24

Primitives de synchronisation

Les opérations atomiques constituent la base sur laquelle repose tout l'édifice de la synchronisation dans les systèmes modernes. Avant d'aborder les mutex, les channels et autres primitives de haut niveau en Go, comprendre ces opérations élémentaires permet de saisir les mécanismes fondamentaux qui garantissent la cohérence des données en environnement concurrent.

Dans les architectures à mémoire partagée modernes, les opérations atomiques représentent la seule méthode pratique pour synchroniser des processus concurrents. Bien qu'il existe des alternatives théoriques comme le passage de messages entre processeurs ayant chacun leur mémoire locale, ou des algorithmes purement logiciels fonctionnant sans support matériel atomique, ces approches introduisent une complexité architecturale et des pénalités de performance prohibitives. La mémoire partagée étant l'élément central permettant la communication entre processeurs, il est naturel que la synchronisation passe par elle via des instructions atomiques matérielles.

Ce chapitre explore les opérations atomiques du point de vue matériel, leur implémentation dans le package `sync/atomic` de Go et leurs implications en termes de performance et de garanties mémoire.

24.1. Fondements matériels des opérations atomiques

Le problème fondamental de la concurrence

Dans un système monoprocesseur sans préemption, les opérations mémoire se déroulent séquentiellement et de manière prévisible. Une simple instruction d'incrémentation `counter++` s'exécute sans risque d'interférence. Mais dès que le parallélisme est introduit (que ce soit par plusieurs cœurs CPU ou par la préemption de threads), cette instruction apparemment atomique se décompose en trois étapes distinctes :

1. lecture de la valeur depuis la mémoire dans un registre ;
2. incrémentation de la valeur dans le registre ;

3. écriture de la nouvelle valeur en mémoire.

Entre chacune de ces étapes, un autre processeur ou thread peut intervenir et modifier la même zone mémoire, créant une race condition. C'est précisément ce problème que les opérations atomiques résolvent en garantissant qu'une lecture ou écriture mémoire s'effectue de manière indivisible, sans qu'une autre goroutine puisse lire un état incohérent ou partiel.

Instructions atomiques matérielles

Comme nous l'avons introduit dans le modèle mémoire, les processeurs modernes fournissent des instructions spéciales qui garantissent l'exécution atomique d'opérations read-modify-write. Ces instructions constituent les primitives fondamentales sur lesquelles toute synchronisation repose. Approfondissons maintenant leur fonctionnement et leurs implications.

Architecture x86/x86-64

Sur x86, le préfixe `LOCK` transforme une instruction ordinaire en opération atomique. Nous avons vu que ce préfixe verrouille temporairement le bus mémoire (100-300 cycles), mais examinons plus en détail son mécanisme :

```
LOCK XADD [address], register  ; Atomique : *address += register
```

L'instruction `CMPXCHG` (compare and exchange) implémente compare-and-swap, la primitive atomique la plus polyvalente que nous utiliserons fréquemment dans ce chapitre :

```
LOCK CMPXCHG [address], new_value
; Si *address == EAX alors *address = new_value
; Sinon EAX = *address
```

Le préfixe `LOCK` assure que pendant l'exécution de l'instruction :

- le bus mémoire est verrouillé ou la ligne de cache est acquise en exclusivité ;
- aucun autre processeur ne peut accéder à cette zone mémoire ;
- les barrières mémoire nécessaires sont automatiquement insérées.

Architecture ARM

Sur ARM, nous avons évoqué l'approche Load-Exclusive/Store-Exclusive. Détaillons maintenant ce mécanisme qui diffère fondamentalement de x86 :

```
; Ancien modèle LL/SC (Load-Link/Store-Conditional)
; Exemple : addition atomique
retry_add:
    LDREX   r1, [r0]        ; Load exclusive (marque la ligne)
    ADD     r1, r1, #1      ; Incrémentation
    STREX   r2, r1, [r0]    ; Store conditionnel
    CMP     r2, #0          ; Vérifier le succès
    BNE     retry_add       ; Réessayer si échec

; Exemple : compare-and-swap
retry_cas:
    LDREX   r1, [r0]        ; Load exclusive
    CMP     r1, r2          ; Comparer avec valeur attendue
    BNE     failed          ; Si différent, échec
    STREX   r3, r4, [r0]    ; Store conditionnel nouvelle valeur
    CMP     r3, #0          ; Vérifier le succès du store
    BNE     retry_cas       ; Réessayer si échec
failed:
```

Le modèle LL/SC (Load-Link/Store-Conditional) permet au processeur de détecter si la zone mémoire a été modifiée entre le load et le store. Si c'est le cas, le store échoue et l'opération doit être réessayée. Ce mécanisme est flexible car il permet d'implémenter n'importe quelle opération atomique, mais nécessite une boucle de réessai explicite et consomme 100-400 cycles en cas de contention.

```
; Nouveau modèle LSE (ARMv8.1+)
; Instructions atomiques dédiées, plus efficaces
    LDADD   r1, r2, [r0]    ; Addition atomique
    CAS     r1, r2, [r0]    ; Compare-and-swap atomique
    SWP     r1, r2, [r0]    ; Swap atomique
```

Le nouveau modèle LSE introduit des instructions atomiques dédiées qui éliminent le besoin de boucles de réessai et offrent de meilleures performances (50-200 cycles en cas de contention).

Cohérence de cache et protocoles MESI

Les opérations atomiques ne se contentent pas de garantir l'atomicité d'une instruction : elles doivent aussi assurer la cohérence des données à travers tous les caches CPU. Les processeurs modernes utilisent des protocoles de cohérence de cache comme MESI (Modified, Exclusive, Shared, Invalid) pour maintenir une vue cohérente de la mémoire.

Quand un processeur exécute une opération atomique :

1. Il doit acquérir la ligne de cache en état Exclusive ou Modified.

2. Les autres processeurs voient leurs copies invalidées (état Invalid).
3. L'opération s'exécute.
4. Les modifications deviennent visibles aux autres processeurs selon les garanties du modèle mémoire.

Ce processus génère un trafic important sur le bus mémoire, expliquant le coût non négligeable des opérations atomiques même en l'absence de contention.

Memory barriers et ordering

Les processeurs modernes réordonnent les instructions pour optimiser les performances. Les opérations atomiques doivent donc inclure des barrières mémoire (memory barriers ou fences) pour garantir que :

- les écritures précédentes sont visibles avant l'opération atomique ;
- les lectures suivantes voient les effets de l'opération atomique ;
- l'ordre apparent des opérations respecte les garanties du modèle mémoire.

Sur x86, les instructions atomiques incluent implicitement des barrières complètes. Sur ARM, les variantes avec acquire/release semantics (LDADD, LDAR, STLR) fournissent des garanties d'ordering spécifiques.

Coût des opérations atomiques

Le coût d'une opération atomique dépasse largement celui d'un accès mémoire normal :

- Accès mémoire ordinaire : quelques cycles CPU si en cache L1.
- Opération atomique sans contention : 20-50 cycles (due aux barrières mémoire et synchronisation de cache).
- Opération atomique avec contention : 100-1000+ cycles (attente de la libération de la ligne de cache, réessais potentiels).

Pour une démonstration concrète de ces coûts, reportez-vous à l'annexe Section 2, Impact des mécanismes de synchronisation sur les performances qui compare les performances d'une boucle simple, d'opérations atomiques et de mutex.

24.2. Package sync/atomic : interface Go

Go expose les opérations atomiques via le package `sync/atomic`, qui fournit une interface de haut niveau sur les instructions matérielles sous-jacentes. Le compilateur Go traduit ces fonctions en instructions atomiques natives de la plateforme cible.