**Programmation Orientée Objet – Prosit 4**

*Prosit Aller : 10/11/2021*

Animateur : Ilhan

Scribe : Mattéo

Secrétaire : Olivier

Gestionnaire : Pierre

**Prendre connaissance de la situation et clarifier**

*Mots-clés / à définir\**

* **Espace Hertzien\* :**Les ondes hertziennes sont une modification des champs électromagnétiques. Elles se propagent à l'image des ondes sur une mare. Leurs longueurs d'onde et leurs fréquences codent toutes sortes d'informations dont la voix et l'image.
* **Communication multiplexée\*** :Le multiplexage (appelé parfois muxing en anglais) est un moyen d'envoyer simultanément plusieurs signaux ou flux d'informations sur une même liaison de communication, sous la forme d'un signal unique et complexe.
* **Multiplexeur\***: Système qui permet d'effectuer un multiplexage.
* **Pool\***: Informatique. Un pool désigne un ensemble de ressources réutilisables, géré de façon commune pour un ensemble d'usagers (processus informatique, utilisateurs, ordinateurs, etc.).
* **Emettre\***: Mettre en circulation
* **Diffuser\***: Répandre dans toutes les directions.
* **Variable d’instance de classe\***: toute instance a sa propre copie de ces variables d'instance déclarées dans la classe, chaque objet contrôle ses propres données, l'ensemble des valeur des varaibles d'instance forme ce que l'on appel l'état de l'instance.
* **Buffer\***: En informatique, mémoire intégrée dans des périphériques (disque dur, imprimante, lecteur de cédérom, scanner…) qui permet un stockage temporaire de données afin de faciliter les échanges d'informations et d'optimiser les flux de données.
* **Bloc chaîné\***: Une blockchain, ou chaîne de blocs, est une technologie de stockage et de transmission d'informations sans organe de contrôle.
* **Signal**
* **Asynchrone**
* **Pointeur**
* **OS**
* **Station FM**
* **Classe Emetteur / Récepteur**
* **Discontinuité**
* **Fréquence**
* **Seuil mémoire**

*Établir le contexte*

Simulation informatique qui a pour but de reproduire le comportement de communications multiplexées dans l’espace Hertzien.

Saut de page

**Analyser le besoin**

*Analyser et décomposer le problème*

* Comment chaîner les blocs ?
* Comment éviter que le programme dépasse le seuil mémoire ?
* Comment mettre en place une mise en écoute asynchrone ?
* Comment la méthode d’une classe A peut appeler la méthode d’une classe B sans la connaître ?
* Comment multiplexer un signal ?

*Livrables*

* Programme fonctionnel à Simulation
* Note explicative
* Diagramme de classe

*Contraintes*

* Asynchrone
* Seuil mémoire
* Espace Hertzien
* « Une classe « émetteur » (et d'une méthode « diffuser ») et d'une classe « récepteur » (et d'une méthode « écouter »). La classe « émetteur » dispose d'une instance de la classe « récepteur » et lors de l'appel de ma méthode « diffuser », un appel à « écouter » est levé via la variable d'instance de la classe « récepteur » » **(partie du texte du prosit)**

**Généraliser**

*Saisir la nature du problème et déterminer la famille de problèmes à laquelle il appartient*

* Classes
* Gestion de la mémoire

**Trouver des pistes de solutions**

*Identifier des approches de solutions possibles*

* On pourrait stocker les messages en utilisant les listes chaînées (Florian)
* On va utiliser les pointeurs (Célian)
* Les listes chaînées seront utilisées depuis des classes à la place des structures (Florian)
* On va utiliser de l’allocation chaînée (Célian)

**Elaborer le plan d’actions**

*Identifier les éléments d’étude nécessaires à la résolution du problème*

* Allocation chaînée
* Gestion de la mémoire à le tas (2h avec le dessus)
* Synchrone / Asynchrone (30 min – 1h)
* Pool (3-4h pour le reste à partir d’ici)
* Pointeurs
* Destructeur
* Buffer de mémoire

*Identifier les éléments de réalisation à produire*

* Programme fonctionnel à Simulation
* Note explicative
* Diagramme de classe

*Liste, organiser, planifier et formaliser les actions à mener pour résoudre le problème*

* Colliger les informations (7h)
* Réaliser les diagrammes (1h)
* Faire le programme fonctionnel + tester la simulation (15h)
* Ecrire la note explicative (5h)
* Corbeille d’exercices (4h)

**Phase AER :**

Dans ce prosit, nous avons vu essentiellement les notions suivantes :

* Les pointeurs de fonctions ;
* Les notions de thread et de variables atomic ;
* Les fonctions async.

Nous allons commencer par les pointeurs de fonctions. Comme une fonction n'est pas un objet ou une variable, il n'est pas possible de passer une fonction directement en argument à notre fonction minimum. Cependant, comme toutes les choses présentes dans un programme C++, les fonctions ont une adresse. Il est donc possible de déclarer un pointeur vers cette adresse et de passer ce pointeur à la fonction minimum.

Pour déclarer un pointeur sur fonction, il faut utiliser la syntaxe suivante :



Il faut savoir que comme toute fonction, nous ne sommes pas obligés de mettre des paramètres (ou arguments). Les types de retours sont les même que les fonctions classiques. Un pointeur sur fonction est un type comme un autre, nous pouvons donc créer un tableau de pointeur sur fonction avec les vector (dans le namespace std) par exemple ou avec un tableau classique. Pour rappel un vector est un type de donnée en une dimension (par défaut), ce qui revient à faire un tableau. Cependant il y a une différence entre les deux :

* Pour un **tableau**, **nous connaissons la taille** de celui-ci lors de la compilation ;
* Pour un **vector**, **leur taille peut changer** au fur et à mesure.

Nous allons maintenant voir comment affecter un pointeur sur fonction. En effet, nous parlons de pointeur sur fonction, il est donc très facile d’affecter un pointeur à une fonction. Il faut simplement récupérer l’adresse mémoire d’une fonction et l’affecter à un pointeur. Pour rappelle les opérateur des adresses et des pointeurs sont les suivants :

* & pour les adresses ;
* \* pour les pointeurs.

Une image contenant texte

Description générée automatiquement

Il faut savoir qu’en C++ il n’est pas forcément obligatoire d’utiliser les opérateurs &.

Nous allons maintenant voir comment utiliser une fonction pointée.

Une image contenant texte

Description générée automatiquement

Comme nous pouvons le voir, avant le main, nous déclarons une fonction. Puis dans le main, nous déclarons un pointeur qui à pour paramètre deux int. Nous affectons la fonction au pointeur. Puis les variables int resultat et resultat\_2 prenne la valeur de retour de la fonction pointée.

Il faut savoir que nous utilisons un pointeur sur fonction de la même manière que nous utilisons la fonction pointée.

Nous allons maintenant voir les cas particuliers des fonctions membres.

Une image contenant texte

Description générée automatiquement

Comme nous pouvons le voir, lorsque nous créons une classe avec une méthode, pour faire appelle à cette méthode, nous devons bien faire appel à la classe aussi.

Une image contenant texte

Description générée automatiquement

Voici comment nous utilisons un pointeur alors que nous utilisons aussi un objet.

Nous pouvons remarquer aussi que nous devons forcément utiliser l’opérateur \* pour les pointeur et les parenthèses.

Nous allons maintenant passer à la notion de thread et de variables atomic. Un thread (que l'on pourrait traduire par "fil d'exécution") est, en gros, une séquence d'instructions qui s'exécutent parallèlement aux autres threads. Chaque programme est constitué d'au moins un thread : le thread principal, qui fait tourner notre fonction main(). Les programmes qui utilisent uniquement le thread principal sont monothreadés, si nous leur ajoutons un ou plusieurs threads alors ils deviennent multithreadés. Donc, pour résumer, les threads offrent un moyen de faire plusieurs choses en même temps. Cela peut être utile, par exemple, pour afficher une animation et réagir aux actions de l'utilisateur pendant le chargement d'images ou de sons. Les threads sont aussi très utilisés en programmation réseau, afin d'attendre que des données soient reçues tout en continuant à mettre à jour et afficher l'application.

Une image contenant texte

Description générée automatiquement

Une image contenant texte, équipement électronique, capture d’écran, moniteur

Description générée automatiquement

Comme nous pouvons le voir, il faut tout d’abord initialiser le thread. Puis il faut le démarrer.



Une fois que nous avons créer le thread avec sf ::Thread nous n’avons plus qu’à le démarrer avec sa fonction launch. launch appelle dans un nouveau thread la fonction qui a été passée au constructeur, et rend la main immédiatement de façon à ce que le thread appelant puisse continuer à tourner en parallèle.

Un thread s'arrête tout seul lorsque sa fonction se termine. Si nous voulons attendre qu'un thread se termine depuis un autre thread, nous pouvons appeler sa fonction wait. La fonction wait est également implicitement appelée par le destructeur de sf::Thread, de façon à ce qu'un thread ne puisse pas continuer à tourner (et être hors de contrôle) après que l'instance de sf::Thread qui l'a créé ait été détruite. Nous devons bien garder cela en tête lorsque nous gérons nos threads (cf. la dernière partie de ce tutoriel).

Une image contenant texte

Description générée automatiquement

Il n'existe aucune fonction dans sf::Thread qui permette de mettre en pause un thread depuis un autre, la seule façon est de le faire directement depuis le code qui tourne dans le thread. En d'autres termes, on ne peut mettre en pause que le thread courant. Pour ce faire, nous pouvons appeler la fonction sf::sleep :



sf::sleep prend un paramètre, qui est la durée de la pause. Cette durée peut être donnée avec n'importe quelle unité/précision, comme expliqué dans le tutoriel sur le temps.

Nous pouvons mettre en pause n'importe quel thread, même le thread principal. sf::sleep est la façon la plus efficace d'attendre dans un thread : tant que le thread "dort", il n'utilise pas le processeur. Les pauses basées sur de l'attente active, comme des boucles while vides, utiliseraient 100% du processeur juste pour... ne rien faire. Cependant, il faut garder en tête que la durée d'attente est juste une indication, selon l'OS cela donnera un résultat plus ou moins précis. Ne ne pouvons donc pas compter sur cette fonction pour produire des timings hyper précis.

Tous les threads d'un programme partagent la même mémoire, ils peuvent accéder à toutes les variables du programme. C'est très pratique mais aussi dangereux : puisque tous les threads tournent en parallèle, cela signifie qu'une variable ou une fonction pourrait très bien être utilisée depuis plusieurs threads en même temps. Et si l'opération en question n'est pas thread-safe, le résultat sera indéterminé (c'est-à-dire que cela pourrait planter ou corrompre des données).

Il existe plusieurs outils de programmation pour vous aider à protéger les variables partagées et rendre notre code thread-safe, ils sont appelés "primitives de synchronisation". Les plus communs sont les mutexs, les sémaphores, les conditions d'attente et les spin locks. Ils sont tous des variations du même concept : ils protègent un morceau de code en autorisant leur accès uniquement à certains threads, tout en bloquant les autres.

La primitive de synchronisation la plus basique (et la plus utilisée) est le mutex. Mutex signifie "EXclusion MUTuelle" : il autorise un seul thread à la fois à accéder aux morceaux de code qu'il entoure. Voyons voir comment un mutex pourrait remettre un peu d'ordre dans le premier exemple que nous avons vu :

Une image contenant texte

Description générée automatiquement

Ce code utilise une ressource partagée (std::cout), et comme nous l'avons vu cela produit un résultat que nous souhaiterions éviter -- tout est mélangé dans la console. Pour nous assurer que les blocs de texte des deux threads sont correctement affichées au lieu d'être mélangées aléatoirement, nous protégeons les morceaux de code correspondant avec un mutex.

Le premier thread à atteindre sa ligne mutex.lock() verrouille le mutex, accède directement au code qui suit et affiche son texte. Lorsque l'autre thread atteint sa ligne mutex.lock(), le mutex est déjà verrouillé et le thread est donc mis en attente (de la même manière qu'avec un appel à sf::sleep, le thread qui dort n'utilise pas du tout le processeur). Lorsque le premier thread déverrouille enfin le mutex, le second thread est réveillé et est autorisé à son tour à verrouiller le mutex et à afficher ses lignes de texte. Les blocs de texte apparaissent donc séquentiellement dans la console, plutôt que mélangés.

Une image contenant texte, équipement électronique, capture d’écran, afficher

Description générée automatiquement

Le mutex n'est pas la seule primitive que nous pouvons utiliser pour protéger nos variables partagées, mais devrait être suffisant pour la plupart des cas.

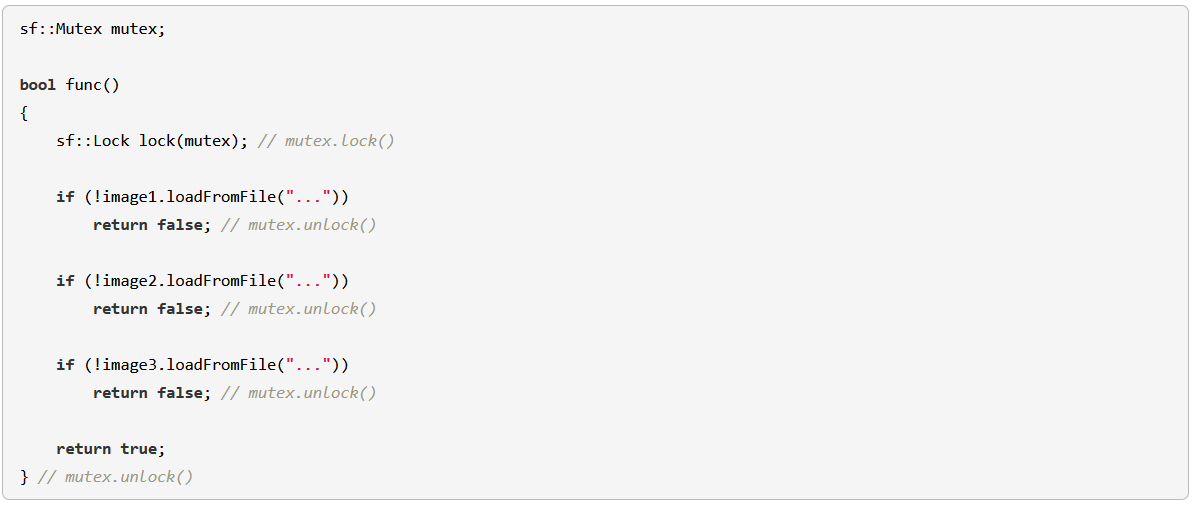
Les mutexs sont déjà thread-safe, il n'y a pas besoin de les protéger. Mais ils ne sont pas exception-safe ! Que se passe-t-il si une exception est lancée pendant qu'un mutex est verrouillé ? Il ne peut pas atteindre la ligne qui le déverrouille, et reste verrouillé pour toujours. Et tous les threads qui essayeront ensuite de le verrouiller seront bloqués pour toujours à leur tour ; notre application toute entière pourrait se retrouver bloquée. Pas cool.

Pour s'assurer que les mutexs sont toujours déverrouillés dans un environnement où des exceptions peuvent être lancées, SFML fournit une classe RAII pour les encapsuler : sf::Lock. Elle verrouille le mutex dans son constructeur, et le déverrouille dans son destructeur. Simple et efficace.

Une image contenant texte

Description générée automatiquement

Nous pouvons noter que sf::Lock peut aussi se révéler utile dans une fonction qui a plusieurs instructions return.



Nous allons maintenant parler des variables atomic. Dans un premier temps, nous allons expliquer l’atomicité. C’est une propriété utilisée en programmation concurrente pour désigner une opération ou un ensemble d'opérations d'un programme qui s'exécutent entièrement sans pouvoir être interrompues avant la fin de leur déroulement. La programmation concurrente est un paradigme de programmation tenant compte, dans un programme, de l'existence de plusieurs piles sémantiques qui peuvent être appelées threads, processus ou tâches. Elles sont matérialisées en machine par une pile d'exécution et un ensemble de données privées.

En règle générale, les instructions assembleur peuvent être considérées comme atomiques, mais cela ne suffit pas pour assurer une cohérence des données. Pour ce faire, des mécanismes de synchronisation doivent être mis en œuvre. Par conséquent et par extension, il est possible de dire qu'une section protégée de la sorte, qualifiée de section critique protégée par une exclusion mutuelle, est aussi atomique.

Les objets de types atomiques sont les seuls objets C++ libres de courses de données; En d'autres termes, si un thread écrit dans un objet atomique pendant qu'un autre thread en lit, le comportement est bien défini.

De plus, les accès aux objets atomiques peuvent établir une synchronisation inter-thread et ordonner des accès à la mémoire non atomique comme spécifié par \_std::memory\_order\_. \_std::atomic<>\_ encapsule les opérations qui, 11 fois avant C++, devaient être effectuées avec (par exemple) fonctions inter-verrouillées avec MSVC ou bultins atomiques dans le cas de GCC .

De plus, \_std::atomic<>\_ nous donne plus de contrôle en autorisant divers ordres de mémoire qui spécifient des contraintes de synchronisation et d’ordre.

Une image contenant texte

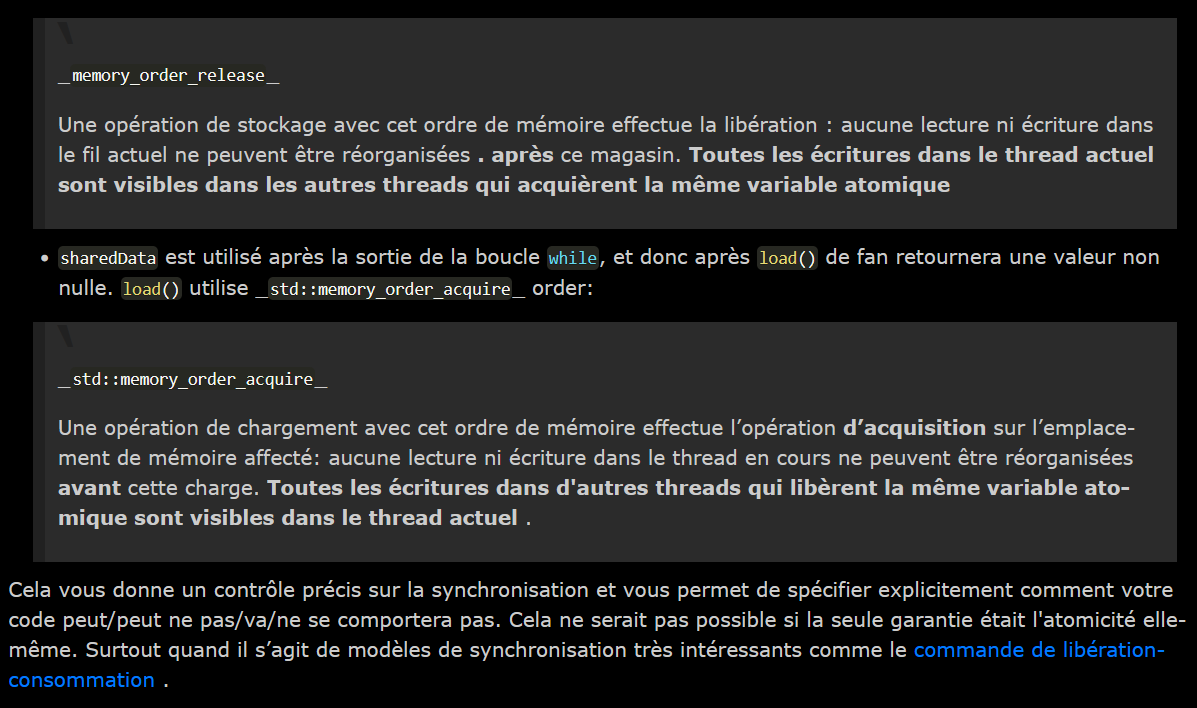
Description générée automatiquement

Une image contenant texte, capture d’écran, moniteur, écran

Description générée automatiquement

Une image contenant texte

Description générée automatiquement



Pour finir, nous allons maintenant parler de la désynchronisation avec la fonction async. Si nous avons besoin du résultat d’une opération asynchrone, alors vous devez bloquer, quelle que soit la bibliothèque que vous utilisez. L'idée est que nous devons choisir le moment où nous voulons bloquer et, espérons-le, ce faisant, nous bloquons pour une durée négligeable, car tout le travail a déjà été effectué.

Notons également que std::async peut être lancé avec les stratégies std::launch::async ou std::launch::deferred. Si nous ne le spécifions pas, l'implémentation est autorisée à choisir, et elle peut également choisir d'utiliser une évaluation différée, ce qui entraînerait tout le travail à effectuer lorsque nous essayons d'obtenir le résultat de l'avenir, ce qui entraînerait un blocage plus long. Si nous voulons nous assurer que le travail est effectué de manière asynchrone, utilisez std::launch::async.

Nous allons maintenant voir la corbeille :

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement