Chapitre 05 :

Gestion du parallélisme et communication entre processus

(Parallelism and communication between processes)

1. Introduction :

La programmation Temps Réel fait souvent intervenir plusieurs tâches plus ou moins indépendantes qui constituent le système.

Il se pose alors un certain nombre de problèmes pour la réalisation d'une application TR. On peut classifier ces problèmes en trois catégories :

1. **L'exclusion mutuelle :** accès concurrent à des ressources critiques.
2. **La synchronisation :** relation d'ordre entre les processus (tâches).
3. **La communication :** échange de données.

Donc, chaque application temps réel est souvent constituée de plusieurs tâches exécutées de façon concurrente :

- ces tâches ne sont pas indépendantes,
- besoin d'échanger des données,
- besoin de synchroniser les moments ou les données sont échangées.

2. Définitions :

- **Ressource :** Entité utilisée par un système informatisé pour un bon fonctionnement :
  - Physique : capteurs, actionneurs, périphériques ...
  - Logique : donnée dans une mémoire, code ...
- **Partage de ressources :** l'exécution en parallèle de plusieurs ressources en fournissant le même résultat qu'une exécution séquentielle (pas d'interférences).
- **Section critique (SC) :** est un ensemble d'instruction d'un programme qui peut engendrer des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents (l'existence de section critique implique l'utilisation des variables partagées).
- **Ressource critique :** Accès concurrent par plusieurs processus à un objet (variable, table, fichier, mémoire, périphérique, …). Par exemple deux processus qui tentent d'envoyer simultanément un fichier sur l'imprimante ; le périphérique imprimante devient ressource critique pour eux.

- **Communication, coopération :** garantir que l'échange d'information entre tâches suit un protocole défini.

3. Exclusion mutuelle :

Le problème de l'exclusion mutuelle se pose lorsque plusieurs tâches demandent l'utilisation d'une ressource commune, alors qu'une seule tâche peut avoir accès à la ressource à la fois.

Pour réaliser une exclusion mutuelle qui est nécessaire pour résoudre le problème des accès concurrents, on admet que certaines contraintes doivent être respectées :

- **Le déroulement :** Le fait qu'un processus ne demande pas à entrer en SC ne doit pas empêcher un autre processus d'y entrer. Cette contrainte exclut les méthodes fondées sur un tour de rôle strict.

- **L'attente finie :** Si plusieurs processus sont en compétition pour entrer en SC, le choix de l'un d'eux ne doit pas être repoussé indéfiniment (pas de famine).

- Tous les processus doivent être **égaux** vis à vis de l'entrée en SC.

Voici donc le principe général d'une solution garantissant que l'exécution simultanée de plusieurs processus ne conduirait pas à des résultats imprévisibles :

1. Avant d'exécuter une SC, un processus doit s'assurer qu'aucun autre processus n'est en train d'exécuter une SC du même ensemble.
2. Dans le cas contraire, il ne devra pas progresser tant que l'autre processus n'aura pas terminé sa SC.
3. Avant d'entrer en SC, le processus doit exécuter un protocole d'entrée. Le but de ce protocole est de vérifier justement si la SC n'est occupée par aucun autre processus.
4. A la sortie de la SC, le processus doit exécuter un protocole de sortie de la SC. Le but de ce protocole est d'avertir les autres processus en attente que la SC est devenue libre.

La structure suivante résume ce principe de fonctionnement :

```
Processus Pi

Début

        …
        … <Instructions hors de la section critique>

        Protocole d'entrée en SC

                SC

        Protocole de sortie de SC

        …
        … <Instructions hors de la section critique>

Fin.
```

On peut réaliser l'exclusion mutuelle par trois méthodes :

1. Solutions logicielles :
    - Algorithme de Dekker (Pour 2 processus)
    - Algorithme du Boulanger (pour plusieurs processus)

2. Solutions matérielles :
    - Masquage des interruptions
    - L'instruction TEST-AND-SET
    - L'instruction SWAP
3. Sémaphores.

3.1. Sémaphores :

Les solutions matérielles et logicielles sont difficiles à mettre en œuvre pour des problèmes de synchronisation complexes. Dans cette section, nous définirons l'outil de synchronisation le plus connu qu'est **le sémaphore**. Les sémaphores ont été introduits par **Dijkstra** (informaticien hollandais) en 1965.

Un sémaphore possède une valeur entière ***S***, définie entre 2 primitives :

- ***Secure***(s) décrémentation de la valeur du sémaphore et blocage du processus appelant si la valeur est devenue < 0 → Prendre.

```
Primitive Secure(S)
Début
   S := S - 1;
   Si S < 0 Alors
      Bloquer le processus appelant et placer le dans la File F(s);
   Finsi
Fin.
```

- ***Release***(s) incrémentation de la valeur du sémaphore pouvant entrainer le déblocage d'un processus bloqué → Libérer.

```
Primitive Release(S)
Début
   S := S + 1;
   Si S <= 0 Alors
      Faire sortir le processus de la File F(s) et l'activer;
   Finsi
Fin.
```

De ce qui précède, on peut facilement proposer un schéma de synchronisation de *n* processus voulant entrer simultanément en SC, en utilisant les deux opérations ***Secure*** et ***Release***. En effet, il suffit de faire partager les *n* processus avec un sémaphore **mutex**, initialisé à 1, appelé sémaphore d'exclusion mutuelle.

Chaque processus *Pi* a la structure suivante :

```
Processus Pi
Début
      Secure(mutex);
      SC
      Release(mutex);
Fin.
```

Pour voir davantage l'efficacité des sémaphores comme outil de synchronisation, considérons l'exemple suivant : Deux processus P1 et P2 exécutent respectivement deux instructions S1 et S2.

```
Processus P1
Début
      S1;
Fin.
```

```
Processus P2
Début
      S2;
Fin.
```

Si on souhaite que S2 ne doit s'exécuter qu'après l'exécution de S1, nous pouvons implémenter ce schéma en faisant partager P1 et P2 un sémaphore commun S, initialisé à 0 et en insérant les primitives ***Secure*** et ***Release*** de cette façon :

```
Processus P1
Début
      S1;
      Release(S);
Fin.
```

```
Processus P2
Début
      Secure(S);
      S2;
Fin.
```

Comme ***S*** est initialisé à 0, P2 exécutera S2 seulement une fois que P1 aura appelé ***Release***(S).

Résumé sur les sémaphores :

Lorsqu'un processus désire acquérir une ressource, il exécute une opération **Secure**. Il est donc bloqué si aucune ressource n'est disponible.

Lorsqu'il libère la ressource, il exécute une opération **Release** qui signale la disponibilité de la ressource et débloque donc un processus éventuellement en attente.

Remarque : Théoriquement, un sémaphore peut être initialisé à n'importe quelle valeur entière, mais généralement cette valeur est positive ou nulle. Dans la plupart des cas pratiques, la valeur initiale = nombre de ressources disponibles.

Exercice : 3 tâches doivent utiliser une ressource critique R, tel que :

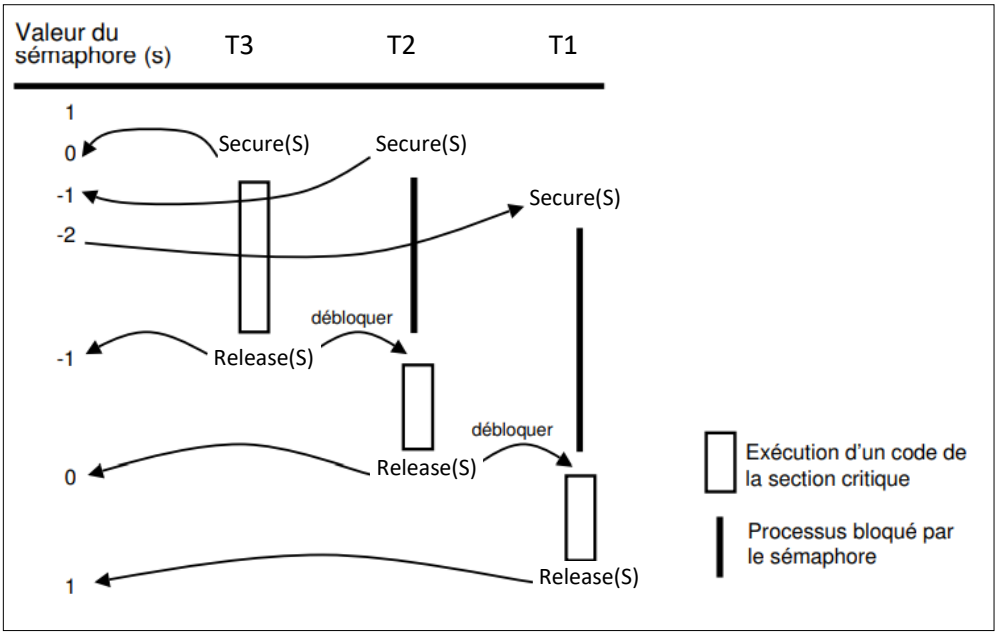
- T1 utilise R pendant 15 UT et demande R à l'instant t=12 UT.
- T2 utilise R pendant 10 UT et demande R à l'instant t=8 UT.
- T3 utilise R pendant 14 UT et demande R à l'instant t=0 UT.

Initialement, ***S***=1 et UT = unité de temps. On utilise un sémaphore **mutex** pour réaliser l'exclusion mutuelle. Que contient la file d'attente et quelle est la valeur de ***S***.

Solution :

Initialisation S=1 (**mutex**)

UT	R	F(s)	***S***
0	T3	Ø	0
8	T3	T2	-1
12	T3	T2 et T1	-2
14	T2	T1	-1
24	T1	Ø	0
39	Ø	Ø	1



4. Synchronisation :

La synchronisation permet de gérer des relations d'ordre de plusieurs tâches. Elle se réalise par sémaphores ou par un évènement.

4.1. Synchronisation par sémaphores :

Chaque tâche (T_i) est définit par un sémaphore **privé** (S_i) : $Ti \rightarrow S_i$

Seule la tâche propriétaire du sémaphore peut exécuter l'opération ***Secure***. Les autres processus ne peuvent exécuter que l'opération ***Release***. En général, la valeur initiale d'un sémaphore privé est 0.

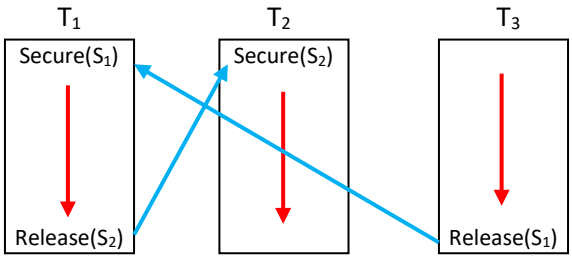
- Initialisation S_i =0,
- T_i ↔ exécute ***Secure***(S_i) seulement;
- T_j ↔ exécute ***Release***(S_i) avec $(i \neq j)$.

Exemple :

$T_3 \rightarrow T_1 \rightarrow T_2$

On définit :

- $S_1 \rightarrow T_1$ et $S_1 = 0$
- $S_2 \rightarrow T_2$ et $S_2 = 0$



4.2. Synchronisation par évènement :

Un évènement (e = event) est une information qui vienne d'une source matérielle (capteurs) ou logicielle (applications). Il peut prendre la valeur "***0 = faux***" ou "***1 = vrai***" et sur lequel 3 primitives sont définies : ***Wait*** = attendre, ***Signal*** = Déclencher, ***Reset*** = Réinitialiser.

```
Primitive Wait(e)
Début
   Si e = faux Alors
      Bloquer le processus dans la File F(e);
   Finsi
Fin.
```

```
Primitive Signal(e)
Début
   Si e = vrai Alors
      Débloquer le processus et placer le à l'état prêt;
   Finsi
Fin.
```

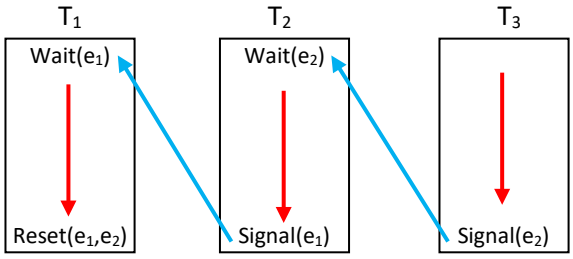
```
Primitive Reset(e)
Début
   e = faux;
Fin.
```

Exemple 01 :

$T_3 \rightarrow T_2 \rightarrow T_1$

Initialisation :

- $e_1 = 0$
- $e_2 = 0$



Exemple 02 : En utilisant les événements, la synchronisation des processus Clavier et Ecran s'exprime comme suit :

```
Var
    c: char;
    e=faux; % évènement initialisé par 0;

Process Clavier;
Var car_lu: char;
debut
    c:= car_lu; %lire un caractère dans la variable car_lu
    Signal(e);
End;

Process Écran;
Var car_à_écrire : char;
debut
    Wait(e);
    car_à_écrire := c;
    % écrire le caractère car_à_écrire sur l'écran.
end;

Begin
    Repeat
        Reset(e);
        Cobegin
            Clavier;
            ecran;
        Coend
    Forever
End
```

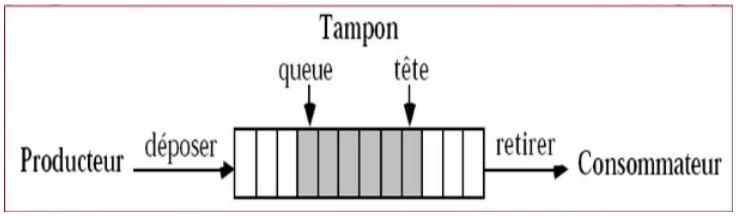
Le processus **Écran** est bloqué uniquement s'il exécute ***Wait(e)*** avant que le processus **Clavier** n'exécute ***Signal(e)***.

Une fois bloqué, le processus **Ecran** est débloqué lorsque le processus **Clavier** exécute ***Signal(e)***. La synchronisation est donc correctement résolue.

Remarque importante :

Les trois primitives (***Wait***, ***Signal*** et ***Reset***) sont utilisées seulement pour la synchronisation par évènement.

5. Problème de Producteur/Consommateur :



Les deux processus (**Producteur** et **Consommateur**) coopèrent en partageant un même tampon (un buffer) :

- Le producteur (**P**) produit des objets qu'il dépose dans le tampon.
- Le consommateur (**C**) retire des objets du tampon pour les consommer.

Problèmes/Conflits :

- Le producteur veut déposer un objet alors que le tampon est déjà plein ;
- Le consommateur veut retirer un objet du tampon alors que celui-ci est vide ;
- Le producteur et le consommateur ne doivent pas accéder simultanément au tampon.

Il y a deux problèmes :

- Les deux tâches ne peuvent pas avoir accès simultanément au buffer → **Exclusion mutuelle**.
- La tâche **C** accède le buffer après la production s'il est vide, tandis que la tâche **P** accède le buffer après la consommation s'il est plein → **Synchronisation**.

Variables et Initialisation :

```
#define N // nombre de places dans la file (taille de tampon)
typedef int sémaphore; //Les sémaphores partagés par tous les processus
sémaphore mutex = 1; // contrôle d'accès section critique (un seul processus en SC)
sémaphore vide = N; // nb. de places libres (la file est toute vide)
sémaphore plein = 0; // nb. de places occupées (aucun emplacement occupé)
```

```
void producteur() {
while (1) {
produire_objet (x); //produire un objet
Secure(vide); // on veut une place vide alors décrémenté des places libres
Secure(mutex); // on bloque la file (début section critique)
mettre_objet(x); // mettre l'objet en file (SC/RC)
Release(mutex); // libération de la file (fin section critique)
Release(plein); // incrémente des places occupées (un objet est à prendre)
}
}
```

```
void consommateur(){
while (1) {
Secure(plein); // attente d'un objet (décrémente des places occupées)
Secure(mutex); // début section critique
retirer_objet(x); // prendre l'objet courant (SC/RC)
Release(mutex); // fin section critique
Release(vide); // incrémente des places vides (une place est à prendre)
consommer_objet(x) //consommer l'objet courant
}
}
```

- ✓ Les processus **producteurs** produisent de l'information vers ces emplacements.
- ✓ Les processus **consommateurs** utilisent cette information et libère la place.
- ✓ Le système synchronise les deux types de processus pour ne pas perdre de données :
    - Bloquer un producteur si pas de place (plein) ou,
    - Bloquer un consommateur si pas d'information disponible (vide).

6. Communication entre tâches :

La communication entre les processus est un échange d'information (données, résultats,...). Elle se concrétise par deux catégories d'outils : les boites aux lettres et la mémoire partagée.

6.1. Boites aux lettres (Mail Box) :

Une boite aux lettres est une structure de données a pour objectif de recevoir ou transmettre des messages (des données) entre les tâches concurrentes et non synchronisées. On peut y écrire des messages et les récupérer.

- ➢ **Écrire message :** Put_mailbox(Identificateur, message)
- ➢ **Lire message :** Get_mailbox(Identificateur, message)

L'identificateur est une variable qui identifie la boite aux lettres. Après la lecture, le message est systématiquement effacé.

6.2. Mémoire partagée :

- Un moyen de communication très efficace entre les tâches :
  - implémente naturellement pour les processus ;
  - ne passe pas par le système d'exploitation (donc rapide).
- Mais dangereux :
  - accès concurrent à une même ressource.
- Elle nécessite l'utilisation des outils de synchronisation (sémaphores).