# QUESTIONS sur le TP8 / Contrôleur de disque et partage des périphériques

## Q1) Que se passe-t-il si deux tâches qui s'exécutent en pseudo-parallélisme sur le même coeur (par multiplexage temporel) font toutes les deux un appel système ioc_read() ?

La concurrence d'accès lié au pseudo-parallélisme pose les mêmes problèmes que la situation où deux tâches A et B s'exécutent en vrai parallélisme, sur deux coeurs différents.

Chacune des tâches A et B a défini son propre tampon mémoire destination dans son propre espace d'adressage. La seule ressource partagée est donc le contrôleur IOC qui déplace les données du disque vers le tampon mémoire. La première tâche (A) qui réussit à prendre le verrou continue son exécution et peut donc lancer le transfert requis entre le disque et la mémoire. La deuxième tâche (B) est bloquée dans la fonction de prise de verrou. Dans le cas du GIET ce blocage est réalisé par une boucle d'attente active ou la fonction de prise de verrou teste la valeur du verrou, jusqu'à obtenir une valeur nulle signalant que le contrôleur IOC est redevenu libre. Heureusement les appels systèmes sont interruptibles, et les autres applications peuvent continuer à s'exécuter dans les tranches de temps qui leur sont allouées. Mais la tâche B utilise toutes ses tranches de temps à faire de la scrutation sur l'adresse du verrou, ce qui est clairement du gâchis. les systèmes d'exploitation généralistes ont une politique plus intelligente consistant à mettre en sommeil la (ou les) tâche(s) bloquée(s) jusqu'à libération du verrou, ce qui évite de consommer inutilement des tranche de temps.

## Q2) Pourquoi le verrou protégeant l'accès exclusif au contrôleur IOC est-t-il pris par les appels système ioc_read() et ioc_write() plutôt que par l'application elle-même ?

La raison la plus importante est que l'OS ne peut pas et ne veut confier à une application particulière la responsabilité de verrouiller et donc de dé-verrouiller une ressource partagée par un grand nombre d'applications. Ce serait en effet prendre le risque que le verrou ne soit pas relâché pr une application mal programmée ou malicieuse.

Par ailleurs l'OS cherche à masquer au code applicatif la complexité de l'accès au disque, et en particulier les mécanismes de partage.

## Q3) Pourquoi ne trouve-t-on pas dans LINUX l'appel système ioc_completed() du GIET ?

Comme indiquéé ci-dessus, les appels systèmes visent à fournir aux applications l'abstraction la plus simple possible pour faciliter l'écriture du code applicatif. Dans LINUX, il n'existe donc que deux appels système read() et write() qui sont bloquant tous les deux et ne rendent la main que lorsque que le transfert est terminé et que le tampon mémoire peut être utilisé. Le blocage de l'application demandeuse n'est pas pénalisant pour le fonctionnement général de la machine, puisque LINUX utilise une politique de mise en sommeil de l'application demandeuse, plutôt qu'une politique de scrutation qui gâche des cycles CPU.

La raison pour laquelle cet appel système a été introduit dans le GIET est que cela permet de mettre en lumière qu'une opération d'entrée/sortie est toujours une valse à trois temps : (i) lancement de l'opération par un appel système, (ii) transfert des données, (iii) signalisation de fin de transfert par interruption, et que la signalisation de fin de transfert n'est pas moins importante que le démarrage. Par ailleurs, cet appel système est nécessaire pour optimiser le code et construire un pipe-line logiciel qui exploite le fait que plusieurs maîtres travaillent en parallèle.

## Q4) Pourquoi le verrou garantissant l'accès exclusif au périphérique IOC (contrôleur de disque) est-il libéré par l'appel système ioc_completed() plutôt que par l'ISR associée à l'interruption signalant la fin du transfert ?

Je pense que c'est tout simplement une erreur de programmation dans le GIET. Cela n'a pas de conséquences négatives dans les cas des utilisations du composant IOC vues en TP, mais c'est clairement un bug inacceptable pour un véritable OS : supposons qu'une application commette une erreur d'adressage après avoir exécuté l'appels système ioc_read() (non bloquant) mais avant d'exécuter l'appel système ioc_completed(). Cette erreur d'adressage déclenche une exception fatale, et l'application va être tuée par l'OS. L'appel système ioc_completed() ne sera donc pas exécuté, et le verrou ne sera pas relâché, ce qui rend le contrôleur de disque inutilisable pour toutes les autres applications, ce qui est un peu ennuyeux. C'est donc en effet l'ISR qui doit relâcher le verrou quand le transfert est terminé.

## Q5) Pourquoi ne pas définir des primitives permettant lire et d'écrire sur le disque octet par octet plutôt que bloc par bloc ?

Lorsqu'un programme ne modifie que quelques octets dans un block de 512 octets, il semble déjà peu efficace de copier le bloc complet du disque vers la mémoire, puis de modifier quelques octets en mémoire, avant de recopier ce block modifié sur le disque. Pourtant la réalité est encore plus compliquée...

En pratique, tous les OS généralistes utilisent un *cache de fichiers* logiciel pour accélérer l'accès aux informations stockées sur le disque. Plus précisément, l'OS crée dynamiquement, dans son espace mémoire protégé, un cache indépendant pour chaque fichier ouvert par au moins une application. Ce fichier est vu par l'OS comme une séquence de *pages* de 4 Koctets. Chaque cache de fichier est le plus souvent organisé comme un arbre (radix-tree) dont les feuilles sont des tampons mémoire de 4 Koctets. Chaque tampon contient une des pages du fichier (4 Koctets = 8 blocs de 512 octets). C'est un arbre incomplet, car seules les pages qui on été accédées en lecture ou en écriture sont présentes dans le cache. A la différence d'un cache matériel de processeur dont la capacité est fixe, ce cache logiciel est extensible: l'OS alloue dynamiquement une nouvelle page quand il faut ramener une nouvelle partie du fichier dans le cache. C'est un cache WRITE-BACK, puisque le fichier n'est mis à jour sur le disque que lorsqu'il est fermé par l'application. Les conséquences sont les suivantes :

- tous les accès read() ou write() effectués par le code applicatif se font en réalité dans le cache du fichier en mémoire, et ces accès se font avec une granularité *octet* (instructions assembleur lb ou sb).
- tous les mouvements de données entre le cache du fichier et le disque sont effectués par l'OS et se font avec une granularité *page*, soit cas de MISS sur le cache de fichier, sot lors de la fermeture du fichier.

## Q6) Pourquoi, à la fin d'un transfert de fichier vers une clé USB, est-il nécessaire de *cliquer* sur "retirez le périphérique en toute sécurité" ?

Une clé USB est un périphérique de stockage de blocs, qui se comporte donc comme comme un disque magnétique. Les temps d'accès sont mille fois plus plus courts que pour un disque, mais restent mille fois plus longs qu'un accès à la mémoire.

Puisqu'une clé USB se comporte comme un disque, le système d'exploitation utilise le même mécanisme de cache logiciel que décrit ci-dessus pour lire ou écrire sur la clé USB. Mais puisqu'il s'agit d'un cache WRITE-BACK, le contenu de la clé USB peut être obsolète par rapport au contenu du cache de fichier, ce qui pose un problème pour un périphérique amovible. Cliquer sur le bouton permet de synchroniser le contenu de la clé USB avec le contenu du cache de fichier.

*Last modified on 9 Jun 2020, 19:33:28*