Parallélisme & Objets

Comment abstraire des notions complexes

Les différents niveaux de parallèlisme

- Au niveau des instructions
  - Exécution de plusieurs instructions par le processeur.
  - Ordonnancement des instructions par le processeur.
- Au niveau des processus
  - Processus léger et Processus
  - Ordonnancement par le Système d'Exploitation, support partiel au niveau du processeur
- Au niveau de grappes d'ordinateur
  - Processus distribués
  - Services distants
- Massivement parallèle
  - Modèle SIMD
  - Modèle GPU

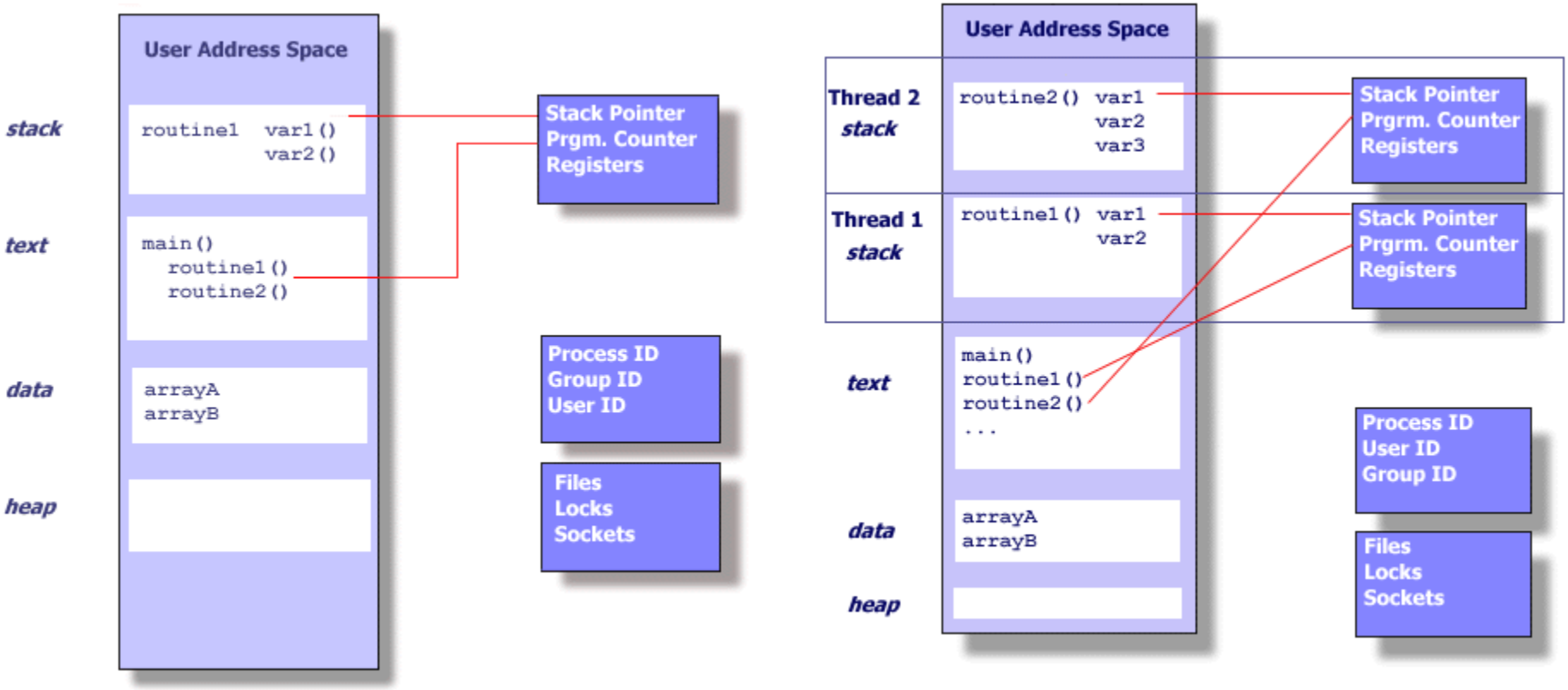
Le support du parallélisme

- Fournit par le système d’exploitation
- Fournit par un middleware
- Fournit par la plateforme virtuelle
- Fournit par une API dédiée
  - API propriétaire
  - API ouverte

L’apport des langages objets

- Capacité à abstraire des notions élémentaires
  - Rend indépendant du système/middleware/Api
  - Offre un accès homogène
- Capacité à fournir des abstractions complexes
- Capacité à fournir des structures de données concurrentes.
- Capacité à aider à la parallélisation.

Processus & Processus léger



Crédits : *Blaise Barney,* https://computing.llnl.gov/tutorials/pthreads/

Information contenue dans un processus/processus léger

- Un code
- Un contexte
  - Données internes
  - Données partagées
  - Propriétés d’exécution
- Un ensemble de fonctions
  - Création
  - Lancement
  - Arrêt
  - Monitoring

Abstraction d'un processus par un objet

- Code :
  - Code propre à la gestion du processus (Méthodes)
  - Code propre au processus.
- Données
  - Données propres au processus
  - Références aux données partagées
  - Propriétés propres au processus

Toutes les informations utiles pour manipuler le processus sont contenues dans l'objet !

Intérêt de l’abstraction

- Simplification du code
- Indépendance de l’API.
- Identification Processus/Action et Objet.
- Possibilité de manipuler des collections de processus
- Possibilité de passer des processus comme des paramètres.

TD --- Partie 1

- Création et manipulation de thread en C++.

Communication et la Synchronisation

- Echange de données par la Mémoire
  - Accès concurrent aux données
  - Garantir l'unicité des opérations
- Signaux & Evènements
  - Envoi de signaux
  - Attente de signaux

Création d'objets (ex. C++)

- Monitor : Gestion des sections critiques
- Mutex : Gestion des exclusions mutelles
- AutoResetEvent, ManualResetEvent : Gestion des évènements.
- Interlocked: Gestion de l'atomicité (Test and Set, Exchange, …)

Intérêt des objets pour la synchronisation

- A chaque objet correspond une notion élémentaire
- A chaque objet est associé un ensemble de services et d'actions
- Méthode prenant ces objets comme argument pour la synchronisation

```
ManualResetEvent mansig;
mansig = new
       ManualResetEvent(false) ;
Console.WriteLine(
    "ManualResetEvent Before
     WaitOne " );
bool b = mansig.
     WaitOne(1000,false); Console.WriteLine(
    "ManualResetEvent After
     WaitOne " + b);
```

Produit à l'exécution :

```
ManualResetEvent After first WaitOne True
ManualResetEvent After second WaitOne False
```

TD --- Partie 2

- Synchronisation par Mutex.

La gestion de l’asynchronisme

- Idée : transformer une opération « longue » en opération « asynchrone »
- L’opération est déléguée à un thread.
- L’opération retourne un objet qui indique si le résultat est disponible ou pas.

Mise en pratique

```
std::future<double> result0 =
        std::async(std::launch::async,
        [] { return exp(1.0); });
```

`std::future<double> result0 :` objet donnant les informations sur l'exécution du code.

`result0.get()`	Retourne le résultat.
`result0.valid()`	Indique l'état du calcul.
`result0.wait()`	Attend que le résultat soit disponible.
`result0.wait_for()`	
`result0.wait_until()`	

TD --- Partie 3

- Asynchronisme

C++ essaye de proposer de nouveaux paradigmes (C++ 17/C++ 20)

- Proposer pour les algorithmes de la STL un ensemble d'algorithmes parallèle

```
std::sort(array.begin(), array.end());

std::sort(std::execution::parallel_policy(),
      array.begin(), array.end());
```

C++ essaye de proposer de nouveaux paradigmes (C++ 17/C++ 20)

- La parallélisation d'ensemble de tâches

```
std::for_each_n(
   array.begin(), 3,
   [](auto& n) { n *= 2; });
```

Produit :
1, 2, 3, 4, 5,
2, 4, 6, 4, 5,

C++ reste en retard sur certains points cf. C#

- Idée : introduire une notion plus abstraite : les tâches.
- Intérêt : simplification du code et de la gestion

```
// create the task
Task<int> task1 = new Task<int>(
   () =>
   {
      int sum = 0;
      for (int i = 0; i < 100; i++)
         sum += i;
      return sum;
   });

Console.WriteLine("Result 1: {0}", task1.Result);
```

Gérer les threads de manière optimale

```
using System;
using System.Threading;
class ThreadDemo {
    public void LongTask1() {
      for (int i = 0; i <= 999; i++) {
         Console.WriteLine("Long Task 1
         is being executed");
      }
   }
  public void LongTask2() {
      for (int i = 0; i <= 999; i++) {
         Console.WriteLine("Long Task 2
         is being executed");
      }
   }
  static void Main() {
    ThreadDemo td = new ThreadDemo();
    for(int i = 0; i < 50; i++) {
       Thread t1 = new Thread(new
              ThreadStart(td.LongTask1));
       t1.Start();
       Thread t2 = new Thread(new
              ThreadStart(td.LongTask2);
       t2.Start();
    }
  }
}
```

```
using System;
using System.Threading;
class ThreadPoolDemo{
    public void LongTask1() {
      for (int i = 0; i <= 999; i++) {
         Console.WriteLine("Long Task 1
         is being executed");
      }
   }
  public void LongTask2() {
      for (int i = 0; i <= 999; i++) {
         Console.WriteLine("Long Task 2
         is being executed");
      }
   }
  static void Main() {
    ThreadPoolDemo tpd = new ThreadPoolDemo();
    for(int i = 0; i < 50; i++) {
       ThreadPool.QueueUserWorkItem(new
          WaitCallback(tpd.LongTask1));
       ThreadPool.QueueUserWorkItem(new
          WaitCallback(tpd.LongTask2)); }
  }
}
```

Les structures de données concurrentes

- Fournir des structures de données qui supportent un accès par plusieurs processus légers en même temps.
- ConcurrentQueue : supporte lecture & écriture par plusieurs tâches concurrentes.
- ConcurrentSet, ConcurrentDictionary, …

Conclusion

- Capture les paradigmes de base de manière uniforme.
- Fournit des abstractions de haut niveau
  - Simplifiant l'écriture de code
  - Simplifiant la mise au point du code
  - Augmentant le niveau de performance.
- Tendance à introduire de nouveau paradigme de programmation dans les langages.