Programmation Système V
Cohérence de la mémoire

Juliusz Chroboczek

28 février 2016

Entre le disque et le processeur, les données apparaissent à plusieurs endroits : sur le disque, dans le cache du système de fichiers, dans les tampons ou les zones de mémoire mappées, dans les caches du processeur. Dès qu'une donnée se trouve à plusieurs endroits se posent les problèmes de *cohérence* ou *consistence* de la mémoire : les différentes copies ne sont pas forcément identiques.

1 Latence de la mémoire et notion de cache

Les différentes mémoires qui se trouvent dans un ordinateur ont des latences très différentes. À un extrême, un disque à rouille tournante a une latence moyenne d'une demi-révolution en moyenne ; à 6000 tours par minute, c'est 5 ms. La mémoire principale (RAM) a une latence intermédiaire, de l'ordre de 50 à 100 ns. Les registres du processeur ont une latence d'une fraction de nanoseconde. Pour acquérir une intuition de ces valeurs, il suffit de multiplier les temps par la vitesse de la lumière : les données sur disque sont très loin de nous, à 1500 km ; les données en mémoire principale sont à 15 à 30 m ; et les registres sont à quelques centimètres (voir figure 1).

Type de mémoire	latence typique	latence × c
Disque dur	5 ms	1500 km
SSD (lecture)	100 ns	30 m
RAM	100 ns	30 m
Cache L2	20 ns	6 m
Cache L1	2 ns	60 cm
Registre	0.2 ns	6 cm

Figure 1 — Latence typique des différents types de mémoire

Pour cacher cette latence, les systèmes implémentent des *caches*[1], c'est à dire des copies d'une partie des données qui se trouvent plus près du processeur. Il existe principalement deux types de caches : le cache du système de fichiers, implémenté par le système d'exploitation, et les caches du processeur, implémentés par le matériel.

1. Prononcé [kæʃ] en anglais, comme le mot *cash*.

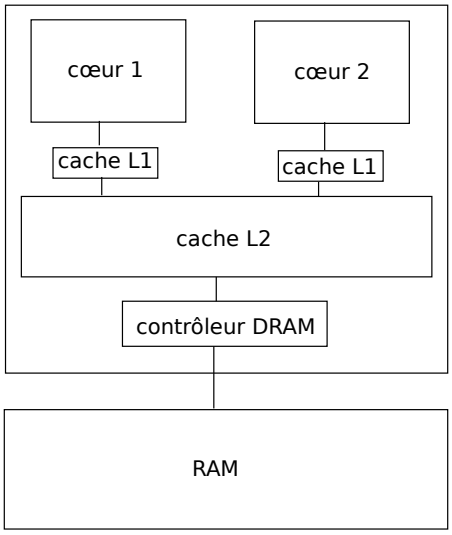

Figure 2 — Le processeur et ses caches

1.1 Le *buffer cache*

Le *buffer cache* ou *page cache* est une structure de donnée maintenue en mémoire principale par le système d'exploitation. Lorsque le système a besoin de faire une lecture depuis le système de fichiers (par exemple parce qu'un processus a fait un appel à `read`, ou parce qu'il faut satisfaire un défaut de page), il consulte le cache : si les données ne s'y trouvent pas, il les recopie depuis le disque dans le cache ; si par contre elles s'y trouvent déjà (parce qu'elles ont été lues récemment), le système s'en sert sans consulter le périphérique de stockage de masse. Si le périphérique de stockage de masse n'est pas occuppé, le système peut parfois anticiper une lecture, et copier les données vers le cache avant que le processus utilisateur les demande : c'est le *prefetching*.

Il en est de même pour les écritures : un appel à `write` recopie les données dans le cache, les données seront écrites sur le périphérique de stockage de masse plus tard, lorsque ce dernier ne sera pas occuppé. Le fait de retarder les écritures permet aussi de les agréger (faire plusieurs écritures en une seule opération) et parfois même de les éviter entièrement (si les données sont écrasées par une lecture subséquente).

Le *buffer cache* a une taille variable, mais bornée[2], il est donc nécessaire de libérer des pages du cache lorsque sa taille croît trop. L'algorithme qui décide quelles donner éliminer du cache s'appelle l'algorithme d'*éviction*, et il dépend du système.

1.2 Les caches du processeur

Les caches du processeur se trouvent entre le processeur et la mémoire principale. Il y a typiquement deux niveaux de cache : un cache de niveau 1 (*level 1*, L1), spécifique à chaque cœur, et un cache de niveau 2 (L2), partagé entre plusieurs cœurs (figure 2). Les processeurs pour serveurs

2. Sous Linux, la commande `free` affiche sa taille.

ont parfois aussi un cache de niveau 3 (L3), les processeurs embarqués n'ont souvent qu'un cache de niveau 1[3].

Lorsque le processeur lit une donnée, il consulte le cache de niveau 1 ; si elle ne s'y trouve pas, il consulte le cache de niveau 2 ; enfin, si elle ne s'y trouve pas, il consulte la mémoire principale. Une fois trouvée, la donnée est recopiée dans la hiérarchie de caches (depuis la mémoire principale vers le cache L2, et depuis le cache L2 vers le cache L1).

Le processeur optimise ces opérations de plusieurs façons. Tout d'abord, lors d'une lecture il lit un bloc entier de données, une *ligne de cache* (64 octets sur Intel), et pas seulement la donnée demandée. De plus, lorsqu'il détecte des lectures séquentielles, il anticipe les lectures en préchargeant *(prefetch)* les données dans le cache avant que le programme en ait besoin. Ces optimisations font que les lectures séquentielles sont souvent 100 fois plus rapides que les lectures aléatoires[4].

Le processeur optimise aussi les écritures. Lorsque le programme écrit dans la mémoire (par exemple en affectant une valeur à une variable globale), la ligne de cache entière est lue dans le cache L1 et modifiée par le processeur ; l'écriture de la ligne de cache *(writeback)* se fera plus tard, ou peut-être jamais.

2 Cohérence

En présence des caches, les données sont dupliquées à plusieurs endroits. Pour écrire des programmes fiables, il est essentiel de prendre en compte la cohérence (ou consistance) de ces différentes copies.

2.1 Cohérence du système de fichiers

Après un `write`, les données se trouvent dans le *buffer cache* ; elles seront écrites sur le disque plus tard, traditionnellement dans la minute qui suit sous Unix[5]. Normalement, ce délai n'est pas détectable par un programme : un appel à `read` retourne les données du cache, ce qui fait que les anciennes données se trouvant sur le disque *(stale)* ne sont pas visibles par le programme utilisateur.

Le délai est visible dans un cas : si le système se plante avant d'avoir pu les écrire. Imaginez un programme qui ouvre un fichier avec `O_TRUNC` puis écrit une nouvelle version des données ; si le système choisit d'effectuer la troncation immédiatement, mais de retarder l'écriture des données, un plantage du système peut mener à une situation où ni l'ancienne ni la nouvelle version des données ne sont disponibles.

L'appel système `fsync` permet de gérer explicitement la cohérence du *buffer cache*. Lorsque `fsync` est exécuté sur un descripteur de fichiers, les données associées à ce descripteur sont stockées de façon persistante avant que `fsync` retourne[6]. Attention, `fsync` est une opération coû-

3. Sous Linux, la commande `cat /proc/cpuinfo` permet de déterminer la taille des caches du processeur.
4. Et maintenant vous savez pourquoi les B-trees sont préférés aux AVL. Le programmeur maniaque choisit l'arité de son B-tree pour qu'un nœud ait exactement la taille d'une ligne de cache.
5. Mais les Unix optimisés pour les ordinateurs portables ont des délais plus longs, jusqu'à 10 minutes sous Linux avec l'option *laptop mode*.
6. `fsync` est boggué sur Mac OS X (regardez la page de man, c'est documenté). Et sûrement sur d'autres systèmes aussi.

teuse, surtout sur les systèmes de fichiers à journal, il est donc important de ne pas en abuser — ne soyez pas *Mozilla*.

Les deux séquences de la figure 3 sont traditionnellement utilisées pour sauvegarder les données ; sur un système de fichiers pas trop boggué, elles devraient garantir qu'au moins une des deux versions d'un fichier se trouve sur le disque quel que soit le moment où le système se plante.

```
fd = open("toto.tmp",
          O_WRONLY | O_EXCL,
          0666);
if(fd < 0)
    ...
rc = write(fd, ...);
if(rc < 0)
    ...
rc = fsync(fd);
if(rc < 0)
    ...
close(fd);
rc = rename("toto.tmp", "toto");
```

```
rc = rename("toto", "toto~");
if(rc < 0)
    ...
fd = open("toto",
          O_WRONLY | O_EXCL,
          0666);
if(fd < 0)
    ...
rc = write(fd, ...);
if(rc < 0)
    ...
rc = fsync(fd);
if(fd < 0)
    ...
close(fd);
rc = unlink("toto~");
```

Figure 3 — Séquences de code pour une sauvegarde fiable

2.2 Cohérence de `mmap`

Lorsque deux processus ont fait un `mmap` partagé sur le même fichier (soit parce que chacun a fait `mmap`, soit parce qu'un `mmap` a été fait avant un `fork`), trois copies des données sont visibles : chacun des deux *mappings*, et la version du système de fichiers, accessible par `read` et `write`. POSIX ne fait à ma connaissance aucune garantie sur la cohérence de ces trois versions.

Gestion explicite de la cohérence Pour être portable, la cohérence doit donc être gérée explicitement par le programmeur. Cela peut se faire à l'aide de l'appel système `msync` :

```
int msync(void *addr, size_t length, int flags);
```

Si le paramètre `flags` vaut `MS_SYNC`, les données de la plage d'adresses sont copiées dans le système de fichiers (le *buffer cache*), ce qui les rend visibles à `read`. Si le paramètre `flags` vaut `MS_SYNC | MS_INVALIDATE`, alors les données sont rendues visibles non seulement au système de fichiers mais aussi aux autres processus qui ont mappé cette plage de données.

Cohérence sur un système raisonnable Si POSIX fait peu de garanties, la plupart des concepteurs de systèmes de mémoire virtuelle sont raisonnables, et n'introduisent pas d'incohérences pour le plaisir. De ce fait, sur la plupart des systèmes les copies en mémoire ont les mêmes propriétés de cohérence que le matériel, et la communication par mémoire partagée est donc possible sans `MS_INVALIDATE` à condition de comprendre les propriétés de cohérence du matériel (voir ci-dessous).

Attention cependant, ceci ne s'applique qu'aux copies visibles en mémoire virtuelle — la copie visible au système de fichiers (`read`) n'est pas cohérente sans utiliser des `msync` explicites.

Il convient de mentionner que tous les systèmes ne sont pas raisonnables. HP-UX, par exemple, a un système de mémoire virtuelle non cohérent, et introduit donc des incohérences gratuites. La meilleure solution à ce problème est de refuser de programmer pour de tels systèmes, ou tout au moins d'éviter d'y utiliser de la mémoire partagée.

2.3 Cohérence matérielle

Comme nous l'avons vu ci-dessus, le matériel introduit des copies supplémentaires des données en mémoire principale du fait de ses caches. Si une zone de mémoire est *mappée* par plusieurs processus, le système peut choisir de les exécuter sur des cœurs ou des processeurs différents, ce qui fera qu'ils verront des caches différents (voir figure 2). Les propriétés de cohérence garanties par le matériel dépendent de l'architecture, et elles sont assez faibles sur plusieurs architectures assez répandues (notamment ARM, utilisé dans les téléphones portables, et POWER/PowerPC), ce qui fait qu'on ne peut pas compter sur une mémoire physique cohérente entre cœurs. En particulier, une implémentation naïve des algorithmes d'exclusion mutuelle de Dekker ou Peterson est incorrecte sur ARM.

La solution consiste à introduire des *memory fences*, des instructions qui imposent un ordre globalement visible aux opérations sur la mémoire. Nous en parlerons plus en détail dans la partie sur les *threads*, pour le moment il est juste nécessaire de savoir que chaque appel système manipulant des primitives de synchronisation (sémaphores POSIX, *mutex pthreads*, etc.) implique une barrière — il n'y a donc pas de problèmes de cohérence dûs au matériel dès lors que toutes les valeurs partagées sont correctement partagées par des sémaphores ou des *mutex*.

3 Sémaphores POSIX

Lorsque plusieurs processus ou *threads* manipulent une zone de mémoire partagée, il est nécessaire de synchroniser les accès qu'ils y font. La bibliothèque *pthreads* contient plusieurs primitives puissantes mais compliquées à utiliser ; POSIX définit donc une primitive facile à utiliser et qui a un style semblable à `mmap` : le *sémaphore POSIX*.

Un sémaphore POSIX est stocké dans une variable de type `sem_t`. Tout comme une zone de mémoire partagée, un sémaphore peut être anonyme (et alors il est partagé après un `fork`) ou nommé. Un sémaphore anonyme est créé à l'aide de `sem_init`, et il faut le placer dans une zone de mémoire partagée. Un sémaphore nommé est créé à l'aide de `sem_open`, et il est détruit à l'aide de `sem_close`. Dans les deux cas, le sémaphore est incrémenté à l'aide de `sem_post` et décrémenté à l'aide de `sem_wait`.