Fiche méthodologique de projet

C.15 Concevoir des objets actifs concurrents

Objectif

Appréhender les mécanismes de synchronisation.

Préambule

L'utilisation de threads dans une application oblige à mettre en place des mécanismes de synchronisation dans la communication entre tâches.

Sémaphore

Un sémaphore permet de **protéger l'accès à une variable partagée** (ou une zone de mémoire partagée) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) dans un environnement de programmation concurrente. Le sémaphore a été inventé par Edsger Dijkstra.

La valeur d'un sémaphore est le nombre d'unités de ressource (exemple : imprimantes...) libres ; s'il n'y a qu'une ressource, un sémaphore à système numérique binaire avec les valeurs 0 ou 1 est utilisé.

Un **sémaphore bloquant** est un sémaphore qui est initialisé avec la valeur 0. Ceci a pour effet de bloquer n'importe quel thread qui effectue P(S) tant qu'un autre thread n'aura pas fait un V(S). Cette utilisation des sémaphores permet de réaliser des **barrières de synchronisation**.

Il existe également le **sémaphore binaire** qui est une **exclusion mutuelle** (voir mutex). Il est toujours initialisé avec la valeur 1.

Problème des lecteurs/rédacteurs

Ce problème traite de l'accès concurrent en lecture et en écriture à une ressource. Plusieurs processus légers (thread) peuvent lire en même temps la ressource, mais il ne peut y avoir qu'un et un seul thread en écriture.

Problème des producteurs/consommateurs

Lorsque des processus légers souhaitent communiquer entre eux, ils peuvent le faire par l'intermédiaire d'une file. Il faut définir le comportement à avoir lorsqu'un thread souhaite lire depuis la file lorsque celle-ci est vide et lorsqu'un thread souhaite écrire dans la file mais que celle-ci est pleine.

Mutex

Un mutex (mutual exclusion, exclusion mutuelle) est une primitive de synchronisation qui permet d'éviter que des ressources partagées d'un système ne soient utilisées en même temps. Un mutex a deux états : verrouillé ou non verrouillé. La plupart des mutex ou des sémaphores ont des effets secondaires qui peuvent bloquer l'exécution, créer des goulets d'étranglement, voire ne pas remplir leur rôle en permettant tout de même l'accès aux données protégées. Un autre effet est le blocage total des ressources, si le programme qui les utilisait n'a pas informé le système qu'il n'en avait plus besoin.

Variable-condition

Une variable-condition est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition soit vérifiée. Deux opérations sont disponibles : wait, qui bloque le processus léger tant que la condition est fausse et signal (ou wake) qui prévient les processus bloqués que la condition est vraie. Il existe également une attente limitée permettant de signaler automatiquement la variable condition si le réveil ne s'effectue pas pendant une durée déterminée.
Une variable condition doit toujours être associée à un mutex, pour éviter les accès concurrents.

L'interblocage (deadlock)

L'interblocage (deadlock) se produit, par exemple, lorsqu'un thread T1 ayant déjà acquis la ressource R1 demande l'accès à une ressource R2, pendant que le thread T2, ayant déjà acquis la ressource R2, demande l'accès à la ressource R1. Chacun des deux threads attend alors la libération de la ressource possédée par l'autre. La situation est donc bloquée.

Chien de garde (watchdog)

Un chien de garde (watchdog), est une technique logicielle utilisée pour s'assurer qu'un programme ne reste pas bloqué à une étape particulière du traitement qu'il effectue. C'est une protection destinée généralement à redémarrer le système, si une action définie n'est pas exécutée dans un délai imparti.

Il s'agit en général d'un compteur qui est régulièrement remis à zéro. Si le compteur dépasse une valeur donnée (timeout) alors on procède à un reset (redémarrage) du système. Si une routine entre dans une boucle infinie, le compteur du chien de garde ne sera plus remis à zéro et un reset est ordonné.

Section critique

Une section critique est une portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'un thread simultanément. Il est nécessaire d'utiliser des sections critiques lorsqu'il y a accès à des ressources partagées par plusieurs thread. Une section critique peut être protégée par un mutex, un sémaphore ou d'autres primitives de programmation concurrente.

Exemple

Sous Qt, on peut mettre en oeuvre les mécanismes suivants :

- des mutex **QMutex** qui dispose de deux méthodes, lock (qui empêche les autres threads d'exécuter la section) et unlock (qui retire le blocage) et des sémaphores **QSemaphore** qui dispose de deux méthodes, acquire (qui empêche les autres threads d'exécuter la section) et release (qui retire le blocage)

- un synchronisateur à écriture exclusive et à lecture multiple **QReadWriteLock** associée à une zone mémoire à accès protégé. Cet objet agit comme une section critique qui permet à plusieurs threads de lire la mémoire qu'il protège à condition qu'aucun thread n'y écrive. Les threads doivent disposer d'un accès exclusif en écriture à la mémoire protégée. Chaque thread qui lit cette mémoire doit au préalable appeler la méthode lockForRead. BeginRead évite qu'un autre thread n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini de lire la mémoire protégée, il appelle la méthode unlock. Tout thread qui écrit dans la mémoire protégée doit au préalable appeler lockForWrite qui évite qu'un autre thread ne lise ou n'écrive simultanément dans la mémoire. Lorsqu'un thread a fini d'écrire dans la mémoire protégée, il appelle la méthode unlock, de sorte que les threads en attente puissent commencer à lire la mémoire.

- des variables conditions **QWaitCondition** qui agissent comme des signaux visibles pour tous les threads. Quand un thread termine une opération dont dépendent d'autres threads, il le signale en appellant wakeAll. wakeAll active le signal afin que les autres threads en attente (wait) soit débloqués.