

Simulation du probleme du barbier endormie

Étudiants : Laaz Abdelaziz Farouk, Kaddour Yazid

Table des matières

1	Introduction générale	2
1.1	Contexte général	2
1.2	Objectifs du projet	2
2	Présentation du problème du barbier endormi	3
2.1	Description informelle du scénario	3
2.2	Contraintes et hypothèses du problème	3
3	Modélisation du salon de coiffure	4
3.1	Processus / threads barbier et clients	4
3.2	Ressources : fauteuil, chaises d'attente, file FIFO	4
4	Mécanismes de synchronisation : semaphores et mutex	4
4.1	Rappel sur les sémaphores	4
4.2	Rappel sur les mutex	5
4.3	Synchronisation entre threads (barbier et clients)	5
5	Algorithmes et comportement des threads	5
5.1	Thread barbier (pseudo-code et description)	5
5.2	Threads clients (pseudo-code et description)	6
5.3	Analyse du déroulement (scénarios typiques)	7
6	Implémentation et environnement de développement	7
6.1	Langage, bibliothèques de threads et APIs de synchronisation utilisées . . .	7
6.2	Gestion des threads : création, paramètres, fin d'exécution	8
6.3	Interface graphique et captures d'écran de l'application	9
7	Conclusion	11
7.1	Bilan du projet	11
7.2	Pistes d'amélioration	11
	Références	12

1 Introduction générale

1.1 Contexte général

L'étude du problème du Barbier Endormi consiste à simuler et à bien gérer les échanges entre un seul serveur (le barbier) et plusieurs clients. Ça montre bien qu'il est dur de faire marcher ensemble plusieurs programmes sans créer de problèmes comme des courses critiques, des blocages ou de la famine, tout en suivant la logique de l'histoire : si le salon est vide, le barbier pionce, et quand un client arrive, il doit soit le réveiller s'il dort, soit patienter dans une salle d'attente pas trop grande. Ce problème soulève des questions importantes sur comment créer des algorithmes, spécialement pour gérer les choses qu'on partage comme le fauteuil du barbier et les sièges.

Ici, les systèmes pour faire marcher les choses ensemble comme les sémaphores et les mutex sont très importants pour contrôler qui accède aux ressources partagées et pour organiser comment se comportent les différents programmes (barbier et clients). Ils permettent de s'assurer qu'un nombre limité de clients puissent patienter sans que ça devienne le bazar, et que le barbier ne soit sollicité que quand il est en état de bosser (réveillé ou endormi). Des modèles aident à montrer exactement comment le système doit marcher, et rendent plus simple la création d'algorithmes qui sont solides, sûrs et sans bugs de synchronisation.

Le challenge c'est de trouver une façon bien foutue de réveiller le barbier seulement quand on a besoin, et d'éviter de le réveiller pour rien. La gestion des sièges doit respecter le nombre de places pour éviter le bordel, ce qui demande une bonne organisation de la salle d'attente et de comment le barbier et les clients interagissent. Ce problème montre qu'il faut une solution qui assure la sécurité, que ça avance et que le système marche bien, tout en étant simple à modifier et à adapter à différentes situations.

1.2 Objectifs du projet

- Modéliser le problème du barbier endormi sous forme de système concurrent comportant un barbier, des clients, une salle d'attente limitée et une file fifo
- Concevoir et implémenter une solution de synchronisation utilisant threads, sémaphores et mutex pour éviter les conditions de cours les blocages et la famine.
- mettre en oeuvre cette solution dans Godot engine (GDScript) en respectant la logique du scénario (barbier qui dort, réveil par les clients, refus si salle pleine)
- Visualiser le fonctionnement du système à travers une interface graphique qui montre l'arrivée des clients, l'état du barbier et l'occupation des places
- Evaluer expérimentalement le comportement de la solution (des différents tests avec des différents nb de chaises nb de clients) afin de vérifier la performance du système

2 Présentation du problème du barbier endormi

2.1 Description informelle du scénario

Le problème du barbier endormi modélise un salon de coiffure avec un seul barbier, un fauteuil de coupe et une salle d'attente de capacité limitée ,lorsque le salon est vide, le barbier dort sur son fauteuil ; dès qu'un client arrive il le réveille pour être servi ou va patienter dans la salle d'attente si le barbier est déjà occupé, si la salle d'attente est pleine le client repart sans être coiffé. Ce scénario illustre un problème de synchronisation où plusieurs clients tentent d'accéder à une ressource unique (le barbier) à travers une file d'attente.



FIGURE 2.1 – Représentation graphique du barbier endormi quand aucun client n'est présent

2.2 Contraintes et hypothèses du problème

Les principales contraintes et hypothèses du problème sont les suivantes :

- Un seul barbier et un seul fauteuil de coupe sont disponibles.
- La salle d'attente possède une capacité fixe de N chaises.
- Au maximum N clients peuvent patienter en même temps (selon le nb des chaises) les autres quittent le salon.
- Tous les clients sont traités de manière homogène, sans priorité particulière.
- La politique de service est de type « premier arrivé, premier servi » (fifo)
- Les changements d'état (arrivée, début/fin de coupe, sommeil/réveil du barbier) doivent être gérés de façon atomique.
- L'accès aux ressources partagées (fauteuil, chaises, compteur de clients, état du barbier) est protégé par des mécanismes de synchronisation.

3 Modélisation du salon de coiffure

3.1 Processus / threads barbier et clients

Dans notre simulation, le barbier et les clients sont comme des acteurs qui agissent en même temps. Le barbier, c'est un peu comme un artiste solo qui attend qu'on lui demande de couper des cheveux. S'il n'y a personne, il patiente et se réveille dès qu'un client arrive. Les clients, eux, sont joués par différentes personnes, chacune gérant son propre parcours : arrivée, attente, coupe. Chaque client fait la même chose : il essaie de trouver une place dans la salle d'attente ou d'aller directement voir le barbier si c'est possible. Pour que tout se passe bien entre le barbier et les clients, on utilise des outils de synchronisation (comme des semaphores et des mutex) afin que chacun ait accès aux ressources sans se marcher sur les pieds.

3.2 Ressources : fauteuil, chaises d'attente, file FIFO

Voici comment le salon de coiffure est organisé : Le fauteuil du coiffeur est super important parce qu'il ne peut y avoir qu'une personne à la fois. Faut donc bien gérer qui s'assoit. La salle d'attente n'a pas beaucoup de places, donc pas trop de clients peuvent attendre ensemble. Pour l'ordre de passage, c'est simple : premier arrivé, premier servi. On a une liste d'attente que le coiffeur et les clients partagent. L'accès à cette liste est protégé pour éviter les embrouilles quand plusieurs personnes veulent ajouter ou enlever des noms en même temps.

4 Mécanismes de synchronisation : semaphores et mutex

4.1 Rappel sur les sémaphores

Un semaphore, c'est un peu comme un compteur qui aide à gérer qui accède à une ressource ou une partie de code en même temps. On le contrôle avec deux actions principales : une qui diminue le nombre (on dit souvent wait ou P). Si le compteur est à zéro, ça bloque le fil d'exécution qui essaie d'accéder. L'autre action augmente le nombre (on dit signal ou V). Ça peut réveiller un fil d'exécution qui attendait. Si on l'utilise comme un compteur normal, ça peut indiquer, par exemple, le nombre de clients qui attendent dans un salon ou le nombre de places libres.

Dans l'histoire du barbier endormi, on utilise les sémaphores pour dire au barbier quand il y a des clients et pour dire aux clients que le barbier est prêt. C'est très important pour éviter que plusieurs fils d'exécution se battent pour la même ressource. On ne peut accéder au barbier que si le semaphore le permet, ce qui assure que tout le monde est bien coordonné.

4.2 Rappel sur les mutex

Un mutex (mutual exclusion lock) est comme un interrupteur qui assure qu'une seule tâche à la fois accède à une zone sensible du code. Il marche avec deux actions : verrouiller et déverrouiller. Quand une tâche a le mutex, les autres doivent patienter jusqu'à ce qu'il soit libre, évitant ainsi le bazar des accès simultanés aux infos partagées.

Dans le scénario du barbier qui pionce, le mutex sert à protéger les actions importantes sur les infos partagées, par exemple, la queue des clients, le nombre de clients qui attendent ou si le barbier roupille. Ainsi, on est sûr que les changements de ces infos se font proprement et qu'il n'y a pas d'erreurs, comme plusieurs clients qui essaient de se mettre au même endroit dans la queue en même temps ou des chiffres incorrects.

4.3 Synchronisation entre threads (barbier et clients)

La synchronisation entre le thread du barbier et les threads clients repose sur une combinaison de sémaphores et de mutex. Lorsqu'un client arrive il entre d'abord dans une section critique protégée par un mutex pour vérifier s'il reste une place dans la salle d'attente et le cas échéant s'ajouter à la file, s'il parvient à s'y insére, il incrémente un semaphore représentant le nombre de clients en attente, ce qui servira de signal pour le barbier. Le barbier, de son côté, exécute une boucle où il attend ce semaphore : lorsqu'il se débloque, cela signifie qu'au moins un client est présent et peut être servi.

Une fois réveillé, le barbier reprend le mutex pour accéder à la file d'attente, retire le client en tête, puis libère le mutex et commence la coupe. À la fin de la coupe il retourne dans son état d'attente en surveillant à nouveau le semaphore des clients. De leur côté, les clients peuvent aussi attendre sur un semaphore indiquant que le barbier est prêt à les prendre en charge. Ce schéma de synchronisation garantit que le barbier n'essaie jamais de servir un client inexistant que les clients ne montent sur le fauteuil qu'à leur tour et que l'accès aux ressources partagées reste cohérent malgré l'exécution concurrente des threads.

5 Algorithmes et comportement des threads

5.1 Thread barbier (pseudo-code et description)

Le comportement du barbier est implémenté dans un thread dédié chargé d'exécuter une boucle de travail tant que la simulation est active. Ce thread attend qu'au moins un client soit présent dans la file logique, sélectionne le client en tête de file, met à jour le nombre de places libres dans la salle d'attente, puis simule la durée de la coupe avant de signaler la fin du service au thread principal. La logique interne s'appuie sur un semaphore pour l'attente de clients, un mutex pour protéger la file d'attente et des événements transmis au thread principal pour mettre à jour l'interface graphique.

Pour illustrer cette logique, la figure ci-dessous présente un extrait de code GDScript correspondant à la fonction de boucle du barbier, montrant notamment l'attente sur le semaphore, l'accès protégé à la file FIFO et l'émission d'événements de type *barber_started* et *barber_finished*.

```

109
110  ↘ func _barber_loop(_userdata: Variant = null) -> void:
111    ↗ while _running:
112      ↗ ↗ _customers_sem.wait()
113    ↗ ↗ if not _running:
114      ↗ ↗ ↗ break
115
116      ↗ ↗ var cust_id: int = -1
117      ↗ ↗ var waiting: int = 0
118      ↗ ↗ var free: int = 0
119
120      ↗ ↗ ↗ _mutex.lock()
121    ↗ ↗ if _waiting_queue.size() > 0:
122      ↗ ↗ ↗ cust_id = _waiting_queue.pop_front()
123      ↗ ↗ ↗ _free_seats = min(num_seats, _free_seats + 1)
124      ↗ ↗ ↗ waiting = _waiting_queue.size()
125      ↗ ↗ ↗ free = _free_seats
126      ↗ ↗ ↗ _mutex.unlock()
127
128    ↗ ↗ if cust_id == -1:
129      ↗ ↗ ↗ continue
130
131    ↗ ↗ ↗ _push_event({
132      ↗ ↗ ↗ "type": "barber_started",
133      ↗ ↗ ↗ "id": cust_id,
134      ↗ ↗ ↗ })
135    ↗ ↗ ↗ _push_event({
136      ↗ ↗ ↗ "type": "stats",
137      ↗ ↗ ↗ "waiting": waiting,
138      ↗ ↗ ↗ "free": free,
139      ↗ ↗ ↗ })

```

FIGURE 5.1 – Extrait de code GDScript illustrant la logique du thread barbier.

5.2 Threads clients (pseudo-code et description)

Le comportement des clients est déclenché à chaque nouvelle arrivée dans la simulation. Lorsqu'un client se présente, le code tente d'abord de réserver une place dans la salle d'attente en appelant une fonction de type *request_seat*, qui vérifie le nombre de sièges disponibles, met à jour la file FIFO si une place est libre et réveille éventuellement le barbier via le sémaphore associé. Si aucune place n'est disponible, le client est immédiatement refusé, ce qui correspond au cas où la salle d'attente est pleine.

Sur le plan visuel, chaque client logique accepté est associé à un client graphique qui se déplace vers une chaise d'attente, puis vers le fauteuil du barbier lorsque celui-ci commence la coupe. La figure suivante montre un extrait de code GDScript gérant la demande de siège et la création des clients dans l'interface, mettant en évidence la coordination entre la logique de synchronisation et la représentation graphique.

```

79
80  ↘ func request_seat() -> int:
81    # Called from main thread when a new customer arrives.
82    ↗ var customer_id: int = -1
83
84    ↗ _mutex.lock()
85    ↗ if _free_seats > 0:
86      ↗ _free_seats -= 1
87      ↗ _next_customer_id += 1
88      ↗ customer_id = _next_customer_id
89      ↗ _waiting_queue.append(customer_id)
90      ↗ var waiting := _waiting_queue.size()
91      ↗ var free := _free_seats
92      ↗ _mutex.unlock()
93
94      ↗ emit_signal("stats_updated", waiting, free)
95      ↗ _customers_sem.post() # wake barber
96      ↗ return customer_id
97    ↗ else:
98      ↗ _mutex.unlock()
99      ↗ return -1
100 |
101

```

FIGURE 5.2 – Extrait de code GDScript illustrant la gestion des clients et de la file d’attente.

5.3 Analyse du déroulement (scénarios typiques)

Un premier scénario correspond à un salon vide : aucun client n'est présent et le barbier dort. À l'arrivée d'un client, celui-ci réveille le barbier et est servi immédiatement, puis le barbier se rendort si aucun autre client n'attend.

Un second scénario est celui d'une salle d'attente partiellement remplie : les clients se rangent dans la file FIFO jusqu'à la capacité maximale, et le barbier enchaîne les coupes en servant toujours le client en tête de file.

Enfin, dans un scénario de forte affluence, la salle d'attente atteint sa limite N et les nouveaux clients sont refusés. Le barbier continue de traiter les clients en attente puis retourne à l'état de sommeil lorsque la file est vide

6 Implémentation et environnement de développement

6.1 Langage, bibliothèques de threads et APIs de synchronisation utilisées

Pour ce projet, on a codé avec Godot Engine, et on a pris GDScript (le langage du moteur) parce qu'il est simple et qu'il se marie bien avec l'environnement de Godot. GDScript nous laissait jouer directement avec les outils de Godot pour gérer plusieurs actions en même temps, ainsi que les signaux et l'affichage.

Pour la concurrence, on s'est servi des classes Thread, Mutex et Semaphore de Godot. On a créé des fils d'exécution pour le barbier et les clients, pendant que le fil principal s'occupait de l'interface et de comment on interagit avec le jeu. Les sémaphores indiquent si des clients sont là et si le barbier est libre, et les mutex évitent que les fils d'exécution se marchent dessus quand ils accèdent aux infos partagées (compteurs, filee d'attente).

6.2 Gestion des threads : création, paramètres, fin d'exécution

Pour faciliter l'observation du comportement concurrent, l'application affiche en temps réel des traces textuelles sur un terminal intégré. La figure 6.1 montre, en mode automatique, la succession des événements liés au minuteur d'arrivée des clients et au déroulement des coupes (création d'un nouveau client, planification du prochain, fin de coupe, etc.). Ces messages permettent de vérifier que le barbier reste actif tant que des clients sont présents et que le rythme d'arrivée est bien contrôlé par les minuteurs. [file :112][web :9]

La figure 6.2 illustre un second type de trace, centré sur les mécanismes de synchronisation. On y visualise, pour chaque client pris en charge, les opérations de verrouillage et de déverrouillage du mutex dans `request_seat()`, les appels aux sémaphores (attente du barbier sur le semaphore de clients, réveil, signalement de l'arrivée d'un client) ainsi que la durée de la coupe. Ce retour visuel permet de confirmer que les sémaphores et mutex sont utilisés conformément au protocole décrit dans le chapitre précédent.



FIGURE 6.1 – Traces automatiques montrant les arrivées de clients et les fins de coupe.



FIGURE 6.2 – Traces de synchronisation mettant en évidence l’usage du mutex et des sémaphores.

6.3 Interface graphique et captures d’écran de l’application

L’interface principale du salon affiche visuellement les clients, les chaises d’attente et le barbier, ce qui permet de suivre facilement l’évolution de la file et l’état du serveur. La figure 6.3 montre une situation avec un nombre limité de clients, où l’on distingue clairement les sièges libres et occupés ainsi que le client en cours de coupe. La figure 6.4 présente un cas de forte affluence, où la salle d’attente est presque saturée, ce qui illustre le rôle de la capacité maximale dans le problème du barbier endormi.

Un panneau de configuration permet également d’ajuster dynamiquement certains paramètres de la simulation, comme la vitesse d’arrivée des clients, la durée des coupes, la vitesse de déplacement ou le comportement général du système. La figure 6.5 montre cet écran de réglage, qui offre à l’utilisateur un moyen simple de reproduire les différents scénarios de test décrits au chapitre suivant.



FIGURE 6.3 – Vue du salon avec quelques clients dans la salle d'attente.



FIGURE 6.4 – Vue du salon en situation de forte affluence.

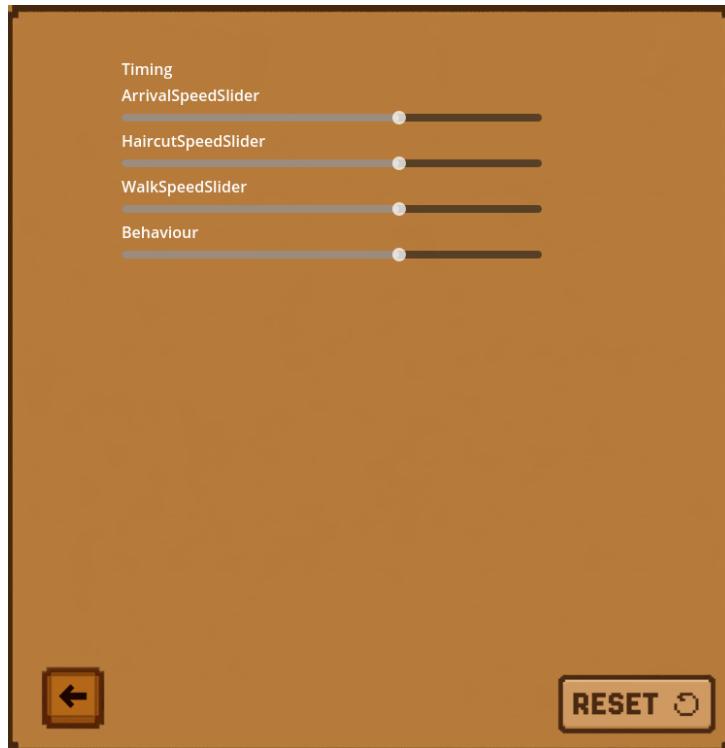


FIGURE 6.5 – Écran de paramètres permettant d’ajuster les vitesses d’arrivée, de coupe et de déplacement.

7 Conclusion

7.1 Bilan du projet

Ce projet a servi à créer une version du problème du barbier qui dort, en utilisant Godot Engine et GDScript. On a pu gérer la logique et l’interface graphique. En combinant des threads, des sémaphores, des mutex et une file fifo on a bien fait marcher les interactions entre le barbier et les clients, tout en respectant les règles du problème.

Les tests ont montré que notre solution marche bien : on gère correctement le nombre de places dans la salle d’attente, les clients sont servis dans l’ordre d’arrivée, et les ressources sont protégées pour éviter les problèmes de concurrence et les blocages. En gros, ce projet montre comment on peut prendre des idées théoriques de synchronisation et les utiliser dans un moteur de jeu pour simuler des systèmes où plusieurs choses se passent en même temps.

7.2 Pistes d’amélioration

Avec ça comme point de départ, on peut imaginer plein d’améliorations. Déjà, on pourrait étendre le modèle à plusieurs coiffeurs qui bossent en même temps. Faudrait juste adapter la file d’attente et la façon dont on gère les clients pour que chacun ait son coiffeur sans que ça parte en cacahuète. On pourrait aussi tester d’autres façons de gérer les clients (priorité, plusieurs files, temps de coupe variable) et voir ce que ça change niveau rapidité et équité. Techniquement, si on utilisait des outils de mesure plus précis

(pour voir ce que font les threads, avoir des stats sur les temps d'attente), on pourrait mieux regarder comment l'appli se comporte et mieux gérer les threads et les données. Et pour finir, on pourrait améliorer l'interface avec des graphiques (charge, historique des files) pour que l'utilisateur comprenne mieux ce qui se passe.

Références

Bibliographie

- [1] Godot Engine, *Documentation officielle de Godot Engine*, consulté en 2025, disponible sur <https://docs.godotengine.org/>.
- [2] W. Stallings, *Operating Systems : Internals and Design Principles*, 8^e édition, Pearson, 2014. (Ouvrage de référence sur les mécanismes de synchronisation, les sémaphores et les verrous pour la programmation concurrente.)
- [3] A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10^e édition, Wiley, 2018. (Présente les problèmes classiques de synchronisation, dont le barbier endormi, et les primitives de synchronisation de type sémaphore et mutex.)
- [4] K. Belarbi, B. Djellali, support de cours
Université des Sciences et de la Technologie d'Oran année universitaire 2025/2026.
(Utilisé pour la formalisation des sections critiques, de l'attente bornée et des sémaphores.)
- [5] Divers outils d'intelligence artificielle générative, *Génération d'assets graphiques pour la simulation du barbier endormi* (images de personnages, éléments de décor), utilisés comme support visuel dans le projet.
- [6] Dépôts GitHub publics, *Exemples de projets de simulation de problèmes de synchronisation et d'utilisation de threads sous Godot*, consultés pour s'inspirer de la structure de code et des bonnes pratiques.