Les threads

http://alexandre-mesle.com

8 novembre 2022

Définition

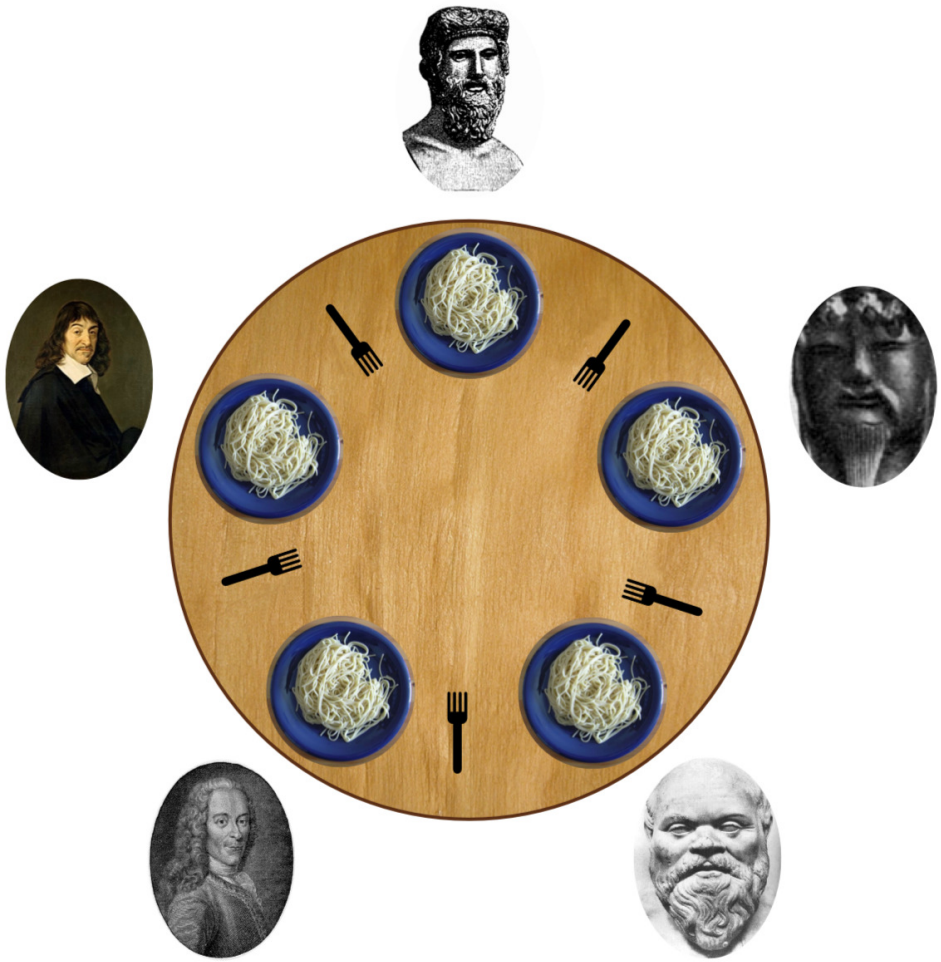
Les **processus légers** (eng. **threads**) sont des programmes s'exécutant en parallèle de façon asynchrone.

Le dîner des philosophes est une illustration des problèmes se posant lorsque l'on manipule des processus.

Définition

Les **processus légers** (eng. **threads**) sont des programmes s'exécutant en parallèle de façon asynchrone.

Le dîner des philosophes est une illustration des problèmes se posant lorsque l'on manipule des processus.



(Illustration par Benjamin D. Esham / Wikimedia Commons, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=56559)

- Un philosophe va utiliser les deux couverts qui sont à côté de son assiette.
- Ses deux voisins ne peuvent pas manger en même temps que lui.

- Un philosophe va utiliser les deux couverts qui sont à côté de son assiette.
- Ses deux voisins ne peuvent pas manger en même temps que lui.

Prendre le couvert gauche
Prendre le couvert droit
Manger
Reposer le couvert droit
Reposer le couvert gauche

- Les philosophes arrivent tous en même temps.
- Prennent chacun le couvert se trouvant à leur gauche.
- Et attendent tous que leur couvert droit se libère.
- $\Longrightarrow$ **interblocage**, (eng. **deadlock**).

- Les philosophes arrivent tous en même temps.
- Prennent chacun le couvert se trouvant à leur gauche.
- Et attendent tous que leur couvert droit se libère.
- $\Longrightarrow$ **interblocage**, (eng. **deadlock**).

- Les philosophes arrivent tous en même temps.
- Prennent chacun le couvert se trouvant à leur gauche.
- Et attendent tous que leur couvert droit se libère.
- $\Longrightarrow$ **interblocage**, (eng. **deadlock**).

- Les philosophes arrivent tous en même temps.
- Prennent chacun le couvert se trouvant à leur gauche.
- Et attendent tous que leur couvert droit se libère.
- $\Longrightarrow$ **interblocage**, (eng. **deadlock**).

- Une solution pourrait être de libérer le couvert gauche si le droit n'est pas disponible.
- $\Longrightarrow$ famine.

- Une solution pourrait être de libérer le couvert gauche si le droit n'est pas disponible.
- $\Longrightarrow$ **famine**.

En java, on définit un thread de deux façons :

- En héritant de la classe `Thread`
- En implémentant l'interface `Runnable`

En java, on définit un thread de deux façons :

- En héritant de la classe `Thread`
- En implémentant l'interface `Runnable`

La classe `Thread` dispose entre autres de deux méthodes

- **public void** `start()` qui est la méthode permettant de démarrer l'exécution du thread.
- public void `run()` qui est la méthode automatiquement invoquée par start quand le thread est démarré.

La classe `Thread` dispose entre autres de deux méthodes

- **public void** `start()` qui est la méthode permettant de démarrer l'exécution du thread.
- **public void** `run()` qui est la méthode automatiquement invoquée par start quand le thread est démarré.

Exemple

```
package threads;

public class BinaireAleatoire extends Thread
{
        private int value;
        private int nbIterations;

        public BinaireAleatoire(int value, int nbIterations)
        {
                this.value = value;
                this.nbIterations = nbIterations;
        }

        @Override
        public void run()
        {
                for (int i = 1; i <= nbIterations; i++)
                        System.out.print(value);
        }

        public static void main(String[] args)
        {
                Thread un = new BinaireAleatoire(1, 30);
                Thread zero = new BinaireAleatoire(0, 30);
                un.start();
                zero.start();
        }
}
```

- L'interface `Runnable` contient une méthode **public void** `run()`
- Constructeur `Thread(Runnable r)`

- L'interface `Runnable` contient une méthode **public void** `run()`
- Constructeur `Thread(Runnable r)`

Exercice

Utilisez l'interface `Runnable` pour lancer `BinaireAleatoire`.

Le modèle producteur/consommateur se construit à l'aide de deux programmes :

- Le producteur transmet des données en les faisant transiter par une mémoire tampon.
- Le consommateur traite les données produites en les récupérant dans la mémoire tampon.

Le modèle producteur/consommateur se construit à l'aide de deux programmes :

- Le producteur transmet des données en les faisant transiter par une mémoire tampon.
- Le consommateur traite les données produites en les récupérant dans la mémoire tampon.

Le modèle producteur/consommateur se construit à l'aide de deux programmes :

- Le producteur transmet des données en les faisant transiter par une mémoire tampon.
- Le consommateur traite les données produites en les récupérant dans la mémoire tampon.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

- Mémoire tampon pleine $\Longrightarrow$ le producteur se met *en sommeil*.
- Mémoire tampon vide $\Longrightarrow$ le consommateur se met en sommeil.
- Le producteur place une donnée dans une mémoire tampon vide $\Longrightarrow$ réveil du consommateur.
- Le consommateur libère une place dans une mémoire tampon pleine $\Longrightarrow$ réveil du producteur.

Si *la mémoire tampon est pleine* **alors**
    (*)
    Se mettre en sommeil
Sinon
    Placer une donnée dans la mémoire tampon
    Si *la mémoire tampon était vide* **alors**
        Réveiller le consommateur
    Fin si
Fin si

Si *la mémoire tampon est vide* **alors**
    Se mettre en sommeil
Sinon
    Récupérer une donnée dans la mémoire tampon
    Si *la mémoire tampon était pleine* **alors**
        Réveiller le producteur
    Fin si
Fin si

La commutation entre les processus peut avoir lieu à n'importe quel moment :

Si *la mémoire tampon est pleine* **alors**
    (* commutation *)
    Se mettre en sommeil
Fin si
...

Le signal de réveil est envoyé par le consommateur.

Si *la mémoire tampon était pleine* **alors**

    Réveiller le producteur
    (* le producteur ne dort pas :-(*)

Fin si

Puis, le producteur s'endort.

Si *la mémoire tampon est pleine* **alors**
    (* commutation *)
    Se mettre en sommeil
Fin si

Le consommateur va vider la mémoire tampon.

Si *la mémoire tampon est vide* **alors**
    Se mettre en sommeil
Sinon
    Récupérer une donnée dans la mémoire tampon
    Si *la mémoire tampon était pleine* **alors**
        Réveiller le producteur
    Fin si
Fin si

Puis s'endormir.

Le consommateur va vider la mémoire tampon.

Si *la mémoire tampon est vide* **alors**
    Se mettre en sommeil
Sinon
    Récupérer une donnée dans la mémoire tampon
    Si *la mémoire tampon était pleine* **alors**
        Réveiller le producteur
    Fin si
Fin si

Puis s'endormir.

- Les deux processus sont en sommeil.
- Et attendent que l'autre les réveille...
- :'-(

- Les deux processus sont en sommeil.
- Et attendent que l'autre les réveille...
- :'-(

- Les deux processus sont en sommeil.
- Et attendent que l'autre les réveille...
- :'-(

Définition

Une **section critique** est un bloc d'instructions qu'il est impossible d'interrompre.

Une section critique se construit avec le mot-clé `synchronized`.

Définition

Une **section critique** est un bloc d'instructions qu'il est impossible d'interrompre.

Une section critique se construit avec le mot-clé **synchronized**.

Définition

Une **méthode synchronisée** verrouille un objet pendant son exécution.

- Elle met en attente les autres threads tentant d'accéder à l'objet.
- On synchronise une méthode en plaçant le mot clé `synchronized` dans sa définition.

Définition

Une **méthode synchronisée** verrouille un objet pendant son exécution.

- Elle met en attente les autres threads tentant d'accéder à l'objet.
- On synchronise une méthode en plaçant le mot clé `synchronized` dans sa définition.

Définition

Une **méthode synchronisée** verrouille un objet pendant son exécution.

- Elle met en attente les autres threads tentant d'accéder à l'objet.
- On synchronise une méthode en plaçant le mot clé `synchronized` dans sa définition.

On synchronise des instructions en les plaçant dans un bloc

```
synchronized(o)
{
  /* ... */
}
```

Où o est l'objet ne pouvant être accédé par deux threads simultanément.

Un thread peut décider de se mettre en attente s'il a besoin pour s'exécuter de données qui ne sont pas encore disponibles. On gère cela avec les instructions suivantes :

- `public void wait()throws InterruptedException` met le thread en attente.
- `public void notify()` réveille un thread en attente.
- `public void notifyAll()` réveille tous les threads en attente.

Un thread peut décider de se mettre en attente s'il a besoin pour s'exécuter de données qui ne sont pas encore disponibles. On gère cela avec les instructions suivantes :

- `public void wait()throws InterruptedException` met le thread en attente.
- `public void notify()` réveille un thread en attente.
- `public void notifyAll()` réveille tous les threads en attente.

Un thread peut décider de se mettre en attente s'il a besoin pour s'exécuter de données qui ne sont pas encore disponibles. On gère cela avec les instructions suivantes :

- **public void** `wait()` **throws** `InterruptedException` met le thread en attente.
- **public void** `notify()` réveille un thread en attente.
- **public void** `notifyAll()` réveille tous les threads en attente.

Un thread peut décider de se mettre en attente s'il a besoin pour s'exécuter de données qui ne sont pas encore disponibles. On gère cela avec les instructions suivantes :

- **public void** `wait()` **throws** `InterruptedException` met le thread en attente.
- **public void** `notify()` réveille un thread en attente.
- **public void** `notifyAll()` réveille tous les threads en attente.

Exemple

```
public class Counter
{
        private int value = 0;
        private int upperBound;
        private int lowerBound;

        public Counter(int lowerBound, int upperBound)
        {
                this.upperBound = upperBound;
                this.lowerBound = lowerBound;
                value = (upperBound + lowerBound) / 2;
        }

        public synchronized void increaseCounter() throws InterruptedException
        {
                while (value == upperBound)
                        wait();
                value++;
                System.out.println("+ 1 = " + value);
                if (value == lowerBound + 1)
                        notify();
        }

        public synchronized void decreaseCounter() throws InterruptedException
        {
```

Exemple

```
                wait();
        value--;
        System.out.println("- 1 = " + value);
        if (value == upperBound - 1)
                notify();
}

public static void main(String[] args)
{
        Counter c = new Counter(0, 100);
        Thread p = new Plus(c);
        Thread m = new Moins(c);
```

Exemple

```
                this.c = c;
        }

        @Override
        public void run()
        {
                while (true)
                        try{c.increaseCounter();}
                        catch (InterruptedException e){}

        }
}

class Moins extends Thread
{
        private Counter c;

        Moins(Counter c)
        {
                this.c = c;
        }

        @Override
        public void run()
        {
                while (true)
                        try{c.decreaseCounter();}
                        catch (InterruptedException e){}
        }
}
```

Exemple

```
                m.start();
        }
}

class Plus extends Thread
{
        private Counter c;
```