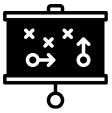

Systèmes concurrents

Définition

▼ Ressource critique

Ressource ne pouvant être utilisé que par **une activité à la fois**. Exemple : L'imprimante, deux processus ne peuvent pas imprimer en même temps sinon la feuille qui va sortir de l'imprimante sera un mélange des deux documents imprimés.

▼ Activité

Une activité est une unité d'exécution indépendante qui peut être un thread ou un processus.

▼ Processus

Un **processus** est un programme en cours d'exécution. Il s'agit d'une entité d'exécution isolée qui ne partage pas de données ni d'informations. Chaque processus a son propre espace d'adressage. Un processus peut avoir plusieurs threads.

▼ Modèle centralisé

Les programmes exécutés partagent des variables globales grâce à une mémoire partagée.

▼ Modèle réparti

Les programmes peuvent échanger des informations via des communication par message (envois et reception de message entre programmes)

▼ Modèle synchrone

Toutes les activités exécutent la **même instruction au même moment**. La progression dans un même code est **la même pour toutes** les activités parallèles d'un même programme, aucune ne va plus vite qu'une autre. Cela est régi par une horloge. C'est compliqué à mettre en place et ce n'est pas ce qui est étudié dans ce cours.

▼ Modèle asynchrone

Chaque activité progresse à son rythme. De ce fait, il est quasi impossible de prédire à l'avance l'ordre d'exécution des instructions de chaque activité. En effet, lors d'une exécution l'activité 1 peut aller plus vite que l'activité 2 et lors de l'exécution suivant, l'activité 2 ira plus vite que l'activité 1. Une exécution spécifique est donc difficilement

reproductible. On dit que le modèle est **non déterministe**. C’est le modèle qui sera étudié dans ce cours.

▼ Exclusion mutuelle

Propriété garantissant qu'une ressource critique (comme une section de code ou une variable partagée) ne peut être utilisée que par **une seule** activité à la fois. Cela évite les conflits et les problèmes de concurrence lors de l'accès à la ressource.

▼ Invariant

Un invariant est une propriété qui doit être toujours vraie à certaines étapes du programme, prouvant ainsi que le programme va correctement s’exécuter.

▼ Propriété de sureté

Propriété définie dans un programme afin de garantir que le programme **n’ira pas** dans une situation incorrecte.

Par exemple, dans le cadre d’un système de réservation de places pour un concert, une propriété de sureté est : le nombre de place vendu doit être inférieur au nombre total de place disponible. La sureté est associé à l’exclusion mutuelle (si personne ne se marche dessus, tout devrait bien se passer)

▼ Propriété de vivacité

Une propriété de vivacité garanti qu’une action va s’exécuter tôt ou tard. Cela empêche la famine d’une activité donc le blocage du système. Par exemple, lors de l’exécution de tâche en parallèle, une propriété de vivacité peut être que chaque activité doit s’exécuter (tôt ou tard). La vivacité est associé à **l’absence de famine**.

On distingue deux types de vivacité :

- La **vivacité faible** (progression) lorsque une activité dépose **plusieurs** demandes d’accès à une ressource et **qu’une** de ses demandes est validé
- La **vivacité forte** (équité faible) lorsque une activité dépose **une** demande et qu’elle sera satisfaire **tôt ou tard.**

La famine arrive donc quand une activité attend indéfiniment qu’une demande soit satisfaite.

▼ Interblocage

L’interblocage se produit quand dans un ensemble de processus, tous les processus sont en attente d’une ressource prise par un autre processus qui ne la libèrera pas.

Exemple :

1. Processus A prend ressource 1
2. Processus B prend ressource 2
3. Processus A demande ressource 2 (bloqué car déjà prise par B)

4. Processus B demande ressource 1 (bloqué car déjà prise par A)

Attente infini.

On associe la vivacité faible à l'absence d'interblocage et la vivacité forte à l'absence de famine.

Il existe plusieurs approches pour éviter l'interblocage telles que :

- Approche par évitement de l'accès exclusif : **Pas de blocage** lors de l'accès à une ressource donc **pas d'interblocage**.
- Approche par classe ordonnées : **Toutes les activités** demandes les ressources dans le **même ordre** donc pas de risque de **croisement** donc par de risque d'**interblocage**.
- Approche par allocation globale : Une activité obtiens soit **toutes ses demandes** de ressource **d'un coup**, soit **rien du tout** donc par d'interblocage. Cela implique un risque de famine et de sur-allocation.
- Approche par libération ré-acquisition : Avant de faire une demande de ressource, une activité **libère toutes les ressources** qu'elle à déjà acquises puis redemande ces même ressource. Vu qu'il y a libération **obligatoire**, il n'y a pas d'interblocage. Par contre il peut y avoir de la **famine**.

▼ Instruction atomique

Une instruction atomique est une instruction du code qui sera exécuté **d'une traite** sans être perturbée par d'autres activités. Cela peut par exemple garantir qu'une variable sera incrémentée sans qu'elle ne soit modifiée pendant l'incrémentation par une autre activité. Cela aide à garantir la cohérence des variables partagés.

▼ Sémaphore

Le concept de sémaphore est un mécanisme qui permet de traiter le parallélisme afin d'éviter que deux processus accèdent en même temps à une ressource critique. Il fonctionne sous la forme d'une **variable partagée** qui peut être **bloquée** et **débloquée** de manière **atomique**. Si un processus essaie de bloquer la variable sémaphore alors quelle est déjà bloquée par un autre alors son appel est rejeté et le processus ne peut pas accéder à la ressource critique.

En général pour bloquer un sémaphore on utilisera l'opération `P()` qui vient du néerlandais *Proberen* (essayé en français). Si le sémaphore est déjà à 0, alors l'appelle sera bloqué.

De même pour débloquer un sémaphore, on utilisera l'opération `V()` qui signifie *Verhogen* (augmenter en français).

▼ Transaction

Une transaction permet d'effectuer une action incertaine sans risque l'altérer l'intégrité des données si l'action échoue. Les transaction sont très utilisées dans le contexte des bases

de données. Une transaction garantie les propriétés **ACID** : (Atomique, Cohérent, Isolé et Durable)

Une solution pour effectuer une transaction peut être d'utiliser la technique des deux phases. Premièrement, il y a acquisition du verrou là ou les actions vont être effectuées. Puis il y a la phase de validation, si c'est validé on dit qu'il y a commit, sinon il y a rollback. Dans tous les cas la cohérence de la base de données est conservé. Ce n'est qu'une solution parmi d'autres.

Analogie : Vous faites vos courses dans un supermarché, vous entrez dans le supermarché (acquisition du verrou) vous prenez les articles que vous souhaitez puis vous allez en caisse pour payer. Si vous arrivez à payer alors la transaction est passée, le magasin à reçu le paiement et réduit son stock. Sinon vous devez reposer vos articles pour que le supermarché ait le même stock qu'avant votre arrivée.

▼ Sérialisation

La sérialisation consiste à assurer qu'un ensemble d'actions **parallèles** donne le **même résultat** qu'une des exécutions **séquentielles** de ce même actions.

▼ Moniteur

Un moniteur est un objet donnant accès à des **procédures** (des opérations). L'accès à ces procédures doit se faire en **exclusion mutuelle**. La partie synchronisation des actions se fait **dans** le moniteur et **pas** dans le programme lui même (l'appelant du moniteur).

Un moniteur possède :

- Une **interfaces** : L'interface est composée de **procédures** aussi appelés **opérations** que le moniteur peut effectuer. Dans le cas d'un buffer ce sera `put(item)` et `get()`.
- Des **prédicats d'acceptation** : Les prédicat d'acceptation sont les conditions pour que les **opérations** puissent être réalisées. Dans l'exemple du buffer, le prédicat d'acceptation pour la procédure `put(item)` serait `count ≤ size` et pour `get()`, `count > 0`
- Des **variables d'état** : Pour pouvoir écrire les prédicats d'acceptation nous avons besoin de **variables d'état** qui vont décrire l'état courant de la ressources critique. Dans l'exemple du buffer, les variable d'états seraient `buffer[]` pour stocker les éléments, `nbElements` pour savoir combien il y a d'éléments dans le buffer et `tailleBuffer` pour avoir la taille maximale du buffer.
- Un **invariant** : L'invariant est une condition qui doit être **toujours vraie** pour que le moniteur fonctionne correctement. Dans notre exemple du buffer l'invariant serait : `0 ≤ count ≤ size`.
- Des **variables conditions** : Les variables conditions permettent d'attendre et de signaler la validité d'un prédicat. Toujours dans l'exemple du buffer, nos variables de conditions serait `notFull` pour `put(item)` et `notEmty` pour `get()`.

Exemple d'implémentation d'un moniteur pour un buffer en Java :

```
class Buffer {
    private int buffer[]; // Le tableau qui va stocker les éléments du buffer
    private int count, in, out, size; // Les variables d'état
    private Condition notFull, notEmpty; // Les variables de condition

    public Buffer(int size) { // Le constructeur de la classe
        this.size = size; // Initialisation de la taille du buffer
        buffer = new int[size]; // Création du tableau de la taille spécifiée
        count = in = out = 0; // Initialisation des variables d'état
        notFull = new Condition(); // Initialisation de la variable de condition pour put(item)
        notEmpty = new Condition(); // Initialisation de la variable de condition pour get()
    }

    public synchronized void put(int item) { // La méthode pour ajouter un élément au buffer
        while (count == size) { // Si le buffer est plein, on attend
            notFull.await();
        }
        buffer[in] = item; // On ajoute l'élément à la position 'in'
        in = (in + 1) % size; // On met à jour la position 'in'
        count++; // On incrémente le compteur
        notEmpty.signal(); // On signale qu'il y a au moins un élément dans le buffer
    }

    public synchronized int get() { // La méthode pour récupérer un élément du buffer
        while (count == 0) { // Si le buffer est vide, on attend
            notEmpty.await();
        }
        int item = buffer[out]; // On récupère l'élément à la position 'out'
        out = (out + 1) % size; // On met à jour la position 'out'
        count--; // On décrémente le compteur
        notFull.signal(); // On signale qu'il y a au moins une place libre dans le buffer
        return item; // On retourne l'élément récupéré
    }
}
```