IMN638 - Interactions visuelles numériques

Chapitre 2.7 - Introduction au parallélisme

Université de sherbrooke

18 novembre 2013

Sommaire

- Notion de calcul parallèle
- Notion de concurrence
- Mécanismes de Synchronisation
- Fils d'exécution et processus

Introduction

Dans un contexte d'interactions visuelles numériques, la performance d'une application pendant son exécution est un facteur particulièrement influent lors de son développement.

Au cours des dernières années, la grande majorité des ordinateurs sont devenus des dispositifs de calcul parallèle, pouvant exécuter plusieurs opérations différentes au même moment. Pour obtenir les meilleures performances possibles lors de l'exécution d'une application interactive, il est primordial de savoir tirer profit de cette nouvelle réalité.

Sans être exhaustif sur le sujet, ce chapitre se penchera sur les concepts de base du parallélisme. Nous explorerons notamment la notion de calcul parallèle, de concurrence et de synchronisation. Nous terminerons en explorant comment se traduit le parallélisme, en pratique, sur un ordinateur, en introduisant les fils d'exécution et les processus.

Introduction :

Pourquoi le calcul parallèle ?

- Pour réduire le temps d'exécution d'une tâche.
- (Corollairement à 1) Pour augmenter la vitesse d'exécution d'une tâche.
- Pour résoudre des problèmes complexes et lourds en calcul.
- Pour tirer profit de ressources de calcul multiples et dispersées.

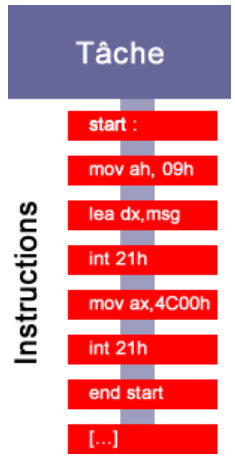
Notion de calcul parallèle

Notion de calcul parallèle

La notion de calcul parallèle s'introduit en opposition à la notion de calcul en série.

Comme nous le savons, un programme informatique, que nous appellerons ici de manière plus générale "une tâche informatique", est consistué d'une liste d'instructions. **Chaque instruction est une action atomique sur le processeur et le processeur exécute les instructions dans l'ordre qu'elles sont définies dans notre tâche (programme)**.

Dans un contexte de calcul en série, les instructions de notre programme sont effectuées les unes à la suite des autres. À un moment précis dans le temps, le processeur n'exécute donc qu'une seule instruction.

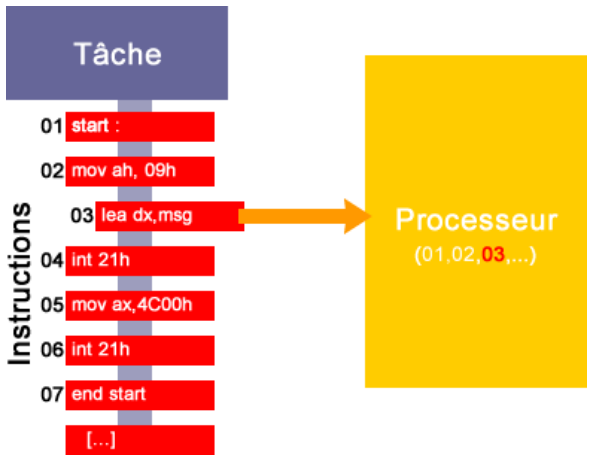


Une tâche est constituée d'une liste d'instructions.

Notion de calcul parallèle

Calcul en série :

Dans le cas d'un calcul en série, **le processeur exécute une seule instruction à la fois pour une tâche donnée**. Dans une tâche, le processeur attend donc que l'instruction $n-1$ ait terminé d'être exécutée, avant d'effectuer l'exécution de l'instruction n.

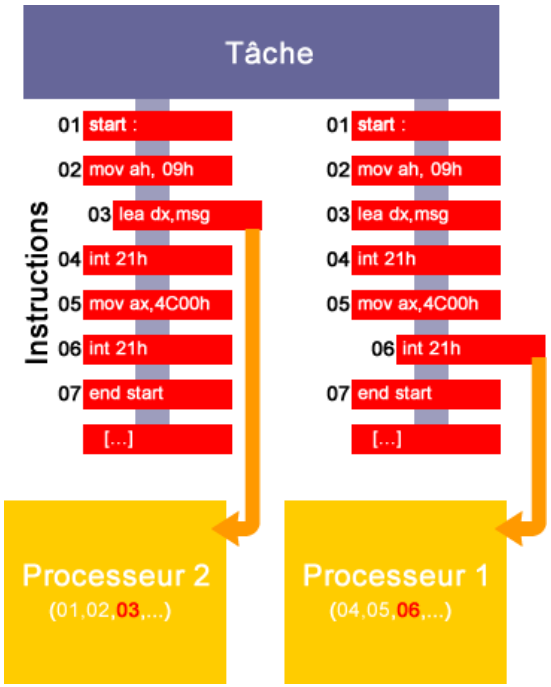


Le processeur exécute séquentiellement les instructions de la tâche. Une instruction est exécutée à la fois.

Notion de calcul parallèle

Calcul parallèle :

Dans le cas d'un calcul parallèle, plusieurs instructions d'une même tâche peuvent être effectuées au même moment, de manière indépendante. La liste d'instructions est donc séparée à un certain moment sur différents processeurs ou différentes unités de calcul et s'exécutent simultanément. Chaque suite d'instructions est exécutée parallèlement à une autre suite d'instructions. Cependant, une suite donnée d'instructions demeure exécutée elle-même en série.



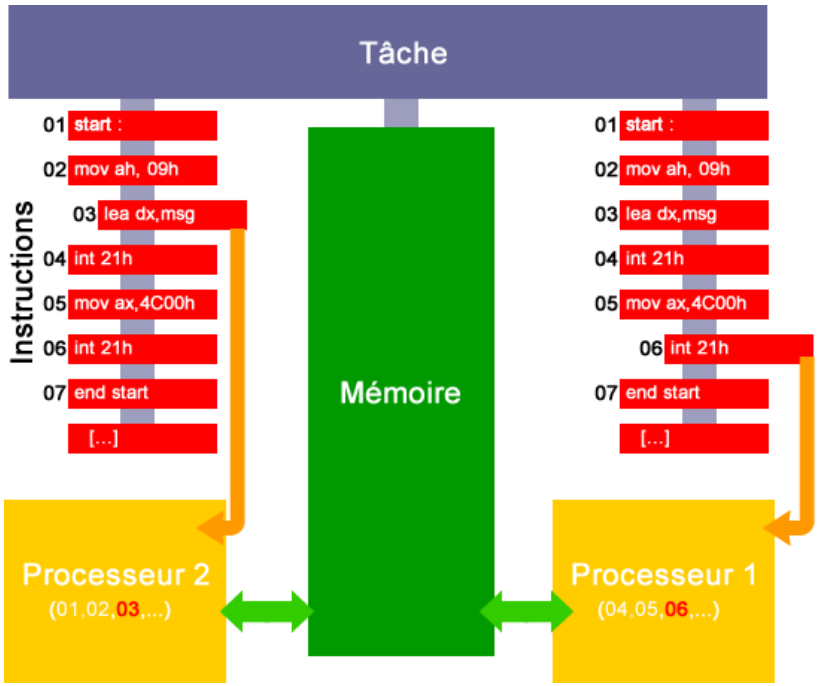
Plusieurs instructions d'une même tâche sont exécutées en même temps par plusieurs unités de calcul, ou plusieurs processeurs différents.

Notion de concurrence

Notion de concurrence

On parle de concurrence lorsque les éléments suivants sont observées dans l'exécution d'une tâche :

- La tâche peut être exécutée en parallèle.
- Les instances parallèles de la tâche peuvent accéder à des ressources communes (mémoire, périphérique, etc.).



Contexte d'exécution d'une tâche en parallèle où la mémoire est partagée entre les instances parallèles. On a donc ici une situation de concurrence.

Notion de concurrence

Somme toute, lorsqu'on parle de concurrence, on parle ici de **concurrence par rapport aux ressources de l'application**. Autrement dit, il y a concurrence s'il y a un **risque que la même information soit accédée exactement au même moment par deux instances parallèles de la tâche**.

Une ressource (par exemple de la mémoire) peut être accédée de différentes façons par une instance parallèle S_i de notre tâche :

- En lecture : $R(S_i)$
- En écriture : $W(S_i)$

Notons que $R(S_i)$ retourne les ressources lues par le processus S_i, et $W(S_i)$ retourne les ressources modifiées/écrites par le processus S_i.

Notion de concurrence

Il est possible de poser certaines contraintes pour que l'exécution d'une tâche concurrente fonctionne correctement.

Une exécution concurrente s'effectue sans problème si les **conditions de Bernstein** sont respectées. Soit :

- $R(S_1) \cap W(S_2) = \emptyset$ (Les emplacements lus par l'instance parallèle S_1 à un moment précis ne sont jamais les emplacements écrits (modifiés) par l'instance parallèle S_2 à ce même moment.)
- $W(S_1) \cap R(S_2) = \emptyset$ (Idem qu'en 1, mais les instances parallèles sont interverties.)
- $W(S_1) \cap W(S_2) = \emptyset$ (Les emplacements modifiés par l'instance parallèle S_1 à un moment précis ne sont jamais les emplacements modifiés par l'instance parallèle S_2 à ce même moment.)

Notion de concurrence - exemple

Admettons un système bancaire où il est possible de faire un retrait à un guichet automatique.

Le retrait s'effectue ainsi (Les lignes sont identifiées entre crochets pour aider à l'explication dans l'acétate suivante.) :

```
void Retrait(int NumeroCompte, float MontantRetrait)
{
    [Get Balance]       float BalanceCompte = RequeteBanque(GET_BALANCE,NumeroCompte);
    [Nouvelle Balance]  BalanceCompte = BalanceCompte - MontantRetrait;
    [Set Balance ]      RequeteBanque(SET_BALANCE,NumeroCompte,NouvelleBalance);

    //Le guichet fait sortir l'argent ici.
}
```

Notion de concurrence - exemple

Supposons maintenant que le compte est en réalité un compte commun, accédé simultanément par l'individu *I*1 et l'individu *I*2. Supposons aussi que le compte possède 1000$ dans sa balance et que les individus *I*1 et *I*2 font un retrait de 50$ environ au même moment sur le compte. L'exécution pourrait se dérouler comme suit :

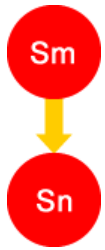
temps	**I1**	**I2**
t_1	[Get Balance] (BalanceCompte = 1000)	
t_2	[Nouvelle Balance] (BalanceCompte = 950)	
t_3		[Get Balance] (BalanceCompte = 1000)
t_4		[Nouvelle Balance] (BalanceCompte = 950)
t_5	[Set Balance] (950$ dans le compte)	
t_6		[Set Balance] (950$ dans le compte)

L'exécution terminée, le compte contient 950$ à sa balance, même si chaque utilisateur a retiré 50$. (Et qu'il devrait donc contenir 900$.) Il y a donc un problème de concurrence puisque, deux personnes pouvaient accéder et modifier le compte (la ressource) simultanément sans que les deux modifications ne soient prises en compte.

Notion de concurrence

Pour représenter le parallélisme en fonction de la concurrence, il est courant de construire des graphes dit "de précédences". Ces graphes mettent en valeur les relations de dépendances dans l'ordre d'exécution du code en regard à l'utilisation des ressources. À partir d'un graphe de précédence, il est très facile de voir quels processus peuvent être effectués parallèlement ou non.

Dans un graphe de précédence, si une section de code *Sm* d'un algorithme modifie des données qu'une autre section de code *Sn* utilise plus loin dans la tâche, on dit que *Sm* a une précédence par rapport à *Sn*. *Sm* et *Sn* doivent donc être exécutés en séquence et ne peuvent être exécutés en même temps, parallèlement.



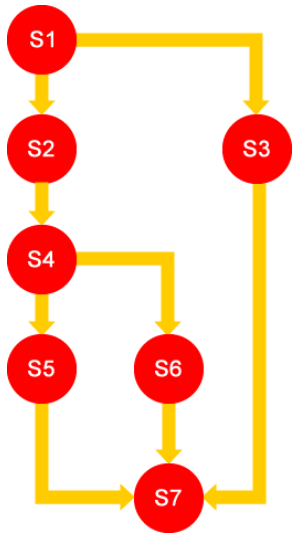
La section *Sm* possède une précédence sur *Sn*.

Notion de concurrence

Exemple d'un graphe de précédence :

Exemple de code

```
a = x + y; //Section 1
b = z + a; //Section 2
c = a - z; //Section 3
d = b + 1; //Section 4
e = d + a; //Section 5
f = d - 1; //Section 6
r = c * (e-f); //Section 7
```



Graphe de précédence correspondant au problème ci-contre.

Dans ce cas-ci, il serait donc possible de paralléliser certaines étapes de l'algorithme. Par exemple, les sections $S2$ à $S6$ ne dépendent pas de $S3$ et pourraient donc être effectuées en parallèle de celui-ci. Les étapes $S5$ et $S6$ pourraient aussi être exécutées en paralllèle.

Notion de concurrence

En programmation parallèle, certaines sections de code auront besoin de l'information modifiée par d'autres sections préalablement exécutées.

Dans l'exemple précédent, la section $S7$ doit s'exécuter seulement lorsque les sections $S3$, $S5$ et $S6$ ont terminé leur exécution, puisque ces sections modifient les données accédées par $S7$.

Ceci mène donc à certaines questions :

- **Comment pouvons-nous garantir qu'une ressource n'est pas modifiée au même moment par une autre instance parallèle ?**
- **Comment s'assurer que les différentes instances parallèle ont terminé le traitement d'une ressource avant qu'une instance dépendant de ce traitement puisse s'en servir ?**

C'est ce que tentent de résoudre les mécanismes de synchronisation, que nous verrons à la section suivante.

Mécanismes de synchronisation

Mécanismes de synchronisation

Les mécanismes de synchronisation servent à contrôler l'exécution d'instances parallèles au sein du code afin de décider quand ces dernières peuvent s'exécuter ou doivent arrêter. On s'en sert notamment pour garantir que deux instances parallèles d'une même tâches n'iront pas modifier la même information au même moment, ou qu'une instance parallèle d'une tâche n'accèdera pas à de l'information alors que celle-ci est modifiée ou n'a pas finie d'être calculée.

Il existe une multitude de mécanismes de synchronisation. Dans le cadre du cours, nous étudierons les deux mécanismes de synchronisation les plus courants :

- Le Mutex
- Le Sémaphore

Mécanismes de synchronisation

Les mécanismes de synchronisation servent très souvent à délimiter des sections de code où l'information est accédée où modifiée. Ces sections de code sont appellées **sections critiques** car l'information modifiée ou lue dans cette section de code ne doit pas être modifiée simultanément par une autre instance parallèle.

Autrement dit, il faut s'assurer qu'à un moment précis une seule instance d'exécution parallèle peut accéder à une seule région critique parmi toutes les régions critiques qui partagent les mêmes ressources.

Si on garanti l'accès unique à une région critique, pour une certaine ressource, on garanti que la ressource n'est pas accédée pendant modification, ou modifiée à plusieurs endroits simultanément, ce qui rempli les conditions de Bernstein.

Mécanismes de synchronisation

Si on reprend notre exemple du guichet automatique, une personne ne pouvoir retirer du compte bancaire tant qu'une autre personne sur le même compte bancaire n'a pas terminé la transaction, afin d'éviter le problème observé plus tôt.

```
void Retrait(int NumeroCompte, float MontantRetrait)
{
//DÉBUT DE LA SECTION CRITIQUE POUR NumeroCompte

    float BalanceCompte = RequeteBanque(GET_BALANC,NumeroCompte);
    BalanceCompte = BalanceCompte - MontantRetrait;
    RequeteBanque(SET_BALANCE,NumeroCompte,NouvelleBalance);

//FIN DE LA SECTION CRITIQUE POUR NumeroCompte

    //Le guichet fait sortir l'argent ici.
}
```

Mécanismes de synchronisation

Étant donné la section critique en exemple à l'acétate précédente, une façon d'éviter que deux guichets exécutent ce code au même moment pour le même compte bancaire serait de mettre en place une sorte de verrou sur le code. Le verrou empêcherait un second utilisateur accédant au même compte d'exécuter la même section de code simultanément, ce qui règlerait le problème de concurrence précédemment observé.

Un tel verrou existe et s'appelle un **mutex**. Nous verrons comment fonctionne un mutex dans les acétates qui suivent.

Mécanismes de synchronisation

Mutex :

Un mutex est un type d'objet permettant un accès mutuellement exclusif (d'où le nom mutex) à une ressource ou une section de code utilisant cette ressource.

La particularité d'un mutex est que son état courant est **global à toutes les instances parallèles**.

Un mutex possède deux états différents, soit l'état **verrouillé** et l'état **déverrouillé**. Deux actions sont aussi possibles au niveau du mutex :

- **Verrouiller :** Acquiert le verrou si le mutex n'était pas verrouillé, sinon l'exécution du code pour l'instance parallèle courante arrête tant que le mutex n'est pas déverrouillé.
- **Libérer :** Libère le verrou du mutex. Le mutex devient déverrouillé et peut être verrouillé à nouveau.

Mécanismes de synchronisation

Mutex (suite) :

Le fonctionnement du mutex est le suivant. Si une instance parallèle de code s'exécute et tente de verrouiller un mutex qui n'est pas déjà verrouillé, l'instance obtient le verrou et continue de s'exécuter.

Si une autre instance parallèle tente par la suite de verrouiller le mutex alors qu'il est déjà verrouillé, elle pausera automatiquement son exécution, tant et aussi longtemps que le mutex n'est pas libéré (déverrouillé) par l'instance parallèle l'ayant verrouillé. Une fois le mutex libéré par l'autre instance, le mutex obtient le verrou et l'éxécution pour cette instance se poursuit.

Lorsqu'on verrouille un mutex, il faut donc s'assurer qu'on le libère lorsque la section critique de code l'utilisant a terminé son exécution.

Mécanismes de synchronisation

Mutex (suite) :

Si on revient à notre exemple de guichet automatique, mais que cette fois on utilise un mutex pour s'assurer qu'aucune autre personne ne puisse exécuter la section critique de code en même temps que nous, on aurait le code suivant :

```
void Retrait(int NumeroCompte, float MontantRetrait)
{
    [Verrouiller Mutex] ObjetMutex.verrouiller();

    [Get Balance]       float BalanceCompte = RequeteBanque(GET_BALANC,NumeroCompte);
    [Nouvelle Balance]  BalanceCompte = BalanceCompte - MontantRetrait;
    [Set Balance ]      RequeteBanque(SET_BALANCE,NumeroCompte,NouvelleBalance);

    [Libérer Mutex]     ObjetMutex.liberer();

    //Le guichet fait sortir l'argent ici.
}
```

Mécanismes de synchronisation

Mutex (suite) :

Si on reprend l'exemple précédent de nos deux individus, mais cette fois-ci avec le mutex, on aurait la séquence d'action suivante :

temps	**I1**	**I2**
t_1	[Verrouiller Mutex] (Obtient le verrou)	
t_2	[Get Balance] (BalanceCompte = 1000)	
t_3	[Nouvelle Balance] (BalanceCompte = 950)	
t_4		[Verrouiller Mutex] (Attente du verrou)
t_5	[Set Balance] (950$ dans le compte)	
t_6	[Libérer Mutex] (Libère le verrou)	
t_7		(Obtient le verrou)
t_8		[Get Balance] (BalanceCompte = 950)
t_9		[Nouvelle Balance] (BalanceCompte = 900)
t_{10}		[Set Balance] (900$ dans le compte)
t_{11}		[Libérer Mutex] (Libère le verrou)

Mécanismes de synchronisation

Mutex (exemple 2) :

Comme deuxième exemple, regardons un système où nous avons 2 modules s'exécutant en parallèle à des vitesses différentes. La première partie du système est un module d'acquisition d'image (caméra) captant entre 5 et 10 images par seconde. La seconde partie du système est un module de traitement d'image analysant 4 à 6 images par seconde.

On souhaite inscrire les images captées par le module caméra dans une zone *ImageTampon* en mémoire. On copie ensuite *ImageTampon* dans une image temporaire pour faire l'analyse de notre image. Il faut donc s'assurer de ne pas inscrire l'image en mémoire au niveau du module de caméra pendant que le module d'analyse copie l'image dans l'image temporaire d'analyse.

On doit donc s'assurer que seulement un module accède à l'image à la fois. Pour ce faire, on peut utiliser un mutex.

Mécanismes de synchronisation

Mutex (exemple 2, suite) :

```
//Fonction exécutée dans l'instance parallèle 1
void ModuleCamera()
{
    while(true) //On acquiert des images sans arrêt
    {
        Image ImageTemporaire = Camera.acquerirImage();

        MutexImageTampon.verrouiller();
        ImageTampon = ImageTemporaire;
        MutexImageTampon.liberer();
    }
}

//Fonction exécutée dans l'instance parallèle 2
void ModuleTraitement()
{
    while(true) //On analyse des images sans arrêt
    {
        Image ImageTraitement;

        MutexImageTampon.verrouiller();
        ImageTraitement = ImageTampon;
        MutexImageTampon.liberer();

        AnalyserImage(ImageTraitement);
    }
}
```

Mécanismes de synchronisation

Sémaphore :

Le second mécanisme que nous étudierons est une généralisation du mutex, soit le **sémaphore**. Globalement, un sémaphore est un mutex pouvant avoir un nombre de verrou supérieur à 1.

Lorsqu'un sémaphore est verrouillé, il est possible de spécifier combien de verrous il tente de verrouiller. Si le sémaphore possède un nombre de verrous libres supérieur ou égal au nombre de verrous demandés, il sera verrouillé et l'instance parallèle continuera son exécution. Sinon, le code sera en attente tant et aussi longtemps que le sémaphore n'aura pas le nombre de verrous demandés de disponible.

Mécanismes de synchronisation

Sémaphore (suite) :

Deux opérations sont possibles avec le sémaphore :

- Vérrouiller(n) : Tente de verrouiller n verrous du sémaphore. Si le sémaphore ne possède pas n verrous ou plus, le code est en attente tant que n verrous ne sont pas disponibles.
- Libérer(n) : Libère n verrous du sémaphore. Il est possible de libérer plus de verrous qu'on en avait verrouillés, ou de libérer des verrous sans en avoir verrouillé.

Mécanismes de synchronisation

Sémaphore (exemple) :

Pour notre premier exemple impliquant des sémaphores, imaginons un système d'analyse d'image divisant chaque image en 16 tuiles. Chaque tuile est traitée de manière indépendante dans une instance parallèle. Une fois que chaque tuile est traitée, l'algorithme prend les 16 tuiles et effectue un second traitement sur ces dernières après les avoir regroupées.

L'algorithme ne doit pas exécuter son second traitement tant que les 16 tuiles n'ont pas été analysées, il doit donc attendre que l'exécution des 16 analyses de tuile soit terminée avant de procéder à son propre algorithme.

Mécanismes de synchronisation

Sémaphore (exemple,suite) :

```
void AlgorithmeTuile(Tuile aTuile)
{
    FaireAnalyse(aTuile);

    //On libère 1 verrou parce que la tuile a terminé
    //d'être traitée. Lorsque chaque tuile aura libéré
    //son verrou, il y aura 16 verrous de libres.
    SemaphoreTuile.libérer(1);
}

void Main(Image aImage);
{
    for(int i = 0; i < 4; ++i)
    {
        for(int j = 0; j < 4; ++j)
        {
            CréerInstanceParallèle(AlgorithmeTuile(aImage.GénérerTuile));
        }
    }

    //Le code bloque ici tant que nos 16 instances parallèle
    //s'occupant de traiter des tuiles d'image n'ont pas
    //terminé de s'exécuter.
    SemaphoreTuile.verrouiller(16);

    FaireTraitementFinal(aImage);
}
```

Mécanismes de synchronisation :

Sémaphore (exemple 2) :

Dans un autre exemple, on souhaite faire un système de barbecue robotisé. Le système s'occupe de griller des steaks, et le barbecue peut faire cuire 10 steaks en même temps.

Notre barbecue automatisé possède un premier bras robotisé mettant 2 steaks crus à la fois sur le gril, et un second bras robotisé pouvant enlever 1 steak à la fois lorsqu'il est cuit. De plus, seulement 1 bras peut bouger à la fois, pour éviter que les bras ne s'entremêlent pendant les manipulations.

La cuisson des steaks est bien sûr externe au système, qui ne bénéficie que d'un capteur indiquant si un steak précis est cuit ou non.

On souhaite programmer le fonctionnement de ce barbecue, en s'assurant que ce dernier garde toujours le plus de steaks possible sur le barbecue.

Mécanismes de synchronisation

Sémaphore (exemple 2) :

```
void BrasSteakCrus()
{
    while(true)
    {
            //On bloque tant qu'il n'y a pas 2 places
            //de libres sur le barbecue car notre bras
            // mets 2 steaks à la fois.
            SemaphorePlaceBBQ.verrouiller(2);

            //Si deux places sont libres, on attend
            //que l'autre bras ne soit pas en train
            //de travailler avant de commencer à
            //mettre les steaks.
            MutexPermissionBras.verrouiller();

            //On place les steaks.
            PlacerDeuxSteakCrusSurGrill();

            //On indique que nous avons fini de
            // travailler avec le bras et qu'un
            // autre bras peut travailler.
            MutexPermissionBras.libérer();
    }
}
```

```
void BrasSteakCuit()
{
    while(true)
    {
        for(int i = 0; i < 10; ++i)
        {
            if(Steak[i].Etat == cuit)
            {
                //Si un steak est cuit,on s'assure que l'autre
                //bras n'est pas en train de travailler avant
                //de sortir le steak du bbq.
                MutexPermissionBras.verrouiller();

                EnleverUnSteakCuit();

                //On indique que nous avons fini
                //de travailler avec le bras
                //et qu'un autre bras peut travailler.
                MutexPermissionBras.libérer();
                //On libère 1 place dans le barbecue
                //car on vient d'enlever un steak.
                SemaphorePlaceBBQ.libérer(1);
            }
        }
    }
}
```

Mécanismes de synchronisation

Sémaphores (exemple 2,suite) :

```
void Main()
{
    SemaphorePlaceBBQ.libérer(10);

    DémarrerInstanceParallèle(BrasSteakCrus());
    DémarrerInstanceParallèle(BrasSteakCuit());
}
```

Inter-blocage

Pour terminer, notons qu'il est important, lors de l'utilisation de mécanismes de synchronisation, d'éviter les situations d'inter-blocage.

Un inter-blocage survient lorsque toutes les instances parallèles d'une application ont cessé leur exécution car elles sont en attente d'un verrou. Dans un tel cas, l'application cesse de s'exécuter puisque toutes les instances parallèles de cette dernière sont en attente qu'un verrou se libère, alors qu'il est déjà possédé par une autre instance qui attend elle-même qu'un verrou soit libéré.

Inter-blocage

Un exemple d'inter-blocage serait le suivant :

```
void Fonction1()
{
    while(true)
    {
        Mutex1.verrouiller();
        Mutex2.verrouiller();

        Mutex2.libérer();
        Mutex1.libérer();
    }
}
```

```
void Fonction2()
{
    while(true)
    {
        Mutex2.verrouiller();
        Mutex1.verrouiller();

        Mutex1.libérer();
        Mutex2.libérer();
    }
}
```

Tôt ou tard, lors de l'exécution, la Fonction1 attendra après le Mutex2, verrouillé par la Fonction2, qui elle attend après le Mutex1, verrouillé par la Fonction1. Il y aura à ce moment inter-blocage.

Fils d'exécution et processus

Fils d'exécution et processus

Pour paralléliser une application, on doit séparer son exécution sur plusieurs instances parallèles différentes. Il existe différentes façons de paralléliser une application à cet égard.

- En créant des nouveaux processus.
- En créant des fils d'exécution.
- En séparant l'exécution sur des machines physiques séparées à travers un réseau.
- etc.

Pour cette section, nous étudierons spécifiquement les fils d'exécution et les processus.

Fils d'exécution et processus

Fils d'exécution :

Par défaut, un programe s'exécutant possède toujours au moins un fil d'exécution (*thread* en anglais). Il est cependant possible de créer des fils d'exécution supplémentaires pour un même programme.

Chaque fil d'exécution exécute du code indépendemment d'un autre fil d'exécution.

La mémoire sur la pile (stack memory) est propre à un fil d'exécution en particulier, mais la mémoire dynamique (heap memory) est commune à tous les fils d'exécution d'un même programme.

Un fil d'exécution est la méthode la plus rapide pour paralléliser un programme dans la mesure où le coût pour communiquer entre deux fils d'exécution, initialiser un nouveau fil d'exécution ou le détruire est très bas.

Fils d'exécution et processus

L'initialisation d'un nouveau fil d'exécution dépend du système d'exploitation. Dans bien des cas, la création d'un nouveau fil d'exécution prendra un pointeur de fonction en paramètre. La fonction pointée sera appellée dans un nouveau fil d'exécution et sera exécutée en parallèle, indépendemment des autres fils d'exécution.

Sous Unix : Posix Thread - pthread_create(...)

Sous Windows : WINAPI Thread - CreateThread(...)

Dans un contexte d'interaction en temps réel, on utilisera habituellement plusieurs fils d'exécution en parallèle, étant donné leur performance et leur faible surchage de coût.

Fils d'exécution et processus

Les fils d'exécution sont habituellement utilisés dans le cas d'applications, possédant différents modules effectuant des calculs séparés mais qui dépendent de l'information transmise des autres modules. Un tel cas survient généralement lorsque les différents modules de l'application doivent partager ou échanger une quantité importante d'information de manière continue.

Un exemple d'utilisation de fils d'exécution serait un jeu interactif 3D. Un tel jeu aura habituellement un fil d'exécution dédié pour chaque type de calcul (physique, audio, rendu 3D, intelligence artificielle, etc.) que doit effectuer le jeu. Les différents fils d'exécution communiquent entre eux, de manière à se mettre à jour en fonction des différents évènements survenant dans le jeu.

Fils d'exécution et processus

Processus :
Lorsqu'un programme est exécuté dans un système d'exploitation, ses fils d'exécution sont exécutés au sein d'un processus. La totalité des fils d'exécutions de ce programme, ainsi que la mémoire qu'ils utilisent, font tous parti du même processus.

Lorsqu'un processus est arrêté pendant ou à la fin de son exécution, tous les fils d'exécution qu'il hébergeait et toute la mémoire que le processus utilisait sont automatiquement libérés par le système d'exploitation.

Un système d'exploitation exécute et synchronise plusieurs processus en parallèle. Chaque processus possède sa propre priorité et obtient un temps de calcul plus ou moins grand sur le (ou les) processeur en fonction de cette priorité.

Une approche pour paralléliser une application est de générer des processus différents, s'exécutant en parallèle, pour chaque module de notre application.

Fils d'exécution et processus

Lorsqu'un programme crée un nouveau processus, le processus est exécuté parallèlement à ce dernier et de manière indépendante. Le processus ainsi créé possède son propre espace de mémoire (pile et dynamique), et ne partage pas d'information avec les autre processus de façon implicite.

La communication entre plusieurs processus utilise des mécanismes de communication présents dans le système d'exploitation :

- Passage de messages
- Mémoire spéciale explicitement partagée entre les processus.

L'avantage de séparer notre applications en multiples processus est qu'un processus peut exister sans que les autres processus soient nécessairement en exécution. Le désavantage vient de la surcharge de coût importante lors de la communication entre les processus.

Fils d'exécution et processus

L'utilisation de processus serait justifiée dans un contexte où une application se divise de façon naturelle en plusieurs modules, et que les modules peuvent fonctionner indépendemment les uns des autres. De telles situations surviennent lorsque les différents modules doivent s'échanger peu de données ou qu'ils ne dépendent pas des données des autres modules pour bien fonctionner.

Un exemple classique de parallélisme avec plusieurs processus est l'hébergement d'une base de donnée locale sur un ordinateur. La base de donnée sera habituellement dans un processus s'occupant exclusivement de la gestion de cette dernière. Les logiciels locaux utilisant la base de donnée sont quant à eux démarrés sur un processus séparé et communiquent avec cette dernière par échange de message (les requêtes SQL dans ce cas-ci).

Fils d'exécution et processus

Un autre exemple d'une application parallèle divisée en processus est la table de mixage interactive μSynth développée en 2008 dans le cadre d'un projet en imagerie.

La table est exécutée sur 3 processus, soit un processus effectuant le rendu graphique, un processus s'occupant du traitement et du rendu audio et un processus s'occupant de l'analyse d'image. Chaque processus fonctionne indépendamment des autres et les modules peuvent fonctionner séparément sans les autre modules en simulant la communication par message ayant normalement lieu entre les modules.

De plus, puisqu'ils sont indépendants, chaque processus (module) peut être exécuté sur un ordinateur séparé plutôt que d'exécuter les trois modules sur le même ordinateur. On obtient ainsi un gain de performance substantiel, et la communication inter-processus (inter-module dans le cas ici) s'effectue alors à travers le réseau, en utilisant exactement les mêmes messages compris par chaque modules.

Pour terminer...

Comment paralléliser une application ?

- En divisant les tâches en parties indépendantes (précédence).
- En s'assurant que les tâches indépendantes s'exécutent correctement en concurrence (sections critiques, mécanismes de synchronisation).
- En tenant compte du fait que plusieurs processeurs travaillent en utilisant les mêmes informations (tenir compte des ressources partagées par les instances parallèle, soit de la concurrence).
- En tenant compte de la surcharge de calcul inhérente à la division d'une tâche en sous-tâches.
- En choisissant le modèle de parallélisation adapté à la tâche (thread, processus, many core ?)

Références suggérées :

Livre A. Silberschatz, P.B. Galvin, and G. Gagne, *Operating System Concepts.*, Wiley, Ltd., Dec 2004.

Livre G. R. Andrews., *"Concurrent programming : principles and practice."*, Benjamin-Cummings Publishing Co. Inc., Redwood City, CA, USA, 1991.