Référence/type[1]			☐ TD ☒ AP
Thématique principale	Programmation multithread en C++ (moderne)		
Thématique(s) secondaire(s)	✓ Les différents types de threads en C++23 (since C++20 au minimum) ✓ Les différents outils de synchronisation de threads et de protection des ressources en C++ moderne ✓ RAII		
Durée	2h	***Niveau[2]***	A
Prérequis	✓ Connaissances de base en programmation parallèle (threads, synchronisation, protection des ressources…). Cf cours de SE en ISIE3		
Compétences opérationnelles et niveau cible[3]	✓ Créer, exécuter, mettre en sommeil, réveiller, arrêter un jthread (***A***)		
Modalités et critères d'évaluation	Auto-évaluation		
Matériel(s), équipement(s), composant(s) nécessaires			
Logiciel(s) nécessaire(s)[4]			
Ressources	https://en.cppreference.com/w/cpp/thread https://en.cppreference.com/w/cpp/thread/jthread/jthread		

Avant la séance…

Revoir, si nécessaire, les éléments de cours (S.E. ISIE3 par exemple) sur la programmation parallèle, notamment ce qui concerne le multithreading (gestion de threads), les problématiques de synchronisation et de protection des ressources (mutex, sémaphore, signaux, messages…).

- Récupérer le fichier « demoThread.cpp » disponible sur Teams ou copier le contenu dans un fichier source local. Analyser le code proposé et essayer de déterminer le rôle de chacune des lignes.
- Produire l'exécutable en activant la macro « THREAD_VERSION », exécuter et observer le comportement.
- Mettre en commentaires les deux lignes appelant les méthodes *join* des deux threads. Produire l'exécutable, exécuter et observer le nouveau comportement. Expliquer.
- Produire l'exécutable en activant la macro « JTHREAD_VERSION », exécuter et observer le comportement. Expliquer la différence entre thread et jthread.

[1] TD : ½ groupe, apports de cours, mise en pratique guidée – TP : ½ groupe, travail autonome majoritaire.
[2] Le niveau se rapporte à la thématique principale. Il peut ici être entendu comme un indicateur de la difficulté du travail (N-Facile, A-Sans grande difficulté, M-Quelques points complexes, MA-Difficile)
[3] Les niveaux cible correspondent pour chaque compétence opérationnelle au niveau d'acquisition de la celle-ci à l'issue du travail dans son intégralité (en incluant les phases préparatoires et de synthèse).
[4] En plus d'un environnement de développement C++

Travail encadré en séance

Apports de cours

Activités

Activité 1 – Découverte des objets std::thread et std::jthread

- Préciser, à partir des observations faites lors de la préparation de la séance, la différence fondamentale entre objets std::thread et std::jthread.
- Prendre pour acquis le fait qu'en C++23 seuls les jthread sont à utiliser.

Activité 2 – Cancellation / Cooperative cancellation

- En s'inspirant du code proposé précédemment, mettre en place un programme mettant en évidence une manière simple (simpliste), à partir d'un thread quelconque (y compris le main) d'arrêter un autre thread. Identifier avantages et inconvénients de la méthode.
- Modifier le programme afin d'obtenir le même comportement en utilisant des objets de la STL. On parle alors de cooperative cancellation, et d'objets de type *std::stop_source* et *std::stop_token*. Evaluer les avantage / inconvénients par rapport à la méthode « basique ».

Activité 3 – Protection de ressources via smart mutex

Le mécanisme présenté ci-après se base sur deux objets de la STL :

- ✓ Le ***std::mutex*** : objet que l'on peut qualifier de smart mutex (c'est autre chose qu'un flag…)
- ✓ Le ***std::scoped_lock*** : objet qui permet de s'approprier (« locker ») un mutex. En profitant du fait que le scoped_lock soit un objet RAII, il est possible de protéger une ressource contre les accès concurrents en ayant la garantie que la ressource sera libérée après utilisation, sans se préoccuper de cette libération (pas de « unlock »).

- Essayer de préciser/expliquer à la fois la mécanique de lock/unlock et l'apport de la RAII dans ce cas précis.
- Imaginer et produire un programme mettant en évidence le fonctionnement des mutexes et des scoped_lock.

Activité 4 – Waiting for data… (aka waiting is always a waste of time…)

Dans les applications multithread, il arrive souvent qu'un processus (thread) doive être réveillé de manière synchrone par un autre thread. Cette situation se produit souvent lorsqu'un thread est chargé de mettre à jour une donnée et un autre de la traiter.

En fonction des environnements et outils, plusieurs solutions peuvent être utilisées (les signaux notamment s'ils sont pris en charge) mais en C++ moderne l'enjeu est de permettre une attente la moins gourmande en ressources temporelles. Ainsi surveiller de manière active le positionnement d'un drapeau ou se baser sur la libération d'un mutex sont des solutions à éviter. Elles nécessitent que le thread soit actif (surveillance de drapeau) ou en semi-activité (attente de libération de mutex). On parle de *busy waiting*. Les « condition variables » optimisent ce temps d'attente.

- Prendre les informations sur l'utilisation de ces objets (std::condition_variable)
- Mettre en place un programme de démo basé sur :

- Un thread « AddData » dont le rôle est d'ajouter périodiquement (la période est un paramètre du thread) une valeur entière aléatoire(comprise entre -5000 et +5000) à un vector (ressource partagée).
- La création de deux instances de ce thread avec des périodes différentes.
- Un thread « SortData » chargé de trier les valeurs de ce vector dès qu'une nouvelle valeur y a été ajoutée.

Activité 5 – Préparation AP5

L'AP5 étant la mise en œuvre de ces concepts, s'il reste du temps, démarrer le travail.

Synthèse

Après la séance…