Ministère de l'Éducation Nationale

Conservatoire National des Arts et Métiers

Institut d'Informatique d'Entreprise

RAPPORT DE PROJET

Administration Système et Réseau

Système de fichiers hybride sous GNU/Linux

Yoann GUILLOT

Chargé de TD : **M. Ivan AUGÉ**

Table des matières

Chapitre 1

Introduction

1.1 Résumé

Le but de ce projet est la réalisation d'un système de fichier de type NFS, implémenté au niveau système dans un noyau de type Linux 2.6. Le système sera une amélioration du système NFS car chaque bloc de données lu sur le réseau sera caché sur un disque local, de manière à accélerer les prochains accès à ces données, et ce de manière totalement transparente pour l'utilisateur.

1.2 Problématique

L'administration d'un ensemble de machine de type Unix est grandement facilité par l'existence du système de fichiers NFS. Le système NFS est un système de fichiers qui permet de partager des fichiers entre différentes machines par l'intermédiaire d'un réseau UDP/IP, sous la forme d'un serveur qui exporte des 'partages', c'est à dire une arborescence de fichiers. Cela permet d'avoir des ordinateurs disposant du meme ensemble de programmes, et on peut ajouter une fonctionnalité très facilement, sans avoir a passer sur tous les postes individuellement.

L'inconvénient de ce type de configuration est que chaque accès aux fichiers passe toujours par le réseau, même en cas d'utilisations répétées des mêmes fichiers.

D'où l'idée d'implémenter une sorte de cache, qui lit une première fois sur le réseau les données qu'on lui demande, mais qui les cache de manière à éviter de passer par la couche réseau à la prochaine utilisation.

1.3 Plan

Je vais tout d'abord parler des principes du système de fichiers virtuel sous Linux. Puis je présenterais le travail que j'ai effectué, et les raisons de l'échec du projet tel que je l'ai concu, et enfin je dirais ce que je pense maintenant que j'aurais du faire pour aboutir un système qui marche.

Chapitre 2

Le système de fichiers virtuel sous Linux

2.1 Contexte

Linux est un noyau concu de manière modulaire. C'est une des causes de son succès.

Il est en effet très facile de lui ajouter des fonctionnalités, ou de faire en sorte qu'il supporte des matériels dont les codeurs originels n'ont jamais eu connaissance. Le noyau dans la série 2.6 est organisé en sections distinctes :

1. Une section liée à la gestion de la mémoire, que l'on trouve dans le code source du noyau dans le répertoire `/mm`
2. Une section liée à la gestion du réseau, dans le répertoire `/net`
3. Une section pour la gestion du son, sous `/sound`
4. Une section de documentation dans `/Documentation`, qui n'est toutefois pas très à jour
5. Une section pour les systèmes de fichiers dans `/fs`
6. Une section pour les mécanismes cryptographiques dans `/crypto`
7. Une section rassemblant tout le code spécifique à une architecture (à une famille de processeur en général) dans `/arch`[1]
8. Le code d'initialisation du noyau est dans `/init`
9. Le code générique du noyau dans `/kernel`
10. Les différents *drivers* pour le matériel sont dans `/drivers`

La partie qui nous intéresse, à savoir `/fs`, contient deux catégories de fichiers : les fichiers sources, qui sont des fichiers génériques, qui implémentent a la fois le système de fichiers virtuel (*VFS, Virtual Filesystems Switch*), et les routines par défaut pour les systèmes de fichiers.

La deuxième catégorie est constituée de l'ensemble des sous-répertoires de `/fs`, et représente les véritables systèmes de fichiers utilisés (un système par répertoire).

Le *VFS* est une couche d'abstraction représentant un système de fichiers générique, il utilise les notions d'**inode**, **dentry**, **pointeur de fichier**, **mapping mémoire**.

Tout système de fichier spécifique doit être traduit pour correspondre à ce paradigme pour pouvoir être utilisé (ou **monté**) par le noyau Linux. Ce système est en fait très proche du système natif de Linux, *ext2*, de manière à pouvoir utiliser celui-ci avec le minimum de surcharge à l'exécution.

[1] Tout le code du noyau est totalement indépendant de l'architecture de l'ordinateur pour lequel il est compilé, à l'exception du code dans `/arch`

2.2 VFS

Dans ce paradigme, les fichiers sont organisés dans des *dossiers*, qui sont eux-même des fichiers contenant la liste des fichiers et dossiers qu'ils contiennent. L'arborescence doit être strictement hiérarchique, c'est à dire qu'un dossier n'a qu'un dossier *parent*, et un dossier ne peut contenir un de ses parents.

Les données (contenu des fichiers) sont strictement séparées des méta-données. Les méta-données sont stockées par fichier dans une structure appelée **inode**, qui contient, entre autres, la taille du fichier, l'identifiant du propriétaire, les permissions, et d'autre méta-données. Elle contient également un ensemble de méthodes pour les différentes opérations autorisées sur les inodes (déplacement du fichier dans le système de fichier, changement de propriétaire, lecture du contenu d'un répertoire..), nommées **inode_operations**, et un ensemble de méthodes relatives au contenu du fichier lui-même (ouverture, lecture, écriture, mmap...) nommées **file_operations**.

Un dossier est une collection de **dentry**. Une **dentry** est un élément de base du système de fichiers virtuel, c'est une structure qui contient le nom d'un fichier, un pointeur vers l'inode de ce fichier, et des références vers d'autres dentry : parent, *siblings* (autres fils du parent)...

Ce sont eux qui forment l'arborescence à proprement dite. Comme toutes les interactions avec l'environnement utilisateur se font à partir de **chemins** absolus (c'est à dire la séquence de répertoires à partir de la racine du système de fichiers) ou relatifs (c'est la même chose mais à partir du répertoire de travail du processus invocateur), un système complexe de cache de dentry (le **dcache**) à été mis au point.

La base de tout système de fichier dans Linux est une structure nommée **superbloc**. Cette structure contient (entre autres) une référence vers la **dentry** racine de l'arborescence de ce système de fichier, ainsi qu'une référence à une structure **vfsmount** qui indique où ce système est monté.

Chapitre 3

Le projet

3.1 Présentation

Le but du projet est de créer d'une manière ou d'une autre une interface entre deux systèmes de fichiers existant, de manière à stocker sur un des deux (que j'appelerais *local* dorénavant) tout morceau de fichier lu sur l'autre (*remote*).

Une autre caractéristique du système virtuel est qu'il doit pouvoir être monté même si le *remote* est indisponible (pas de réseau pour un système NFS par exemple).

Ceci implique que le système *local* soit une réplique quasi-identique du *remote*.

3.1.1 Questions - Réponses

Il faut alors trouver un moyen pour enregistrer quelles parties de chaque fichier ont déjà été recopiées ; on doit également définir l'état initial du système *local*.

On peut envisager d'utiliser des fichiers, que l'on cachera à l'utilisateur (en truquant le résultat des appels systèmes de lecture du contenu d'un répertoire par exemple). Toutefois ces fichiers seront bien réels sur le système *local*, et de ce fait devront posseder un numéro d'inode ; il faudra alors veiller à ce que ce numéro ne soit pas utilisé sur le système *remote*. On peut aussi utiliser des numéros d'inode différents entre les deux systèmes, mais cela compliquerait inutilement le projet.

Pour la question de l'initialisation, trois scénarios sont envisageable :

1. Initialisation totale. On recréée toute l'arborescence du *remote* sur le *local*, répertoires et fichiers. Les fichiers sont initialisés comme complets à 0%. On créé aussi les méta-fichiers le cas échéant. C'est la solution la plus simple.
2. Initialisation partielle. Dans ce cas, on recréé toute l'arborescence, mais sans spécifier les fichiers, juste les répertoires. On peut encore créer les méta-fichiers, il faut alors faire attention aux numéros d'inode.
3. Initialisation nulle. Là on n'initialise rien du tout, et le système de recopie fera tout le travail. On a alors un problème avec les méta-fichiers, sauf si l'on n'en utilise qu'un à la racine du système. Il faut également faire attention à son numéro d'inode.

Pour ma part, je choisis l'initialisation totale, avec un méta-fichier par répertoire qui indexe tous les fichiers de ce répertoire. Les répertoires n'ont pas besoin d'être indexés puisque tout leur contenu est présent.

Un autre problème vient du fait que la taille d'un bloc de données peut changer d'un système de fichier à l'autre (ext2 autorise même différentes tailles de blocs, c'est une option à la création du système de fichier).

Il est alors plus simple de stocker les parties déjà présentes sur le système *local* sous forme de plages de bytes.

Un problème auquel j'ai été confronté et auquel je n'avais pas pensé est le fait que l'on peut lire un fichier avec une autre méthode que l'appel système sys_read, à savoir sys_mmap. C'est un appel système qui mappe le contenu d'un fichier dans un segment de la mémoire d'un processus, et qui est implémenté sans faire appel à la fonction read(), mais qui passe par un ensemble d'opération sur l'espace mémoire appelé **address_space_operations**, qu'il faudra également prendre en compte dans le système de lecture/écriture.

Chapitre 4

Travail effectué

J'ai essayé de réaliser ce projet d'une manière le moins intrusive possible dans le noyau, sous forme d'un système de fichiers indépendant. Ce système prend au montage des options lui permettant de monter les systèmes *local* et *remote* de manière virtuelle, et d'effectuer sur ceux-ci les manipulations qui s'imposent. Je l'ai souhaité indépendant de la nature exacte de ces systèmes de fichiers.

Je nommerai par la suite mon système de fichier comme étant *virtuel*.

4.1 Interface utilisateur

Pour fonctionner, mon système monte deux systèmes de fichiers, de manière virtuelle (c'est à dire que les **dentry** racine de ceux-ci ne sont pas accessible directement par l'utilisateur. Pour cela ils sont montés sur des **vfsstruct** qui ne sont pas rattachées à un point de l'arborescence existente. Ils ne sont donc accessible qu'indirectement en passant par le système *virtuel*.

4.1.1 Préinitialisation

Pour fonctionner le projet à besoin d'un système *remote* contenant des fichiers, normalement celui-ci existe déjà.

Le projet à également besoin d'un système *local*. Celui-ci doit être de taille suffisamment grande pour pouvoir contenir tout le contenu du *remote*. En effet, étant donné que mon code se place au niveau du **VFS**, il n'a pas accès directement à la gestion des blocs physiques de données, et ne peut donc pas par exemple définir un fichier de X méga-octets sans allouer les blocs de données correspondant.

On peut toutefois noter qu'avec une bonne gestion des erreurs, cette disposition n'est plus nécessaire : en effet, si le système *local* est plein, le processus de cache ne fonctionnera pas et tous les accès passeront par le *remote*. Cependant la fonctionnalité de ne pas monter le *remote* risque d'être affectée.

Le système *local* doit impérativement être vide avant le premier montage, sous peine de comportement indéfini. Les fichiers seront probablement inaccessibles, toutefois si un répertoire existe et porte le même nom qu'un répertoire du *remote*, celui-ci et tous ses fils risquent d'être inutilisables.

4.1.2 mount

Pour pouvoir monter un système de fichiers, le noyau à besoin de 4 éléments :

1. Un point de montage. Celui-ci n'est nécessaire que pour le système *virtuel*, puisque les autres ne sont pas implantés dans l'arborescence.
2. Un type de système de fichier. C'est une chaine de caractères identifiant le type de système de fichiers à monter (exemple : "ext2").
3. Un périphérique. Pour les systèmes de fichiers sur disque, c'est l'identifiant du disque contenant le système de fichier à monter. Le périphérique est passé sous la forme d'un **chemin absolu** vers un fichier de type **périphérique bloc**. Ceux-ci sont en général créés dans le répertoire /**dev**. Example : "/dev/hda1". Facultatif[1].
4. Des options. Ces options peuvent être classées en 2 catégories, les options génériques (comme *read-only*, *nosuid*...) et les options spécifiques à un système de fichiers (type de journalisation pour ext3...). Ces deux catégories sont bien distinctes au niveau du noyau. Les premières sont passées sous forme d'un ensemble de drapeaux d'un bit selon qu'elles sont activées ou pas, les autres sont passées sous forme de chaine de caractères à la fonction de chargement du système de fichier. C'est le programme utilisateur *mount* qui sépare ces deux catégories et qui créée le drapeau à passer à l'appel système.

Il faut donc trouver un moyen pour passer ces trois arguments au noyau pour chacun des systèmes de fichier.

J'ai choisi d'utiliser les options de mon système *virtuel* pour passer une chaine de caractères au noyau, ou elle sera analysée et traduite en ces éléments.

Pour cela je définis des options avec paramètre, de deux lettres chacune, la première indiquant à quel système elle doit être appliquée ("l" pour *local* ou "r" pour *remote*), et la deuxième indiquant la nature de l'information ("o" pour *option*, "t" pour *type* du système de fichier, et "d" pour le périphérique — *device* en anglais). J'utilise également une autre lettre ("n") pour indiquer que ce système ne doit pas être monté, et "i" pour indiquer que le système doit être initialisé (normalement utilisé uniquement au premier montage du *local*, avec le remote monté).

Le code renvoie une erreur si au moins un des systèmes n'a ni type ni l'option indiquant qu'il ne doit pas être monté.

Comme le *device* est facultatif, si l'option correspondante n'est pas passée, le code définit par défaut "none" comme valeur à passer au système subordonné.

Les options sont séparées par des virgules.

Si l'on choisi "yakafs" comme nom pour le système de fichiers *virtuel* et que l'on souhaite utiliser comme système *local* un système "ext2" sur "/dev/hda2" avec les options "ro,nosuid,errors=remount-ro", et sans monter le système *remote*, on utilisera la commande suivante :

```
root@localhost# mount -t yakafs none /mnt/point \
> -o rn,lt=ext2,ld=/dev/hda2,lo=ro,lo=ro,lo=nosuid,lo=errors=remount-ro
```

Comme une partie du travail est normalement faite par le programme utilisateur *mount*, par exemple le parcourt de la liste des options du système de fichier pour séparer les options génériques dans un drapeau des autres, il m'a fallu écrire le code faisant ce même travail pour les deux système de fichiers subordonnés. Etant dans le noyau, j'en ai fait une version simplifiée, c'est à dire qu'il est impossible d'utiliser deux montages en **loopback** étant donné la complexité des opérations nécessaire pour cela.

On peut toutefois s'arranger pour avoir un des deux systèmes de fichiers en **loopback** : avec le *remote* par exemple, on peut faire

[1] Certains systèmes de fichiers sont **virtuels**, comme "proc", ils n'ont alors pas besoin de périphérique, et on peut fournir n'importe quelle chaine de caractères pour cet argument.

Il existe également une technique nommée **loopback** permettant de monter un système de fichier stocké dans un fichier normal. On passe alors le chemin vers ce fichier au programme *mount*, qui mappe alors ce fichier sur un fichier de périphérique spécial et passe le nom de ce périphérique au véritable appel système **mount**, par exemple "/dev/loop0".

```
root@localhost# mount -t yakafs /path/to/file /mnt/point \
> -o ln,rt=ext2,rd=/dev/loop0,loop
```

Explication : Le programme *mount*, voyant l'option "loop" dans la liste des options va utiliser le premier périphérique de **loopback** disponible, et y mapper le fichier passé en tant que *device*, ici "/path/to/file". Il ne nous reste donc qu'à deviner le nom du périphérique qui sera choisi, ce qui est facile car *mount* utilise le premier périphérique disponible, donc "/dev/loop0" si aucun système de fichier n'est déjà monté en **loopback**. L'option "loop" est une option utilisée seulement par *mount* et n'est pas passée au noyau.

4.2 Code noyau

Le code noyau est sous forme d'un module que l'on peut rajouter au noyau Linux version 2.6.5 et certainement toutes les versions 2.6, voire 2.4. La seule modification nécessaire à faire à la base du noyau, mise à part la modification de */fs/Makefile* est l'export de la fonction *do_kern_mount*, qui permet de monter un système de fichiers sur une structure *vfsmount* en lui passant n'importe quels arguments, et sans la rattacher à l'arborescence.

4.2.1 Principe

Le principe sous-jacent à mon implémentation est que chaque élément du système de fichier virtuel soit un élément virtuel qui ait une référence vers son alter-égo dans chacun des systèmes subordonnés : chaque **inode** virtuelle possède un pointeur vers l'inode *locale* et vers l'inode *remote*. Les inodes subordonnées ne sont pas modifiées du tout, mais l'inode virtuelle est créée de manière à appeler des fonctions spécifiques pour chaque opération qu'on lui demande, qui sont censé gerer l'aiguillage entre le *remote* et le *local*, ainsi que le cache. De même pour chaque **dentry**, **pointeur de fichier**, et même pour le **superbloc**.

Comme on le verra par la suite, cette approche ne fonctionne pas.

4.2.2 Fonctionnement du code

A l'insertion du module (si le code est compilé en module, sinon a l'initialisation du noyau), le code enregistre le nouveau type de système de fichiers "yakafs" avec un pointeur sur **yakafs_get_sb**, qui est la fonction chargée d'initialiser le superbloc au montage du système.

Cette fonction appelle **yakafs_fill_super**, qui fait le gros du travail d'initialisation. Cette fonction recoit en arguments un pointeur vers la structure superbloc à remplir, et la chaine de caractères passée commme argument "options" à l'appel système **sys_mount**.

Elle commence par analyser la chaine d'options pour remplir deux structures **yakafs_mntparams**, une pour chaque système de fichier subordonné. Si il manque des paramètres pour l'un des systèmes, la fonction renvoie un code d'erreur et le montage est annulé.

Si tout se passe bien, les deux systèmes sont montés sur des points de montage virtuels le cas échéant, suivant les options. Si l'un des montages échoue, le montage est également annulé. Des références vers les points de montages virtuels sont enregistrées dans le superbloc de notre système.

Une structure **yakafs_inode** est allouée et initialisée, avec une référence vers l'inode de la racine de chacun des systèmes de fichiers subordonnés, que l'on récupère grace au fait qu'après le montage virtuel, on possède une référence vers chaque **superbloc** subordonné qui contient

une référence vers la **dentry** racine qui contient elle-même une référence vers l'**inode** qui nous intéresse.

Enfin une **dentry** est allouée et initialisée, avec une référence vers les deux dentry racines subordonnées (obtenues de la même manière). Une référence vers celle-ci est ajoutée dans le superbloc en tant que racine du système de fichier *virtuel*.

4.3 Problème

Je me suis rendu compte de l'inutilisabilité de mon travail en regardant le code d'allocation des inodes.

Le problème vient du fait que chaque superbloc définit une méthode **alloc_inode** qui alloue l'espace mémoire pour une inode. Les systèmes de fichiers utilisent cette méthode pour définir une *superstructure* inode spécifique à leur système, de manière à pouvoir stocker toute sorte d'informations.

Cependant cette fonction est seulement utilisée pour allouer l'espace mémoire, et la fonction qui l'appelle fait ensuite d'autres choses pour initialiser cet espace. Le problème est que plusieurs fonctions appellent celle-ci, et font ensuite différente choses avec l'inode qu'on leur renvoie. Or le principe sous-jacent à mon implémentation est de répercuter tout ce que l'on fait à mes structures virtuelles sur les structures subordonnées ; et là je ne sais pas ce que font ces fonctions.

Pour que cela marche ainsi, il faudrait certainement patcher tout le code du VFS, et ce de manière non triviale.

L'implémentation en utilisant une arborescence parallèle n'est donc pas réaliste.

Chapitre 5

Pistes à explorer

5.1 Modèle

En fait plutot que de maintenir pour chaque structure un couple de référence équivalent *local* - équivalent *remote*, chaque structure disposant de son propre lot de fonctions, il faudrait plutot utiliser une seule structure (*virtuelle*), mais jouer sur les pointeurs de fonctions : on pourrait définir les fonctions virtuelles, qui effectueraient le travail de cache et de lecture/écriture, tout en gardant une référence vers chaque fonction des systèmes de fichiers originels, et en appelant celles-ci avec la structure unique.

Il faudrait cependant prendre garde au fait que certaines fonctions puissent modifier les pointeurs de fonctions de leur structure (ou d'un autre), et inverser ces modifications sous peine de voir tout le système échouer, et je ne sais pas si ceci est faisable.

De plus si l'on souhaite faire une interface générique sans entrer dans le détail des systèmes de fichiers sous-jacents, il faut savoir que certains systèmes sont particuliers, par exemple les systèmes liés à Windows ne font pas la différence entre minuscules et majuscules, ce qui n'est pas le cas pour un système Unix. Dans ce cas il faudrait éviter des erreurs liées à ces phénomènes (par exemple si le *remote* est sensible à la casse alors que le *local* ne l'est pas, une erreur surviendra si deux fichiers diffèrent seulement par la casse sur le *remote* ; et à l'opposé si le *local* est insensible à la casse alors que le remote l'est, il faudra veiller à ce qu'un programme qui essaye d'accéder à un fichier qui existe mais qui n'a pas la même casse que celui demandé lui soit présenté, ce qui est le comportement correct).

D'autres écueils de ce type existent certainement, c'est pourquoi ce projet est quelquechose d'assez compliqué.

Mais comme tout sous linux, il est réalisable, à condition d'avoir le temps et les compétences nécessaire, sans parler des techniques de test et de debuggage.