Programmation système

Accès concurrents, blocages

loig.jezequel@univ-nantes.fr

Difficultés de la programmation concurrente

On commence par un exemple : race.go

Accès concurrents à une ressource

Ce qu'il s'est passé

- Toutes les goroutines modifiaient les valeurs des mêmes variables en même temps (de manière concurrente)
- On ne peut pas faire d'hypothèse sur l'ordre d'exécution des instructions dans un contexte de concurrence
- L'échange de valeurs de variables n'est pas atomique

Exercice

Indiquer comment doit sans doute être réalisée l'échange des valeurs de deux variables au niveau du code compilé. En déduire les différentes sorties possibles dans le cas où on a deux goroutines dont chacune échange les valeurs des deux mêmes variables (on pourra représenter pour cela les entrelacements possibles des opérations réalisées par ces goroutines).

Accès concurrents à une ressource (suite)

Race condition

On parle d'accès concurrents (ou de race condition) à une ressource lorsque plusieurs accès à une variable (dont au moins une écriture) ont lieu de manière concurrente.

Placer des temps d'attente ne résout pas les problèmes

race2.go

Exercice

Proposer des idées pour éviter le problème d'accès concurrents à une ressource dans l'exemple précédent (race.go)

Exclusion mutuelle

Section critique

On appelle section critique l'endroit du code où on accède à une ressource partagée

Exclusion mutuelle

Assurer l'exclusion mutuelle c'est assurer qu'un seul fil d'exécution à la fois accède à une section critique. Dans ce cas, il n'y a pas d'accès concurrents à la ressource partagée.

Une première solution

race3.go

Exclusion mutuelle (suite)

Exercice

Pourquoi la première solution proposée pour réserver l'accès à la section critique (fichier race3.go) n'est-elle pas satisfaisante ?

Exclusion mutuelle (suite)

Exercice

Pourquoi la première solution proposée pour réserver l'accès à la section critique (fichier race3.go) n'est-elle pas satisfaisante ?

Opérations atomiques

Si on est capable de tester une variable et de changer sa valeur en une seule opération (qu'on appelle atomique) alors on peut résoudre le problème de l'exclusion mutuelle.

Implantation

race4.go

Famine

Exercice

Si chaque fil d'exécution veut accéder plusieurs fois à la section critique, voyez-vous un problème qui peut arriver avec l'implantation qu'on vient de voir ? (fichier race4.go)

Famine

Exercice

Si chaque fil d'exécution veut accéder plusieurs fois à la section critique, voyez-vous un problème qui peut arriver avec l'implantation qu'on vient de voir ? (fichier race4.go)

Famine (starvation)

Un fil d'exécution peut demander l'accès à une ressource (section critique) et ne jamais l'obtenir (il suffit qu'il y ait toujours un autre fil qui la demande en même temps).

Solution possible dans notre cas

race5.go

Semaphores

Exercice

En plus de sa lenteur, voyez-vous un problème avec l'implantation précédente ? (fichier race5.go) Par exemple, si les goroutines ne démarrent pas dans l'ordre de leurs ID, que se passe-t-il ?

Semaphores

Exercice

En plus de sa lenteur, voyez-vous un problème avec l'implantation précédente ? (fichier race5.go) Par exemple, si les goroutines ne démarrent pas dans l'ordre de leurs ID, que se passe-t-il ?

Semaphore

- Une variable K initialisée à 1
- Une file de processus (FIFO)
- Deux opérations :
  - P (réservation)
    - $K = K - 1$
    - Si $K < 0$ le fil d'exécution appelant est mis en file d'attente
    - Si $K \geq 0$ le fil d'exécution appelant continue son exécution
  - V (libération)
    - $K = K + 1$
    - si $K \leq 0$ le fil d'exécution en tête de file d'attente continue

Semaphores (suite)

Remarques : gestions de quantités de ressources disponibles

- K peut être initialisé à plus que 1
- P et V peuvent faire varier K de plus de 1

En Go

On dispose dans le paquet sync d'un type Mutex qui correspond (à peu de choses près) à un semaphore (pour des raisons pratiques, la gestion de la file d'attente est plus complexe).

Implantation finale

race6.go

Attention !

Utiliser de telles synchronisations est source d'erreurs

- Réutiliser une ressource après l'avoir libérée (voir err.go)
- Libérer une ressource sans l'avoir réservée (voir err1.go)
- Utiliser une ressource sans l'avoir réservée (voir err2.go)
- Garder une ressource trop longtemps (voir err3.go)
- Oublier de libérer une ressource (voir err4.go)

Et plus on a de ressources partagées plus c'est compliqué !

Deadlocks

Deadlock (interblocage)

On parle de deadlock lorsque tous les fils d'exécution sont bloqués en même temps : le programme ne peut plus progresser et n'a aucun espoir de se débloquer.

Quand ?

En général il y a des risques de deadlocks quand plusieurs mécanismes de synchronisation sont mis en place en même temps :

- plusieurs ressources partagées, chacune protégée par un verrou différent
- une ressource partagée et une lecture dans un canal
- une ressource partagée et une attente d'évènement
- etc

???

On regarde le fichier map.go, quel est le problème avec ce code ?