Systèmes informatiques et applications concurrentes

Partie 4 - SGF - Systèmes de gestion de fichiers

Ivan KURZWEG

28 avril 2010

Systèmes informatiques et applications concurrentes
by Ivan KURZWEG

Copyright © 2010 Ivan KURZWEG, *ivan.kurzweg@gmail.com, 2010*

Permission to use, copy, modify, and distribute this documentation *for any purpose with or without fee* is here by granted, provided that *the above copyright notice and this permission notice appear in all copies*.

The documentation is provided "as is" and the author disclaims all warranties with regard to this documentation including all implied warranties of merchantability and fitness. In no event shall the author be liable for any special, direct, indirect, or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this documentation.

Table des matières

1 Notions de *filesystem* **1**
1.1 Supports de stockage 1
1.2 Définition d'un *filesystem* 2
1.3 Cohérence et intégrité 2
1.4 Montage et démontage d'un *filesystem* 3
1.4.1 "*Remontage*" après *crash* 3

2 Fichiers et répertoires **4**
2.1 inodes et répertoires 4
2.1.1 Inodes 4
2.1.2 Répertoires 5
2.2 Types de fichiers 5
2.2.1 Fichiers réguliers 5
2.2.2 Fichiers spéciaux 6
2.3 Droits 6
2.4 Droits étendus 7
2.4.1 Bit SUID 7
2.4.2 Le bit SGID 7
2.4.3 Le *sticky bit* 7

3 Virtual file system **7**
3.1 Présentation 7
3.2 vnodes 7

4 Exercices **8**
4.1 Commandes Unix 8
4.1.1 Exercice 1 8
4.2 *filesystem* 8
4.2.1 Liens 8
4.2.2 Graphe acyclique 8
4.2.3 Droits 9

5 Corrections **9**
5.1 Translation 9
5.2 Performances 9
5.2.1 Partie 1 9
5.2.2 Partie 2 10
5.3 Remplacement de page (Source : Ivan Boule, Examen SMB137 2004) 10
5.4 Chronogramme 11
5.5 Comparaison d'algorithmes 11

6 Références **12**
6.1 Références 12

Table des figures

1 Disque dur 1
2 Plateau d'un disque dur 2
3 Structure d'un inode (Source : The Design and Implementation of the 4.4BSD Operating System) 4
4 5
5 Droits d'accè à un fichier 6
6 Droits d'accès à un répertoire 6
7 Exercice 3 11

Résumé

Partie 4 du cours "Systèmes informatiques et applications concurrentes". Cette quatrième partie du cours aborde les notions de fichiers et systèmes de gestion de fichiers. On se concentrera sur l'implémentation de type *Unix*

1 Notions de *filesystem*

1.1 Supports de stockage

Les supports de stockage sur disque sont de différents natures :

- Disques durs
- Disquettes, supports de masse USB
- Disques optiques (CD, DVD, BD)
- ...

Concernant les disques durs, on définit généralement les propriétés suivantes :

- La plus petite unité adressable est un *bloc*
- La taille d'un bloc est généralement de 512 octets
- La zone de stockage d'un bloc sur le plateau d'un disque est le *secteur*
- Les disques d'anciennes générations utilisaient une adressage constitué de cylindres, pistes et secteurs.
- Les disques de nouvelles générations utilisent la technologie LBA (*Logical Block Adressing*)
- Il est désormais possible d'écrire et de lire plusieurs blocs contiguës en une seule commande, et ainsi de définir pour le SGF des blocs de taille multiple de la taille des secteurs.

Les disques durs sont composés de plateaux selon l'architecture suivante (source wikipedia) :

FIG. 1 Disque dur

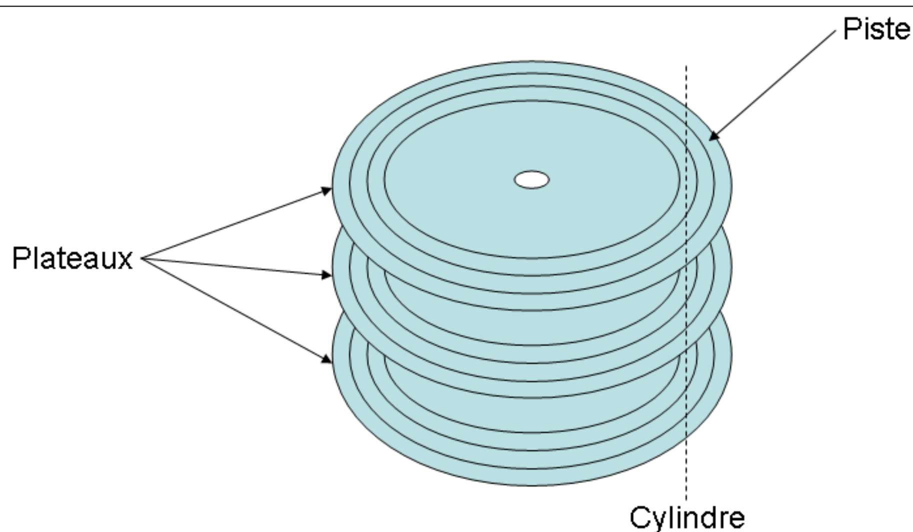
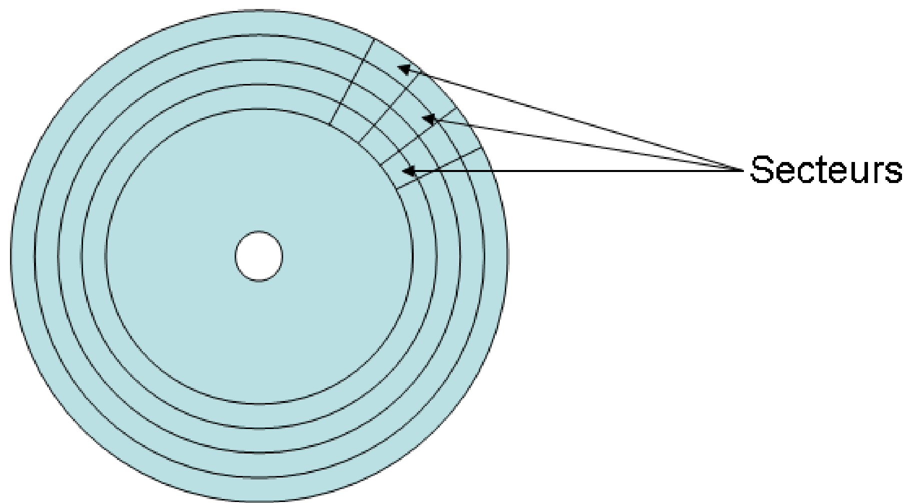


FIG. 2 Plateau d'un disque dur



Comme nous l'avions présenté dans la première partie du cours, les transports de données vers et depuis le disque sont généralement gérés par un contrôleur DMA (*Direct Memory Access*). Les accès disque se font donc sans recourir au processeur. Pour initier un échange DMA, un bloc de commande est envoyé au contrôleur, contenant :

– Le type de l'opération (lecture, ecriture)
– L'adresse sur le disque
– L'adresse en mémoire centrale
– Le nombre d'octets à transférer

A la fin de l'opération, le contrôleur DMA génère une interruption, informant le système d'exploitation que le transfert est terminé.

1.2 Définition d'un *filesystem*

Globalement, on peut considérer un *filesystem* comme une stratégie d'organisation des ressources de stockage. On définit généralement quatre élements fondamentaux composant un *filesystem*

1. Un *espace de nom*, c'est-à-dire une hiérarchie organisationelle dans laquelle il est possible de nommer des objets (les fichiers, les répertoires, ...)
2. Une API (*Application Program Interface*), c'est-à-dire un ensemble de primitives (induisant des appels systèmes) permettant de gérer les objets et de naviguer dans le *filesystem*
3. Un *modèle de sécurité*, c'est-à-dire une stratégie permettant de partager et de sécuriser (protéger) l'accès aux objets
4. une *implémentation*

Il existe de nombreux systèmes de gestion de fichiers : ext2/ext3/ext4, reiserfs, JFS, XFS, ZFS, FFS, UFS2, VFAT, NTFS, SMB, NFS, ... On remarquera que les deux derniers sont des *filesystem* réseaux, mais ils possèdent malgré tout les éléments précédemments cités.

1.3 Cohérence et intégrité

Au sein d'un *filesystem* on désigne par méta-données [*metadata*], les structures de données (description et pointeurs tels que *inodes*, répertoires et *free block maps*) qui permettent

– d'identifier et de structurer les secteurs disques en fichiers (logiques),
– de décrire les changements à effectuer sur le *filesystem* suite à une modification

Un *filesystem* doit maintenir l'intégrité de ces méta-données, notamment en cas de *crash* ou d'arrêt imprévu. La plupart de ces évènements conduisent à la perte des informations contenues dans la mémoire centrale, aussi les informations sur les supports non volatiles (disques, ...) doivent être suffisemment consistantes pour permettre une reconstruction du *filesystem* le mettant dans un état cohérent.

Pour garantir la consistance (c'est-à-dire la cohérence) de l'état du *filesystem*, le système doit opérer de manière atomique et synchrone (donc à la vitesse du disque), dans les trois cas suivants :

– Ecrire le nouvel inode (cf chapitre suivant) alloué avant que sa désignation ne soit inscrite dans le répertoire qui le contient ;
– Inversement, supprimer une désignation avant que son inode associé ne soit désalloué ;
– Ecrire l'inode désalloué sur disque avant de le placer dans le bitmap des blocs libres du groupe de cylindres.

Ces opérations assurent :

– qu'un répertoire référence toujours des *inodes* valides ;
– qu'un bloc de données n'est jamais réclamé par plus d'un *inode* ;

Cependant, du fait que le *filesystem* doive faire deux opérations synchrones pour la création ou la destruction de chaque fichier, le débit du filesystem est limité par la vitesse d'écriture disque notamment lors d'opérations de création ou suppression de masse [bulk].

En cas de *crash*, lié par exemple à une coupure de courant, le *filesystem* doit être entièrement vérifié pour corriger toute inconsistance (par exemple, un bloc désalloué annoncé comme occupé...). Le problème de cette vérification (menée par **fsck**) est que sa durée est proportionnelle à la taille de la partition, ce qui allonge de manière considérable le délai de reprise.

1.4 Montage et démontage d'un *filesystem*

Contrairement aux environnement Windows™, où le contenu de chaque disque et chaque partition (les volumes) apparait dans une arborescence distincte, sous Unix ces volumes sont accessibles depuis une même racine (*root /*). On appelle *montage d'un volume* l'opération qui rend accessible son contenu, et *démontage* l'opération qui le rend inaccessible.

1.4.1 "*Remontage*" après *crash*

Les recherches dans ce domaine ont dégagé trois axes principaux pour résoudre ce problème :

1. Des techniques basées sur des technologies de *mémoires vives permanentes* (à base de SRAM). Extrêment efficaces mais (encore actuellement) très onéreuses.
2. L'introduction du concept de *journalisation* (*journalling*), issue du monde des bases de données et qui consiste à tenir un journal des méta-données qui est écrit sur disque préalablement à toute autre modification des données. Ce journal permet de définir des points de synchronisation où la consistance du filesystem est garantie. En cas de *crash*, le système va relire son journal et se synchroniser relativement à la dernière transaction (validée).

   Cette approche ne garantie pas que des données ne soient pas perdues (du fait d'une transaction avortée car non conduite à son terme), mais garantie par contre un état consistant après le rejeu du journal des transactions. La reprise est beaucoup plus rapide puisqu'elle ne nécessite pas un *scan* complet du filesystem, mais le surcoût lié à la gestion du journal est loin d'être négligeable et reste un problème ouvert en recherche.

   Cette stratégie a été adoptée par tous les *Unix* propriétaires et elle est désormais suivie par le système *GNU/Linux* qui dispose de plusieurs implémentations (deux libres) : ext3fs,ext4fs (extension journalisée du filesystem Linux traditionnel (ext2fs), reiserfs et deux autres originellement propriétaires (qui ont fait l'objet de dons à la « communauté du libre ») : XFS (de SGI™) et JFS (d'IBM™).
3. Les *softupdates* du monde *BSD* ; cette technique consiste à réordonner les opérations d'écritures disques, c'est-à-dire à maintenir un ordre partiel sur les opérations de mises à jour, autrement dit les méta-données et ce de telle sorte qu'elles soient toujours dans un état consistant, ce qui implique la nécessité de résoudre des cycles de dépendances non triviaux. L'utilisation de techniques de rollback partiels (on ignore momentanément, en les mémorisant, certaines opérations de mises à jour pour rompre le cycle puis on les ré-active) pour éliminer ces cycles de dépendances permet alors d'éliminer 95% des écritures synchrones (les softupdates utilisent une approche asynchrone).

Cette dernière approche permet de minimiser les inconsistances des méta-données et facilite l'opération de recouvrement en cas de *crash* (les seules erreurs possibles sont les blocs et inodes marqués comme alloués alors qu'ils sont libres), néanmoins cette dernière requiert un *scan* complet du *filesystem*. Les

erreurs restantes, non fatales sont corrigées avec un **fsck** s'exécutant en arrière plan (**background fsck**), lors du redémarrage de la machine, ce qui permet de repartir immédiatement au prix il est vrai d'un ralentissement lié au scan.

2 Fichiers et répertoires

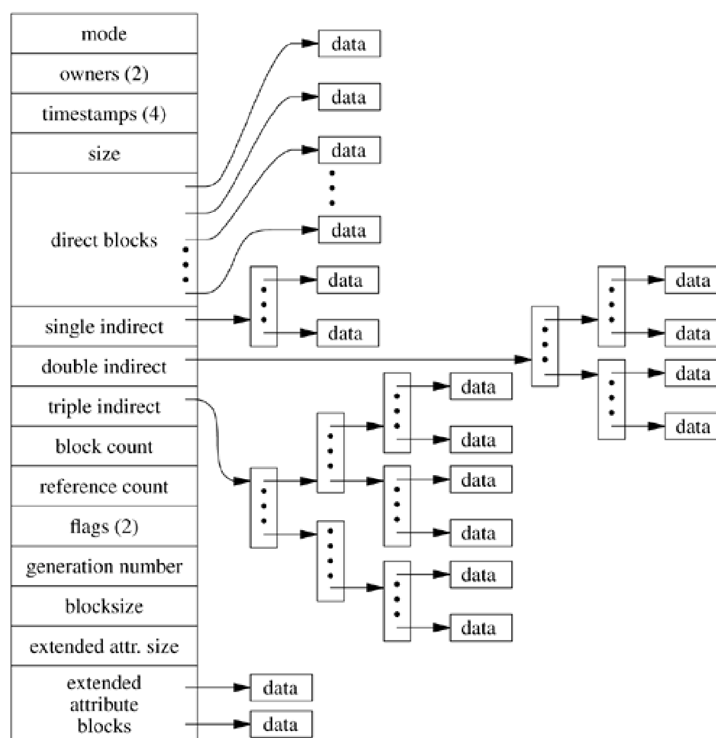
2.1 inodes et répertoires

2.1.1 Inodes

Chaque fichier du *filesystem* correspond à une entrée dans une table contenant l'ensemble de ses attributs (à l'exeption notable de son nom!). Cette entrée (*index*) est appelée *inode (index node*) : il s'agit en fait d'une structure de données comprenant en particulier les champs suivants :

– le type du fichier
– le mode d'accès au fichier
– le propriétaire, le groupe
– la date du plus récent accès en lecture et écriture
– la date la plus récente de mise à jour de l'inode
– la taille exprimée en octet
– Le nombre de blocs physiques utilisé par le fichier (incluant les éventuels blocs indirects contenant les pointeurs vers les blocs de données)
– le nombre de référence sur l'inode
– les drapeaux décrivant les caractéristiques du fichier
– le nombre généré (identifiant unique, utilisé par exemple par le protocole NFS)
– le tableau des pointeurs référençant les blocs de données du fichier.

FIG. 3 Structure d'un inode (Source : The Design and Implementation of the 4.4BSD Operating System)

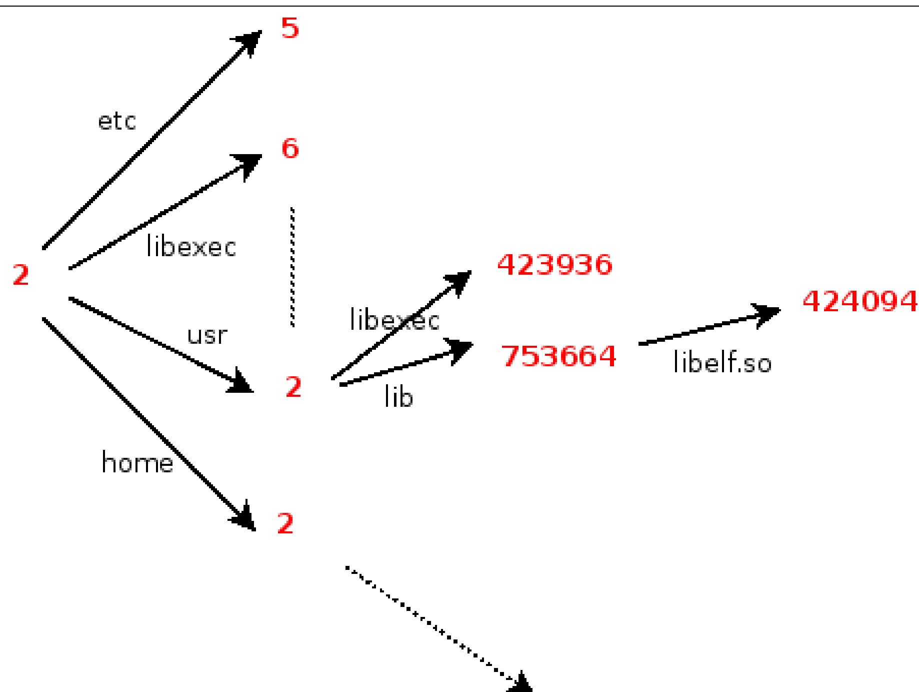


2.1.2 Répertoires

Les répertoires sont des fichiers d'un type particulier : ils contiennent des pointeurs vers des fichiers qui peuvent être eux-mêmes des répertoires (définition récursive). La hiérarchie obtenue est structurée en arbre (en fait un *graphe acyclique*, ce qui permet de conserver des algorithmes simples). L'exemple ci-dessous montre une telle organisation sur un système *FreeBSD* :

Exemple 2.1 Arborescence BSD

FIG. 4



Chaque noeud de l'arbre (en rouge) correspond à un inode. La racine absolue (*root*), d'index 2 contient entre autres les entrées vers les inodes 5 (le lien `etc`), 6 (le lien `libexec`).... On peut remarquer que les inodes correpondant aux liens `usr` et `home` sont 2 : ils sont racines d'un disque logique qui est un point de montage. Enfin, un fichier, `libelf.so`, est représenté à titre d'exemple, d'index 424094.

Un répertoire Unix est alloué en unités appelées *chunks*. La taille d'un *chunk* est choisie de telle sorte que toute allocation puisse être transférée sur disque en une seule opération. Cette propriété caractéristique permet de garantir une mise à jour atomique des répertoires.

Les *chunks* sont fragmentés en entrées de longueurs variables ce qui permet des noms de fichier de longueur quelconque (en pratique limité à 255 caractères tant sous BSD que dans le standard POSIX). Chaque entrée de répertoire est structurée comme suit :

- un index dans la table des inodes disques
- la taille de l'entrée exprimée en octet
- le type de l'entrée (fichier régulier, répertoire, ...)
- la longueur du nom de fichier contenu par l'entrée, exprimée en octet
- le nom du fichier terminé par le caractère NULL et complété (éventuellement) par un bourrage pour l'alignement

2.2 Types de fichiers

2.2.1 Fichiers réguliers

Les fichiers réguliers (*regular files*) sont des fichiers sur disque, dont le contenu est non structuré : il s'agit d'une suite de caractères, possédant une longueur. Il n'y a pas au niveau du système d'exploitation

de notion d'organisation (séquentielle, indexée, relative, ...), ce sont les applications qui utilisent ces fichiers qui doivent implémenter si nécessaire ces organisations. Le contenu de ces fichiers peut être des programmes (binaires ou sources), ou des données de programmes utilisateurs ou systèmes.

2.2.2 Fichiers spéciaux

Nous l'avons déjà plusieurs fois souligné, les *Unix* sont basés sur le *credo* du 'tout est fichier' : pour les utilisateurs, il n'y a pas de différence entre les fichiers réguliers, les répertoires et les organes périphériques du point de vue de leur référencement. Les fichiers spéciaux sont ainsi associés à un dispositif physique, et traités par le système comme des fichiers classiques (lecture, écriture, ..). Mais bien sûr, ces opérations activent des périphériques physiques. On distingue deux types de fichiers spéciaux :

- les fichiers en *mode bloc* (les disques durs par exemple), pour lesquels les données sont envoyées/-reçues par bloc, et transitent par des caches du sytème.
- les fichiers en *mode caractère*, comme les terminaux.

2.3 Droits

Les *Unix* définissent les droits d'accès sur un objet du *filesystem* selon trois droits fondamentaux, applicables aux fichiers comme aux répertoires :

- *r* : lecture (*read*); cette valeur binaire détermine si le fichier peut être lu, ou si le répertoire peut être listé.
- *w* : écriture (*write*); cette valeur binaire détermine si le fichier peut être modifié, renommer ou supprimer, ou si il est possible de supprimer des fichiers dans le répertoire.
- *x* : exécution (*execution*); cette valeur binaire détermine si le fichier peut être exécuté, ou si le répertoire peut être traversé.

Ces droits sont répartis pour trois types d'utilisateurs pour chaque fichiers :

- *u user*: l'utilisateur normal, son propriétaire, bien souvent son créateur, qui n'a pas pour autant tous les droits sur lui !
- *g group*: son groupe, ensemble d'utilisateurs ayant parfois des "permissions" particulières ;
- *o others* : tous les autres.

Lors de l'exécution de la commande **ls -l** on obtient des informations sur les fichiers et répertoires :

FIG. 5 Droits d'accè à un fichier

FIG. 6 Droits d'accès à un répertoire

2.4 Droits étendus

2.4.1 Bit SUID

Sa présence permet à un fichier exécutable de s'exécuter sous l'identité du propriétaire et donc avec ses droits, ce qui permet à un utilisateur (non propriétaire) d'avoir des droits plus étendus, indispensables pour cette exécution. Ce droit est noté symboliquement *s* et se positionne à la place du *x* du propriétaire (classe *u*, mais en conservant le droit *x*). Sa valeur octale est : 4000

Exemple 2.2 Exemple de bit SUID

```
kaitan# ll /usr/bin/passwd
-r-sr-xr-x  2 root  wheel  6120 Mar 29 19:56 /usr/bin/passwd
kaitan# ll /etc/master.passwd
-rw-------  1 root  wheel  2254 Apr  9 02:02 /etc/master.passwd
```

Ici, chacun peut exécuter la commande **/usr/bin/passwd**. Mais comme seul `root` a les droits d'écriture sur le fichier `/etc/master.passwd` (qui contient les mots de passe cryptés), le bit SUID permettra à un utilisateur de changer son propre mot de passe.

2.4.2 Le bit SGID

Pour un fichier exécutable, il fonctionne de la même façon que le SUID, mais transposé aux membres du groupe. Positionné sur un répertoire, ce droit modifie le groupe propriétaire d'un fichier créé dans ce répertoire. Un fichier créé dans un tel répertoire, verra son groupe propriétaire modifié :

- Ce ne sera plus le groupe primaire de son créateur et propriétaire (règle habituelle), mais le groupe propriétaire du répertoire lui-même
- On peut dire que ce droit posé sur un répertoire fait hériter tous ses fichiers de son groupe propriétaire
- Notation symbolique s, mis à la place du x du groupe

Sa valeur octale est : 2000.

2.4.3 Le *sticky bit*

Ce droit spécial a surtout un rôle important sur les répertoires. Il réglemente le droit *w* sur le répertoire, en interdisant à un utilisateur quelconque de supprimer un fichier dont il n'est pas le propriétaire. Ce droit noté symboliquement t occupe par convention la place du droit *x* sur la catégorie *other* de ce répertoire, mais bien entendu il ne supprime pas le droit d'accès x (s'il est accordé). Justement, si ce droit *x* n'est pas accordé à la catégorie other, à la place de *t* c'est la lettre *T* qui apparaîtra. Sa valeur octale associée vaut 1000. L'exemple le plus connu est bien sûr le répertoire `/tmp`.

3 Virtual file system

3.1 Présentation

Les premiers *Unix* étaient conçu de manière à ce que le système fasse directement référence aux inodes. La nécessité de s'interfacer avec d'autres systèmes de gestion de fichiers a abouti finalement à la mise au point par Sun™ d'une couche intermédiaire entre les fichiers manipulés par le système et les *inodes*, le VFS (*Virtual FileSytem*) et ses *vnodes*.

3.2 vnodes

Le VFS est une interface orientée-objet extensible. Un *vnode* contient des informations génériques pour chaque fichier actif du système, chaque répertoire courant, etc .., auquel sont associés des informations spécifiques détenues par le *filesystem* contenant ce fichier. Le système maintient une table résidente en mémoire principale de tous les vnodes ; les entrées inactives sont quant à elles réutilisées sur la base d'une stratégie de type LRU .

Parmi les informations stockées dans un *vnode*, on trouve :

- des flags utilisés pour identifier les attributs génériques. Exemple : un flag permettant de déterminer qu'un vnode représente un objet qui est racine d'un filesystem.
- des compteurs de référence tels que :
  - compteur indiquant le nombre de fichiers ouvert en L et/ou E référençant le vnode.
  - compteur indiquant le nombre de fichiers ouvert en E référençant le vnode.
  - le nombre de pages et de buffers associés avec les vnodes.
- un pointeur sur la structure montée, décrivant le filesystem qui contient l'objet décrit par son vnode associé.
- des informations permettant des lectures anticipées .
- une référence sur la structure vm_object associée au vnode.
- une référence sur l'état des périphériques spéciaux, sockets, FIFOs.
- un mutex protégeant les flags et les compteurs du vnode.
- un gestionnaire de vérouillage qui permet de protéger les parties du vnode susceptibles de changer lors des opérations d'E/S.
- des champs utilisés par le cache de nom, pour conserver les associations nom, vnode.
- un pointeur sur un ensemble d'opérations du vnode définis pour l'objet.
- un pointeur sur des informations privées nécessaires à l'objet (sous-jaccent au vnode). Exemples : pour un filesystem local c'est typiquement une référence sur un inode, pour un filesystem réseau de type NFS c'est une référence à un nfsnode.
- le type de fichier sous-jaccent (fichier régulier, répertoire, ...). Cette information n'est pas stricto-sensu nécessaire, en effet un appel à une opération du vnode permet d'obtenir cette information. Remarquons cependant que 1) cet appel a un coût, 2) la réponse reste invariante pour un vnode (actif) donné et 3) cette information est souvent nécessaire ; la mise en cache est plus efficace.
- la liste des clean buffers (ces buffers non encore modifiés ou modifiés et écrits sur disques) et des dirty buffers (buffers contenant des données modifiées mais non encore écrites sur disque).
- enfin, un compteur des buffers faisant l'objet d'opérations d'écriture (pour des raisons de performances, le kernel gère ces écritures de manière asynchrone sur tous les dirty buffers à la fois).

4 Exercices

4.1 Commandes Unix

4.1.1 Exercice 1

Modifier la valeur du masque courante `umask`, pour respecter la contrainte suivante

1. Tous les droits pour le propriétaire seul.
2. Tous les droits pour le propriétaire, droits de lecture/exécution pour le groupe du propriétaire.
3. Tous les droits pour l'utilisateur, droits de lecture/exécution pour le groupe, lecture seul pour les autres.

Dans chaque cas, tester le en créant un fichier, un répertoire. Quelle est la valeur courante de votre `umask` ? signification ? Dans quel fichier sa valeur est-elle définie ?

4.2 *filesystem*

4.2.1 Liens

Expliquer la différence entre liens symboliques et liens durs. Pourquoi les liens durs ne peuvent pas appartenir à des filesystem différents ?

4.2.2 Graphe acyclique

1. Tester la transitivité de la notion de lien symbolique (c'est-à-dire un lien vers un lien vers une désignation d'un objet du filesystem).
2. Tester l'acyclicité du SGF en créant une circularité de liens symboliques.

4.2.3 Droits

Soit la séquence de commandes suivante, dont les 3ème et 5ème commandes ont été masquées par des étoiles (les commandes ne sont pas réalisées par le super utilisateur *root*) :

```
$ ls tmp/
fic1 fic2 fic3
$ ls -li tmp/
total 14
117062441 -rwxr-xr-x  2 ikare  bsd  13845 Apr 26 21:05 fic1
117062442 -rwxr-xr-x  1 ikare  bsd      0 Apr 26 18:48 fic2
117062441 -rwxr-xr-x  2 ikare  bsd  13845 Apr 26 21:05 fic3
$ sudo *************
$ ls -l tmp/
total 0
ls: tmp/: Permission denied
$ sudo *************
$ ls -l tmp/
ls: fic1: Permission denied
ls: fic2: Permission denied
ls: fic3: Permission denied
total 0
```

On suppose qu'aucune autre mainpulation sur le SGF n'a été faite entre les commandes. *Donner les commandes qui ont été masquées par les* * Un utilisateur fait ensuite les manipulations suivantes :

```
$ sudo rm tmp/fic3
$ ls -l tmp
ls: fic1: Permission denied
ls: fic2: Permission denied
total 0
```

Si l'utilisateur souhaitait retrouver son fichier « fic3 », quelle serait votre démarche ?

5 Corrections

5.1 Translation

Un programme contenant du code translatable a été créé, en partant du principe qu'il serait chargé à l'adresse 0. Dans son code, le programme fait référence aux adresses suivantes: 25, 65, 112, 156, 213. Si le programme est chargé en mémoire en commençant à l'emplacement 250n comment doivent être ajustées ces adresses ?

Réponse : On doit rajouter 250 à chaque fois : 275, 315, 362, 406, 463

5.2 Performances

5.2.1 Partie 1

Sur un système, vous observez des performances en deça de vos espérances. Après un monitoring sur plusieurs heures en fonctionnement normal, vous remarquez les points suivants :

- Taux d'utilisation du processeur faible
- Utilisation du *swap* très haute
- Utilisation des périphériques faibles

Réponses :

1. Remplacement du processeur par un processeur plus puissant : *peu d'impact*
2. Augmentation de la taille de la partition de *swap* : *aucun effet*
3. Installation d'un espace de *swap* plus rapide (sur un nouveau disque dur par exemple) : *Oui*
4. Installation de RAM supplementaire : *Oui, on aurait moins de permutations*
5. Installation de RAM plus rapide : *aucun effet*

6. Accroissement du degré de multi programmation (par exemple en augmentant la valeur du maximum de processus autorisé) : *encore pire : plus de processus, donc plus de permutations*
7. Diminution du degré de multi programmation (par exemple en diminuant la valeur du maximum de processus autorisé et en supprimant certains programmes lancés automatiquement) : *Oui, directement efficace*
8. Changement de certains périphériques par des plus rapides : *peu d'impact*

5.2.2 Partie 2

Même question si cette fois ci le monitorig avait révélé :
- Taux d'utilisation du processeur faible
- Utilisation du *swap* Faible
- Utilisation des périphériques élevé

REPONSES :
1. Remplacement du processeur par un processeur plus puissant : *peu d'impact*
2. Augmentation de la taille de la partition de *swap* : *aucun effet*
3. Installation d'un espace de *swap* plus rapide (sur un nouveau disque dur par exemple) : *Pas d'effet*
4. Installation de RAM supplementaire : *Pas d'effet*
5. Installation de RAM plus rapide : *Pas d'effet*
6. Accroissement du degré de multi programmation (par exemple en augmentant la valeur du maximum de processus autorisé) : *sans doute une amélioration*
7. Diminution du degré de multi programmation (par exemple en diminuant la valeur du maximum de processus autorisé et en supprimant certains programmes lancés automatiquement) : *Encore pire*
8. Changement de certains périphériques par des plus rapides : *sans doute une amélioration*

5.3 Remplacement de page (Source : Ivan Boule, Examen SMB137 2004)

On dispose d'un système de mémoire paginée à la demande et de deux algorithmes A et B. On observe l'exécution d'un programme auquel le système alloue 3 cases de mémoire centrale et qui accède successivement aux pages : 2 3 5 2 1 5 2 4 5 3 2 5 2 3.

Chaque colonne représente, à un instant donné, de haut en bas, l'ordre de remplacement des pages en mémoire selon l'algorithme utilisé. Avec l'algorithme A, on constate qu'il y a successivement en mémoire les pages suivantes :

```
état        2 2 2 3 5 2 1 5 2 4 5 3 3 5
après         3 3 5 2 1 5 2 4 5 3 2 5 2
chargement      5 2 1 5 2 4 5 3 2 5 2 3
                    X     X   X X
```

Avec l'algorithme B, on constate qu'il y a successivement en mémoire les pages suivantes :

```
état        2 2 2 2 3 3 5 1 2 4 5 5 5 5
après         3 3 3 5 5 1 2 4 5 3 3 3 3
chargement      5 5 1 1 2 4 5 3 2 2 2 2
                    X   X X X X X
```

1. Lequel des deux algorithmes correspond à l'algorithme FIFO et lequel correspond à l'algorithme LRU ? Justifier votre raisonnement. Déterminer dans chaque cas le nombre total de défauts de page
   A. LRU - 3 chargements, 4 remplacements
   B. FIFO - 3 chargements, 6 remplacements
2. Quelles auraient été les pages en mémoire avec l'algorithme Optimal? et quel serait le nombre total de défauts de page?
   OPT : 3 chargements, 3 remplacements

```
état         2 3 3 3 1 1 1 4 4 3 2 5 5 3
après          2 5 2 2 5 2 2 5 5 3 3 2 2
chargement       2 5 5 2 5 5 2 2 5 2 3 5
remplacement         X       X     X
```

5.4 Chronogramme

On considère une mémoire centrale avec 4 cases initialement vides. Un processus P effectue les références suivantes aux pages de son espace virtuel : 0,2,9,4,5,2,0,9,3,2,0,9,5,3 Complétez le diagramme ci dessous pour représenter l'évolution de la mémoire centrale au fur et à mesure des accès du processus à ses pages et notez les défauts de pages existants :

FIG. 7 Exercice 3

Acces	0	2	9	4	5	2	0	9	3	2	0	9	5	3
Case 1														
Case 2														
Case 3														
Case 4														

1. pour un remplacement de pages de type FIFO (*First In, First Out*)
2. pour un remplacement de pages de type LRU (*Least Recently Used*)

5.5 Comparaison d'algorithmes

Considérant un programme qui a pour chaîne de référence les pages [0,1,4,2,0,1,3,0,1,4,2,3] donner la suite des pages présentes en mémoire centrale et le nombre de défauts de pages dans les cas suivants :

1. Algorithme de remplacement FIFO et espace de 3 cadres de pages allouables : 9

```
référence           0 1 4 2 0 1 3 0 1 4 2 3
défauts             X X X X X X X     X X
                    0 0 0 2 2 2 3 3 3 3 3 3
                      1 1 1 0 0 0 0 0 4 4 4
                        4 4 4 1 1 1 1 1 2 2
```

2. Algorithme de remplacement LRU et espace de 3 cadres de pages allouables

```
référence           0 1 4 2 0 1 3 0 1 4 2 3
défauts             X X X X     X X X X X X
                    0 0 0 0 0 0 3 3 3 3 2 2
                      1 1 1 1 1 1 0 0 0 0 3
                        4 4 4 4 4 4 1 1 1 1
                          2 2 2 2 2 2 4 4 4
```

3. Algorithme de remplacement FIFO et espace de 4 cadres de pages allouables : 10 défauts de page (anomalie de Belady : plus de défauts malgré plus de pages)

```
référence           0 1 4 2 0 1 3 0 1 4 2 3
défauts             X X X X X X X     X X X
                    0 0 0 2 2 2 3 3 3 4 4 4
```

```
                    1 1 1 0 0 0 0 0 0 2 2
                      4 4 4 1 1 1 1 1 1 3
```

4. Algorithme de remplacement LRU et espace de 4 cadres de pages allouables : 8 défauts de page

```
référence          0 1 4 2 0 1 3 0 1 4 2 3
défauts            X X X X     X     X X X
                   0 0 0 0 0 0 0 0 0 0 0 3
                     1 1 1 1 1 1 1 1 1 1 1
                       4 4 4 4 3 3 3 3 2 2
                         2 2 2 2 2 2 4 4 4
```

6 Références

6.1 Références

[1] Paolo ZanellaYves Ligier, *Architecture et technologie des ordinateurs* , Dunod , isdn : ISBN 2 10 003801 X, 199,2002.

[2] Matt WelshKalle DalheimerLar Kaufman, *Le systeme Linux* , O'Reilly , 1995, 1997, 1999, 2000.

[3] Joelle Delacroix, *Linux* , Dunod , 2003.

[4] Jean-Marie Rifflet, *La programmation sous Unix*, EdiScience , 1986, 1989, 1993, 2002.

[5] Michael W. Lucas, *Absolute Freebsd: The Complete Guide to Freebsd*, No Starch Press , 2007.

[6] Michael W. Lucas, *FreeBSD 7.0*, No Starch Press , 2007.

[7] Nicholas P. Carter, *Architecture de l'ordinateur*, EdiScience - Schaum's , 2002.

[8] J. Archer Harris, *Systèmes d'exploitation*, EdiScience - Schaum's , 2002.

[9] Eric Steven Raymond, *The art of Unix Programming*, Thyrsus Enterprises , 2003.

[10] Pascal Picard, *Eléments du système de gestion de fichiers*, Corto, Inc , 2005.

[11] Marshall Kirk McKusickKeith BosticMichael J. KarelsJohn S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Longman, Inc , 2010.