) peut s'assurer l'exclusivité d'une ressource par la fonction WaitForSingleObject() qui correspond à un P de sémaphore d'exclusion mutuelle. On libère le sémaphore par un ReleaseMutex(). Les fonctions InterLockedIncrement() et InterLockedDecrement() sont applicables à des sémaphores généralisés.

3. Quelques API Windows pour manipuler les process et les threads

⇒ HANDLE CreateProcess(..) : permet de créer un process et retourner son Handle (zéeo su echec)

⇒ DWORD GetCurrentProcessId() : retourne l'ID (niveau noyau) du process en cours d'exécution

⇒ HNADLEGetCurrentProcess : retourne le handle (niveau noyau) du process en cours d'exécution

⇒ DWORD WaitForSingleObject(HANDLE hproces ,..) permet d'attendre la fin d'un process. Ce process est mis à l'état bloqué.

⇒ DWORD WaitForMultipleObjects (int nbProcess, HANDLE* hprocess_s ,..) permet d'attendre la fin de nbProcess Process parmi les process handle qui sont dans la table hprocess_s. Ce process est mis à l'état bloqué.

⇒ TerminateProcess(HANDLE hproces ,..) permet de tuer un process dont le HANDLE est hprocess

⇒ HANDLE CreateThread(..., threadProc, ...) permet de créer un thread don't sa procedure d'exécution est threadProc. Elle retourne son handle

⇒ TerminateThread(hThread, dwExitCode) : permet de tuer le thread hThread

⇒ SuspendThread(hThread) : suspend le thread hThread et le met à l'état suspendu

⇒ ResumeThread (hThread) : remet le thread hThread à l'état prêt (et éventuellement le relancer)

4. Gestion de la Section critique

l'objet : CRITICAL_SECTION

```
CRITICAL_SECTION cs1;
volatile DWORD N = 0, M;
/* N is a global variable, shared by all threads. */
InitializeCriticalSection (&cs1);
   ...
EnterCriticalSection (&cs1);
if (N < N_MAX) { M = N; M += 1; N = M; }
LeaveCriticalSection (&cs1);
   ...
DeleteCriticalSection (&cs1);
```

createmutex(…), releasemutex(…), openmutex, waitforsingleobject(hmutex, INFINI), … InterLockedIncrement() et InterLockedDecrement() sont les principales fonctions de gestion de la section critique.

VI. Conclusion

En s'exécutant, les processus ne sont pas toujours en compétition, ils peuvent se définir des contraintes de coopération fixant leurs déroulements dans le temps pour la consommation de ressources.

Le problème de la synchronisation consiste à définir un mécanisme permettant à un processus actif P de :

⇒ D'en **bloquer** un autre process Q ou de se bloquer lui-même en attendant un signal d'un autre process

⇒ D'activer le process Q en lui transmettant un signal d'activation

Les processus voulant se synchroniser doivent :

⇒ Définir, chacun, ses points de synchronisation.

⇒ Associer à chaque point de synchronisation les conditions de franchissement exprimés au moyen de variables d'états de synchronisation

Les sémaphores sont des outils puissants et généraux de synchronisation.

La synchronisation est un cas particulier de la communication interprocess.

La communication interprocess peut se résume en la possibilité d'envoyer et recevoir des messages émanant d'un autre process.

Les boite aux lettres constituent un moyen efficace de communication