Département d'informatique et de mathématiques

8INF342 – Système d'exploitation

UQAC

Rapport du Laboratoire #2 – Multithreading et synchronisation En C++

Travail réalisé par :

KEITA SARAN MADY

Date de modification : 26/10/2025

A) Introduction:

Le laboratoire #2 a pour objectif de mettre en pratique les mécanismes de **synchronisation entre threads** à l'aide de **mutex** en langage C++.

Le but est de simuler la création d'objets (comme *Coupe*, *Porte*, *Table*, etc.) nécessitant chacun deux matériaux distincts (ex. *Or* et *Argent*). Chaque type d'objet est créé par un thread distinct, pour un total de **7 threads** travaillant en parallèle et partageant **7 ressources**.

L'exercice vise à garantir une exécution correcte et équitable des threads, sans **interblocage (deadlock)** ni **famine (starvation)**, tout en produisant au moins **10 objets de chaque type**.
Le programme doit également afficher les messages d'état à l'écran et dans un fichier `sortieCreation.txt`.

Méthodologie

1. Outils et bibliothèques

Le programme a été développé en **C++** en utilisant :

- `<thread>` pour la gestion des threads ;
- `<mutex>` pour les verrous ;
- `<chrono>` pour les temporisations ;
- `<fstream>` pour l'écriture dans le fichier `sortieCreation.txt`.

Une interface graphique est fournie par le laboratoire via la bibliothèque **X11**, permettant d'afficher l'état des objets et des ressources.

2. Structure du programme

Le code fourni contenait déjà la structure principale :

- La création des threads,
- L'interface graphique,
- Et la fonction `start()` pour lancer chaque cycle.

La seule fonction à modifier était **`void create(int i, craft_t *c)`**, qui décrit la logique de création d'un objet par un thread donné.

3. Synchronisation avec les mutex

Chaque ressource (Argent, Or, Bois, Métal, etc.) est associée à un **mutex distinct**, déclaré comme suit :

```
mutex ressource_mutex[7];
```

Avant de créer un objet, un thread doit obtenir les **deux mutex correspondant à ses matériaux**.
Par exemple :

- Le thread "Coupe" doit verrouiller les mutex de **Argent (m1 = 0)** et **Or (m2 = 1)**.

Les verrous sont acquis **uniquement si les deux ressources sont disponibles**. Cela évite de bloquer une ressource inutilement.
L'accès au fichier `sortieCreation.txt` et à la console est aussi protégé par des mutex (`file_mutex` , temps_mutex et `message_mutex`) pour empêcher des écritures concurrentes.

4. Prévention de la famine et des interblocages

Le risque d'interblocage se produit lorsqu'un thread prend une ressource et attend indéfiniment l'autre.
Pour l'éviter :

- L'acquisition des ressources se fait à l'aide de **`try_lock()`** au lieu de `lock()`.
- Si le thread ne peut pas obtenir les deux mutex, il **libère immédiatement** la ressource qu'il détient et **attend un court instant** avant de réessayer.
- De plus, l'ordre de prise des verrous est **aléatoire** pour éviter un ordre fixe (comme prendre toujours m1 avant m2), qui pourrait favoriser certains threads et provoquer une **famine**.

5. Nombre de mutex utilisés

- **7 mutex** pour les 7 ressources.
- **1 mutex** pour le fichier `sortieCreation.txt`.
- **2 mutex** pour la console (`cout`).

Total : **10 mutex**.

6. Pourquoi ne pas bloquer une ressource en attendant l'autre ?

Il est interdit de verrouiller une ressource et d'attendre la seconde, car cela peut mener à un **interblocage (deadlock)**.

Exemple :

- Le thread A prend "Argent" et attend "Or".
- Le thread B prend "Or" et attend "Argent".
  Les deux threads restent bloqués pour toujours.

La bonne approche est donc :

1. Vérifier si **les deux ressources sont disponibles** (avec `try_lock()`),
2. Si l'une manque, **libérer immédiatement** celle obtenue,
3. Attendre un court délai, puis **réessayer plus tard.**

Cette méthode garantit que les ressources ne restent jamais bloquées inutilement, et qu'aucun thread ne monopolise une ressource pendant qu'il attend une autre.

Résultats et discussion

Le programme produit un affichage dans la console et dans le fichier `sortieCreation.txt`, indiquant :

Les créations réussies, ex. :

```
Thread : 0 crée un objet : [Coupe] avec Argent et Or
```

Ou les attentes :

```
Thread : 6 attend : Argent
```

Chaque type d'objet est créé **au moins 10 fois**, comme exigé.
L'utilisation de `try_lock()` et des délais aléatoires permet une exécution fluide sans famine ni interblocage.

Le temps total d'exécution dépend du nombre de cycles (environ 2 à 3 secondes pour 10 objets de chaque type).
Tous les résultats sont correctement consignés dans `sortieCreation.txt`.

Conclusion

Ce laboratoire nous a permis d'appliquer les concepts de **synchronisation, exclusion mutuelle et prévention des interblocages** dans un contexte multithreadé.
L'utilisation de **mutex indépendants** pour chaque ressource, combinée à une **prise aléatoire et non bloquante**, a permis d'éviter à la fois la famine et les deadlocks.

La logique de synchronisation implémentée assure que :

- Les ressources ne sont utilisées que lorsqu'elles sont toutes disponibles,
- Aucun thread ne monopolise une ressource,
- Et la progression globale du programme est garantie.

Le résultat final satisfait toutes les exigences du laboratoire : affichages corrects, synchronisation robuste, et absence d'interblocage.