Threads et Synchronization

Pourquoi utiliser les threads?

- Pour effectuer du travail en background, sans forcer le reste de l'application à attendre
  - GUI
  - Lecture audio/vidéo
- Pour diviser un travail de calcul en plusieurs morceaux, et ainsi accélérer ce calcul avec plusieurs processeurs
  - Raytracing, graphiques, encodage vidéo
  - Jeux vidéos, calculs physiques/AI
  - Calculs scientifiques (e.g.: météo), milliers de processeurs

Difficultés liées aux threads

- Le flot d'exécution n'est plus linéaire, difficile à prévoir. L'ordre des événements n'est plus déterministe
- Certains calculs ont quand même des contraintes d'ordonnancement qui doivent êtres maintenues
  - eg: B et C doivent attendre A
- Partage et transmission d'informations et de ressources
  - Risque de corruption de données
  - Risque de deadlock

Protection des ressources partagées

- En utilisant un verrou ("lock"), on peut s'assurer qu'un seul thread à la fois utilise une ou plusieurs ressources
- En Java, un verrou séparé est associé à chaque classe et chaque objet
  - On peut s'assurer qu'une méthode n'est exécutée que par un seul thread à la fois en utilisant le mot clé synchronized
  - Il est aussi possible d'acquérir un verrou explicitement sur un objet en utilisant un bloc synchronized

Pourquoi un verrou?

- On ne veux pas qu'un thread lise une ressource partagée pendant qu'on la modifie, et qu'on a seulement a moitié fini la modification
  - La ressource pourrait être lue dans un état incohérent
- On ne veux pas qu'un autre thread écrive par dessus nos modifications pendant qu'on est en train de modifier la ressource nous même
  - Les modifications pourraient êtres mélangées et laisser la ressource dans un état corrompu

Mot clé synchronized

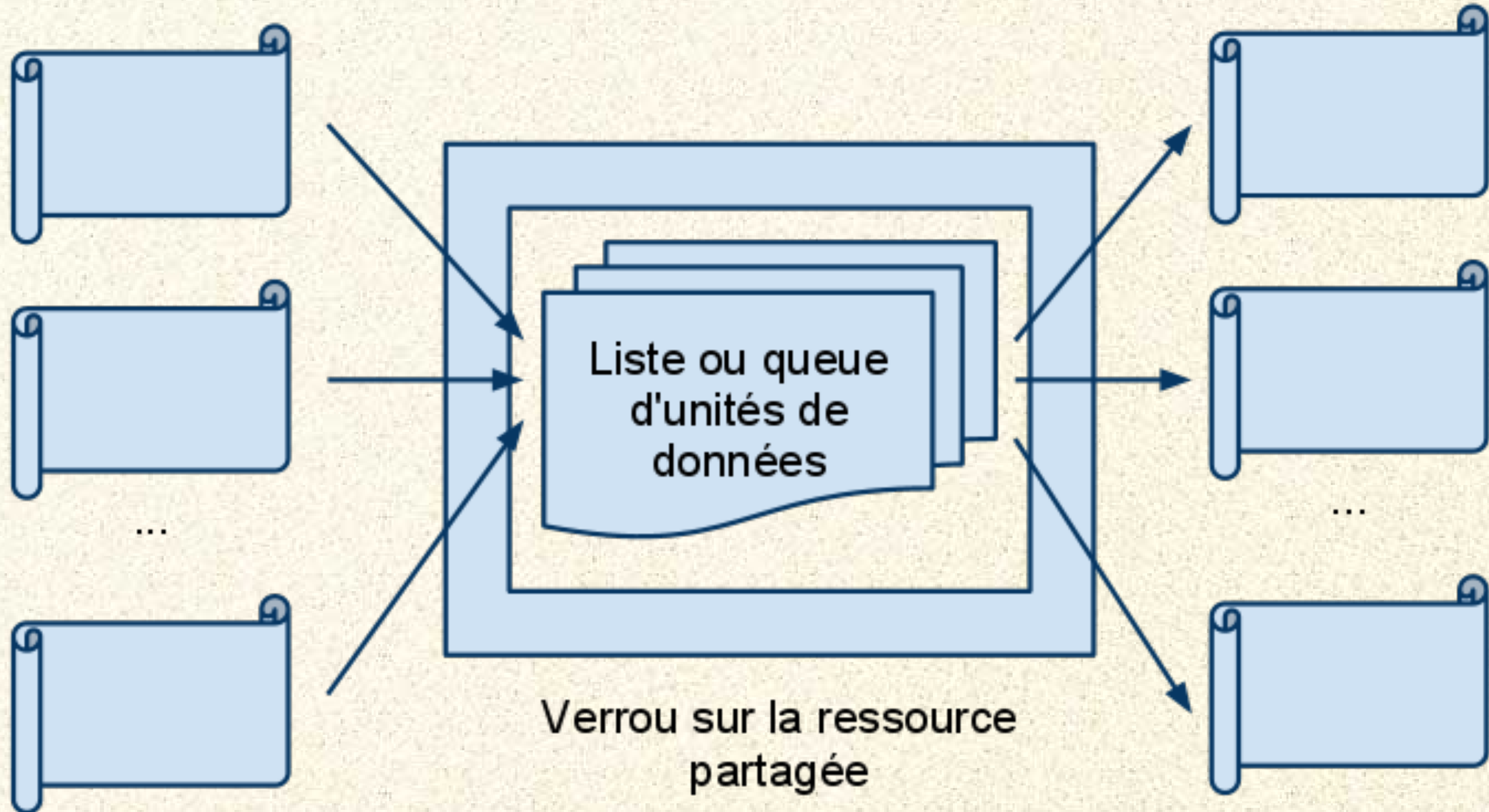
```
public synchronized void foo()
{
    // Opération atomique sur
    // l'instance (this)
}

public void bar()
{
    synchronized (object)
    {
        // Opération atomique sur
        // l'objet. Fonctionne sur
        // n'importe quel objet,
        // incluant les collections
    }
}
```

Partage de ressources

- Un scénario fréquent est l'architecture producteur-consommateur
  - Implique qu'un ou plusieurs threads <u>producteurs</u> produisent des unités de données pour un ou plusieurs threads <u>consommateurs</u> qui utilisent ces données
- Si ces threads utilisent une liste d'unités de données, ils doivent tous se synchronizer sur cette liste. Les producteurs y rajoutent des données, les consommateurs doivent périodiquement vérifier que des nouvelles données sont disponibles

Partage de ressources



Partage de ressources

- Java fourni un moyen pratique de gérer ce genre de situation, avec les méthodes wait et notify présentes sur chaque objet. Celles si s'appellent à l'intérieur d'un bloc synchronized sur l'objet
- Un thread consommateur peut appeller wait pour attendre d'être notifié plus tard, quand une unité de donnée sera disponible
  - Le verrou du bloc est ainsi relâché, et le thread entre en sommeil
- Un thread producteur ajoute des données dans un bloc synchronized, et notifie (avec notify) un consommateur quand les données sont prêtes