Alink

EPITA - SRS

Sébastien Hénaff
David Brochoire
Nicolas Vigier

Conception et implémentation des liens actifs pour le noyau Linux 2.6.4. Un lien actif (alink), est une extension proposée d'UNIX. Il s'agit d'un nouveau type de fichier, inspiré des liens symboliques.

Mai 2004

Sommaire

Introduction **5**

1 Description du projet **6**
1.1 Spécifications 6
1.1.1 Les alink 6
1.1.2 Les programmes d'indirection 6
1.2 Exemple 7
1.3 Environnement choisi 8

2 Gestion de projet **9**
2.1 Organisation de l'équipe 9
2.2 Méthodes et outils de travail 9
2.2.1 Procédures de communications 9
2.2.2 Outils de développement 10
2.2.3 Système de tests 11
2.3 Planning 13

3 Étude et conception **14**
3.1 Tour d'horizon du noyau 14
3.1.1 L'arborescence 14
3.1.2 Les bibliothèques de fonctions 15
3.1.3 Contexte d'exécution 16
3.1.4 Les appels système 17
3.1.5 Le système kbuild 17
3.1.6 Débogage 18
3.2 Le système de fichiers 18
3.2.1 Les inodes 19
3.2.2 Les liens 19
3.2.3 Virtual File System 20
3.2.4 Ext2fs 21
3.3 Conception 22
3.3.1 Création des liens actifs 22

3.3.2 Résolution des liens actifs 22
3.3.3 Programme d'indirection 24

4 Développement **26**
4.1 Modifications du noyau 26
4.1.1 Ajout des options 26
4.1.2 La création des liens actifs 27
4.1.3 La résolution des liens actifs 33
4.1.4 La conservation des liens actifs 37
4.2 Modifications de la bibliothèque C 38
4.3 Modifications des applications utilisateurs 38
4.3.1 coreutils 38
4.3.2 find 40
4.4 La compilation 41

Conclusion **42**

Introduction

Dans le cadre de nos études de spécialisation SRS (Systèmes Réseaux Sécurité) à l'EPITA (Ecole Pour l'Informatique et les Techniques Avancées), nous devions réaliser un projet de modifications d'un noyau de type UNIX. La durée du travail s'étalait sur environ un mois et demi, mais en raison d'une période relativement chargée, la conception et la réalisation s'est réellement effectuées pendant les trois dernières semaines précédant la date du rendu des livrables, le 29 mai 2004.

Chapitre 1

Description du projet

1.1 Spécifications

Un alink (active link), lien actif en français, est une extension proposée d'UNIX. Il s'agit d'un nouveau type de fichier, inspiré des liens symboliques (symlinks).

1.1.1 Les alink

Quand le système résout un symlink, il lit le contenu du fichier, et ce contenu est le résultat souhaité. Quand le système résout un alink, il lit le contenu du fichier ; ce contenu est le chemin d'accès à un programme exécutable, appelé programme d'indirection, situé dans le filesystem. Ce programme est exécuté, et le résultat souhaité est ce qu'il imprime lors de son exécution.

La structure d'un alink est très semblable à celle d'un symlink ; la différence essentielle est que la donnée contenue dans le lien n'est pas le nom de fichier pointé, mais le nom d'un programme qui, exécuté, renverra le nom de fichier pointé. Le chemin d'accès au programme peut être absolu ou relatif à l'emplacement du alink ; de même, le retour du programme peut être absolu ou relatif à l'emplacement du alink.

1.1.2 Les programmes d'indirection

Un processus d'indirection, instance d'exécution d'un programme d'indirection, doit être considéré comme un fils du programme ayant accédé au alink. Il hérite donc de tout son environnement d'exécution, à deux exceptions près :

- Le descripteur de fichier 0 est fermé
- Le descripteur de fichier 1 n'est pas le même que celui du processus appelant. C'est un descripteur ouvert en écriture, sur lequel le programme

d'indirection écrit le résultat de la résolution du alink. Ce résultat sera utilisé verbatim par le noyau.

Le programme d'indirection prend un argument sur la ligne de commande, qui est le nom de l'alink qui a causé son exécution. Son code de retour doit être 0 pour que l'appel système nécessitant une résolution d'alink retourne avec succès. Tout autre code de retour provoquera un échec de l'appel système ; la variable errno devra alors avoir une valeur significative, justifiée, et conforme à la spécification de l'appel système.

À l'inverse d'un véritable fils du processus appelant, un processus d'indirection doit être parfaitement invisible pour le processus appelant. En particulier, sa mort ne doit jamais provoquer de SIGCHLD, et le PID du processus d'indirection ne doit jamais être renvoyé si le processus appelant effectue un appel système de type wait.

1.2 Exemple

```
$ id
uid=588(ska), gid=100(users), groups=100(users)
$ pwd
/home/ska
$ cat mytmp.c
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>

int main ()
{
  struct passwd *pw = getpwuid(getuid()) ;
  if (pw == NULL)
  {
    perror("mytmp: unable to find user name") ;
    if (errno == 0) errno = ENOENT ;
    return errno ;
   /* assuming the kernel interprets the exit code as
   the right errno to provide */
  }
```

```
  printf("/tmp/%s", pw->pw_name) ;
  return 0 ;
}
$ gcc -o bin/mytmp mytmp.c
$ minln -a bin/mytmp kikoo
$ ls -l kikoo
arwxrwxrwx      1 ska   users   9   Mar 16 15:00        kikoo -> bin/mytmp
$ cd kikoo
cd: no such file or directory: kikoo
$ mkdir /tmp/ska
$ cd kikoo
$ pwd
/tmp/ska
```

1.3 Environnement choisi

Nous avons choisi de modifier un système Linux et plus particulièrement la version 2.6.4 (www.kernel.org) du noyau. Celle-ci offre suffisamment de stabilité pour nous permettre de travailler et la branche 2.6 offre d'autres avantages comme un système de compilation plus simple que ses prédécécesseurs ainsi que le support de l'architecture UML.

Ce choix est également motivé par nos préférences personelles pour ce système par rapport à ses concurrents BSD par exemple.

Les autres programmes que nous avons été amené à modifier sont listés ci-dessous. Il s'agit pour la plupart des versions standards excepté pour la bibliothèque C, où nous avons préféré se concentrer sur une version plus légère que celle developpé par GNU.

- dietlibc (http://www.fefe.de/dietlibc/)
- coreutils (http://www.gnu.org/software/coreutils/)
- find (http://ftp.gnu.org/gnu/findutils/)

Chapitre 2

Gestion de projet

2.1 Organisation de l'équipe

- Sébastien Hénaff (chef de projet)
- David Brochoire
- Nicolas Vigier

2.2 Méthodes et outils de travail

Afin de ne pas se trouver limité ou retardé par des insuffisance ou des incertitudes, un certains nombres de procédures classiques ont été mises en place dès le début du projet.

Chacun des services nécessaires sont volontairement hébergés sur des domaines différents afin de limiter les risques de paralysie globale en cas d'indisponibilité.

2.2.1 Procédures de communications

Liste de diffusion

Plusieurs méthodes de communications ont été défini afin d'assurer l'échange des informations. La première et principale est une liste de diffusion (mailing list) dédié au projet. Celle-ci est gérée par le logiciel libre "mailman" (http://www.list.org/) et est administrée par nos soins sur le nom de domaine "biais.org".

D'un point de vue hiérarchique, c'est ce moyen de communication qui fait foi et les comptes rendu des réunions ne sont considérés valides qu'une fois postés sur la liste.

Wiki

Tasks (http://www.alexking.org) est un wiki (page web modifiable) orienté gestion des tâches. Il permet très simplement d'ajouter des tâches et des sous-tâches avec une durée déterminée et permet par la suite de définir le niveau d'avancement de celles-ci.

Sa mise en oeuvre est simple, il s'installe sur un serveur web supportant le "PHP" ainsi qu'une base de donnée "MySQL". Tous ceci est administrés par nos compétences sur le domaine "gouarec.com".

Les caractéristiques de tasks définies par son auteur :

- Hiérarchie dynamique de la vue des tâches sous la forme d'un arbre.
- Dates d'établissement du programme et association d'URLs aux tâches.
- Rappels quotidiens par e-mail.
- Représentation des taches sur un calendrier (iCalendar).
- Fonctionnalités de recherche avancée.

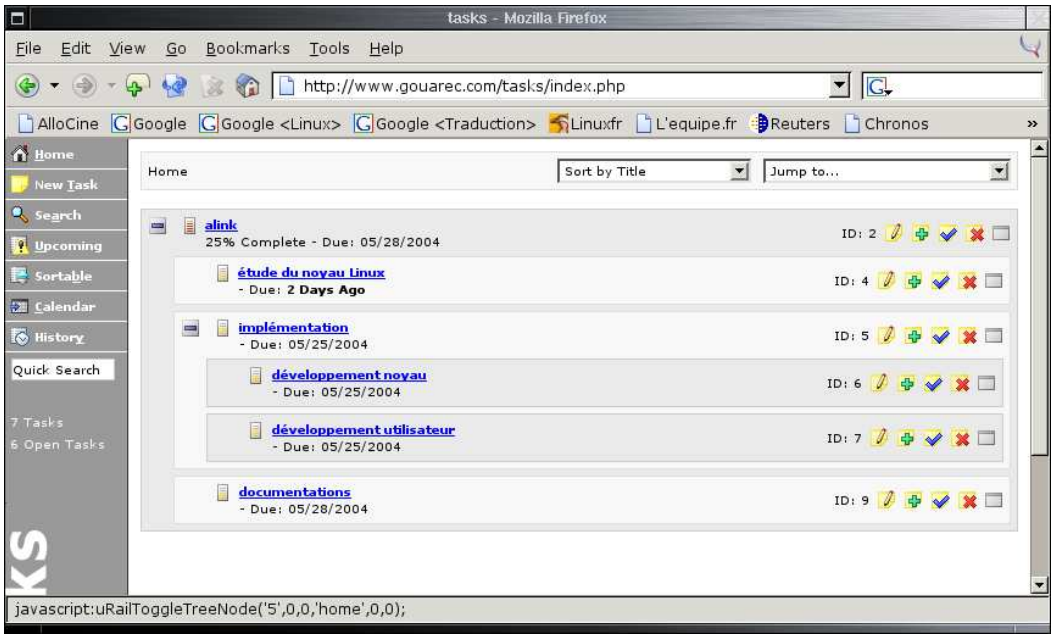

FIG. 2.1 – Tasks

2.2.2 Outils de développement

CVS

CVS, (Concurrent Version System) est un outil permettant de gérer l'évolution dans le temps d'un ensemble de fichiers . CVS permet d'extraire des

sources, de les modifier, de soumettre ses modifications, de garder l'historique des modifications et de restaurer n'importe quelle version précédente. Mais l'atout majeur de CVS est de permettre à plusieurs développeurs de travailler sur le même groupe de fichiers : chacun travaille de manière indépendante dans son environnement personnel et CVS gère l'ensemble.

Le serveur CVS du projet est hébergé sur le domaine "cvs.zeio.org". Chacun des membres du groupe dispose d'un login et d'un mot de passe pour pouvoir accéder aux sources via SSH.

Compilateur

Le compilateur C utilisé est celui de GNU dans sa version 2.95.

Édition de manuel

Le programme gmanedit (http://gmanedit.sourceforge.net/) a été utilisé pour éditer et modifier les mans.

2.2.3 Système de tests

Pour tester le nouveau noyau, le faire tourner sur une machine peut paraître hasardeux, particulièrement lors de la modifications des systèmes de fichiers. En outre, rebooter systématique à chaque nouveau changement peut vite devenir pénible. C'est pour ces raisons que nous avons choisi d'utiliser une machine virtuelle et plus particulièrement UML (User Mode Linux).

Cependant, l'utilisation d'une machine virtuelle peut présenter quelques désavantages, notamment l'environnement diffère forcément de celui d'une véritable architecture. Cette problématique a été prise en compte, et nous avons régulièrement effectué des tests sur une véritable machine i386 dédié.

UML

UML est un projet libre (http://user-mode-linux.sourceforge.net/) qui permet de faire tourner un Linux dans une machine virtuelle depuis n'importe quelle distribution Linux. Depuis les versions des noyaus 2.6, UML fait même partie intégrante des sources du noyau (arch/um).

La mise en oeuvre d'UML est relativement simple, il faut dans un premier temps récupérer les sources du noyau Linux et son patch UML associé :

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/
```

```
linux-2.6.4.tar.bz2
$ wget http://prdownloads.sourceforge.net/user-mode-linux/
uml-patch-2.6.4-1.bz2
```

Puis créer ou récupérer un système de fichier compatible :

```
$ wget http://prdownloads.sourceforge.net/user-mode-linux/
Debian-3.0r0.ext2.bz2
$ wget http://prdownloads.sourceforge.net/user-mode-linux/
root_fs_slack8.1.bz2
```

Ensuite, il suffit d'appliquer le patch et de configurer le noyau en spécifiant l'architecture UML. Enfin, la compilation proprement dite crée un exécutable **linux**.

```
$ tar -xvjf linux-2.6.4.tar.bz2
$ cd linux-2.6.4
$ bzcat ../uml-patch-2.6.4-1.bz2 | patch -p1
$ make defconfig ARCH=um
$ make menuconfig ARCH=um
$ make linux ARCH=um
```

Une fois l'exécutable créé, il se lance comme n'importe quel binaire du système avec éventuellement des arguments comme le système de fichiers et le réseau.

Pour activer le réseau, il suffit d'ajouter à la ligne de lancement de l'UML : *eth*0 : *tuntap*, , , 192.168.0.2. Il est également nécessaire d'avoir les *uml_ utilities* d'installées et que le binaire *uml_ net* se trouve bien dans le path et ait les bonnes permissions. Le réseau se configure ensuite comme sous n'importe quel Linux. A noter que le système hôte doit posséder le module "tun.o" qui permet la réception et la transmission de paquets pour les programmes en espace utilisateur.

```
$ ./linux root=/dev/ubd/0 ubd0=../Debian-3.0r0.ext2 devfs=mount
eth0=tuntap,,,192.168.0.2

uml:~# uname -a
Linux uml 2.6.4-1um $1 Wed May 5 19:58:57 CEST 2004 i686 unknown
```

Les tests en environnement de production

Afin de s'assurer que les liens actifs fonctionnaient également sur une véritable architecture i386, des tests de compilation et d'installation du nouveau noyau ont été effectué sur une Debian Sarge installée sur un portable Dell Inspiron-4200.

2.3 Planning

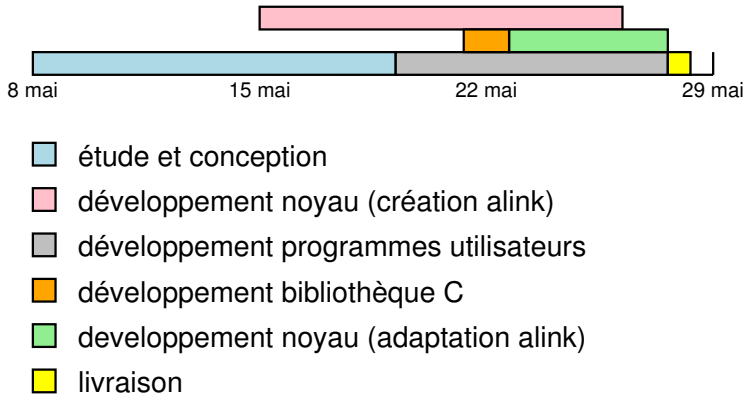

FIG. 2.2 – Planning

Chapitre 3

Étude et conception

3.1 Tour d'horizon du noyau

La programmation noyau est une opération délicate car une petite erreur peut être lourde de conséquences. Dans le mode utilisateur, en cas d'erreur de d'implémentation ou de corruption de mémoire, seuls le processus et son environnement proche peuvent être impactés. Dans le noyau, il n'y a rien pour rattraper les erreurs puisque c'est justement le rôle de celui-ci, ainsi, la machine peut planter, les systèmes de fichiers peuvent être corrompus...

L'impact de chaque manipulation à l'intérieur du noyau doit être maîtrisé et ceci commence par une bonne connaissance du fonctionnement global de celui-ci.

3.1.1 L'arborescence

Une archive standard contient les sources pour une quinzaine d'architectures. Toutes ces architectures partagent une bonne partie du noyau, mais ont également leurs spécificités.

Une fois extraites, on trouve à la racine des sources du noyau quelques fichiers de documentation, le *Makefile* dont les règles sont dans le fichier *Rules.make*, un répertoire *Documentation* et un répertoire *scripts* contenant les divers scripts utilisés pour configurer le noyau (le menu de configuration, le programme *mkdep* pour générer les dépendances, etc...). Enfin, le reste des répertoires contient les sources du noyau à proprement parler.

Le répertoire include/

Ce répertoire contient les différentes en-têtes utilisées par les différentes parties du noyau, ainsi que ceux utilisés par la *libc* qui communique avec le

noyau. Aucun autre programme ne doit les utiliser.

Il y a notamment *linux*, *net*, *pcmcia*, ...Il y aura également *asm* qui n'existe pas encore, mais qui sera un lien symbolique vers le répertoire *asm-** correspondant à l'architecture pour laquelle le noyau sera compilé.

Le répertoire arch/

Il contient, dans des répertoires séparés, toutes les parties du code dépendant de l'architecture. On y trouve en particulier, pour chacune d'entres-elles, le code d'entrée des appels systèmes (*arch/*/kernel/entry.S*), l'implémentation des sémaphores et l'implémentation de la bibliothèque de fonctions (manipulation de chaînes, de blocs mémoire, ...) spécifiques et optimisée pour chaque architecture (*arch/*/lib/*).

Le répertoire init/

Il contient le code utilisé au démarrage du noyau, en particulier le fichier *main.c*, qui est chargé d'analyser les paramètres passés, d'initialiser chaque sous-système, et de créer ex nihilo, le premier processus (le processus 0, aussi connu sous le nom de "**idle task**" puis de "forker" pour exécuter */sbin/init* ou autre chose.

Le répertoire lib/

Ce répertoire contient une implémentation de tous les services de base (manipulations de chaînes, de mémoire, ...), qui seront utilisés quand la version optimisée pour l'architecture n'est pas disponible.

Les répertoires kernel/, fs/, net/, ipc/, mm/, drivers/

Ces répertoires contiennent chaque sous-système du noyau, soit, respectivement, le code principal, celui des systèmes de fichiers, le réseau, la communication inter-processus, la gestion de la mémoire et enfin les différents pilotes de périphériques.

3.1.2 Les bibliothèques de fonctions

Le noyau est totalement autonome, et implémente lui-même les fonctions dont il a besoin. La documentation de l'api est disponible dans le répertoire *Documentation/DocBook/kernel-api.tmpl*.

Ainsi, il n'est pas possible d'utiliser la *libC*, on dispose donc des fonctions répondant à des besoins spécifiques du noyau, comme divers types de sémaphores, la gestion de listes ou d'arbres...

Affichage

Pour imprimer, il existe un petit frère de "printf()", "printk()", qui finissent en général dans le */var/log/kern.log*.

Manipulations mémoire

Pour la manipulation de chaînes ou de blocs mémoire, on trouvera tout ce qui est disponible dans */linux/lib/string.c : strcpy(), strncat()* et autres *bcopy()*.

L'allocation de la mémoire se fait grâce aux fonctions *kmalloc()* et *kfree()*. Il existe d'autres fonctions plus spécialisées, pour allouer juste une page, par exemple.

Appels systèmes

Le noyau met à disposition plus de 200 appels systèmes. Il a le droit de les utiliser, lui aussi. Ainsi, moyennant quelques manipulations, le noyau peut lire des fichiers sur un système de fichiers, ouvrir des "sockets", exécuter des programmes, ...

3.1.3 Contexte d'exécution

Contrairement au mode utilisateur, il existe plusieurs contextes dans le quels du code noyau peut s'exécuter. Selon ce contexte, certaines choses peuvent être faites ou non, certaines hypothèses sont valides ou non.

Contexte utilisateur

Le contexte utilisateur est le contexte dans lequel on se trouve lorsqu'on arrive d'un appel système. Attention à ne pas confondre le mode utilisateur (mode dans lequel se trouve le processeur pour exécuter des processus normaux), l'espace utilisateur (l'environnement d'exécution des applications) et le contexte utilisateur (le contexte d'exécution particulier du noyau où le processeur est en mode noyau).

Dans ce contexte, la macro *"current"* pointe sur la structure *"task_struct"* du processus qui a appelé l'appel système.

Contexte d'interruption matérielle

C'est le contexte des gestionnaires d'interruptions matérielles, les bouts de code qui s'occupent de répondre aux requêtes du matériel, par exemple de lire le code de la touche tapée au clavier, ou de lire le paquet tout juste arrivé sur la carte réseau.

Lorsque le gestionnaire d'une interruption donnée est en cours, les interruptions suivantes sont mises en queue.

Ce type de gestionnaire doit s"exécuter le plus rapidement possible. Ainsi, ils ne s'occupent en général que du minimum (lecture de l'événement) puis le mettent en queue (*task queue immediate*), le traitement proprement dit de cet événement sera effectué dans le contexte d'interruption logicielle.

Contexte d'interruption logicielle

A chaque fois qu'un gestionnaire d'interruption matérielle ou qu'un appel système se termine, tous les traitements mis en queue par les interruptions matérielles sont exécutés.

3.1.4 Les appels système

Un appel système est une fonction du noyau qui peut être appelée par un programme en espace utilisateur. Chaque appel système est identifié par son numéro dans une table.

Un appel système sous Linux se présente sous la forme d'une fonction C classique, dont le nom commence conventionnellement pas *"sys_ "*. Son prototype est précédé du mot clef *asmlinkage*, qui indique au compilateur de ne pas tenter d'optimiser le passage d'arguments et de bien tout prendre dans la pile.

Le code de retour d'un appel système est positif ou nul si tout s'est bien passé. En cas d'erreur, il est négatif et correspond à l'opposé du code *errno*.

3.1.5 Le système kbuild

C'est l'ensemble des fichiers qui permettent à l'utilisateur de configurer et compiler le noyau. Les programmes de configuration vont lire le fichier *config.in* de l'architecture pour laquelle on construit le noyau ainsi que les fichiers inclus. Après configuration, un fichier **.config** est écrit à la racine de l'archive. Il contient toutes les variables correspondant aux options qui ont été activées ainsi que leur valeur : "y" pour une option activée, "m" pour une option à compiler en tant que module, ou encore parfois des valeurs (entiers, chaînes de caractères).

Ces variables sont définies pour être utilisées à la fois dans les Makefiles ($(CONFIG_EXT2)) et comme variables d'environnement et comme macros dans les sources (#ifdef CONFIG_EXT2).

3.1.6 Débogage

Il est nettement plus difficile de déboguer un noyau que du code en espace utilisateur. Voici quelques techniques que nous avons mis en oeuvre.

Utilisation de printk

La toute première et la plus simple des méthodes. Il s'agit d'utiliser la fonction *printk()* au fonctionnement analogue à *printf()*. Toute chaîne sortie par *printk()* est censée être préfixée par une valeur de priorité. Les huit macros définies pour cela vont de KERN_EMERG à KERN_DEBUG, et sont définies dans "include/linux/kernel.h".

Par exemple :

```
printk(KERN_DEBUG, "alink = %s\n", alink);
```

On peut également, comme dans tout programme C, (mais c'est surtout dans ce contexte que l'on se rend compte de tout son intérêt), enrichir la sortie à l'aide des chaînes prédéfinies de gcc comme __FILE__, __FUNCTION__ ou __LINE__.

Utilisation de gdb

Puisque UML représente un noyau Linux et qu'il se comporte comme un simple programme, on va pouvoir déboguer ce noyau comme un programme habituel.

Pour cela, il suffit d'ajouter au lancement d'UML l'option **debug**, ce qui permet d'avoir un accès à un terminal contenant gdb, qui pourra suivre le fonctionnement du noyau virtuel.

```
# ./linux root=/dev/ubd/0 ubd0=../uml/Debian-3.0r0.ext2
devfs=mount debug
```

3.2 Le système de fichiers

Le système de fichier est le mode de représentation et d'agencement des fichiers dans le disque dur. Une des bases du fonctionnement d'un système

Unix est que tout dans l'ordinateur est associé à un fichier.

3.2.1 Les inodes

La représentation interne d'un fichier est assurée par un inode (index node), stocké dans une structure sur le système de fichier. Cette structure contient des informations sur le fichier, tel que les droits d'accès, disposition sur le disque du fichier, etc...

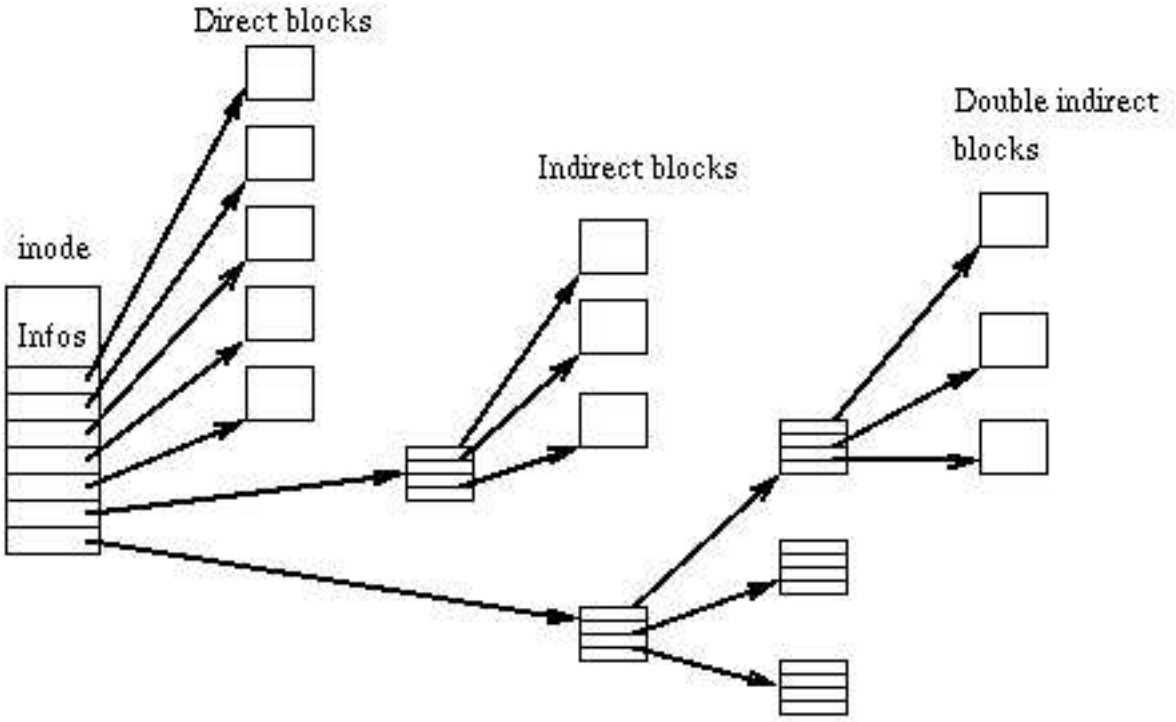

FIG. 3.1 – Structure d'un inode

3.2.2 Les liens

Historiquement, sous les systèmes UNIX, à un fichier peut correspondre plusieurs noms, chaque nom étant en relation avec un inode. Ces noms sont appelés des liens. Lorsqu'un programme fait appelle à un fichier par son lien, le noyau extrait l'inode (résout le lien) et vérifie les autorisations d'accès à celui-ci avant de permettre toutes interactions.

Il existe deux types de liens :

- Lien physique
- Lien symbolique

L'objectif du projet est d'ajouter au noyau Linux un troisième type de lien, les liens actifs.

Liens physiques

Un fichier est référencé par une unique inode dans un système de fichier. Mais plusieurs entrées peuvent référencer cette inode. Chaque entrée dans un répertoire crée un lien physique (lien hard) d'un nom de fichier vers cette inode.

Lorsque tous les liens vers ce fichier sont supprimés alors l'inode est supprimée. Si un lien vers un fichier est effacé et que le nom du fichier est recréé avec un nouveau contenu, les autres liens pointeront vers l'ancienne inode et donc l'ancien contenu. Un lien physique est forcément dans le même système de fichier et aussi dans la même partition.

Un lien hard n'a pas d'inode propre, il a l'inode du fichier vers lequel il pointe.

Liens symboliques

Un lien symbolique par contre est un lien sur un autre chemin, ce qui permet donc de passer à travers différentes partitions et systèmes de fichier. Ils peuvent pointer sur des répertoires, contrairement aux liens physiques. De plus, un lien symbolique possède son propre inode.

Il sera traité de manière particulière dans le noyau à l'aide du flag S_IFLNK. Ce traitement particulier sera entre autre «readlink», permettant la résolution du lien, lors de toutes actions sur ce lien, tel que «open», «stat» et «lstat»...

3.2.3 Virtual File System

Le noyau Linux contient un système de fichier virtuel (VFS) qui est utilisé lors des appels systèmes concernant les fichiers. Il s'agit en fait d'une couche d'abstraction qui appelle les fonctions nécessaires sur le système de fichier physique (Ext2fs, Reiserfs, ...).

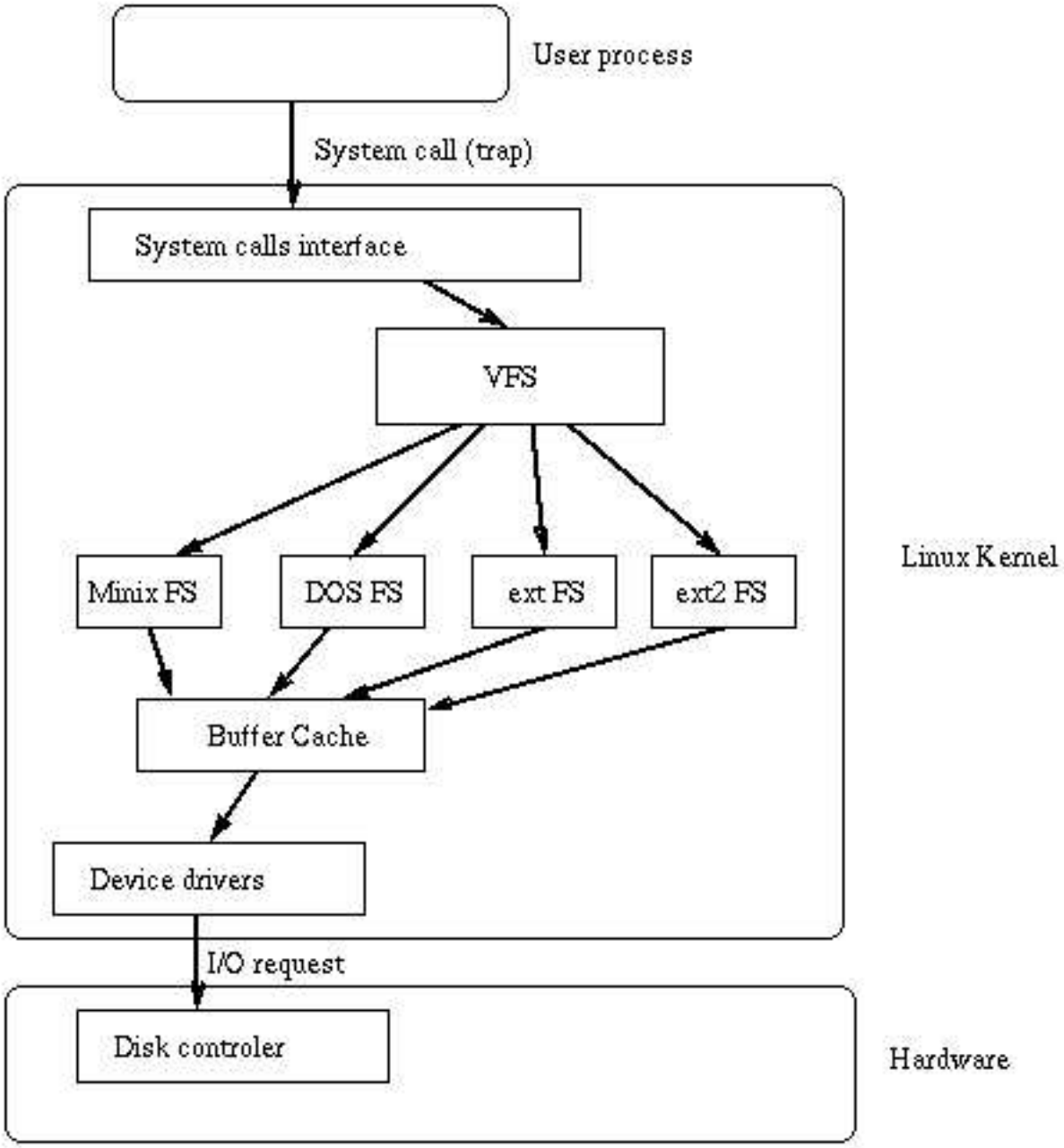

FIG. 3.2 – VFS

3.2.4 Ext2fs

Le système de fichier Ext2fs supporte les fichiers standards (fichiers réguliers, répertoires et liens). Il est relativement simple, accepte des partitions

jusqu'à 4 To et des noms de fichiers de 255 caractères.

Sa structure est définie dans le fichier **/usr/include/linux/ext_2fs.h**.

3.3 Conception

La difficulté du projet réside en grande partie dans la quantité astronomique des sources. Environ 200 Mo pour le noyau et plus de 50 Mo pour la bibliothèque C et les différents programmes utilisateurs.

Se lancer dans le développement sans un minimum de réflexion et de méthode n'aurait aucune chance d'aboutir. C'est pourquoi, avec les conseils de Marc Espie, nous avons réalisé une étude préalable sur les moyens de parvenir intelligemment à modifier le noyau.

3.3.1 Création des liens actifs

Les liens actifs présentent beaucoup de similitudes avec les liens symboliques. En réalité, seule la résolution est différente. Le premier travail consiste donc à dupliquer les liens symboliques afin de créer un troisième type de liens qui aurait dans un premier temps exactement les mêmes fonctionnalités que ces derniers.

Ce premier travail, bien que simple en apparence, est assez fastidieux. Il faut s'appliquer pour ne rien "oublier" dans la duplication et cependant il faut également faire attention à ne pas renommer des fonctions qui n'ont pas besoin de l'être.

Des outils très pratique pour ce type de travail sont regroupés dans le package "id-utils". Ceux-ci permettent de référencer toutes les occurrences d'un identifiant dans des sources écrites en C. Ainsi nos recherches et duplications se sont orientés sur les appels systèmes "symlink" (création d'un lien symbolique) dupliqué en "alink" et "readlink" (résolution d'un lien symbolique) dupliqué en "readalink".

3.3.2 Résolution des liens actifs

La sémantique des appels systèmes *stat*, *lstat* et *astat* sur les liens actifs suit les résolutions suivantes :

Fichier régulier :

- stat() → statut du fichier.
- lstat() → statut du fichier.

- astat() → statut du fichier.

Lien symbolique

- stat() → statut du fichier pointé par le lien symbolique.
- lstat() → statut du lien symbolique.
- astat() → statut du lien symbolique.

Lien actif

- stat() → statut du fichier pointé par l'exécution du programme d'indirection.
- lstat() → statut du programme d'indirection pointé par le lien actif.
- astat() → statut du lien actif.

Étant donné la forte ressemblance entre les liens symboliques et les liens actifs dans la façon d'être stockés dans le système de fichier, la méthode de résolution d'alink est elle aussi très proche. Elle se découpe en deux étapes successives. La première consiste à traiter le lien actif comme un lien symbolique afin d'obtenir le chemin du programme d'indirection. La deuxième étape est constituée de l'exécution de ce programme et de la récupération du nouveau chemin donné.

Pour réaliser la première résolution, nous avons tout simplement dupliqué celle des liens symboliques en prenant soin de renommer les fonctions et constantes de façon cohérente (ex : S_IFLNK en S_IFALNK). À partir de là nous devions donc :

- lancer un nouveau processus
- lire la sortie du processus fils

Techniquement cela s'est traduit par l'utilisation de différentes fonctions et appels systèmes servant chacun à réaliser une tâche précise :

- kernel_thread
  créé un nouveau thread dans le kernel qui se met à exécuter une fonction passée en paramètre. Nous l'avons utilisé pour initialiser le lancement d'un processus fils
- do_pipe
  créé un pipe, tout comme la fonction pipe de la libc. Nous l'avons par la suite partagé entre le processus fils et père pour récupérer la sortie du programme d'indirection (le fils)

- set_fs
  change le mode d'adressage pour que les appels systèmes acceptent des pointeurs de l'espace kernel, au lieu d'accepter uniquement ceux de l'espace d'adressage utilisateur. Cette fonction a été utilisée quand nous avons dû allouer de la mémoire à partir du kernel
- sys_close
  ferme un descripteur de fichier, tout comme la fonction close de la libc. Et ceci nous a permit de nettoyer les descripteurs de fichiers ouvert à l'aide du pipe
- sys_read
  lit des données à partir d'un descripteur de fichier, et les met dans un buffer. Dans notre cas, le descripteur de fichier était le pipe
- sys_wait4
  bloque le processus courant pour attendre la fin de l'exécution d'un processus fils précisé en argument. Ce qui a permit de récupérer dans le père le code de retour du fils
- sys_dup2
  redéfini un descripteur de fichier en un autre. Son utilisation courante est de *dérouter* la sortie standard d'un processus vers un autre descripteur de fichier (comme un pipe)
- do_execve
  remplace le thread kernel courant par l'exécution d'un programme passé en argument. C'est donc à cette fonction que nous avons demandé de lancer le programme d'indirection

3.3.3 Programme d'indirection

À plusieurs reprises, les explications nous ont permit de comprendre que le système des liens actifs nécessite l'exécution d'un programme. L'une des contraintes de ce sujet spécifiait que le processus père demandant une résolution d'un alink (par un appel à readalink par ex) ne devait pas voir l'existence de ce fils.

Le comportement par défaut du noyau linux est d'envoyer le signal SIGCHLD au père, lorsque l'un de ses fils meurt. Il fallait donc supprimer cet envoi du signal SIGCHLD qui était une façon de percevoir l'existence d'un fils.

Une deuxième contrainte du sujet était de considérer que la sortie standard du processus d'indirection était valide, si et seulement si ce processus avait un exit status à 0.

Or lors de nos recherches, nous avons trouvé une première solution répondant à la première contrainte mais ne remplissant pas la deuxième, et inversement pour une autre solution existante. En effet, pour demander à linux de

trouver l'exit status du fils, nous devons nous mettre en attente de la fin de son exécution (cf sys_wait4). Or si nous demandons au kernel de ne pas envoyer de SIGCHLD au père quand le fils meurt, le père devient incapable de se mettre en attente de la fin d'exécution du fils.

Étant donné que nous n'avons pas réussi à trouver une solution répondant aux deux contraintes, nous avons du faire un choix sur celle à remplir, et nous avons décidé de récupérer l'exit status du fils.

Chapitre 4

Développement

L'ajout des "liens actifs" dans le noyau nécessite trois étapes de développement distinctes :

- L'ajout des appels systèmes dans les sources du noyau.
- L'ajout de l'interface de ces appels systèmes dans la bibliothèque C.
- Le développement des programmes en espace utilisateur.

4.1 Modifications du noyau

4.1.1 Ajout des options

De façon à pouvoir facilement activer ou désactiver les alink du noyau que nous allons compiler, nous avons ajouté des options de compilation. Cela se fait en éditant le fichier Kconfig du répertoire concerné.

L'option ALINK doit être activée pour bénéficier du support des alink. Pour utiliser les alink sur un système de fichiers ext2 (le seul pour lequel nous proposons les alink actuellement), l'option EXT2_FS_ALINK doit être activée.

include/linux/

- **Kconfig :**

```
config ALINK
bool "ALink support"
help
  Enable alink support for fs that implement it (currently ext2).
  http://skarnet.org/epita/srs2005/alink.html
```

```
config EXT2_FS_ALINK
bool "Ext2 ALink support"
depends on EXT2_FS && ALINK
help
  Allow ALinks to be created.
```

Dans le code, on entoure de #ifdef CONFIG_ALINK les parties devant être compilées quand l'option ALINK est activée :

```
#ifdef CONFIG_ALINK
  /* code à exécuter quand l'option a été activée */
#else
  /* code à exécuter quand l'option n'a pas été activée */
#endif /* CONFIG_ALINK */
```

4.1.2 La création des liens actifs

À ce niveau, les modifications permettent de créer un nouveau type de fichier (S_IFALNK) qui possédera son inode propre mais qui aura un comportement identique à celui des liens symboliques.

include/linux/

- **stat.h :**

Définition du type de fichier "lien actif".

```
#define S_IFMT  00170000
#define S_IFLNK  0120000     /* symlink */
#ifdef CONFIG_ALINK
# define S_IFALNK 0110000     /* alink */
#endif


#define S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)
#ifdef CONFIG_ALINK
# define S_ISALNK(m)    (((m) & S_IFMT) == S_IFALNK)
#endif
```

- **fs.h :**

Ajout des nouvelles opérations possibles sur les inodes.

```
struct inode_operations {
  int (*symlink) (struct inode *,struct dentry *,const char *);

#ifdef CONFIG_ALINK
  int (*alink) (struct inode *,struct dentry *,const char *);
  int (*readalink) (struct dentry *, char __user *,int);
  int (*follow_alink) (struct dentry *dentry, struct nameidata *nd);
#endif
  ....
}
```

Externalisation des prototypes.

```
#ifdef CONFIG_ALINK
 extern int vfs_alink(struct inode *, struct dentry *, const char *);
 extern int vfs_readalink(struct dentry *, char __user *, int, const char *);
 extern int page_readalink(struct dentry *, char __user *, int);
 extern int page_follow_alink(struct dentry *, struct nameidata *);
 extern int page_alink(struct inode *inode, const char *symname, int len);
 extern struct inode_operations page_alink_inode_operations;
#endif
```

- **syscall.h :**

Définition des prototypes des nouveaux appels systèmes.

```
 asmlinkage long sys_readalink(const char __user *path,
                               char __user *buf, int bufsiz);
 asmlinkage long sys_alink(const char *old, const char *new);
 asmlinkage long sys_astat(char __user *filename,
                           struct __old_kernel_stat __user *statbuf);
```

include/asm-i386/

- **unistd.h :**

Ce fichier contient les numéros de tous les appels systèmes.

```
#ifdef CONFIG_ALINK
# define __NR_readalink          274
# define __NR_alink              275
# define __NR_astat              276

# define NR_syscalls 277
```

```
#elif

# define NR_syscalls 274
#endif
```

arch/i386/kernel/

- **entry.S :**

Ce fichier contient les routines bas niveau des appels systèmes.

```
.data
ENTRY(sys_call_table)
     ....
     .long sys_readalink     /* 274 */
     .long sys_alink         /* 275 */
     .long sys_astat         /* 276 */
```

fs/

- **stat.c :**

Implémentation au niveau du VFS (Virtual File System) de l'appel système "astat".

```
/*
 * vfs_astat effectue une résolution du nom passé en
 * paramètre (en utilisant la fonction link_path_walk)
 * et récupère les informations sur le fichier pour
 * remplir la structure de type ``struct kstat''
 */
int vfs_astat(char __user *name, struct kstat *stat)
{ .... }

/*
 * L'appel système astat fait en fait appel à la fonction
 * vfs_astat avant de recopier le résultat dans le buffer en
 * mémoire utilisateur.
 *
 * Le comportement de astat est identique à lstat sur les liens
 * et fichiers non alink. Sa particularité est de ne pas suivre
 * les liens alink.
 * D'après nos tests le comportement de stat sur les fichiers
 * alink est correcte. lstat par contre n'essaie pas du tout de
```

```
 * résoudre les alink alors que le comportement correct qui
 * nous a été demandé veut que lstat s'exécute sur le programme
 * d'indirection.
 * Un comportement correcte demande quelques modifications de la
 * fonction link_path_walk du fichier namei.c, ce que nous
 * n'avons pas eu le temps de faire (et pas osé modifier au
 * dernier moment de peur de tout casser, cette fonction étant
 * assez complexe).
 */
asmlinkage long sys_astat(char __user * filename,
                          struct __old_kernel_stat __user * statbuf)
{ .... }

/*
 * L'appel système readalink récupère l'inode du fichier
 * passé en paramètre, et si c'est bien un fichier de
 * type alink (si l'opération readalink existe sur cet
 * inode), alors il exécute l'opération readalink sur
 * cet inode (la fonction namei.c:page_alink est alors
 * appelée).
 */
asmlinkage long sys_readalink(const char __user * path,
                              char __user * buf, int bufsiz)
{ .... }


EXPORT_SYMBOL(vfs_astat);
```

- **namei.c :**

Implémentation au niveau du VFS (Virtual File System) des appels systèmes “alink” et “readalink”.

```
/*
 * La fonction exécute_alink prend en paramètre un fichier
 * qu'elle va exécuter pour nous donner dans un buffer le
 * résultat de sa sortie standard. Le fonctionnement de
 * cette fonction est décrit plus en détails dans le chapitre
 * ''La résolution des liens actifs''.
 * Cette fonction est appelée dans vfs_readalink et
 * page_follow_alink.
 */
int execute_alink(struct s_readalink *li, char *buf, int buflen)
```

```
{ .... }

 /*
  * La fonction page_readalink est la fonction qui est appelée
  * sur un inode par l'appel système readalink pour récupérer
  * la chaîne de caractères renvoyée par le programme
  * d'indirection sur lequel pointe le alink.
  * Cette fonction récupère le nom du programme d'indirection
  * sur lequel le alink pointe (à l'aide de la fonction
  * page_getlink) avant de le passer en paramètre à la
  * fonction vfs_readalink qui va se charger d'exécuter le
  * programme d'indirection pour récupérer le résultat qu'elle
  * va recopier dans un buffer en mémoire utilisateur (à l'aide
  * de la fonction copy_to_user).
  */
int page_readalink(struct dentry *dentry, char __user *buffer,
   int buflen, const char __user *path)
{ .... }

 /*
  * La fonction vfs_readalink alloue un buffer en mémoire noyau
  * d'une taille identique à celle du buffer en mémoire utilisateur
  * puis appelle la fonction execute_alink pour récupérer dans ce
  * buffer le résultat de l'exécution du programme d'indirection.
  * Le buffer en mémoire noyau est ensuite recopié dans le buffer
  * en mémoire utilisateur avant d'être libérée.
  */
int vfs_readalink(struct dentry *dentry, char __user *buffer, int buflen,
  const char *link, const char __user *path)
{ .... }

 /*
  * La fonction link_path_walk est une fonction de résolution
  * permettant de récupérer une structure dentry on lui
  * fournissant un nom de fichier et quelques flags pour lui
  * demander ou non de résoudre les link et alink finaux
  * (les appels systèmes stat, astat et lstat appellent tous
  * les trois cette même fonction en fournissant des flags
  * différents).
  * Nous avons eu à rajouter dans cette fonction quelques
  * appellent à la fonction do_follow_alink permettant de
  * résoudre un alink.
```

```
  */
int fastcall link_path_walk(const char * name, struct nameidata *nd)
{ .... }

 /*
  * La fonction do_follow_link est chargée de résoudre un alink.
  * Pour cela elle commence par incrémenter quelques compteur
  * permettant de faire des testes pour éviter les boucles
  * qui pourraient être crées par des link ou alink qui pointent
  * sur d'autres link ou alink formant une boucle.
  * Elle execute ensuite l'opération follow_alink de l'inode,
  * qui est en fait un appel à la fonction page_follow_alink.
  */
static inline int do_follow_link(struct dentry *dentry, struct nameidata *nd)
{ .... }

 /*
  * La fonction page_follow_alink récupère le nom du programme
  * d'indirection à l'aide de la fonction page_getlink, alloue
  * un buffer et appelle execute_alink sur ce buffer et le
  * nom du programme d'indirection.
  */
int page_follow_alink(struct dentry *dentry, struct nameidata *nd)
{ .... }

 /*
  * Cet appel système permet de créer un alink (fonctionnement
  * identique à l'appel système de création de lien.
  */
asmlinkage long sys_alink(const char __user * oldname,
                          const char __user * newname)
{ .... }

EXPORT_SYMBOL(vfs_alink);
EXPORT_SYMBOL(vfs_readalink);
EXPORT_SYMBOL(page_alink);
EXPORT_SYMBOL(page_follow_alink);
EXPORT_SYMBOL(page_readalink);
```

fs/ext2/

Implémentation physique des inodes "alink". A noter que le système de fichier ext2 différencie deux types de liens symboliques (fast and slow symlink) pour des raisons d'optimisation. Pour plus de simplicité et de clarté, les liens actifs s'identifieront aux comportements des "slow symlink", qui sont les plus génériques.

- **namei.c :**

Création d'un inode "alink".

```
#ifdef CONFIG_EXT2_FS_ALINK
static int ext2_alink (struct inode * dir, struct dentry * dentry,
        const char * symname)
{
  ....
  inode = ext2_new_inode (dir, S_IFALNK | S_IRWXUGO);
  ....
}
#endif
```

4.1.3 La résolution des liens actifs

Au niveau de la résolution des liens actifs, nous avons deux points d'entrées différents dans le kernel. Le plus simple à visualiser est l'appel système readalink. Dans cette fonction, le buffer pour stocker le résultat de l'indirection est fourni par l'utilisateur, et le nombre d'indirection à calculer est toujours de un (si le résultat de l'indirection est le chemin d'un lien actif, nous ne ferons pas de nouvelle résolution).

Le deuxième point d'entrée est utilisé en interne par le kernel, lorsque celui-ci cherche à résoudre un chemin relatif vers un chemin absolue. Dans ce cas la résolution est récursive, et le buffer pour stocker le résultat est fourni par le kernel.

Ces deux points d'entrée ont du code en commun constitué du couple pipe/fork. C'est pourquoi nous avons placé cette partie dans une fonction commune aux deux points d'entrées.

fs/

- **namei.c :**

```
/*
** fonction partagée par les deux systèmes de résolution
```

```
*/
int execute_alink(struct s_readalink *li, char *buf, int buflen)
{
  char c = ' ';
  mm_segment_t old_fs;
  int len, pid, error, pid_waited, exit_status;

  error = do_pipe(li->pipe);
  if (error < 0) {
    return error;
  }

  /*
  ** nous lançons un nouveau thread kernel qui exécutera la fonction
  ** passée en premier argument
  */
  error = (pid = kernel_thread(fn_readalink, (void *)li, SIGCHLD));

  old_fs = get_fs();
  set_fs(KERNEL_DS);

  if (pid <= 0)
    goto out_error;
  if ((error = sys_close(li->pipe[1])) != 0)
    goto out_error;

  /*
  ** nous recopions la sortie standard du thread kernel à partir
  ** du pipe dans le buffer prévu à cet effet
  */
  error = 0;
  for (len = 0; (len < buflen) && (c != '\0'); ++len) {
    if ((error = sys_read(li->pipe[0], &c, 1)) > 0) {
      buf[len] = c;
    }
    else {
      if (error == -EINTR) {
        --len;
      }
      else {
        if (error < 0)
          goto out_error;
```

```
        break ;
      }
    }
  }

  /*
  ** nous attendons la fin de l'exécution du fils
  */
  pid_waited = -EINTR;
  while ((pid_waited == -EINTR) ||
         ((pid_waited > 0) && (pid_waited != pid))) {
    pid_waited = sys_wait4(pid, &exit_status, 0, NULL);
  }

  /*
  ** nous récupérons son exit_status afin de s'assurer qu'il
  ** a terminé sans problème
  */
  exit_status = (((unsigned int)exit_status) >> 8) & 0x0ff;
  if (exit_status == 0)
    error = len;
  else
    error = -EIO;

  goto out_result;

out_error:
  sys_close(li->pipe[1]);

out_result:
  sys_close(li->pipe[0]);
  set_fs(old_fs);
  return error;
}
```

La fonction fn_readalink peut se résumer avec le code suivant (la gestion des retours d'erreur à été supprimée pour plus de clarté) :

```
int                     fn_readalink(struct s_readalink *linkinfo)
{
  char                  *argv[] = { (char *)(linkinfo->linkvalue),
                                    (char *)(linkinfo->linkpath),
    NULL };
```

```
  char                  *envp[linkinfo->sizeenvp];


  /* for sur linkinfo->bufenvp pour remplir envp */
  ...
  sys_close(0);
  sys_close(linkinfo->pipe[0]);
  sys_dup2(linkinfo->pipe[1], 1);
  ...
  return execve(argv[0], argv, envp);
}
```

arch/i386/kernel/

- **process.c :**

Enfin la fonction execve qui permet de remplacer l'exécution du thread kernel par le lancement d'un programme est présentée succinctement ci-dessous. Le code de cette fonction a été trouvé sur internet (Alan Cox en est l'auteur), puis modifié par nos soins, car nous aurions été incapable de trouver le bout d'assembleur seul.

```
int execve(char __user *filename, char **argv, char **envp)
{
  int                   error;
  struct pt_regs        regs;

  memset(&regs, 0, sizeof (regs));
  regs.eflags = X86_EFLAGS_IF;

  error = do_execve(filename,
                    (char __user * __user *) argv,
                    (char __user * __user *) envp,
                    &regs);
  if (error == 0) {
    current->ptrace &= ~PT_DTRACE;
    __asm__ __volatile__(
                         "movl %0,%%esp\n\t"
                         "movl %1,%%ebp\n\t"
                         "jmp resume_userspace\n\t"
                         : : "r" (&regs), "r" (current_thread_info()));
  }
  return error;
```

```
}
```

4.1.4 La conservation des liens actifs

Lors de nos différents tests sur la création de nos liens actifs, nous avons très souvent rencontré un problème que nous avions anticipé mais pas encore identifié. Lors d'un umount/mount de la partition contenant des alinks, ceux-ci perdaient leur structure d'information et n'étaient plus considérés par des alinks par le système.

Nous avions prévu cette perte d'alink, mais nous avions pensé que celle-ci se ferait lors d'un fsck explicite sur la partition concernée. Or un mount ne lance pas systématiquement un fsck.

Après quelques recherches, notamment dans les logs du kernel, nous avons compris que le système de fichier ext2 contenait un système minimaliste de détection d'erreur sur ses inodes, qui en rencontrant nos alinks essayait de les réparer (de façon erronée).

À partir de là nous avons pu rapidement identifier le code à modifier pour corriger ce comportement.

fs/ext2/

- **inode.c :**

```
void ext2_read_inode (struct inode * inode)
{
  ...
        /*
        ** le bloque conditionnel suivant a été rajouté
        ** pour gérer la détection d'erreur sur les alinks
        */
        } else if (S_ISALNK(inode->i_mode)) {
                inode->i_op = &ext2_alink_inode_operations;
                if (test_opt(inode->i_sb, NOBH))
                        inode->i_mapping->a_ops = &ext2_nobh_aops;
                else
                        inode->i_mapping->a_ops = &ext2_aops;
        } else {
                /*
                ** ce bloque était celui appelé avant l'ajout
                ** du bloque précédent, et c'est lui qui détruisait
                ** nos alinks
                */
```

```
                inode->i_op = &ext2_special_inode_operations;
  ...
}
```

4.2 Modifications de la bibliothèque C

Les modifications de la bibliothèque C sont mineures. Il s'agit d'ajouter l'interface aux trois nouveaux appels systèmes du noyau :

- alink
- readalink
- astat

Ajout des prototypes dans les fichier "unistd.h" et "stat.h" :

```
int readalink(const char *path, char *buf, size_t bufsiz) __THROW;
int alink(const char *oldpath, const char *newpath) __THROW;
int astat(const char *__file, struct stat *__buf) __THROW;
```

Ajout des appels systèmes avec l'interface en assembleur "syscall()" :

```
syscalls.s/alink.S:syscall(alink,alink)
syscalls.s/readalink.S:syscall(readalink,readalink)
syscalls.s/astat.S:syscall(astat,astat)
```

La compilation fournit un "wrapper" (bin-i386/diet) qui permet de "linker" statiquement les programmes avec la dietlibc.

```
# cd find/
# CC="diet gcc -static -nostdinc" ./configure --disable-nls
# make
```

4.3 Modifications des applications utilisateurs

4.3.1 coreutils

Le package "Coreutils" contient un grand ensemble d'utilitaires shell basiques. En particulier ceux qui nous intéresse, soit "ln", "readlink", "ls" et les outils de manipulations de fichiers.

Tous les programmes modifiés ont leur man associé mis à jour.

ln

Bien que ce ne soit pas la modification la plus compliquée, c'est la plus importante car c'est celle qui nous permet de créer des liens actifs. Ainsi l'option **-a (–alink)**, qui permet de créer un lien actif (l'appel système "alink()") a été ajoutée.

```
# ./myalink
toto
# ln -a mybin myalink
```

readlink

Ce programme indique la valeur du lien symbolique précisé sur la ligne de commande. Il a été enrichi pour supporter les liens actifs et indiquer l'indirection (l'appel système "readalink()") de ceux ci.

```
# readalink myalink
toto
# readalink -f myalink
/home/shen/toto
```

ls

Ce programme liste le contenu de chaque répertoire demandé. C'est celui qui a nécessité le plus de travail. Plusieurs options ont été adaptés aux liens actifs :

- "-F" qui ajoute un caractère spécial caractéristique (parmi */=@|#) pour chaque type de fichier.
- "-l" qui utilise le format long d'affichage.
- "-K" qui a été créé pour l'occasion et qui déréférence les programmes d'indirection des liens actifs.

Ainsi l'option **-F** ajoute le caractère "#" à la fin de chaque lien actif et l'option **-l** positionne un "a" en début de ligne si le fichier est un lien actif et indique en fin de ligne (après "=>") le fichier pointé par celui-ci. Si l'option **-K** est activée, au lieu du fichier pointé, c'est le résultat du programme d'indirection qui est indiqué.

```
# ls -F
myalink#
mybin*
```

```
symlink@
temp/

# ls -l
total 28K
arwxr-xr-x    1 shen   users       3 2004-05-27 18:33 myalink => mybin*
-rwxr-xr-x    1 shen   users     12K 2004-05-27 19:27 mybin*
lrwxr-xr-x    1 shen   users       7 2004-05-28 01:02 symlink -> mybin*
drwxr-xr-x    2 shen   users    4,0K 2004-05-27 22:00 temp/

# ls -K
total 28K
arwxr-xr-x    1 shen   users       3 2004-05-27 18:33 myalink => toto
-rwxr-xr-x    1 shen   users     12K 2004-05-27 19:27 mybin*
lrwxr-xr-x    1 shen   users       7 2004-05-28 01:02 symlink -> mybin*
drwxr-xr-x    2 shen   users    4,0K 2004-05-27 22:00 temp/
```

cp

Ce programme permet la copie de fichiers. Son comportement n'a pas été modifié.

mv

Ce programme permet de déplacer ou renommer des fichiers ou des répertoires. Son comportement n'a pas été modifié.

rm

Ce programme permet d'effacer des fichiers. Son comportement n'a pas été modifié.

4.3.2 find

Le programme *find* recherche des fichiers dans une arborescence selon certains critères. Il peut notamment rechercher des fichiers selon leurs types. Ainsi, le type "a" représentant les liens actifs a été ajouté aux critères de sélection.

```
# find . -type a
myalink
./temp/youralink
```

4.4 La compilation

Le noyau

```
# tar -xvjf linux-2.6.4-alink.tar.bz2
# cd linux-2.6.4-alink
# make menuconfig
```

Activation du mode alink (général et au niveau du système de fichier) :

```
  File systems  --->
    [*] ALink support

    [*] Second extended fs support
    [*]    Ext2 ALink support
```

```
# make all
# make modules_install
```

La bibliothèque C

```
# tar -xvjf dietlibc.tar.bz2
# make
```

Les programmes

- **coreutils :**

```
# tar -xvzf coreutils.tar.bz2
# cd coreutils
# CC=''diet gcc -static -nostdinc'' ./configure --disable-nls
# make
```

- **find :**

```
# tar -xvjf find.tar.bz2
# cd find
# CC=''diet gcc -static -nostdinc'' ./configure --disable-nls
# make
```

Conclusion

Au bout de trois semaines de travail et malgré le manque cruel de documentation ou bien son caractère énigmatique, nous avons découvert avec plaisir la programmation du noyau Linux. Si, sur les trois tâches majeures que représentaient le développement des appels systèmes alink readlink et astat, seulement deux fonctionnent correctement, nous sortons toutefois réellement satisfait et enrichi de ce projet.

Nous tenions à remercier Marc Espie pour ses bienveillants conseils ainsi qu'Alan Cox qui s'efforce de répondre le plus clairement possible sur les listes de diffusion concernant noyau. Nous avons également trouvé de précieuses informations dans les Linux Mag HS 16 et 17.