# *Multiprogrammation*

## Table des matières


Introduction....3
Apport des systèmes d'exploitation des années 70....3
Époque des années 80 à aujourd'hui....4
Système d'exploitation: Rôle et Utilité....5
Systèmes d'exploitation multitâches....6
Systèmes multi-processeurs....6
Les types de systèmes d'exploitation....6
Programmation multitâche....8
Introduction....8
Notion de processus....8
Les états d'un processus....8
Qu'est-ce qu'un « thread » ?....10
Dans quel cas utilise-t-on un thread ?....10
Appel de fonctions versus threads....11
Création de thread avec C++....13
Création d'un thread (classe thread)....13
Exemple 1 (Programme de la fonction de la page 11 sous forme de thread)....13
Utilisation de "join" pour attendre la fin d'un thread....14
Exemple 2 (Passage d'un paramètre au thread)....15
Exemple 3 (Passage d'un méthode d'une classe ou d'une structure)....17
Accès concurrents et communication entre thread....18
Définition d'un accès concurrent....18
Exemple de problème d'accès concurrents....18
Exclusion mutuelle par attente active....20
1ère méthode : Masquage des interruptions....20
2ème méthode : Variable de vérouillage....20
3ème méthode : L'instruction TSL....22
Gestion des accès concurrents en C++11....25
mutex....25
Exemple 1 : En utilisant une classe....25
Exemple 2 : En utilisant une fonction et une variable globale....29
Usage de std::lock_guard....30
Communication entre threads....31
Variables de condition....31
Création d'une variable de condition....32
Exemple : Attente d'un signal par le main et envoie de ce signal par un thread....32
Exemple 2 : Attente d'un signal par un thread et envoie de ce signal par un autre thread....35

# Introduction

La multiprogrammation est un terme qui désigne tout type de programme faisant appel à des fonctions permettant la programmation multitâche ou parallèle.

Multiprogrammation : Concept permettant de faire tourner un ou plusieurs autres programmes quand un autre programme attend quelque chose (comme la libération d'un périphérique). **Définition**

En fait, la multiprogrammation est plutôt la première forme de multitâche qui fût implanté dans le début des années 1970.

## Apport des systèmes d'exploitation des années 70.

- **Multiprogrammation**: On partitionne la mémoire pour pouvoir exécuter plusieurs tâches. C'est une solution au temps d'attente du processeur lors d'entrées/sorties.
- **Spool**: Charge et exécute une autre tâche dès qu'une partition se libère. (L'ancêtre du multi-tâche d'aujourd'hui)
- **Partage de temps**: Temps de réponse plus rapide. Chaque usager possède maintenant un terminal.
- 1er système à temps partagé: **MULTICS (MULT**iplexed **I**nformation and **C**omputing **S**ervice)
- Développement des mini-ordinateurs: famille des PDP 1-7-11

2 systèmes d'exploitation :
1. **MULTICS**
2. **UNIX**

- Développement sur un PDP-7 de **UNICS** (**UN**iplexed **I**nformation and **C**omputing **S**ervice) qui devint par la suite **UNIX. Unics, son vrai nom original,** était à l'époque un système d'exploitation **mono-usager**. C'est aussi l'époque où est apparue le langage C.

## Époque des années 80 à aujourd'hui

C'est l'époque des circuits LSI *(Large Scale Integration)* et VLSI *(Very Large Scale Integration)*. Les puces contiennent maintenant des centaines de milliers de transistors.

Cette formidable miniaturisation des composantes a rendu possible la mise en marché d'ordinateurs personnels *(Personnal Computer, PC)*.

Aux langages de contrôle compliqués de la troisième génération (ex: JCL pour OS/360) ont succédé des logiciels conviviaux et des interfaces usagers graphiques (ex: MacIntosh, Windows, Presentation Manager).

Deux système d'exploitation dominent:

- MSDOS
- UNIX

La quatrième génération d'ordinateurs voit aussi la prolifération des systèmes d'exploitation en réseau (ex: Novell Netware, Banyan).

Dans ce type de système, chaque ordinateur fonctionne avec son propre système d'exploitation, mais l'utilisateur peut se connecter à une machine distante et transférer des fichiers.

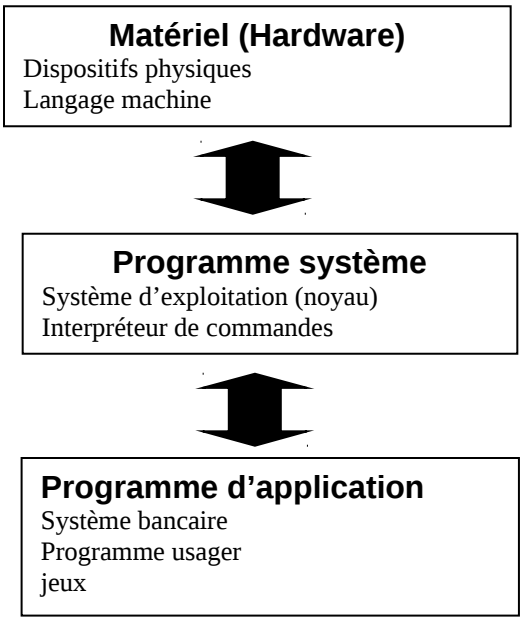
## Système d'exploitation: Rôle et Utilité

Sans ses logiciels, un ordinateur n'est qu'un morceau de métal inutile. Les logiciels se répartissent en 2 grandes catégories :

1. Programme système : Ensemble de programme qui permet de gérer les principales fonctions de l'ordinateur.
2. Programme d'applications : Ensemble de programme qui effectue les principales tâches de l'usager.

Le programme système le plus fondamental est le ***système d'exploitation.***

Le **système d'exploitation** se situe dans la couche « **Programme système** ». Il y a 3 grandes couches qui caractérisent les ordinateurs. La figure ci-contre illustre ces différentes couches.



## Systèmes d'exploitation multitâches

Les systèmes d'exploitation multitâches permettent de partager le temps du processeur entre plusieurs programmes, ainsi ceux-ci sembleront s'exécuter simultanément.

Pour réaliser ce processus, les applications sont découpées en séquence d'instructions que l'on appelle tâches ou processus. Ces tâches seront tour à tour actives, en attente, suspendues ou détruites, suivant la priorité qui leur est associée .

Un système est dit préemptif lorsqu'il possède un ordonnanceur (aussi appelé *planificateur*), qui répartit, selon des critères de priorité le temps machine entre les différentes tâches qui en font la demande.

Le système est dit à **temps partagé** lorsqu'un quota de temps est alloué à chaque processus par l'ordonnanceur. C'est notamment le cas des systèmes multi-utilisateurs qui permettent à plusieurs utilisateurs d'utiliser simultanément sur une même machine des applications différentes ou bien similaires : le système est alors dit "**système transactionnel**". Pour ce faire, le système alloue à chaque utilisateur une tranche de temps.

### Systèmes multi-processeurs

Ces systèmes sont nécessairement multitâches puisqu'on leur demande d'une part de pouvoir exécuter simultanément plusieurs applications, mais surtout d'organiser leur exécution sur les différents processeurs (qui peuvent être identiques ou non). Ces systèmes peuvent être soit architecturés autour d'un processeur central qui coordonne les autres processeurs, soit avec des processeurs indépendants qui possèdent chacun leur système d'exploitation, ce qui leur vaut de communiquer entre eux par l'intermédiaire de protocoles.

### Les types de systèmes d'exploitation

On distingue plusieurs types de systèmes d'exploitation, selon qu'ils sont capables de gérer simultanément des informations d'une longueur de 16 bits, 32 bits, 64 bits ou plus.

| Système | Codage | Mono-utilisateur | Multi-utilisateur | Mono-tâche | Multitâche |
|---|---|---|---|---|---|
| DOS | 16 bits | X | | X | Aucun |
| Windows3.1 | 16/32 bits | X | | | Coopératif |
| Windows95/98/Me | 32 bits | X | | | préemptif |
| WindowsNT/2000 | 32 bits | | X | | préemptif |
| MAC/OS X | 32 bits | | X | | préemptif |
| VMS | 32 bits | | X | | préemptif |
| Unix/Linux | 32 / 64 bits | | X | | préemptif |
| Windows Vista/7 | 32 / 64 bits | | X | | préemptif |

Le terme préemptif : Se dit d'un système d'exploitation multitâche lorsque celui-ci peut arrêter à tout moment n'importe quelle application pour passer le temps processeur à une autre application.

| | |
|---|---|
| Le terme coopératif : | Se dit d'un système d'exploitation multitâche quand ce dernier doit attendre qu'une application lui rende le processeur avant de pouvoir donner celui-ci à un autre programme. |

Quelques définitions :

Multitâche :

Un SE est dit multitâche lorsqu'il permet de faire fonctionner plusieurs applications simultanément. Exemples : Amiga DOS, Solaris, OS/2, Unix. Contraire : MS-DOS, Windows 3.1, Windows 95

SMP[1] :

Symmetric MultiProcessing. Système multiprocesseur distribuant symétriquement les tâches entre différents processeurs partageant une mémoire commune, en s'assurant qu'ils ne vont pas se mettre à écrire tous à la même adresse en même temps. Opposé à ASMP et à MPP. Le principe est apparu au milieu des années 1970.

Temps réel [2]:

Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés.

[1] Source: http://www.linux-france.org/prj/jargonf/index2.html
[2] Source: http://fr.wikipedia.org

# Programmation multitâche

Dans cette section, nous discuterons de la réalisation des threads. Nous y traiterons de :

- Notion de processus et threads (processus « lourd » vs processus « léger »;
- Création de threads avec C++;
- Passage de paramètres aux threads;
- Gestion des accès concurrents et accès aux ressources communes.
- Communication entre les threads.

## Introduction

### Notion de processus

Un processus est habituellement, sous sa représentation la plus simple, un programme en exécution. Autrement dit tous programmes, utilitaires ou commandes que vous envoyez au système devient un processus.

Un processus possède, entre autre, :

- son propre espace d'adressage mémoire qui est indépendant des autres processus en mémoire (ceci a une conséquence sur les threads que nous verrons plus loin);
- sa pile d'exécution (stack);
- son tas (heap);
- et ses registres d'état.

### Les états d'un processus

Lorsqu'un processus n'a pas toutes les ressources dont il a besoin pour s'exécuter, il est nécessaire de le bloquer en attendant que ces ressources soient disponibles. La figure ci-dessous montre les différents états que peut prendre un processus, lorsqu'il existe. Lors de la création il est mis, en général, dans l'état bloqué, en attendant qu'il ait toutes les ressources dont il a besoin initialement. Sa destruction peut subvenir dans n'importe quel état à la suite d'une décision interne s'il est actif, ou externe s'il est dans un autre état. Dans ce cas, il faut récupérer toutes les ressources qu'il possédait.

Il est assez naturel de considérer de façon particulière la ressource processeur physique. Pour le moment, c'est encore une ressource chère, qu'il vaut mieux ne pas gaspiller. Par ailleurs, il est inutile de la donner à un processus à qui il manque une autre ressource, puisque ce processus ne peut alors évoluer: le processeur ne pourrait que constater ce manque et attendre la disponibilité de cette ressource (c'est ce que l'on appelle l'attente active). C'est pourquoi on distingue trois états:

- l'*état actif* où le processus dispose de toutes les ressources dont il a besoin,
- l'*état bloqué* où le processus a besoin d'au moins une ressource autre que le processeur physique,
- l'*état prêt* où le processus dispose de toutes les ressources à l'exception du processeur physique.

La transition 1 survient lorsque, le processus étant actif, il exprime le besoin de disposer d'une nouvelle ressource. Cette expression peut être explicite, sous forme d'une demande au système, ou implicite, sous forme d'un accès à cette ressource non allouée. Le processus doit évidemment disposer du processeur pour pouvoir exprimer ce besoin. Il peut se faire que la transition vers l'état bloqué par manque de ressources, entraîne que le système lui retire d'autres ressources qu'il possédait, en plus de la ressource processeur physique, comme par exemple de la mémoire centrale. En général, on ne lui retire que des ressources chères, qu'il est relativement facile et peu coûteux de lui restituer ultérieurement, dans l'état où elles étaient au moment où on les lui a enlevées. L'état bloqué correspond alors au manque d'une ressource autre que le processeur physique, et l'état prêt correspond alors au manque de la seule ressource processeur.

La transition 4 est la conséquence d'un événement extérieur au processus. Celui-ci était bloqué en attente d'une ou plusieurs ressources. Lorsque le système constate qu'il a pu lui allouer toutes les ressources (autres que le processeur) dont il a besoin, il le fait passer dans l'état prêt. La disponibilité de la ressource attendue se présente, en général, sous l'une des deux formes suivantes:

- C'est le résultat d'une action par un autre processus. Il peut s'agir de la libération de la ressource, ou de la création de la ressource elle-même. Ainsi la fin d'exécution d'un processus crée la ressource événement attendue par son père.
- C'est le résultat d'une action extérieure. Il peut s'agir d'une interruption en provenance d'un périphérique, par exemple, ou d'un événement déclenché par l'opérateur ou un utilisateur.

Les transitions 2 et 3 sont sous la responsabilité de la partie du système qui alloue le processeur physique. Certains systèmes n'implantent pas la transition 2. Il s'ensuit que lorsqu'un processus est actif, il le reste jusqu'à ce qu'il soit terminé ou qu'il ait besoin d'une ressource qu'il n'a pas. L'allocation du processeur consiste alors à choisir un processus parmi les processus prêts, et à le faire passer dans l'état actif. Cela peut entraîner que les processus prêts peuvent attendre très longtemps pour devenir actif, si le choix a conduit à rendre actif un processus qui calcule sans exprimer le besoin de nouvelles ressources. Ceci peut être évité par le biais de la transition 2 qui doit être la conséquence d'une action extérieure au processus actif. C'est en général un des rôles attribués à l'interruption d'horloge de déclencher cette transition.

## Qu'est-ce qu'un « thread » ?

Un thread est une unité d'exécution faisant partie d'un processus (programme). C'est une unité d'exécution indépendante du programme principal qui possède :

- sa propre pile,
- ses états des registres CPU et
- un espace d'adressage commun avec le processus principal.

Un thread partage toutes les ressources du processus principal y compris ses variables globales et ses descripteur de fichers.

Par exemple, la mémoire et les fichiers sont deux ressources.

Un thread consiste en :

- un ou plusieurs autres threads,
- son code,
- ses données (sa propre pile) et
- ses autres ressources en mémoire.

Un programme s'exécute lorsque l'ordonnanceur le choisi pour s'exécuter dans le CPU. C'est l'ordonnanceur qui détermine les threads qui doivent s'exécuter et le moment de leur exécution.

Chaque thread dans un programme s'exécute de façon indépendante des autres threads. Ceux-ci s'exécutent individuellement et ne sont pas au courant de l'existence des autres threads dans le programme.

## Dans quel cas utilise-t-on un thread ?

Les threads sont utiles lorsque vous devez programmer des tâches qui peuvent être exécutées en arrière-plan (background). Ils permettent aussi de meilleures temps de réponse car les temps liés aux changements de contexte sont généralement minimal puisque les threads partagent le même espace d'adressage mémoire. À cet égard, on les appelle souvent les « processus légers » (lightweight process en anglais).

Exemple d'utilisation:

Les threads peuvent être utile dans le cas où vous voulez que la tâche qui s'exécute ne ralentisse pas le temps de réponse de l'application ou lorsque vous ne voulez pas que l'usager attende trop longtemps après la fin d'une opération.

## Appel de fonctions versus threads

Un programme appelant une fonction traditionnelle pour exécuter une tâche :

```
#include <iostream>

using namespace std;


void Fonction();        //Prototype de la fonction qui exécutera la tâche

int main()
{

        cout << "Appel de la fonction ..." << endl;

        Fonction();

        sleep(3);
        cout << "Je continue l'exécution du programme principal..." << endl;
        sleep(5);

        return 0;
}


void Fonction()
{

        cout << "Je suis dans la fonction... " << endl;
        sleep(6);

        cout << "La fonction termine son travail..." << endl;
}
```

La fonction exécute ce qui est demandé sans problème mais dû la nature même de la programmation procédurale, le programme principal ne peut continuer son exécution tant et aussi longtemps que la fonction n'aura pas terminé son exécution. Dans certains cas, il est souhaitable que le programme principal ou le processus principal puisse continuer son traitement pendant que la fonction exécute ses instructions. La séquence procédurale pose donc problème dans certains cas. Voici un exemple:

Un programme doit constamment vérifier la température d'un certain mélange de produits. Imaginez un instant qu'une fonction doit être appelée pour ajouter un produit au mélange.

Qu'arrive-t-il ?

| | |
|---|---|
| Programme principal surveille la température | |
| On doit ajouter un produit (appel à la fonction AjouteProduit() ) | |
| Aucune surveillance de température | Début de la fonction AjouteProduit() |
| Programme principal contnue sa surveillance de température | |

Comme vous l'avez constaté, ce schéma d'exécution ne permet pas de garantir la surveillance de température de façon constante et continue. Il en résulterait peut-être des dégâts ou un mauvais mélange qui pourrait affecter la chaîne de montage au complet. Nous avons donc besoin d'un mécanisme qui nous assurera qu'une tâche puisse continuer de s'exécuter « en même temps » qu'une autre. Il faut donc viser le schéma suivant :

| | |
|---|---|
| Programme principal surveille la température | |
| On doit ajouter un produit (appel fonction AjouteProduit() ) | |
| Surveillance de température continue<br>Surveillance de température continue<br>Surveillance de température continue | Début de la fonction AjouteProduit() |
| Programme principal continue la surveillance de température | |

Le moyen d'y arriver est de recourir à la notion de thread. On transformera la fonction « AjouteProduit » en thread pour permettre une exécution « parallèle » avec le programme principal.

# Création de thread avec C++

## Création d'un thread (classe thread)

La classe « thread » permet la création d'un objet thread à l'intérieur de l'espace mémoire du processus appelant. Cette classe est définie comme suit:

Entête: <thread>

La classe "thread" représente une unité d'exécution indépendante du programme principal. Permet à plusieurs portion de code de s'exécuter de façon asynchrone et simultané.

### Exemple 1 (Programme de la fonction de la page 11 sous forme de thread)

```
#include <thread>
#include <chrono>  // Pour les temps d'attente
#include <iostream>

using namespace std;
using namespace std::chrono;  // Va avec #include <chrono>

void  Fonction();     //Prototype de la fonction qui exécutera la tâche

int main()
{

         cout << "Le thread est parti..." << endl;

        thread MonThread(Fonction) ;  //On créé l'objet thread et on y associe le code de la
fonction

        this_thread::sleep_for(seconds(3));  // Temps attente de 3 secondes

        cout << "Prog. Principal: Je continue mon travail..." << endl;
        cout << "Prog. principal: J'attends la fin du thread..." << endl;


     return 0;
}

void Fonction()
{
        cout << "Thread: Je commence mon travail... " << endl;
        this_thread::sleep_for(seconds(6));
        cout << "Thread: J'ai termine mon travail..." << endl;

}
```

**#include <thread>**

Nécessaire pour déclarer les classes relatives aux threads.

**void Fonction( );  //Prototype de la fonction qui exécutera la tâche**

La fonction « thread » peut prendre plusieurs signatures, ce qui n'était pas le cas précédemment. Le programmeur a donc maintenant une plus grande flexibilité quand à l'écriture de la fonction. Cette dernière peut également accepter des paramètres ou retourner une valeur.

La fonction qui sert de thread

Le contenu de cette fonction n'a rien d'extraordinaire :

```
void Fonction()
{
        cout << "Je suis dans le thread... " << endl;
        this_thread::sleep_for(seconds(6));  //Simulation d'un temps d'exécution...
        cout << "J'ai termine ce que j'avais a faire dans le thread..." << endl;

}
```

Dans l'exemple précédent, le programme démarre le thread et avant même que le thread termine son exécution, le programme principal "plante" avec un message de terminaison anormale.

La raison vient du fait que le programme principal doit attendre la fin du thread qu'il a lancé avant de se terminer lui-même. Rappelez-vous qu'un thread partage les ressources et la mémoire avec son processus parent et donc à ce titre, si le processus principal meurt avant son thread, lorsque le thread accédera aux données partagées, celles-ci ne seront plus disponible pour le thread. Conséquence, le programme principal arrête brusquement.

## Utilisation de "join" pour attendre la fin d'un thread

Pour remédier à la situation précédente, nous devons dire au programme principal d'attendre la fin du thread avant de poursuivre.

Voici l'ajout que nous devons faire au programme précédent:

```
cout << "Je continue mon travail dans le programme principal ..." << endl;

MonThread.join();  // attente de la fin du thread

return 0;
```

Cette fois-ci, le programme poursuit sont exécution, attends patiemment la fin du thread et continue.

### Exemple 2 (Passage d'un paramètre au thread)

Dans certains cas, un thread doit recevoir des données provenant du programme appelant pour exécuter la tâche demandée. En guise d'exemple, un thread reçoit une donnée, exécute un certain calcul sur cette donnée et retourne le résultat.

Voici la syntaxe pour envoyer une donnée au thread. On suppose ici que la donnée est de type entière (integer).

```
// ... inclusions d'usage

void  FonctionThread(int Donnee);      //Prototype de la fonction qui exécutera la tâche

int main()
{

            cout << "Le thread est parti..." << endl;

           thread MonThread(FonctionThread, 5) ;
//On créé l'objet thread et on passe la valeur 5 en paramètre au thread
```

La fonction maintenant :

```
void FonctionThread(int Donnee)
{
          cout << "Thread: Je commence mon travail... " << endl;
          cout << "Voici le parametre recu: " << Donnee << endl;

}
```

Que faire si la donnée doit être modifiée par le thread ?

On peut avoir le réflexe de passer le paramètre par référence comme ci-dessous:

```
void  FonctionThread(int &Donnee);     //Prototype de la fonction qui exécutera la tâche

int main()
{
   int Valeur = 10;

   cout << "Le thread est parti..." << endl;

   thread MonThread(FonctionThread, Valeur) ;  //On créé l'objet thread et on y associe le code de la fonction

   this_thread::sleep_for(seconds(3));  // Temps attente de 3 secondes


  cout << "Prog. principal: J'attends la fin du thread..." << endl;
  MonThread.join();

   cout << "Prog. Principal: La valeur de la donnee maintenant: " << Valeur << endl;

    return 0;
}

void FonctionThread(int &Donnee)
{
            cout << "Thread: Je commence mon travail... " << endl;

            Donnee++;

            cout << "Le parametre dans le thread: " << Donnee << endl;

}
```

On obtient le résultat suivant:

Le thread est parti...
Thread: Je commence mon travail...
Le parametre dans le thread: 11
Prog. principal: J'attends la fin du thread...
Prog. Principal: La valeur de la donnee maintenant: 10
Appuyez sur une touche pour continuer...

On remarque que la valeur a bel et bien changé dans le thread mais la valeur de la variable dans le programme principal n'a pas changé.

On peut utiliser l'instruction-clé “ref” pour passer la donnée en référence.

Ainsi l'appel de la création du thread se fera ainsi:

```
thread MonThread(FonctionThread, ref(Valeur)) ;
```

On obtient maintenant le résultat attendu:
Le thread est parti...
Thread: Je commence mon travail...

Le parametre dans le thread: 11
Prog. Principal: La valeur de la donnee maintenant: 11

En fait, il est possible de passer n'importe quel type du langage à un thread. On peut également passer un objet d'une classe ou une structure. Voyons voir comment cela peut s'écrire.

### Exemple 3 (Passage d'un méthode d'une classe ou d'une structure)

```
#include <iostream>
#include <thread>

using namespace std;

struct Eleve
{
	int NumAdm;
	int Moyenne;
	int Session;
};

void Fonction(Eleve &eleve);

class Affichage
{
	int Valeur;
  public:
	Affichage():Valeur(10){};
	Affichage(int Val) { Valeur = Val;};
	int get()
	{
		return Valeur;
	}
	void Message()
	{
		cout<<"Bonjour"<<endl;
	}
	void Transformation()
	{
		Valeur = Valeur * Valeur;
	}
};

int main()
{
   Affichage X(5);
	Eleve MonEleve;

	thread MonThreadObjet(&Affichage::Transformation, &X);
	thread MonThreadStruct(Fonction, ref(MonEleve));

	MonThreadObjet.join();
	MonThreadStruct.join();
	cout << "Prog. principal: La valeur de la donnee membre: " << X.get() << endl;
	cout << "Prog. principal: La moyenne de l'eleve: " << MonEleve.Moyenne << endl;

}
void Fonction(Eleve &eleve)
{
	eleve.Moyenne = 90;
	eleve.NumAdm = 1001;
	eleve.Session = 1;
}
```

On obtient:
Prog. principal: La valeur de la donnee membre: 25
Prog. principal: La moyenne de l'eleve: 90

## Accès concurrents et communication entre thread

Une application informatique échange souvent des données et effectue des traitements sur ces données. Il arrive fréquemment que ces données soient communes à l'application ou au module. Ces données communes deviennent souvent la cible de problème de concurrence. De même l'accès aux ressources communes doit être protégé pour éviter les problèmes liés à la concurrence des modules ou des threads[3].

### Définition d'un accès concurrent

Dans un système monotâche, l'accès aux zones de données communes est imposé par la structure du programme; il est **séquentiel**. Dans un système multitâche, l'accès dépend de l'ordonnancement des tâches: il est donc généralement imprévisible.

**Dans un système multitâche, l'accès aux variables dépend de l'ordonnancement des tâches qui est imprévisible.**

Nous avons déjà discuté du fait que les threads possèdent une zone de mémoire commune avec le processus principal. Cet espace de mémoire partagé peut devenir la source de problèmes d'accès concurrents.

Un **accès concurrent** se produit lorsque deux tâches ou plus lisent ou écrivent des données partagées et où le résultat dépend de l'ordonnancement des processus.

### Exemple de problème d'accès concurrents

Au début des années 1970, on voulait faire une étude sur l'utilisation des terminaux dans un système à temps partagé. Pour chaque terminal du système, on lançait un programme qui comptabilisait le nombre de lignes tapées par l'usager. À chaque fois qu'un processus de surveillance détectait la touche "Enter", il incrémentait une variable globale nommé "NCR".

[3]Voir les incidents documentés sur le Therac-25

Les instructions relatives en assembleur à l'incrémentation d'une variable pourraient se traduire de la façon suivante:

```
LOAD  NCR    /* Charge le contenu de la variable NCR */
INC  NCR     /* Incrémente la variable */
STORE NCR  /* enregistre le contenu de la variable en mémoire */
```

Le problème de l'accès à une zone de données partagée est illustrée par l'exemple suivant:

À un moment donné, l'ordonnancement entraîne la séquence suivante:

(Supposez NCR égal à 2500)

Le thread X effectue : LOAD NCR
Le thread X effectue : INC NCR  //NCR vaut alors 2501

(à ce moment, le thread X perd le CPU car il a atteint la fin de sa tranche de temps et n'a pas le temps de sauvegarder le contenu de son registre...)

thread Y: LOAD NCR  // Le registre vaut toujours 2500 car on n'a pas eu le temps de l'enregistrer précédemment.
thread Y: INC NCR
thread Y: STORE NCR  // À ce point, NCR = 2501

(à ce moment, le thread Y perd le CPU et le thread X revient en mémoire)

//Le thread X continue le code où il s'était arrêté
thread X: STORE NCR  // À ce point, NCR = 2501

Malgré le fait que deux tâches aient voulu signaler un "Enter", la variable globale n'a été ***incrémentée que de 1 seulement***!

Dans l'exemple précédent, il y a eu *accès concurrent*. C'est à dire une situation où plusieurs tâches manipulent des données partagées. Il est impossible d'écrire un programme fiable de cette façon. Il faut absolument éviter que plusieurs tâches accèdent en même temps à des données partagées.

La portion de code d'une tâche qui accède à une ressource commune est appelée **section critique**.

Le principe de **l'exclusion mutuelle** empêche une tâche d'accéder à une ressource si celle-ci est déjà utilisée par un autre processus.

En général, une tâche effectue la majeure partie du temps des opérations qui ne manipulent pas des ressources partagées. Malgré tout, comment éviter les conflits d'accès ? Pour éviter les problèmes dans les situations qui mettent en oeuvre le partage de la mémoire, des fichiers et de tout autre objet, il faut trouver un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'une tâche (thread) à la fois. En d'autres termes, il nous faut une exclusion mutuelle qui empêche les autres processus d'accéder à un objet partagé si cet objet est en train d'être utilisé par une autre tâche.

Dans l'exemple précédent, le processus Y a utilisé une variable partagée avant que X ait fini de s'en servir. C'est de là que provient l'erreur.

## Exclusion mutuelle par attente active

Les méthodes suivantes permettent d'assurer qu'aucune autre tâche se trouvera dans une section critique d'une autre tâche. L'attente active provient du fait que la tâche doit attendre que la ressource se libère avant de l'utiliser.

### 1ère méthode : Masquage des interruptions

Il s'agit du moyen le plus simple d'assurer l'exclusion mutuelle: chaque processus masque (empêche les interruptions de se produire) avant d'entrer dans sa section critique et les restaure en sortant. Le processus courant garde alors le processeur pour toute la durée de sa section critique. Il y a cependant de sérieux inconvénients à cette solution:

1. Augmente le temps de réponse aux interruptions: les contraintes de temps reliées aux tâches matérielles pourraient ne plus être respectées.

2. Si le processus oublie de restaurer les interruptions, le système tombe.

### 2ème méthode : Variable de vérouillage

Il s'agit d'une solution purement logicielle. On utilise une variable partagée (un peut comme un cadenas) initialisée à zéro. Tout processus doit tester ce verrou avant d'entrer en section critique.

Le fonctionnement général est le suivant:

- Si le verrou est à 0, le processus le met à 1 et entre en section critique.
- Si le verrou est à 1, le processus attend qu'il repasse à 0.

Exemple en langage C:

```
int verrou = 0;
.
.
.
while(verrou == 1);  /* Attendre que le verrou passe à 0 */
verrou = 1;
/* Début de la section critique */
....
/* fin de la section critique */

verrou = 0;
.
.
.
```

Quel est le problème avec cette solution ? Pouvez-vous trouver un exemple qui ne fonctionne pas ?

### 3ème méthode : L'instruction TSL

Il s'agit cette fois d'une solution matérielle. Tous les processeurs modernes possèdent une instruction de type TSL (Test and Set Lock) qui fonctionne de la manière suivante:

Cette instruction garantie que les deux instructions (lecture et écriture) sont **indivisibles**.

- Charge le contenu d'un mot mémoire dans un registre (cycle de lecture)
- Met une valeur non nulle à cette adresse (cycle d'écriture)

Avec l'instruction TSL, on utilise utilise un drapeau initialisé à 0. Lorsqu'un processus trouve le drapeau à 0, il peut accéder à la mémoire partagée (entrer en section critique).

```
entrer_region;

/* Début de la section critique */
...
/* Fin de la section critique */
quitter_region;
```

Toutes les solutions précédentes ont la particularité qu'elle provoque une attente de la part du processus qui demande la ressource. Ceci veut dire qu'un processus occupe le CPU pour uniquement exécuter une boucle qui ne fait rien!! Le CPU pourrait être utilisé à de meilleures fins pendant ces temps d'attente et exécuter un autre processus.

---

### Les sémaphores

Il s'agit d'une des solutions les plus utilisées pour assurer l'exclusion mutuelle. Un sémaphore est une variable entière à valeur non négative et qui agit comme un compteur sur lequel deux types d'opérations sont réalisables:

- Lock : la décrémentation de la variable qui correspond à un **verrouillage** d'une ressource et,
- UnLock :l'incrémentation de la variable qui correspond à un **déverrouillage**

**Le bon fonctionnement des sémaphores repose sur le fait que les opérations d'incrémentation et de décrémentation doivent être indivisibles (non interruptibles).**

Les sémaphores constituent une solution pour résoudre le problème de l'exclusion mutuelle et permettent en particulier de régler les conflits d'accès concurrents de processus distincts à une même ressource.

Comme les opérations sont indivisibles, aucun autre processus ne pourra entrer en région critique tant que le processus présentement dans sa région critique n'exécutera pas l'opération UnLock. La valeur de départ de la variable détermine le nombre de processus qui peuvent faire appel à Lock sans se bloquer.

Fonctionnement général:
Supposons que la variable qui représente le sémaphore se nomme "sem" alors:

```
/* On doit entrer dans une région critique */
        SI (Lock(sem))   //Vérifier si on peut entrer dans une région critique

        // Exécution des instructions qui doivent être protégées

            UnLock(sem);          //Déverrouillage de la portion de code qui doit être
                                  protégée */
        SINON
            /* Aller dormir.  C'est-à-dire, mettre la tâche en attente */
```

**Exemple**

Reprenons l'exemple concernant l'étude de l'utilisation des terminaux. Implantons la solution en utilisant les sémaphores.

```
int sem = 1;      /* Initialisation du sémaphore. Ce nombre représente le nombre de tâche qui
                     peuvent accéder à la ressource en même temps */
int NCR = 0;

void processus_TERMINAL1
{
   /* Instructions de la fonction */
         SI ("Enter" est détecté)
         /* On doit entrer en région critique et protéger les instructions qui incrémente la variable
         NCR */
                  P(sem);  /* Si P(sem) peut se faire, les instructions se continue sinon on va dormir
                              et revenir à cette instruction lorsqu'on sera réveillé */
                    NCR++;
                  V(sem);
         FINSI

  }

void processus_TERMINAL2
{
   /* Instructions de la fonction */
         SI ("Enter" est détecté)
         /* On doit entrer en région critique et protéger les instructions qui incrémente la variable
         NCR */
                  P(sem);  /* Si P(sem) peut se faire, les instructions se continue sinon on va dormir
                              et revenir à cette instruction lorsqu'on sera réveillé */
                    NCR++;
                  V(sem);
         FINSI
  }
```

De cette façon, le système d'exploitation bloque l'accès à la portion de code critique. Le système assure donc qu'une seule tâche à la fois pourra exécuter ce code.

## Gestion des accès concurrents en C++11

Il existe une classe qui permet de gérer adéquatement les sections critiques, il s'agit de « std ::mutex ».

### mutex

entête : <mutex>

Méthodes principales :

lock() : Le thread appelant barre le mutex.

- Si le mutex n'est pas bloqué par un autre thread, le thread appelant bloque le mutex et ce jusqu'à ce que ce dernier appele la méthode “unlock”.
- Si le mutex est présentement bloqué par un autre thread, le thread se met alors en attente jusqu'à ce que l'autre thread libère le mutex.

Unlock(): Débloque le mutex.

Si d'autres threads sont bloqués sur ce mutex, un thread parmi ceux en attente de ce mutex obtiendra ce mutex et pourra continuer son exécution. Les autres attendront encore d'obtenir le mutex.

Prenons un exemple :

#### Exemple 1 : En utilisant une classe

voici une classe toute simple.

```
Class Objet
{
  public :
    int Valeur = 0 ;
    Objet(int Val) : Valeur(Val){} ;

    void Ajoute(){Valeur++} ;

    int  Get(){return Valeur;}
} ;
```

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

class Objet
{
   public:
   int Valeur = 0;
   Objet(){Valeur = 0;};
   Objet(int Val):Valeur(Val){};

   void Ajoute()
   {
      for(int i = 0; i <1000; i++)
      {
         Valeur++;
      }
   }

   int Get(){ return Valeur;}
};


int main()
{
   Objet MonObjet;
   thread MesThreads[10];

   for(int i = 0; i <10; i++)
   {
      MesThreads[i] = thread(&Objet::Ajoute, &MonObjet);
   }

   for(auto &t:MesThreads)
   {
     t.join();
   }

   cout << "Valeur finale: "<< MonObjet.Get() << endl;

   return 0;
}
```

Voici 5 exécutions de ce programme :

Valeur finale : 9616
Valeur finale : 9947
Valeur finale : 9916
Valeur finale : 9729
Valeur finale : 8802

Dans mon cas, je n'ai jamais obtenu la valeur attendue soit 10 000.
J'ai des intervalles d'erreurs de 0.53 % à 12 % . Ceci en utilisant que 10 threads qui compte jusqu'à 1000 seulement.

En utilisant 5 threads dont chacun compte jusqu'à 1 000 000. Je devrais normalement obtenir la somme finale de 5 000 000. Voyons voir :

Valeur finale : 1 835 821
Valeur finale : 1 305 764
Valeur finale : 1 529 677
Valeur finale : 1 629 683
Valeur finale : 1 262 251

J'ai des intervalles d'erreur de l'ordre de 70 %. En fait, à mesure que la boucle doit ajouter un nombre plus élevé, le taux d'erreur devient de plus en plus grand.

Solution :

Nous allons utiliser un mutex pour régler ce problème.

Réécrivons le tout en utilisant ce mutex.

```
class Objet
{
   public:
   int Valeur = 0;
   mutex  MonMutex;

   Objet(){Valeur = 0;};
   Objet(int Val):Valeur(Val){};

   void Ajoute()
   {
      for(int i = 0; i <1000000; i++)
      {
         MonMutex.lock();
         Valeur++;
         MonMutex.unlock();
      }
   }

   int Get(){ return Valeur;}
};
```

Les ajouts à la classe sont en gras.

Ajout d'une données membres de la classe std::mutex → mutex MonMutex ;

La section critique ici se situe au niveau de l'incrémentaion de la donnée membre « Valeur » qui est accédée par plusieurs threads à la fois. Cette portion de code n'est donc pas sécuritaire pour l'usage de thread. On utilisera donc le mutex pour barrer et débarrer la section critique. :

```
MonMutex.lock() ;
        // Section critique...
MonMutex.unlock() ;
```

**Exemple 2 : En utilisant une fonction et une variable globale**

Usage d'une fonction thread et d'une variable globale.

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

int Valeur = 0;

mutex MonMutex;

void Ajoute()
{
    for(int i = 0; i <1000000; i++)
    {
       MonMutex.lock();
       Valeur++;
        MonMutex.unlock();
    }
 }


int main()
{

   thread MesThreads[5];

   for(int i = 0; i <5; i++)
   {
      MesThreads[i] = thread(Ajoute);
   }

   for(auto &t:MesThreads)
   {
     t.join();
   }

   cout << "Valeur finale: "<< Valeur << endl;

   return 0;
}
```

Dans ce cas-ci, en utilisant les mutex, on arrive au bon résultat à tout coup.

## Usage de std::lock_guard

Cette classe permet d'exécuter l'opération de bloquage et de débloquage de façon automatique. Lorsque l'objet lock_guard est créé, il appele automatiquement l'opération « lock() » sur le mutex. Le mutex est libéré automatiquement à la destruction du lock_guard.

Voici un exemple d'utilisation. Reprenons l'exemple précédent avec la classe.

Exemple 1 : Utilisation de lock_guard avec une classe.

Vous n'avez qu'à ajouter l'instruction en gras ci-dessous pour protéger la section critique. Remarquez les “lock” et “unlock” qui disparaissent.

```
for(int i = 0; i <1000000; i++)
    {
      lock_guard<mutex> Cadenas(MonMutex);
      Valeur++;

    }
```

# Communication entre threads

## Variables de condition

Les variables de condition permettent, entre autre, la synchronisation des tâches. C'est un mécanisme qui est spécialement utilisé lorsqu'on veut « signaler » l'état de quelque chose à un autre thread.

- Les variables de conditions permettent ainsi à un thread d'attendre qu'il se produise un événement pour pouvoir continuer sa tâche. Cet événement est signalé par l'intermédiaire d'un autre thread. Sans l'usage de variable de condition, il faudrait continuellement vérifier l'état de la condition par le thread ce qui entraînerait une perte considérable de temps CPU.
- Une variable de condition est toujours utilisée en conjonction avec un mutex. Le mutex protège alors la zone de code qui permet de vérifier l'état de la variable de condition et la zone qui signale que la variable a atteint l'état désiré.

En C++11, il existe deux façons d'utiliser les variables de condition :

- condition_variable: requiert qu'un thread qui doit attendre après cette variable utilise un std::unique_lock avant tout.
- condition_variable_any: est une implantation plus générale qui fonctionne avec n'importe quel type d'algorithme qui permet de barrer et de débarrer une section critique. (principalement, n'importe quel objet qui permet un lock() ou un unlock()).

Coment fonctionne une variable de condition ?

- Il doit y avoir au moins un thread qui attend qu'une condition devienne vraie. Le thread en attente doit acquérir un mutex à l'aide de “unique_lock”. Ce “cadenas” est passé à la méthode wait() qui débarre le mutex automatiquement et met le thread en attente jusqu'à ce que la condition soit signalée. Lorsque le signal est envoyé, le thread en attente est réveillé et ce dernier reprend le mutex.
- Au moins un thread doit signaler que la condition est arrivée. Le signal est envoyé grâce à la méthode notify_one() qui débloque un thread (il peut y en avoir plusieurs qui attendent le signal) ou avec la méthode notify_all() qui débloque tous les threads en attente de cette condition.
- Il est conseillé de vérifier si la condition est toujours vraie après s'être fait réveillé pour être certain que ce n'est pas un réveil autre que celui dû à l'arrivée de la condition.

**Création d'une variable de condition**

Il faudra utiliser l'entête <condition_variable> pour faire usage de ces algorithmes.

```
#include <thread>
#include <mutex>
#include <condition_variable>


std::condition_variable  MaCondition ;
std::mutex  MonMutexCond ;
```

**Exemple : Attente d'un signal par le main et envoie de ce signal par un thread**

Dans cet exemple, le programme principal se met en attente de la réception d'un signal provenant d'un thread qui lui signale que le traitement est terminé.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>


using namespace std;
using namespace std::chrono;


mutex  MonMutexVarCond;

condition_variable MaCondition;
bool  Condition = false;



int main()
{


  std::cout << "main roule..." << std::endl;

  thread LeThreadA (Traitement);;


    std::cout << "main attend le signal du thread..." << std::endl;

  std::unique_lock<std::mutex> lock(MonMutexVarCond);
  while(!Condition)
      MaCondition.wait(lock);

  std::cout << "main a recu le signal du thread..." << std::endl;

 thread1.join();
  return 0;
}
```

```
void Traitement()
{

    cout << "Thread roule.." << std::endl;


    this_thread::sleep_for(seconds(6)); //simule un temps de traitement

    std::unique_lock<std::mutex> locker((MonMutexCout));

    std::cout << "thread envoie le signal..." << std::endl;


  std::unique_lock<std::mutex> lock(MonMutexVarCond);
  Condition = true;
  MaCondition.notify_one();
}
```

**Exemple 2 : Attente d'un signal par un thread et envoie de ce signal par un autre thread.**

Dans cet exemple, le Thread A  attend le signal du Thread B.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>


using namespace std;
using namespace std::chrono;


mutex  MonMutexVarCond;

condition_variable  MaCondition;

bool  Condition = false;

void  Traitement();
void Traitement2();

int main()
{


  std::cout << "main roule..." << std::endl;

  thread LeThreadA (Traitement);
  Thread  LeThreadB(Traitement2);


   LeThreadA.join();
   LeThreadB.join();

  return 0;
}
```

```
void  Traitement()
{
    cout " Le thread A roule"  << endl;

   for(int i = 0; i < 1000000; i++)
  {
     if (i == 100000)
     {
            std::unique_lock<std::mutex> lock(MonMutexVarCond);
            Condition = true;
            MaCondition.notify_one();
     }
  }


}


void Traitement2()
{

     unique_lock<std::mutex> lock(MonMutexVarCond);

     while(!Condition)
        MaCondition.wait(lock);

    cout << "J'ai reçu le signal..." << endl;

    cout << "J'exécute le traitement du thread B" << endl;

   this_thread::sleep_for(seconds(6)); //simule un temps de traitement

}
```