Communications inter-processus

Les processus, même s'ils sont indépendants sont rarement isolés et doivent pouvoir :

- o partager des ressources sans provoquer d'inter-blocages
- o ou partager des données sans provoquer d'incohérences

  et cela même si ces partages se font à leur insu.

S'ils sont interdépendants, ils doivent pouvoir

- o se synchroniser
- o ou échanger de données

Les processus doivent donc pouvoir communiquer. Les systèmes d'exploitation prenant en charge le multitâche fournissent pour cela des primitives de communication et de synchronisation des processus.

1. Partage d'informations

Les processus peuvent partager des informations de diverses manières, voici quelques techniques envisageables :

- o L'échange d'informations via des fichiers.

  Les informations écrites par un processus peuvent être lues par un autre. Le principe est valable même pour des processus qui ne s'exécutent pas simultanément.
- o Le partage d'emplacements mémoire.

  C'est essentiellement de cette manière que communiquent les *threads* qui rappelons-le, sont des "sous processus" qui partagent un même espace d'adressage.
- o Les pipes, les redirections d'E/S ou un système de messagerie pour processus prévu par le système d'exploitation.

2. Partage des ressources d'un système informatique

Les processus même s'ils sont apparemment indépendants, cohabitent dans l'ordinateur. Ils doivent partager certaines ressources et risquent d'entrer en conflit. Voyons d'abord dans quelles conditions les ressources sont partageables ou non.

- o Sont partageables :

  Les ressources qui peuvent être utilisées par plusieurs processus sans provoquer d'interférences : le processeur, les fichiers en lecture seule et les zones de mémoire qui ne contiennent que des procédures ou des données protégées contre toute modification.
- o Ne sont pas partageables :

  Les ressources qui ne peuvent être utilisées que par un seul processus à la fois ;
  - soit parce qu'il est matériellement impossible de partager cette ressource,
  - soit parce qu'elle contient des données susceptibles d'être modifiées.

  Le résultat est imprévisible si pendant qu'un processus lit une donnée, un autre la modifie.

Une donnée ou une ressource commune à plusieurs processus est non partageable quand l'un des processus peut la modifier.

3. Exemple de conflit d'accès

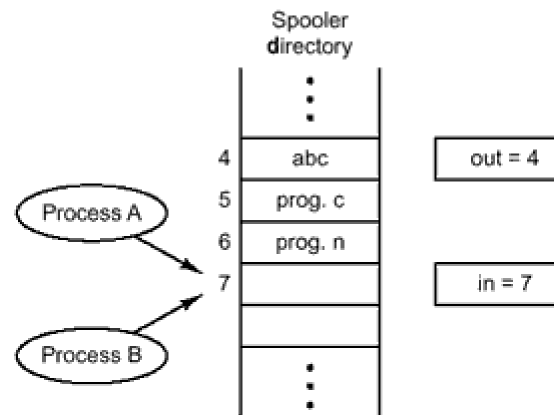
Prenons l'exemple de l'enregistrement de travaux d'impression dans un spooler[1]. Exemple emprunté au cours « Système d'exploitation » d'Andrew Tanenbaum.

L'imprimante est une ressource qui ne peut pas être utilisée simultanément par plusieurs processus. Les travaux d'impression doivent donc être enregistrés dans un répertoire de spool en attendant d'être imprimés.

Les processus qui veulent imprimer ont accès à deux variables partagées *In* et *Out* associées au spooler :

In le numéro à attribuer au prochain fichier qui entre dans le spooler

Out le numéro du prochain fichier à imprimer.



Un processus qui décide de placer un fichier dans le répertoire de spool, doit lire la variable *In*, se servir de cette valeur pour inscrire à l'endroit voulu le fichier à imprimer, puis modifier la variable *In* pour lui faire indiquer l'emplacement libre suivant.

```
i = In;
InscrireFichier( i);
In = i + 1;
```

Les quelques instructions qui se situent entre le moment où on lit la variable partagée *In* et le moment où cette elle est modifiée forment ce qu'on appelle un section critique.

Imaginons que deux processus (A et B) entreprennent presque simultanément d'envoyer un document à l'impression.

Si le processus A est interrompu pendant l'exécution de la section critique pour qu'entre temps l'ordonnanceur passe la main au processus B, la variable *In* sera relue par B et réutilisée une deuxième fois consécutivement avant d'avoir été incrémentée par le processus A. Les deux processus vont donc inscrire le fichier à imprimer au même emplacement du spool et, bien que l'état du répertoire de spooler semble cohérent, un des deux fichiers ne sera jamais imprimé.
Pour éviter ces ***conflits d'accès***, il faut empêcher la lecture ou l'écriture de données partagées à plus d'un processus à la fois. La solution est l'***exclusion mutuelle*** pour les ***accès concurrents***.

4. La synchronisation des processus

La synchronisation des processus concerne les processus parallèles qui utilisent des ressources ou données non partageables.
Il existe un certain nombre d'algorithmes pour y parvenir. Ils sont généralement assez courts mais la compréhension de leur comportement est rendue difficile parce qu'ils s'exécutent en parallèle dans chaque processus mais de manière asynchrone et imprévisible.

[1] SPOOL = Simultaneous Peripheral Operation OnLine

Il faut être sûr, non seulement que les interférences soient évitées, mais aussi qu'ils ne provoquent pas de blocages et qu'ils soient équitables pour chaque processus. (interblocage et famine)
Les systèmes d'exploitations multitâches fournissent de tels mécanismes.

Section critique

L'exemple du répertoire de spool nous montre que les deux processus A et B ne sont pas totalement indépendants. Les variables (*In* et *Out)* susceptibles d'être modifiées sont des ***ressources critiques***. La partie de programme où se produit le conflit est appelée ***section critique***. Il faut donc contrôler les accès aux variables partagées. Le problème est résolu si on peut être sûr que deux processus ne sont jamais en section critique en même temps.

Cette première condition permet d'éviter les conflits d'accès mais il y a d'autres types d'interférences à éviter. En dehors des sections critiques aucun processus ne peut bloquer les autres et aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

Les mécanismes qui mettent en œuvre l'exclusion mutuelle doivent satisfaire les quatre conditions suivantes :

1. Deux processus ne peuvent se trouver simultanément dans la même section critique.
2. Aucune hypothèse ne peut être faite ni sur les vitesses relatives des processus ni sur le nombre de processeurs.
3. Aucun processus hors d'une section critique ne peut bloquer les autres.
4. Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

Blocage

Le blocage ou interblocage *(deadlock)* peut se produire quand les ressources requises pour certains processus sont utilisées par d'autres qui en revanche attendent eux aussi des ressources utilisées par les premiers.
Le système d'exploitation doit être capable de prévenir ou d'atténuer l'effet des blocages

Famine

La famine *(starvation)* est la situation d'un processus qui reste indéfiniment bloqué dans l'attente d'une ressource sans pour autant être en situation d'interblocage. Cette situation dépend des autres processus alors que dans le cas de l'interblocage il faut que le système d'exploitation intervienne d'autorité en retirant une ressource à l'un des processus.

5. Mécanismes d'exclusion mutuelle

5.1 Désactiver les interruptions

Nous savons que dans un système monoprocesseur c'est un ordonnanceur qui tour à tour passe la main à chaque processus pour donner une impression de parallélisme. Il est possible d'éviter l'intervention de l'ordonnanceur dans une section critique en s'interdisant d'exécuter des instructions susceptibles de provoquer le déroutement et en désactivant momentanément les interruptions.

```
DisableInterrupt()
. . .     // Section critique
EnableInterrupt()
```

Le processus qui entre dans la section critique monopolise ainsi le processeur en excluant tout autre processus.

Inconvénients de ce mécanisme :

- il ne convient pas s'il y a plusieurs processeurs
- certaines interruptions risquent d'être perdues si elles ne sont pas traitées à temps
- les processus qui attendent risquent la famine

5.2 L'instruction TSL

Les processeurs conçus pour la multiprogrammation prennent en charge des instructions de type TSL (***Test and Set Lock***) Cette instruction exécute un succession ininterruptible de deux opérations : **tester & verrouiller**.

L'instruction TSL fait référence à une variable partagée. Appelons la *Lock*. Lorsque cette variable est à 0, n'importe quel processus peut, via l'instruction TSL, la mettre à 1 *« Set Lock »* puis entrer dans la section critique. La remise à 0 de la variable *Lock* ne nécessite pas une instruction indivisible.

NB. Le processeur qui exécute cette séquence, s'il est dans un système multiprocesseur, doit verrouiller en même temps le bus mémoire pour en interdire l'accès aux autres processeurs durant l'instruction TSL.

```
EntreeSectionCritique :
      TSL   Reg, LOCK               ;Si LOCK = 0 , le mettre à 1
      CMP   Reg, 0                  ;Si la valeur lue par TSL != 0
      JNE   EntreeSectionCritique   ; recommencer le test en attendant
      . . . section critique
      MOV   LOCK, 0
      RET
```

Les solutions matérielles présentées provoquent des boucles d'attente. Bien qu'une telle boucle soit appelée *« attente active »*, elle occupe le processeur sans rien faire d'utile. Il serait préférable dans ce genre de situation que le système d'exploitation bloque le processus en attente et libère le processeur pour d'autres processus.

5.3 Les verrous

Le verrou *(lock)* est un mécanisme destiné à mettre en œuvre l'exclusion mutuelle. C'est un objet système auquel sont associées deux opérations systèmes, verrouiller et déverrouiller.

- *Verrouiller* est l'opération qui permet à un processus d'acquérir la ressource si elle est disponible. Dans le cas contraire, le processus passe dans l'état bloqué en attendant la ressource.
- *Déverrouiller* est l'opération système par laquelle le processus libère la ressource qu'il monopolisait.

Ces opérations aussi appelées primitives système sont des opérations indivisibles. Le système d'exploitation garantit qu'elles ne pourraient être interrompues par d'autres processus.

5.4 Les sémaphores

Le sémaphore est un mécanisme proposé par le mathématicien et informaticien néerlandais Dijkstra en 1965.

Le sémaphore est une généralisation du verrou pour des ressources pouvant être utilisées non pas par un seul processus mais par plusieurs processus simultanément.

Le sémaphore est en quelque sorte un distributeur de jetons d'accès à la ressource. Le nombre de jetons pouvant être distribués dépend du nombre de processus qui peuvent accéder en même temps à la ressource. Le nombre de jetons est fixe. Il est initialisé lors de la création du sémaphore.

Chaque processus, pour avoir le droit d'accéder à la ressource, doit emprunter un jeton qu'il rend après utilisation.
Le sémaphore est donc une variable entière dont la valeur positive est fixée lors de sa création et qui n'admet par la suite que deux opérations P(s) et V(s) du néerlandais *Proberen* = essayer et *Verhogen* = incrémenter.

Synonymes de P / V: Down() / Up() Wait() / Signal()

Les sémaphores sont facilement mis en œuvre par les systèmes d'exploitation. Le sémaphore est un objet système qui encapsule la variable entière *« niveau du sémaphore »* (=nombre de jetons restants) et une file d'attente de processus.

L'opération P(s) décrémente la valeur du niveau de sémaphore si elle est supérieure à 0 (s'il y a des jetons disponibles). Quand la décrémentation est impossible le processus est bloqué et attend dans la file d'attente que la décrémentation redevienne possible. Il attend donc qu'un autre processus rende un jeton et le signale par l'opération V(s) qui remonte le niveau de sémaphore.

L'intérêt de ces opérations P(s) et V(s) réalisés par des fonctions du système d'exploitation est que ce sont des opérations indivisibles, elles ne peuvent être interrompues. On dit encore qu'elles sont exécutées de façon atomique.

On désigne par le nom de ***mutex*** ou par l'expression sémaphore binaire, les sémaphores qui n'admettent que deux valeurs 0 et 1 (Verrouillé et déverrouillé) C'est en quelque sorte un verrou qui fonctionne en logique négative + le mécanisme de la file d'attente.

6. Application : le problème des lecteurs-rédacteurs

Nous avons vu que les données sont partageables tant qu'elles sont en lecture seule. Les risques d'incohérence apparaissent dès qu'un processus modifie les données. Appelons les processus qui lisent les données des *lecteurs* et ceux qui modifient ces données les *rédacteurs*.

Voici les règles à observer :

- les données peuvent être lues simultanément par plusieurs lecteurs
- elles ne peuvent pas être lues pendant qu'un rédacteur les modifie
- elles ne peuvent être modifiées que par un rédacteur à la fois.

```
Initialisation()
{
      mutex_L = 1;
      mutex_R = 1;
      nl = 0 ;          // Nombre de lecteurs = 0
}

Ecrire()
{
      P(mutex_R);       // ⇒ Un seul rédacteur
      …
      écriture
      …
      V(mutex_R);
}

Lire()
{
      P(mutex_L);
      nl++;             // Un lecteur de plus
      if ( nl==1)       // Le premier lecteur
      {                 //   Vérifie qu'il n'y a pas déjà un rédacteur
         p(mutex_R)     //   et empêche l'arrivée d'un rédacteur
      }
      V(mutex_L);
      …
      lecture
      …
      P(mutex_L);
      nl--;
      if ( nl==0)       // S'il n'y a plus de lecteur
         v(mutex_R) ;   //   autoriser les rédacteurs
      V(mutex_L);
}
```