Programmation Répartie 2

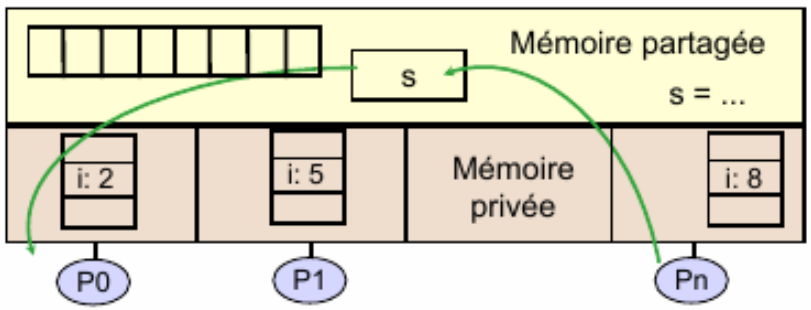
-

1 Thread

2 Gestion de la mémoire partagée

3 Synchronisation des threads

Parallélisme en mémoire partagée


Mémoire partagée
s
s = ...
i: 2
i: 5
Mémoire
privée
i: 8
P0
P1
Pn

Processus et Thread

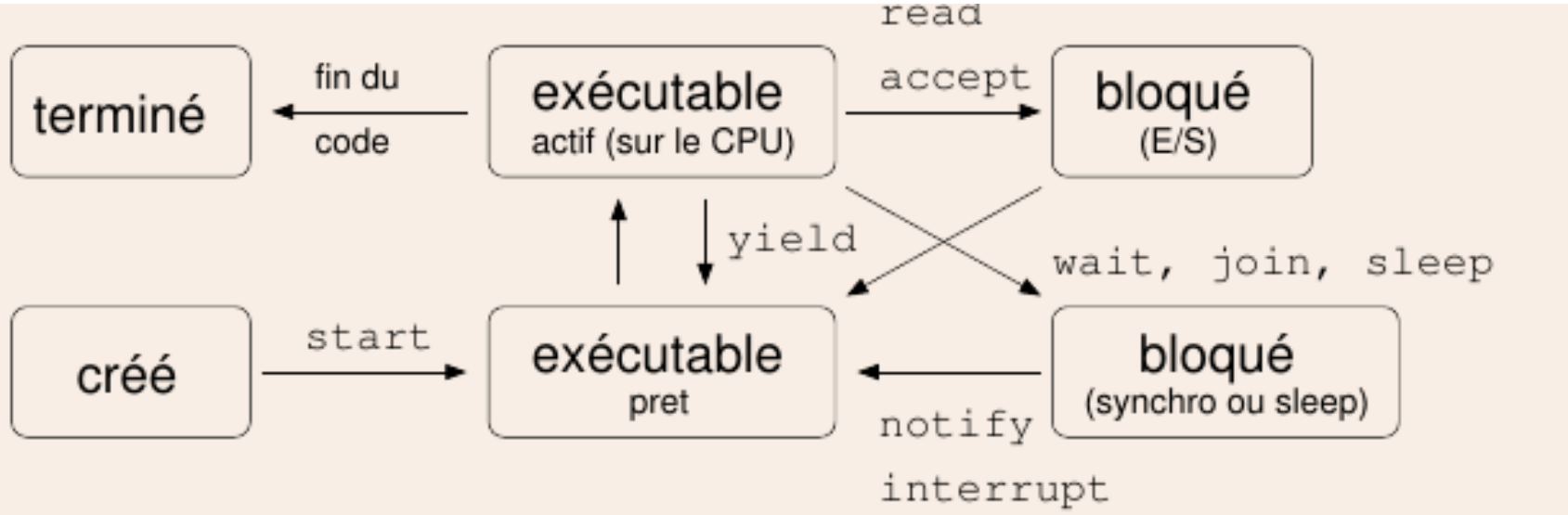
Processus

- Contexte d'exécution d'un programme
- Entité d'attribution du CPU du point de vue du système d'exploitation
- Espace mémoire propre
- Coûteux à créer et à détruire
- Géré par le système
- Temps partagé avec les autres processus
- Communication entre processus difficile
- Difficilement utilisable en programmation (en Java `fork()`)

Thread (ou processus léger)

- Espace mémoire partagé (tas)
- Espace mémoire (pile)
- Communication via données partagées
- Gestion logicielle ou système
- Temps partagé dans le processus

cycle de vie d'un thread



Création d'un thread en Java : Thread

- Définition
  - Classe qui hérite de la classe `Thread`
  - Surcharge de la méthode `run`
- Création
  - Création de l'objet
  - Invocation de la méthode `start`

```
class T extends Thread{

    public void run(){
        //Programme du thread
    }

    public static void main(String[] args){
        T t = new T();
        t.start();
    }

}
```

Création d'un thread en Java : `Runnable`

La version précédente ne permet pas d'hériter d'une autre classe

- Définition
  - Classe qui hérite de l'interface `Runnable`
  - Surcharge de la méthode `run`
- Création
  - Création de l'objet
  - Création d'un objet de type `Thread` avec l'objet précédent en paramètre
  - Invocation de la méthode `start`

```
public class T2 implements Runnable{
    public void run(){
      //Programme du thread
    }

    public static void main(String[] args){
        T2 t2 = new T2();
        Thread t = new Thread(t2);
        t.start();
    }
```

Méthodes de la classe Thread

- `join` : Attend que le thread se termine.
- `isAlive` : revoit un booléen indiquant si le thread est vivant ou non
- `sleep(long ms)` : Cesse de s'exécuter pendant le temps spécifié en millisecondes
- `currentThread` : méthode statique retournant le thread courrant

Gestion de la mémoire partagée

Exemple compteur

```
class Compteur{

int c=0;

public void inc(){
  c = c+1;
}

public void dec(){
  c = c-1;
}

public static void
        main(String [] args){

Compteur ct = new Compteur();

T1 t1 = new T1(ct);
T2 t2 = new T2(ct);
t1.start(); t2.start();

t1.join();t2.join();
System.out.println(ct.c);
}

}
```

```
class T1 extends Thread{

    Compteur ct;

    public T1(Compteur ct){
        this.ct = ct;
    }

    public void run(){
        ct.inc();
    }

}
```

```
class T2 extends Thread{

    Compteur ct;

    public T2(Compteur ct){
        this.ct = ct;
    }

    public void run(){
        ct.dec();
    }

}
```

- Ce que l'on attend : (+1,-1) ou (-1 ;+1), donc dans tous les cas 0
- Résultat : -1 ou 1 sont également des résultats possibles d'éxécution

l'exécution de c = c+1 ou c = c-1 ne se fait pas en une étape (non atomique).

Exemple d'éxécution

- t1 lit la valeur de ct.c ;
- t2 lit la valeur de ct.c ;
- t2 calcule la nouvelle valeur, à partir de la valeur qu'il a dans sa pile (0) :-1 ;
- t2 écrit cette nouvelle valeur dans ct.c (dans le tas) ;
- t1 calcule la nouvelle valeur, à partir de la valeur qu'il a dans sa pile (0) : 1 ;
- t1 écrit cette nouvelle valeur dans ct.c (dans le tas) ; il écrase la valeur écrite par t2. La valeur finale est donc 1.

Conflit

Deux accès concurrent à une **même** donnée partagée dont au moins l'un des accès est une **écriture**.

Éviter les conflits $\Rightarrow$ assurer l'exclusion mutuelle

Plusieurs solutions existent pour assurer l'exclusion mutuelle

- verrous
- sémaphores
- moniteurs
- transactions
- ...

Solution la plus courante en Java

- basée sur les moniteurs
- `synchronized`
- Se place devant
  - méthode
  - bloc d'instructions

```
class Compteur{

    int c=0;

public synchronized
    void inc(){
        c = c+1;
}

public synchronized
    void dec(){
        c = c-1;
}

public static void
    main(String [] args){

Compteur ct = new Compteur();

T1 t1 = new T1(ct);
T2 t2 = new T2(ct);
t1.start(); t2.start();

t1.join();t2.join();
System.out.println(ct.c);
}

}
```

```
class T1 extends Thread{

    Compteur ct;

    public T1(Compteur ct){
        this.ct = ct;
    }

    public void run(){
        ct.inc();
    }

}
```

```
class T2 extends Thread{

    Compteur ct;

    public T2(Compteur ct){
        this.ct = ct;
    }

    public void run(){
        ct.dec();
    }

}
```

- le thread `t1` appelle la méthode `inc` de `ct` qui est `synchronized`. Comme personne n'exécute de méthode `synchronized` sur `ct` , il peut l'exécuter
- le thread `t2`, appelle la méthode `dec` sur le même objet `ct`. Comme `t1` est toujours dans la méthode `inc` de cette objet, il a l'accès exclusif aux blocs `synchonized`. `t2` est mis en attente.
- `t1` termine la méthode `inc`. Il libère donc l'accès exclusif des méthodes `synchonized` de cet objet. `t2` est réveillé, et retente d'exécuter `dec`.

Deux compteurs

```
class Compteur{

    int c=0;

public synchronized
    void inc(){
        c = c+1;
}

public synchronized
    void dec(){
        c = c-1;
}

public static void
    main(String [] args){

Compteur ct1 = new Compteur();
Compteur ct2 = new Compteur();

T1 t1 = new T1(ct1);
T2 t2 = new T2(ct2);
t1.start(); t2.start();

t1.join();t2.join();
System.out.println(ct.c);
 }

}
```

```
class T1 extends Thread{

    Compteur ct;

    public T1(Compteur ct){
        this.ct = ct;
    }

    public void run(){
        ct.inc();
    }

}
```

```
class T2 extends Thread{

    Compteur ct;

    public T2(Compteur ct){
        this.ct = ct;
    }

    public void run(){
        ct.dec();
    }

}
```

- Personne n'exécute de méthode `synchronized` de l'objet `ct1`
- le thread `t1` appelle la méthode `inc` de `ct1` qui est `synchronized`. Comme personne n'exécute, il peut l'exécuter.
- le thread `t2`, appelle `dec` sur un objet différent, `ct2`. `t2` peut l'exécuter.

Synchronisation plus fine

- Synchroniser une méthode complète n'est pas forcément nécessaire.
- Seul un bloc d'instructions sensibles nécessite d'être en exclusion mutuelle. Ce sont les **sections critiques**
- Utilisation de `synchronized(objet)`
- Proche de la notion de verrous

Equivalence des deux solutions

```
public synchronized void inc(){
 c = c+1;
}
```

```
public void inc(){
  synchronized(this){
  c = c+1;
  }
}
```

`Synchronized` sur des méthodes statiques

- Ne se synchronise pas sur l'objet, mais sur la classe
- Au maximum un seul thread dans ces méthodes

`Synchronized` interdit

- Pas de `synchronized` sur les constructeurs : pas de sens car l'objet n'existe pas encore
- Pas de `synchronized` sur les variables

Appel d'une méthode `synchronized` dans une méthode `synchonized`

Même objet

- Si l'on est entré dans une méthode `synchronized` d'un objet, on peut rappeler d'autres méthodes `synchronized` du même objet, on ne sera pas bloqué.
- **Réentrance**

Objet différent

- Comme les autres threads
- Si le thread n'obtient pas l'accès exclusif sur l'autre objet, le thread est mis en attente, sans libérer l'accès exclusif qu'il détient sur l'objet.
- **Deadlock** (ou **interblocage**) : Deux threads attendent mutuellement la ressource que détient l'autre

Limites de synchronized

- l'appel d'une méthode `synchronized` est plus coûteux
- Dans la librairie Java, plusieurs versions d'une même classe existent si les méthodes sont `synchronized` ou non : `ConcurrentMap`, `ConcurrentHashMap`, ...

Synchronisation trop forte

- Lecteur/écrivain
- `synchronized` pour les méthodes lecture/écriture
- Problème pas de lecteurs concurrents

```
class Livre{

    String texte;

    public synchronized void lecture(){
        // Lecture
    }

     public synchronized void ecriture(){
        // Ecriture
    }

}
```

Difficultés : gestion du grain du verrouillage

- Large : Facile à concevoir/ Mauvaise performance
- Fine : Difficile à concevoir/ Meilleure performance

- Nous avons vu que les threads sont mis en attente et réveillés par la JVM avec `synchronized`
- Il est possible de mettre en attente explicitement un thread, et de le réveiller

Uniquement dans des blocs `synchronized`

Synchronisation

- `wait` : libère l'objet, et bloque le thread
- `notify` : réveille un thread bloqué sur cet objet
- `notifyAll` : réveille tous les threads bloqués sur cet objet

Synchronisation

- Les threads se mettent en attente sur un objet
- Plus tard, un thread peut invoquer la méthode `notify` ou `notifyAll` sur cet objet. Cela réveille les threads en attente sur cet objet.
- Les notifications doivent être envoyées lorsque l'objet change d'état.

`Wait/notify et synchronized`

- Lorsqu'un thread est en attente avec le `wait`, il libère l'accès exclusif qu'il avait obtenu avec le bloc `synchronized`
- Les autres threads peuvent ensuite avoir l'accès exclusif sur ce même objet
- Lorsqu'un thread se réveille, il doit obtenir de nouveau l'accès exclusif à l'objet

Réveil parasite

- Même lorsqu'un thread se réveille, il se peut que la condition qu'il attendait soit toujours fausse. Par exemple :
  - La notification ne correspond pas à la modification de la condition attendue
  - Un thread a entre-temps modifié de nouveau l'objet
  - ...
- Nécessaire de vérifier à nouveau la condition avant de quitter le `wait`

```
synchronized m(){

    if (condition())
        wait();

    //En sortant du wait
    //la condition n'est pas reverifiee

    //Code de la methode

}
```

```
synchronized m(){

    while (condition())
        wait();

    //En sortant du wait
    //la condition est reverifiee

    //Code de la methode

}
```

```
public class ConnectionPool {
    private List<Connection> connections = createConnections();

    private List<Connection> createConnections() {
        List<Connection> conns = new ArrayList<Connection>(5);
        for (int i = 0; i < 5; i++) {
            //ajout connexions
        }
        return conns;
    }

    public synchronized Connection getConnection() throws InterruptedException {
            while (connections.isEmpty()) {
                connections.wait();
            }
            return connections.remove(0);
    }

    public synchronized void returnConnection(Connection conn) {
            connections.add(conn);
            connections.notify();
    }

}
```