Démo IFT2245

Synchronisation

Pourquoi?

Nous avons vu les dernières semaines comment un OS peut permettre plusieurs processus et threads qui opèrent de manière concurrente ou même parallèle.

Très cool comme idée, mais on introduit plein de problème.

- Ressources limitées en nombre (ex: un seul disque dur, ne peut lire qu'un morceau de fichier à la fois)
- Ressources partagées temporairement inconsistantes (ex: mémoire partagée entre threads)
- On ne peut plus être certain de l'ordre d'éxecution quand on partage le travail

Solutions

On a accès à plusieurs primitives de synchronisation qu'on peut utiliser pour attaquer ces problèmes :

- Mutex/Lock
- Sémaphore
- Moniteurs
- Opérations atomiques

Très difficiles à implémenter soi-même, absolument utiliser les primitives de l'OS. Si c'est vraiment nécéssaire, le spinlock est le plus simple à implémenter.

Mutex

Lock simple. Nom vient du principe d’exclusion mutuelle que le mutex implémente.

- Le premier processus à utiliser la ressource dans sa section critique s’approprie le mutex
- Quand il n’a plus besoin de la ressource, le même processus libère le mutex
- Si le mutex n’est pas disponible, le processus est bloqué. Quand le mutex est libéré, si un ou des processus attendent, au moins un est réveillé (et ré-essate d’acquérir le mutex).

Sémaphore

- Tout simplement une variable qu'on utilise pour faire notre propre synchronisation.
- Plus simple, sémaphore binaire, est 1 ou 0. Presque équivalent au mutex avec 1 - libre, 0 - occupé, mais plus flexible.
- Surtout, les sémaphores sont des chiffres généraux, pour des ressources dont on a plus d'une. Par exemple, si on a plusieurs copies d'un livre dans un biblio, on pourrait avoir une sémaphore qui commence le nombre total. Emprunt : on décrémente, retour : on incrémente. Si c'est 0, l'emprunt doit attendre un retour.

Mais comment attendre?

Comment faire attendre un processus pour qu'un lock ou sémaphore change?

- Le plus simple c'est un spinlock - une boucle sans fin qui vérifie constamment le statut de la variable d'intérêt. Attention, il faut immédiatement verouiller si c'est libre, de manière atomique
- Plus efficace c'est d'avoir une sorte de queue où ceux qui attendent sont réveillés - c'est une des meilleures méthodes d'éviter la famine avec une bonne queue!
- On peut aussi réveiller tous ceux qui attendent et faire une seule itération de spinlock

Atomique?

Une opération atomique est une opération indivisible. Donc, toute l'opération agit sur la mémoire d'un coup, sans qu'une autre opération puisse changer la mémoire qui nous intéresse au milieu.

- Test and set
- Compare and swap
- Opérations arithmétiques plus générales (mais surtout addition)

C'est ultimement une propriété du matériel. Presque tout le matériel en supporte au moins une, ce qui rend l'implémentation des primitive beaucoup plus facile - mais c'est possible même sans! (Voir algorithme de la boulangerie)

Condition Variable

La variable de condition est une primitive très puissante, souvent utilisée pour le moniteur (quand on ajoute un mutex).

- Permet de signaler qu'une condition est devenue vraie
- Consiste d'une liste/queue de processus en attente de la condition
- Quand un processus décide d'attendre, il sort de la section critique, et va généralement aussi relâcher un mutex - on dit qu'il y alors un mutex associé à la condition
- Un processus peut notifier un (par queue) ou tous les processus que la condition est devenue vraie
- Les processus vont ensuite revérifier la condition quand réveillés, et si oui réacquérir le mutex associé, et sinon revenir à l'attente.

Moniteur

C’est un “design pattern” avec un mutex et au moins une variable de condition. On l’utilise pour signaler et protéger l’état d’un objet ou ressource. Il faut donc l’accès au mutex et possiblement une condition supplémentaire pour pouvoir avoir accès à l’objet: le moniteur protège donc l’objet ou la ressource.

- Exemple : un objet et ses méthodes dans un programme concurrent

Problème des barbiers

- Un barbier paresseux s’endort dès qu’il n’y a pas de clients
- Si un client vient alors que le barbier est endormi, il va le réveiller
- Il y a un nombre limité de places dans la salle d’attente, qui est séparée, donc un client quittera pour revenir plus tard
- Le barbier vérifie la salle d’attente avant de dormi.

Qu’est-ce qui pourrait mal tourner? Comment régler ça?

Problème de verrous

- Quel problème pourrait-il y avoir quand plusieurs tâches ont besoin de plusieurs ressources exclusives qu'on verrouille ? Comment régler ce problème ?
- Quelle étape dans le sémaphore et le mutex poserait-elle un problème en cas de crash ? Est-ce qu'il y aurait une solution ?
- Qu'en est-il de la programmation fonctionnelle ? Quels problèmes est-ce que l'on pourrait avoir si, par exemple, on exécute une fonction-paramètre dans la section critique ?

Exercices

- Un programme est parallélisé en affectant un thread au réseau, qui écoute pour recevoir des images dans un tampon circulaire. Ensuite, plusieurs threads opèrent sur ces images, afin de le convertir en différents formats pour les écrire dans un autre tampon circulaire. Finalement, un dernier thread lit ce tampon et écrit ces images au disque. Comment feriez-vous la synchronisation ? Quels problèmes essayez-vous d'éviter ?
- Vous écrivez un jeu vidéo, et la génération de l'image se fait en ajoutant plusieurs parties de l'image générées par différents threads qui gèrent un GPU. Ce dernier est parallèle au niveau du pixel, mais chaque thread partage la même image. Comment synchroniseriez-vous le dessin, en sachant que chaque partie de l'image à une priorité différente ? Quels avantages et quels problèmes ont votre approche ?

TP1!