HEIG-VD | PCO - Laboratoire 5

Modélisation par moniteur de Mesa

Auteurs : Harun Ouweis & Christen Anthony
Date : 20.12.2023

Introduction

Dans le cadre de ce laboratoire, nous sommes chargés de compléter un programme C++ simulant la gestion d'un salon de coiffure. Le barbier et chaque clients sont contrôlés par un thread qui leur est propre.

Fonctionnalités clés du logiciel :

- Multi-threading : Chaque personnage est "matérialisé" par un thread permettant des opérations parallèles et synchronisées.
- Synchronisation : Les accès aux états critiques sont synchronisés pour un déroulement cohérent de la simulation.
- Gestion des ressources : Les ressources sont protégées vis-à-vis des accès concurrents.
- Interface de suivi : Une intreface graphique, qui nous a été fournie, permet de visualiser l'évolution du programme et de ses états. Des consoles permettent de voir par quels états passent les différents personnages.
- Terminaison propre : La simulation se termine de manière ordonnée, en s'assurant que tous les threads concluent leurs tâches dans un état stable.

Objectif

L'objectif de ce projet est de mettre en œuvre la logique de contrôle d'un salon de coiffure qui permet à la simulation de fonctionner de manière cohérente et de protéger les potentiels accès concurrents aux sections critiques de la simulation comme la chaise de travail et la salle d'attente. Les données permettant le fonctionnement de la simulation doivent également être protégées vis-à-vis des accès concurrents potentiels.

Implémentation de notre solution

`Barber`

La classe `Barber` permet de modéliser le barbier du salon de coiffure. Nous avons dû implémenter la méthode `run()` qui est exécutée par le thread représentant le barbier. Des appels aux méthodes propres au barbier de la classe `PcoSalon` sont effectués afin de faire passer le barbier d'un état à l'autre de la simulation.

`Client`

De manière semblable àl a classe `Barber` , la classe `Client` permet de modéliser les clients du salon de coiffure. Nous avons dû implémenter la méthode `run()` qui est exécutée par les threads représentant les clients. Des appels aux méthodes propres aux clients de la classe `PcoSalon` sont effectués afin de faire passer les clients d'un état à l'autre de la simulation.

`PcoSalon`

Cette classe permet de gérer la synchronisation entre le babrier et les différents clients du salon. Des méthodes propres au barbier et propres aux clients ainsi que des méthodes permettant de mettre à jour l'affichage de la simulation s'y trouvent.

Accès au salon et attente

Un client peut accéder au salon seulement s'il reste au moins une place assise. S'il ne peut pas y accéder, il part faire un tour et retentera sa chance plus tard. S'il peut accéder au salon, il est placé dans la liste `_clients` . Cette liste nous permet de conaître l'ordre d'arrivée des clients. Si le client est le premier de la liste (`_clients.front()`), il peut se rendre sur la chaise de travail pour se faire couper les chevuex. Sinon, il doit attendre son tour sur l'une des chaises de la salle d'attente. La mise en attente d'un client s'effectue en appelant la méthode `wait()` sur la variable de condition (`_clientTurn`). Etant donné que lorsqu'un `wait()` est effectué le mutex est relâché et doit être ré-acquis une fois le thread libéré, une boucle `while` englobe la mise en attente afin de retester la condition d'attente (`_clients.front != clientId`) et d'éviter tout problème dû à une préemption par un autre thread.

Coupe des cheveux

Lorsque le client peut se rendre sur la chaise de travail pour se faire couper les cheveux, le barbier l'attend à l'aide de la variable de condition `_clientSeated` . La variable boléenne `_clientSupported` est mise à jour. Une fois le client assis sur la chaise de travail, il libère (`notifyOne()`) le barbier de son attente. Le client attend alors que la coupe soit terminée à l'aide de la variable de condition `_beautify` . Une fois que le barbier a terminé son travail, il `notifyOne()` le

client que la coupe est terminée. Le client est alors retiré de la liste des clients (`_clients`) et il part faire un tour le temps que ses cheveux repoussent. La variable boléenne `_clientSupported` est mise à jour.

Au suivant !

Lorsque le barbier souhaite informer ses clients qu'il est disponible, il appelle la méthode `pickNextClient` . La variable booléenne `_clientSupported` permet de savoir si un client est actuellement pris en charge par le barbier. Elle permet d'effectuer un `notifyAll()` sur la variable de condition `_clientTurn` que lorsque c'est vraiment nécessaire. Ainsi, tous les clients en attente sont libérés et testent à nouveau la condition (`_clients.front != clientId`). Si le client est le premier de la liste, il est pris en charge par le barbier. Sinon, il se remet en attente en restant sur sa chaise d'attente.

Sieste

Lorsqu'il n'y a plus de clients dans le salon, le barbier se rend dans l'arrière boutique afin d'y effectuer un petit somme. Son thread est mis en attente sur la variable de condition `_barberSleep` . Le premier client à arriver le réveillera en effectuant un `notifyOne()` sur `_barberSleep` . Le barbier étant réveillé, les autres clients ne chercheront pas à le réveiller.

Arrêt de la simulation

Lorsque l'utilisateur effectue une entrée en console, la variable booléenne `inService` permet d'indiquer que le salon est fermé. Les clients se trouvant à l'extérieur rentrent chez eux. Le babrier s'occupe des clients se trouvant dans le salon avant de terminer sa journée (fin de la simulation). Si le barbier est en train de dormir alors que l'utilisateur effectue une entrée, il est réveillé (`notifyOne()`) afin que son thread puisse se terminer.

Les 4 variables de conditions (`_clientTurn` , `_clientSeated` , `_beautify` et `_barberSleep`) utilisées et décrites précédemment se comportent de la manière suivante : le barbier libère un/des client/s et un client libère le barbier.

Animations

Dans un premier temps, nous avons rencontré quelques problèmes lors de la mise à jour de l'interface graphique. Etant donné que les animations s'effectuent en dehors du moniteur, nous nous sommes retrouvés confrontés à des situations incohérentes. Ensuite, nous avons pu les placer aux bons endroits afin de rendre le système fontionnel.

Remarques

La fonction `endService` réveille (`notifyOne()`) le barbier s'il dort afin de libérer son thread de son état "waiting" et de faire en sorte qu'il puisse se terminer. Si nous ne faisions pas cela, le thread resterait en attente et le programme ne se terminerait pas.

La variable booléenne `_clientSupported` n'est pas indispensable pour le bon fonctionnement de notre programme mais nous avons décidé de la consever pour des raisons de performances afin d'effectuer un appel `notifyAll()` sur la variable de condition `_clientTurn` uniquement lorsque cela est nécessaire.

Concernant l'interface graphique, les clients qui doivent se mettre en attente ne vont pas nécessairement s'asseoir sur la chaise d'attente dont l'index correspond à leur position d'arrivée - 1. Autrement dit, le premier client arrivé n'ira pas nécessairement s'asseoir sur la première chaise d'attente (index = 0). Cela est dû au fait que, dans la méthode `accessSalon()` , l'animation `animationClientAccessEntrance()` est placée entre l'ajout du client dans la liste des clients (`_clients`) et l'obtention d'une chaise libre. Etant donné que le mutex est perdu et réaquis dans l'animation, l'ordre dans lequel les chaises sont attribuées aux clients peut différer de l'ordre dans lequel ils sont arrivés. En revanche, les clients sont bien pris en charge par le barbier dans leur ordre d'arrivée.

Diagramme de synchronisation

Le diagramme de synchronisation que nous avons élaboré dans la Figure 1 détaille précisément comment notre simulation de salon de coiffure gère la synchronisation. Il montre l'interaction entre les threads des clients et du barbier, qui doivent coordonner l'accès à des ressources partagées comme les chaises.

Ce flux d'opérations correspond à ce qui est décrit dans le diagramme de synchronisation : les threads clients et barbier exécutent leurs actions dans un ordre logique et synchronisé, assurant que les ressources partagées (les chaises du salon) sont accédées de manière ordonnée et sans conflit. Les animations intégrées via les méthodes telles que animationClientAccessEntrance() et animationBarberCuttingHair() ajoutent un élément visuel à l'interaction, rendant le processus plus compréhensible pour les observateurs.

Il est essentiel de comprendre que chaque wait() qui se termine est le résultat direct d'un notify() depuis une autre partie du code, illustrant l'interdépendance entre les threads dans ce système complexe. Mais pour de raison de lisibilité nous avons préféré montré chaques parties individuellements.

Tests effectués

Les tests décrits ci-dessous ont été effectués afin de s'assurer du bon focntionnement de la solution que nous avons mise en place.

Description	`NB_CLIENTS`	`NB_SIEGES`	Remarques	Résultat
Valeurs fournies par défaut	10	2	/	OK
Aucun client	0	2	Le barbier dort jusqu'à ce que la simulation soit arrêtée par une entrée utilisateur.	OK
1 client	1	2	Le barbier dort entre chaque coupe des cheveux du client. Le client le réveil à chacune de ses venues.	OK
Aucun siège	10	0	Les clients entrent un par un.	OK
1 siège	10	1	/	OK
Nombre de sièges d'attente égal au nombre de clients	10	10	Tous les clients peuvent entrer dans le salon.	OK
Nombre de sièges d'attente supérieur au nombre de clients	10	15	Tous les clients peuvent entrer dans le salon.	OK
Arrêt de la simulation alors qu'il y a des clients en attente.	10	2	Le barbier s'occupe des clients en attente avant de fermer son salon. Fin de la simulation.	OK
Arrêt de la simulation alors que le barbier dort.	1	2	Le barbier se réveille et ferme son salon. Fin de la simulation.	OK

Description	`NB_CLIENTS`	`NB_SIEGES`	Remarques	Résultat
Arrêt de la simulation alors que le barbier attend le client.	2	2	Le barbier s'occupe du client et ferme son salon. Fin de la simulation.	OK

Nous avons également laissé tourner la simulation durant des laps de temps conséquents afin de s'assurer qu'elle fonctionne correctement sur la durée. Pour chacun des tests effectués, à aucun moment plusieurs clients se sont retrouvés sur la même chaise (d'attente et de travail). Il n'y a pas non plus eu plus de clients qu'il n'y a de chaises dans le salon (`_capacity`) et les clients ont été pris en charge dans leur ordre d'arrivée.

Tous les tests effectués sont passés avec succès !

Conclusion

Ce laboratoire a été l'occasion de consolider notre compréhension des techniques de synchronisation et de gestion des accès concurrents dans un contexte multi-threadé. Nous avons pu mettre en pratique l'utilisation d'un moniteur de Mesa et ainsi consolider notre compréhension de son fonctionnement. La gestion d'une interface graphique mise à jour en dehors du moniteur a été une contrainte supplémentaire qui a été très intéressante à gérer. Notre solution, fruit d'une conception minutieuse et d'une série de tests rigoureux, démontre l'importance d'une approche méthodique dans le développement de systèmes parallèles fiables.

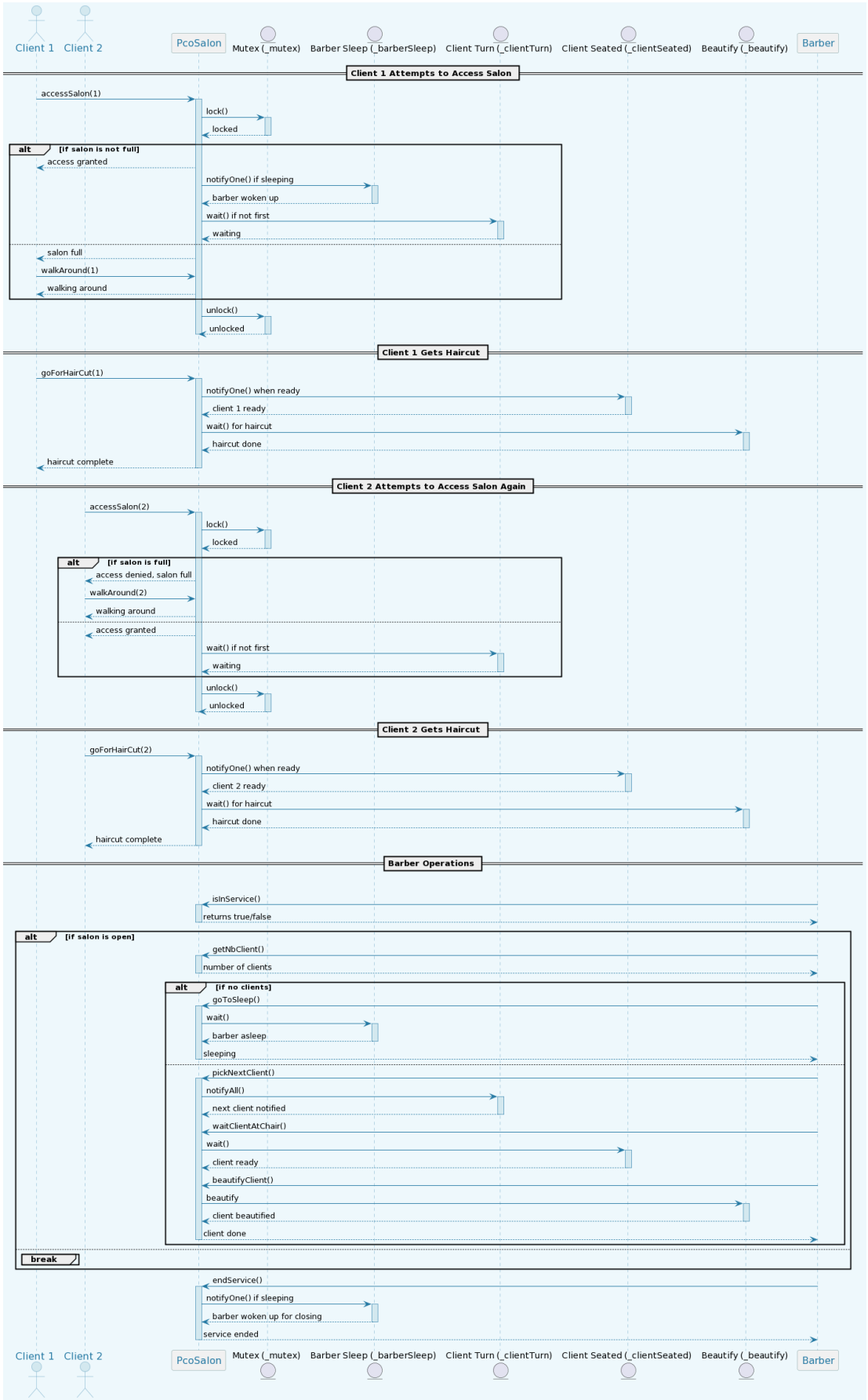

Figure 1 - Diagramme de synchronisation élaboré pour le laboratoire 5