N° Ordre : 3134

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention : INFORMATIQUE

par

Yoann PADIOLEAU

Équipe d'accueil : IRISA (projet LANDE)
École doctorale : MATISSE
Composante universitaire : IFSIC

Titre de la thèse :

Logic File System

un système de fichier basé sur la logique

soutenue le 11 février 2005 devant la commission d'Examen

Mr	Jean-Pierre	Banâtre	Président
Mr	Gilles	Muller	Rapporteurs
Mme	Marie-Christine	Rousset	
Mr	Marc	Shapiro	Examinateurs
Mr	Nicolas	Spyratos	
Mr	Jean-François	Boulicaut	Invité
Mr	Olivier	Ridoux	Directeur

Résumé

Les systèmes d'information offrent des moyens pour *organiser*, *chercher*, et *manipuler* l'information. Ils deviennent de plus en plus important avec l'avènement de l'âge numérique et l'augmentation de la variété et du nombre des documents digitaux (fichiers musicaux, images, vidéos, e-mails, pages Web, sources de programmes, documents XML). Pour rechercher ces documents, les systèmes d'information traditionnels, comme les systèmes de fichiers ou le Web avec ses moteurs de recherche, fournissent chacun des outils de *navigation* et d'*interrogation*, mais ne permettent pas de les combiner. D'un côté, la navigation est intuitive et progressive mais elle implique une classification des données rigide et unique. De l'autre côté, l'interrogation apporte flexibilité et expressivité, mais ne possède pas les avantages de la navigation. Pour manipuler le contenu de ces documents, d'autres outils existent, comme les éditeurs de texte ou les environnements de développement, mais ils souffrent des mêmes limitations. Nous proposons un nouveau paradigme pour les systèmes d'information basé sur la *logique* : le *Logic File System* (LFS), qui offre une organisation expressive, une recherche combinant interrogation et navigation, et une facilité de manipulation, à la fois des fichiers et de leurs contenus ; le tout d'une façon intégrée et mis en œuvre au niveau système de fichier.
Pour atteindre cette intégration, ce paradigme associe des *propriétés logiques* aux fichiers et *parties* de fichiers, et la *déduction logique* sert de base pour naviguer et interroger. *Les chemins sont des formules*. Les répertoires représentent des requêtes logiques et déterminent des ensembles de fichiers et de parties de fichiers dont les propriétés *satisfont* la requête. Le répertoire racine représente la formule *vrai*, et les sous-répertoires d'un répertoire sont déterminés par les propriétés les plus générales permettant de *raffiner* la requête, combinant ainsi navigation et interrogation. Le contenu des fichiers est déterminé par les parties du fichier original qui satisfont la requête. Cela permet des accès simultanés en lecture et écriture sur différentes *vues* d'un même fichier, afin d'aider l'utilisateur à séparer et discerner les différents *aspects* de l'information qui sont contenus dans ce fichier. Les propriétés peuvent être attachées aux fichiers manuellement par l'utilisateur et automatiquement par des programmes appelés *transducteurs*, et peuvent être ordonnées manuellement par l'utilisateur pour former des taxinomies ou automatiquement par des *moteurs de déduction logique*. Les utilisateurs peuvent étendre dynamiquement le système en fournissant leurs propres moteurs de déduction logique et transducteurs.
Nous présentons les principes, usages, algorithmes, structures de données et détails d'implémentation de LFS. Nous présentons aussi un *modèle de sécurité* pour ce système de fichier, basé sur la logique, et qui s'intègre naturellement avec «l'esprit logique» de LFS. Nous présentons des extensions à cet esprit logique comme la possibilité de formuler des requêtes Prolog, et des extensions réflexives unifiant fichiers et propriétés. Finalement, des expérimentations concrètes utilisant un prototype de LFS sont présentées, démontrant la viabilité de l'approche LFS pour la gestion de l'information, à la fois en terme d'efficacité et d'utilité.
Mots-clefs : système de fichier, logique appliquée, interrogation, navigation, vue

Remerciements

Je ne sais pas si ce genre de travail mérite des remerciements. Après tout ce n'est qu'une thèse. Cependant, on n'a peu l'occasion de remercier les gens qu'on aime, par gêne, et c'est plus facile par écrit ; je vais donc en profiter.

J'aimerais tout d'abord remercier trois personnes dont j'ai beaucoup appris : Pascal Rigaux (*aka* Pixel), Pascal Fradet, et Olivier Ridoux. Pixel m'a donné encore plus l'envie de devenir programmeur. Mon maître Jedi il est. Il est aussi mon partenaire lors des ICFP contest. Il m'a initié à Emacs, à la beauté et la puissance des langages déclaratifs, ou encore à Linux. Il m'a fait découvrir de nombreux livres informatiques et l'intérêt même de lire des livres (plutôt que d'aller aux cours). C'est aussi lui qui m'a révélé le néant qui se cachait très souvent derrière les lettres grecques, les formules, et de manière générale derrière les travaux très formels, qui peuvent souvent à première vue impressionner. Rien ne vaut des exemples concrets, des explications informelles et intuitives, et une implémentation pour bien comprendre un concept informatique. Comme le dit aussi le professeur Chazarain : «Implanter pour mieux comprendre». C'est aussi Pixel qui m'a orienté vers Olivier Ridoux lors de mon DEA, choix qui se révélera payant (enfin je le crois) plus tard.

Pascal Fradet a été mon premier contact avec le monde de la recherche en m'encadrant lors d'un stage d'été. J'ai appris beaucoup de lui, même si je m'en suis rendu compte uniquement lorsqu'il est parti. Il a un peu tempéré mon jugement sur les méthodes formelles. Il m'a fait comprendre que les petits formalismes et faire abstraction de certain détails étaient souvent des facteurs clefs pour pouvoir résoudre un problème compliqué, de manière carré. C'est un homme très intelligent avec beaucoup de bons principes et aussi très drôle. L'IRISA aurait été un lieu bien triste sans sa présence, et les conversations scientifiques et autres auraient été beaucoup moins intéressantes et amusantes sans lui. Je regrette juste qu'il ne soit pas informaticien (c.a.d. qu'il ne programme pas).

Quant à Olivier, que dire. Il est intelligent, passionné, curieux de tout. C'est aussi la personne la plus gentille que je connaisse. Il a une très grande honnêteté intellectuelle. C'est un vrai scientifique, comme je les imaginais lorsque j'étais petit. C'est le meilleur directeur de thèse que je connaisse (c'est aussi le seul auquel j'ai été confronté). Je le remercie de m'avoir proposé ce sujet qui m'a passionné. Là où la plupart des gens se contentent des outils qu'on leur propose (les systèmes de fichier hiérarchiques), et n'y voient aucun problème, Olivier se révolte, identifie le problème et propose ou entrevoit des solutions originales. Olivier fait aussi partie de ces rares chercheurs qui s'attachent à résoudre des problèmes réels auxquels ils sont confrontés tous les jours et qui cherchent à programmer des outils dont il se serviront ensuite vraiment. Olivier balance habilement théorie et pratique. Encore une fois, je te remercie Olivier pour tout ce que tu as fait pour moi lors de cette thèse.

Je remercie Gilles Muller et Marie-Christine Rousset d'avoir accepté la lourde charge d'être les rapporteurs de cette thèse. Je remercie les membres du jury pour l'intérêt qu'ils ont porté à cette thèse. J'aimerais remercier plus particulièrement Jean-Pierre Banâtre, Marc Shapiro, et encore une fois Gilles Muller pour leurs encouragements, pour leur enthousiasme, et aussi pour leur aide et soutien lors de ma recherche d'emploi. C'est aussi Gilles qui m'a introduit dans la communauté système française en m'invitant à CFSE. Je l'en remercie.

Je remercie Richard Stallman, Linus Torvalds, Xavier Leroy, Larry Wall, Donald Knuth, et leurs acolytes, pour Emacs, Linux, OCaml, Perl, et TeX. Mon travail, et celui de nombreux autres étudiants seraient considérablement plus

difficile sans la présence de ces outils *open-source* de grande qualité. J'espère apporter avec mon travail à mon tour une petite pierre dans l'édifice GNU.

Je remercie Sébastien Ferré sans qui cette thèse serait beaucoup moins carré, et qui est à l'origine de nombreuses idées (voire trop) présentées dans cette thèse. Sébastien est sans doute la personne la plus intelligente et créative que je connaisse.

Je remercie mes copains du monde réel : Gaetan LeGuelvouit, David Sahuc, Pascal Rigaux, Yoann Picaud, Stephane Heintz, Pascal Garcia, Yacine Zemali, et Guillaume Cottenceau. Je remercie aussi mes copains du monde virtuel : Diablus, R1k1, Dmo, St3iner, Sournois et les autres membres de l'équipe FuN de Day of Defeat. L'ordinateur est le plus bel outil du monde, et le plus amusant.

Je remercie mes collègues à l'IRISA : Pascal Fradet, David Pichardie, Fréderic Besson, Sébastien Ferré, Jacques Noye, Florimont Ployette, Thomas Genet, Thomas Jensen, et Bertrand Jeannet.

Je remercie l'État Français d'avoir financé ma thèse et mes études. Je remercie les enseignants de l'INSA de Rennes pour leurs enseignements de qualité.

Enfin, j'aimerais dire un grand merci à mon papa et à ma maman, sans qui, je ne serai pas là (forcément), qui m'ont toujours supporté (dans tous les sens du terme), et qui ont tout de même financé une partie de ma thèse (papa, maman, la recherche française avance grâce à vous). Papa, maman, je vous aime. J'aimerais encore remercier ma maman qui a corrigé de nombreuses fautes d'orthographe dans ce manuscrit. Si il en reste, c'est donc de sa faute. J'aimerais aussi remercier mes deux sœurettes, qui grâce à leurs présences féminines ont fait de moi cet être si sensible (peut-être trop).

Bon, place à la science :)

Cette thèse est dédiée à la mémoire d'Helsi.

Table des matières

Introduction

We've all heard that a million monkeys banging on a million typewriters will eventually reproduce the entire works of Shakespeare. Now, thanks to the Internet, we know this is not true.

Robert Silensky

In the future, search engines should be as useful as HAL in the movie 2001 : A Space Odyssey–but hopefully they won't kill people.

Sergey Brin

Sur la gestion de l'information

Les systèmes d'information offrent des moyens pour *organiser*, *chercher*, et *manipuler* l'information. Ils deviennent de plus en plus important avec l'avènement de l'âge numérique. De nos jours, les outils informatique donnent accès à une très grande variété et à un très grand nombre de documents numériques. Par exemple des e-mails, pages Web, fichiers audio, images et vidéos pour un usage personnel ; des programmes, spécifications, tests, documentations, pour un usage plus professionnel. Mais avons nous de bons outils pour gérer ces informations ?

L'information peut être vue sous différentes granularités : par exemple, le *Web* est un ensemble de *sites*, où chaque site est un ensemble de *fichiers*, où chaque fichier est un ensemble de *lignes*. Une URL reflète cette décomposition précisément avec une désignation standard : `http://site/fichier#ligne`. Des outils pour gérer ces informations existent à chaque niveau. Gérer autant d'information est une tâche complexe. Une difficulté importante pour l'utilisateur est de trouver ou retrouver rapidement une information parmi ce très grand nombre. En effet, avec des millions de sites, de fichiers et de lignes, on ne peut pas utiliser la méthode naïve consistant à les visiter tous.

Au niveau du *Web*, pour trouver un site, un utilisateur peut utiliser un moteur de recherche comme Yahoo ! et *naviguer* dans une *hiérarchie*. Cette approche est intuitive et progressive, mais la classification des sites montrée à l'utilisateur n'est pas forcément celle qu'il aimerait utiliser, et il peut donc se perdre. En effet, Yahoo ! ne supporte qu'une seule classification, en domaines et sous-domaines, ce qui est insuffisant. Par exemple, un utilisateur voudrait commencer une navigation dans une classification par pays (puis finir pourquoi pas ensuite par une navigation dans une classification par domaine). Bien sûr, on peut toujours se satisfaire d'une seule classification et donc d'un seul critère de recherche, mais le nombre de sites répondant à un critère peut être encore très grand, et la méthode naïve de recherche exhaustive peut s'avérer encore très longue. La conjonction de différents critères permettrait de restreindre de manière significative ce nombre.

Une approche plus flexible et expressive est d'utiliser un moteur de recherche comme Google et de formuler des *requêtes logiques*, par exemple une requête combinant le nom d'un pays et le nom de plusieurs domaines. On peut en plus des *conjonctions* formuler des *disjonctions* et des *négations*. Mais le nombre de sites satisfaisant une requête peut parfois être très grand[1] et l'utilisateur peut encore se retrouver perdu. On aimerait une approche plus progressive ; être capable de naviguer dans ces réponses. De plus, formuler des

[1] Google en classant les réponses par un algorithme de *ranking* [PBMW98] fait que ce grand nombre de réponses n'est en général pas gênant. En effet, souvent seules les premières réponses ont besoin d'être parcourues. Cependant, parfois cette technique ne suffit pas.

requêtes, c'est à dire *interroger*, est plus difficile que *naviguer*. En effet, très souvent l'utilisateur n'a qu'une vague idée de ce qu'il cherche et dans ce cas il est plus facile de reconnaître ce que l'on cherche que d'exprimer ce que l'on cherche.

Au niveau d'un *site*, un utilisateur peut aussi organiser ses données dans une hiérarchie via les répertoires et sous-répertoires d'un système de fichier, et naviguer avec un *shell* ou un *explorateur* pour trouver un fichier. Mais, comme pour Yahoo !, les systèmes de fichier ne permettent qu'une seule classification. Ainsi l'utilisateur peut classer ses fichiers musicaux par genre, ou par artiste, mais pas les deux en même temps. Les utilisateurs surmontent cette limitation en ordonnant les classifications entre elles, par exemple en classant d'abord par genre, puis par artiste. Un ordonnancement va rendre facile une recherche, par exemple trouver les artistes appartenant à un genre, mais va en rendre d'autres très difficile, par exemple lorsque l'utilisateur connaît l'artiste mais pas le genre d'un fichier musical. Ce problème est particulièrement visible sous Unix où la classification par type de fichiers (librairie, binaire, ou manuel), vient après d'autres classifications (par type d'installation, par cible). Ainsi, l'utilisateur cherchant un programme doit passer du temps à essayer de le trouver dans `/usr/bin`, ou dans `/usr/local/bin`, peut être dans `/opt/local/etc/bin`, ou encore dans `/opt/unsupported/bin`. En fait aucune décomposition en répertoires ne peut satisfaire tous les besoins à la fois.

Les systèmes d'exploitation fournissent des outils d'interrogation comme `find` pour combler certains de ces problèmes, mais ils n'ont pas la progressivité de la navigation. En effet, avec la navigation, l'ordinateur aide l'utilisateur, il lui propose des répertoires, permettant à l'utilisateur de trouver un fichier étape par étape de manière incrémentale en naviguant depuis les répertoires généraux vers les répertoires spécialisés. Avec `find` ou Google, l'utilisateur se retrouve sans aide pour trouver son chemin. De plus, le résultat de `find` n'est pas «de première classe» ce qui veut dire qu'un utilisateur ne peut le manipuler, alors qu'il peut manipuler le résultat de la navigation, par exemple en ajustant les fichiers et répertoires en utilisant les commandes `cp`, `rm`, et `mv`. Cela montre le manque d'intégration de la commande `find` avec le système de fichier. Si l'utilisateur choisit d'interroger, alors il ne peut plus utiliser les services offerts par le système comme la navigation ou la manipulation de fichiers avec les commandes standard.

Au niveau d'un *fichier*, prenons l'exemple de la programmation : un programmeur peut décomposer son problème en fonctions et sous-fonctions dans le cadre de la programmation fonctionnelle, ou bien en classes et sous-classes dans le cadre de la programmation objet. Il peut même utiliser un *explorateur* de classes ou de fonctions pour naviguer dans son programme, pour trouver rapidement les lignes appropriées qu'il veut manipuler. Cependant, le choix d'une organisation va rendre facile de trouver une méthode si l'on connaît sa classe, mais ne va pas aider pour trouver une méthode si l'on connaît uniquement son nom, ou son type, ou son créateur.

Chaque fonction ou opération peut elle même être décomposée en différents *aspects* : son interface, sa spécification, sa documentation, ses différentes versions, son code proprement dit qui peut aussi contenir différents aspects comme un aspect de déboggage, un aspect vérification pour la sécurité, des commentaires. Malheureusement, la structure rigide d'un fichier force l'utilisateur à favoriser un aspect, par exemple la décomposition en fonctions, et à projeter les autres aspects sur cette structure, par exemple en regroupant ensemble tout ce qui est relatif à une fonction, que ce soit son code, sa documentation, sa spécification. Cette organisation permet facilement à l'utilisateur de se faire une idée complète d'une fonction, mais elle rend difficile pour l'utilisateur de discerner ces différents aspects les uns des autres. Ainsi, l'utilisateur ne peut se faire facilement une idée générale de la spécification d'un programme, car l'aspect spécification est éparpillé un peu partout dans le fichier. En fait, comme pour la classification des fichiers en différents répertoires, aucun ajustement de lignes dans un ou même plusieurs fichiers ne peut satisfaire tous les besoins.

Ce qui est vrai pour les programmes, est vrai pour tout document digital un tant soit peu complexe. Par exemple, dans le domaine de la publication, un auteur décompose le source d'un livre (ou d'un document de thèse) en différents thèmes correspondant aux chapitres, et chaque chapitre est encore décomposé selon une autre classification avec régulièrement une introduction, des exemples, des principes et enfin un résumé du chapitre. Malheureuse-

ment, cette organisation rend difficile pour l'auteur (ainsi que pour le lecteur) d'avoir une vue générale de tous les résumés, afin par exemple de vérifier si ils sont en harmonie de style.

Pour pallier ces problèmes, le programmeur peut en première approche formuler des requêtes en utilisant un outil comme `grep`, par exemple pour chercher une méthode par son nom ou pour pouvoir se focaliser uniquement sur l'aspect spécification en sélectionnant les lignes appropriées. Mais l'usage de ces outils est moins intuitif. L'utilisateur aimerait aussi simplement qu'il navigue de fonctions en fonctions, naviguer d'aspects en aspects. De plus, comme pour `find` avec les fichiers, le résultat de `grep` n'est pas de première classe et donc où les commandes *Copy*, *Cut*, et *Paste* sont inutilisables.

Notre thèse

Quelque soit le niveau, les outils traditionnels rendent donc difficile d'organiser, de rechercher et de manipuler l'information. Quelque soit le niveau, la gestion de l'information souffre des mêmes problèmes, et suit le même schéma :

1. des moyens d'organisation très pauvres forcent l'utilisateur à décomposer l'information selon un seul point de vue avec une seule classification, ou à ordonner des choses orthogonales entre elles, ce qui ne peut répondre à tous les besoins de navigation.
2. la rigidité de Yahoo !, des répertoires et des fichiers limite donc les possibilités de navigation. Cela force l'utilisateur à utiliser des outils d'interrogation comme Google, `find` ou `grep`. Malheureusement, l'interrogation et la navigation sont deux modes de recherche qui s'ignorent l'un l'autre, et où les avantages d'un mode ne sont pas mis à profit afin d'éliminer les inconvénients d'un autre mode. Ce problème de combinaison se retrouve d'ailleurs dans chacun de ces mondes puisqu'il n'est pas possible de mélanger deux outils de navigations, deux outils d'interrogation, et de combiner de manière générale deux outils du fait du manque de principes généraux et d'un formalisme commun sous-tendant ces outils.
3. ces capacités de recherche limitées, alliées au fait que les résultats d'une interrogation ne sont pas de première classe, à leur tour limitent les possibilités de manipulation de l'information pour l'utilisateur.

Comme solution à ces problèmes, nous proposons dans cette thèse un nouveau paradigme pour les systèmes d'information offrant une organisation expressive, une recherche combinant étroitement interrogation et navigation, et une facilité de manipulation, à la fois des fichiers et de leurs contenus ; le tout d'une façon intégrée, au niveau système de fichier, et qui peut être implémenté de manière raisonnablement efficace.

Pour éviter le problème d'une unique classification des données, menant à une organisation pauvre de l'information, nous proposons des descriptions riches de l'information exprimées dans un langage *logique*. De plus, la *déduction logique* fournit en même temps les notions de *satisfaisabilité* et d'*ordre* ce qui offre des possibilités pour intégrer l'interrogation et la navigation.

En fait, l'idée d'un système d'information basé sur la logique (un LIS pour *Logic Information System*) a déjà été défendue par Ferré et Ridoux dans [Fer99, Fer02, FR04]. L'approche était théorique. Nous proposons donc de l'appliquer concrètement dans le contexte des systèmes de fichier. Implémenter et utiliser un vrai système de fichier a fait apparaître de nouveaux problèmes et de nouveaux besoins, et donc de nouvelles fonctionnalités. De plus, un vrai système de fichier doit être tolérant aux fautes, sécurisé, efficace en terme de vitesse et d'espace, et supporter de multiples utilisateurs, des problèmes dont [Fer02] ne parle pas ou peu. Ce nouveau système de fichier s'appelle le *Logic File System* (LFS).

Le fait d'implémenter ce nouveau type de système d'information au plus bas niveau, c'est à dire comme un système de fichier, permet d'offrir de nouveaux services facilement non seulement à l'utilisateur, mais aussi à toutes les applications de ce système sans que celles-ci aient eu besoin d'être recompilées. Les fonctionnalités offertes par LFS «irradieront» ainsi sur ces applications qui pourront faire de nouvelles choses, choses que même le concepteur de cette application n'avait pas imaginé. De plus, les principes des systèmes de fichier sont bien connus des utilisateurs, ce qui augmente les chances de

succès et d'adoption de LFS par ces utilisateurs.

Les problèmes évoqués dans cette introduction se retrouvent partout, dans n'importe quel type de système de gestion de l'information comme les gestionnaires d'e-mails, de bookmarks, ou d'agenda. Ces outils reproduisent à leur échelle les mêmes structures hiérarchiques, les même technologies, et donc souffrent des mêmes limitations. La cible d'utilisation de LFS est donc très large, et nous espérons faciliter la vie du particulier dans de nombreux secteurs de son utilisation de l'ordinateur mais aussi la vie du programmeur, de l'écrivain, du manager, de l'administrateur système, ou encore du chef de projet. LFS permettra d'utiliser plus largement les possibilités qu'offrent les technologies numériques.

Plan

Cette thèse se décompose en 3 parties, la première a pour but de présenter le *contexte* de cette étude, la deuxième les *contributions* de celle-ci, la dernière son *évaluation*.

La première partie présente les travaux existants autour des systèmes de fichier. Nous décrirons au Chapitre 1 les systèmes de fichier classiques, les extensions apportées à la notion de fichier et de répertoire, les modèles de sécurité associés aux systèmes de fichier classiques et enfin leur implémentation. Nous présenterons aussi des systèmes de fichier avancés répondant partiellement aux problèmes évoqués dans cette introduction.

La deuxième partie présente les contributions de cette étude, et présente donc en détail LFS. Nous présentons d'abord au Chapitre 2 les principes et l'usage concret de LFS. Nous y expliquerons aussi comment ce nouveau système de fichier résout les problèmes évoqués dans cette introduction. Puis, nous exposerons au Chapitre 3 l'implémentation de LFS avec ses algorithmes, structures de données et détails concrets d'implémentation. Nous présenterons au Chapitre 4 un nouveau modèle de sécurité pour ce nouveau système de fichier. En effet, les nouvelles fonctionnalités proposées par LFS permettent comme par effets de bord d'améliorer aussi l'expressivité des modèles de sécurité traditionnels. Nous présenterons au Chapitre 5 des extensions et de nouvelles fonctionnalités. Ces explications sont séparées pour des raisons didactiques du Chapitre 2 qui présente uniquement le noyau dur de LFS. Parmi ces extensions, la possibilité de formuler des requêtes Prolog et de pouvoir traiter un répertoire comme un fichier, et inversement, permet d'augmenter l'expressivité de LFS. Ces extensions rapprochent LFS des bases de données relationnelles ou déductives, tout en restant simple, et en préservant les atouts de l'approche LFS qui manquent aux bases de données, parmi eux la navigation.

La dernière partie présente l'évaluation de LFS pour la gestion de l'information. Le Chapitre 6 présente une évaluation pratique à la fois en termes d'efficacité et d'utilité. Des expérimentations effectuées avec un prototype de LFS y sont présentées. Nous y montrerons, encore un peu plus qu'au Chapitre 2, l'intérêt d'utiliser LFS pour gérer les données mentionnées dans cette introduction : des e-mails, fichiers musicaux, sources de programmes, documentations, etc.

Pour finir, nous présenterons dans la conclusion la synthèse de cette étude ainsi que des pistes de recherche.

Les Annexes A et B décrivent précisément la spécification formelle et l'implémentation du prototype de LFS, et peuvent être mises en relation respectivement avec les Chapitres 2 et 3 qu'elles complètent. Enfin, l'Annexe C contient le code d'exemples de *plug-ins*, qui comme nous le verrons sont le moyen qu'a l'utilisateur pour étendre les fonctionnalités de LFS. Afin de poursuivre la tradition du professeur Chazarain [Cha96] d'*implanter pour mieux comprendre*, nous présenterons donc dans ces annexes du code concret (dans le langage O'Caml [LDG+04]). De plus, encore selon cette tradition, nous irons lors de cette thèse *de la pratique vers la théorie*, ce qui assurera que cette théorie est effectivement utile, et que la pratique repose sur des bases et des principes solides.

À propos de ce document

Cette thèse est sur la gestion de l'information et décrit un outil pour mieux gérer les documents numériques. Cette thèse est aussi un document possédant une forme numérique, formant même alors un objet digital très complexe puisque ce document réunit plusieurs entités numériques dans le même document. En effet, cette thèse contient à la fois la documentation de LFS, sa spécifica-

tion, son code, ou encore des jeux de tests (par le biais d'exemples). Comme de plus LFS est une réalité, on peut donc l'utiliser pour «mieux» lire ce document. Le prototype LFS peut être téléchargé à partir de l'URL suivante :

`http://www.irisa.fr/LIS/LFS`

En fait, nous avons utilisé LFS pour écrire ce document, pour programmer LFS[2], ainsi que pour gérer nos autres fichiers (musiques, e-mails, etc). La Section 6.2 décrit certaines de ces expériences.

La version numérique de ce document, qui peut donc bénéficier de LFS, offre ainsi plus de services que la version papier. On peut alors avoir accès à différents points de vue sur cette thèse ou encore trouver plus facilement une information spécifique. On ne peut cependant pas décrire précisément les services et avantages de LFS sur la version papier dans cette introduction car c'est l'un des buts du reste de ce document.

Il faut noter cependant que la version papier offre tout de même certaines des fonctionnalités qu'offrent LFS avec la version numérique. Ainsi, si il est classique pour un document d'avoir une table des matières, ce qu'il l'est moins (du moins pour une thèse) c'est d'avoir un index. De plus, la bibliographie contient des références en arrière sur le texte (cité page x). Cela offre un premier moyen pour naviguer de différentes manières sur le document, ou pour trouver plus facilement certains types d'information.

[2] L'utilisation de LFS pour écrire le programme LFS amène alors à un phénomène de *bootstrapping*, comme pour les compilateurs.

Première partie

Contexte

Chapitre 1

Les systèmes de fichier

Nothing is more important than to see the sources of invention which are in my opinion more interesting than the invention themselves.

Leibniz

With an absurd oversimplification, the «invention» of the calculus is sometimes ascribed to two men, Newton and Leibniz. In reality, the calculus is the product of a long evolution that was neither initiated nor terminated by Newton and Leibniz, but in which both played a decisive part.

Richart Courant and Herbert Robbins

1.1 Introduction

Après l'*âge de la pierre*, puis l'*âge du papier*, nous connaissons de nos jours un nouvel âge de l'information : l'*âge numérique*. Le stockage et l'organisation de l'information passe d'une gestion par l'homme dans le monde *réel*, à une gestion assistée par la machine dans un monde *virtuel*. Ainsi, les livres, les documents, les lettres, les photos passent progressivement du monde du papier au monde numérique. Il en va de même pour la musique et la vidéo. Même les programmes informatiques étaient au départ stockés sur papier sous forme de cartes perforées et sont passés les premiers dans le monde numérique.

On peut se poser la question de l'intérêt de cette migration. La raison en est que dans le monde réel, nous sommes malheureusement soumis aux contraintes physiques du monde réel. Ainsi, ces données papiers prennent énormément de place ce qui rend difficile leur organisation et manipulation. Un particulier peut difficilement emmener sa bibliothèque avec lui en voyage. De même, l'exploitation de ce grand volume de données, par exemple faire une recherche, est fastidieuse car elle doit être exécutée par une machine très lente : l'homme. On a certes inventé des outils et des machines pour aider l'homme, par exemple pour trier automatiquement les lettres selon leurs destinataires, mais ces outils du monde réel sont coûteux et difficiles à construire. Toutes ces difficultés font parties des raisons d'être des *supports numériques*, de l'*ordinateur*, et du principe d'*outil logiciel*.

Ainsi, on eut l'idée de stocker ces données dans la machine en espérant qu'elles seraient plus faciles à manipuler dans un monde virtuel. Il est aujourd'hui par exemple reconnu que l'édition de texte est plus facile à l'aide d'un ordinateur qu'à l'aide d'une machine à écrire.

Cela dit, autant il paraît facile de passer de l'âge de pierre à l'âge du papier, car cela reste dans un même monde, autant passer du papier à l'ordinateur, du réel au virtuel n'est pas si évident, même si cela paraît naturel de nos jours[1]. Ainsi, même si l'on inventa des moyens de stockages numérique *persistents* (bandes magnétiques, disques durs, CDROM, DVD), cela ne reste que des moyens de stockage et pas d'organisation. Ils offrent des services qui, tout comme pour la mémoire, restent de très bas niveau, c'est à dire la possibilité de lire et d'écrire un

[1] En fait, comme nous le verrons dans cette thèse ce passage n'a pas été complètement réussi.

nombre limité d'information dans une *cellule*. Dans le cas d'un disque on appelle une cellule un *secteur*. Malheureusement, très souvent un document ne tient pas entièrement dans une cellule. De plus, ces cellules ont finalement elles mêmes un emplacement physique dans le monde réel et sont donc soumises aux mêmes problèmes que les documents eux même.

La gestion la plus naïve d'un disque dur serait de le considérer comme une feuille géante composée d'un amas de petites cellules (les carreaux d'une feuille). Mais dans ce cas, il serait très fastidieux pour l'utilisateur de gérer un ensemble de documents, car il serait obligé de gérer manuellement des séparations pour ne pas confondre les documents, de devoir parfois déplacer des informations d'un document car elles déborderaient sur les cellules occupées par un autre document. Il fallut donc trouver un moyen d'organiser nos données sur ce support. On eut alors l'idée de s'inspirer du monde réel.

Entre le niveau des informations et celui des cellules, les concepts de *fichiers* et *répertoires* ont été introduits pour masquer certains aspects les plus physiques. L'explication de l'origine et des principes du fichier et du répertoire qui va suivre peut sembler évidente voir inutile. Après tout, ces concepts appartiennent à la vie courante avec la démocratisation de l'ordinateur. Cependant, il est important de comprendre précisément l'origine et la philosophie de ces concepts car ils seront justement remis en cause dans cette thèse.

1.1.1 Le fichier

On peut voir un secteur comme une feuille. Les livres, les documents, les lettres sont certes composés de plusieurs feuilles, mais elles sont reliées entre elles par un lien (reliure, trombone, enveloppe) pour former une entité clairement séparée des autres. On unifia alors ce principe d'organisation sous un concept unique : le *fichier*.

Dans le monde virtuel, le fichier représente l'entité de base. Le lien entre les secteurs est géré de manière interne. En fait, l'analogie avec le mondé réel est encore plus évidente lorsque l'on sait que le mot fichier désigne à l'origine une boîte contenant une collection de fiches. Ainsi, l'utilisateur au lieu de manipuler une structure de bas niveau et très physique, le secteur, peut travailler sur une structure de plus haut niveau et plus logique, le fichier. La machine se charge de transformer les opérations logiques en opérations physiques.

Un fichier est composé de deux éléments : un *contenu* et un *nom*[2]. Le nom représente le moyen de haut niveau pour accéder à ce document, un peu comme le nom d'un livre sur sa reliure permet de le retrouver lorsqu'il est disposé dans une bibliothèque. Le principe des *systèmes de fichier* est de fournir une abstraction du disque, quelque chose de plus facile à manipuler pour l'utilisateur. Son premier service est de permettre à l'utilisateur de gérer ses fichiers et leurs contenus.

1.1.2 Le répertoire

Cependant, la multiplication des données entraîne une multiplication des fichiers. Le fichier comme seul moyen d'organisation ne suffit pas. En effet, avec des milliers de fichiers, il paraît difficile pour un utilisateur de s'y retrouver. Ainsi, après avoir organisé les données en les plaçant dans différents fichiers, il faut organiser ces fichiers eux mêmes. Là encore on s'inspira du monde réel où pour pouvoir retrouver rapidement quelque chose, on classifie. Ainsi, on stocke nos affaires dans différentes pièces, puis dans différentes commodes, puis dans différentes étagères, puis classeurs, feuillets, et ainsi de suite. On unifia alors ce principe d'organisation sous un concept unique : le *répertoire*.

Un répertoire est aussi composé d'un *nom* (l'étiquette d'une étagère), et d'un *contenu*, lui même divisé en deux parties : une *liste de fichiers* (les livres de cette étagère), qui peut être vide, et une liste de *sous-répertoires* (des sous-étagères), qui peut elle aussi être vide. Les répertoires sont ainsi dans une relation père-fils, et l'on appelle *racine* le plus grand ancêtre, celui qui n'a pas de père. Le deuxième service d'un système de fichier est donc de permettre à l'utilisateur de ranger ses fichiers à l'aide de répertoires, ce qui permet ensuite à l'utilisateur de *naviguer* parmi ses fichiers.

Ainsi, pour gérer l'information, nous sommes passés du papier au fichier et de l'étagère au répertoire. Ces deux concepts, le fichier et le répertoire, forment les deux concepts majeurs d'un système de fichier. Cette vision

[2] En fait, un fichier possède aussi des *propriétés systèmes* comme le nom du propriétaire du fichier. Ces propriétés seront décrites plus loin.

des choses, la *métaphore physique*, avec ses analogies entre les concepts du monde réel et ceux du monde virtuel, domine et forme ce que l'on appelle la classe des *systèmes de fichier classiques* ou *hiérarchiques*.

Cette métaphore a finalement peu évolué depuis les premiers jours des systèmes de fichiers, et ces systèmes de fichiers hiérarchiques restent les seuls vraiment utilisés de nos jours. Cependant, en s'inspirant du monde réel, les systèmes hiérarchiques restent finalement des modèles très physiques et donc assez contraignants. C'est donc cette métaphore physique et les principes originels du fichier et répertoire qui seront remis en cause dans cette thèse.

Plan

Dans les sections qui suivent nous allons dresser un panorama de tout ce qui tourne autour des systèmes de fichier, domaine très riche en travaux de recherche. Nous allons tout d'abord décrire plus précisément les systèmes de fichier hiérarchiques et leur interface du point de vue d'un utilisateur. Puis, on présentera les travaux qui se sont concentrés sur le pouvoir d'abstraction et d'unification du fichier et du répertoire. Ces travaux étendent ce pouvoir en utilisant le fichier et le répertoire pour autre chose que le stockage et l'organisation des documents d'un utilisateur. Nous verrons ensuite de véritables systèmes de fichier alternatifs, qui contrairement aux travaux précédents n'étendent pas, mais changent en profondeur les principes d'organisation des systèmes de fichier hiérarchiques. Ils s'attaquent essentiellement au principe originel du répertoire. Nous parlerons ensuite d'un aspect des systèmes de fichier que l'on n'a pas encore évoqué : la sécurité. Pour reprendre l'analogie avec le monde réel, certains tiroirs ont une clef, les fichiers et répertoires aussi. Là aussi une solution classique s'est imposée, mais quelques alternatives existent, dont la nôtre qui sera présentée au Chapitre 4. Enfin, nous expliquerons comment sont implémentés les systèmes de fichier.

Il existe aussi de nombreux travaux visant à améliorer l'efficacité et la tolérance aux fautes des systèmes de fichier. Ces deux dernières caractéristiques sont souvent considérées comme les plus importantes d'un système de fichier, et représentent finalement la majeure partie ou la partie la plus visible des travaux autour des systèmes de fichier. Cependant, dans cette bibliographie nous nous sommes concentrés sur l'aspect *fonctionnel* des systèmes de fichier.

1.2 Système de fichier classique

Comme dit précédemment, les deux principaux concepts d'un système de fichier sont le fichier et le répertoire. Ces répertoires sont organisés dans une relation père-fils. On peut représenter l'organisation de l'information sous la forme d'un *arbre*, un peu comme pour un arbre généalogique, avec aux nœuds les répertoires et aux feuilles les fichiers. La Figure 1.1 en montre un exemple.

Les travaux à l'origine du système de fichier hiérarchique sont les travaux autour de Multics [Org72, DN65] et d'Unix [RT74] qui a popularisé son usage. Le système de fichier est la partie du *système d'exploitation* qui fournit des services pour gérer la forme de cet arbre, et le contenu de ses feuilles.

On peut décrire ces services de différentes manières. En effet, un système d'exploitation au sens général du terme est composé de différentes couches, et l'utilisateur peut accéder aux services du système à travers ces différentes couches. Au plus bas niveau il y a le *noyau*, où les services sont représentés par des *appels systèmes*, et au plus haut niveau les *applications*, où les services sont représentés par exemple par l'interface graphique de l'application. Nous avons choisi de décrire les services du système de fichier à un niveau intermédiaire, celui du *shell* [Bou78, AL92]. À ce niveau, nous ne sommes pas embêtés par des particularités de plateformes ou des détails d'implémentation. Une description de plus bas niveau noierait les concepts fondamentaux dans une masse de détails. De plus, au niveau du *shell*, les différents services sont clairement identifiés et correspondent chacun à un nom de commande précis. Une description de plus haut niveau, comme celle d'une application graphique comme un explorateur de fichiers, serait certes plus moderne, mais elle serait moins précise. L'exécution d'une commande met en jeu moins de concepts que la description d'un clic de souris, et est donc plus claire.

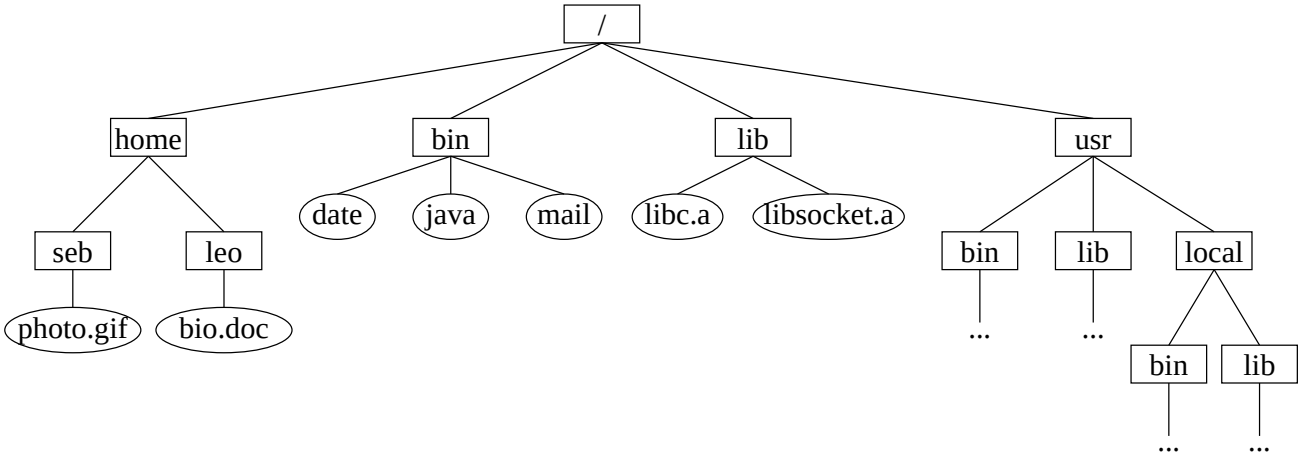

FIG. 1.1 – Un exemple typique d'organisation

1.2.1 Le chemin

L'introduction de répertoires implique que le nom d'un fichier seul ne suffit plus pour pouvoir *désigner* ce fichier. En effet, désormais plusieurs fichiers différents peuvent posséder le même nom du moment qu'ils ne sont pas situés dans le même répertoire. Ainsi, afin que l'utilisateur puisse indiquer à la machine quel fichier il veut manipuler, il lui faut un moyen non ambiguë pour pouvoir le désigner dans l'arbre. Pour cela l'utilisateur a accès à deux méthodes de désignation : le *chemin absolu* ou le *chemin relatif* d'un fichier.

Un chemin absolu est constitué d'une liste de noms de répertoires qu'il faut emprunter depuis la racine de l'arbre avant d'arriver au répertoire contenant le fichier désiré, d'où le terme de chemin ou *path* en anglais. Ces noms de répertoire sont reliés entre eux par le symbole `/` et un chemin absolu est précédé aussi du symbole `/`, cette fois plus en tant que séparateur, mais en tant que symbole représentant la racine. Ainsi, le chemin absolu du fichier `bio.doc` dans la Figure 1.1 est `/home/leo/bio.doc`.

L'autre méthode, celle du chemin relatif, fonctionne en relation avec un autre concept, celui du *répertoire de travail* (*working directory*). Il serait fastidieux pour un utilisateur de toujours avoir à spécifier les chemins absolus des fichiers qu'il veut manipuler du fait qu'ils peuvent parfois être très longs. Ainsi, chaque processus, et donc en particulier un *shell*, possède un répertoire de travail, c'est à dire un endroit de l'arbre duquel toutes les commandes lancées partiront. Ainsi, si le répertoire de travail d'un utilisateur est `/home`, il ne suffit plus alors à l'utilisateur qu'à mentionner le chemin relatif `leo/bio.doc` pour combler le vide et pour pouvoir accéder au fichier désiré. Pour se différencier d'un chemin absolu, un chemin relatif n'est pas précédé du symbole `/`.

1.2.2 Les commandes *shell* standards

Nous allons maintenant décrire l'ensemble des services offerts par un système de fichier, c'est à dire un ensemble de commandes accessibles à l'aide d'un *shell*. Ces commandes prennent en paramètres des noms de répertoires ou de fichiers. Elles peuvent aussi prendre en paramètre des chemins absolus ou relatifs de ces répertoires ou fichiers, ainsi que des options, mais nous nous bornerons à des exemples simples dans ce qui suit. Les commandes sont :

`pwd` pour *print working directory*. Cette commande permet d'afficher le chemin absolu du répertoire de travail, l'«endroit courant». On lui associe en général la variable *shell* `PWD` pour *process working directory*.

`cd` pour *change directory*. Cette commande permet à l'utilisateur de se déplacer dans l'arbre, c'est à dire de changer de répertoire de travail. Elle peut prendre en paramètre le nom d'un sous-répertoire du répertoire courant, ce qui permet de se déplacer vers le bas dans l'arbre. Elle peut aussi reconnaître deux paramètres spéciaux : `..` et `/`. La commande `cd ..` permet de remonter d'un cran dans l'arbre, c'est à dire de se placer dans le répertoire parent du répertoire courant. La commande `cd /` permet de remonter d'un coup à la racine de l'arbre.

`ls` pour *listing*. Cette commande permet de lister le contenu du répertoire courant, c'est à dire les noms des sous-répertoires et fichiers de ce répertoire. C'est cette commande associée à la commande `cd` qui permet la navigation. De nos jours, ces deux opérations sont en général combinées comme dans les explorateurs de fichiers à la Windows. Le clic de la souris sur un répertoire représente le `cd` et la commande `ls` est automatiquement lancée pour rafraîchir le contenu de l'interface graphique.

`touch/rm` Ces commandes permettent respectivement de créer et détruire une feuille de l'arbre, c'est à dire un fichier. Elles prennent donc en paramètre le nom d'un fichier, et ajoutent ou effacent une entrée dans la liste des fichiers du répertoire courant.

`mkdir/rmdir` pour respectivement *make directory* et *remove directory*. Comme leurs noms l'indiquent, ces commandes permettent respectivement de créer et détruire un nœud de l'arbre, c'est à dire un répertoire. Elles prennent donc en paramètre le nom d'un répertoire, et ajoutent ou effacent une entrée dans la liste des sous-répertoires du répertoire courant.

read/write En fait, il n'existe pas à proprement dit de commandes *shell* pour lire ou écrire le contenu d'un fichier. L'utilisateur peut utiliser pour lire un utilitaire comme `cat` ou `less` ou un éditeur comme Emacs [Sta81]. Pour écrire dans un fichier, il peut utiliser un éditeur, ou bien la combinaison d'une commande comme `echo` avec des commandes de *redirection* comme dans `echo "contenu foo" > foo.txt`, ou bien encore une commande comme `sed`.

`mv` pour *move*. Cette commande joue plusieurs rôles à la fois. Elle peut tout d'abord servir à renommer un fichier ou répertoire. Dans ce cas elle prend deux paramètres, l'ancien nom et le nouveau nom. Elle peut aussi servir à déplacer dans l'arbre un fichier ou un répertoire. Dans ce cas elle prend aussi deux paramètres, le nom d'un fichier ou sous-répertoire du répertoire courant dit répertoire *source*, et le chemin relatif ou absolu du répertoire de destination dit répertoire *cible* dans lequel résidera désormais ce fichier ou répertoire. Il faut noter pour le cas du déplacement d'un répertoire, c'est le répertoire et tout son contenu qui est déplacé ; on déplace donc toute une branche de l'arbre. Ainsi, le répertoire cible ne peut être un membre de cette branche ; cela ne peut être qu'un ancêtre, un frère ou un cousin du répertoire source.

Il existe d'autres commandes *shell* qui sont liées à la manipulation du système de fichier, c'est à dire de l'arbre, mais elles se résument à l'usage combiné de ces commandes. On peut donc voir les commandes que nous avons présentées comme les commandes fondamentales d'un système de fichier. Ainsi, la commande `cp`, qui sert à copier un fichier, peut être vue comme du *sucre syntaxique* autour de l'usage combiné de `read` et `write`. En effet, `cp foo.c bar.c` équivaut à la commande `cat foo.c > bar.c`.

Encore une fois ces explications peuvent sembler évidentes voir inutiles. Cependant, on va se servir de ces commandes tout au long de cette thèse, il est donc important de bien les comprendre. Ces explications permettent de rappeler précisément l'ensemble des services et fonctionnalités fondamentales offerts par un système de fichier. De plus, c'est en terme de ces commandes que nous décrirons la sémantique de LFS en Annexe A.

1.2.3 Les liens

Comme nous l'avons dit précédemment, la structure d'organisation des données est un arbre. Une extension importante des systèmes de fichier hiérarchiques, les *liens*, permet de passer de cette structure en arbre à une structure en *graphe*, plus expressive. On distingue deux sortes de liens, les liens *hard*, dit aussi matériels ou physiques, qui ne concernent que les fichiers, et les liens *soft*, dit aussi symboliques ou logiques, qui peuvent concerner

aussi les répertoires. La Figure 1.2 montre un exemple d'organisation contenant ces deux sortes de liens.

`ln` pour *link*. Cette commande prend en paramètre le chemin d'un fichier (existant) de l'arbre, et un nom et ajoute dans la liste des fichiers du répertoire courant une nouvelle entrée qui «pointera» vers ce fichier. Ainsi, ce fichier sera désormais accessible de deux endroits différents de l'arbre.

L'intérêt des liens pour l'utilisateur est qu'il peut ranger le même fichier dans différents répertoires, et qu'il bénéficie ainsi de plus de chance pour retrouver ce fichier. En effet, le problème majeur des systèmes strictement hiérarchiques est qu'ils forcent l'utilisateur, tout comme dans le monde physique, à placer un fichier à un seul endroit ; or un fichier, tout comme un livre, peut appartenir en même temps à différentes catégories. Contrairement au monde physique, l'utilisateur pourrait facilement copier ce fichier dans deux répertoires, mais la modification d'une copie n'aurait malheureusement pas d'impact sur l'original. De plus, la copie occuperait elle aussi de la place sur le disque, ressource précieuse pour l'utilisateur.

Il faut noter que le nom du fichier peut être différent dans chacun de ces répertoires, d'où le deuxième argument de la commande `ln`. Il faut noter aussi que l'effacement d'un fichier d'un répertoire détruit en fait le lien qui unit ce fichier au répertoire. Ainsi, l'effacement réel d'un fichier du disque n'est effectif que si il ne reste plus aucun lien vers ce fichier.

`ln -s` Cette commande prend en paramètre une chaîne de caractère, correspondant au chemin d'un fichier ou d'un répertoire (qui peut ne pas encore exister), et un nom, et ajoute de la même manière une nouvelle entrée dans le répertoire courant. Les liens *soft* rendent plus ou moins obsolète l'usage des liens *hard*. Contrairement à ces derniers, les liens *soft* peuvent pointer aussi bien vers des fichiers que vers des répertoires. Pour les répertoires, cela permet de créer des sortes de *raccourcis* qui évitent d'avoir à emprunter le chemin complet d'un répertoire.

Tout comme nous verrons au Chapitre 2 comment LFS change la sémantique (mais pas l'interface) des commandes *shell* standards pour offrir de meilleures services, nous verrons au Chapitre 5 lorsque nous présenterons les extensions de LFS la nouvelle sémantique et les nouveaux services des liens sous LFS.

1.3 Une interface unifiée

Comme dit dans l'introduction, le système de fichier propose une vision uniforme de l'information et de son organisation. En effet, une même entité, le fichier, est utilisée pour stocker aussi bien des données musicales que des vidéos, du texte, des images, ou des programmes. De la même manière, une même entité, le répertoire, est utilisé pour classer aussi bien des données musicales que des vidéos. Cette uniformité fait qu'un outil fonctionnant pour un type de fichier ou répertoire va pouvoir fonctionner aussi sur un autre type. Ainsi, la même commande, `cp`, peut être utilisée pour copier n'importe quel type de fichier. De même, la même commande, `find`, peut être utilisée pour parcourir récursivement n'importe quel type de classification.

De nombreux travaux sont allés plus loin dans cette uniformité, dans ce pouvoir unificateur du fichier et du répertoire. Ils rendent le système encore plus simple, les outils encore plus *génériques*, et comme nous allons le voir, rendent le terrain du système de fichier encore plus fertile pour voir naître de nouvelles idées et de nouveaux usages.

1.3.1 Extensions du fichier

Une des grandes avancés d'Unix [RT74] est de considérer tout comme un fichier, ou du moins de tendre vers cet objectif. Cette tendance s'est poursuivie d'ailleurs après, et a été même poussée à l'extrême dans le système d'exploitation Plan9 [PPD+95]. Ainsi, de nombreux concepts sont entrés au fur et à mesure dans ce moule.

Périphérique et fichier

Le premier d'entre eux fut le concept système de *périphérique*. En effet, sous Unix, les périphériques sont accessibles par l'intermédiaire de fichiers du répertoire `/dev` comme par exemple `/dev/mouse`, `/dev/hard-disk`, `/dev/printer`, ou

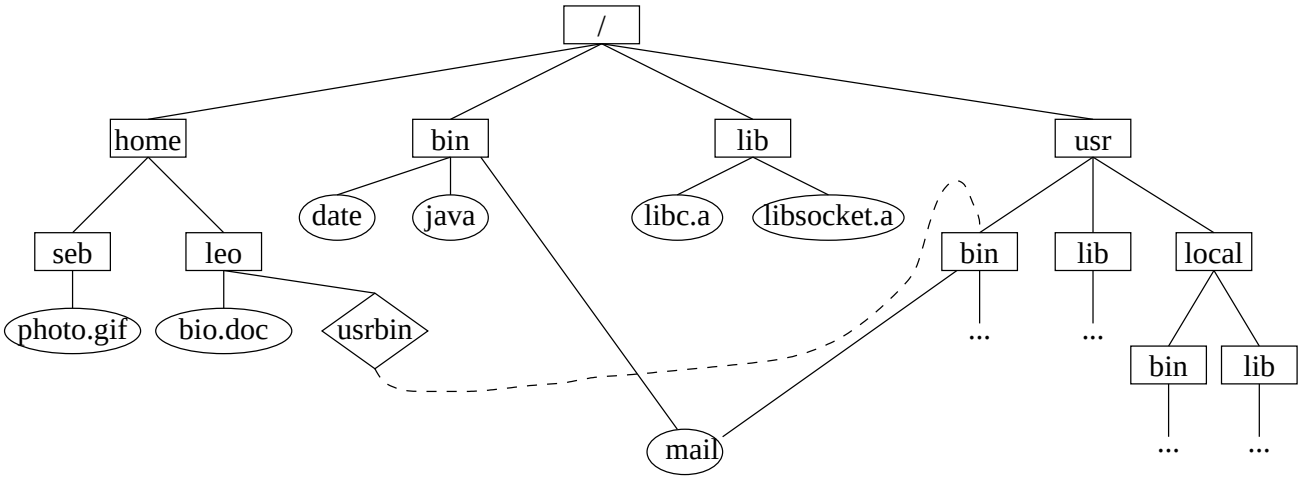

FIG. 1.2 – Les liens

`/dev/screen` pour respectivement accéder à la souris, au disque dur, à l'imprimante ou à l'écran. Cette vision uniforme a surtout un impact sur le programmeur qui peut utiliser un même protocole, quel que soit le type de périphérique, pour exploiter ce périphérique. Dans tous les cas, il suffit simplement de lire ou d'écrire des octets dans un fichier. Ainsi, l'émission d'un *beep* sonore est rendue intuitive et facile en écrivant tout naturellement dans le fichier `/dev/beep`. Cela paraît logique car après tout, on écrit et lit aussi sur un périphérique ; il n'y a donc pas de raison d'utiliser une interface différente de celle des fichiers, ce qui compliquerait inutilement les choses.

Cette intégration donne lieu aussi à des combinaisons intéressantes avec les autres fonctionnalités du système. En effet, on peut par exemple, aussi simplement que pour la copie d'un fichier normal, sauvegarder le contenu de l'écran (un *snapshot*) dans un fichier image en utilisant la commande `cp /dev/screen monimage.ppm`. On peut aussi copier le contenu d'un disque dur dans un autre à l'aide de la commande `cp /dev/hard-disk1 /dev/hard-disk2`, ou bien encore lancer une recherche brutale sur le disque à l'aide de la commande `grep foo /dev/hard-disk*`.

Une autre combinaison est celle des liens avec les périphériques. Une souris, une imprimante ou un lecteur de CDROM peuvent être connectés à la machine de différentes manières. Ainsi, selon certaines configurations, on pourra par exemple accéder à la souris par le biais du fichier `/dev/ps2` si elle est connectée à un port de type PS2, ou bien par le biais du fichier `/dev/parport` dans le cas d'un port parallèle. Pour simplifier la vie du programmeur, il suffit de créer les liens `/dev/mouse`, `/dev/cdrom`, `/dev/printer` et de les faire pointer vers les périphériques appropriés. Ainsi, là encore, le concepteur d'un lecteur logiciel de CD, en plus de ne pas à avoir à se soucier du type de matériel que possède l'utilisateur, n'aura pas non plus à se soucier de la manière dont est connecté ce matériel à la machine. L'ensemble des fonctionnalités du système travaillent de concert pour faciliter la vie de l'utilisateur. Se placer dans le monde des systèmes de fichier c'est ouvrir l'accès à de nombreux services et possibilités.

Enfin, tout comme on classe nos fichiers musicaux à l'aide de répertoires, on peut aussi classer les périphériques afin de mieux pouvoir ensuite les manipuler. Ainsi, on peut regrouper ensemble les périphériques liés aux entrées (la souris, le clavier), ou bien ceux liés au stockage d'information (disques durs, graveurs). Il suffit pour cela de simplement créer des répertoires (par exemple `mkdir /dev/input`), puis de déplacer les périphériques dans ces répertoires, tout naturellement à l'aide de la commande `mv` (par exemple `mv /dev/mouse /dev/input/`).

Processus et fichier

Un autre concept système intégré au monde du système de fichier est celui de *processus*. De la même manière que `/dev` contient les périphériques, le répertoire `/proc` [Kil84] contient les processus, tout du moins des informations sur les processus. Ainsi, ce répertoire contient un ensemble de sous-répertoires dont les noms sont chacun le numéro d'identification d'un processus (le *pid*), et qui contiennent chacun des fichiers représentant des informations sur ce processus. Ainsi, le répertoire `/proc/976/` contient entre autre les fichiers `cmdline`, `environ`, `cwd`, `mem` et `status` qui contiennent respectivement la ligne de commande responsable du lancement du processus `976`, les variables d'environnement (comme la variable `PATH` par exemple) du processus, le répertoire courant du processus, l'image mémoire du processus et enfin des informations concernant entre autre la consommation mémoire du processus. On peut voir cela comme une sorte de *réification* du noyau ; ses structures de données internes sont exposées à l'utilisateur.

L'intérêt de cette intégration est tout d'abord qu'il devient plus facile de comprendre le système ; il suffit de naviguer, d'observer les fichiers, leurs noms, leurs contenus. Rien n'est caché à l'utilisateur. Cela peut éviter aussi au programmeur d'avoir à retenir le nom de certaines fonctions systèmes, d'avoir à connaître certaines API (*Application Programming Interface*). Ainsi, pour tracer l'évolution de l'occupation mémoire d'un processus au cours du temps, il suffit tout simplement à intervalles réguliers de lire le fichier `status` correspondant.

Tout comme pour les périphériques, l'intégration des processus dans l'espace des fichiers donne lieu, ou plus précisément pourrait donner lieu à des combinaisons intéressantes avec les autres fonctionnalités du système. On pourrait par exemple tuer un processus en utilisant tout simplement la commande `rmdir /proc/976` au lieu d'utiliser une commande adhoc comme `kill`. Après tout, pourquoi inventer une interface différente lorsque les concepts sont finalement assez semblables. On pourrait aller plus loin en classant ces processus dans des répertoires selon leur type, leur taille, ou selon le nom du propriétaire du processus. Cela permettrait à l'administrateur de facilement formuler des requêtes évoluées comme la fermeture de l'ensemble des processus appartenant à l'utilisateur `foo` à l'aide de la commande `rmdir /proc/user:foo/*`. On pourrait lancer un processus de calcul, sauvegarder son état en copiant dans un fichier normal l'image mémoire courante de ce processus avec `cp /proc/976/mem /home/pad/bigcalcul`, redémarrer la machine et garder néanmoins la possibilité de poursuivre ce calcul en recopiant tout simplement ce fichier à son emplacement d'origine avec `cp /home/pad/bigcalcul /proc/976/mem`. Même si actuellement ces commandes ne sont pas possibles, rien ne s'y oppose dans l'absolu. Un point important est que certaines de ces idées, bien qu'avancées, viendraient naturellement à l'esprit d'un utilisateur. Tout du moins, elles viendraient plus naturellement que l'idée de chercher une commande s'appelant `kill` sous Unix, ou de trouver la séquence de touche `Ctrl-Alt-Suppr` sous Windows.

Entrée/sortie de processus et fichier

Une autre grande innovation d'Unix est de considérer l'*entrée standard* et la *sortie standard* d'un processus (le *stdin* et *stdout*) comme des fichiers. L'entrée standard est le moyen qu'a l'utilisateur pour communiquer son intention au processus via le clavier, et la sortie le moyen qu'a le processus pour communiquer son résultat à l'utilisateur via l'écran. Cette intégration est peu visible sous Unix car il n'y a pas à proprement parler de fichiers `/stdin` ou `/stdout`. En effet, afin de tirer au mieux profit du fait que le système est multitâche et multi-utilisateur, le système propose de multiples points d'interaction entre l'utilisateur et les applications : les *terminaux virtuels*. On ne peut donc pas avoir un fichier unique `/stdin` qui serait lié directement à `/dev/keyboard` et un ficher `/stdout` qui serait lié directement à `/dev/screen`, et dont les accès seraient partagés en même temps par tous les processus. Imaginons cependant temporairement que le système soit plus simple, et qu'il ne supporte qu'un seul terminal. Dans ce cas, on peut imaginer l'existence de ces fichiers `/stdin` et `/stdout` pointant vers les périphériques appropriés. On s'accorde ensuite pour que les fonctions d'affichage des librairies, comme la fonction `printf`, écrivent toujours dans le fichier `/stdout`, et les fonctions de lecture de données, comme la fonction

`scanf`, lisent toujours dans le fichier `/stdin`.

Comme un périphérique est lui même un fichier, il devient possible d'imaginer un moyen de lui substituer un fichier normal, et donc de dérouter les entrées/sorties vers des fichiers. C'est précisément ce que propose le mécanisme de *redirection* du *shell*, qui peut être vu d'une manière simplifiée comme du sucre syntaxique évitant à l'utilisateur d'avoir à ajuster lui-même les liaisons des fichiers `/stdin` et `/stdout`. Ainsi, la commande `ls > listing.txt` permet de sauvegarder la sortie standard du processus `ls` dans le fichier `listing.txt`. On peut ainsi ensuite utiliser d'autres commandes sur ce fichier comme `grep`, `sed` ou éditer ce fichier pour effectuer des traitements complémentaires. Cette intégration fournit donc un premier moyen pour combiner l'usage de différents programmes. On peut aussi imprimer le résultat d'un processus en redirigeant sa sortie vers `/dev/printer`. Un point important est que pour bénéficier de cette fonctionnalité supplémentaire, le source de l'application n'a pas été modifié. Avant cette intégration, la sauvegarde dans un fichier ou l'impression de la sortie d'un processus devaient être gérées manuellement par le programmeur de l'application et considérées comme des fonctionnalités supplémentaires à programmer, qui nécessiteraient l'utilisation d'un jeu de commande différent de celui utilisé pour imprimer un caractère à l'écran. Dans le cas d'Unix, l'application bénéficie gratuitement de ces services. Le jour où un nouveau service est accessible, par exemple avec l'introduction du périphérique `/dev/null`, il devient possible de lancer les applications en mode silencieux sans avoir à reprogrammer ces applications. On peut aussi rediriger l'entrée standard d'un processus comme dans `mail pad@irisa.fr < message.txt`. La commande Unix `mail` permet d'envoyer un e-mail et lit sur son entrée standard le message à envoyer.

Cette intégration conduit à son tour à une autre innovation d'Unix : le *pipe* (tube), qui offre un moyen encore plus simple pour combiner des programmes. Son invention résulte d'une observation très simple : du fait que l'entrée et la sortie d'un processus ont désormais tous les deux le statut de fichier, et que donc ils ont le même statut, il paraît possible d'imaginer que la sortie d'un processus soit directement mise en relation avec l'entrée d'un autre processus. Le mécanisme de *pipe* dans un *shell* peut être vu, là encore d'une manière simplifiée, comme du sucre syntaxique évitant à l'utilisateur d'avoir à créer lui même un fichier intermédiaire et d'avoir à ajuster ensuite lui-même les liaisons des fichiers `/stdin` et `/stdout`. Ce sucre rend encore plus facile et naturelle l'idée de combiner des processus. Ainsi, avec un utilitaire comme `grep` permettant de rechercher une chaîne, et un utilitaire comme `find` permettant d'afficher sous une forme textuelle l'ensemble des fichiers et sous-répertoires du répertoire courant, on peut concevoir très rapidement un nouvel utilitaire qui permet de rechercher la présence et localiser un fichier `foo.txt` en tapant simplement la commande `find | grep foo.txt`. Là encore, on voit qu'une fonctionnalité supplémentaire de `find` vient de lui être ajoutée sans que cette application ait été modifiée. Cette possibilité de combiner les processus encourage l'invention d'un grand nombre de petits utilitaires sous Unix que l'on peut combiner à volonté. Ainsi, la réalisation d'une tâche assez évoluée peut être résolue facilement avec une ligne de commande combinant de multiples utilitaires (voir par exemple [Ben85] qui décrit un vérificateur orthographique dont le code se réduit à la combinaison de 4 petits utilitaires : `sort`, `tr`, `comm`, et `uniq`, le tout dans une seule ligne de commande, ou encore l'excellent livre Software Tools [KP76]).

Ces mécanismes de redirection et de combinaison sont en fait des conséquences de l'intégration des périphériques dans le monde des fichiers.

Communication de processus et fichier

L'entrée et la sortie standard d'un processus ne sont pas ses seuls moyens de communication. Par exemple, dans le cadre des applications *concurrentes*, un processus aimerait communiquer avec de multiples processus, à travers de multiples *canaux*. Ces processus sont dans des espaces d'adressage différents, ce qui a de nombreux avantages, mais ce qui a l'inconvénient de rendre impossible les échanges d'information entre processus par *partage de variables*. Ces processus partagent cependant un espace commun : le système de fichier. Ainsi, un premier moyen (un peu brutal) pour faire communiquer deux processus est de leur faire s'échanger des informations par l'intermédiaire d'un fichier. De multiples canaux d'information peuvent être facilement représentés par de multiples fichiers. Les processus n'ont plus qu'à s'accorder

sur les noms de fichiers sur lesquelles ils se donneront rendez-vous. Cependant, cette méthode est simpliste au sens où elle ne fournit pas vraiment de moyens de *synchronisation*.

C'est justement ce genre de problèmes que résoud le domaine des fonctionnalités système que l'on appelle IPC pour *Inter Process Communication*. Parmi les mécanismes d'IPC qu'un système d'exploitation peut fournir on trouve le mécanisme de *mémoire partagée* avec comme moyen de synchronisation l'utilisation de *sémaphores*, ou bien encore le mécanisme des *messages*.

Le *pipe* est aussi un moyen de communication entre processus. Cependant, c'est un moyen implicite ; les applications se sont pas au courant qu'elles communiquent. Dans certaines situations cela est souhaitable et c'est d'ailleurs ce qui rend le *pipe* intéressant puisque deux applications ne se connaissant pas peuvent se combiner pour former une troisième application plus puissante. Cependant, ce genre de communication a ses limites car il ne fait intervenir que deux processus et qu'un seul canal de communication. En fait, on peut aussi utiliser le *pipe* de manière plus consciente, voir même utiliser plusieurs *pipes*, en utilisant les appels systèmes appropriés. Cependant une limitation importante est que les processus utilisant ce moyen de communication doivent être descendant d'un ancêtre commun qui a mis en place les outils de la communication.

On introduit alors un nouveau type de fichier : le *named pipe* [CQ83] ou *tube nommé* ou encore *fifo*. Comme son nom l'indique c'est une sorte de *pipe*, mais matérialisé de manière plus concrète et plus visible. Ainsi, la commande `mkfifo /tmp/comm.pipe` crée un tube nommé `comm.pipe`, visible de tous, dans le répertoire `/tmp`. Une application peut créer plusieurs tubes nommés. De multiples processus peuvent ainsi se donner *rendez-vous* sur ces fichiers. Ce mécanisme fournit aussi un moyen de synchronisation, car tout comme pour le *pipe* normal, les accès à ce fichier sont gérés de manière spéciale par le noyau. Il n'y a pas d'*attente active*, grâce à un système classique de sémaphore protégeant un *buffer* selon le principe des *producteurs/consommateurs*.

On peut pousser un peu plus loin cette idée du fichier comme moyen de communication. Ainsi, en plus de son utilisation pour les applications concurrentes, on l'utilise aussi pour les applications *distribuées*, par l'intermédiaire là encore d'un nouveau type de fichier : le *socket* [LFJ83]. Tout comme le *pipe*, le *socket* représente une sorte de tube dans lequel l'écriture d'une donnée à un bout peut être ensuite récupérée à l'autre bout. On peut voir le *socket* comme une généralisation du tube nommé, permettant en plus de faire communiquer les processus à travers un *réseau*. Deux processus sur deux machines différentes peuvent décider de communiquer entre eux en créant chacun de leur côté un *socket* et en les liant ensemble par un mécanisme d'adressage (une adresse internet et un port), d'une façon très semblable au principe du téléphone.

Tout comme pour le *pipe* normal, la création d'un *socket* n'entraîne pas la création d'un fichier concret dans le système de fichier, puisque ce fichier ne serait visible que d'une seule machine et ne présenterait donc aucun intérêt. Cependant, cette création retourne un *objet système* (un *descripteur de fichier*) ayant le même statut qu'un fichier. Ainsi, on lit et écrit dans un *socket* de la même manière que dans un fichier.

L'intérêt de l'intégration du *named pipe* et du *socket* dans le monde des fichiers est de rendre ces différentes formes de communication uniformes pour le programmeur.

Pseudo-périphérique et fichier

Nous avons présenté dans une section précédente une version simplifiée du mécanisme d'entrée/sortie qui ne prenait pas en compte la présence de terminaux virtuels. Cette simplification avait pour but d'aller à l'essence même de l'idée de l'intégration des entrées/sorties dans le monde des fichiers, et supposait donc l'existence d'un fichier `/stdin` et `/stdout`.

En fait, sous Unix il existe des périphériques spéciaux dans `/dev` qui ne représentent pas directement un périphérique matériel, mais qui agissent en temps que couche intermédiaire d'accès à un vrai périphérique. On les appelle les *pseudo-périphériques* [WO88] (périphérique virtuel). Ainsi, à chaque terminal virtuel correspond un pseudo-périphérique, par exemple `/dev/tty3` pour le troisième terminal, qui agit comme un intermédiaire pour l'accès de `/dev/screen` et `/dev/keyboard`, se chargeant de démultiplexer ces ressources. Le pseudo-périphérique se charge par exemple de mémoriser la sortie des processus, même lorsque le terminal n'est pas ac-

tif.

En fait, il existe même des pseudo-périphériques qui n'ont rien à voir avec des périphériques, qui n'agissent même pas en tant qu'intermédiaire, comme par exemple `/dev/null`, `/dev/zero`, ou bien `/dev/random` qui permettent respectivement de rediriger la sortie standard vers un trou noir (pour lancer les commandes en mode silencieux), d'initialiser le contenu d'un fichier avec des zéros, et enfin de récupérer un nombre aléatoire. Le fichier représente dans ce cas une sorte de service. Le code dans le noyau du *driver* correspondant offre un service, une fonctionnalité, par l'intermédiaire de l'interface d'un fichier. Ce code donne une sémantique particulière aux opérations de lecture et écriture sur ce fichier. Cette idée d'un service via un fichier fut popularisée par le système d'exploitation Plan9 [PPD+95] qui rendit plus facile que sous Unix pour un programmeur d'ajouter ce genre de services.

Une autre fonctionnalité de Plan9, le *namespace per process* permet à un processus de voir ou faire voir à un autre processus le système de fichier selon son propre point de vue. Ainsi, dans le système Plan9 il y a effectivement un fichier `/stdin`, ou plus exactement chaque processus voit un fichier `/stdin` à chaque fois lié à un périphérique différent.

Sous Plan9, le *système de fenêtrage*, qui représente là encore un composant important d'un système d'exploitation, utilise le même mécanisme que celui des terminaux virtuels : chaque *fenêtre*, en fait chaque processus responsable d'une fenêtre, voit un fichier `/dev/screen` qui représente un sous-ensemble de l'écran (la partie de l'écran allouée à la fenêtre), et un fichier `/dev/mouse` qui représente les coordonnées (et l'état) de la souris relativement à la fenêtre. Le système de fenêtrage est aussi une application mais elle voit les «vrais» `/dev/screen` et `/dev/mouse` et elle crée des pseudo-périphériques `/dev/screen` pour chaque fenêtre lancée. Là encore on démultiplexe la ressource écran et souris. Ainsi, la lecture par une application graphique de `/dev/mouse` va être déroutée sur le système de fenêtrage, qui va pouvoir accéder au vrai périphérique `/dev/mouse`, qui va vérifier si le pointeur de la souris est effectivement dans le périmètre de la fenêtre correspondante, qui va translater si nécessaire les coordonnées de la souris, et qui va placer dans le pseudo-périphérique correspondant les valeurs adéquates. Sous Plan9 l'ajout d'une fonctionnalité comme la possibilité d'avoir plusieurs bureaux virtuels serait très facile ; il suffirait d'ajouter encore une couche logiciel, très mince, entre le vrai `/dev/screen` et le `/dev/screen` que verra le système de fenêtrage. Là encore, le code du système de fenêtrage n'aura pas à être modifié. Un autre point assez remarquable est que comme Plan9 offre aussi un système de fichier distribué à la NFS (voir Section 1.3.2 page 21), la combinaison de ce système de fichier avec le système de fenêtrage rend ce système de fenêtrage aussi distribué gratuitement. On retrouve là encore l'intérêt d'intégrer un service dans le système de fichier puisqu'on peut ensuite facilement combiner ces différents services. À l'opposé, le système Xwindow [SG86] a du coder cette fonctionnalité de distribution. Il en résulte de plus un système de fenêtrage et une API bien moins élégante que le système de fenêtrage de Plan9 (appelé $8^{1/2}$ [Pik91]).

Un point important est que l'arrivée de nouvelles fonctionnalités, comme les terminaux virtuels, ne perturbe pas les applications qui ont la chance de croire vivre encore dans un monde simple. Nous évoluons dans un monde complexe et un rôle essentiel pour un système d'exploitation est de simplifier ce monde, ou du moins de faire croire qu'il est simple pour les utilisateurs et les applications. La simplicité du fichier contribue à cela.

Une autre façon d'associer une sémantique particulière à un fichier par l'intermédiaire d'un programme est celle du système de fichier intentionnel IFS [EP92]. Une *intention* est à opposer à une *extension*. On peut décrire un ensemble d'objets soit par son intention, c'est à dire par une formule qui décrit cet ensemble, par exemple $\{x \in \mathcal{N} \mid 0 < x < 10\}$, ou par son extension, c'est à dire en listant explicitement les objets de cet ensemble, par exemple $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. On peut voir un fichier comme un ensemble d'octets, et cet ensemble est décrit en extension. IFS propose de décrire un fichier aussi sous la forme d'une intention. Il implémente pour cela une extension aux liens symboliques. Ainsi la commande `ln -s '(date)' maintenant.txt` crée un fichier `maintenant.txt` dont le contenu sera déterminé par la sortie du processus `date`. Ce contenu est bien sûr dynamique. L'intention ou formule est donc le nom d'une commande.

La principale motivation d'IFS vient de l'observation que les machines modernes étant très rapides, le com-

promis temps-de-calcul/espace-disque peut devenir plus intéressant avec des fichiers intentionnels. En effet, le temps d'accès à un fichier `foo.tar.gz` sera certes quasi nul avec un fichier normal mais son espace disque sera important. Avec un fichier intentionnel, issu d'une commande comme `ln -s '(tar cf - foo/)' foo.tar.gz`, le temps d'accès ne sera pas si important et son espace disque nul. Cette méthode assure de plus la cohérence de tous les instants entre le répertoire `foo/` et son fichier d'archive `foo.tar.gz`. Tous les fichiers résultant d'un calcul peuvent ainsi être représentés de manière intentionnelle. Cela permet d'introduire certaines fonctionnalités de l'utilitaire `make` [Fel79] dans le système de fichier par exemple avec les commandes `ln -s '(gcc foo.c)' foo.o` et `ln -s '(gcc *.o)' foo.exe`.

1.3.2 Extensions du répertoire

Comme nous l'avons vu dans l'introduction, le répertoire permet à l'utilisateur de ranger ses documents. Il a bien d'autres usages. Il peut déjà servir à ranger les utilisateurs eux-mêmes. Ainsi, sous Unix chaque utilisateur a son propre répertoire dans `/home`. Comme nous l'avons vu dans les sections précédentes, il peut servir aussi à classer les processus, les périphériques. Un administrateur système peut aussi s'en servir pour spécifier à quelle fréquence doit être lancé un script de maintenance, un *cron*, en plaçant tout simplement ce script dans le répertoire adéquat, par exemple dans `/etc/cron.monthly/`. On se sert de cette même technique pour spécifier quels démons lancer au démarrage du système en fonction du *runlevel*. Un utilisateur `foo` peut accéder à ses e-mails en accédant au fichier `/var/spool/mail/foo` au lieu d'utiliser un protocole de communication adhoc. Les informations systèmes (dans `/var/log`), la gestion des impressions, ou la configuration des logiciels utilisent toutes les répertoires et fichiers.

Sous Unix, on se sert des répertoires et des fichiers pour organiser et hiérarchiser un grand nombre de choses. L'avantage de cette structuration très visible des choses est que l'on peut utiliser un utilitaire du genre de `find` comme méthode de recherche pour trouver un tas de choses : des services, des périphériques, des configurations de logiciels. Le système Windows par exemple n'a pas cette philosophie, et préfère une structuration plus interne. Ainsi, les données de configuration des logiciels d'un utilisateur sont toutes stockées dans un seul endroit, la base de registres. Ce système utilise très peu le répertoire comme structure de rangement. Il en résulte un système plus opaque que Unix, où la recherche d'entités variées nécessite l'utilisation d'interfaces très différentes les unes des autres, ce qui la rend moins régulière et donc plus difficile.

Tout comme pour le fichier, des travaux ont aussi essayé de réunir sous l'égide du répertoire un certain nombre de concepts, d'usages, nécessitant cette fois autre chose que de simples conventions.

Périphérique et répertoire

Une machine peut posséder différents périphériques de stockage : plusieurs disques durs, un lecteur de disquette, de CDROM, des bandes magnétiques. Chacun de ces périphériques possède en général un format particulier pour stocker les données, et donc un système de fichier particulier. De même, on peut utiliser différentes techniques pour implémenter un système de fichier, ou faire cohabiter sur le même périphérique différents types de systèmes de fichier, par exemple lorsque l'on veut utiliser différents systèmes d'exploitation. Il en résulte toute une faune de systèmes de fichier. Cependant, ces systèmes ne sont pas si différents les uns des autres, et possèdent tous les mêmes principes avec à chaque fois la notion de répertoire et de fichier. Ces systèmes diffèrent dans leurs détails d'implémentation et dans les limitations qu'ils imposent à ces entités, comme par exemple la longueur maximale autorisée pour le nom d'un fichier.

Le système Unix propose un moyen de les intégrer tous dans le même univers avec la technique du *système de fichier virtuel* ou VFS pour *Virtual File System* [Kle86]. Sous Unix, le système possède un périphérique principal et un système de fichier principal sur lequel se trouve les fichiers fondamentaux pour son bon fonctionnement. L'utilisateur peut ensuite *importer* ou greffer sur certains répertoires de ce système de fichier d'autres systèmes de fichiers, et donc d'autres périphériques, sur un autre. Par exemple, la commande `mount /dev/cdrom /mnt/cdrom -t iso9660` permet à l'utilisateur d'accéder aux données stockées sur son CDROM (dans le format `iso9660`) en naviguant sim-

plement dans le répertoire `/mnt/cdrom`. Là encore, il s'en suit que l'utilisateur croit vivre dans un monde idéal, où toutes ses données sont stockées au même endroit, sur le même périphérique de stockage, et que tout n'est que répertoire et sous-répertoire. Là encore, l'utilisateur peut utiliser la commande `find` pour trouver un fichier, et ce quelque soit l'endroit physique où il réside. à l'aide de la commande `mount`. On *monte* un système de fichier

Ce principe de système de fichier virtuel n'est pas présent dans d'autres systèmes comme DOS ou Windows. Dans ces systèmes, l'accès à une disquette, au disque dur, ou au réseau, utilise des jeux de commandes différents de ceux utilisés pour naviguer dans les répertoires. La programmation d'outils de recherche devient ainsi plus complexe.

Réseau et répertoire

L'arrivée d'un *réseau* introduit de nouveaux besoins : un utilisateur veut pouvoir accéder à des données stockées sur une machine distante. Un premier moyen est d'utiliser un programme particulier comme `ftp` (*File Transfer Protocol*), qui utilise ensuite en interne des *sockets* comme moyen de communication. Cependant, c'est un moyen très primitif ; il n'est par exemple pas possible d'éditer le contenu de ces fichiers. Ainsi, le système de fichier *Unix united system* [BMR82], et son descendant plus connu NFS pour *Network File System* [SGK+85] offrent une meilleure intégration du réseau. Un *site*, c'est à dire une machine, tout comme un périphérique de stockage, est représenté par un répertoire. Ainsi, la commande `mount bar.irisa.fr:/ /bar -t nfs` permet à l'utilisateur (par exemple de la machine `foo.irisa.fr`) d'accéder aux données stockées sur la machine distante `bar.irisa.fr` en naviguant tout simplement dans le répertoire local `/bar` de sa machine. L'accès à un fichier ou sous-répertoire de `/bar` induira une requête sur le réseau, traitée par un *serveur* spécial de la machine distante `bar.irisa.fr`, qui pourra accéder directement aux données locales en utilisant le système de fichier local, et qui renverra sur le réseau le résultat.

Cette fois ci, il est possible de modifier un fichier distant aussi facilement qu'un fichier local. On peut lancer des processus qui utilisent ces fichiers comme par exemple un éditeur de texte. La localisation d'un fichier est rendue *transparente*, et il se peut que l'utilisateur ne sache même pas que ses données sont en fait stockées sur une machine distante. Cela simplifie certaines tâches d'un administrateur système qui peut changer la configuration des machines ou la répartition du stockage des données des utilisateurs sur des disques du réseau, sans déranger ces utilisateurs. Là encore, un utilisateur peut lancer une recherche sur le réseau en utilisant toujours le même utilitaire : `find`. Toutes les applications du système peuvent désormais accéder à des données distantes sans que ces applications aient eu à être modifiées. Il est clair que l'intégration d'un concept, ici le réseau, dans le monde des systèmes de fichier est préférable à l'invention d'une application particulière, qui ne fournira pas tous les services, et qui ne bénéficiera pas aux autres applications du système.

Tout comme l'intégration des périphériques, cette intégration a des effets de bords intéressants. Elle fournit un premier moyen de communication pour les applications distribuées. Le système de fichier représente un espace commun de partage, et des processus situés sur différentes machines peuvent communiquer entre eux des données ou se synchroniser par l'intermédiaire de fichiers. Le concept de *socket* pourrait même devenir plus ou moins redondant puisqu'on pourrait lui substituer la combinaison des tubes nommés avec NFS[3]. De même, la combinaison d'un `/proc` plus avancé avec NFS permettrait à un utilisateur de migrer, dupliquer ou déployer des processus sur différentes machines, par exemple à l'aide de la commande `cp /proc/100/mem /machine*/proc/100/`.
L'idée d'utiliser ce genre de commandes arriverait naturellement à un utilisateur, du moins plus naturellement que l'utilisation d'API particulières comme PVM [Sun90] (*Parallel Virtual Machine*). Dans un monde unifié l'imagination se développe plus facilement.

Malheureusement, nous vivons dans un monde complexe, et toutes les machines ne sont pas reliées entre elles par NFS. Il existe donc d'autres protocoles réseau de partage de données comme HTTP (*HyperText Transfer Protocol*) ou FTP (*File Transfer Protocol*) et qui impliquent l'usage d'applications particulières (un

[3] Ce n'est cependant pas le cas pour l'instant car NFS impose encore des limites sur ce que l'on peut exporter.

browser Web, `ftp`). Des travaux se sont attachés cependant à minimiser ces différences en permettant d'accéder à toutes ces données de la même manière, en rendant ainsi transparent le protocole d'accès à un fichier. Ainsi, sous le système de fichier Alex [Cat92], un utilisateur peut simplement utiliser une commande comme `cd /ftp/cs.columbia.edu/pub/` pour naviguer sur un site `ftp`. En fait, ce système autorise même la commande `cd /edu/columbia/cs/ftp/pub/`, et donc intègre le système de nommage et de classification d'Internet, ce qui rend ainsi encore plus visible le parallèle entre l'organisation hiérarchique des machines sur le réseau et l'organisation hiérarchique de répertoires sur un disque. L'utilisateur peut cette fois utiliser n'importe quelles applications du système sur ces fichiers comme `grep`, et copier naturellement des données avec la commande `cp`. Là encore, fournir des services au plus bas niveau, via le système de fichier, c'est offrir plus de possibilités à toutes les applications du système.

Les données d'un utilisateur peuvent être réparties sur différents disques, voir même désormais avec NFS sur différentes machines, et donc dans des répertoires différents, à cause par exemple de capacités de stockage limitées. Cela montre que des contraintes physiques perdurent dans le monde virtuel ; l'utilisateur se voit obligé pour manipuler ses données de les chercher dans différents répertoires. L'intégration du réseau fait apparaître encore plus le besoin de mélanger le contenu de plusieurs répertoires. Certains systèmes comme Plan9 ou [PM95] permettent ainsi de voir un répertoire comme l'*union* de plusieurs répertoires, et par conséquent l'union de différents périphériques ou différentes machines.

Le système Prospero [Neu92] va même plus loin que l'union de répertoires en permettant à l'utilisateur de voir un répertoire comme une *fonction* de plusieurs répertoires. On associe un script qu'on appelle *filtre* sur un répertoire qui sera justement chargé de filtrer selon un critère choisi le contenu de différents répertoires.

Fichier et répertoire

Nous avons jusqu'à présent dressé une ligne de séparation forte entre le concept de répertoire et celui de fichier, entre le *contenant* et le *contenu*. Cependant, cette séparation n'est pas toujours si évidente.

Les fichiers d'archives par exemple contiennent eux-mêmes d'autres fichiers et répertoires. Le système de fichier IFS [EP92] permet aussi à un utilisateur ou à une application de naviguer dans ces fichiers d'archives, par exemple avec la commande `cd toto.tar#/`, sans avoir à les décompresser manuellement et à les effacer ensuite. gilles : dispo sous xp

Ce problème de séparation se retrouve à d'autres endroits. Un utilisateur est souvent soumis à une tension entre le fait de placer certaines données dans un même fichier, à les concaténer, car ces données ont effectivement toutes un lien entre elles et parlent de la même chose, et les classer encore un peu plus en les séparant dans différents fichiers (voir même ensuite de classer ces fichiers dans différents répertoires). Où s'arrête le répertoire et où commence le fichier ? En fait, on sépare en général un fichier lorsque celui-ci atteint une certaine taille, une masse critique au delà de laquelle l'édition et la compréhension du document devient trop difficile. Cependant cette décision n'est pas toujours facile à prendre car on peut parfois se trouver dans un état intermédiaire. De plus, chacun de ces deux «modes», la séparation et la concaténation, a ses avantages, et rend plus facile la réalisation de certaines tâches. La séparation des données permet une meilleure navigation sur ces données, et permet de mieux comprendre la structure des données. La concaténation des données peut permettre une édition plus facile.

L'observation que le contenu des fichiers est lui même très hiérarchique dans sa structure a conduit à la naissance et à la prolifération d'outils de navigation sur ces fichiers, qui permettent à l'utilisateur de bénéficier des avantages des deux modes, comme par exemple les explorateurs de classes, les gestionnaires d'e-mails. En fait, certains systèmes de fichier permettent de rentrer dans certains de ces fichiers, tout comme IFS permet de rentrer dans les fichiers d'archive. Ces systèmes miment ainsi les outils précédents. Une des extensions du système UserFS [Fit96] permet par exemple de naviguer dans le fichier qui stocke les e-mails d'un utilisateur. Chaque e-mail est représenté par un répertoire et chacun de ces répertoires contient une liste de fichier correspondant à un champ du mail, par exemple un fichier `From`. Cela permet d'utiliser la navigation dans les répertoires du système de fichier comme moyen uniforme pour explorer n'importe quelle forme de structure hiérarchique. Cette intégration évite aussi aux ap-

plications d'avoir à réécrire maintes et maintes fois le même genre de fonctionnalités, et permet de réutiliser ce qui a déjà été fait. Ainsi, dans la gestion des e-mails par un système de fichier, la fonctionnalité de recherche est fournit gratuitement par le système avec la commande `grep`, avec par exemple une requête comme `grep ridoux@irisa.fr emails/*/From`.

Ces systèmes de fichiers souffrent cependant de certaines limitations. Il n'est par exemple pas possible de modifier les fichiers représentant les parties du fichier dans lequel on est rentré. De plus, tout comme pour l'organisation des données dans un système de fichier classique, une seule classification (la décomposition du message en fichiers) est proposée à l'utilisateur. Le choix de cette classification n'est pas forcément celle qu'aimerait l'utilisateur. En fait, aucune décomposition n'est vraiment bonne puisque très souvent la structure d'un fichier est complexe et possède différentes *facettes*.

Nous reviendrons plus loin dans cette thèse sur ces problèmes de navigation à l'intérieur des fichiers, de tension entre séparer et concaténer, de modification ou encore de facettes puisque LFS offre justement des solutions à tous ces problèmes Section 2.3.2.

1.3.3 Discussions

Nous vivons dans un monde compliqué et l'*uniformiser* c'est le *simplifier*. Un des fils directeurs des travaux précédents est de faire croire aux utilisateurs et applications qu'ils vivent encore dans un monde simple, alors qu'en fait le nombre des fonctionnalités, que ce soit le réseau avec NFS ou les terminaux virtuels avec les pseudo-périphériques, a considérablement augmenté. La complexité a été cachée à l'utilisateur ; cette *transparence* fait que l'utilisateur n'est pas dérangé par des détails inutiles.

Cette uniformité est bénéfique à l'utilisateur qui n'a à apprendre que deux concepts : le fichier et le répertoire. Dans un monde unifié, on apprend une idée, une méthode, un outil, et l'on est alors capable de l'appliquer dans un grand nombre de situations.

Le principe d'*unification* est très recherché car même si il regroupe plusieurs concepts sous la même égide, c'est paradoxalement pour permettre plus de choses. L'utilisation d'un même formalisme fait que l'on imagine plus facilement des *combinaisons* entre ces différents concepts du fait qu'ils ont désormais le même statut. La force de l'esprit d'Unix où tout est fichier, est que le programmeur conçoit des outils qui permettent des choses qu'il n'avait même pas envisagé.

Nous verrons plus loin dans cette thèse, notamment Section 2.3.2 page 78 et Section 6.2.7 page 159, comment LFS en intégrant aussi de nouveaux concepts dans le système de fichier permettra de nouvelles combinaisons. Dans le passé, considérer les fichiers comme un ensemble plat de caractères a été une abstraction fructueuse permettant la combinaison d'outils via les *pipe*, les redirections. LFS permettra de retrouver de la structure tout en permettant encore la combinaison fructueuse d'outils et en permettra même de nouvelles.

Un autre point important et particulier au domaine des systèmes de fichier est que ce pouvoir d'abstraction est géré au plus bas niveau, au niveau du système d'exploitation. Ainsi l'ajout d'une fonctionnalité par l'intermédiaire du système de fichier «irradie» sur toutes les entités situées au dessus, et donc en particulier améliore les librairies et applications du système gratuitement, sans que celles-ci aient eu à être modifiées. Comme le disent Gifford et al. [GJSJ91] «*previous research supports our view that overloading file system semantics can improve system uniformity and utility when compared with the alternative of creation a new interface that is incompatible with existing applications*».

Là aussi, nous verrons notamment au Chapitre 2 et surtout au Chapitre 6 comment les nouvelles fonctionnalités offertes par LFS irradient sur les applications.

1.4 Systèmes de fichier avancés

Les systèmes précédents ne changent pas fondamentalement les caractéristiques du système de fichier hiérarchique. Ils généralisent ce qu'est un fichier et un répertoire, mais ces extensions ne concernent en général qu'un nombre réduit de fichiers ou répertoires. Un répertoire spécial va permettre par exemple de faire le lien entre deux mondes de stockage, différents dans leur statut, comme un stockage réseau et un stockage local, mais semblables au fond dans leur organisation en restant hiérarchiques. Ces extensions ne concernent donc pas l'organisation globale de ces fichiers et répertoires. Ainsi, ces travaux n'apportent rien pour aider l'utilisateur dans

son utilisation principale et basique du système de fichier : le rangement et la recherche de ses documents.

Comme dit dans l'introduction de ce chapitre, l'origine des idées du système de fichier est l'analogie entre les concepts du monde virtuel et ceux du monde réel. Cela a l'avantage de rendre ces concepts intuitifs puisque l'utilisateur les connaît déjà. Cependant, en copiant les principes d'organisation du monde réel, on a aussi copié leurs limitations.

Dans la réalité, on va classer par exemple les livres d'une bibliothèque scientifique en les plaçant dans différentes armoires puis étagères correspondant respectivement au domaine et sous-domaine scientifique des livres, pour pouvoir les retrouver plus facilement.

Le premier problème de ce genre d'organisation est qu'un livre peut parler de différents sujets, et donc appartenir en même temps à différents domaines, mais ne peut être placé qu'à un seul endroit.

Le deuxième problème est que cette organisation ne supporte qu'une seule classification. On aimerait aussi classer les livres selon le nom de leur auteur. En fait, cette deuxième classification est plus ou moins supportée dans la réalité en ordonnant dans chaque étagère les livres par le nom de l'auteur selon l'ordre alphabétique. Cependant, cette organisation ordonne les classifications entre elles et donne ainsi une priorité à l'une d'entre elles, ici la classification par domaine scientifique. Il n'est donc par exemple pas possible de chercher le livre par son auteur si l'on ne connaît pas déjà les domaines et sous-domaines de ce livre. De plus, cette méthode ne peut supporter qu'un nombre très limité de classification. En effet, une fois ces livres ordonnés par auteur, il devient inutile de les classer un peu plus car le nombre de livres d'un auteur sera souvent très petit. Malheureusement, on aimerait plein d'autres classifications pour ces livres, par éditeurs, par années, par langues. On retrouve ces mêmes problèmes avec les systèmes de fichiers hiérarchiques.

Les systèmes du monde virtuel peuvent faire mieux car ils peuvent justement transcender ces barrières physiques. C'est ce que tentent les systèmes de fichiers présentés dans les sections suivantes.

1.4.1 Semantic File System

Le Semantic File System (SFS) [GJSJ91] fut le premier système de fichier qui se démarqua de manière très forte des systèmes de fichiers hiérarchiques, le premier à vouloir changer radicalement l'organisation des choses, à changer la façon de voir un répertoire. Comme tout papier séminal il marqua un tournant important et inspira de nombreux autres travaux, notamment de nombreux autres systèmes de fichier avancés.

SFS s'appuie sur le constat que le contenu d'un fichier contient certainement de nombreuses informations intéressantes pour retrouver ensuite ce fichier, pourvu qu'on puisse en extraire les informations les plus utiles et qu'on puisse structurer ensuite ces informations. Ainsi, SFS associe à chaque type de fichier (un programme C, un e-mail ou encore un fichier musical) un programme spécial appelé *transducteur*, chargé d'extraire à partir du contenu des propriétés intéressantes sous la forme d'*attributs valués*. Ainsi, pour un programme C le transducteur retournera par exemple la propriété `function:main`, pour un e-mail la propriété `from:john`. Le nom *sémantique* vient du fait que SFS ne voit pas les fichiers comme une suite d'octets mais au contraire comprend d'une certaine manière le sens de ces fichiers. SFS, ou plutôt le transducteur, reconnaît ainsi la notion de fonction pour un programme C ou de destinataire pour un e-mail. On peut voir ces propriétés comme des extensions aux propriétés systèmes qui déjà pouvaient être vues comme des attributs valués, par exemple `owner:toto`. En fait, SFS utilise aussi les propriétés systèmes, ajoutant ainsi aux propriétés extraites par les transducteurs le nom du fichier (`name:toto`), son extension (`ext:c`), ses répertoires (`dir:bin`, `dir:usr`), sa date de création, son propriétaire et enfin son groupe.

Il ne reste plus alors qu'à exploiter cette connaissance pour permettre à l'utilisateur de retrouver rapidement un fichier. SFS propose l'idée de voir non pas le répertoire comme un moyen de *navigation* mais au contraire comme un moyen d'*interrogation*[4]. Le nom du répertoire est dans SFS le moyen qu'a l'utilisateur pour indiquer sa *requête* au système et le / est le moyen d'exprimer une *conjonction*, permettant ainsi de combiner plusieurs critères de recherche. SFS appelle ses répertoires des *répertoires virtuels*. En effet,

[4] Nous définirons plus loin dans cette thèse Section 2.1.1 plus précisément les principes de navigation et d'interrogation ainsi que les avantages et inconvénients de chacun de ces deux modes de recherche d'information.

leur contenu n'est pas déterminé par la lecture physique d'une partie du disque mais résulte d'un véritable *calcul*. Ainsi, sous SFS l'utilisateur peut formuler la requête `cd /ext:c/owner:foo/function:main/` pour récupérer l'ensemble des fichiers C possédant une fonction `main` et appartenant à l'utilisateur `foo`. Le contenu de ce répertoire contiendra des liens qui pointeront vers les fichiers adéquats. En effet, SFS n'est pas un système de fichier complet. SFS vient se greffer sur un système de fichier existant dont il indexe les fichiers grâce aux transducteurs, par exemple une fois par jour. Ces fichiers peuvent ensuite être accédés soit de manière normale avec le système de fichier existant, par exemple en accédant au fichier `/home/foo/projet/bar.c`, soit par le système des répertoires virtuels de SFS à partir d'un point de montage particulier comme `/sfs`. Ainsi, les répertoires virtuels `/sfs/ext:c/owner:foo/` et `/sfs/function:foo/ext:c/` contiendront tous deux un lien symbolique `bar.c`, créé bien sûr automatiquement par SFS *à la volée*, et qui pointera vers le fichier du système de fichier existant.

Un exemple de suite de commandes *shell* utilisant SFS est présenté ci-dessous[5]. Le nombre de fichiers mises en jeu est petit pour des raisons didactiques. Cependant, c'est avec un grand nombre de fichiers que l'utilisation de SFS prend tous son sens. Ainsi, même si certaines commandes peuvent paraître inutiles dans l'exemple, il faut les imaginer dans un contexte plus large.

```
[1] % cd /sfs
[2] % ls
foo.c@ bar.c@ bio.txt@
paper.tex@ prop.tex@
toto.exe@ xv.doc@
compil.o@
...
[3] % ls FIELD:/
ext:/ name:/ dir:/ owner:/ date:/
function:/ subject:/ from:/
[4] % cd ext:/;  ls
tex/ c/ txt/ doc/ o/ exe/
[5] % cd c/; ls
foo.c@ bar.c@
[6] % ls -l
foo.c  --> /home/pad/src/foo.c
bar.c  --> /home/jones/bar.c
[7] % cd owner:/jones/; ls
bar.c@
[8] % cd /sfs/owner:/; ls
jones/ pad/ smith/
[9] cd jones/; ls
bio.txt@ paper.tex@ prop.tex@ bar.c@
```

[5] Le symbole ';' permet d'exécuter une sequence de commandes. Le symbole @ indique la présence d'un lien symbolique.

Contrairement aux systèmes de fichier hiérarchiques, SFS supporte de nombreuses classifications et permet donc des recherches selon de nombreux critères, dans n'importe quel ordre.

On peut voir en partie SFS comme l'intégration des utilitaires `find` et `grep` dans le monde des répertoires. Les transducteurs permettent de plus de combler certaines déficiences de ces outils en structurant un peu plus les fichiers, en passant d'un formalisme basé sur des mots-clefs à un formalisme basé sur des attributs valués, plus précis. SFS est de plus facilement extensible ; si l'utilisateur veut formuler de nouvelles requêtes sur un nouveau genre de propriété, ou si un nouveau type de fichier apparaît, il suffit de créer le transducteur adéquat. Avec `find` et `grep` le nombre de ces propriétés était fixe.

Là encore, l'intégration d'un service dans le système de fichier rend le système plus simple. En effet, la requête SFS `cd /sfs/ext:c/owner:toto/function:main/` est plus homogène et donc plus simple pour un utilisateur que la commande *shell* correspondante `find -name "*.c" -and -user toto | xargs grep -l main`.

Là encore, l'intégration d'un service dans le système de fichier a un impact bénéfique sur toutes les applications du système qui bénéficient désormais d'un *moteur de recherche*. SFS permet d'une certaine manière à toute les applications de bénéficier facilement de services offerts traditionnellement par un logiciel de *base de données*.

Cependant, cette intégration n'est pas vraiment complète. En effet, certains services offerts normalement par un système de fichier ne sont pas accessibles sous SFS et dans les répertoires virtuels. Il n'est par exemple pas possible d'effacer ou de créer un fichier dans ces répertoires virtuels. Il n'est donc pas possible d'associer manuelle-

ment des propriétés à un fichier comme par exemple le nom du projet auquel appartient ce fichier. De plus, suite à la commande `8` si un utilisateur se plaçait dans le répertoire `/sfs/owner:/jones/ext:/`, SFS proposerait comme répertoires l'ensemble de toutes les extensions possibles comme par exemple `exe` alors qu'aucun des fichiers de `jones` n'a ce format. SFS propose donc des répertoires pouvant mener à une impasse ne contenant aucun fichier. Il serait préférable d'élaguer ces répertoires non «pertinents». Cela nécessite cependant un outillage algorithmique important pour rendre cette opération rapide. Nous verrons plus loin Section 3.1.2 comment LFS sur un problème encore plus général résoud ce problème efficacement.

1.4.2 Content Addressable Typed File System

Le Content Addressable Typed File System (CATFS) [Gia93] reprend certaines idées de SFS pour en faire un système de fichier complet, qui se suffit à lui même. Il décompose un fichier en deux parties, une partie *donnée*, classique, et une partie *méta-donnée* contenant donc des informations sur l'information avec des propriétés sous la forme d'attributs valués. Contrairement à SFS, l'utilisateur peut cette fois assigner manuellement des propriétés à un fichier. CATFS ne supporte cependant pas l'idée de transducteurs associés automatiquement à un type de fichier mais permet tout de même à des outils d'associer aussi des propriétés à un fichier, puisqu'il le permet pour l'utilisateur. CATFS est aussi basé sur l'interrogation avec la notion de répertoire virtuel représentant une requête qui détermine un ensemble de fichiers.

Cependant, l'utilisateur peut cette fois manipuler les fichiers présents dans ces répertoires. En effet, CATFS ne requiert pas la présence d'un système de fichier hôte, et les fichiers présents dans les répertoires virtuels ne sont pas des liens qui pointent vers un autre système. On peut donc effacer des fichiers dans ces répertoires. On peut aussi y créer des fichiers. La sémantique de la commande `touch /projet:foobar/avancement:bon/foo.c` est donc de créer un fichier `foo.c` et de lui ajouter les propriétés `projet:foobar` et `avancement:bon`. On peut ajuster ces propriétés, en rajouter ou effacer à l'aide des commandes spéciales `addattr` et `delattr`. C'est d'ailleurs ces commandes que peuvent utiliser des outils spéciaux, par exemple pour rajouter au fichier `test.c` la propriété `ext:c` avec la commande `addattr test.c ext:c`.

CATFS supporte en plus des requêtes conjonctives avec le `/` les requêtes *disjonctives* et *négatives* avec respectivement le `|` et le `!`. La conjonction de critères un peu vague permet paradoxalement souvent de délimiter un ensemble de fichier petit. L'utilisateur peut donc exécuter la commande `cd /projet:toto|projet:tata/ext !=c/`.

CATFS propose de plus une certaine forme d'intégration entre la navigation et l'interrogation par l'intermédiaire de la propriété spéciale `ATTR:`. Par exemple, l'exécution de la commande `ls ATTR:` dans le répertoire `/projet:toto/ext !=c` va retourner l'ensemble des propriétés que possèdent les fichiers satisfaisant cette requête, et uniquement celles-la, ainsi que le nombre de fichiers ayant cette propriété, par exemple `ext:h=12 ext:txt=4 projet:toto=16`. Un peu comme avec la navigation dans les systèmes hiérarchiques, la machine aide l'utilisateur en lui proposant des *compléments de requêtes* possibles.

Cependant, l'exécution de cette commande à la racine proposera l'ensemble de tous les attributs possibles. Il faudrait structurer aussi cette réponse car sinon l'utilisateur sera submergé par un ensemble très grand de propriétés comme il l'avait été par un ensemble très grand de fichiers sous SFS.

CATFS propose une autre forme de navigation avec la notion de *fichier-vue*, par l'intermédiaire de la commande spéciale `mkview` qui résoud partiellement le problème précédent. Ces fichiers-vues sont des fichiers spéciaux suffixés par le symbole spécial `':'` et ils offrent un moyen pour l'utilisateur de matérialiser certaines requêtes. Le principe est de ne créer qu'un nombre réduit de fichiers-vues, afin de mémoriser les requêtes les plus intéressantes, les plus utilisées. Contrairement aux répertoires, si l'on crée un nouveau fichier celui-ci sera automatiquement «classé». L'utilisateur crée en fait des sortes de classifieurs, qui sont dynamiques, et surtout visibles. Un fichier-vue représente une sorte de porte-drapeau pour un ensemble de fichiers normaux. Au lieu qu'une requête et donc un répertoire contienne des mil-

liers de fichiers, ce qui peut être pénible à parcourir, les fichiers seront regroupés et résumés par leurs portes-drapeau. De plus, un fichier peut être «résumé» par différents fichiers-vues, et donc appartenir à différents regroupements en même temps.

Un exemple de suite de commandes *shell* utilisant CATFS est présenté ci-dessous :

```
[1] % cd /catfs
[2] % ls
foo.c bar.c foo.txt foo.o
bio.tex biotype.txt
paper.tex prop.tex
toto.exe xv.doc
compil.o zo.c zi.c
...
[3] % cd projet:foobar/; ls
foo.c bar.c foo.txt foo.o
[4] % touch makefile; ls
foo.c bar.c foo.txt foo.o makefile
[5] % addattr makefile avancement:debut
[6] % cd avancement:debut/; ls
makefile foo.txt
[7] % cd /catfs/avancement:debut/
        ext!=c/; ls
makefile foo.txt biotype.txt bio.tex
[8] % ls ATTR:
ext:txt=2 ext:tex=1
projet:foobar=2 projet:biology=2
avancement:debut=4
[9] % cd projet:biology/; ls
biotype.txt bio.tex
[10] % cd /catfs/projet:foobar/ext:c/
[11] % mkview Foobar-src:
[12] % cd /catfs; ls
Foobar-src:
foo.txt
bio.tex biotype.txt
paper.tex prop.tex
toto.exe xv.doc
compil.o
...
[13] % cd /catfs/ext:c/; ls
Foobar-src:
zo.c zi.c
[14] % printview Foobar-src:
'projet:foobar&ext:c'
[15] % cd projet:foobar; ls
foo.c bar.c
```

Un problème important de CATFS est qu'il n'aborde tout de même pas la question de l'attribution automatique de propriétés à un fichier. En effet, il suppose l'existence d'outils, un peu comme les transducteurs, qui ajoutent de temps en temps des propriétés aux fichiers. Cependant, on voit bien que la modification d'un fichier, par exemple le renommage d'un fichier `.c` en `.h` aura pour effet d'enlever la propriété `ext:c` pour rajouter la propriété `ext:h`. Il n'est pas clair qu'un outil sache quelles propriétés enlever car il ne faut pas toucher aux propriétés attribuées manuellement par l'utilisateur au fichier. La différence entre ces deux types de propriétés requiert un support interne de CATFS, qui justement n'est pas fourni. Nous verrons plus loin Section 2.3.1 page 73 et Section 2.4.2 page 80 comment LFS résoud ce problème.

1.4.3 Nebula

Le système de fichier Nebula [BDB+94] développe de manière indépendante certaines idées de CATFS. Il permet aussi des requêtes conjonctives, disjonctives et négatives, et même des *opérations relationnelles* sur les valeurs d'attributs comme dans `cd size:>100/` ou des *expressions régulières* (*regexp*) comme dans `cd projet:.*to.*/`.

Nebula propose aussi un système de vue, mais différent de celui de CATFS, avec cette fois des *répertoires-vues*. L'utilisateur peut créer une vue, qui se présente cette fois sous la forme d'un répertoire, lui donner un nom et lui associer une requête. L'utilisateur peut aussi créer des sous-vues en créant des sous-répertoires à l'intérieur d'un répertoire-vue, qui représenteront ainsi des requêtes plus précises. Les vues sont donc liées entre elles, et les fichiers présents dans une sous-vue sont ceux qui satisfont la requête de cette vue mais aussi la requête du parent de cette vue et ainsi de suite. Cela permet à l'utilisateur de superposer une structure hiérarchique sur une base d'information décrite par des attributs valués, et de naviguer sur un système basé surtout sur l'interrogation. L'utilisateur choisit ainsi de matérialiser certaines requêtes, de leur donner un nom, et même de

les classer entre elles. L'utilisateur peut choisir de naviguer de requêtes en requêtes en suivant les répertoires, ou bien d'interroger ce qui le placera dans un répertoire «temporaire» contenant les fichiers satisfaisant cette requête. Il peut décider de rendre ce répertoire concret par le biais d'un répertoire-vue et de classer ensuite les fichiers satisfaisant ce répertoire en créant de nouveaux sous-répertoires avec des requêtes plus précises. Comme pour CATFS, cette structure est très dynamique, et donc contrairement aux répertoires classiques, la modification des propriétés (ajout, supression) du fichier entraînera automatiquement la reclassification du fichier dans la hiérarchie des vues. C'est vrai aussi pour la création de nouveaux fichiers. De même, l'utilisateur peut décider de changer les requêtes associées aux répertoires, ce qui induira là aussi une reclassification automatique des fichiers.

Un exemple de suite de commandes *shell* utilisant Nebula est présenté ci-dessous. Il repart de l'étape 10 de l'exemple de CATFS :

```
[10] % cd /nebula; ls
foo.c bar.c foo.txt foo.o
makefile
bio.tex biotype.txt
paper.tex prop.tex
toto.exe xv.doc
compil.o zo.c zi.c
...
[11] % mkview documentation
         'ext:txt|ext:doc|ext:tex'
[12] % mkview work
         'projet:foobar|projet:biology'
[13] % ls
documentation/
work/
toto.exe
compil.o zo.c zi.c
...
[14] % ls documentation/
foo.txt bio.tex biotype.txt
paper.tex prop.tex xv.doc
[15] % ls work/
foo.txt foo.c bar.c foo.o makefile
bio.tex biotype.txt
[16] % cd work; mkview src 'ext:c'; ls
src/
foo.txt foo.o makefile
bio.tex biotype.txt
[17] % cd /nebula; ls
documentation/
work/
toto.exe
compil.o zo.c zi.c
...
[18] % cd name:z*|ext:exe/; ls
zo.c zi.c toto.exe
[19] % cd /nebula/
[20] % touch ext:txt/projet:biology/
          newbio.txt
[21] % ls documentation/
newbio.txt
foo.txt bio.tex biotype.txt
paper.tex prop.tex xv.doc
```

Cependant, même si Nebula répond à certains problèmes des systèmes hiérarchiques, ainsi même si cette fois un fichier peut avoir différentes propriétés, faire partie de plusieurs vues en même temps, même si l'utilisateur peut interroger et combiner différents critères qui utilisent ces propriétés, même si il peut naviguer, il ne peut naviguer qu'à travers les répertoires qui auront été créés manuellement par l'utilisateur. Le jour où celui-ci lancera une requête complexe qui n'a encore jamais été faite, le système ne lui proposera aucun répertoire pour affiner sa recherche, et ce même si cette requête est proche voir équivalente à la requête d'une vue déjà construite. On ne peut donc pas réutiliser le travail de classification et de regroupement qui aura déjà été fait sur d'autres vues. On retombe en fait sur les mêmes problèmes des hiérarchies puisque l'utilisateur aura fait là encore le choix d'une hiérarchie, d'un chemin à suivre. Si l'utilisateur s'en écarte, il se retrouvera perdu. Nous verrons plus loin Section 2.3.1 page 69 et Section 2.5.2 page 81 comment LFS résoud ce problème.

De plus, comme pour CATFS, Nebula ne résoud pas la question de l'attribution automatique de propriétés à un fichier.

1.4.4 Hierarchy And Content

Le but du système Hierarchy and Content (HAC) [GM99] est de mélanger les avantages des systèmes classiques avec ses hiérarchies et classifications manuelles, et des systèmes d'interrogation basés sur le contenu comme SFS avec classification automatique. HAC, comme SFS, nécessite un système hôte, et les requêtes HAC retournent aussi un ensemble de liens vers les fichiers qui satisfont la requête. Un peu comme sous Nebula, l'utilisateur peut créer des répertoires et sous-répertoires avec des noms concrets correspondant à des requêtes (des répertoires vues). Ces répertoires seront visibles pour les futures navigations. Cependant, sous HAC l'utilisateur peut effacer ou ajouter des liens dans ces répertoires. L'effacement d'un lien permet d'ajuster la requête de ce répertoire, mais n'efface pas ce fichier du disque. En fait, on peut voir la création d'un répertoire-requête comme un premier jet, où le nom de ce répertoire correspond à une notion abstraite, représentée approximativement par une requête. Le système aide l'utilisateur en créant automatiquement des liens correspondant à cette requête. L'utilisateur peut ensuite «corriger» ce premier jet pour prendre en compte les anomalies ; ex. un fichier satisfaisant a priori cette notion même si il ne satisfait pas la requête, ou inversement un fichier satisfaisant la requête alors qu'il ne satisfait pas la notion. Cette méthode peut s'avérer plus rapide que d'avoir à chercher une requête plus précise correspondant mieux à cette notion abstraite. Il est aussi possible de créer des répertoires normaux ne correspondant pas à des requêtes, et donc de créer des taxinomies et d'y placer des liens. On peut aussi raffiner ces répertoires en créant cette fois des répertoires correspondant à des requêtes. En fait, on peut mélanger à volonté les répertoires classiques et les répertoires avancés. Là encore, à sa création, un nouveau fichier sera placé automatiquement dans le bon, voir les bons répertoires avancés.

Un point important est que l'utilisateur peut modifier la requête associée à un répertoire avancé. Son contenu devrait être donc réactualisé, ce qui pourrait anéantir les ajustements manuels faits par l'utilisateur précédemment. De même, l'effacement d'un lien dans un répertoire devrait avoir une conséquence sur le contenu de ces sous-répertoires. HAC gère ces problèmes d'inconsistance en se rappelant pour chaque répertoire avancé les liens qui auront été ajoutés par l'utilisateur, et les liens qui auront été effacés. La modification de la requête d'un répertoire recalcule son contenu et réapplique les actions faites par l'utilisateur précédemment.

Un exemple de suite de commandes *shell* utilisant HAC est présenté ci-dessous.

```
[1] % cd /hac
[2] % ls
foo.c@ bar.c@ bio.txt@
paper.tex@ prop.tex@
toto.exe@ xv.doc@
compil.o@
...
[3] % mkdirAdv foobar-project:
          'name:foo'; ls
foobar-project:/
bar.c@ bio.txt@
...
[4] % cd foobar-project:/; ls
foo.c@ foo.txt@
[5] % mv ../bar.c .; ls
foo.c@ bar.c@ foo.txt@
[6] % mkdir src
[7] % mv *.c src/; ls
src/ foo.txt@
[8] % ls -l
src/
foo.txt --> /home/pad/proj/foo.txt
[9] % touch /home/pad/proj/newfoo.doc
[10] % reindex_HAC
[11] % cd /hac/foobar-project:/
ls -l
src/
foo.txt --> /home/pad/proj/foo.txt
newfoo.doc --> /home/pad/proj/newfoo.doc
```

Même si HAC permet à l'utilisateur de bénéficier de nouveaux services tout en gardant la possibilité d'utiliser ses anciens réflexes d'organisation, et donc de fournir une transition plus douce vers un moyen d'organisation plus puissant, HAC reste tout de même très limité. Comme pour tous les autres systèmes, la création d'un nouveau répertoire avec une nouvelle requête ne retournera qu'un ensemble de liens. L'utilisateur ne pourra pas

naviguer. Il lui faudra recréer à partir de ce répertoire une nouvelle classification, alors qu'il se peut que cette requête soit au fond proche d'autres répertoires qui auront été déjà classés.

1.4.5 Discussions

Ces systèmes de fichiers avancés apportent tous des services supplémentaires appréciables pour l'organisation et la recherche de documents. Cependant, aucun ne résoud vraiment par exemple le problème de la modélisation d'une librairie scientifique avec de multiples taxinomies, par domaines et sous-domaines, par éditeurs, par année, et d'un support pour une recherche utilisant à bon escient l'ensemble de ces taxinomies pour trouver rapidement un livre. Ainsi, aucun de ces systèmes de fichier anvancé n'est meilleur qu'un autre ni même meilleur qu'un système de fichier hiérarchique ; ils ont chacun leurs avantages. Chacun de ces travaux apportent une «brique» mais à chaque fois dans une direction différente. Il persiste donc de nombreuses limitations dans chacun de ces systèmes. Ainsi, des systèmes (système de fichier classique, CATFS, HAC) vont permettre d'associer manuellement des propriétés à un fichier mais vont rendre plus difficile son indexation automatique, ou l'inverse (SFS, Nebula). De même, un système va permettre d'interroger après une navigation mais pas l'inverse. Ces systèmes possèdent donc de nombreuses dissymétries dans leurs fonctionnalités. Nous verrons plus loin Chapitre 2 comment LFS résoud ces problèmes et permet à l'utilisateur de combiner sans restriction ces différentes fonctionnalités.

Il faut noter que même si ces systèmes de fichier avancés ne sont pas connus du grand public, certaines de leurs «briques» ont été incorporées et même combinées dans des systèmes de fichier de systèmes d'exploitation grand public. Ainsi, le système de fichier BeFS [Gia99] du système d'exploitation BeOS offre à la fois un système de fichier hiérarchique, et par le biais d'une API spécifique des services proches de SFS. De même, le futur système de fichier SpotLight [App04] du futur MacOS offre un système de fichier hiérarchique, une indexation supplémentaire automatique à la SFS exploitable par une API spécifique ainsi que la possibilité de créer des répertoires-vues à la Nebula. Le principe de répertoire-vue est aussi présent dans certains logiciels comme les logiciels de messagerie, sous une forme et un nom différent comme le principe de boîte virtuelle. Enfin, le futur système de fichier LongHorn [Mic06] du futur Windows, même si encore à l'état de *vaporware*, est annoncé comme un système de fichier offrant les mêmes genres de services que Nebula. Tout ceci montre que l'organisation et la recherche de fichier, et l'amélioration des systèmes de fichier hiérarchiques sont des sujets très chauds. On peut ainsi s'attendre à ce que LFS soit présent dans Windows aux environs 2030.

Il existe une autre catégorie de systèmes de fichier avancés que l'on n'a pas évoqué jusqu'à présent : les systèmes de fichier liés au temps (3DFS [Roo92], Elephant [SFHV99]). Un peu comme les travaux précédents brisent la *contrainte spatiale*, qui fait qu'un objet dans la réalité ne peut se trouver qu'à un seul endroit, il existe des travaux qui brisent la *contrainte temporelle*, qui fait qu'il n'est pas possible dans la réalité de revenir en arrière. Ainsi, l'un des systèmes de fichier fourni par Plan9 [PPD+95] permet à l'utilisateur de lancer une requête comme `cd /time/1995/03-15/` pour se placer à la racine du système tel qu'il était le 15 mars 1995. Cela permet par exemple de trouver rapidement quand un bug fut fixé simplement avec la commande `grep 'mouse bug fix' 1995/*/src/foo.c`. Un peu comme SFS permet d'intégrer `find` et `grep` directement dans le système de fichier, les systèmes de fichier liés au temps permettent d'intégrer un outil comme RCS (*Revision Control System*) [Tic85]. Là encore, on voit l'intérêt de placer un service au plus bas niveau, ici la gestion des versions d'un fichier, puisque cela rend ensuite possible des combinaisons avec des outils comme `grep` pour résoudre rapidement un problème. Cependant, comme il est très facile de copier un fichier dans le monde virtuel contrairement à la réalité, ces systèmes de fichier avancés sont surtout un moyen pour rendre plus transparentes et efficaces les procédures de sauvegarde. Ils ne changent pas comme SFS fondamentalement l'organisation des données.

1.5 Sécurité

Le besoin de sécurité dans les systèmes de fichiers se fait sentir dès que plusieurs utilisateurs peuvent avoir ac-

cès au même fichier. L'intégration du réseau avec NFS a encore augmenté un peu plus le nombre des utilisateurs potentiels ayant accès à un même fichier. Le premier mécanisme de sécurité d'un système d'exploitation est celui du *login* et du *mot de passe* qui permet à un utilisateur de s'identifier auprès de la machine. Les fichiers créés ensuite par cet utilisateur lui appartiennent, et par défaut lui seul peut accéder et modifier ces fichiers.

Si les utilisateurs ne voulaient pas accéder aux données des autres, ni partager les leurs, la sécurité dans les systèmes de fichier serait relativement simple. Cependant ce n'est pas le cas. Le responsable de la machine, l'administrateur appelé aussi le *root*, veut déjà permettre à ses utilisateurs d'accéder aux programmes installés sur cette machine et d'accéder à certaines données comme la documentation du système, mais pas à d'autres. Un utilisateur voudrait aussi par exemple permettre à un autre de lire un fichier qu'il a composé [6]. En fait, en plus de l'accès en *lecture* à ces données, on aimerait parfois même donner l'accès en *écriture*. En effet, un autre intérêt des systèmes multi-utilisateurs est de permettre le *travail coopératif*. Ainsi, on aimerait que plusieurs utilisateurs puissent travailler et donc modifier le même programme, le même document. Ainsi, un autre rôle du système de fichier, en plus de permettre de stocker, organiser voir même comme on l'a vu précédemment de chercher et manipuler des données, est d'offrir les moyens à un utilisateur à la fois de *partager* et de *protéger* ses données.

Dans le monde physique, on règle ces besoins opposés de protection et de partage en plaçant par exemple des données dans des tiroirs fermés à clef pour les protéger physiquement. L'emplacement de cette clef est ensuite gardé secret, ou peut être partagé. Le système de mot de passe du *login* est aussi une forme de clef. Comme dans la réalité, une manière de partager tout en protégeant ses données est donc de confier cette clef à des personnes de confiance.

Cependant, dans un monde virtuel on attend mieux, comme un contrôle plus fin permettant par exemple à certains utilisateurs de voir uniquement le contenu, mais pas de le modifier. De plus, dans la réalité on peut avoir différents tiroirs et différentes clefs, ce qui n'est pas le cas avec le mot de passe d'un utilisateur.

[6] ou récupéré sur le Web.

Nous allons dans les sections qui suivent présenter deux *modèles de sécurité* pour les systèmes de fichier qui formalisent ces notions de partage et de protection. Ces deux modèles sont les plus connus et sont présents dans deux systèmes d'exploitation populaires, celui d'Unix avec la notion de *groupe*, et celui de Windows NT avec la notion de *liste de contrôle d'accès* (ACL). En fait, le principe des ACL est désormais présents dans de nombreux systèmes, y compris dans les Unix modernes comme Linux. Le modèle Unix est sans doute le plus connu des deux, il est d'un certain point de vue plus pratique que celui des ACL mais il est aussi moins expressif. Paradoxalement le principe des ACL, plus expressif, fut inventé en premier sous Multics, puis connut une période où on lui préféra le modèle Unix. Il connaît de nos jours une renaissance, sans doute à cause de l'augmentation du nombre des utilisateurs et donc d'un besoin de contrôle d'accès plus fin.

Cependant, ces moyens de sécurité sont logiciels, et un utilisateur pourrait accéder de manière physique et directe au disque pour shunter cette couche de protection. L'intégration du réseau augmente encore un peu plus les failles puisqu'on pourrait intercepter des données sur le réseau. Nous verrons pour finir des moyens de protection plus «durs» répondant à ces problèmes avec l'intégration des principes de la *cryptographie* dans le système de fichier.

1.5.1 Le modèle Unix

Sous Unix, chaque fichier et répertoire a un *propriétaire*, l'*identité* de la personne qui a créé ce fichier, l'*uid* (*User IDentity*). Chaque fichier a aussi un *groupe*, le groupe dont fait partie la personne qui a créé ce fichier, le *gid* (*Group IDentity*). En effet, sous Unix les utilisateurs sont regroupés dans différents groupes par l'administrateur, par exemple le groupe `users` regroupe tous les utilisateurs basiques de la machine, le groupe `admin` tous les utilisateurs un peu plus dignes de confiance et qui auront ainsi des droits pour modifier certaines données systèmes. Le système de sécurité d'Unix va ensuite exploiter ces groupes afin de permettre à un utilisateur de donner l'accès à ses données à tous les membres de son groupe, mais pas aux autres.

Ainsi, chaque fichier et répertoire possède comme propriétés, en plus du nom du propriétaire et du groupe, 3

droits d'accès :

1. le droit pour le *propriétaire*, noté `user` en anglais ou en abrégé `u`.
2. le droit pour le *groupe*, noté `group` en anglais ou en abrégé `g`.
3. le droit pour les *autres* qui sont les personnes qui ne sont ni le propriétaire, ni membre du groupe du fichier : `other` en anglais ou en abrégé `o`.

Chaque droit est ensuite lui même formé de 3 booléens :

1. le droit de *lire* le contenu du fichier ou du répertoire : `read` ou `r`.
2. le droit d'*écrire* dans ce fichier ou répertoire : `write` ou `w`.
3. le droit d'*exécuter* le fichier ou de rentrer dans le répertoire : `execute` ou `x`.

En plus des commandes `cd` ou `ls` vues Section 1.2.2 page 12, l'arsenal des commandes pour manipuler un système de fichier s'enrichit donc des commandes `chown`, `chgrp`, et `chmod` pour respectivement changer le propriétaire, le groupe et les droits d'un fichier. Ainsi, la commande `chmod g+rw foo.c` permet de rajouter (`+`) au droit du groupe (`g`) la possibilité de lire et d'écrire (`rw`) dans le fichier `foo.c`. En fait, seul le *root* peut utiliser la commande `chown`, et seul le propriétaire du fichier peut changer les droits de ce fichier et donc lancer la commande `chmod` sur ce fichier. Un utilisateur peut faire partie de plusieurs groupes en même temps, et donc seul aussi le propriétaire du fichier peut exécuter la commande `chgrp` sur ce fichier, et uniquement pour changer pour un groupe dont il fait aussi partie.

L'utilisateur peut consulter les droits d'un fichier à l'aide de la commande `ls -l` qui permet de voir l'ensemble des attributs systèmes d'un fichier.

```
[1] % ls -l main.tex
-rw-r--r-- 1 pad user 750 jun 2 foo.txt
```

Le résultat de cette commande indique que le fichier `foo.txt` est possédé par l'utilisateur `pad` du groupe `user`, avec des droits en lecture et écriture, lecture seule, et lecture seule pour respectivement le propriétaire, le groupe, et les autres utilisateurs.

L'accès en lecture ou écriture sur un fichier ou répertoire par un utilisateur est conditionné par ces propriétés et vérifié par le système d'exploitation. Par exemple, si l'utilisateur voulant accéder à un fichier fait partie du groupe de ce fichier, ce sont les droits du groupe qui seront pris en compte, sinon ce sont les droits de `other`.

Les droits sur un répertoire, la lecture, la modification, et l'exécution, correspondent respectivement à la possibilité de voir le contenu de ce répertoire, c'est-à-dire de voir le nom des fichiers et sous-répertoires présents dans ce répertoire, la possibilité de modifier ce contenu, c'est à dire d'effacer ou de créer des fichiers ou sous-répertoires, et finalement la possibilité de rentrer dans ce répertoire. Cette dernière possibilité peut paraître un peu redondante avec le fait de voir son contenu, mais en fait il est possible de rentrer dans un répertoire sans pour autant voir son contenu ; si l'utilisateur connaît précisément le nom d'un fichier ou sous-répertoire, il pourra y accéder. Cela peut offrir un mécanisme rudimentaire pour donner l'accès à un fichier à une seule personne en se servant du nom du fichier comme un mot de passe. Cela reste cependant assez anecdotique et la propriété d'exécution reste moins utile que celle de lecture ou de modification. Il en est de même pour le fichier. Il est rare que l'on veuille moduler les droits d'exécution d'un programme. En effet, une fois lancé, le processus aura de toute façon comme propriétaire l'identité de l'utilisateur qui l'a lancé et sera donc soumis aux mêmes contraintes que cet utilisateur. Il sert donc plus comme une sorte de *tag* que comme une propriété de sécurité.

Même si ce modèle peut paraître très simple à comprendre et à utiliser, certaines combinaisons de propriétés peuvent amener à des comportements étranges, et finalement pas très logiques. Ainsi, il faut noter que le droit sur un fichier est en fait le droit sur le contenu de ce fichier. Les droits du répertoire contenant ce fichier peuvent donc aussi influer sur ce fichier. Ainsi, même si le fichier n'est pas modifiable, il peut être effacé si le répertoire autorise la modification. De même, le nom du fichier peut être lisible même si son contenu ne l'est pas. Dans le même genre d'usage exotique des propriétés d'un fichier, il est possible qu'un fichier ait la propriété de modification sans avoir celle de lecture. On peut donc écrire dans ce fichier et écraser son contenu mais pas le lire. De même, un fichier peut être exécutable mais pas lisible. Cela peut permettre de rendre un programme

exécutable sans pour autant pouvoir lire son code, par exemple pour le décompiler afin de trouver une faille dans ce programme. Avec ce schéma, il est même possible qu'un fichier donne plus de droits au groupe qu'à son propriétaire Ainsi, ce modèle offre de nombreuses combinaisons de propriété, mais beaucoup restent peu utiles et les propriétés essentielles restent donc la possibilité de lire et d'écrire dans un fichier ou répertoire.

Certaines extensions à ces propriétés ont vu le jour au cours des années. Le *sticky* bit (`chmod +t`) permet lorsqu'il est appliqué à un répertoire de n'autoriser l'effacement d'un fichier que par son propriétaire. On l'utilise en général pour le répertoire `/tmp`. En effet, on voudrait que tout le monde puisse écrire dans ce répertoire, ce qui induirait donc de lui donner la propriété `w` pour *other*. Cependant, cela voudrait dire aussi que tout le monde pourrait détruire les fichiers de tout le monde. Le *sticky* bit résoud ce problème.

Un autre extension est le *setuid* bit (`chmod +s`). Il permet lorsqu'il est appliqué à un fichier exécutable, que le lancement de ce programme prenne non pas l'identité de celui qui l'a lancé, mais celui du propriétaire du fichier. C'est en général l'administrateur de la machine, le *root* qui utilise cette propriété. En effet, on aimerait par exemple permettre à l'utilisateur de modifier certaines informations systèmes comme son mot de passe, lui permettre d'accéder à sa disquette ou à un CDROM et donc de monter certains périphériques. Dans chacun de ces cas, cela impliquerait de donner l'accès en écriture aux utilisateurs à certains fichiers, un fichier normal comme `/etc/passwd` ou un fichier périphérique comme `/dev/floppy`. Cela serait dangereux. Au lieu de cela, on donne sa confiance non pas à un utilisateur mais à un programme. Ainsi, le programme `passwd` avec comme propriétaire *root* possède ce bit spécial *setuid*, et lorsque l'utilisateur lance ce programme, il prend donc temporairement l'identité *root* et peut donc modifier le fichier `/etc/passwd`. Ces programmes appliquent souvent eux mêmes une politique de sécurité avant d'exécuter la requête d'un utilisateur. En effet, le système fournit à ces processus le moyen de savoir quelle est la véritable identité de l'utilisateur ayant lancé ce processus. Ainsi, le programme `passwd` va s'assurer que l'utilisateur ne puisse pas modifier le mot de passe d'un autre utilisateur. On peut donc voir le bit *setuid* comme un moyen d'étendre le modèle de sécurité d'Unix afin de s'adapter aux besoins particuliers des applications. Ils sont cependant une source de faille de sécurité importante. Il existe aussi un bit spécial *setgid* qui permet cette fois de faire croire au système que l'utilisateur qui lance le processus fait partie de tel ou tel groupe. Ainsi, plutôt que de confier à *root* l'intégralité des données systèmes, on classe ces données en les attribuant à différents groupes systèmes, par exemple `mail`, `man`, et certains programmes se voient ensuite attribuer le bit *setgid* pour pouvoir ensuite accéder et modifier ces données. Cela permet de réduire un peu les failles de sécurité en compartimentant le système.

1.5.2 Liste de contrôle d'accès

Le modèle Unix offre certes des moyens de protections importants, mais il n'offre pas des moyens de partage très évolués. En effet, puisqu'un utilisateur ne peut pas donner des droits à un utilisateur particulier, celui-ci préfère ne pas partager ce fichier plutôt que de prendre le risque de donner les droits au groupe entier. Le grain du groupe est trop grossier. Le modèle de sécurité doit être suffisamment expressif et précis pour permettre le travail coopératif.

On peut voir le problème de sécurité comme une matrice avec pour les lignes les fichiers, et pour les colonnes les utilisateurs. Le principe des *listes de contrôle d'accès* (ACL) est de permettre à l'utilisateur de remplir cette matrice en mentionnant pour chaque fichier la liste des utilisateurs ayant des droits sur ce fichier. Si un utilisateur n'est pas mentionné, cela veut dire implicitement que cet utilisateur n'a aucun droit.

Cependant, il serait fastidieux d'émuler les groupes avec les ACL. En effet, si l'on voulait donner au moins l'accès en lecture sur un fichier à tous les utilisateurs, ou à tout un groupe, il faudrait avec les ACL tous les mentionner. Les groupes ont donc tout de même certains avantages. Ainsi, il est aussi possible, par exemple sous NT, de mentionner dans les listes de contrôles d'accès des noms de groupes avec les droits associés. On peut même cette fois en plus donner une liste de groupes. Certains systèmes autorisent aussi l'utilisation d'expression régulière, par exemple `*:r` pour donner l'accès à tous en lecture.

Le modèle de sécurité de Windows NT étend dans une autre direction le modèle Unix en ajoutant aux droits de

lecture, d'écriture, et d'exécution, le droit de *changer les droits*. Ainsi, on pourrait permettre à d'autres utilisateurs d'utiliser la commande `chmod` sur son fichier. Cependant, ces droits avancés, comme pour le droit d'exécution, restent moins utiles.

Une limitation des ACL est qu'elle ne permettent que de donner de manière *positive* des droits à un fichier. On ne peut ainsi pas facilement exprimer le fait que l'on aimerait donner l'accès à un fichier à tous *sauf* à une personne. De plus, on ne peut explorer la matrice que par ligne. En effet, même si l'on peut voir l'ensemble des droits que possède un fichier, on ne peut pas voir l'ensemble des fichiers caractérisés par un droit. Nous verrons plus loin au Chapitre 4 le modèle de sécurité de LFS qui résoud ces problèmes en permettant d'exprimer la négation et en permettant de naviguer parmi les propriétés de sécurité en plus de permettre de naviguer parmi les fichiers. De plus le modèle LFS, en plus donc d'être plus expressif sur les moyens de protection permettra aussi d'améliorer les moyens de partage.

1.5.3 Cryptographie

Les modèles de sécurité précédents assurent certes une protection des fichiers, mais cela reste une protection virtuelle. En effet, un utilisateur malveillant ayant accès physiquement au disque dur stockant ces données pourrait shunter ce système de protection même si il n'avait pas le mot de passe de *root*. C'est d'autant plus vrai depuis l'apparition des portables que l'on peut perdre ou se faire voler. Il suffirait donc à cette personne malveillante de démonter physiquement de la machine ce disque pour le placer sur une autre machine dont il serait l'administrateur puis de monter ce disque sur cette nouvelle machine. Étant cette fois le *root*, il pourrait accéder à tous les fichiers présents sur ce disque. L'utilisation du réseau augmente encore un peu plus les possibilités de violation de confidentialité. En effet, les données d'un utilisateur, ses fichiers, ses e-mails, sont souvent stockées sur une machine distante de celle sur laquelle il se connecte, et l'accès à ces fichiers impliquera le transit sur le réseau du contenu de ces fichiers. Là encore, un utilisateur malveillant pourrait physiquement avoir accès au cable du réseau et observer et intercepter les données y transitant.

On peut difficilement empêcher ces accès physiques, à part en blindant les cables et les machines, mais on peut les rendre inoffensifs en *cryptant* les données. Il existe d'ailleurs des solutions matérielles comme des disques durs et des nœuds réseaux cryptés. Dans le cas du disque dur, l'utilisateur possède une clef matériel, contenant la *clé cryptographique*, qu'il connecte au disque dur pour pouvoir accéder en clair à ses fichiers. L'utilisateur peut ensuite laisser ce disque dur sur place sans risque (si il garde avec lui sa clef). Cette solution matérielle a l'intérêt de rendre le problème de cryptographie transparent. L'utilisateur continue comme auparavant à lire et écrire simplement dans des fichiers. Cependant, cette solution est coûteuse et n'est pas très flexible, comme c'est souvent le cas avec des solutions matérielles. En effet, le disque ne supporte qu'une seule clef à la fois, et donc qu'un seul utilisateur à la fois. L'intégralité du disque est cryptée de la même manière. Pour pouvoir supporter plusieurs utilisateurs, le matériel devrait connaître à quels utilisateurs correspondent les fichiers, ce qui est du domaine du système et non du matériel.

On pourrait bien sûr utiliser une solution logicielle comme PGP (*Pretty Good Privacy* [Zim95]) un service cryptographique à clef publique qui répond à ces problèmes. En effet, chaque utilisateur peut crypter à sa manière, avec sa propre clef, ses données. Cependant, cette fois l'utilisation de la cryptographie ne serait plus transparente, et rendrait plus pénibles les tâches quotidiennes. La manipulation d'un fichier forcerait d'abord l'utilisateur à décrypter ce fichier, à l'éditer, puis à le recrypter, et à effacer le fichier temporaire *clair* créé auparavant. Avec des centaines de fichiers cela deviendrait très vite trop fastidieux. Ainsi, on préfère souvent ne pas les utiliser, ou réserver la cryptographie aux fichiers les plus critiques, mais tout fichier est potentiellement critique. De plus, avec cette solution logicielle, on peut très facilement oublier une opération comme le recryptage après modification. La transparence et l'automatisation sont deux caractéristiques essentielles pour assurer l'efficacité des moyens cryptographiques. Il existe certaines applications qui gèrent automatiquement de manière transparente l'accès à des données cryptées, cependant elles restent peu nombreuses, et l'utilisateur préférerait utiliser les applications qu'il connaît déjà.

Là encore, une bonne idée, et un bon compromis entre la solution purement matérielle et purement logicielle, est de placer un service, ici la cryptographique, dans le

système de fichier. Sous CryptFS [Bla93], chaque utilisateur peut placer dans chacun de ses répertoires un fichier spécial qui contient une clef publique de cryptage. L'utilisateur peut ensuite monter ce répertoire dans un autre répertoire, en passant en paramètre la clef privé correspondante, et accéder de manière transparente à ses données cryptées. Le scénario suivant illustre un exemple de session :

```
[1] % cd /home/pad
[2] % echo ghjk56 > crypt_key.pub; ls
crypt_key.pub
[3] % mount /home/pad/   /decrypt
       -t cryptfs  -privkey xv56gh
[4] % cd /decrypt; ls
[5] % echo "message en clair" >
        confidential.txt
[6] % cat confidential.txt
message en clair
[7] % ls /home/pad/
crypt_key.pub confidential.txt
[8] % cat /home/pad/confidential.txt
degegrfkgrkgtk565243FSFDFGDG
```

La commande `1` crée un fichier spécial dans le répertoire `/home/pad`, et la commande `2` permet à l'utilisateur `pad` d'accéder aux données de son compte en clair à partir du point de montage `/decrypt`. Tous les fichiers qui seront créés, ou modifiés à partir de `/decrypt` seront automatiquement cryptés à l'aide de la *clef publique* `ghjk56`. Tous les fichiers qui seront accédés en lecture depuis `/decrypt` seront automatiquement décryptés à l'aide de la *clef privé* `xv56h`. Le répertoire `/decrypt` est une sorte de miroir virtuel du répertoire `/home/pad`, et ne contient pas à proprement parler de fichiers. L'utilisateur peut ainsi démonter le répertoire `/decrypt` pour ne laisser aucune trace (car la clef privé nécessaire au montage n'est connue que de lui). En fait, même le nom du fichier est crypté, ainsi le répertoire `/home/pad` ne contient pas un fichier `confidential.txt` mais un fichier avec un nom illisible.

Cette technique rend inoffensif l'accès physique au disque, puisque une fois le répertoire `/decrypt` démonté, le disque ne contiendra que des données cryptées et des clefs publiques. Elle rend aussi inoffensif l'accès physique au réseau. En effet, la combinaison de NFS et de CryptFS résoud gratuitement le problème. Le répertoire `/home/pad` pourrait être le résultat d'un montage NFS d'un répertoire distant. L'accès à un fichier de `/decrypt` impliquera donc l'accès au fichier miroir dans `/home/pad`, ce qui impliquera à son tour une requête sur le réseau et le transit du contenu de ce fichier. Cependant, les seules données qui seront transmises seront des données cryptées. Le décryptage se fera sur la machine locale de l'utilisateur, grâce à la clé privée passée en local à la commande `mount`.

Là encore, on voit bien qu'il est préférable d'offrir un service via le système de fichier plutôt que par une application particulière. La cryptographie est ainsi transparente pour l'utilisateur. Les éditeurs de texte, compilateurs ne sont même pas au courant. C'est aussi plus simple pour l'utilisateur, et plus sûr puisque toutes les tâches sont automatisées. Enfin, cette intégration a plus d'impact et permet des combinaisons intéressantes avec les autres services comme nous l'avons vu précédemment avec NFS.

Un service proche de la cryptographie est celui de la *compression*. Dans les deux cas, l'utilisateur peut utiliser des applications particulières, `pgp` ou `gzip`, dans le premier cas pour protéger ses données, dans le deuxième pour gagner de la place sur le disque. Cependant, dans ce deuxième cas aussi, l'utilisation d'une application particulière est pénible, et la compression est là aussi gérée par peu d'applications pour pouvoir accéder aux données de manière transparente. Ainsi, comme CryptFS, GzipFS [CG91] offre le service de compression automatiquement via le système de fichier. Il n'est donc plus utile d'implémenter la fonctionnalité de compression dans chaque application.

1.5.4 Discussions

Une fonctionnalité liée à la sécurité dont on n'a pas parlé est le principe des *quotas* [MJLF84]. Les utilisateurs peuvent créer chacun dans leurs coins leurs fichiers, et être assurés que leurs contenus ne seront pas écrasés par l'ajout d'un autre fichier par un autre utilisateur. Il reste cependant que tous ces utilisateurs partagent une même ressource : le disque et ses secteurs. Un utilisateur pourrait ainsi s'accaparer tout le disque. Le système des quotas est constitué d'une base de données gérée par

le *root* qui attribue des quotas à chaque utilisateur, puis chaque opération système d'écriture sur le disque est ensuite interceptée et soumise à l'approbation du système des quotas.

1.6 Implémentation des systèmes de fichier

Nous avons fait dans les sections précédentes un tour d'horizon des possibilités offertes par les systèmes de fichier. Nous n'avons cependant pas encore répondu à une question importante : comment ça marche ? Il y a cependant trop de systèmes de fichier différents, trop de fonctionnalités offertes par les fichiers pour qu'on puisse les décrire tous. Ainsi, on ne parlera pas des mécanismes liés à la gestion des périphériques, des *pipe*, ou bien encore du fonctionnement interne de NFS. On va ainsi se borner à décrire l'implémentation d'un système de fichier simplifié, dans un système d'exploitation simplifié, servant juste à stocker les documents d'un utilisateur sur un disque classique.

Nous allons tout d'abord décrire les *structures de données* que manipule un système de fichier pour ensuite décrire ses *opérations*.

1.6.1 Structures de données

Contrairement aux applications classiques, un système de fichier gère surtout, en plus de structures de données en *mémoire*, des structures de données sur le *disque*. En effet, non seulement les fichiers et répertoires prennent trop de place pour pouvoir tous les stocker en mémoire mais en plus ils doivent *persister* après un redémarrage de la machine. Nous allons présenter d'abord les structures de données présentes sur le disque puis nous verrons celles présentes en mémoire qui stockent temporairement celles du disque, et qui agissent comme une sorte de *tampon* entre les processus et le disque.

Sur le disque

Une préoccupation importante des systèmes de fichiers est l'organisation de l'information sur le disque. En effet, le disque est en général fait d'un ensemble de *blocs* de données, les *secteurs*, et le *driver* d'un disque (le code dans le noyau chargé de gérer l'accès à ce disque) fournit juste comme API la possibilité d'accéder et de modifier un bloc indexé par son numéro (son *adresse*). Le programmeur ne voit le disque que comme un gros tableau où chaque entrée fait la taille d'un secteur. C'est donc une vision assez pauvre et représenter un arbre et des fichiers de tailles variables dans un tableau n'est pas si évident.

Les services fournis par les systèmes de fichier sont d'abstraire, via les fichiers et répertoires, cet amas de blocs, et d'utiliser au mieux cet amas, cette ressource. Il est évident que le contenu de ces fichiers et répertoires sera au final stocké dans des blocs. Nous allons commencer par décrire un schéma naïf pour l'organisation de ces données, ce qui permettra de se faire une idée générale rapidement de l'ensemble des notions et difficultés liées à un système de fichier, pour développer ensuite petit à petit des solutions plus élaborées.

Une organisation naïve On peut décider en première approche que tous les blocs contenant les informations d'un fichier ou d'un répertoire sont placés de manière *contiguë* sur le disque. Pour accéder au contenu d'un fichier ou répertoire il suffirait donc de connaître uniquement l'*adresse* de son premier bloc. On pourrait réserver de la place au début du premier bloc pour stocker des informations systèmes sur ce fichier comme sa taille ou son propriétaire. Les blocs d'un répertoire pourraient contenir la liste des fichiers et sous-répertoires de ce répertoire en stockant une liste de triplets avec la chaîne de caractères représentant le nom du fichier ou du répertoire, le type de l'entité (fichier ou répertoire), et enfin l'adresse de cette entité avec l'entier correspondant au premier bloc de cette entité. On peut donc considérer le répertoire comme une sorte de fichier, puisqu'on utilise les mêmes mécanismes d'accès avec les adresses de bloc. Dans la suite on emploiera le mot entité pour désigner aussi bien un fichier qu'un répertoire. On pourrait ensuite stocker de manière fixe au tout début du disque, dans le premier secteur, l'adresse du premier bloc du répertoire racine pour pouvoir amorcer le processus. La Figure 1.3 illustre un exemple d'organisation du disque reprenant ces choix.

Cette structuration du disque est complète. Elle permet de répondre à tous les besoins. Pour accéder

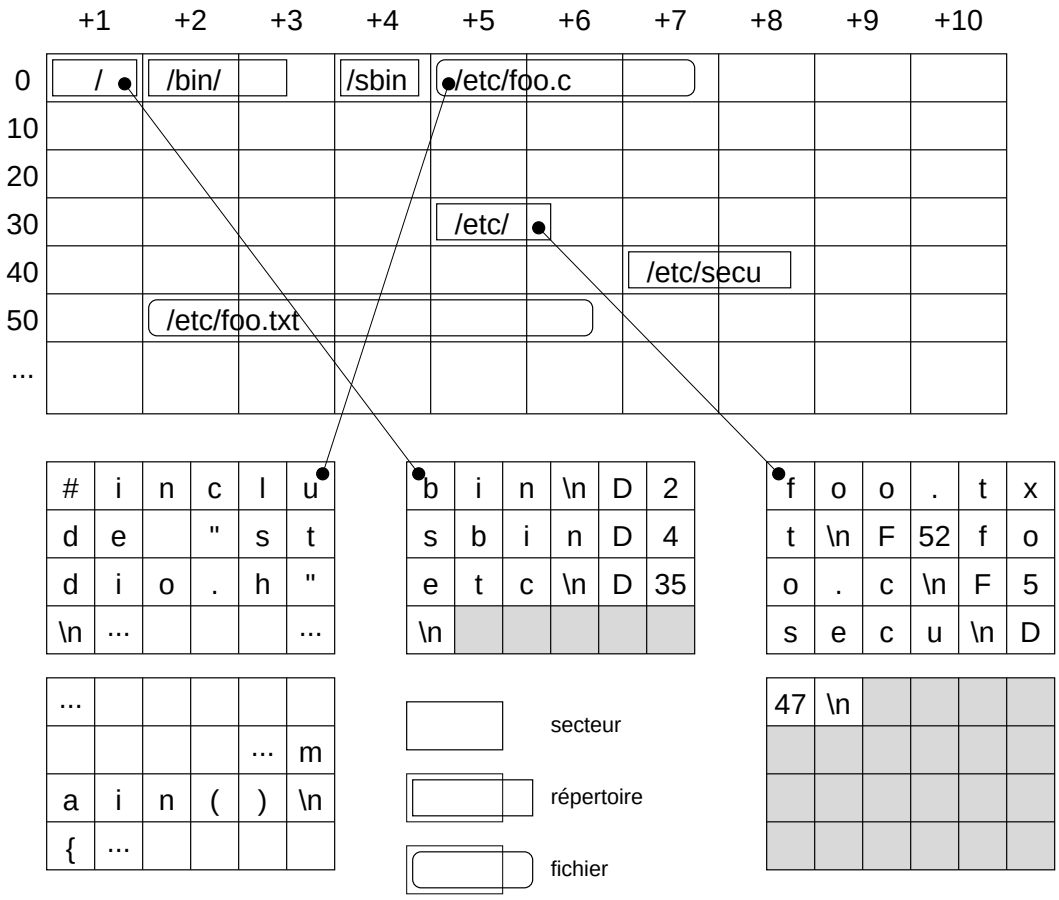

FIG. 1.3 – Organisation naïve du disque

par exemple au contenu du fichier `/etc/foo.c`, il suffit tout d'abord de récupérer l'adresse du premier bloc de la racine, qui est toujours placée au début du disque. Avec cette adresse, on trouve les blocs contenant les informations du répertoire `/`, dont le triplet correspondant à `etc`. Le troisième élément de ce triplet nous donne l'adresse du premier bloc de `/etc`. On procède donc de même que pour la racine en localisant cette fois le triplet correspondant à `foo.c`. On récupère donc cette fois l'adresse du premier bloc du fichier `/etc/foo.c`, et il ne reste plus qu'à lire ce premier bloc et les suivants pour lire le contenu de ce fichier.

Fragmentation Cette structuration naïve est cependant inefficace. Ainsi, lorsque ces fichiers grossiront, il arrivera forcément un moment où la fin d'un fichier, son dernier bloc, sera à côté du premier bloc d'un autre fichier, et si le premier fichier grossit encore un peu plus, on risque d'écraser les données du deuxième fichier. Il faudrait alors déplacer le fichier à un autre endroit avec suffisamment de blocs de libres qui se suivent, et mettre à jour le répertoire référençant ce fichier puisque son adresse changerait. On peut cependant très vite arriver à un *jeu de taquin* pour pouvoir caser tous les fichiers. La contrainte de contiguité est donc trop forte. On préfère donc relacher cette contrainte en représentant un fichier par un ensemble de blocs, possiblement *éparpillés* sur le disque. On *fragmente* le fichier. Il faut du coup représenter le *lien* qui unit tous ces blocs. On pourrait décider de réserver à la fin de chaque bloc de la place pour stocker l'adresse du prochain bloc, et donc de *chaîner* entre eux ces blocs. Cependant, l'accès direct à la fin d'un fichier (par exemple avec la commande Unix `tail`) prendrait beaucoup de temps puisqu'il obligerait à parcourir de manière exhaustive ce chaînage. L'implémentation des systèmes de fichier classiques préfère au chaînage, la construction d'une structure d'*indexation*, qui étant donné un fichier ou répertoire permettra de récupérer l'ensemble de ses blocs, et ce dans le bon ordre, et dont on pourra rapidement par exemple récupérer le dernier bloc de données. Il faut cependant désormais un moyen de faire le lien entre un fichier et cette structure d'indexation. On pourrait là encore réserver de la place dans le début du premier bloc d'un fichier pour stocker cette structure, mais comme on va le voir, cette structure peut prendre beaucoup de place.

Table des *inodes* Sans doute en partie pour des raisons de génie logiciel[7], les systèmes classiques préfèrent séparer plus clairement les *méta-données* des *données* elles-mêmes, en regroupant ensemble toutes les méta-données au début du disque dans une table. Chaque entrée de cette table contient donc des informations sur un fichier ou répertoire particulier, ses adresses de blocs avec la structure d'indexation, mais aussi ses informations systèmes avec sa taille, son propriétaire, son groupe, ses droits d'accès, ses dates d'accès et de modification. On appelle *inode* pour *information-node* ces informations. Il faut donc désormais un moyen pour faire le lien entre l'entrée d'une table, un *inode*, et le fichier ou répertoire correspondant. On associe à chaque fichier un numéro appelé *numéro d'*inode qui correspond au numéro de l'entrée de cette table que l'on appelle donc *table des inodes*. Le triplet des répertoires contient donc désormais non plus l'adresse du premier bloc d'un fichier mais son numéro d'*inode*. C'est à partir de cet *inode* que l'on pourra ensuite retrouver l'adresse des blocs d'un fichier. De plus, le type d'une entité (fichier ou répertoire) est en fait stocké dans la table des *inodes*. Le contenu des répertoires est donc en fait composé d'un ensemble de doublets (nom, *inode*). La Figure 1.4 illustre un exemple d'organisation du disque avec cette table des *inodes*.

Désormais, pour accéder au contenu du fichier `/etc/foo.c`, on récupère les blocs correspondant à la racine en utilisant l'entrée 1 de la table des *inodes*, qui par convention correspond toujours au numéro d'*inode* de la racine. On lit ensuite ces blocs et on localise le doublet correspondant à `etc`. Le deuxième élément de ce doublet est le numéro d'*inode* du répertoire correspondant ; on récupère alors les blocs du répertoire `/etc` en lisant encore dans la table des *inodes* les informations concernant cet *inode*, et ainsi de suite.

Structure d'indexation Il faut noter que pour que l'on puisse calculer où se trouve l'entrée correspondant

[7] Et aussi pour des raisons de tolérance aux fautes comme on le verra dans la Section 1.6.1 page 42.

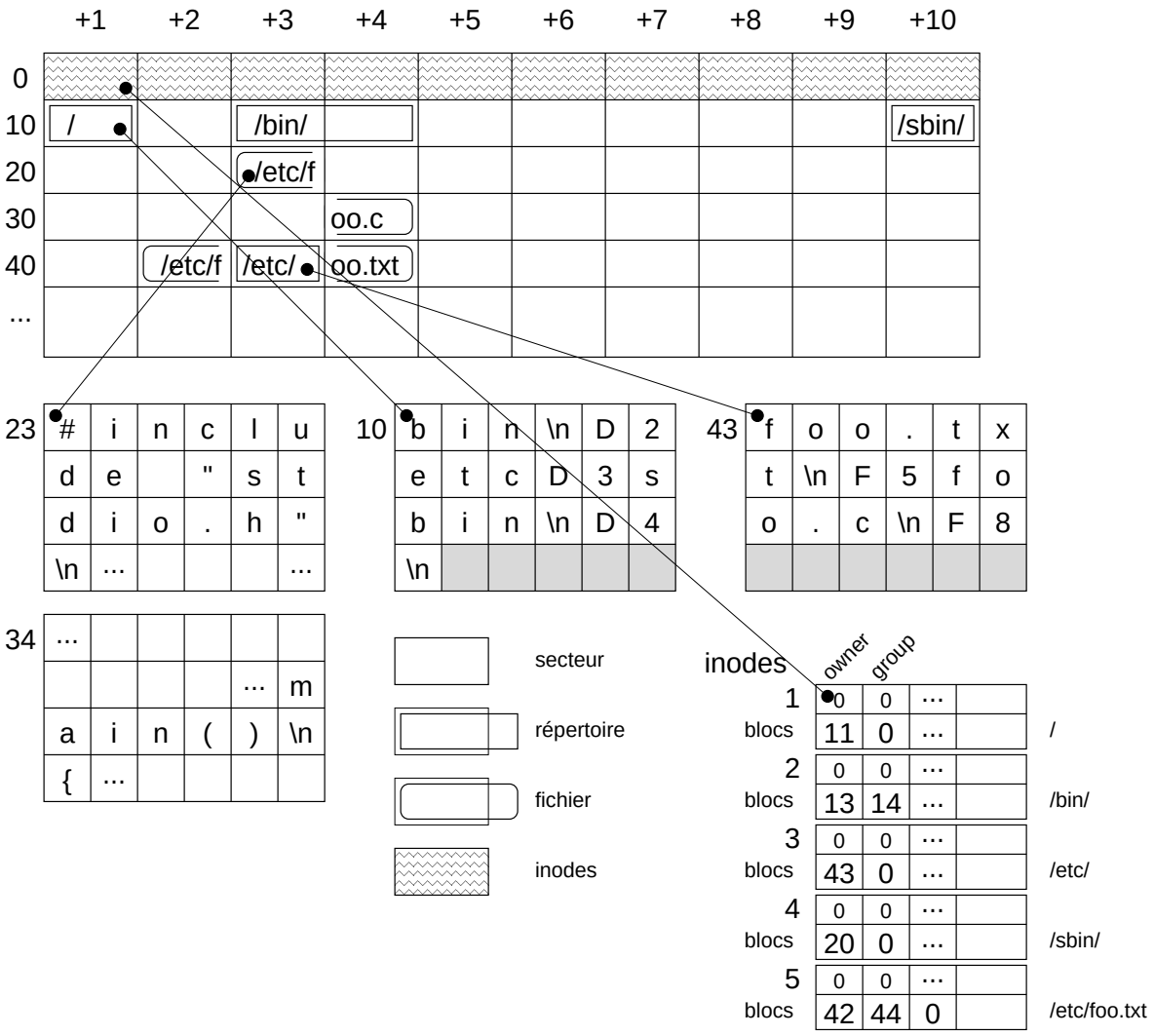

FIG. 1.4 – Organisation avec une table des *inodes*

à un numéro d'*inode* dans la table des *inodes*, il faut que chaque entrée, chaque *inode*, ait la même taille. Cependant les fichiers sont de tailles variables, et l'on peut aussi bien avoir de très gros fichiers que des très petits. L'utilisation d'un tableau d'adresses comme structure d'indexation d'un fichier est trop coûteuse. On utilise donc une autre technique qui consiste au lieu de stocker la liste des adresses de blocs d'un fichier sous la forme d'un tableau dans la table des *inodes*, à les stocker à l'extérieur de la table des *inodes* sous une forme proche d'un *arbre binaire*. Ainsi, on stocke uniquement dans la table des *inodes* l'adresse d'un bloc, ce qui permet donc de rendre à la fois la taille d'une entrée de la table des *inodes* fixe et petite. Ce bloc peut contenir lui même 128 adresses de bloc, pouvant contenir eux mêmes 128 adresses de blocs et ainsi de suite pour arriver à des blocs qui contiendront les adresses directes des blocs d'un fichier. En choisissant une profondeur de 3, on peut gérer des fichiers allant jusqu'à 1Go[8]. En grossissant le *grain* à des blocs de 4ko[9] au lieu de 512 octets, on arrive à gérer des fichiers de 8Go. C'est donc une structure d'indexation en forme d'arbre, et elle permet d'atteindre rapidement tous les octets d'un fichier en suivant le bon chemin. Si la taille d'un fichier est nulle, on placera la valeur spéciale 0 comme adresse de bloc dans le tout premier bloc, sorte de bloc racine. Ainsi, l'espace pris par cette structure d'indexation pour ce fichier sera de 1 bloc seulement. En procédant de manière similaire pour des fichiers prenant quelques blocs, on aura besoin uniquement de 3 blocs pour stocker ces adresses de blocs. En fait, pour des raisons d'optimisation, la table des *inodes* ne contient pas que l'adresse du bloc racine, mais contient directement aussi les adresses des dix premiers blocs d'un fichier qu'on appelle les *blocs directs*. L'accès aux méta-données d'un petit fichier ne requiert donc la lecture que d'un bloc (le bloc contenant l'*inode*) au lieu de quatre, ce qui est critique dans le monde des systèmes de fichier et ce qui est précisément ce que l'on cherche à minimiser. Pour le même genre de raison, la table des *inodes* contient de plus l'adresse d'un bloc dit *indirect*, l'adresse d'un bloc dit *doublement indirect* et finalement l'adresse d'un bloc *triplement indirect*, qui correspondent d'une certaine manière à des blocs racines d'arbres allant d'une profondeur 1 à 3. La Figure 1.5 illustre l'organisation de ces adresses de bloc pour un fichier d'une taille ne nécessitant d'aller que jusqu'au bloc simplement indirect.

[8] $512/4 = 128 = 2^7, 2^7 * 2^7 * 2^7 * 512 (= 2^9) = 2^{30} = 1Go$

[9] En obligeant à avoir un peu de contiguité de blocs ce qui fait qu'une adresse désigne une suite de 8 blocs.

Organisation finale générale Pour finir, en plus de la table des *inodes*, le système de fichier a besoin d'autres méta-données pour fonctionner en maintenant sur le disque la liste des statuts de blocs libres ou utilisés, et aussi la liste des statuts d'*inodes* libres ou utilisés. La destruction d'un fichier pourra ainsi libérer de la place pour de prochaines créations. On stocke en général ces deux listes sous la forme d'un *bitmap*, où chaque bit représente l'utilisation ou non de ce numéro de bloc ou d'*inode*. La Figure 1.6 résume l'organisation (un peu simplifiée) des structures de données d'un système de fichier classique comme EXT2 [CTT94] (le système de fichier par défaut sous Linux) sur un disque. Le début du disque est réservé aux méta-données, la fin aux données elles-mêmes. Le début de la section méta-donnée est en général utilisé pour stocker ce qu'on appelle le *superblock* qui contient des informations globales sur le système de fichier comme par exemple la place restante disponible. Vient ensuite le bitmap des *inodes*, des blocs, puis la table des *inodes*. La section donnée contient les blocs représentant le contenu des fichiers, des répertoires, mais peut aussi contenir des méta-données avec les blocs indirects qui contiennent des adresses de blocs.

Optimisations Il existe de nombreux travaux visant à optimiser les systèmes de fichier et donc l'organisation précédente. Une référence importante dans ce domaine est le *Fast File System* (FFS) [MJLF84]. Là encore, contrairement aux applications classiques, les optimisations concernant les systèmes de fichier visent surtout non pas à optimiser les temps de calcul CPU, car il y en a peu, mais à optimiser les «temps de calcul» du disque. De même, d'autres optimisations visent surtout à réduire l'occupation du disque et non l'occupation mémoire.

Une caractéristique importante des disques durs, contrairement à la mémoire, est que le temps de dé-

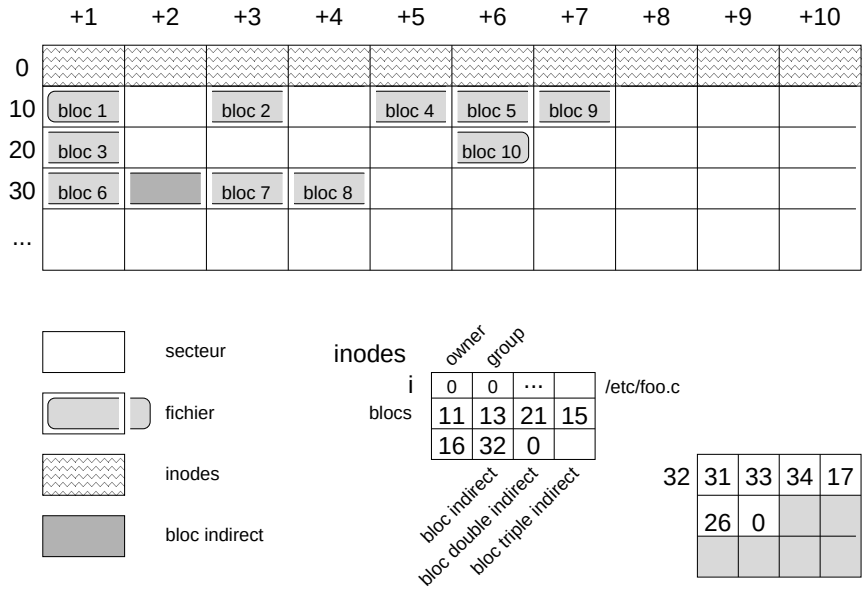

FIG. 1.5 – Indirections de blocs

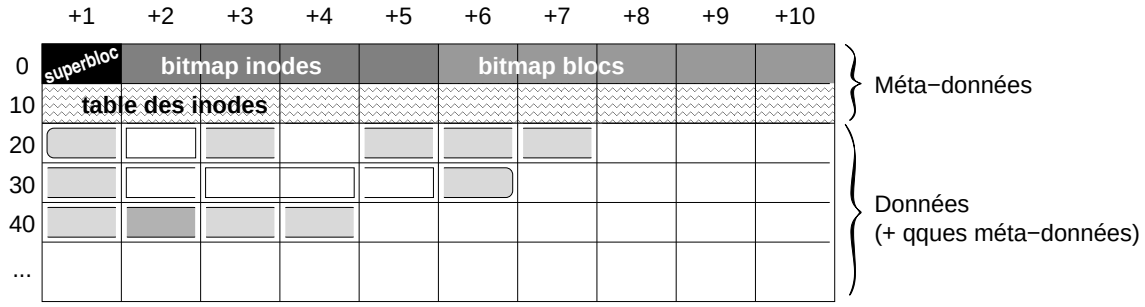

FIG. 1.6 – Organisation sur le disque

placement de la tête de lecture est très long (le *seek time*) lorsque cette tête doit parcourir une grande distance comme d'aller du début à la fin du disque. Beaucoup d'optimisations visent donc à minimiser ces déplacements.

Une de ces optimisations concerne la fragmentation. Nous avons vu que l'on ne peut forcer les blocs d'un fichier à être placés de manière contiguë sur le disque puisque la présence de nombreux fichiers entraînerait forcément des conflits et des jeux de taquin pour pouvoir caser tous ces fichiers. On relâche donc cette contrainte en permettant aux blocs d'être éparpillés sur le disque. Cependant, on n'aimerait pas non plus que les blocs d'un fichier soient trop éparpillés. En effet, dans ce cas la lecture d'un fichier entraînerait sans arrêt des va-et-vient sur le disque et donc de nombreux déplacements de la tête de lecture.

La politique d'allocation de blocs du système de fichier FAT du système DOS consiste lors de la création d'un nouveau fichier à prendre le premier bloc libre à l'aide du *bitmap* des blocs libres. Cependant, l'ajout et la suppression fréquente de fichiers peut très vite faire ressembler le début de ce *bitmap* à un gruyère. Les blocs des fichiers créés ont ainsi de grande chance de se retrouver éparpillés. Ce genre de système de fichier force ensuite l'utilisateur à utiliser un utilitaire comme `defrag` pour de temps en temps réarranger les blocs des fichiers (ce qui prend énormément de temps) pour rendre plus rapide le système. Une meilleure politique lors de la création d'un nouveau fichier consiste non pas à prendre le premier bloc libre, mais à prendre le premier bloc d'une grande zone libre comme c'est le cas sous EXT2[10]. On place donc le fichier à un endroit où il y a une suite importante de blocs libres ; on *prévoit* et on *réserve*. Cela n'empêche pas la fragmentation mais ça la retarde au maximum au point qu'il est en général inutile de défragmenter ce genre de systèmes de fichier.

Une autre optimisation visant à minimiser le *seek time* consiste à rapprocher les méta-données des données. La lecture d'un fichier entraîne tout d'abord la lecture de son *inode* dans la table des *inodes* pour récupérer les adresses de ses blocs, puis la lecture effective de ces blocs. Cependant, si la table des *inodes* est stockée au début du disque, cela peut entraîner un long déplacement si les blocs du fichier sont à la fin. Une des optimisations de EXT2 consiste à tronçonner la table des *inodes* et à la répartir sur le disque, par exemple une partie au début, une au milieu et une à la fin. Le système essaye ensuite de placer les blocs d'un fichier près de sa table des *inodes* correspondante. De même, le système essaye de placer les blocs des fichiers d'un répertoire près des blocs de ce répertoire.

Une optimisation apparue récemment dans les systèmes de fichier grand public consiste à représenter le contenu d'un répertoire non plus comme une liste de doublets mais comme un arbre de recherche indexé par la chaîne de caractère du nom de l'entrée via un B-arbre [Com79](*B-tree*). Les entrées du répertoire sont d'une certaine manière triées. Ainsi, si un utilisateur ou une application sait précisément le nom du fichier ou sous-répertoire auquel il veut accéder, le système de fichier pourra retourner rapidement l'*inode* correspondant. Cette optimisation est utile pour les répertoires contenant un grand nombre d'entrées.

En ce qui concerne l'occupation disque, nous avons déjà vu une optimisation avec GzipFS qui compresse de manière transparente certains fichiers, en général les fichiers les moins fréquemment utilisés et les plus gros. Une autre optimisation concerne cette fois le stockage des petits fichiers. En effet, du fait de l'adressage par bloc, un petit fichier occupera tout un bloc même si il ne le remplit pas complètement. En fait, comme on veut aussi gérer des fichiers de très grandes tailles, le grain de l'adressage est souvent supérieur au bloc et la création d'un petit fichier ou d'un répertoire presque vide entraîne en général le gaspillage de plusieurs blocs. Cela peut parfois surprendre l'utilisateur puisque la taille de l'ensemble des fichiers retournée par la commande `du` est souvent très inférieure à la taille effective prise par ces fichiers sur le disque comme retournée par la commande `df`. On peut ainsi arriver à saturation du disque plus rapidement que prévu. Ainsi, certains systèmes de fichier qui modifient le système d'adressage du contenu des fichiers, peuvent emmagasiner dans le même bloc plusieurs petits fichiers ce qui évite ainsi tout gaspillage.

Tolérance aux fautes Un aspect important des systèmes de fichier que nous n'avons pas encore évoqué est

[10] Cette différence de politique est d'ailleurs souvent citée pour vanter les mérites de Linux sur Windows et de manière plus générale des logiciels libres sur les logiciels commerciaux [Cos98].

celui de la *tolérance aux fautes*. En effet, c'est une fonctionnalité primordiale d'un système de fichier. Le système de fichier doit être exploitable même si la machine a subit une défaillance, par exemple un *reboot* brutal suite à une coupure d'électricité.

L'exécution d'une commande *shell* entraîne en interne la modification de nombreuses tables et blocs et donc une suite d'actions internes plus élémentaires. Si le système est arrêté en plein milieu d'une telle suite d'actions, le système de fichier peut se retrouver dans un état incohérent. Par exemple, suite à l'effacement d'un fichier avec la commande `rm`, le système doit mettre à jour les blocs du répertoire contenant ce fichier pour supprimer le doublet correspondant, libérer les blocs pris par ce fichier et donc mettre à jour le *bitmap* des blocs libres, libérer les blocs pris par les adresses de blocs de ce fichier (les blocs indirects, doublement indirects), puis libérer l'*inode* en mettant à jour le *bitmap* des *inodes* libres. Si la machine redémarre brutalement par exemple juste après avoir mise à jour les blocs du répertoire, les blocs et l'*inode* du fichier n'auront pas encore été libérés et un fichier accessible de nulle part prendra inutilement de la place sur le disque.

On peut ordonner certaines actions internes, comme libérer d'abord les blocs du fichier puis à la fin seulement modifier les blocs du répertoire. Ce faisant, même si la machine redémarre brutalement au milieu de l'action, l'opération d'effacement n'aura certes pas été complète mais elle laissera tout de même le système de fichier dans un état cohérent (l'entrée du répertoire pointera vers un fichier de taille vide).

Cependant, de nombreuses actions ne peuvent pas être ordonnées. On ne peut par exemple libérer les blocs d'un fichier sans mettre à zéro aussi les adresses de ces blocs dans l'*inode* de ce fichier (ou les blocs indirects) et inversement. Dans un cas des blocs resteront inutilisés par le système, dans l'autre ils seront trop utilisés puisqu'ils pourront être alloués ensuite en même temps à un autre fichier.

Pour éviter les risques d'incohérence, une technique consiste suite à un redémarrage brutale à lancer un processus chargé de vérifier l'état du système (par exemple avec la commande `fsck.ext2` pour EXT2). Pour détecter ces pannes, il suffit par convention lors du montage de placer un certain bit du disque à 0 et de le remettre à 1 lors du démontage. Si au prochain montage le bit est toujours à 0 c'est qu'une panne est arrivée avant que l'utilisateur ait pu éteindre proprement la machine. Le processus de vérification consiste essentiellement à vérifier que les *invariants* concernant les structures de données sont toujours respectés, par exemple qu'un bloc ne peut être à la fois libre et mentionné dans les adresses de blocs d'un fichier. Ainsi, pour cet exemple d'invariant, au redémarrage le système parcourra l'ensemble de tous les fichiers accessibles depuis la racine pour vérifier que tous les blocs pris par ces fichiers sont marqués comme utilisés dans le *bitmap* des blocs libres. Selon le type d'incohérence, le système peut essayer de réparer la faute en inférant ce qui n'a pas marché pour poursuivre ou annuler l'action entamée avant la panne, par exemple en remarquant un bloc comme utilisé dans le *bitmap* des blocs libres. Pour d'autres incohérences le système peut s'en remettre à l'utilisateur. Par exemple, si le système s'aperçoit que l'*inode* d'un fichier est marqué comme utilisé mais n'est référencé par aucun répertoire, on peut re-référencer cet *inode* en l'associant à un fichier du répertoire `lost+found/`. Il peut y avoir beaucoup d'invariants à contrôler et le processus global de vérification est en général très long.

Une autre technique, la *journalisation* [RO92], vient de l'observation que seule la dernière opération effectuée avant le crash sur le système peut avoir corrompu le système. Le problème lorsqu'une suite d'actions internes est interrompue est que le système au redémarrage ne se rappelle plus ce qu'il lui restait à faire. Sous un système de fichier journalisé, chaque action à exécuter est d'abord enregistrée dans un *log* à un endroit précis du disque. Ce *log* contient toutes les informations concernant cette action comme les *inodes* ou blocs mises en jeu pour cette action et les modifications à faire sur ces données. Une fois enregistré le *log*, le système procède à l'exécution effective de la suite d'actions internes puis efface ce *log* du disque. Si jamais le système se crashe avant la fin de l'opération, il suffira au redémarrage grâce aux informations contenues dans le *log* de rejouer les actions. Bien sûr, l'écriture du *log* peut elle même être interrompue. Il suffit de choisir un marqueur spécial de fin de *log*. Si le système n'a pas eu le temps d'écrire ce marqueur, le *log* ne contient donc pas toutes les informations nécessaires et il suffit d'annuler l'opération puisque de toute façon elle n'aura pas été commencée.

On peut assimiler cette technique à la technique em-

ployée par les éditeurs de texte pour permettre aux utilisateurs de revenir en arrière sur leurs actions ou de rejouer ces actions par le biais de boutons *undo* et *redo*. En effet, les éditeurs maintiennent en interne la liste des actions faites par l'utilisateur. Ils ne sauvegardent pas l'état complet du processus entre deux modifications mais uniquement le *delta*. Nous verrons Section 3.3 page 109 que l'on utilisera cette technique de journalisation pour rendre LFS tolérant aux fautes.

Il faut noter que ces différentes techniques ne peuvent pas prévenir des failles matérielles. Les disques durs contiennent déjà une forme de protection contre certaines défaillances avec la gestion en interne de *codes correcteurs* [Bro00]. Cependant, un secteur ou même le disque dur entier peut devenir défectueux et inutilisable. La seule technique de tolérance aux fautes est alors la *redondance*. L'utilisateur est encouragé à sauvegarder sur d'autres disques ses données, à faire des *backups*. On peut aussi rendre plus transparentes, plus efficace, et plus fréquentes ces procédures de sauvegarde avec la coopération du noyau en faisant d'un ou plusieurs disques durs les miroirs du disque de l'utilisateur (le système RAID [CLG+94]).

En mémoire

Le programmeur ne peut accéder ni modifier directement un octet d'un bloc du disque. En effet, l'adressage du disque ne peut se faire qu'au niveau du grain du bloc. Pour pouvoir modifier un bloc sur le disque, ou un octet de ce bloc, il faut d'abord charger le bloc en mémoire, modifier sa copie en mémoire, puis sauver la copie vers le disque. Comme dit au début de cette section, la mémoire sert essentiellement de tampon au disque et mime d'une certaine manière une partie des structures de données présentes sur le disque. Pour des raisons d'efficacité, il serait coûteux de relire à partir du disque à chaque accès à un fichier (ou répertoire) les blocs d'informations contenus dans la table des *inodes*, les blocs contenant les informations d'un répertoire, ou encore les blocs de données d'un fichier. Le système d'exploitation fournit donc des structures de *cache* pour accélérer chacun de ces accès :

- Le *buffer cache* cache les blocs de données d'un disque, et est donc indexé par un numéro de bloc.
- Le *inode cache*, ou *icache*, cache les informations d'un *inode* et est indexé par un numéro d'*inode*. Chaque entrée de ce cache mime le contenu d'un *inode* sur le disque.
- Le *directory cache*, ou *dcache*, cache le contenu d'un répertoire et est indexé par un numéro d'*inode*, celui du répertoire correspondant. Chaque entrée de ce cache contient une table de *hash* qui étant donnée une chaîne de caractère, correspondant à un nom de fichier ou sous-répertoire, retourne rapidement son numéro d'*inode*. Ce cache mime donc en mémoire une partie de la hiérarchie du système de fichier.

La modification d'une entité, que ce soit un bloc de données, un *inode* ou un répertoire s'effectue d'abord sur le cache, et est ensuite répercutée sur le disque. De plus, différentes modifications espacées dans le temps sur le même bloc peuvent n'entraîner qu'une seule lecture et une seule écriture d'un bloc sur le disque.

Nous allons maintenant voir une structure de donnée en mémoire supplémentaire qui cette fois n'est pas présente sur le disque : le *descripteur de fichier*. L'utilisateur et le programmeur d'applications n'utilisent pas directement des numéros d'*inode* pour accéder à un fichier, mais son nom, ce qui est plus pratique. Ainsi, la fonction `open` qui sert à ouvrir l'accès à un fichier prend en paramètre le chemin d'accès à un fichier sous la forme d'une chaîne de caractère. Le système de fichier se charge d'effectuer la translation de nom de fichier vers *inode* en utilisant les *dcache* et *icache*, voir le disque si les informations suffisantes ne sont pas encore chargées en mémoire.

Il faut noter qu'il faut vérifier que l'utilisateur a effectivement le droit d'accéder à cet *inode*. On fait cette vérification une fois pour toute dans l'opération `open`. Ainsi, l'opération `open` retourne un *objet système* qu'on appelle *descripteur de fichier* (ou *fd* pour *file descriptor*). Ce descripteur est un entier qui pointe dans une *table des descripteurs*. L'entrée de cette table contiendra le numéro d'*inode* correspondant au fichier. Les opérations comme `read` prendront en paramètre ce descripteur de fichier. On peut voir un descripteur comme une structure de donnée *opaque*, intermédiaire entre l'utilisateur et le noyau, et gérée par le noyau. L'utilisateur ne peut donc pas *forger* son propre descripteur de fichier ; seule l'opération `open` permet de construire ce genre d'objet système.

Pour résumé, on peut voir la chaîne de caractère représentant le chemin d'un fichier comme le moyen d'accès *symbolique*, le descripteur de fichier comme le moyen d'accès *sécurisé*, et enfin l'*inode* comme le moyen d'accès *physique* à un fichier.

1.6.2 Opérations

Un système d'exploitation est un système complexe. Une bonne habitude issue du génie logiciel est de structurer un système en différentes *couches* pour gérer cette complexité. Chaque couche se repose sur les abstractions et services fournis par la couche du dessous, et ajoute ses services dont pourra bénéficier la couche du dessus. On retrouve ce principe de couches pour le système de fichier.

Ainsi, on a déjà vu avec les structures de données différents niveaux, avec au plus haut niveau la chaîne de caractère représentant le nom d'un fichier, puis le descripteur de fichier, puis l'*inode*, et enfin le secteur. Il en est de même pour les opérations, avec au plus haut niveau les *applications*, puis les *librairies*, et enfin le *noyau*. Ils correspondent à différents acteurs, l'utilisateur, le programmeur, et enfin le programmeur système. Chacun de ces acteurs voit donc le système de fichier d'un point de vue différent, avec des types de données différents, et des noms de fonctions différents aussi.

Les différentes couches

Au niveau *application*, on voit les services du système de fichier au travers de boîtes de dialogue ou de boutons, qui permettent par exemple de choisir un fichier dans une liste. Au même niveau, on retrouve le *shell* avec un ensemble de commandes comme `cd`, `ls`, `cat`, etc. À ce niveau on n'utilise que des chaînes de caractères pour les paramètres de ces commandes, qui peuvent représenter des chemins absolus ou relatifs.

Le code de ces applications graphiques et de ces commandes *shell* repose sur des *librairies*. Il peut même exister différentes couches au niveau des librairies avec une librairie de haut niveau dans un langage comme C++ qui utilise elle même une librairie de plus bas niveau. Cependant, la différence entre ces librairies du point de vue du système de fichier n'est pas très grande, et la plupart des applications reposent donc sur la librairie C. Les fonctions ne s'appellent pas `cd`, `ls`, ou `cat` mais `readdir`, `open`, ou `read`. C'est à ce niveau que l'on voit apparaître la notion de descripteur de fichier. La fonction `open` prend donc en paramètre le nom d'un fichier et retourne un entier, le descripteur, qui pourra être passé en paramètre de la fonction `read` par exemple.

Le code de la librairie C s'appuie lui même sur une sorte de librairie : le *noyau*. Les fonctions de cette librairie ne sont plus accessibles par un simple appel de fonction mais par un appel via le mécanisme d'*interruption*. Par exemple, sous Linux le code assembleur de la fonction `open` dans la librairie C place la valeur 5 dans le registre `ax`, ce qui correspond au 5ème appel système géré par le noyau (`SYS_OPEN`), place l'adresse de la chaîne de caractère représentant le nom du fichier à ouvrir dans le registre `bx`, puis appelle l'interruption système correspondant à un appel au noyau avec `int 0x80`.

Les différentes couches du noyau

Le noyau est en fait lui même divisé en différentes couches. Ainsi on a le *core* qui est constitué des fonctionnalités primordiales, avec par exemple la gestion de la mémoire, de l'initialisation, et qui définit des *interfaces* génériques, par exemple l'interface des périphériques sons. On a ensuite les *modules* qui sont des morceaux de code qui implémentent une instance particulière d'une de ces interfaces générales, par exemple un module chargé de gérer un périphérique particulier.

On retrouve encore ces même principes de couches, d'interfaces, et de modules avec la partie système de fichier du noyau. En effet, tout comme pour les périphériques, il existe différents types de systèmes de fichier, avec chacun leurs spécificités dans les détails, mais qui au fond sont assez semblables. Ainsi, Unix définit une interface générique pour tous ces systèmes de fichiers, avec un ensemble de fonctions à implémenter, par exemple les fonctions `open`, `read`. La fonction du noyau `sys_open` se charge donc d'appeler la *méthode* `open` du module approprié.

On appelle VFS (pour *Virtual File System*) [Kle86] la partie du noyau qui gère tous ces systèmes de fichiers, et qui se charge selon la requête, c'est à dire la valeur des paramètres des appels système liés au système de fichier, d'appeler le module approprié avec

les paramètres appropriés. C'est la commande `mount` qui permet d'indiquer au système quel doit être le module à appeler en fonction du chemin. Ainsi, suite à la commande `mount /dev/cdrom /mnt/cdrom -t iso9660`, le noyau saura que tout accès à un fichier à partir de `/mnt/cdrom` devra faire appel aux méthodes fournies par le module `iso9660` (ce module saura lui même aussi que les données pourront être récupérées à partir du périphérique `/dev/cdrom`, géré lui aussi par un module). Le VFS permet non seulement d'avoir accès à différents systèmes de fichier en même temps, mais il permet aussi de factoriser du code. Ainsi, c'est le VFS qui se charge de gérer les différents caches, de décomposer un chemin en parties élémentaires, de gérer les correspondances entre descripteurs de fichier et *inodes*. La seule chose que doit gérer un module qui implémente un système de fichier particulier est justement les particularités du format utilisé par ce système (ou de la sémantique particulière des opérations fondamentales données par ce système comme on le verra Section 3.3 pour LFS).

Au niveau d'un module VFS, on ne parle quasiment qu'en terme d'*inode*. Par exemple, la fonction `open` prend un *inode* en paramètre. Le travail de translation nom vers *inode* aura été principalement géré par le VFS. Les endroits où le module sera appelé avec une chaîne de caractère, seront accompagnés aussi d'un *inode*, et cette chaîne sera forcément une partie élémentaire d'un chemin. Par exemple, la méthode `lookup` (que l'on peut relier à la commande *shell* `cd`) prend en paramètre un *inode* correspondant à un répertoire, et une chaîne de caractère correspondant à un nom de fichier ou sous-répertoire, et retourne son *inode*.

Un module VFS repose lui même sur un autre service du noyau, l'accès au périphérique. En effet, l'utilisateur peut stocker ses données aussi bien sur un disque dur IDE que SCSI, et il serait idiot que chaque système de fichier réimplémente son propre code pour gérer ces différents périphériques. Ainsi, un module VFS peut utiliser l'interface générique des périphériques avec essentiellement les deux fonctions `read` et `write` qui prennent en paramètre des numéros de bloc et un numéro de périphérique.

Un exemple d'interaction entre couches

Pour mieux comprendre comment interagissent ensemble toutes ces couches, on peut décortiquer l'appel d'un utilisateur à la commande `cat /tmp/foo.c`. Le code de cette commande consiste essentiellement dans le programme C présenté ci dessous :

```
int fd = open("/tmp/foo.c");
char buffer[200];
while(read(fd, buffer, 200)) {
 print(buffer);
}
close(fd);
```

On peut décortiquer à son tour l'appel de la fonction `open` de la librairie C. Son code consiste à lancer une interruption pour se brancher dans le code du noyau avec l'appel système `SYS_OPEN`. Le code de `sys_open` appelle la fonction `namei` du VFS qui est chargé de traduire la chaîne de caractère correspondant au nom du fichier en un numéro d'*inode*. Une fois cet *inode* récupéré, la fonction `sys_open` allouera dans la table des descripteurs de fichier de ce processus une nouvelle entrée représentée par un entier et y fera correspondre le numéro d'*inode*. Elle retournera ensuite cet entier au processus appelant. Cette valeur est stockée dans la variable `fd` dans le programme ci-dessus. Ainsi, lors des appels systèmes suivants, par exemple l'appel à la fonction C `read`, le noyau pourra retrouver dans la table des descripteurs l'*inode* correspondant à ce fichier.

Il nous reste cependant à décortiquer l'appel de la fonction `namei`. Cette fonction analyse la chaîne correspondant au nom du fichier et la décompose en parties élémentaires. Ces parties sont séparées par le symbole spécial `/`. Si cette chaîne commence aussi par ce symbole, il faut commencer par lire l'*inode* de la racine. Le noyau maintient dans une table la correspondance entre chaque point de montage et le périphérique associé ainsi que le système de fichier correspondant. On peut d'ailleurs afficher cette table en tapant juste la commande `mount`. Cette table contient aussi en interne les numéros d'*inode* correspondant aux racines de ces systèmes de fichier. Il suffit donc de lire la racine de toutes ces racines qui correspond donc au point de montage initial, le `/`. À partir de cette racine, et de la connaissance du type de système de fichier et du périphérique associé, on peut appeler la

méthode `lookup` du module responsable de ce système de fichier, avec en paramètre ce numéro d'*inode* et la chaîne `"tmp"`. Cette méthode va retourner l'*inode* correspondant à ce répertoire. Pour ce faire, elle va utiliser la table des *inodes* et le disque comme vu dans la section précédente. La fonction `namei` va procéder ainsi de suite pour chaque partie élémentaire du chemin en partant de l'*inode* calculé à l'étape d'avant. Il peut arriver au cours du chemin que l'on tombe sur un point de montage d'un autre système de fichier, auquel cas le module à appeler changera.

La situation réelle est légèrement plus complexe. Par exemple, le code de `namei` doit prendre en compte et gérer les différents caches (voir [Com84, SG94] pour plus d'information sur le fonctionnement interne d'un système d'exploitation). Cependant, l'essentiel est là. La Figure 1.7 essaye d'illustrer l'ensemble des actions décrites précédemment.

Alternatives au VFS

Avant d'attaquer la présentation plus en détail du VFS, qui représente la manière traditionnelle pour implémenter un nouveau système de fichier, nous allons rapidement décrire d'autres styles d'implémentation plus exotiques, utilisés pour implémenter des systèmes de fichier souvent plus exotiques aussi.

Interception avant le noyau Du fait de l'existence de différentes couches dans le système d'exploitation, on peut faire croire à la présence d'un nouveau système de fichier en *interceptant* à différents niveaux la requête d'un utilisateur ou d'une application. Différentes possibilités d'interception sont résumées dans [AISS98]. Ainsi, il existe par exemple de nombreuses implémentations différentes du système de fichier permettant d'accéder de manière transparente aux services FTP à partir de `/ftp` (voir Section 1.3.2 page 21), de celui permettant l'accès transparent au contenu d'un `tar` (voir Section 1.3.2 page 22) ou encore de celui rendant le cryptage transparent (voir Section 1.5.3 page 34). On peut au lieu de modifier le noyau modifier les applications ou modifier les librairies.

Tout d'abord au plus haut niveau, l'application Emacs [Sta02] permet par exemple à l'utilisateur d'accéder à des fichiers FTP de manière transparente en utilisant le même jeu de commande que pour accéder aux fichiers locaux. Si le chemin du fichier auquel veut accéder l'utilisateur commence par `/ftp`, Emacs n'appelle pas le système de fichier sous-jacent mais *déroute* l'appel vers ses propres routines. Emacs *émule* la présence d'un nouveau système de fichier. On retrouve cette technique dans les systèmes de fenêtrage modernes comme Kde, Gnome, Windows ou MacOS. La même interface, celle de l'explorateur, permet de naviguer de la même manière sur le disque dur, sur le Web, sur FTP et même sur la configuration et les préférences du système. On peut aussi modifier le *shell* comme c'est le cas avec le *shell* Midnight Commander. Cette technique permet d'ajouter un service via l'interface des systèmes de fichier et rend l'accès à différentes formes d'information uniforme. Elle n'a cependant pas l'impact d'un vrai système de fichier ; elle n'améliore que quelques applications. Cependant, en choisissant les applications les plus utilisées, ici l'explorateur, le *shell* et Emacs, on s'assure tout de même d'un bon impact sur l'utilisateur.

À un niveau plus bas, on peut modifier la librairie C. Ainsi, en modifiant par exemple le code de `open`, on peut là encore faire croire à la présence d'un nouveau système de fichier en faisant en sorte que lorsque la chaîne passée en paramètre commence par `/ftp`, on n'appelle pas pas le système de fichier sous-jacent mais ses propres routines, Cette technique a un bon impact puisque la plupart des applications reposent sur la librairie C. On peut même éviter d'avoir à recompiler les applications pour bénéficier de ces nouveaux services en tirant partie du mécanisme des *librairies dynamiques partagées*. En fait, on n'est même pas obligé de modifier et recompiler les sources de la librairie C. On peut par le biais de la variable *shell* `LD_PRELOAD` interposer une librairie devant la librairie C. Cette librairie peut fournir là encore sa propre version de `open` puis appeler parfois ensuite la version normale de la librairie C.

Enfin, la dernière interception possible concerne l'appel système. L'utilitaire `strace` permet grâce à une fonctionnalité du noyau d'intercepter tous les appels systèmes effectués par une application pour les afficher à l'écran en vue de debugger cette application. On peut réutiliser ce même mécanisme pour fournir une sémantique particulière à certains appels systèmes, par exemple ceux concernant le système de fichier.

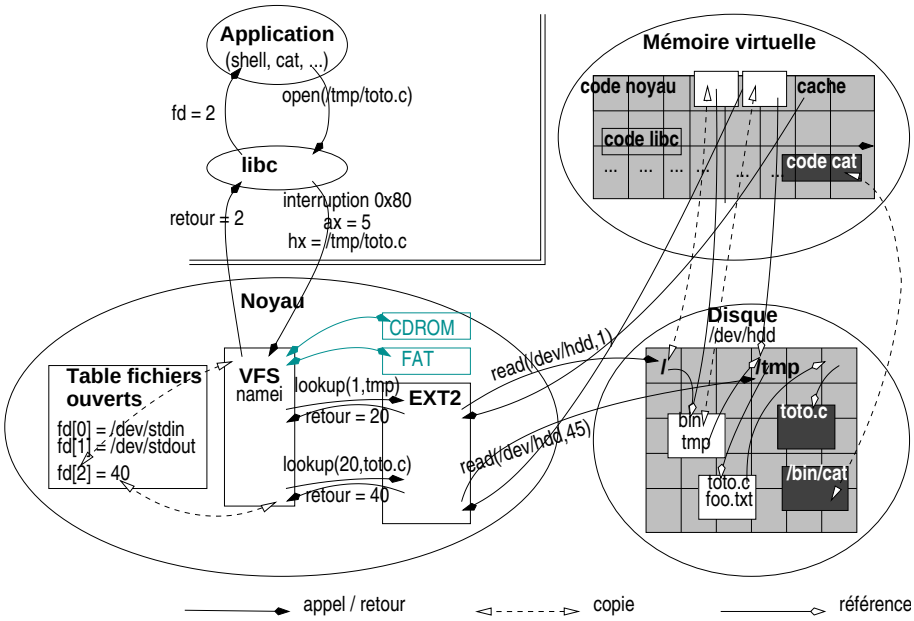

FIG. 1.7 – Diagramme de séquence de `cat /tmp/toto.c`

Les motivations qui ont conduit à l'utilisation des techniques précédentes pour l'implémentation d'un système de fichier viennent du fait que le développement à l'intérieur du noyau (*kernel-level*) est plus difficile qu'à l'extérieur (*user-level*). Il n'est pas facile de prototyper de nouvelles idées en les codant directement dans le noyau, même si cela reste tout de même la manière de faire la plus propre. Ainsi, même si le VFS a permis de factoriser du code, pour éviter par exemple à l'implémenteur d'un nouveau système de s'occuper des caches, même si le VFS définit des interfaces (presque) claires, le développement d'un module du VFS reste tout de même une tâche plus complexe que par exemple la création puis l'interposition d'une librairie dynamique.

Le noyau n'est pas une application normale. Le debuggage, le test, les compilations répétées liées au développement incrémentale ou la réutilisation sont plus pénibles dans le noyau. En ce qui concerne la réutilisation, en se plaçant à l'extérieur du noyau le programmeur peut facilement se servir de librairies existantes, de programmes existants (par exemple en appelant l'utilitaire `ftp`) et même de systèmes de fichier existants. En effet, pour l'implémentation de système de fichier comme CryptFS, on n'aimerait pas recoder toute la gestion des blocs, des *inodes* ou de la tolérance aux fautes déjà présente dans les autres systèmes de fichier. CryptFS a de nombreux points communs avec un système de fichier classique et ne diffère que pour certains appels systèmes où il rajoute un comportement spécifique. Ainsi, avec par exemple la technique d'interception d'appels système, on peut très facilement coder CryptFS en se basant sur un système de fichier comme EXT2 et sur le programme `crypt`. Il suffit de *déléguer* la réalisation de la plupart des appels système au système de fichier sous-jacent. Nous verrons Section 6.1.2 page 144 que l'on utilisera ces mêmes principes de réutilisation et délégation pour implémenter LFS.

Même si des mécanismes récents facilitent le développement à l'intérieur du noyau comme le chargement dynamique de modules, la possibilité du noyau d'appeler des processus, ou la possibilité grâce au mouvement *open source* de réutiliser le code d'autres systèmes de fichier, cela reste tout de même laborieux. De plus, en effectuant le travail à l'extérieur du noyau et donc en évitant de modifier le noyau, on évite aussi qu'un bug

dans le nouveau système de fichier puisse corrompre l'intégrité du système.

Interception après le noyau En fait, la plupart des motivations précédentes sont à l'origine de l'invention des *micro-kernels*. Dans ces systèmes d'exploitation, par exemple sous Plan9, le système de fichier est un processus comme un autre, une sorte de *serveur*. Le noyau se charge juste de rediriger les requêtes concernant le système de fichier à ce processus par un mécanisme système de *message*. La commande `mount` permet juste d'associer à un point de montage un programme ; le système d'exploitation joue donc uniquement le rôle de plateforme de routage. Un processus interroge le noyau qui interroge lui-même ensuite un autre processus : le serveur. On n'intercepte plus la chaîne d'appel ; on la rallonge. C'est un bon compromis entre les techniques d'interception et l'implémentation dans le noyau d'un module VFS. Le codage d'un nouveau système de fichier, en fait d'un nouveau serveur, peut là aussi reposer sur des librairies, programmes et système de fichier existants puisque ce serveur est une application comme une autre tournant au niveau utilisateur. Cette solution a de plus la propreté du VFS. En effet, il n'est pas forcément facile de faire coopérer plusieurs systèmes de fichier avancés par la technique d'interruption d'appels système.

Le même genre de technique fut reprise pour l'implémentation de NFS sous Unix, pour donc un système d'exploitation plus conventionnel. Une petite partie du code de NFS est placée dans le noyau (de la machine locale qui monte) dans un module VFS qui se charge juste comme sous les micro-kernel de rediriger la requête à un serveur (de la machine distante qui exporte) par l'envoi de messages. La communication, qui est bidirectionnelle, entre le noyau et le serveur se fait ici pas l'intermédiaire d'un *socket* puisque avec NFS le serveur tourne sur une machine distante. La majeure partie du code de NFS réside donc dans le code de ce serveur. Le serveur en dehors de la gestion de la communication se charge juste d'appeler le système de fichier sous-jacent comme le font les autres applications du système. On a donc là encore une chaîne d'appel de l'application (le serveur NFS) à la librairie puis au noyau mais sur la machine distante. Cela permet ainsi d'exporter n'importe quel type de système de fichier sous-jacent. Comme NFS possède toute l'architecture logicielle qui permet de mimer les fonctionnalités offertes par les micro-kernel, de nombreux systèmes de fichier comme par exemple SFS sont implémentés en modifiant le code d'un serveur NFS. Au lieu d'appeler directement le système de fichier sous-jacent, le nouveau serveur NFS peut appeler des programmes ou réutiliser le système de fichier sous-jacent en rajoutant un comportement spécifique comme le cryptage.

Cependant NFS ne fut pas conçu pour permettre l'implémentation d'autres systèmes de fichier. C'est un effet de bord inattendu. Ainsi, des clones NFS épurés ont vu ensuite le jour comme UserFS [Fit96]. Ces clones ont évacué tout l'aspect réseau pour se concentrer uniquement sur le fait de rendre facile l'implémentation de nouveaux systèmes de fichier. En fait, il mime cette fois vraiment les mécanismes présents dans les micro-kernel. La communication entre le noyau et le serveur pour l'envoi de messages se fait par l'intermédiaire d'un *pipe* ou par l'utilisation d'un pseudo-périphérique dédié. Nous utiliserons l'un de ces clones appelé PerlFS [Cal01] pour l'implémentation de LFS Section 6.1.1 page 142 et nous réutiliserons nous aussi un système de fichier sous-jacent (EXT2). Le choix de PerlFS sur les autres techniques d'implémentation vient du fait que c'est le seul système qui permet d'implémenter un système de fichier autrement que par le langage C. Les autres techniques comme les librairies dynamiques, ou l'interception d'appel système possèdent toutes une architecture logicielle qui favorise C. Pourtant, Perl est un meilleur langage pour prototyper que C.

Interception dans le noyau La dernière technique appelée *stackable filesystem* [HP94] mélange certaines qualités des autres styles (et certains défauts). Le but est de permettre aussi la réutilisation de systèmes de fichier existants, pour par exemple rendre facile le codage de CryptFS en se basant sur EXT2, tout en ayant l'efficacité et la propreté d'une implémentation directe dans le noyau. En effet, les techniques s'inspirant des micro-noyaux ont un surcoût à l'exécution puisque la chaîne d'appel est rallongée, et les techniques d'inter-

ception ne sont pas forcément très élégantes. En fait, on peut déjà réutiliser dans le noyau d'autres systèmes de fichier puisque le source de ces systèmes est souvent accessible. Cependant, ces sources ont souvent une taille importante et il n'est pas forcément très facile de savoir ou se «brancher» pour rajouter une fonctionnalité. Ainsi, certaines améliorations du VFS fournissent l'architecture logicielle permettant non seulement d'avoir les uns à coté des autres différents systèmes de fichier, mais aussi d'avoir différents systèmes de fichier les uns par dessus les autres. On retrouve là encore le principe des couches. L'implémentation par exemple d'un module VFS CryptFS appelle un autre module VFS comme EXT2. On peut avoir ainsi une tour de systèmes de fichier (empilés), chacun se reposant sur le système du dessous, et ne nécessitant aucune modification du système de fichier du dessous.

Le système Fist [ZN00] va même plus loin en fournissant un *langage dédié* (DSL pour *Domain Specific Language*) [Ben86] permettant de plus facilement exprimer les modifications et ajouts à faire sur un système de fichier existant. Le programmeur écrit une sorte de *patch*. L'utilisation d'un DSL rend aussi le processus de développement plus sûr puisque le programme DSL n'a pas accès aux structures de données du noyau. Ces détails sont réglés dans le compilateur du DSL. Le programme DSL étant restreint ne peut pas (ou moins) corrompre le système. Cependant, du fait que le code généré est inséré dans le noyau, l'implémentation d'un GzipFS ou CryptFS nécessite d'incorporer le source des programmes `gzip` et `crypt` dans le noyau. Cette extraction n'est pas forcément toujours facile car ces programmes peuvent eux même utiliser d'autres programmes ou librairies. Il faut donc tout dérouler et incorporer tous les sources. On ne peut pas réutiliser ces programmes tel quel.

Le VFS

Nous allons maintenant détailler l'interface d'un système de fichier d'un point de vue interne en décrivant les opérations (méthodes) du VFS. C'est à l'aide de cette interface que l'on décrira plus tard l'implémentation des opérations LFS Section 3.3. On peut diviser les opérations VFS en différentes catégories :

– l'initialisation et la finalisation du système.
– la consultation des répertoires.
– la consultation des fichiers.
– la création et la suppression de fichiers ou de répertoires.
– la gestion des *inodes*.

On peut en première approximation dresser un lien entre les différentes opérations et les commandes *shell* leur correspondant le mieux. Ces commandes sont indiquées dans ce qui suit entre parenthèses derrière le nom de l'opération.

Initialisation et finalisation

– `read_super` (`mount`) prend en paramètre le périphérique sur lequel le système de fichier est installé. Elle se charge d'initialiser en mémoire des structures de données internes, comme par exemple l'emplacement des différentes tables (table des *inodes*, *bitmaps*, etc) sur le disque. Ces informations sont placées dans une structure que l'on appelle le *superblock*, et l'opération rend donc cette structure. Parmi ces informations il y a le numéro d'*inode* correspondant à la racine (même si en général c'est le nombre 1). Il faut noter que cette opération ne prend pas en paramètre le point de montage. En effet, cette information ne lui est pas utile, et c'est le VFS, la partie générique, qui l'utilise.
– `put_super` (`umount`) est le pendant de l'opération précédente. En général, elle libère la place mémoire qui avait été allouée.
– `stat_fs` (`df`) retourne une liste d'entiers, chacun correspondant à une information globale du système, par exemple la taille restante disponible, le nombre de fichiers. Ces informations sont en général lues au début du disque.

Consultation des répertoires

– `readdir` (`ls`) prend en paramètre l'*inode* d'un répertoire, et un entier i et retourne un doublet correspondant au nom et au numéro d'*inode* de la ième entrée (qui peut être un fichier ou un sous-répertoire) de ce répertoire. En général, on appelle cette fonction de manière répétitive en partant de l'entier 0 et en l'incrémentant. À la fin `readdir` renverra un doublet avec comme numéro d'*inode* le nombre 0 pour indiquer qu'il n'y a plus d'autres entrées. Cette

fonction, comme toutes celles qui doivent lire un contenu (de fichier ou de répertoire), manipule les adresses de blocs directs et indirects. En effet, il faut calculer les endroits où se trouvent les blocs contenant les entrées de ce répertoire.
– `lookup` (`cd`) prend comme `readdir` en paramètre l'*inode* d'un répertoire, mais avec cette fois une chaîne de caractère représentant le nom d'un fichier ou sous-répertoire de ce répertoire. Elle retourne le numéro d'*inode* correspondant à cette entité, si il existe effectivement une telle entité. Cette opération peut sembler inutile puisqu'elle peut être implémentée de manière générique dans le VFS en s'appuyant sur `readdir`. Cependant, elle peut permettre à certains systèmes de fichier, ayant une structure plus évoluée qu'une simple liste de doublets pour le contenu d'un répertoire sur le disque (par exemple un *B-tree* [Com79]), d'offrir une version spécialisée qui va plus directement au but. De plus, elle peut permettre à l'utilisateur d'aller dans un sous-répertoire non listé. Cette fonctionnalité rend ainsi possible l'implémentation de l'auto-montage mais aussi des répertoires virtuels (voir Section 1.4.1 page 24). Ainsi, sous CATFS (voir Section 1.4.2 page 26) l'utilisateur peut exécuter du répertoire `/catfs` la commande `cd projet:foo|ext!=c/` alors que ce sous-répertoire n'existait pas. Ce type de sous-répertoire est créé *à la volée*. Comme nous le verrons Section 3.3, `lookup` est une opération essentielle sous LFS.

Consultation des fichiers

– `open` prend en paramètre un numéro d'*inode*, celui du fichier que l'on veut consulter, et des propriétés que l'on appelle *flags* indiquant dans quel mode on veut ouvrir le fichier, par exemple en lecture seule ou en écriture. Elle retourne un booléen indiquant si l'opération est possible en fonction des propriétés de sécurité du fichier. Ces propriétés sont disponibles dans l'*inode* de ce fichier. Contrairement à la fonction `open` de la librairie C, l'opération `open` ne prend pas de chaîne de caractère en paramètre. Le travail de translation de nom vers *inode* aura été fait avant (avec des appels répétés à `lookup`). En fait, cette opération est rarement implémentée dans un module puisque la sécurité peut être gérée de manière générique dans le VFS. En effet, les informations sont disponibles en mémoire dans les *inodes*, et la plupart des systèmes de fichier appliquent la même politique de sécurité. Cependant le VFS offre la possibilité d'ajouter une autre couche de sécurité. C'est ce qui permettra plus tard au Chapitre 4 de fournir un nouveau modèle de sécurité pour LFS tout en restant compatible avec le VFS et donc Unix.
– `read` (`cat`) prend en paramètre un numéro d'*inode* correspondant à un fichier, un entier correspondant à l'endroit dans le fichier où l'on veut commencer la lecture, et un autre entier correspondant au nombre d'octets que l'on veut lire. Elle retourne un pointeur vers un *buffer* contenant les octets lus et le nombre d'octets effectivement lus puisque la fin du fichier peut arriver avant le nombre demandé. Comme pour `readdir`, cette fonction doit aussi utiliser les adresses de blocs et la table des *inodes* pour localiser l'endroit sur le disque où doit commencer la lecture.
– `write` (`emacs`) est le pendant de `read` et prend en paramètre un numéro d'*inode*, une position, un pointeur vers un *buffer* contenant les informations que l'on veut écrire et la taille de ce *buffer*.
– `release` est le pendant de `open`. Cette opération est appelée à la fermeture du fichier, c'est à dire lorsque l'on indique au système que l'on ne se servira plus de ce fichier. Comme pour `open`, cette opération ne fait en général rien. Elles sont là essentiellement pour encadrer une série de `read` et `write` et pour permettre au programmeur du système de fichier d'avoir la main pour par exemple allouer et désallouer des structures internes. Cette possibilité sera utilisée par LFS et décrite Section 3.3.

Création et supression

– `create` (`touch` et `mkdir`) prend en paramètre l'*inode* d'un répertoire, une chaîne de caractère représentant le nom d'un fichier ou sous-répertoire, et un entier correspondant à un *mode* de création. Ce mode contient le type de l'entité que l'on veut créer

(fichier ou répertoire) ainsi que les droits qui seront associés à cette entité. Cette opération regroupe la création d'un fichier et d'un sous-répertoire car dans un système de fichier hiérarchique ces deux types de création provoquent les mêmes opérations en interne. En effet, dans les deux cas `create` récupère un numéro d'*inode* libre (grâce à l'un des *bitmaps*), remplit l'entrée dans la table des *inodes* correspondant à ce numéro avec les droits et types contenus dans le mode, place aussi la valeur nulle pour les adresses de bloc directes et indirectes (car le contenu de cette nouvelle entité est pour l'instante vide), puis ajoute le doublet (nom, numéro d'*inode*) dans le contenu du répertoire passé en paramètre (ce qui implique de modifier sur le disque des blocs comme pour l'opération `write` avec les fichiers). Sous LFS les fichiers et répertoires auront un statut très différent. L'opération LFS `create` appelera donc deux algorithmes différents en fonction du type de l'entité passé en paramètre dans le mode.

– `unlink` (`rm`) défait ce qui a été fait dans `create`.

– `rmdir` (`rmdir`) défait ce qui a été fait dans `create`. Cette fois les deux opérations `unlink` et `rmdir` ne sont pas mélangées car la destruction d'un répertoire implique un algorithme légèrement différent puisqu'il faut d'abord vérifier que ce répertoire n'est pas vide.

– `rename` (`mv`) prend en paramètre l'*inode* d'un répertoire dit *source*, l'*inode* d'un répertoire dit *cible*, l'ancien nom de l'entité (fichier ou répertoire) présente dans le répertoire source, et le nouveau nom que prendra cette entité dans le répertoire cible. Là encore la même opération régit le déplacement et renommage d'un fichier et d'un répertoire car ces différentes fonctions provoquent en interne les mêmes opérations. En effet, `rename` consiste essentiellement à modifier un doublet (et uniquement la première composante de ce doublet : le nom de l'entité) voir à déplacer un doublet d'un contenu de répertoire à un autre. Ainsi, le déplacement d'un répertoire pouvant contenir des milliers de fichiers (par exemple avec `mv /bin /usr/local/`) est instantané. En cela, un déplacement ne peut être réduit à une copie suivie de la destruction de l'original.

– `link` (`ln`) prend en paramètre l'*inode* d'un répertoire, le nom du futur lien, et le numéro d'*inode* du fichier pointé par ce lien. Cette opération est proche de la deuxième partie de `create`, celle où l'on rajoute juste un doublet dans le contenu du répertoire. Elle n'a pas sa première partie où l'on crée un nouvel *inode* puisque celui-ci est passé en paramètre. En fait, l'*inode* d'un fichier possède en plus des autres champs un entier correspondant à un *compteur*. La création d'un nouveau fichier avec `create` place la valeur 1 dans ce compteur. L'opération `link` incrémente ce compteur. Ainsi, l'opération `unlink`, d'où son nom, n'efface vraiment un fichier que lorsque ce compteur est à 1. Dans le cas contraire elle efface juste une entrée dans un répertoire et décrémente le compteur de ce fichier.

– `symlink` (`ln -s`) est proche de `create`. Elle prend en paramètre l'*inode* d'un répertoire, le nom du futur lien symbolique et une chaîne de caractère correspondant en général au chemin d'un fichier ou répertoire (dans le cas d'IFS, voir Section 1.3.1 page 19, la chaîne correspond au nom d'une commande). On peut voir un lien symbolique comme un fichier dont le contenu est cette chaîne[11], mais qui est traité de manière spéciale en interne par le noyau. Désormais le type d'une entité (stocké dans son *inode*) peut être soit FICHIER, RÉPERTOIRE, ou LIENSYMBOLIQUE[12]. Ainsi, lorsque `open` se rend compte que le fichier passé en paramètre est un lien symbolique, la méthode `readlink` est appelée pour récuperer la cible de ce lien et le travail de translation du VFS nom vers *inode* repart de zéro puisqu'il faut aussi analyser cette chaîne qui peut contenir des «/» ou «..». On récupère suite à ce travail un nouvel *inode*, la cible, et l'on réapplique alors `open` sur ce nouvel *inode*. Il se peut aussi que il n'y ait pas de fichier cible correspondant auquel cas l'opération retourne un code d'erreur. On appele ces types de liens «morts» des *dangling links*.

– `readlink` (`ls -l`) prend en paramètre l'*inode*

[11] Pour des raisons d'efficacité, cette chaîne est stockée directement dans l'*inode* du lien symbolique lorsqu'elle est petite.

[12] Il n'y a pas de type LIEN puisqu'un lien est une forme de fichier, et qu'il est impossible de savoir qui est l'original et qui est le lien.

d'un lien et retourne sa chaîne.

Gestion des *inodes*

- `read_inode` prend en paramètre un numéro d'*inode* et retourne un *inode*. Elle lit donc une entrée de la table des *inodes* sur le disque pour qu'elle soit placée ensuite dans le *icache* pour pouvoir être ensuite passée en paramètre aux autres opérations.
- `notify_change` (`chmod`, `chown`, `chgrp`) prend en paramètre un *inode* ainsi que de nouvelles valeurs pour les droits, propriétaire et groupe de l'entité et ajuste les valeurs dans l'*inode* (en mémoire).
- `write_inode` est l'opération inverse de `read_inode` et modifie donc une entrée de la table des *inodes* en se basant sur l'*inode* passé en paramètre (dont les différents champs peuvent avoir changés suite à un `notify_change`).
- `put_inode` est appelé lorsque l'*inode* n'est plus utilisé pour permettre au programeur du système de fichier de libérer la mémoire qui aurait pu être allouée dans `read_inode`. En effet, le noyau ne peut se permettre d'utiliser la mémoire pour stocker un grand nombre d'*inodes*. Les différents caches ont donc une politique qui fait que de temps en temps cette opération est appelée.

1.7 Synthèse

Nous venons de voir dans ce chapitre une partie importante des travaux autour des systèmes de fichier. C'est un domaine très riche. Le système de fichier est un élément clef du système d'exploitation. Après tout, le fichier et le répertoire sont des concepts fondamentaux auquels les utilisateurs sont confrontés tous les jours. Ainsi, le système de fichier est sans doute la partie la plus visible de l'OS du point de vue de l'utilisateur. D'ailleurs, comme on l'a vu, de nombreux concepts du système d'exploitation ont leurs «avatars» visibles dans le système de fichier : les périphériques, les processus, le réseau, le système de fenêtrage, ou encore les communications inter processus.

C'est le cas aussi pour de nombreuses applications qui ont été integrées dans le système de fichier au fur et à mesure : `find`, `grep`, ainsi que d'une certaine mesure les bases de données (via SFS et les autres systèmes de fichier avancés), PGP (via cryptFS), `gzip` (via gzipFS), `rsh` (via NFS) `ftp` et les *browsers* Web (via Alex), les gestionnaires d'e-mails (via UserFS), Xwindow (via $8^{1/2}$ de Plan9), `ps` (via `/proc`), `make` (via IFS), `tar` (via IFS), ou encore RCS (via 3DFS et ses sucesseurs).

La raison de toutes ces intégrations vient du fait que placer une fonctionnalité dans le système de fichier est souvent préférable à la placer dans une application spécialisée (comme `ftp`, `gzip`, etc). En ramenant tous aux notions de fichier et répertoire, et donc en unifiant de nombreux concepts, le système est plus uniforme et donc plus simple, les outils plus génériques, et les combinaisons entre ces concepts deviennent alors possible faisant du système de fichier une terre fertile pour voir naître de nouvelles idées.

Nous nous servirons dans le reste de ce document des nombreuses notions developpées dans cette partie bibliographique, en partant de ces notions pour les étendre. Ainsi, nous partirons du principe originel du fichier et répertoire décrit Section 1.1, avec un modèle se basant sur une analogie avec des concepts du monde réel, pour proposer un autre modèle, celui de LFS, plus virtuel au Chapitre 2. Nous verrons que sous LFS les répertoires *et* les fichiers seront plus virtuels. Ils ne seront plus le resultat d'un contenu physique sur le disque mais le fruit d'un calcul. Ces «nouveaux» fichiers et répertoires permettront, comme tous les autres travaux décrits Section 1.3 intégrant une fonctionnalité dans le système de fichier, de nouvelles combinaisons, rendront les outils plus génériques. LFS est aussi un système de fichier avancé, mais il ira plus loin que les autres travaux décrits Section 1.4. Nous nous servirons de l'interface du *shell* (`cd`, `ls`, etc) décrite Section 1.2.2, pour décrire LFS sous la forme d'un tutorial dans la Section 2.3. C'est aussi en ces termes que nous decrirons la sémantique de LFS en Annexe A. Nous partirons des modéles de sécurité décrits Section 1.5 pour en proposer un plus général au Chapitre 4. Nous adapterons l'usage des liens, décrits Section 1.2.3, pour LFS au Chapitre 5 afin d'intégrer de nouvelles fonctionnalités dans LFS. Nous étendrons pour LFS au Chapitre 3 et notamment Section 3.2.2 les structures de données internes des systèmes de fichier (comme les *inodes* et la table des *inodes*), qui ont été décrites dans la Section 1.6.1. Nous décrirons précisément l'implémentation de LFS dans la

Section 3.3 en terme des opérations VFS décrites Section 1.6.2. Lorsque nous aborderons l'architecture logicielle de LFS dans la Section 6.1, nous réutiliserons l'une des techniques pour implémenter un système de fichier décrite Section 1.6.2 page 47 (l'interception après le noyau avec un clone de UserFS). Enfin, nous décrirons les méthodes de tolérance aux fautes sous LFS Section 3.3 et Section 6.1.2 qui se basent sur la technique de journalisation décrite Section 1.6.1 page 43.

Deuxième partie

Contributions

Chapitre 2

Principes

As simple as possible - but not simpler.

Einstein

La perfection est atteinte non pas quand il n'y a plus rien à rajouter mais quand il n'y a plus rien à enlever.

Voltaire

Ce chapitre présente à quoi sert LFS et qu'est ce que LFS : le «pourquoi» et le «quoi». Le chapitre suivant décrit le «comment». Ainsi, nous décrirons dans ce chapitre LFS en terme de commandes *shell*, comme `cd` ou `ls`, parce qu'elles sont orientées utilisateur et donc plus apte à transmettre le «quoi». Nous ne parlerons des véritables opérations système, comme `lookup` et `readdir`, que lorsque nous décrirons le «comment» lorsque nous entrerons plus dans les détails concrets d'implémentation de LFS au chapitre suivant.

Le «pourquoi» a déjà été evoqué en partie dans l'introduction générale et dans certaines parties du chapitre bibliographique en exposant les problèmes et limitations des outils classiques, des systèmes de fichier classiques, et même des systèmes de fichier avancés. Nous commencerons ce chapitre en poussant plus loin l'analyse de ces problèmes. Le but est qu'en reformulant de manière plus précise ces problèmes, on puisse mieux les identifier et donc mieux les comprendre, pour pouvoir en dégager les causes essentielles et donc les solutions essentielles. Nous donnerons ensuite une vision générale de LFS, ce qui permettra de voir rapidement l'ensemble de toutes les fonctionnalités offertes par LFS et de voir l'interaction entre ces fonctionnalités. Puis, nous décrirons au fur et à mesure, avec un exemple de session utilisant un *shell*, ces différentes fonctionnalités, ainsi que l'usage concret de LFS, ce qui jouera le rôle d'un tutoriel LFS. Enfin, nous décrirons plus précisément et plus formellement les grands mécanismes de LFS.

Nous présentons aussi une sémantique formelle complète de LFS en Annexe A.

2.1 Motivations

Les limitations des systèmes de fichier hiérarchiques sont le point de départ qui a motivé la conception de LFS. Les principes d'origine du répertoire et du fichier viennent de l'analogie entre les concepts du monde réel et ceux du monde virtuel. Cependant, en copiant les principes d'organisation du monde réel, on a aussi copié leurs limitations. En s'inspirant du monde réel, les systèmes hiérarchiques restent finalement des modèles très physiques et donc assez contraignants. Le but de LFS est d'offir un meilleur système de fichier en relachant les contraintes qui pèsent sur le répertoire et le fichier. Nous allons donc voir plus précisément dans les deux sections suivantes les problèmes liés à l'organisation, la recherche, et la manipulation d'un ensemble de fichiers (via entre autre des répertoires) puis à l'organisation, la recherche, et la manipulation du contenu des fichiers.

La première classe de problèmes a déjà été evoquée Section 1.4 avec la description de systèmes de fichier avancés. Ces systèmes remettent en cause le principe d'origine du répertoire. Cependant, là où les systèmes de fichier classiques privilégient la navigation (même si ils fournissent aussi un outil comme `find` pour bénéfi-

cier d'une forme d'interrogation), les systèmes de fichier avancés privilégient l'interrogation (même si certains fournissent aussi une forme de navigation par exemple avec les répertoires-vues). Comme nous allons le voir dans la section suivante, chacun de ces deux modes de recherche a ses propres avantages et aucun de ces systèmes ne fournit l'ensemble de ces avantages. Aucun de ces systèmes n'est donc vraiment meilleur qu'un autre.

La deuxième classe de problème n'a jusqu'ici pas été évoquée (ou très brièvement dans la Section 1.3.2 page 22). La raison en est qu'avec l'avancé d'Unix, les travaux autour des systèmes de fichier ne se sont pas intéréssés au contenu des fichiers, à leurs organisations internes. Comme le disent les inventeurs d'Unix Ritchie et Thompson dans [RT74] : *"No particular structuring is expected by the system [...] the structure of files is controlled by the programs that use them*".

2.1.1 Sur la gestion des fichiers

De nos jours, un utilisateur a accès à une très grande variété et à un très grand nombre de documents numériques. Gérer autant d'information est une tâche complexe ; une difficulté importante pour l'utilisateur est de trouver ou retrouver rapidement un document parmi ce très grand nombre. C'est le problème de la *recherche d'information*. En effet, avec des milliers de documents, on ne peut pas utiliser la méthode naïve consistant à les visiter tous car c'est un processus trop long.

Une des grandes découvertes du domaine de la recherche d'information (*information retrieval* [BYRN99, BDMS94]) est d'avoir identifié deux grandes méthodes de recherche : la *navigation* et l'*interrogation*. Ces méthodes de recherche sont en général fortement liées à des méthodes d'organisation ; ils forment des *paradigmes*. Ainsi, on a pour la navigation le *paradigme hiérarchique*, et pour l'interrogation le *paradigme booléen*.

Dans les deux sous-sections suivantes, nous allons présenter les avantages et inconvénients de ces méthodes de recherche et d'organisation à travers un exemple jouet. Le problème posé sera de gérer des documents représentant des cartes de villes. Nous aurons à notre disposition certaines informations sur ces villes, comme son pays, son style, ce qui permettra de les classer afin qu'un utilisateur puisse les retrouver plus rapidement. Nous verrons plus loin comment LFS propose sur ce même exemple jouet un troisième paradigme qui combine les avantages de ces deux paradigmes sans leurs inconvénients.

La navigation

Les systèmes de fichier traditionnels appartiennent au *paradigme hiérarchique*, tout comme certains portails Internet comme Yahoo ! [FY94], ainsi que de nombreux gestionnaires de messagerie ou de *bookmarks*, ou bien encore des outils comme les explorateurs de *classe* dans la programmation orientée objet. L'information est organisée en créant tout d'abord une *hiérarchie* de *concepts*, ce qui induit une structure d'arbre, puis en plaçant des *objets* dans cet arbre, ce qu'on appelle le processus de *classification*. L'information est recherchée en se déplaçant vers le haut ou vers le bas dans cet arbre ; c'est le processus de *navigation*. Par exemple, dans un système de fichier les concepts sont les répertoires et les objets les fichiers.

La Figure 2.1 illustre un exemple d'organisation pour gérer l'information sur les villes de notre exemple jouet. Ce n'est pas la seule façon d'organiser ces villes, mais comme nous le verrons plus loin, quelque soit la façon d'organiser ces villes, quelque soit l'arbre, les limitations du paradigme hiérarchique resteront les mêmes.

Avantages Le principe de navigation/classification est concret et intuitif puisqu'il reprend la façon naturelle d'organiser les choses par l'homme dans le monde réel. Par exemple, les bibliothèques sont une forme d'organisation hiérarchique ; les livres sont classés dans différentes rangées puis dans différentes étagères (organisées en domaines, sous-domaines), puis ordonnés par nom d'auteurs

Le système *propose* à l'utilisateur des aides de navigation, par exemple des noms de répertoires. Même si l'utilisateur ne se rappelle plus ou ne connaît pas précisément la classification d'un objet et le système de classification, il pourra la retrouver de lui-même en observant simplement ce que lui propose la machine.

De plus, le système propose seulement des aides de navigation pertinentes. En effet, le système ne serait pas d'une très grande aide s'il proposait directement l'ensemble tout entier des concepts, car cet ensemble peut être tout aussi grand que l'ensemble des objets. En

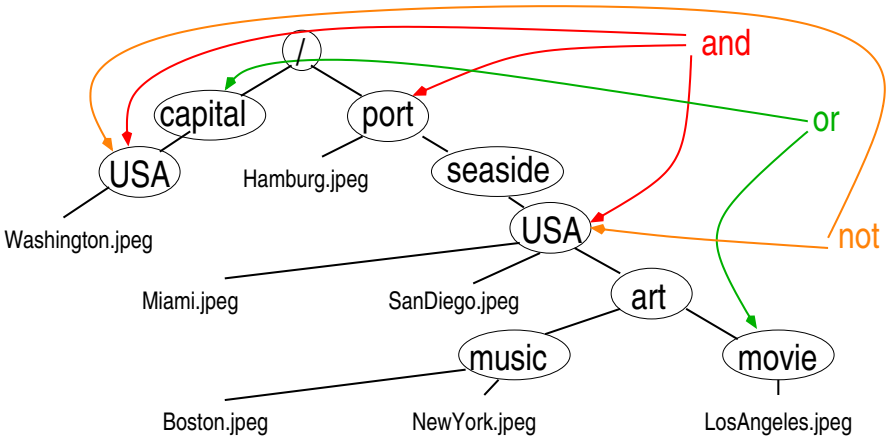

FIG. 2.1 – Organisation hiérarchique

proposant d'abord des concepts *généraux* (par exemple `art`), puis ensuite des concepts plus *spécifiques* (dans l'exemple `movie` et `music`), le système permet à l'utilisateur de retrouver *progressivement* un objet étape par étape.

Cela permet aussi à l'utilisateur de retrouver un objet rapidement. En effet, à chaque étape, l'utilisateur a le choix entre un nombre limité de concept, ce qui est très important car moins il y a de choix, plus facile est le choix. De plus, chaque étape réduit de manière significative le nombre d'objets possibles répondant aux critères, ce qui permet à l'utilisateur de converger très vite vers la solution.

Inconvénients Un certain nombre de requêtes ne peuvent être exprimées. Dans l'exemple, l'utilisateur ne peut pas faire positionné à la racine `cd USA/` car le concept `USA` est éparpillé un peu partout dans l'arbre. Ce problème est bien connu sous Unix avec la prolifération de répertoires `bin` ce qui oblige l'utilisateur à une recherche exhaustive dans chacun de ces répertoires lorsqu'il veut trouver un programme.

De plus, le système n'autorise qu'une forme très réduite de requêtes conjonctives et n'autorise ni les disjonctions, ni les négations. En effet, même si la requête `cd port/seaside/USA/` est possible, et peut être vu comme une conjonction, la requête équivalente `cd USA/port/seaside/` elle n'est pas possible. De même, ni la requête `cd !USA`, ni `cd capital|port` ne sont autorisées. Ces limitations font que même si l'utilisateur sait certaines choses sur ce qu'il cherche, comme le fait que la ville ne se trouve pas aux USA, et que c'est soit une capitale soit un port, le système ne pourra pas utiliser ces connaissances. Or très souvent l'utilisateur ne sait pas précisément ce qu'il cherche, et la disjonction et la négation permettent justement d'exprimer ce genre de choses. Elles permettent, malgré cette notion d'imprécision, de réduire de manière significative les possibilités. Ainsi dans l'exemple précédent, une seule ville répondait aux critères : `Hamburgh`.

On pourrait penser que pour permettre certaines requêtes comme `cd USA/`, il suffisait simplement de changer l'agencement des répertoires en plaçant directement à la racine un unique répertoire `USA`. Mais dans ce cas, ce serait à leur tour d'autres requêtes qui deviendraient impossibles.

En fait, le problème sous-jacent est que tout comme dans la realité, on ne peut placer un objet qu'à un seul endroit, ce qui veut plus ou moins dire que l'on ne peut associer à un objet qu'un seul concept. Les utilisateurs surmontent cette limitation en se servant de la hiérarchie pour représenter la conjonction, mais ce faisant, ils ordonnent entre eux des concepts orthogonaux. Autant il paraît logique que `movie` soit un sous-répertoire de `art`, car on peut voir le concept `movie` comme une forme de sous-concept de `art`, autant être aux `USA` n'a

rien à voir avec le fait d'être un `port`. Cette sorte de tricherie conduit forcément au choix d'un schéma où l'on privilégiera une classification au détriment d'une autre. Dans notre exemple on a donné beaucoup d'importance au fait d'être ou non une capitale. Malheureusement, aucun schéma ne peut satisfaire tous les besoins, et cette rigidité et inexpressivité de l'organisation se retrouve dans l'opération de recherche.

Renoncer à associer différents concepts à un fichier est encore pire. C'est pourtant ce que finit par faire l'utilisateur face à la rigidité du système ; il finit par se satisfaire d'un seul critère, d'une seule classification. Ce problème est d'autant plus important de nos jours avec le multimédia. On aimerait classer les fichiers musicaux par artiste, genre, année de production, studio, les photographies par année, endroit, personnes présentes sur la photo, heure de la journée. Être obligé de choisir une classification et d'oublier les autres fait que dans le futur l'utilisateur ne pourra plus accéder à un document en utilisant une classification non prise en compte, et donc ce document est virtuellement perdu. De plus, comme ces documents diffèrent souvent les uns des autres seulement par peu d'aspect, ne pas être capable de les décrire précisément (en mentionnant tous les concepts auxquels le document appartient) fait que beaucoup de documents se retrouveront dans le même répertoire. Ainsi, l'utilisateur sera forcé d'explorer ce grand nombre de documents qui n'auront malheureusement pas été mieux classés.

Certains systèmes hiérarchiques comme les systèmes de fichiers proposent une solution partielle à ce problème : les liens. Ainsi, un utilisateur pourrait avoir différentes classifications clairement séparées, et non pas ordonnées entre elles comme c'est souvent le cas, et des fichiers placés cette fois dans de multiples répertoires.

Une autre solution serait d'encoder dans le nom du fichier un ensemble de concepts (ce qui est déjà plus ou moins le cas avec l'extension du fichier permettant de spécifier le type de document), et d'utiliser un outil comme `find` pour trouver un fichier. Cela dit, comme nous le verrons dans la section suivante, cela reviendrait essentiellement à utiliser une structure booléenne, et nous perdrions dans ce cas tous les avantages de la navigation. En effet, avec les liens, même si l'utilisateur pourrait cette fois exécuter `cd USA/` ou `cd port/` de la racine, aucun sous-répertoire ne lui serait proposé pour compléter ou affiner sa recherche.

L'interrogation

Les moteurs de recherche sur Internet comme AltaVista [SRR97] ou Google [BP98] appartiennent au *paradigme booléen*, tout comme les systèmes de recherche comme WAIS [KM91] ou d'un certain coté les utilitaires comme `find` et `grep`. L'information (dans le cas du *Web* ce sont des fichiers HTML) est organisée en associant des mots-clefs à chaque document. L'information est recherchée en spécifiant une liste de mots-clefs que la description du document doit contenir. Cette méthode peut être étendue avec l'emploi de formules booléennes comme dans $(\textit{port} \vee \textit{USA}) \wedge \textit{capital}$.

Les systèmes de fichier avancés comme SFS [GJSJ91] peuvent être vus aussi comme appartenant au paradigme booléen. Les mots-clefs sont remplacés par des attributs valués (assignés parfois manuellement, parfois automatiquement au fichier) et les requêtes peuvent contenir des opérations relationnelles sur ces valeurs comme dans `size>100`.

La Figure 2.2 illustre l'organisation booléenne pour gérer l'information sur les villes de notre exemple jouet.

Avantages Contrairement au paradigme hiérarchique, la possibilité de décrire un document à l'aide d'une conjonction de concepts est supportée depuis le début puisqu'elle constitue le principe même de ce genre d'organisation. Ce paradigme rend alors possible l'indexation automatique des objets (fichiers, pages Web) et l'utilisation de propriétés très différentes pour retrouver un fichier. On peut par exemple rechercher un fichier (ou une page Web) en spécifiant les mots que le fichier doit contenir. Cette méthode de recherche est flexible du fait que les opérations booléennes sont commutatives ; les requêtes $\textit{port} \wedge \textit{USA}$ et $\textit{USA} \wedge \textit{port}$ sont équivalentes, ce qui paraît logique, alors qu'elles ne le sont pas sous un système hiérarchique. En effet, `port/USA` et `USA/port` ne sont pas du tout équivalents. Cette méthode est aussi plus expressive, elle supporte la conjonction mais aussi la disjonction et la négation.

De plus, lorsque l'on sait très précisément ce que l'on cherche, l'interrogation est plus rapide car on peut aller directement au but, sans avoir à traverser une multitude d'intermédiaires, par exemple en mentionnant `movie` sans avoir à passer avant par `art`.

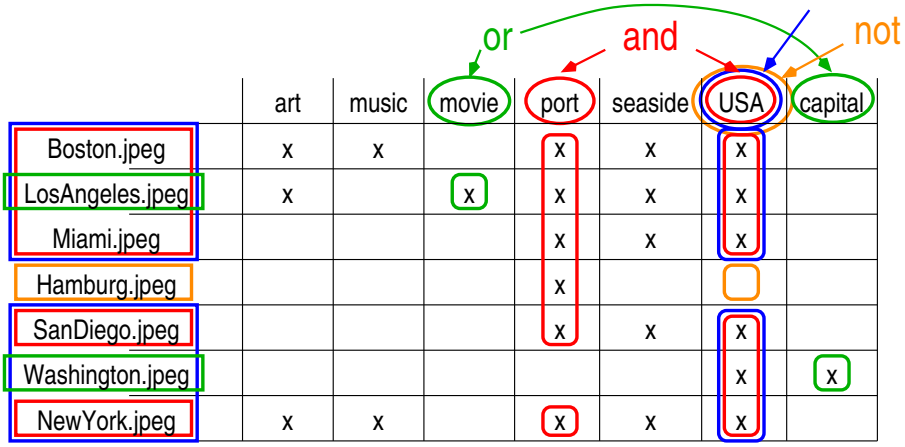

FIG. 2.2 – Organisation booléenne

Inconvénients Malheureusement ces avantages ont leur prix. Contrairement au paradigme hiérarchique, le système ne répond pas suite à une requête, par un ensemble de mots-clefs complémentaires (sous-répertoires) permettant d'affiner cette recherche. À la place, le système retourne comme réponse une liste de documents dont la description satisfait la requête. Ainsi, les mots-clefs permettant de trouver un document doivent être devinés par l'utilisateur, et ce de plus de manière précise ; cette fois le système n'aide pas.

Puisque le système retourne une liste de documents, le résultat d'une requête peut être très long, ce qui peut vouloir dire inutile car l'utilisateur ne peut et ne veut pas parcourir cette longue liste pour trouver le document désiré. Ainsi, dans notre exemple, le résultat de la requête *port* $\wedge$ *USA* retourne quasiment la totalité des objets, ce qui n'est pas d'un très grand secours pour l'utilisateur. Bien sûr, dans cet exemple jouet le nombre des documents reste «gérable», mais dans la réalité, ce n'est pas forcément le cas. Ainsi, il n'est pas rare avec Google de recevoir comme réponse à une requête des milliers de pages *Web*. Cela veut dire que l'utilisateur doit encore trouver de lui-même des mots-clefs permettant de restreindre les possibilités. Cela mène très souvent à un va-et-vient entre une liste de réponse quasi vide, et une liste contenant un trop grand nombre de réponses.

Sous une organisation hiérarchique, le principe de rangement en concepts et sous-concepts (par exemple en répertoires et sous répertoires) permet d'avoir dans le résultat d'une requête, c'est à dire dans le contenu d'un répertoire, un nombre très limité de documents et de sous-concepts. Premièrement, car il y a souvent peu de sous-concepts associés à un concept plus global. Deuxièmement, les fichiers sont souvent placés aux feuilles de l'arbre, et si la classification est bien faite, il n'y aura pas des centaines de fichiers avec exactement la même description, c'est à dire dans le même répertoire.

L'interrogation rend donc difficile pour l'utilisateur de trouver un document lorsqu'il n'a seulement qu'une vague idée de ce qu'il cherche. Il est reconnu qu'il est plus facile de reconnaître ce que l'on cherche, que d'exprimer ce que l'on cherche, c'est à dire plus facile de naviguer, que d'interroger. Ainsi, on aimerait que les systèmes booléens proposent aussi des mots-clefs complémentaires qui permettraient d'affiner la recherche ; on voudrait pouvoir naviguer après une interrogation.

Il faut noter que nous n'avons pas évoqué un autre grand paradigme de la gestion de l'information : le *paradigme relationnel* [Cod70, Dat99], avec par exemple les bases de données. La raison en est que ce paradigme du point de vue de la recherche d'information n'est pas fondamentalement différent du paradigme booléen. Dans le paradigme relationnel, la table unique est remplacée par une multitude de tables. Pour la recherche, la logique des propositions est remplacée par un langage plus complexe comme SQL. Même si ce modèle est plus expressif que

le modèle booléen, c'est encore un modèle basé uniquement sur l'interrogation, et donc où les problèmes évoqués précédemment existent toujours. La recherche d'information dans les bases de données n'est ni intuitive, ni progressive.

Conclusion

Comme nous venons de le voir, chacun de ces paradigmes a ses avantages et inconvénients. En fait, les avantages de l'un répondent précisément aux inconvénients de l'autre. On peut résumer la situation à l'aide du Tableau 2.1.

On peut voir l'interrogation et la navigation comme deux phases *duales* d'un dialogue homme-machine où chacun instruit l'autre :

- Pour l'interrogation, *c'est l'utilisateur qui instruit la machine sur ce qu'il veut*. Il est alors important que l'utilisateur puisse exprimer ce qu'il veut et ce qu'il sait sur ce qu'il cherche. Cela passe par exemple par la possibilité d'utiliser la conjonction, la disjonction ou la négation, et par le fait que les fichiers soient précisément décrits par de nombreuses propriétés.
- Pour la navigation, cette fois *c'est la machine qui instruit l'utilisateur*. La machine propose à l'utilisateur des compléments de requêtes pertinents pour qu'il progresse dans sa recherche. De plus, il est important que la machine propose d'abord des notions générales puis spécifiques ce qui permet d'éviter de proposer un trop grand nombre de compléments qui noyeraient l'utilisateur.

Le système de recherche idéal est un système où ce dialogue reste toujours possible. Par exemple, un utilisateur ne sachant pas précisément ce qu'il cherche et les termes à employer peut commencer par naviguer. Au fur et à mesure de sa navigation il peut mieux comprendre ce qu'il cherche et alors interroger pour accélerer la recherche. Cependant, si jamais il récupere trop de réponses, il peut choisir de renaviguer pour affiner sa recherche.

Il faut bien comprendre que fournir ces deux formes de recherche n'est pas suffisant. Il faut pouvoir les combiner librement. Ainsi, même si les systèmes de fichier avancés proposent différentes formes d'intégration entre l'interrogation et la navigation, aucun de ces systèmes ne permet suite à une nouvelle interrogation de naviguer intelligemment. Le dialogue se rompt à un moment ou à un autre. Sous Nebula [BDB+94] et HAC [GM99], l'utilisateur peut interroger ou bien naviguer avec les répertoires-vues. Cependant, suite à une nouvelle requête, le système ne propose plus de répertoires-vues. Sous SFS [GJSJ91], le système propose certes à l'utilisateur des noms d'attributs possibles (avec la commande `ls FIELD`:) puis des valeurs possibles pour ces attributs, mais il les propose toujours toutes, quelque soit la requête. Les aides de navigation ne sont ainsi pas toujours pertinentes et peuvent même mener à une impasse. CATFS [Gia93] propose sans doute la forme d'intégration la plus interessante avec la commande `ls ATTR`:. Cette commande permet d'afficher, suite à n'importe quelle interrogation, les propriétés que possèdent les fichiers satisfaisant la requête courante. Cela permet ainsi d'inférer des compléments de requêtes possibles. Cependant, du fait d'un formalisme basé sur les attributs valués, il n'y a pas de notion de général et spécifique ce qui fait qu'à la racine du système, CATFS propose toutes les propriétés ce qui rend la navigation difficile.

Nous verrons plus loin comment LFS sur ce même exemple intègre les deux formes de recherche. Sans rentrer dans les détails, LFS permettra notamment les actions suivantes :

```
[1] % cd /lfs
[2] % cd port&USA/; ls
art/
miami.jpg  sandiego.jpg
[3] % cd art/; ls
movie/ music/
[4] % cd movie/; ls
los-angeles.jpg
[4] % cd /lfs/(capital|port)/!USA/
[5] % ls
hamburg.jpg
```

2.1.2 Sur la gestion du contenu des fichiers

Après avoir organisé ses fichiers et recherché des fichiers, l'utilisateur est amené à manipuler et travailler sur le contenu de ses fichiers. Les activités typiques d'un utilisateur lorsqu'il manipule un document numérique sont :

	Flexible	Expressive	Intuitive	Progressive
Interrogation	Oui	Oui	Non	Non
Navigation	Non	Non	Oui	Oui

TAB. 2.1 – Interrogation et Navigation

1. chercher (encore) des données spécifiques,
2. lire et comprendre ces données,
3. modifier ces données.

On retrouve par exemple ce genre de situation avec le travail sur des sources de programmes (C, Perl, etc) ou de livres (LaTeX), avec les bases de données textuelles comme les fichiers BibTeX ou les agendas éléctroniques, ou plus récemment avec les pages Web et les documents XML.

Les difficultés pour éditer ces documents arrivent lorsque la taille du document augmente. En effet, la recherche d'une entité, que ce soit par exemple le début d'une fonction pour un programme ou d'un chapitre pour un livre, prend forcément plus de temps lorsque le nombre de ces entités augmente. La méthode de recherche naïve qui consiste à *scroller* dans le fichier peut prendre du temps. De même, la bonne compréhension d'un morceau du document nécessite en général de bien en comprendre toutes ses interactions avec le reste du document, et donc d'un certain côté de bien comprendre la globalité du document, ce qui est là encore plus difficile lorsque la taille de celui-ci augmente. Enfin, la modification est elle aussi plus difficile du fait de ces nombreuses interactions entre les parties d'un document ; la modification d'une partie a en général un impact sur d'autres parties, qu'il faut retrouver et à leur tour modifier.

Heureusement ces documents ne sont pas faits d'un immense bazar et ont en fait une structure interne, définit par le créateur du document, qui permet de gérer cette complexité en fournissant des repères. Cette structure *implicite* peut être exploitée par des outils pour assister le créateur ou d'autres utilisateurs souhaitant travailler sur ce document.

La structure de ces documents est en général complexe. Elle est tout d'abord souvent hiérarchique avec donc de multiples niveaux, par exemple des parties, chapitres puis sections pour le source d'un livre, des modules, classes, sous-classes, puis fonctions pour la programmation, ou bien encore des années, mois, et jours pour un agenda électronique.

D'autres structures, encore plus implicites, viennent ensuite traverser de part en part cette hiérarchie. En effet, tout comme le chapitre d'un livre permet de regrouper au même endroit des choses traitant du même sujet, il existe d'autres regroupements possibles selon d'autres *points de vues* et critères plus subtiles. Ainsi, on retrouve des notions revenant régulièrement dans le document mais qui sont éparpillées. Par exemple, le fait d'avoir une introduction et une conclusion revient régulièrement dans différents chapitres, le nom de la même méthode peut revenir régulièrement dans le source de différentes classes, un même type de rendez-vous peut revenir régulièrement dans différents mois, ou encore le même auteur peut revenir régulièrement dans différentes entrées BibTeX.

La Figure 2.3 illustre de manière symbolique la structure de ces différents documents pour 2 niveaux. Les différents quadrillages peuvent par exemple correspondre à différents chapitres ou différentes classes selon le type de document (source de programme ou source de livre), et les bandes de couleur à un sous-niveau avec différentes méthodes ou sections. Deux bandes de couleurs dans différents quadrillages peuvent partager la même couleur. En effet, les couleurs peuvent correspondre à des notions transversales revenant régulièrement comme une introduction, un développement puis une conclusion, ou bien le code de méthodes portant le même nom mais dans des classes différentes.

Il faut noter que si l'on zoomait sur cette structure, par exemple en développant une des couleurs d'un des quadrillages, on retrouverait pour certains documents à une autre échelle ce même schéma, tel une *fractale*. Le code d'une méthode peut par exemple être encore décomposé en différents niveaux, avec différents quadrillages pour

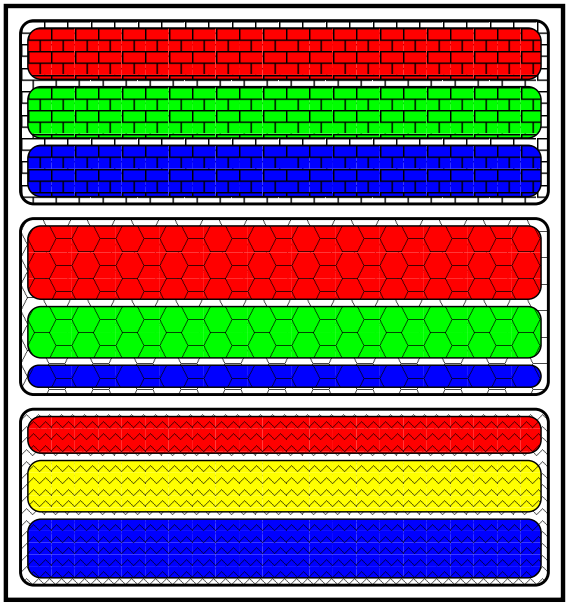

FIG. 2.3 – Représentation symbolique du contenu d'un fichier

la documentation, le type, puis le code de cette méthode, le code pouvant encore être lui même décomposé avec différentes bandes de couleurs correspondant à différents blocs d'instructions (puis les instructions avec différentes expressions, sous-expressions, etc). On retrouve là aussi à cette échelle des notions transversales. Le fait d'avoir de la documentation pour une méthode se retrouve par exemple partout. Plus on descend dans la structure, plus des notions transversales apparaissent, encore plus éparpillées. De la même manière pour les couleurs avec les blocs d'intructions, les *aspects* de déboggage, d'assertion, d'optimisation, de gestion d'erreur ou d'allocation mémoire peuvent se retrouver plusieurs fois tout au long du code de la même méthode. En fait, si on zoomait en arrière, ces aspects se retrouveraient même plusieurs fois tout au long du code de l'ensemble d'un projet.

La structure d'un document est donc très complexe. Elle possède de nombreux *niveaux* et de nombreuses *facettes*.

Nous allons maintenant voir comment les outils existants pour ces différents types de document exploitent cette structure pour aider l'utilisateur à rechercher, comprendre et modifier le document.

Rechercher

Comme dit précédemment, la méthode de recherche naïve qui consiste à se déplacer dans le fichier en *scrollant* continuellement d'un bout à l'autre peut prendre du temps. L'utilisateur peut là encore, cette fois pour la recherche à l'intérieur de documents, utiliser les deux techniques de recherche classiques : la *navigation* et l'*interrogation*.

Comme dit précédemment, de nombreux documents ont notamment une structure hiérarchique. Une méthode très largement utilisée, et universelle, pour naviguer plus facilement à l'intérieur d'un document est d'utiliser justement les services de navigation des systèmes de fichier hiérarchiques. On sépare le document en plusieurs entités, dans différents fichiers voir même ensuite dans différents répertoires. Par exemple, avec un document LaTeX, l'utilisateur peut avoir un fichier principal *incluant* ensuite différents fichiers correspondant à différents chapitres. Ces «fichiers-chapitres» peuvent même à leur tour inclure d'autres fichiers disposés dans des répertoires correspondant au nom de ce chapitre, et ainsi de suite sur plusieurs niveaux. Ce principe est largement pratiqué en programmation où l'on organise le source d'une application (qui peut être vue comme un gros document) en sous-projets, puis pour chaque sous-projet en différents modules ou différentes classes. Par exemple, les sources de Linux s'étalent sur de nombreux répertoires et sur de nombreux fichiers. Bien sûr, cette structuration peut être motivée par des besoins de compilations séparées, mais les besoins de navigation pèsent aussi sur cette décision. L'utilisateur peut ainsi, au lieu de *scroller*, naviguer sur son document plus efficacement en utilisant le *shell* pour se déplacer dans les répertoires puis en ouvrant un des fichier composant le document pour aller plus directement à ce qu'il cherche. Une méthode moins universelle pour naviguer consiste à utiliser un outil particulier au type de document. Par exemple, les explorateurs de classe à la Smalltalk [Gol84] pour les environnements de développement, ou les tables des matières *hypertexte* pour les

applications de traitement de texte.

Au lieu de naviguer, l'utilisateur peut parfois préferer interroger. Une méthode très largement utilisée, et universelle, est tout simplement la recherche d'une chaîne de caractère dans un fichier par le biais d'un bouton de l'application gérant le document (comme `C-s` sous Emacs). Cette technique peut être étendue avec l'utilisation d'expressions régulières.

On pourrait penser que cette technique rend obsolète la décision précédente de séparer le document en plusieurs fichiers. L'ensemble du document pourrait être placé dans le même fichier mais l'utilisateur pourrait tout de même se déplacer rapidement dans le fichier de chapitre en chapitre, de section en section en mentionnant le nom de l'entité qu'il cherche dans une recherche. Cependant comme on l'a vu auparavant, l'interrogation n'est pas la navigation. Chacun de ces modes a ses propres avantages. Ainsi, très souvent l'utilisateur ne se rappelle plus précisément le nom d'un chapitre, d'une section, ou d'une fonction auquel cas il préfère qu'on le lui rappelle.

Il n'est cependant là encore pas possible de combiner interrogation et navigation. On aimerait pouvoir lancer une recherche de chaîne de caractère sur le document et se voir proposer comme compléments de requête les noms de sections de la table des matières couvrant les parties du document contenant effectivement cette chaîne. Cela permettrait d'accélérer la recherche puisque cela éviterait à l'utilisateur d'avoir à parcourir une par une chacune des occurrences de cette chaîne en allant directement à la section en question.

Certains systèmes proposent des interrogations plus avancées que la recherche de chaîne, qui est sémantiquement pauvre, comme [Rit89] qui à l'aide d'une *logique de type* permet à l'utilisateur de retrouver plus facilement une fonction. D'autres systèmes [PD90] offrent une navigation plus avancée avec de multiples classifications pour les fonctions (classifications appelées *facettes*). Cependant, il n'est là non plus pas possible de combiner ces outils.

Comprendre

La structuration du document résulte surtout, plus que du besoin de navigation, du besoin de comprendre et faire comprendre un document. Elle permet de gérer la complexité du document. On localise ; on essaye de séparer le plus possible en différents groupes les choses indépendantes pour que l'on puisse ensuite travailler sur une partie sans avoir à penser au reste du document (*separation of concern* [Par72]). On *divise pour régner*. Le fait de regrouper au même endroit les choses parlant du même sujet implique que le travail et la compréhension de ce sujet est plus facile que si les discussions autour de ce sujet étaient éparpillées sur tous le document. Lorsque l'on arrive à trop de groupes, trop de discussions, on sépare là encore les différents groupes en thèmes plus généraux, que l'on hiérarchise ainsi de suite. On *abstrait pour régner*. L'abstraction à différents niveaux est l'outil classique de l'humain pour gérer la complexité.

La structure joue donc un rôle clef pour comprendre un document. Cependant, on peut parfois ne plus la voir car elle peut être noyée par le texte. Ainsi, pour mieux comprendre un document une pratique courante est de proposer des *vues* incomplètes mais plus simples du document pour rendre plus explicite cette structure.

Pour mieux voir la structure hiérarchique, une famille populaire de vues est obtenue en voyant le document à différentes profondeurs. On zoome en arrière ou en avant sur le document. La table des matières d'un livre offre une vue superficielle mais aide à comprendre le document en donnant une *vue d'oiseau* du document. Les structures de navigation comme les explorateurs de classes sont donc déjà des formes de vues. En plus d'aider la navigation elles aident la compréhension.

Ces structures peuvent être vues à la fois comme un moyen de trouver une partie du document ou de manière duale de cacher une partie du document. En fait, le terme de *vue* n'est pas forcément adéquat puisqu'au contraire le but de ces vues est de ne pas voir. On peut mieux voir quelque chose en le mettant en valeur en cachant les autres choses.

Emacs [Sta02] fournit un moyen pour rendre invisibles des parties du document. Le texte de ces parties est remplacé et résumé à l'écran par une *marque d'absence* : trois petits points «...». Ce mécanisme est exploité par de nombreux *modes* sous Emacs. L'un de ces modes, le *outline-mode*, permet ainsi pour une grande gamme de documents de zoomer en avant ou en arrière sur le document. Ce mode s'appuie sur le fait que la plupart des documents possèdent des entêtes ou titres facilement identifiables. Il suffit de spécifier à l'aide d'une expression ré-

gulière pour un type de fichier le format de ces titres et de leurs assigner une profondeur (par exemple pour LaTeX la profondeur 1 si la ligne contient `chapter`, 2 si elle contient `section`, etc). Emacs fournit ensuite un jeu de commandes qui permet d'avoir une vue où uniquement les entêtes d'une certaine profondeur sont affichés (par exemple uniquement les entêtes de chapitres pour LaTeX, ou les entêtes de fonctions pour les programmes), et permet de développer ou cacher (plier/déplier) certaines branches de la structure hiérarchique (par exemple afficher le corps d'un certain chapitre).

Cependant, ce mode ne comprend que la structure hiérarchique du document. À une certaine profondeur, de nombreuses vues peuvent aussi être définies. Par exemple, on pourrait montrer uniquement la spécification d'un programme, cacher son code de déboggage, ses commentaires, ou montrer uniquement les fonctions partageant une certaine variable. Chacune de ces vues aide à comprendre un aspect du programme et aide le lecteur à se concentrer sur une tâche puisqu'il n'est pas visuellement perturbé par des détails sans importance. En effet, une partie très importante pour la compréhension est tout simplement de voir uniquement l'information pertinente à ce que l'on essaye de comprendre.

Emacs offre toute l'artillerie pour cacher n'importe quelle partie du document, et donc théoriquement permet d'accéder à n'importe quelle vue, mais Emacs n'offre aucun mécanisme pour spécifier *facilement* les parties que l'on veut cacher. Par exemple, on ne peut facilement avoir accès à une vue où toutes les sections conclusion d'un document sont réunies, ou plus symboliquement dans la Figure 2.3 une vue où toutes les parties rouges sont réunies, si ce n'est en dépliant manuellement ces sections une par une (ou en ouvrant différentes fenêtres en même temps sur le même document couvrant différentes parties de ce document). En fait, il existe une infinité de vues potentiellement intéressantes, et il convient donc d'offrir des mécanismes pour trouver ces vues, et donc d'interroger ou naviguer parmi les vues possibles.

Beaucoup de travaux se sont attachés à permettre à l'utilisateur d'écrire de multiples vues graphiquement différentes concernant des aspects très variés du même projet comme c'est le cas avec Pecan [Rei84] ou plus récemment avec UML [Kru95, RJB99]. Cependant, ces travaux sont trop ambitieux car il est difficile d'établir ensuite des liens entre ces vues (même si ils existent effectivement) et donc de pouvoir modifier ces vues, ni même d'utiliser à bon escient ces vues pour générer par exemple du code. On ne peut donc pas vraiment les appeler des vues. Nous pensons que l'approche inverse, c'est à dire de partir du document pour en extraire des vues est certes moins ambitieuse mais plus effective, et a un champs d'application plus large que la programmation.

Modifier

Après avoir trouvé la partie du document que l'on veut manipuler, apès l'avoir comprise, l'activité finale est de la mettre à jour. La structuration du document aide aussi sa modification. Le fait de regrouper au même endroit les choses parlant du même sujet implique que le travail autour de ce sujet est plus facile. Cependant, il existe aussi les structures transversales. La modification d'une partie peut donc avoir tout de même un impact sur les autres parties.

Pour aider l'utilisateur à modifier un fichier de manière transversale, les outils comme Emacs permettent par exemple de remplacer une chaîne par une autre sur tout le fichier. Par exemple, lorsque l'on veut changer le nom d'une fonction, il faut aussi changer tous les endroits où l'on appelle cette fonction. Emacs permet aussi de définir des *macro* pour automatiser facilement une tâche répétitive.

Pour des modifications et des besoins de cohérence plus subtiles, les vues, transversales ou de zoom, peuvent être très utiles. Ainsi, en rassemblant par exemple toutes les conclusions d'un livre dans la même vue, l'utilisateur peut plus facilement les mettre à jour de manière cohérente. En effet, une incohérence de style peut plus facilement sauter aux yeux. Pour la programmation, en reprenant l'opposition entre la programmation objet et fonctionnelle, une vue fonctionnelle sur un projet objet peut permettre de plus facilement modifier de manière cohérente une opération implémentée dans différentes classes, par exemple lorsque l'interface de l'opération doit changer pour prendre un paramètre en plus. De manière opposée, une vue objet sur un projet fonctionnel facilite la modification de différentes opérations fonctionnant sur le même type de structure de données lorsque par exemple la structure de donnée évolue. De la même manière, en voyant d'un seul coup d'œil la table des matières, on peut se rendre compte plus facilement

par exemple d'incohérences entre les titres de chapitre.

De plus, les opérations élémentaires de modification décrites précédemment peuvent être plus faciles avec les vues. En effet, on aimerait parfois remplacer un mot par un autre uniquement sur une portion du document. De même, la programmation de macros ou de scripts de modification peut être plus facile sur ces vues, car elles simplifient le document et cachent des parties qui pourraient perturber ou rentrer en conflit avec la programmation. Enfin, le déplacement de grande partie de texte, comme le déplacement d'une section d'un chapitre à un autre peut être rendue plus facile en effectuant cette opération à un plus haut niveau sur la table des matières (lorsque cela est permis).

Cependant, selon la tâche il faut avoir la bonne vue et comme dit précédemment l'utilisateur n'a souvent le choix qu'entre très peu de vues. De plus, dans de nombreux cas, les outils supportant ces vues ne permettent pas leurs modification. Par exemple, la modification du document produit par un outil de programmation littéraire n'aura aucun impact sur le document original qui contient le code et la documentation.

Conclusion

Nous venons de voir que la plupart des documents numériques possèdent une structure interne complexe, faite de différents *niveaux* et de différentes *facettes*. Il existe une multitude d'outils, chacun travaillant sur un type de document particulier, sur un niveau particulier, sur une facette particulière, permettant une forme d'interrogation, de navigation, ou de vue particulière. Il faut noter que chacun de ces outils nécessite un effort de programmation important. Cependant, il n'est pas possible de combiner ces outils, que ce soit la combinaison de deux outils d'interrogation entre eux, de deux outils de navigation, de deux outils spécifiant deux vues différentes pour pouvoir former une troisième vue, ou bien d'un outil d'interrogation avec un outil de navigation.

Le problème avec tous ces outils est qu'ils manquent de principes généraux partagés. Il manque un formalisme commun sous-tendant ces outils qui rendrait possible leurs combinaisons.

Il faut noter que proposer une plateforme commune où divers outils sont intégrés, comme c'est le cas avec Emacs [Sta81] ou plus récemment Eclipse [com00], n'est pas suffisant. Même si ces plateformes fournissent des services appréciables, surtout pour les documents de type source de programme, elles ne fournissent aucun mécanisme pour pouvoir combiner l'usage de ces outils. En faisant un parallèle avec le monde des outils plus conventionnels, inventer le principe de système d'exploitation qui permet de lancer plusieurs outils, puis y intégrer les outils `find` et `grep` est une chose, inventer ensuite la notion de *pipe* pour pouvoir combiner facilement l'usage de ces outils, ce qui décuple l'intérêt de chacun de ces outils, en est une autre (les intégrer dans le système de fichier comme avec SFS pour permettre encore plus de combinaisons en est encore une autre).

Sans rentrer dans les détails, LFS permettra notamment les actions suivantes en prenant comme exemple le travail sur le document symbolique (voir aussi Figure 2.4) :

```
[1] % ls
quadrillage/ couleur/
entete/
symbolic.doc
[2] % ls couleur/
bleu/ rouge/ vert/ jaune/
quadrillage/ entete/
symbolic.doc
[3] % cd vert; ls
brique/  abeille/
entete/
symbolic.doc
[4] % emacs brique/symbolic.doc
[5] % emacs !jaune/brique|vague/
          symbolic.doc
[6] % cat entete&abeille/symbolic.doc
\chapitre{abeille}
\séction{rouge}
\séction{vert}
\séction{bleu}
[6] % cat brique&rouge/symbolic.doc
\séction{rouge}
bla bla bla
les briques rouges
c'est beau.
```

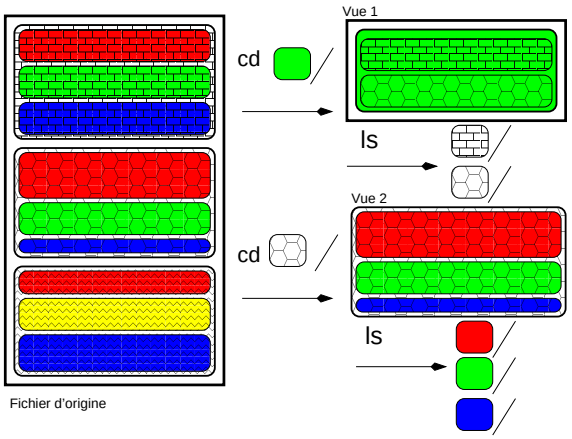

FIG. 2.4 – Navigation et vues sur le contenu d'un fichier

2.2 Vision générale

Le but de LFS est d'offrir une organisation expressive, une recherche combinant étroitement interrogation et navigation, et une facilité de manipulation, à la fois des fichiers et de leurs contenus ; le tout d'une façon intégrée, au niveau système de fichier.

Pour atteindre cette intégration, LFS associe des propriétés logiques aux fichiers et parties de fichiers, et la déduction logique sert de base pour naviguer et interroger. Sous LFS *les chemins sont des formules*, ce qui veut dire que les répertoires représentent des requêtes logiques ; ils déterminent des ensembles de fichiers et de parties de fichiers dont la description *satisfait* la requête. Le répertoire racine représente la formule *vrai*, et les sous-répertoires d'un répertoire sont déterminés par les propriétés les plus générales permettant de *raffiner* la requête. En effet, le principe de la navigation est de progresser, d'où la condition de raffinement, et de progresser doucement, d'où la recherche des propriétés les plus générales. Cela permet de combiner interrogation et navigation.

Le contenu des fichiers est déterminé par les parties du fichier original qui satisfont la requête. Cela permet des accès simultanés en lecture et écriture sur différentes *vues* d'un même fichier, afin d'aider l'utilisateur à séparer et discerner les différents *aspects* de l'information qui sont contenus dans ce fichier.

Les propriétés peuvent être attachées aux fichiers manuellement par l'utilisateur, ou automatiquement extraites du fichier lui-même par des programmes appelés *transducteurs*. Les propriétés peuvent être ordonnées manuellement par l'utilisateur pour former des taxinomies, ou automatiquement par des *moteurs de déduction logique*, permettant de naviguer des propriétés générales vers les plus spécifiques. Les utilisateurs peuvent étendre dynamiquement le système en fournissant leurs propres moteurs de déduction logique et transducteurs par un système de *plug-ins*.

Tout ceci fait intervenir de nombreux mécanismes :

1. Le *mécanisme d'organisation*, permettant aux formules d'un langage logique d'être ordonnées et attachées à l'information manuellement ou automatiquement, et qui peut être étendu par l'utilisateur.
2. Le *mécanisme de recherche*, avec l'interrogation, qui compare les formules constituant le chemin avec les formules qui décorent l'information, et la navigation, qui propose des sous-répertoires après une requête.
3. Le *mécanisme de manipulation*, permettant à l'utilisateur de déplacer, effacer et copier des fichiers ou parties de fichiers en utilisant les commandes traditionnelles d'un système de fichier, ce qui a pour effet de modifier leurs propriétés.

La section suivante illustre concrètement, à travers un long exemple, ces différents mécanismes. Les 3 sections suivant cet exemple s'attacheront à préciser plus formellement le fonctionnement de chacun de ces mécanismes. L'Annexe A va encore plus loin puisqu'elle donne une sémantique formelle complète de LFS.

2.3 Tutoriel

Un exemple de suite de commandes *shell* utilisant LFS pourrait être :

```
[1] % mkfs.lfs /home/pad/meta_lfs
[2] % mount.lfs /home/pad/meta_lfs /lfs
[3] % cd /lfs
```

La commande `1` réserve un espace sur le disque où résideront les structures de données utilisées en interne par

LFS. La commande 2 «monte» cette structure, via LFS, sous le point de montage `/lfs`. Cela veut dire que toutes les commandes lancées depuis ce point de montage seront gérées par LFS. En effet, LFS est un outil qui fonctionne via l'interface des systèmes de fichier, et comme il modifie la sémantique habituelle des opérations d'un système de fichier, il modifie par effet de bord la sémantique des commandes qui utilisent ces opérations.

2.3.1 Sur les fichiers

Nous allons voir dans cette section comment organiser, chercher et manipuler des fichiers à l'aide de LFS.

Organiser des fichiers

```
[4] % mkdir art
[5] % mkdir music
[6] % mkdir movie
[7] % mkdir port
[8] % mkdir seaside
[9] % mkdir USA
[10] % mkdir capital
```

Les commandes 4 à 10 créent des répertoires comme sous un système de fichier traditionnel. Un répertoire sous LFS doit être vu comme une *propriété* qu'un fichier peut avoir.

```
[11] % cp /pictures/los-angeles.jpg
     /lfs/art/movie/port/seaside/USA/
[12] % cp /pictures/washington.jpg
     /lfs/USA/capital/
[13] % cp /pictures/miami.jpg
     /lfs/port/seaside/USA/
[14] % cp /pictures/boston.jpg
     /lfs/art/music/port/seaside/USA/
[15] % cp /pictures/hamburg.jpg
     /lfs/port/
[16] % cp /pictures/san-diego.jpg
     /lfs/port/seaside/USA/
[17] % cp /pictures/new-york.jpg
     /lfs/art/music/port/seaside/USA/
```

Les commandes 11 à 17 copient sous LFS des images correspondant à des cartes de villes, et leur associent des propriétés leur correspondant. Par exemple, comme Boston est aux Etats-Unis, est un port, est près de la mer, et est une ville où l'art et particulièrement la musique est important, on lui associe les propriétés `USA`, `seaside`, `port`, `art` et `music`. Ces propriétés constituent la *description* du fichier `boston.jpg`.

	art	music	movie	port	seaside	USA	capital
Boston.jpeg	x	x		x	x	x	
LosAngeles.jpeg	x		x	x	x	x	
Miami.jpeg				x	x	x	
Hamburg.jpeg				x			
SanDiego.jpeg				x	x	x	
Washington.jpeg						x	x
NewYork.jpeg	x	x		x	x	x	

FIG. 2.5 – Un contexte

Contrairement aux systèmes de fichiers hiérarchiques, `USA` n'a pas à être un sous-répertoire de `art`, ni `seaside` un sous-répertoire de `music`. Le *slash* (`/`) doit être vu comme une *conjonction*; il est donc commutatif. Ainsi, `music/port` est équivalent à `port/music`. L'utilisateur peut associer de multiples propriétés à un fichier sans être obligé d'ordonner ces propriétés en répertoires et sous-répertoires, ce qui conduirait à une décomposition rigide et une navigation forcément limitée.

L'état de cette organisation se représente bien à l'aide d'une matrice *fichier* $\times$ *propriété*, formant ce qu'on appelle le *contexte* (voir Figure 2.5).

Chercher des fichiers

```
[18] % cd port/USA
[19] % ls
music/   movie/    art/
miami.jpg    san-diego.jpg
```

La commande 18 change le répertoire de travail courant (la variable `PWD` sous certains *shell*) pour `/lfs/port/USA`. Ce chemin correspond de manière interne à une *requête logique*, dans ce cas précis $port \wedge USA$. Créer un fichier dans ce répertoire aura pour effet de lui associer ces 2 propriétés. Le résultat de la commande `ls` a 2 parties. La première est composée

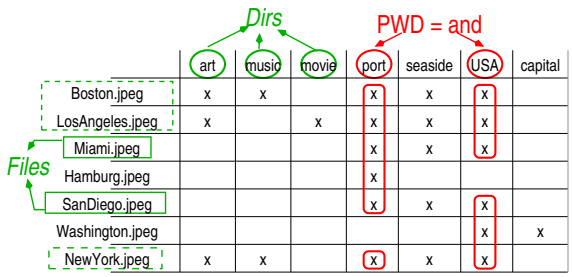

FIG. 2.6 – Calcul de `ls`

de sous-répertoires, `music/`, `movie/` et `art/`, correspondant à des propriétés qui *raffinent* la requête courante, sans pour autant la rendre vide (voir Figure 2.6). Par exemple, ne sont proposées ni la propriété `seaside`, parce que tous les fichiers ayant `port` et USA ont aussi cette propriété, ni la propriété `capital`, parce qu'aucun des fichiers ayant `port` et USA n'a cette propriété. LFS propose pour naviguer seulement des compléments de requêtes intéressants ; nous appelons ces sous-répertoires des *incréments*. La seconde partie du résultat de la commande `19` est composée de fichiers, `miami.jpg` et `san-diego.jpg`, dont la description *satisfait* la requête de ce répertoire, mais pas les requêtes correspondant aux sous-répertoires (voir Figure 2.6).

```
[20] % cd /lfs/capital|movie/!seaside
[21] % ls
washington.jpg
```

La commande `20` illustre les possibilités du langage de requête, combinant ici la *négation* (notée `!`), et la *disjonction* (notée `|`). Le *slash* correspond à la conjonction, ainsi le chemin `/lfs/capital|movie/!seaside` correspond logiquement à $(capital \vee movie) \wedge \neg seaside$. En fait, l'utilisateur peut aussi utiliser `&` à la place de `/` pour exprimer la conjonction [1]. Sous LFS *les chemins sont des formules logiques*, et après chaque requête, quelque soit sa complexité, LFS calculera les propriétés intéressantes permettant d'affiner cette requête. C'est ce qui permet de combiner navigation et interrogation.

[1] La différence étant juste que de `/a/b/c`, la commande `cd ..` ramène au chemin `/a/b`, alors que de `/a/b&c`, elle ramène à `/a`.

Dans l'exemple précédent il n'y a évidemment pas de compléments proposés car la requête a restreint les possibilités à un seul fichier.

Manipuler des fichiers

```
[22] % pwd
/lfs/capital|movie/!seaside
[23] % ls
washington.jpg
[24] % rm washington.jpg
```

Contrairement aux outils d'interrogation comme `find`, le résultat d'une requête logique sous LFS est de *première classe*, et forme un répertoire comme les autres. Ainsi, l'utilisateur a toujours la possibilité d'exécuter des commandes sur ce résultat. La commande `24` enlève le fichier `washington.jpg` de l'organisation, comme dans un système de fichier traditionnel. Elle efface une ligne dans la matrice *fichier* $\times$ *propriété*.

```
[25] % cd /lfs
[26] % mkdir hot
[27] % cd port/USA
[28] % ls
music/ movie/ art/
miami.jpg  sandiego.jpg
[29] % mv miami.jpg ../hot/
```

Comme les fichiers évoluent, leurs descriptions évoluent aussi. Ainsi, l'utilisateur aimerait mettre à jour la description d'un fichier. Dans un système de fichier traditionnel, le chemin du répertoire où réside le fichier est la description courante de ce fichier. Lorsque l'utilisateur veut modifier cette description, il a juste à changer ce fichier de place en utilisant la commande `mv`. Cela marche de la même manière sous LFS. La commande `26` crée la nouvelle propriété `hot`, et la commande `29` «ajuste» la description de `miami.jpg` en lui enlevant la propriété USA et en lui ajoutant la propriété `hot`. Il faut noter que les propriétés comme `seaside`, qui ne sont pas mentionnées dans la commande et qui pourtant font partie de la description du fichier sont gardées ; elles font toujours partie de la description du fichier.

Organiser des propriétés manuellement

```
[30] % cd /lfs
```

```
[31] % rmdir movie
[32] % rmdir music
[33] % rmdir art
```

Les commandes `31` à `33` enlèvent des propriétés de l'organisation. Ces commandes n'effacent pas les fichiers possédant ces propriétés ; elles effacent juste les propriétés et les associations entre les fichiers et ces propriétés. Ces commandes effacent des colonnes dans la matrice *fichier* × *propriété*.

```
[34] % cd /lfs
[35] % mkdir art
[36] % mkdir art/music
[37] % mkdir art/movie
```

Le résultat de la commande `19` montre du doigt quelques problèmes dans l'organisation des cartes de villes sous LFS. En effet, LFS retourne comme compléments de requête pour la navigation les 3 propriétés `art`, `music` et `movie`. Étant donné que la musique et le cinéma sont des formes d'art, il y a une sorte de redondance à présenter ces 3 propriétés, ce qui pollue le processus de navigation. Le principe de la navigation est bien évidemment de proposer des propriétés qui affinent la recherche, mais aussi de proposer les propriétés les plus générales, permettant à l'utilisateur de naviguer progressivement du *général* vers le *spécifique*. Ainsi, sous LFS l'utilisateur peut organiser les propriétés entre elles, indiquant lesquelles sont générales, lesquelles sont spécifiques, en utilisant la commande `mkdir`. Les commandes `36` et `37` permettent de déclarer que `movie` et `music` sont des *sous-propriétés* de la propriété `art`.

```
[38] % cd /lfs/port&USA
[39] % ls
sandiego.jpg boston.jpg
los-angeles.jpg new-york.jpg
[40] % mv boston.jpg new-york.jpg music/
[41] % mv los-angeles.jpg movie/
[42] % ls
art/
sandiego.jpg
[43] % cd art/
[44] % ls
movie/ music/
```

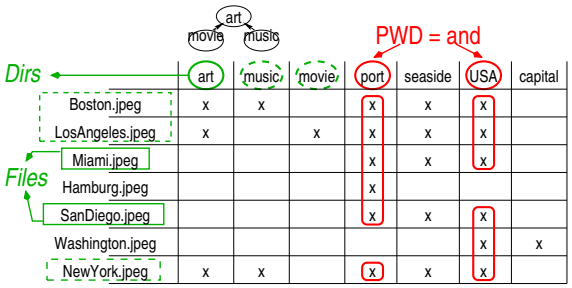

FIG. 2.7 – Calcul de `ls` avec la présence de sous-propriétés

La commande `42` montre l'effet qu'a l'*ordre* entre les propriétés sur la navigation. Maintenant, LFS propose les propriétés les plus générales affinant une requête (voir Figure 2.7). Désormais l'utilisateur est capable de créer une hiérarchie de propriétés, une *taxinomie*.

Cependant cet ordre entre les propriétés est utilisé par la navigation et n'est pas imposé à l'utilisateur ; LFS ne ré-introduit pas les problèmes des systèmes de fichier hiérarchiques, c'est à dire une navigation limitée et rigide. En effet, l'utilisateur peut formuler des requêtes logiques arbitrairement complexe, tout en gardant la possibilité de naviguer après ces requêtes. De plus, ces requêtes peuvent mentionner des sous-propriétés sans avoir à mentionner d'abord les propriétés plus générales. Par exemple, dans la commande `40`, l'utilisateur n'a pas besoin de mentionner la propriété `art`, lorsqu'il veut ajouter la propriété `music` à un fichier. Enfin, comme LFS permet à l'utilisateur d'associer plusieurs propriétés à un fichier, celui-ci n'a pas besoin d'ordonner différentes classifications entre elles comme dans les systèmes de fichier traditionnels, par exemple en classant des fichiers musicaux d'abord par genre, puis par artiste. Sous LFS, l'utilisateur peut garder ces différentes classifications clairement séparées, par exemple en ayant des propriétés et sous-propriétés concernant le genre d'une musique, d'autres propriétés et sous-propriétés concernant les artistes. Puis, après avoir associé différents types de propriétés aux fichiers, l'utilisateur peut chercher un fichier en utilisant différents points de vue, par exemple en cherchant des fichiers musicaux en naviguant d'abord dans la classification par artiste, puis dans la classifica-

tion par genre, ou bien l'opposé.

Le paradigme LFS combine ainsi le paradigme booléen et hiérarchique dans la même structure.

```
[45] % cd /lfs
[46] % cp /pictures/chicago.jpg
          /lfs/USA&music/
[47] % ls
art/ port/ USA/ seaside/ hot/
[48] % cd art
[49] % ls
music/ movie/
port/ seaside/
```

Il faut noter qu'exécuter les commandes `cd art/` puis `ls` n'a pas pour effet de lister uniquement les sous-propriétés de `art`, comme pourrait l'indiquer le résultat de la commande `44`. LFS liste toutes les propriétés les plus générales permettant d'affiner la recherche. Cela inclut souvent bien-sûr les sous-propriétés de `art` comme `music` et `movie`, mais aussi dans d'autres configurations comme celle de la commande `47` d'autres propriétés comme `port` et `seaside`. La Section 5.2.2 page 132 présente des variations pour la sémantique de `ls` permettant de contrôler plus précisément le type des propriétés complémentaires à afficher après une requête.

```
[50] % cd /lfs
[51] % mkdir geography
[52] % mv seaside/    geography/
[53] % mv port/ USA/ capital/
          geography/
```

Tout comme peuvent évoluer les descriptions de fichier, l'ordre entre les propriétés peut évoluer aussi. La commande `51` crée la nouvelle propriété `geography` et la commande `52` «ajuste» l'ordre entre les propriétés `seaside` et `geography`, faisant de la première une sous-propriété de la seconde. La commande `53` fait la même chose pour les propriétés `port`, `USA` et `capital`.

Organiser des propriétés automatiquement

```
[54] % cd /lfs
[55] % mkdir type:
[56] % cp /pictures/misc.jpg
          /lfs/type:picture/
[57] % cp /programs/foo.c
          /lfs/type:program/
[58] %  ls
type:/ art/ geography/ hot/
[59] % ls type:/
type:program/ type:picture/
```

L'utilisateur peut organiser les propriétés en les ordonnant manuellement, afin de bénéficier d'une meilleur navigation. LFS peut aussi organiser des propriétés automatiquement si elles ont une forme particulière. Lorsqu'une propriété a la forme d'un *attribut valué*, avec la syntaxe `attribut:valeur`, et que la propriété correspondante `attribut:` a déjà été créée, alors l'utilisateur peut mentionner cet attribut valué sans avoir eu à le créer d'abord comme pour les autres propriétés. On peut le voir par exemple dans les commandes `56` et `57`, avec les attributs valués `type:picture` et `type:program`, et la propriété `type:` crée avec la commande `55`. De plus, ces attributs valués sont automatiquement faits sous-propriétés de l'attribut correspondant comme le montre le résultat de la commande `59`. En effet, LFS propose d'abord la propriété général `type:`, puis les propriétés plus spécifiques `type:picture` et `type:program`.

```
[60] % mkdir plugins
[61] % mkdir plugins/logic:
[62] % mkdir year:
[63] % cp /solver/int_logic
          /lfs/logic:year/
[64] % cp /pictures/vacation-corsica.jpg
          /lfs/year:2000/
[65] % cp /pictures/vacation-england.jpg
          /lfs/year:2001/
[66] % cp /pictures/vacation-france.jpg
          /lfs/year:2002/
[67] % cd year:>2000/
[68] % ls
year:2001/ year:2002/
[69] % ls /lfs/year:
year:2000/ year:>2000/
```

En fait, ce système de gestion des attributs valués, très naïf, peut être étendu par l'utilisateur via un système de

plug-ins. En effet, un utilisateur aimerait entre la propriété générale, représentée par l'attribut seul, et les propriétés spécifiques, représentées par les attributs valués, des propriétés intermédiaires. Cela permettrait de classer encore un peu plus ces attributs valués, afin de bénéficier d'une meilleure navigation. Par exemple, pour des attributs valués par des chaînes de caractères, un utilisateur aimerait des catégories intermédiaires représentant la première lettre de la chaîne de caractère, permettant de classer alphabétiquement ces attributs valués. Les commandes `60` à `63` montrent comment procéder pour avoir des attributs valués par des entiers représentant des années, et pour avoir des catégories intermédiaires représentant des intervalles d'années, le tout ordonné automatiquement. La commande `61` crée la propriété `logic:` qui a un sens très spécial pour LFS [2]. Pour indiquer quel programme va gérer l'ordre entre les différents attributs valués, l'utilisateur doit copier le programme d'un *moteur de déduction logique* sous LFS.

Ce programme, qui est un fichier traditionnel, doit posséder la propriété spéciale `logic:`, ainsi que la propriété correspondant à l'attribut qu'il gérera, rassemblées via un attribut valué comme indiqué par la commande `63`. Ce moteur de déduction logique permet de manipuler des attributs valués classiques, comme indiqué dans les commandes `64` à `66`, mais aussi d'autres propriétés mentionnant des intervalles, comme indiqué dans la commande `67`. Les commandes `68` et `69` montrent comment ces nouvelles propriétés, sorte de *propriétés-formules* ont été ordonnées, et comment elles sont utilisées par la navigation pour aller du général vers le spécifique. En effet, la propriété `year:>2000` est plus générale que les propriétés `year:2001` et `year:2002`.

La Figure 2.8 illustre l'ordre entre toutes ces propriétés. Il faut noter que cette taxinomie de propriétés n'est pas la structure de navigation, auquel cas nous réintroduirions les limitations des systèmes hiérarchiques. En particulier, `movie` a beau ne pas être une sous-propriété de `port`, il se peut que `movie` soit proposé comme sous-répertoire du répertoire `/lfs/port` car certain fichiers ayant la propriété `port` auraient aussi la propriété `movie`.

[2] En fait, une option de `mkfs.lfs` permet de créer par défaut les propriétés `logic:` et `plugins` ce qui évite à l'utilisateur d'avoir à taper les commandes `60` et `61`.

```
[70] % /solver/int_logic '2001' '>2000'
yes
[71] % /solver/int_logic '>2000' '2001'
no
```

Les commandes `70` et `71` montrent l'interface d'un moteur de déduction logique. Le programme prend 2 paramètres correspondant à 2 propriétés, et retourne `yes` lorsque la première propriété est plus spécifique ou égale à la deuxième, `no` dans le cas contraire. Ces moteurs de déduction logique sont utilisés de manière interne par LFS pour le mécanisme de navigation ; ils ont aussi une influence sur le mécanisme d'interrogation. En effet, l'utilisateur n'est plus limité à des requêtes booléennes, mais peut formuler des requêtes arbitraires, pourvu qu'un moteur de déduction logique pour cette requête ait été défini.

Ces propriétés exotiques, sorte de *propriétés-formules* comme celles représentant des intervalles, sont traitées de la même façon que les autres propriétés, comme indiqué par le résultat de la commande `69`. Elles peuvent notamment être utilisées dans la description d'un fichier. Cela permet de rendre LFS *générique* vis-à-vis de la logique employée. De plus, le fait de voir ces formules permet à l'utilisateur de deviner la syntaxe admise par le moteur de déduction logique. Ainsi, sans connaissance a priori, un utilisateur peut formuler des requêtes complexes par imitation en suivant l'exemple. Un des problèmes de l'interrogation est qu'il requiert de l'érudition de la part de l'utilisateur. LFS résoud ce problème puisqu'il combine interrogation et navigation.

Organiser des fichiers automatiquement

```
[72] % cd /lfs
[73] % mkdir size:
[74] % mkdir name:
[75] % mkdir ext:
[76] % mkdir mypictures
[77] % cp /pictures/big.jpg mypictures/
[78] % cp /pictures/small.jpg mypictures/
[79] % ls ext:jpg/
size:1Ko/ size:23Ko/
name:big/ name:small/
```

Être capable de décrire un fichier avec des attributs valués est appréciable, mais cela nécessite toujours une

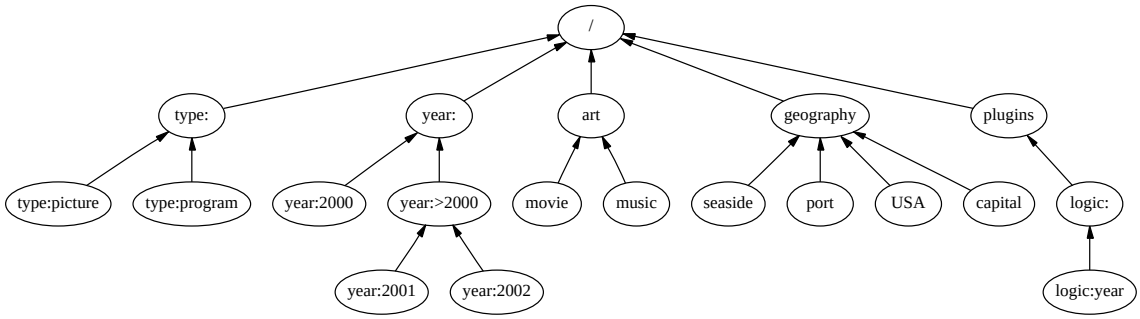

FIG. 2.8 – Une taxinomie de propriétés

intervention manuelle pour remplir les valeurs. Cela est gênant. Par exemple, considérons un attribut valué représentant la taille d'un fichier, sa valeur peut changer dès qu'une application édite le fichier ; on ne peut pas envisager de forcer l'utilisateur à la maintenir manuellement. Ce genre de propriétés peut être inféré à partir du contenu des fichiers, et LFS fournit des moyens pour les extraire automatiquement. Les propriétés systèmes d'un fichier, comme sa taille (`size`:x), son nom (`name`:x), son extension (`ext`:x, le suffixe du nom), ou sa date de dernière modification (`mtime`:x)), sont automatiquement extraites par LFS et rajoutées à la description du fichier [3]. La commande `79` montre l'effet de ces extractions lorsqu'un utilisateur crée de nouveaux fichiers.

```
[80] % cd /lfs
[81] % mkdir plugins/transducer:
[82] % cp /transducer/mp3_transducer
         /lfs/transducer:mp3/
[83] % mkdir mymusic
[84] % cd mymusic
[85] % mkdir  artist:  genre:
[86] % cp /musics/staying_alive.mp3 .
[87] % cp /musics/ete_indien.mp3 .
[88] % ls
genre:Disco/ genre:Pop/
artist:BeeGees/ artist:JoeDassin/
name:ete_indien/ name:staying_alive/
size:3Mo100Ko/ size:4Mo200Ko/
```

[3] Comme précédemment avec la propriété spéciale `logic`:, l'option de `mkfs.lfs` permet d'éviter à l'utilisateur d'avoir à taper les commandes `73` à `75` en créant en standard ces propriétés.

Comme pour les moteurs de déduction logique, qui permettent à l'utilisateur d'étendre les fonctionnalités logiques offertes par le *noyau* de LFS, l'utilisateur peut aussi étendre les fonctionnalités d'extraction de LFS en fournissant des programmes appelés *transducteurs*. Les commandes `81` et `82` montrent comment procéder pour extraire automatiquement des propriétés musicales à partir de fichiers musicaux de type MP3. La commande `81` crée la propriété `transducer`:, qui comme pour la propriété `logic`: a un sens spécial pour LFS. Pour indiquer quel programme sera chargé de l'extraction automatique de propriétés d'un type de fichier, l'utilisateur doit copier le programme d'un transducteur sous LFS. Ce programme, qui est un fichier traditionnel, doit posséder la propriété spéciale `transducer`:, ainsi que la propriété représentant l'extension (le suffixe du nom) du type des fichiers qu'il gérera, dans le cas présent la propriété `mp3`, comme indiqué par la commande `82`. La commande `88` montre l'effet de cette nouvelle extraction lorsque l'utilisateur copie sous LFS des fichiers MP3.

```
[89] % cat staying_alive.mp3 |
       /transducer/mp3_transducer
genre:Disco/artist:BeeGees
```

La commande `89` montre l'interface d'un transducteur. Le programme prend sur son entrée standard le contenu d'un fichier et retourne sur sa sortie standard une liste de chaînes de caractères séparées par un `/` ; une chaîne par propriété extraite du fichier. Ces transducteurs sont ré-appliqués aux fichiers à chaque fois que le contenu du fichier change, maintenant ainsi la cohérence entre la description et le contenu d'un fichier.

La profusion de propriétés, spécialement d'attributs valués, fait que le résultat d'une requête peut parfois contenir de nombreux sous-répertoires, comme l'indique le résultat de la commande `88`. La Section 5.2.2 page 132 présente de légères variations pour la sémantique de `ls` permettant de contrôler plus précisément le type des propriétés complémentaires à afficher après une requête. Cela permet par exemple de n'afficher à la racine que les attributs `genre:`, `artist:`, `name:` et `size:`. L'utilisateur peut ensuite «développer une branche», par exemple avec la commande `cd genre:/`, tout en gardant la possibilité plus tard d'ouvrir d'autres branches. Une autre variation permet d'afficher dans un répertoire tous les fichiers satisfaisant la requête. C'est parfois plus pratique, lorsque le nombre de fichier est petit, que le mode par défaut qui n'affiche pas les fichiers satisfaisant les requêtes des sous-répertoires.

Le scénario suivant illustre ces nouvelles possibilités en supposant l'existence de beaucoup plus de fichiers sous LFS et d'une organisation des propriétés légèrement différente :

```
[1] % cd /lfs/; ls
system/ plugins/
geography/ mymusic/ mypictures/
[2] % cd system/; ls
size:/ ext:/ name:/ type:/
plugins/
geography/ mymusic/ mypictures/
[3] % cd ext:/; ls
ext:jpg/ ext:mp3/ ext:c/ ext:txt/ ...
size:/ name:/ type:/
plugins/
geography/ mymusic/ mypictures/
[4] % cd ext:mp3/; ls
size:/ name:/ mymusic/
[5] % cd mymusic/; ls
genre:/ artist:/ album:/ note:/ year:/
size:/ name:/
[6] % cd genre:/artist:/; ls
genre:Disco/ genre:Rock/ genre:Pop/ ...
artist:Beatles/ artist:Bee Gees/ ...
album:/ note:/ year:/   size:/ name:/
[7] % cd genre:Disco&year:<1970/; ls
artist:Bee Gees/
artist:DonnaSummer/ artist:KoolGang/
year:1967/ year:1968/
album:/ note:/  size:/ name:/
staying_alive.mp3
love-to-love-you-baby.mp3
beach.mp3
[8] % mv staying_alive.mp3
         note:excellent/
[9] % cd !note:excellent/; ls
note:moyen/ note:mauvais/
artist:DonnaSummer/ artist:KoolGang/
year:1968/ album:/  size:/ name:/
love-to-love-you-baby.mp3
beach.mp3
```

2.3.2 Sur le contenu des fichiers

Nous venons de voir comment LFS permet à l'utilisateur de mieux organiser, chercher et manipuler ses fichiers. Nous allons voir maintenant comment LFS procède pour offrir ces mêmes services pour la gestion du contenu des fichiers.

Organiser le contenu des fichiers

```
[90] % cd /lfs
[91] % mkdir myprograms
[92] % cp /programs/foo.c myprograms/
[93] % cat -n myprograms/foo.c
     1  int f(int x) {
     2  int y;
     3  assert(x > 1);
     4  y = x;
     5  fprintf(stderr, "x = %d", x);
     6  return y * 2
     7  }
     8  int f2(int z) {
     9  return z * 4
    10  }
[94] % cat myprograms/foo.c |
      /transducer/c_adv_transducer |
      head -n 2
function:f/var:x
function:f/var:y
```

En plus des transducteurs qui indexent des fichiers entiers, LFS peut aussi utiliser des transducteurs indexant

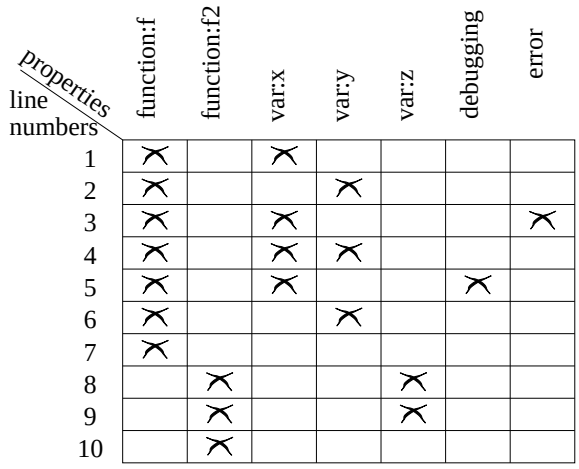

FIG. 2.9 – Un contexte de fichier

des *parties* de fichier. Dans le prototype actuel de LFS, les parties de fichiers sont des lignes, mais les principes resteraient les mêmes si ces parties étaient des paragraphes, ou bien juste des mots. La commande `93` montre le contenu d'un fichier C. La commande `94` montre l'interface de ces nouveaux types de transducteurs que nous appellerons *transducteurs avancés*. Un transducteur avancé prend sur son entrée standard le contenu d'un fichier, mais retourne cette fois ligne par ligne une liste de chaînes de caractères (séparées encore par des `/`); une chaîne par propriété extraite de cette ligne. Dans cet exemple, les propriétés concernées sont «la définition de f occupe cette ligne» (`function`:f), «la variable x est mentionnée dans cette ligne» (`var`:x), «cette ligne a un aspect déboggage» (`debugging`) et «cette ligne a un aspect gestion d'erreurs» (`error`).

```
[95] % mkdir function:
[96] % mkdir var:
[97] % mkdir debugging
[98] % mkdir error
[99] % mkdir plugins/adv_transducer:
[100] % cp /transducer/c_adv_transducer
          /lfs/adv_transducer:c/
[101] % cd myprograms
[102] % ls
foo.c
[103] % cd parts
```

Les commandes `95` à `98` créent les propriétés que l'utilisateur souhaite manipuler. Les commandes `99` et `100` procèdent de la même manière que pour les transducteurs normaux. Elles permettent d'indiquer quel transducteur avancé sera responsable de la gestion des fichiers C. La commande `103`, et plus particulièrement la propriété `parts` a un sens spécial pour LFS. Elle permet d'indiquer que les objets que l'utilisateur souhaite désormais manipuler ne sont plus des fichiers, mais des parties de fichiers. Cette commande permet d'indexer, via les transducteurs avancés appropriés définies auparavant, l'ensemble des fichiers qui satisfaisaient la requête précédente, dans notre exemple seulement le fichier `foo.c`. Comme pour les fichiers, l'état de cette organisation se représente bien à l'aide d'une matrice, cette fois une matrice *ligne* $\times$ *propriété*, formant ce qu'on appelle le *contexte du fichier* (voir Figure 2.9). Il faut noter que l'indexation n'est pas locale à une partie. Par exemple, la ligne 7 a la propriété `function`:`f` car une déclaration d'une fonction `f` a été trouvée 6 lignes plus haute. Un changement à la ligne 1 pourrait affecter les propriétés des lignes 2 à 7. La commande `103` permettra à l'utilisateur, comme nous allons le voir, non plus de naviguer parmi les fichiers, mais *dans* les fichiers. Nous appelons *fichier original* le fichier qui sert de base au travail de l'utilisateur (ici le fichier `foo.c`).

```
[104] % ls
debugging/ error/
function:f/ function:f2/
var:/
foo.c
[105] % wc --line foo.c
10 foo.c
```

La commande `104` a 2 effets. Le premier est de créer un fichier `foo.c` qui correspond à une *vue* du fichier original, contenant les parties de ce fichier satisfaisant la requête courante. Comme aucune propriété n'a encore été sélectionnée, cette vue a exactement le même contenu que le fichier original, comme le montre la commande `105`. Le deuxième effet est de présenter des sous-répertoires à l'utilisateur (comme `function`:`f`, `debugging`), correspondant à des propriétés qui raffinent la requête courante, sans pour autant la rendre vide, comme dans la section précédente avec les fichiers.

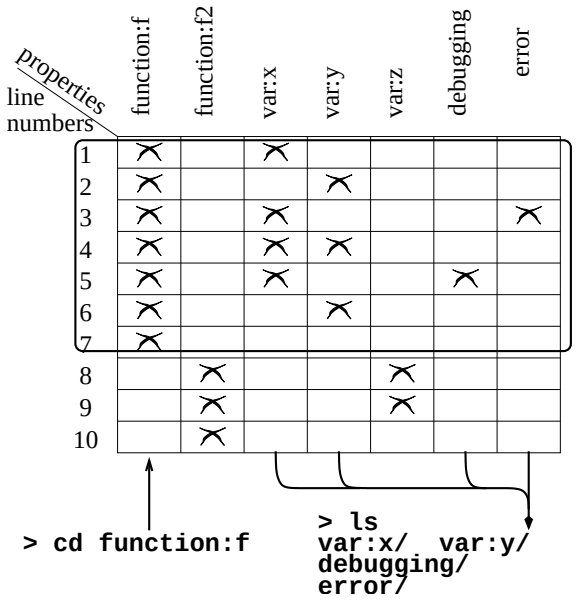

FIG. 2.10 – Navigation dans un fichier

Rechercher et sélectionner le contenu des fichiers

```
[106] % cd function:f/var:/; ls
var:x/ var:y/
debugging/ error/
foo.c
[107] % wc --line foo.c
7 foo.c
```

La commande `106` raffine la vue, sélectionnant les parties du fichier ayant la propriété `function`:`f` et parlant de variables. Là encore, les commandes `106` et `107` montrent comment une nouvelle vue vient d'être créée, contenant cette fois moins de lignes que le fichier original (7 au lieu de 10). Comme pour les fichiers, la commande `106` montre comment les propriétés proposées pour naviguer dépendent de la requête courante. En effet, sous `function`:`f/var`:`/`, la propriété `var`:`z` n'est pas proposée car aucune ligne de la vue ne parle de la variable `z`. De plus, la propriété `function`:`f` n'est pas proposée non plus car la vue correspondante serait la même vue que la vue courante, et donc ne la raffinerait pas (voir Figure 2.10).

```
[108] % cd !(debugging|error)
```

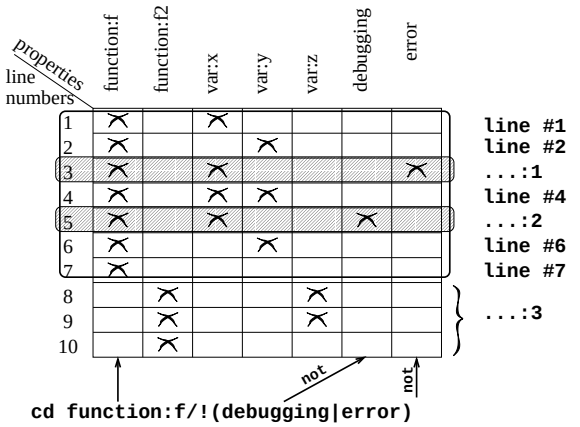

FIG. 2.11 – Création d'une *vue*

```
[109] % ls
var:x/ var:y/
foo.c
[110] % cat foo.c
int f(int x) {
int y;
.............:1
y = x;
.............:2
return y * 2
.............:3
```

La commande `109` montre que les propriétés qui raffinent ont été réduites à `var`:`x` et `var`:`y` (voir Figure 2.11). La commande `110` montre le contenu de la vue correspondante. Les lignes ne satisfaisant pas la requête courante ont été filtrées et remplacées par des *marques d'absence*, comme `...........`:`1`. Ces marques permettront de *propager* la modification d'une vue sur le fichier original.

Grâce à LFS, l'utilisateur peut manipuler son fichier selon différents points de vue. En fonction de la tâche qu'il a à accomplir, l'utilisateur peut sélectionner tel ou tel *aspect* jugé pertinent pour la réalisation de cette tâche, ou au contraire filtrer tel ou tel aspect car il polluerait son travail. Dans cet exemple, la possibilité de filtrer les commandes de déboggage permet à l'utilisateur de se concentrer plus facilement sur l'aspect fonctionnalité. De la même manière, l'utilisateur, s'il voulait juste se faire

une idée des fonctions, pourrait sélectionner via une vue adéquate uniquement les lignes qui incluent le prototype des fonctions ainsi que leurs spécifications, rendant ainsi plus facile la compréhension générale du programme.

Manipuler le contenu des fichiers

```
[111] % cat foo.c | sed -e s/y/z > foo.c
[112] % ls
var:x/ var:z/
foo.c
```

Contrairement aux outils d'interrogation comme `grep`, le résultat d'une requête logique sous LFS est de *première classe*, une vue est un fichier comme les autres. La commande `111` montre que les vues peuvent être *mises à jour*, et peuvent être modifiées par n'importe quel outil (ici `sed`). L'effet de cette commande est de remplacer toutes les occurrences de `y` par `z`. La commande `112` montre comment la modification d'une vue affecte le processus de navigation (comparer avec le résultat de la commande `109`).

```
[113] % pwd
/lfs/myprograms/parts/function:f/
   var:/!(debugging|error)/
[114] % cd /lfs/myprograms/parts/
[115] % cat foo.c
int f(int x) {
int z;
assert(x > 1);
z = x;
fprintf(stderr, "x = %d", x);
return z * 2
}
int f2(int z) {
return z * 4
}
```

Enfin, la commande `115` montre comment la modification d'une vue affecte aussi les autres vues, ainsi que le fichier original, en propageant la modification.

LFS aide ainsi l'utilisateur à organiser, chercher ou manipuler ses données, comme par exemple des fichiers musicaux ou des images comme on l'a vu précédemment. LFS aide aussi le programmeur à gérer les sources de ses programmes. Les mêmes principes s'appliquent pour la gestion de vidéos, d'e-mails, ou bien encore pour la gestion de sources LaTeX. Le Chapitre 6 présente d'autres types de données que ceux présentés dans ce chapitre et donne ainsi une idée sur les nombreuses applications possibles de LFS. Dans ces applications, LFS aidera ainsi entre autre l'écrivain, le manager ou l'administrateur système.

2.4 Organiser

Nous allons maintenant préciser dans cette section, ainsi que dans les deux suivantes, les principes mis en jeu dans l'exemple précédent. Cette section s'attachera à préciser les principes d'*organisation*, les sections suivantes les principes de *recherche* et de *manipulation*.

Comme le titre de cette thèse l'indique, LFS parle de logique et de fichiers :

- Une *logique* [Cha96, p475-589][Kel96] est définie par un langage F, c'est à dire la syntaxe de ses formules, et une relation de *déduction logique* notée habituellement $\models$. $f1 \models f2$ est vrai si à chaque fois que $f1$ est vrai, $f2$ l'est aussi. Par exemple, $a \wedge b \models a$ est vrai dans la logique propositionnelle. En effet, quelque soit les *valuations* de a et b rendant la formule $a \wedge b$ vraie, ces valuations rendront forcément aussi vraie la formule a.
- Un *fichier* est constitué de deux parties : une *description* et un *contenu*. Sous un système de fichier traditionnel, la description d'un fichier est le chemin du répertoire où réside ce fichier, ainsi que le nom de ce fichier. Sous LFS, la description est une formule.

LFS est un système combinant logique et fichiers. Le langage F fournit ce dont nous avons besoin pour décrire de manière expressive les fichiers ; la relation de déduction $\models$ ce dont nous avons besoin pour chercher les fichiers. En effet, étant donné la description d'un fichier représentée par une formule d, et une requête représentée par une formule q, la logique peut déterminer si un fichier satisfait une requête en vérifiant que $d \models q$. Par exemple, un fichier ayant pour description la formule $seaside \wedge USA \wedge port$ satisfera la requête `cd USA/` car $seaside \wedge USA \wedge port \models USA$. Les répertoires jouent sous LFS le rôle de formules logiques, et ils sont utili-

sés pour décrire et interroger les fichiers. Ils sont aussi utilisés pour naviguer. En effet, le principe de la navigation est d'aller du général vers le spécifique et là encore la logique peut déterminer si une formule f est plus générale qu'une formule g en vérifiant que $g \models f$. Par exemple la formule «> 100» est plus générale que la formule «> 200» (elle couvre plus de cas), qui elle même est plus générale que la formule «250» (i.e. «$= 250$»), car avec une logique d'entier $250 \models > 200 \models > 100$. La logique et les formules permettent ainsi de décrire, interroger et naviguer.

Ainsi, une organisation LFS, que l'on appelle un *contexte*, est constituée de :

1. Un ensemble de propriétés $\mathcal{P}$, où chaque propriété est exprimée dans un langage F.
2. Un ensemble d'objets $\mathcal{O}$, où chaque objet a une description d, constituée d'un ensemble de propriétés venant de $\mathcal{P}$, et un contenu c.
3. Une logique $\mathcal{L}$ représentant $\models$, et qui gouverne l'*ordre* entre les propriétés de $\mathcal{P}$.

Nous parlons d'objets et non de fichiers car LFS fournit des services à la fois pour gérer des fichiers et leurs contenus. Ainsi, un objet peut soit représenter un *fichier* soit une *partie de fichier*. En effet, le contenu d'un fichier peut être divisé en différentes parties, où chaque partie peut être vue comme un objet, avec aussi une description et un contenu. La commande `cd parts` permet de passer du monde des fichiers vers le monde des parties de fichiers, indiquant ainsi à LFS quel genre d'objet l'utilisateur veut manipuler.

LFS est fait d'un *noyau* composé d'une logique rudimentaire, qui peut après être étendue par l'utilisateur. La logique de ce noyau a la même syntaxe que la logique des propositions (qui peut être vue aussi comme une logique booléenne ou une logique d'ensemble du fait que sous LFS les formules dénotent des ensembles d'objets). Ainsi, une formule peut être soit une simple *propriété atomique* (a), soit une conjonction de formules ($f \wedge g$), soit une disjonction de formules ($f \vee g$), soit la négation d'une formule ($\neg f$). Ces formules sont notées `a`, `f&g`, `f|g`, et `!f` dans la syntaxe concrète. Le *slash* (`/`) peut aussi représenter la conjonction. Les règles de déduction pour ces formules sont composées de règles habituellement utilisées pour la logique des propositions par exemple $a \wedge b \models a$, $a \models a \vee b$, $a \models \neg b$.

La description d'un fichier doit être une conjonction de propriétés. L'utilisateur ne peut utiliser la négation et la disjonction que pour la recherche. Ainsi, si un utilisateur crée un fichier dans un chemin mentionnant un élément avec une disjonction ou une négation, ces éléments ne sont pas ajoutés à la description d'un fichier. Par exemple dans `touch a&b/c|d/!f/o`, seul les propriétés `a` et `b` feront parties de la description du fichier `o`. La raison étant que pour la négation, pour qu'un fichier satisfasse la formule `!f` il suffit simplement de ne pas lui ajouter la propriété `f`, et pour la disjonction, un fichier s'il avait la propriété `a|b` ne pourrait être retrouvé ni par la requête `a`, ni par la requête `b` car $a \vee b \models a$ et $a \vee b \models b$ sont faux.

Concrètement :

1. `mkdir a`; `mkdir b`; `mkdir c` ajoute les propriétés a, b et c dans $\mathcal{P}$.
2. `touch a/b/c/o` ajoute un fichier o dans $\mathcal{O}$ avec une description $d = \{a, b, c\}$ ($= a \wedge b \wedge c$), et un contenu vide.
3. `cd b/c/` exprime la requête $q = b \wedge c$, le fichier précédent satisfera cette requête car $a \wedge b \wedge c \models b \wedge c$.

Comme dit précédemment, la logique de ce noyau peut être étendue par l'utilisateur, le principe étant de permettre à l'utilisateur de fournir sa propre logique. Cela permet d'avoir une logique $\models$ plus riche, et donc aussi un pouvoir d'expression pour les requêtes et descriptions plus grand. Cette liberté offrant la possibilité d'avoir n'importe quel type de formule dans la description des fichiers, ainsi que dans les requêtes, rend ainsi LFS *générique* vis-a-vis de la logique.

En fait, et les règles logiques $\models$, et les moyens d'assigner des propriétés à un fichier, peuvent être étendus. Tous deux peuvent être gérés *manuellement* par l'utilisateur, ou *automatiquement* via des programmes comme décrit dans les deux sections suivantes.

2.4.1 Les logiques

LFS fournit deux moyens pour étendre la logique : soit à l'aide d'*axiomes*, soit à l'aide de nouvelles *règles d'inférence* permettant de gérer d'autres domaines que celui des atomes.

Axiomes

On peut ordonner certaines propriétés entre elles pour former des *taxinomies*, permettant à l'utilisateur de représenter les connaissances qu'il a d'un domaine. Par exemple, dans le domaine de l'informatique, la propriété *Unix* peut être considérée comme plus spécifique que la propriété *OperatingSystems*, car tout document parlant d'Unix est un document qui parle d'un système d'exploitation. Cela veut dire que si un objet o satisfait la requête *Unix*, en terme logique si $d(o) \models$ *Unix*, alors il devrait aussi satisfaire la requête *OperatingSystems*, en terme logique $d(o) \models$ *OperatingSystems*. Ce besoin de transitivité est résolu en considérant les relations taxinomiques comme des axiomes, par exemple *Unix* $\models$ *OperatingSystems*. Ainsi, lorsque l'utilisateur fera `cd OperatingSystems/`, tous les fichiers ayant la propriété *Unix* satisferont aussi cette requête. De plus, l'utilisateur pourra maintenant naviguer de la catégorie générale *OperatingSystems*, vers la catégorie spécifique *Unix*.

Concrètement, les axiomes sont créés par l'usage combiné des commandes `mkdir` et `cd`. Ainsi, `mkdir a`; `mkdir b`; `cd a/b/`; `mkdir ab` ajoute les axiomes $ab \models a$ et $ab \models b$, faisant de ab une *sous-propriété* de a et de b.

Règles d'inférence

Il existe des domaines où l'utilisateur ne devrait pas avoir besoin de fournir des axiomes pour ordonner des propriétés. Par exemple dans le domaine des entiers, il est évident que la propriété «être entre 1 et 10» est plus générale que la propriété «être entre 2 et 5». Cela requiert d'incorporer un démonstrateur automatique de théorème : un *moteur de déduction logique* constitué de règles d'inférence pour représenter la relation $\models$. Même si certaines logiques sont indécidables, comme la logique des prédicats, il y en a beaucoup d'autres, très utiles, dont l'usage peut s'avérer viable en terme d'efficacité. Par exemple, une logique de type pour les composants d'un programme, une logique d'intervalles pour exprimer des dates.

Sous LFS les propriétés peuvent être des *attributs valués*, et les valeurs peuvent représenter des formules arbitraires d'un domaine géré automatiquement par un moteur de déduction logique dédié. L'utilisateur peut fournir différents moteurs de déduction, un pour chaque domaine de valeur. Pour rendre le système extensible, l'utilisateur peut fournir ces moteurs de déduction via un système de *plug-ins*, comme décrit Section 2.3.1 page 72. Un moteur de déduction représente $\models$, son interface est donc de prendre deux formules f et g, et de répondre à la question $f \models g?$.

2.4.2 Les fichiers

La description d'un fichier peut être divisée en deux parties : une partie subjective, formant ce qu'on appelle les *propriétés extrinsèques*, et une partie objective, formant ce qu'on appelle les *propriétés intrinsèques*.

Propriétés extrinsèques

Ces propriétés permettent à l'utilisateur de donner sa propre opinion sur un fichier, par exemple la qualité subjective d'un fichier musical avec la propriété `excellent`. De telles propriétés ne peuvent pas être inférées à partir du contenu du fichier. Pour assigner ces propriétés à un fichier, l'utilisateur peut sélectionner un ensemble de propriétés en construisant un chemin qui les contient toutes, et créer un fichier à cet endroit. L'utilisateur peut ajouter ou enlever des propriétés à un fichier en utilisant la commande `mv`.

Propriétés intrinsèques

Beaucoup de propriétés intéressantes peuvent être inférées à partir du fichier lui même, par exemple la taille d'un fichier. Ces propriétés peuvent être assignées à un fichier en écrivant des programmes spéciaux appelés *transducteurs*, qui seront en charge d'extraire automatiquement ces propriétés du fichier. Pour rendre le système extensible, l'utilisateur peut fournir ses propres transducteurs via un mécanisme de *plug-ins*, comme décrit Section 2.3.1 page 73. L'interface d'un transducteur est de prendre en paramètre le contenu c d'un fichier, et de retourner un ensemble de propriétés qui seront ajoutées dans la description d d'un fichier. Pour associer des propriétés intrinsèques à un fichier, l'utilisateur a juste à copier ce fichier sous LFS ; il sera automatiquement indexé

par les transducteurs appropriés. Il existe aussi des transducteurs dit avancés qui sont plus précis et qui assignent des propriétés à chaque partie de fichier, et non plus au fichier entier. Ils seront décrits plus loin.

Contrairement aux propriétés extrinsèques, l'utilisateur ne peut pas utiliser la commande `mv` pour modifier ces propriétés, sinon il pourrait briser la cohérence entre la description d et le contenu c. Ainsi, le seul moyen pour modifier ces propriétés est de modifier le contenu du fichier. Ces propriétés doivent donc être recalculées chaque fois que le fichier est modifié, d'où l'intérêt de diviser la description d d'un fichier en deux parties (une partie subjective, une partie objective) afin que le recalcul de la seconde partie ne perturbe pas la première.

2.5 Rechercher

La relation de déduction $\models$ peut être vue comme une relation de déduction ($d \models q?$) ou comme une relation d'*ordre* ($f \models g?$). Ces deux points de vue peuvent être associées aux deux modes de recherche offerts par LFS : l'interrogation et la navigation. La relation de déduction fournit la base pour permettre à LFS de combiner étroitement ces deux paradigmes de recherche.

Nous parlons de $\models$ comme une relation unique. Cependant, comme on l'a vu précédemment, différents mécanismes contribuent à sa définition : la logique du noyau, les axiomes, et les moteurs de déduction logique (qui définissent chacun localement leurs propres $\models$). Cependant, ces différents mécanismes peuvent se combiner pour d'un point de vue formel ne former qu'une seule relation de déduction $\models$ régissant globalement l'ensemble des propriétés.

2.5.1 En utilisant l'interrogation

Une notion clef de LFS est l'*extension* d'une formule logique f. Elle se définit par l'ensemble des objets dont la description satisfait la formule :

$\mathit{ext}(f) = \{o \in \mathcal{O} \mid d(o) \models f\}$

Concrètement l'utilisateur peut formuler une requête par le biais de la commande `cd` f, délimitant ainsi un ensemble de fichiers (ou de parties de fichier), définis précisément par l'extension de cette formule. Cette extension est utilisée ensuite par le processus de navigation pour proposer les sous-répertoires et les fichiers à lister après la commande `ls`, comme nous le verrons dans la section suivante.

Ainsi, dans l'exemple de la Section 2.3.1 page 70, avec les axiomes $movie \models art$ et $music \models art$, et la description $d(boston.jpg) = music \wedge port \wedge seaside \wedge USA$, on peut déduire que $boston.jpg \in ext(art)$ car :

$d(boston.jpg) = music \wedge port \wedge seaside \wedge USA$
$\models music$ (car $a \wedge b \models a$ avec la logique du noyau)
$\models art$ (car $music \models art$ est un axiome du système)

On peut aussi noter que
$\mathit{ext}(f \wedge g) = \mathit{ext}(f) \cap \mathit{ext}(g)$,
$\mathit{ext}(f \vee g) = \mathit{ext}(f) \cup \mathit{ext}(g)$,
$\mathit{ext}(\neg f) = \mathit{complement}(\mathit{ext}(f))$.

La formule correspondant à la racine du système du fichier est la formule *vrai*, car de la racine tous les fichiers doivent être accessibles, et donc tous les fichiers doivent satisfaire cette requête, et en effet d'après la logique quelque soit d, la formule $d \models$ *vrai* est toujours vraie. Nous verrons Section 6.2 page 147 que la formule *faux* a aussi son avatar sous LFS.

Lorsque les objets manipulés sont des parties de fichiers l'extension sert aussi à définir le contenu des vues (voir Section 2.6.2 page 84).

2.5.2 En utilisant la navigation

L'extension d'une requête pouvant être très large, typiquement à la racine du système de fichier, le besoin de navigation se fait très vite sentir. Sous LFS, les sous-répertoires d'un répertoire sont déterminés par les propriétés les plus générales dont l'extension intersecte strictement l'extension du répertoire. Cela donne une base logique solide pour considérer la navigation comme un processus calculant des *compléments de requêtes intéressants* permettant d'*affiner* une requête. La commande `ls` liste des sous-répertoires mais aussi des fichiers. Sous LFS, les fichiers listés dans un répertoire sont les fichiers dont la description satisfait la requête de ce répertoire, mais ne satisfait aucun de ses sous-répertoires.

Plus formellement, soit $\mathcal{P}$ l'ensemble des propriétés, soit $\mathit{ext}(f) = \{o \in \mathcal{O} \mid d(o) \models f\}$ l'extension d'une formule f, alors la réponse de `ls` dans un `PWD` pwd

est divisé en 2 parties, les sous-répertoires *Dirs*, et les fichiers *Files*, tel que :

$$\mathit{Dirs} = \mathit{max}_{\models}\{p \in \mathcal{P} \mid \emptyset \subset \mathit{ext}(pwd \wedge p) \subset \mathit{ext}(pwd)\}$$
$$\mathit{Files} = \mathit{ext}(pwd) - \bigcup_{p \in \mathit{Dirs}(pwd)} \mathit{ext}(p)$$

avec

$$\mathit{max}_{\models}(\mathcal{P}) = \{p \in \mathcal{P} \mid \neg\exists p' \in \mathcal{P}, p' \neq p \ \wedge \ p \models p'\}$$

Les sous-répertoires *Dirs* sont aussi appelés *incréments*.

Ainsi, dans l'exemple de la Section 2.3.1 page 70, avec les axiomes $movie \models art$ et $music \models art$, et le `PWD` $port \wedge USA$, on peut déduire que $\mathit{Dirs} = \{art\}$. En effet, $\mathit{ext}(pwd) = \{boston, losangeles, miami, sandiego, newyork\}$, et seul les formules $pwd \wedge art$, $pwd \wedge movie$ et $pwd \wedge music$ raffinent cette extension sans la rendre vide. Par exemple, $\mathit{ext}(pwd \wedge music) = \{boston, newyork\} \subset ext(pwd)$. Enfin, art est la propriété la plus générale car $max_{\models}(\{art, movie, music\}) = \{art\}$ du fait des axiomes.

Les fondements de LFS sont donc les notions de *logique*, d'*ensemble* et de *raffinement*, et les opérations de déduction logique et de manipulation d'ensembles qui vont avec.

2.6 Manipuler

Comme dit précédemment, LFS peut manipuler deux types d'objet : des fichiers et des parties de fichier. La commande `cd parts` permet de passer d'un monde à l'autre. Ce sont sensiblement les mêmes mécanismes d'organisation et de recherche qui sont utilisés dans les deux cas. Ainsi, les formules qui spécifient le comportement de la commande `ls` *ext* et *Dirs* sont les mêmes pour la navigation parmi les fichiers ou dans les fichiers. Seul le type des objets o dans ces formules diffère. On passe d'opérations sur des ensembles de fichiers à des opérations sur des ensembles de parties de fichier. La seule différence concerne la composante *Files*. Dans le monde des parties de fichier, la composante *Files* de la commande `ls` vaut toujours le même nom de fichier. Ce fichier est la *vue* du répertoire et son contenu est déterminé par l'extension *ext* de ce répertoire. La logique est la même dans les deux mondes. Enfin, tout comme les transducteurs permettent d'associer des propriétés aux fichiers, les transducteurs avancés permettent d'associer des propriétés aux parties de fichier.

Ce sont aussi sensiblement les mêmes mécanismes de manipulation et de modification qui sont utilisés. Dans le monde des fichiers les commandes de manipulation sont par exemple `rm`, `cp` et `mv`. Même si ces commandes ne sont pas autorisées dans le monde des parties de fichier, leurs alter-ego `cut`, `copy` et `paste` fonctionnent de manière similaire et offrent les mêmes services pour les parties de fichier. Nous allons préciser la sémantique de ces commandes de modification dans le monde des fichiers, puis dans le monde des parties de fichier.

2.6.1 Des fichiers

Le monde des fichiers est le monde par défaut ; toutes les commandes d'un *shell* sont autorisées. Il faut noter que comme LFS fonctionne au niveau du système de fichier, toutes les fonctionnalités de manipulation de fichiers offertes par le *shell* sont disponibles. L'utilisateur peut ainsi utiliser le *globbing*, c'est à dire l'étoile, par exemple avec la commande `rm f*`, ou bien encore la *completion* (voir [AL92]).

La commande `rm` permet d'enlever un objet o de $\mathcal{O}$. Elle efface une ligne dans la matrice formant le contexte. La commande `mv` permet d'ajuster la description d'un objet o. Le schéma général de cette commande est `mv p1/f1 p2/f2` ; cette commande a pour effet de changer le nom du fichier `f1` en `f2`, et de changer sa description en effaçant les propriétés mentionnées dans `p1` et en ajoutant celles mentionnées dans `p2`. Par exemple, du répertoire `/a/b/`, la commande `mv f ../c/` est équivalent à la commande `mv b/f c/f` exécutée depuis le répertoire `/a/` et ajoute donc `c` et enlève `b` à la description du fichier `f` (qui garde ici le même nom). Enfin, la modification du contenu d'un fichier, par exemple à l'aide d'un éditeur de texte, implique sa réindexation par les transducteurs appropriés.

Le même genre de schéma s'applique pour les propriétés. Ainsi, la commande `rmdir` supprime une propriété p de $\mathcal{P}$. Elle efface une colonne dans la matrice formant le contexte. La commande `mv` lorsqu'elle s'applique à une propriété modifie un peu comme pour les fichiers la description de cette propriété, c'est à dire les axiomes gouvernant cette propriété. Ainsi, la commande `mv /a/b/ab/ /a/c/ac/` renomme la propriété `ab`

en `ac` et enlève l'ancien axiome $ab \models b$ pour le remplacer par l'axiome $ac \models c$. Tous les fichiers qui avaient la propriété `ab` auront désormais la propriété `ac` et seront désormais disponibles à partir de `/a/c` (de même que de `/c` ou `/a`). Le comportement du `mv` sous LFS est ainsi très proche du comportement sous un système de fichier hiérarchique. Un ensemble de fichiers a été déplacé et est accessible d'un autre endroit.

2.6.2 Le contenu des fichiers

Comme dit auparavant, le monde des parties de fichier est accessible par le biais de la commande `cd parts`. Dans ce monde, l'utilisateur ne peut plus utiliser les commandes manipulant les fichiers comme `mv` ou `rm`. Désormais, les seules opérations permises sont les opérations modifiant le contenu des fichiers, en fait le contenu de vues.

Les transducteurs avancés

L'indexation d'un fichier se fait cette fois à l'aide d'un transducteur dit avancé. Comme les transducteurs «normaux», les transducteurs avancés prennent aussi en paramètre (sur leurs entrées standard) le contenu d'un fichier. En fait, ce contenu est vu cette fois comme une liste de parties : les parties du fichier. Un transducteur avancé ne retourne pas un unique ensemble de propriétés par fichier, mais un ensemble de propriétés par parties de fichier. Ainsi, lorsque l'utilisateur lance la commande `cd parts` dans un répertoire, LFS récupère d'abord l'extension de ce répertoire et applique ensuite les transducteurs avancés appropriés sur le contenu de ces fichiers et construit un nouveau contexte.

La définition de ce qu'est une partie peut être *orientée-ligne* ou *orientée-structure*. Dans le premier cas les parties sont des lignes, tandis que dans le second les parties sont faites d'une suite de mots formant un composant structuré, par exemple une expression dans le source d'un programme. Nous allons développer dans cette thèse uniquement l'orientation-ligne. Cependant, les principes de LFS ne dépendent pas de cette orientation et tout ce qui se dira sur les lignes pourrait s'appliquer aussi bien pour des parties avec une granularité plus fine comme des mots, ou plus grosse comme des paragraphes.

Plusieurs transducteurs peuvent être définis pour chaque type de fichier. Ces transducteurs sont appelés en séquence, et leurs résultats sont accumulés. Par exemple, si une partie reçoit la propriété a d'un transducteur, et la propriété b d'un autre transducteur, cette partie aura pour description la formule $a \wedge b$. Cela rend ainsi le processus d'indexation facilement extensible. Par exemple, le transducteur suivant définit ce qu'est un «aspect» gestion mémoire pour des programmes C :

```
foreach line
 if line matches
   (malloc|calloc|new|delete|free)
 then print "aspect:memory_managment\n"
 else print "\n"
```

Plus formellement, un transducteur avancé a pour type :

$$list(partcontent) \rightarrow list(set(property)) \ .$$

Il prend en paramètre une liste de parties et retourne un ensemble de propriétés pour chacune de ces parties. La convention d'interface est que les éléments de la liste sont séparés par un caractère de fin de ligne (`"\n"`), et les éléments de l'ensemble par un *slash*.

Il faut noter qu'un transducteur avancé prend en paramètre l'ensemble des parties du fichier. Le transducteur précédent pourrait laisser à penser qu'il suffirait de prendre les parties de manière indépendante, et donc d'enlever du transducteur avancé la ligne `foreach line`[4]. Cependant, contrairement aux fichiers les parties peuvent avoir des *liens* entre elles qu'il serait intéressant d'exploiter. Ainsi, en prenant l'ensemble des objets en paramètre, c'est à dire la liste des parties du fichier, les transducteurs avancés peuvent garder un *état* afin que l'indexation d'une partie puisse dépendre des autres parties. Par exemple, lorsque le transducteur avancé pour les programmes C passe sur la première ligne de la définition d'une fonction, il peut mémoriser dans son état le nom f de cette fonction, et ainsi ajouter aussi aux lignes suivantes la propriété `function`:f.

[4] Ce schéma montrerait d'ailleurs un plus grand parallèle entre les deux types de transducteurs. En effet, tout comme un transducteur normal indexe un fichier o en s'appliquant sur son contenu c, un transducteur avancé avec des objets étant des parties et non plus des fichiers, avec leurs propres descriptions d et leurs propres contenus c, indexerait une partie o en s'appliquant uniquement sur le contenu c de cette partie.

Ainsi, même avec un schéma *orienté-ligne*, LFS n'est pas *limité aux lignes*. LFS peut analyser des fichiers structurés comme des programmes. À l'opposé, des outils populaires comme `grep` sont limités aux lignes car ils ne peuvent gérer des «formes» qui ne rentrent pas sur une seule ligne. Par exemple, il n'y a aucun moyen avec `grep` ou avec le bouton de recherche d'un éditeur de chercher des paragraphes s'étalant sur plusieurs lignes contenant deux mots précis. Avec LFS, l'utilisation d'un transducteur avancé Perl de 5 lignes et d'une simple requête comme `cd contains:foo/contains:bar/` résoud le problème.

En fait, les propriétés attachées à une partie peuvent dépendre non seulement des parties précédentes, mais aussi des parties qui suivent. En effet, en imaginant une propriété `calls:`f («cette partie appelle la fonction f»), les propriétés de la première ligne d'une fonction dépendent du fait qu'il y ait ou non un appel à une fonction f dans les lignes suivantes.

Les marques d'absence

Ainsi, muni de ce nouveau contexte, construit par les transducteurs avancés, un utilisateur peut lancer des requêtes pour naviguer dans un fichier. Chaque répertoire contient une vue. Le contenu *View* d'une *vue* d'un répertoire *pwd* est défini par :

View(*pwd*) $= \{c(o), o \in \mathcal{O} \mid d(o) \models f\}$

Les parties sont ordonnées comme dans le fichier original. En fait, le contenu d'une vue est augmenté par des *marques d'absence*. En effet, afin de pouvoir *propager* les modifications d'une vue sur le fichier d'origine, LFS doit combiner les parties présentes dans la vue (modifiées ou non) et les parties manquantes. Le résultat doit être prévisible, et cohérent avec l'intention de l'utilisateur. Pour ce faire, ces parties manquantes doivent être rendues explicites dans la vue. En effet, lorsqu'un utilisateur ajoute une ligne dans une vue, LFS doit être capable de décider si il faut placer cette ligne avant ou après une partie manquante.

Supposons une vue schématique d'un fichier d'origine `abc` (où `a`, `b` et `c` représentent des groupes de lignes), et une vue qui cache `b`. Cette vue est affichée comme `a....c`, où `....` représente une marque d'absence. Si les parties manquantes n'étaient pas marquées (avec par exemple une vue affichée `ac`), l'insertion d'un `x` entre

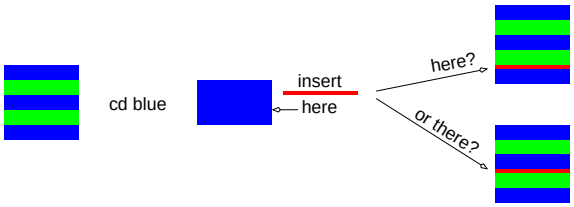

FIG. 2.12 – Intérêt des marques d'absence

`a` et `c` pourrait être interprétée soit comme l'insertion d'un `x` avant la partie manquante, soit comme l'insertion d'un `x` après la partie manquante, et pourrait donc être interprétée comme l'intention d'aboutir soit au résultat final `axbc`, soit au résultat `abxc` (voir Figure 2.12). De manière similaire, changer `a` en `a'` sans savoir où se situent les parties manquantes pourrait être interprété soit comme le désir d'aboutir à `a'bc`, ou `ba'c`, et même à `a'c`. Ainsi, il serait impossible d'inférer l'intention de l'utilisateur et donc l'effet attendu par la modification d'une vue sur le fichier d'origine. C'est donc pour résoudre ce problème que LFS insère des marques d'absence dans les vues à tous les endroits où des parties sont manquantes.

Afin de ne pas polluer une vue avec de trop nombreuses marques d'absence, LFS ne génère qu'une marque pour des parties manquantes consécutives. Un numéro unique est assigné à chaque marque afin de pouvoir désigner telle ou telle partie manquante. Cela permet de pouvoir effacer ou déplacer une partie manquante dans une vue, et de propager le résultat sur le fichier original. Ainsi, avec une vue n'affichant par exemple que les commandes de section d'un fichier LaTeX (`\chapter`, `\section`, etc) l'utilisateur peut déplacer des gros blocs de texte en ne déplaçant que deux lignes (le titre de la section et la marque d'absence qui suit ce titre).

Après la modification d'une vue, le transducteur avancé doit être réappliqué au fichier d'origine pour le réindexer car les propriétés des parties peuvent avoir changées. En effet, considérons par exemple le renommage d'une fonction ; cela change la propriété `function`: de toutes les lignes composant la définition de cette fonction, alors même que ces parties n'ont pas changées. Cette réindexation peut s'avérer coûteuse, mais nous verrons Section 3.1.3 page 102 un moyen pour

réduire la taille de la région à réindexer.

2.7 Conclusion

LFS supprime la rigidité des répertoires et fichiers en les rendant moins physiques. Ils ne sont plus le résultat d'un contenu physique sur le disque mais le fruit d'un calcul. Les répertoires et fichiers sont désormais tous les deux plus *virtuels*. D'un point de vue système, LFS va ainsi dans le sens des autre innovations systèmes comme la mémoire virtuelle, les pseudo-périphériques ou périphériques virtuelles, le multi-tâche qui peut être vue comme un service offrant des CPU virtuels, ou encore NFS qui là encore rend le réseau plus virtuel.

D'un point de vue plus général, LFS utilise plus les possibilités offertes par les technologies numériques. Normalement on utilise ces technologies pour justement s'affranchir des problèmes du monde physique. Cependant les systèmes de fichier hiérarchiques n'ont fait que copier ces problèmes dans le monde numérique, et n'ont ainsi pas su vraiment exploiter les avantages qu'offrent ces technologies. Certains [Pau90] pensent que l'évolution des découvertes suit régulièrement le même schéma avec 3 étapes. Il y a d'abord la découverte physique, purement technologique, dans notre cas le disque dur. Puis, on applique des techniques connues sur cette nouvelle technologie, dans notre cas on réapplique les principes d'organisation de l'information du monde réel (les principes hiérarchique d'organisation des livres, bibliothèques, etc) au disque. Nous pensons que LFS fait rentrer l'âge numérique encore un peu plus dans la troisième et dernière étape, l'étape où l'on exploite vraiment les nouvelles possibilités offertes par cette nouvelle technologie en offrant des services que seul cette technologie peut supporter.

La contribution de LFS par rapport aux autres systèmes de fichiers avancés, en plus d'avoir changé aussi le principe originel du fichier, est de permettre la combinaison sans restriction de l'*interrogation* et la *navigation* ainsi que de l'assignation *manuelle* et *automatique* de propriétés. Sous LFS, l'utilisateur peut notamment naviguer dans le résultat de n'importe quelle requête, et se voir proposer n'importe quel type de propriétés (intrinsèques ou extrinsèques).

Pour plus d'informations, l'Annexe A présente la spécification formelle de LFS et permet de combler les zones d'ombres qui pourraient subsister. Le Chapitre 3 présente les algorithmes et structures de données qui font de LFS une réalité. Même si les opérations mises en jeu comme avec par exemple la commande `ls` peuvent paraître coûteuses, car elles impliquent des calculs (ce qui n'était pas le cas dans les systèmes de fichier classique et ce qui restait très rudimentaire dans les systèmes de fichier avancés), les algorithmes utilisés par LFS ainsi que la puissance des processeurs modernes rendent tout à fait acceptables ces temps de calcul. Le Chapitre 6 confirme cela en présentant des benchmarks réalisés avec un prototype de LFS sur un grand nombre et une grande variété de données. Le Chapitre 4 parle d'un aspect important des systèmes de fichier qui n'a pas été évoqué dans ce chapitre : la sécurité. Enfin, le Chapitre 5 présente des extensions et raffinements. Ces différents chapitres sont indépendants et peuvent être lus dans n'importe quel ordre.

Chapitre 3

Algorithmes et structures de données

The best way to accelerate a Macintosh is at 9.8 m/s^2.

Marcus Dolengo

Nous venons de voir les principes et fonctionnalités offerts par LFS. Nous allons maintenant voir les algorithmes et structures de données pour implémenter efficacement ces fonctionnalités. Nous allons tout d'abord présenter les algorithmes et structures de données LFS répondant aux principaux problèmes de faisabilité d'un tel système. Nous résumerons ensuite l'ensemble de ces structures de données, ce qui fournira une bonne base pour comprendre les détails d'implémentation de LFS que nous présenterons juste après. LFS se conforme à l'interface des systèmes de fichiers Unix. Nous décrirons donc le code d'opérations concrètes comme `readdir` ou `lookup`.

Tout comme l'Annexe A permet de décrire complètement et précisément les principes de LFS, l'Annexe B décrit complètement et précisément les algorithmes et structures de données évoqués dans ce chapitre en présentant le code concret du prototype de LFS.

3.1 Algorithmes

Une solution naïve pour implémenter LFS serait de commencer par organiser l'information en enregistrant sur le disque sous la forme de méta-données : l'ensemble des propriétés $\mathcal{P}$ avec leurs noms, l'ensemble des axiomes gouvernant ces propriétés, et enfin l'ensemble des fichiers $\mathcal{O}$ avec leurs noms, leurs descriptions $d(o)$ et leurs contenus $c(o)$. Certains de ces fichiers, comme les moteurs de déduction logique et les transducteurs, pourraient être marqués comme spéciaux, afin de les retrouver plus facilement. Ces fichiers sont faciles à identifier car ils possèdent dans leurs descriptions l'une des propriétés spéciales `logic:`, `transducer:` ou `adv_transducer:`.

Ensuite, pour le calcul des sous-répertoires et fichiers d'un répertoire, ainsi que pour le calcul du contenu d'une vue, on pourrait suivre scrupuleusement la spécification de *Dirs*, *Files* (et de *ext* et $max_{\models}$) comme décrit dans la Section 2.5.2 page 81, et la spécification de *View* comme décrit Section 2.6.2 page 84. Il suffit pour cela de parcourir l'ensemble des propriétés $\mathcal{P}$, de parcourir l'ensemble des fichiers $\mathcal{O}$ et plus particulièrement l'ensemble des descriptions $d(o)$, et d'utiliser les axiomes et moteurs de déduction logique pour pouvoir comparer les formules.

Enfin, la modification d'une vue (voir Section 2.6.2 page 83) pourrait être gérée en propageant la modification sur le fichier original, et en rappelant les transducteurs pour réindexer toutes les parties de ce nouveau fichier. Ainsi, les calculs des sous-répertoires et des futurs vues seraient cohérents avec ce nouveau contenu.

En fait, l'Annexe A décrit précisément et complètement cette solution naïve. En effet, la spécification de LFS est une spécification exécutable. L'inconvénient de cette solution est qu'elle est trop coûteuse. Le temps de réponse d'un système de fichier doit être suffisamment court pour ne pas déranger l'utilisateur.

Ainsi, les principaux *challenges* algorithmiques pour

LFS sont :

1. organiser les propriétés de telle sorte que l'on puisse éviter le plus possible d'avoir à appeler les moteurs de déduction logique.
2. rendre rapide le calcul des sous-répertoires et fichiers d'un répertoire en évitant de parcourir l'ensemble des objets ou des propriétés.
3. trouver les moyens d'éviter de réindexer toutes les parties d'un fichier lorsque celui-ci se trouve modifié par l'utilisateur.

Les 3 sections suivantes présentent des réponses à chacun de ces challenges, c'est à dire comment *organiser*, *rechercher* et *manipuler* l'information efficacement.

3.1.1 Comment organiser ?

Les capacités de recherche de LFS font fortement usage de la déduction logique $\models$ avec la démonstration de formules comme $d(o) \models q$?, $d(o) \models f \wedge p$?, ou encore $f \models g$?.

Une optimisation importante de LFS, qui a déjà été évoquée de manière implicite dans le chapitre 2, est la *spécialisation* de la *logique du noyau* LFS. Ainsi, une partie des démonstrations de formules, comme celles concernant les conjonctions, disjonctions ou négations de propriétés simples (qui représentent une partie importante des requêtes d'un utilisateur) n'impliquent pas l'appel à des moteurs de déduction externes mais peut être réglé en interne par des algorithmes de déduction spécialisés.

Cependant, la relation de déduction $\models$ est définie aussi, en plus de la logique du noyau, par des axiomes et des moteurs de déduction logique externes. La composante interrogation de la commande `ls` (*Files*) cherche des fichiers, et a besoin de vérifier si la description d'un fichier, avec des propriétés spécifiques, satisfait le `PWD`, qui peut contenir des propriétés générales et des propriétés-formules complexes. La composante navigation de la commande `ls` (*Dirs*), cherche des sous-répertoires correspondant à des propriétés qui raffinent la requête, et a besoin de vérifier aussi si la description des fichiers satisfaisant la requête courante satisfait aussi ces propriétés. Enfin, la navigation compare les propriétés entre elles pour calculer les propriétés les plus générales. Tout ceci implique donc beaucoup de comparaisons de propriétés vis-à-vis de leur *ordre* logique. Il serait trop coûteux d'appeler les moteurs de déduction logique et de manipuler la liste des axiomes à chaque fois que l'utilisateur exécute la commande `ls`.

Le cache logique

La principale caractéristique de notre implémentation est de mémoriser le résultat de ces comparaisons dans un *cache logique*, ce qui évitera ainsi pour un grand nombre de comparaisons d'avoir à appeler un moteur de déduction.

Cependant, rien qu'avec un millier de propriétés, il existe déjà trop de comparaisons possibles entre deux propriétés pour pouvoir toutes les mémoriser. Ainsi, l'autre caractéristique du cache logique tire son idée du constat que le résultat de certaines comparaisons peut être utilisé pour inférer le résultat d'autres comparaisons. Par exemple, sachant que $a \models b$, et que $b \models c$, on peut inférer par transitivité que $a \models c$, sans avoir à appeler un moteur de déduction logique. Ainsi, au lieu de ne mémoriser qu'un sous-ensemble des comparaisons entre propriétés, on peut inférer toutes les comparaisons possibles en représentant dans le cache logique, via une hiérarchie des propriétés, l'ordre qu'impose $\models$. Le cache logique représentera de manière statique la déduction logique $\models$. Il permettra de classer les propriétés.

La Figure 3.1 présente un exemple de cache logique. Cet exemple pourrait résulter d'une suite de commandes de l'utilisateur : celui-ci aurait créé un ensemble de propriétés, fournit un ensemble d'axiomes (via `mkdir`) ainsi qu'un moteur de déduction logique (via ici la propriété `logic:prop`). Dans cet exemple, les propriétés sont soit les atomes a, b, c et d, soit des attributs valués $prop{:}x$ où la valeur x peut représenter une formule de la logique des propositions. L'ordre entre ces propriétés est déterminé par les axiomes, $b \models a$, $c \models b$, et un moteur de déduction logique pour la logique des propositions.

Il ne faut pas confondre la logique des propositions associée ici à la propriété `prop:` et gérée par un moteur de déduction externe, et la logique des propositions du noyau LFS. Ainsi, la requête `cd prop:w|z/` exprime la disjonction des propriétés `prop:w` et `z` et est gérée par la logique du noyau tandis que la requête `cd prop:(w|z)/` n'implique qu'une propriété, qui est gérée par le moteur de déduction externe. Les parenthèses

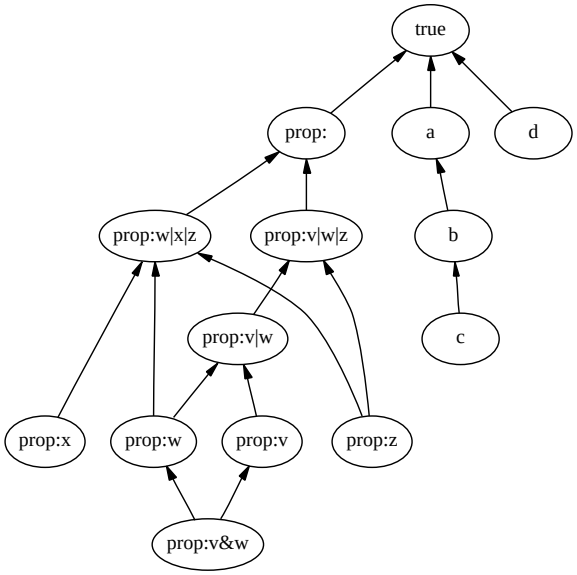

FIG. 3.1 – Un cache logique

permettent à l'utilisateur de lever l'ambiguïté (elles ont été omises dans les figures pour les simplifier). Ce moteur de déduction pour la logique des propositions n'est donc pas vraiment utile[1] car la logique du noyau permet déjà à l'utilisateur d'exprimer des conjonctions, disjonctions et négations de propriétés. Nous aurions pu prendre comme exemple de moteur de déduction logique une logique d'intervalle ou de type. Cependant, la logique des propositions est une logique plus complexe et permet donc de montrer toutes les difficultés associées à la gestion du cache logique. Les algorithmes que nous allons présenter plus loin doivent gérer n'importe quel type de logique, y compris les plus complexes. Des exemples avec une logique d'intervalle n'auraient pas mis en évidence toutes les subtilités de ces algorithmes.

Le cache logique est un *graphe orienté* où les nœuds sont des propriétés et où les arcs traduisent une relation de déduction. De manière plus générale, un graphe où les arcs traduisent une relation d'ordre partielle entre des nœuds s'appelle un *diagramme de Hasse* [DP90]. Bien sûr, l'idée de modéliser la logique via un graphe vient naturellement à l'esprit lorsque l'on veut représenter les axiomes. Après tout ces axiomes représentent une taxinomie, et donc une forme de hiérarchie et donc une forme de graphe. Cependant, la gestion de ce graphe est plus délicate lorsque l'on veut représenter l'ordre entre des propriétés issues d'une logique complexe.

Dans le cache logique, une propriété x est située sous une autre propriété y lorsque $x \models y$, c'est à dire lorsque x est plus spécifique que y. Par exemple, dans la Figure 3.1 `prop:v&w` est sous `prop:v|w` car effectivement dans la logique des propositions $v \wedge w \models v \vee w$ (tout document parlant de v et w parle forcément au moins de v ou w).

Ainsi, pour déterminer l'ordre entre deux propriétés, il suffit de parcourir ce graphe. Pour déterminer si $x \models y$, il suffit de trouver le nœud correspondant à x (ce qui peut être rendue efficace en utilisant une technique classique de *hash* [AU92] sur les noms des propriétés), et de remonter dans le graphe en testant la présence d'un nœud y.

Ajout de propriétés dans le cache logique

Toute la difficulté est de créer ce graphe, et de le modifier lorsque de nouvelles propriétés sont créées. Lorsqu'un nouvel atome et ses axiomes sont créés, par exemple à l'aide de la commande `mkdir a&d/e`, il faut localiser dans le graphe l'ensemble des nœuds correspondant aux propriétés du `PWD` (ici `a` et `d`), et rajouter sous ces nœuds un nouveau nœud correspondant à la nouvelle propriété (ici `e`, voir Figure 3.2).

Lorsqu'un nouvel attribut valué est créé, par exemple à l'aide de la commande `cd prop:(w|z)/`, la situation est plus compliquée. En effet, la valeur de l'attribut valué peut être une formule complexe. Cette nouvelle propriété peut représenter une catégorie intermédiaire : une *propriété-formule*. Il faut donc l'intercaler de la bonne manière parmi les autres propriétés du graphe. Le sous-ensemble de ce graphe relatif à l'attribut `prop:` est présenté dans la Figure 3.3. En fait, même une propriété «simple» comme `prop:w` doit être intercaler de la bonne manière car les autres propriétés peuvent etre des propriétés-formules.

L'insertion d'un attribut valué requiert donc un al-

[1] En fait nous verrons Section 6.2 page 147 que ce moteur peut être utile pour gérer certains domaines afin notamment de pouvoir matérialiser la notion *faux* tout comme la racine permet de matérialiser la notion *vrai*. La propriété *faux* sera tout en bas du cache logique alors qu'à l'opposé *vrai* est tout en haut (comme le montre la Figure 3.1).

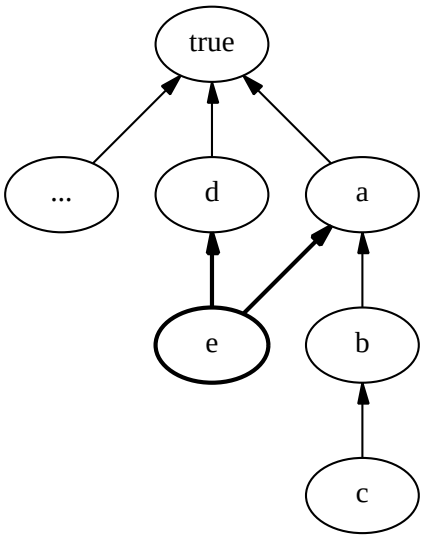

FIG. 3.2 – Ajout d'un axiome dans un cache logique

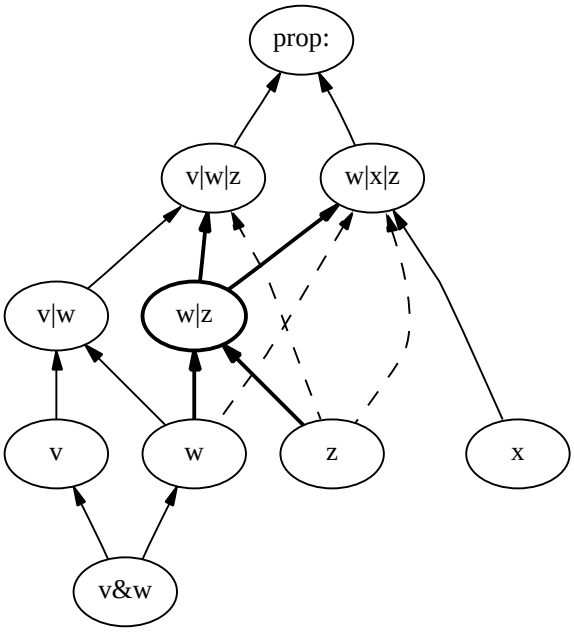

FIG. 3.3 – Ajout d'une propriété-formule dans un cache logique

gorithme plus sophistiqué que la simple création d'un nouveau nœud. L'idée générale de l'insertion d'une propriété f dans un graphe de propriétés $\mathcal{G}$, géré par un moteur de déduction logique $\models$, se décompose en 3 parties :

1. trouver les «parents» les plus spécifiques de la formule f dans $\mathcal{G}$; l'idée étant de parcourir les nœuds x de $\mathcal{G}$ de haut en bas tant que $f \models x$.
2. trouver les «enfants» les plus généraux de la formule f dans $\mathcal{G}$; l'idée étant de parcourir les nœuds x de $\mathcal{G}$ de haut en bas en partant des parents jusqu'à ce que $x \models f$.
3. insérer un nouveau nœud correspondant à la formule f dans $\mathcal{G}$ et ajuster le graphe en ajoutant des arcs entre f et ses parents, entre f et ses enfants, et en enlevant certains arcs qui deviendraient redondants.

Cet algorithme entraine un certain nombre de comparaisons de formules, et donc des appels au moteur de déduction logique responsable de ces formules. En effet, pour pouvoir insérer un attribut valué au bon endroit, on ne peut éviter d'avoir à le comparer avec les autres attributs valués. C'était le cas aussi dans la solution naïve présentée au début de ce chapitre. Cela dit, il est préférable, en terme d'efficacité, d'effectuer une fois pour toute ces comparaisons à la création de l'attribut valué qu'à chaque exécution par l'utilisateur de la commande `ls`.

On cherche les «parents» les plus spécifiques et les «enfants» les plus généraux afin d'ajouter ensuite un nombre d'arcs minimum (mais nécessaire) au bon fonctionnement du cache logique. C'est pour cette même raison que l'algorithme efface les arcs redondants.

Les trois sections suivantes détaillent chacune une des parties de l'algorithme d'insertion de formules.

Trouver les parents

La fonction find_parents prend en paramètre un graphe de formules `graph`, une formule `f` et un nœud de départ `parent`. Elle retourne, en parcourant récursivement `graph` vers le bas à partir de `parent`, un ensemble de nœuds qui correspondent aux formules du graphe les plus spécifiques impliquées par `f`. Le pseudocode de cette fonction est présenté ci-dessous :

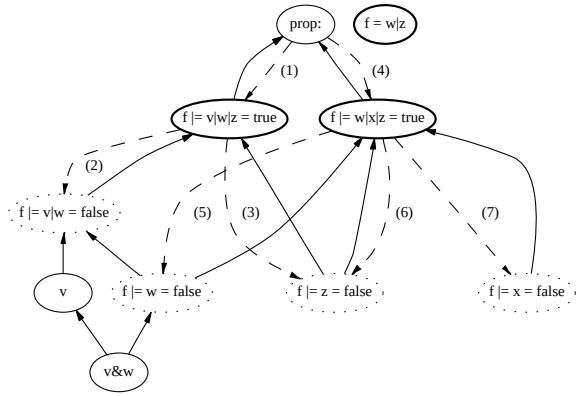

FIG. 3.4 – Calcul des parents

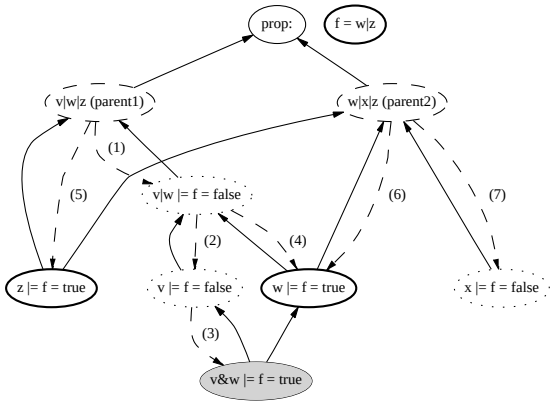

FIG. 3.5 – Calcul des enfants

```
find_parents(graph, f, parent)
 let children = successor(graph, parent)
 let implied = filter child from chidren
                such as (f |= child)
 in
  if empty(implied) then parent
  else
   foreach newparent from implied
    apply and collect
     find_parents(graph, f, newparent)
```

Cette fonction sera appelée par l'algorithme principal avec comme nœud de départ (l'argument `parent`) la racine du graphe. La fonction `successor` opère sur un graphe et permet de retourner les successeurs d'un nœud. La Figure 3.4 illustre le processus de calcul des parents de la propriété-formule `prop:(w|z)` dans le graphe de formules des exemples précédents.

Trouver les enfants

La fonction find_children prend en paramètre un graphe de formule `graph`, une formule `f` et un ensemble de nœuds de départ `children`. Elle retourne, en parcourant récursivement `graph` et `children`, un ensemble de nœuds qui correspondent aux formules du graphe les plus générales qui impliquent `f`. Le pseudo-code de cette fonction est présenté ci-dessous :

```
find_children(graph, f, children)
 if empty(children) then empty_set
 else
  let child  = head(children)
  let others = tail(children)
  in
   if (child |= f)
   then
    let final_children =
     find_children(graph, f, others)
    in
    add_child(child, final_children)
   else find_children(graph, f,
          union(successor(graph, child),
         others))
```

Cette fonction sera appelée par l'algorithme principal avec comme ensemble de départ (l'argument `children`) l'ensemble des enfants des parents de `f`. Ces parents auront été calculés à l'aide de la fonction find_parents. La Figure 3.5 illustre le processus de calcul des enfants de la propriété-formule `prop:(w|z)` dans le graphe de formules des exemples précédents.

La fonction find_children fait appel à la sous-fonction add_child. Cette fonction prend en paramètre une formule `child`, correspondant à un enfant, et un ensemble de formules `other_children`, correspondant à un ensemble d'enfants. Elle retourne, en ajoutant `child` dans `other_children`, un nouvel ensemble d'enfants. Le pseudo-code de cette fonction est présenté ci-dessous :

```
add_child(child, children)
```

```
if exist other from children
    such as (child |= other)
then children
else
  let newchildren =
   filter other from children
     such as not(other |= child)
  in
  union(child, newchildren)
```

Cette fonction n'ajoute pas simplement un élément dans un ensemble. En effet, la structure étant un graphe et non un arbre, certaines branches peuvent se rejoindre. Ainsi, les appels récursifs à find_children peuvent tomber une deuxième fois sur le même nœud. Dans ce cas, add_child ne doit pas ajouter cet enfant. En fait, dans certaines situations, il se peut que le parcours du graphe fait que `children` contienne en fait des enfants plus spécialisés que `child`. Dans ce cas, afin que find_children retourne les enfants les plus généraux, add_child remplace dans `children` les enfants plus spécialisés que `child` par `child`. La Figure 3.5 illustre d'ailleurs un exemple de ce genre de situation. Le nœud grisé est en première approche considéré comme un fils possible de la propriété-formule `prop:(w|z)` à l'étape 3 avant d'être finalement rejeté suite à la découverte à l'étape 4 qu'un enfant plus général existait (`prop:w`).

Ajuster

L'algorithme final insert_property, qui fait appel aux deux fonctions find_parents et find_children, est décrit ci-dessous.

```
insert_property(graph, f)
 let parents =
   find_parents(graph, f, top(graph))
 let candidates  =
   foreach parent from parents
    apply and collect
      successor(graph, parent)
 let children =
    find_children(graph, f, candidates)
 in
  if singleton(parents) and
     head(parents) |= f
then do nothing
else
 let fx = add_node(graph, f)
 in
 foreach parent in parents
  add_arc(graph, parent, fx)
  foreach child in
      inter(successor(graph, parent)
            children) do
    del_arc(graph, parent, child)
 foreach child in children
  add_arc(graph, fx, child)
```

Comme dit précédemment, cette fonction commence par chercher les parents et enfants de la formule à insérer. Lorsque cette formule n'a qu'un parent, et que ce parent implique aussi la formule, cela veut dire que les deux formules sont égales ou sémantiquement équivalentes. Dans ce cas il ne sert à rien de créer un nouveau nœud. Le graphe reste donc inchangé. Dans le cas contraire, on crée un nouveau nœud et on lie ce nœud avec ses parents et ses enfants. Enfin, le fait d'intercaler un nouveau nœud dans le graphe fait que les parents de certains enfants sont remplacés par leur nouveau père. La Figure 3.6 illustre le processus complet pour l'insertion de la propriété-formule `prop:(w|z)` dans le graphe de formules des exemples précédents.

Les différents algorithmes responsables de la gestion du cache logique sont aussi décrits en Annexe B.2.2 page 203.

Même si l'algorithme limite le nombre de comparaisons de formules nécessaires à l'insertion une nouvelle formule dans le cache logique (par exemple, ne sont pas considérées toutes les propriétés descendants d'une propriété qui n'est pas un parent), l'insertion d'une nouvelle formule dans le cache logique reste une opération coûteuse. En effet, chaque comparaison implique l'appel au moteur de déduction externe et implique donc une communication entre deux processus. Cependant, effectuer ce travail uniquement à la première mention d'une nouvelle formule est bien moins coûteux à long terme que de le faire à chaque `ls`.

De plus, comme dit précédemment, une optimisation importante de LFS est la *spécialisation*[2] de la lo-

[2]Il faut noter que cette spécialisation n'empêche cependant pas LFS d'être générique vis-à-vis de la logique puisque cette logique peut être

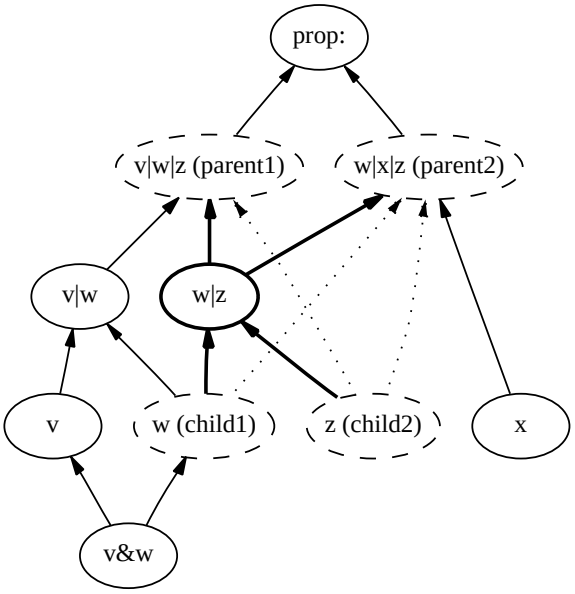

FIG. 3.6 – Insertion d'une formule

gique du noyau LFS. Ainsi, la mention de conjonctions, de disjonctions ou négations de propriétés, qui représentent une partie importante des requêtes d'un utilisateur (en particulier, rien que le fait de rentrer dans un sous-répertoire ou de se voir proposer un sous-répertoire conduit à la formation d'une nouvelle conjonction), n'entrainent pas l'insertion de formules dans le cache logique. Seul la mention de nouveaux attributs valués dont l'attribut est géré par un moteur de déduction entraîne l'appel aux algorithmes décrits précédemment (et donc l'appel au moteur de déduction), comme par exemple lorsque l'utilisateur formule pour la première fois la requête `cd size:>100/`.

Optimisation

Une autre optimisation importante concernant la logique consiste à spécialiser encore un peu plus la logique du noyau en déplaçant du travail fait normalement par les moteurs de déduction externes en interne.

L'idée de cette optimisation vient du constat que lorsqu'un attribut comme `size:` n'est pas géré par un moteur de déduction, l'utilisateur ne peut certes pas formuler des requêtes avancées avec des propriétés-formules comme dans `cd size:>100/`, mais il peut tout de même utiliser des propriétés «simples» comme `size:100`, et les différentes opérations LFS sont alors très rapides. Il peut naviguer parmi ces propriétés, formuler des disjonctions ou négations sur ces propriétés. C'est donc déjà un service appréciable. La Section 2.3.1 page 72 illustrait ces possibilités avec l'attribut `type:` et des propriétés comme `type:picture` ou `type:program`. Cependant, si l'utilisateur décide au bout d'un moment d'associer un moteur de déduction sur ce même attribut, pour pouvoir ensuite formuler des requêtes plus avancées (par exemple une logique d'entier sur `size:`), la mention d'une nouvelle propriété, même si ce n'est pas une propriété-formule, même si le cache logique ne contient que des propriétés simples, impliquera bien plus de calculs. Le fait d'associer un moteur de déduction à un attribut change donc du tout au tout la complexité des opérations. D'une certaine manière, l'utilisateur «paye pour ce qu'il n'utilise pas encore», puisque sans se servir des propriétés-formules, et donc avec un service équivalent, les opérations deviennent bien plus coûteuses.

En effet, avec un domaine non géré par un moteur de déduction, tous les attributs valués sont rangés à plat sous l'attribut principal (ici `type:` ou `size:`) et l'insertion d'une nouvelle propriété consiste juste à insérer un nouveau fils dans le cache logique sous l'attribut (comme pour les axiomes). LFS ne s'occupe pas de savoir si la valeur de l'attribut valué est un entier, ou une expression régulière. Il traite toutes ces valeurs de manières homogènes comme de simples chaînes de caractères. Ainsi, l'ajout en bloc de n nouvelles propriétés est très rapide, et ne nécessite bien sûr aucun appel à un programme externe. En fait, c'est comme si ces propriétés étaient gérées par une logique par défaut très simple : l'égalité ($\models \equiv =$). On teste juste si la propriété est déjà présente, ce qui est très rapide en utilisant une table de *hash* sur les propriétés.

Avec un domaine géré par un moteur de déduction, le problème est que LFS ne sait pas si la propriété à insérer est comme précédemment une propriété simple, auquel cas on pourrait parfois appliquer l'ancienne technique, ou si c'est une propriété-formule. Seul le moteur externe connaît cette information. Ainsi, lorsque tous les attributs valués sont à plat (comme c'est souvent le cas au dé-

étendue avec les moteurs de déduction.

but), l'insertion d'une nouvelle propriété-simple passe de $\mathcal{O}(1)$ avec la logique par défaut ($=$), à $\mathcal{O}(n)$ avec la logique du moteur de déduction, où n représente le nombre de propriétés du domaine déjà présentes dans le cache. En effet, l'algorithme insert_property doit comparer la propriété-simple avec toutes les autres propriétés.

Ainsi, l'idée est d'optimiser ce schéma en faisant en sorte que LFS sache aussi si la propriété en jeu est une propriété-simple, que l'on peut appelé *propriété-valeur* par opposition à *propriété-formule*. Par exemple, avec une logique d'entier la propriété `size:100` est une propriété-valeur et `size:>100` une propriété-formule, avec une logique de chaîne `function:toto` est une valeur et `function:.*ot?` une formule, avec la logique des propositions vues précédemment `prop:v` est une valeur et `prop:(w|z)` une formule. De nombreuses logiques ont ces notions de valeur et de formule (qui divisent donc le domaine de cette logique en 2 grandes classes de propriétés), et peuvent donc bénéficier de cette optimisation. Ainsi, lors de l'insertion d'une propriété-simple x, si LFS lors de l'exécution de insert_property a besoin de comparer x à une propriété y (avec $x \models y$? ou $y \models x$?), et que y est aussi une propriété-simple, alors il n'est nul besoin d'appeler le moteur de déduction car le moteur retournerait le même résultat que la logique par défaut $=$, c'est à dire faux.

Concrètement, il suffit d'augmenter l'interface des moteurs de déduction pour qu'ils reconnaissent l'option `-is_formula`. En reprenant la session de la Section 2.3.1 page 73, les commandes suivantes illustrent cette nouvelle interface :

```
[72] % int_logic -is_formula  '2001'
no
[73] % int_logic -is_formula  '>1000'
yes
```

Lors de la mention d'une nouvelle propriété x, LFS interrogera le moteur de déduction, ce moteur indiquera à LFS quelle est la classe de cette propriété (formule ou valeur), LFS stockera cette information en interne pour accélérer les futures opérations d'insertion, et enfin LFS appellera l'algorithme insert_property pour cette propriété x.

Avec cette optimisation l'insertion d'une propriété-valeur (ce qui représente la majorité des insertions), dans un cache logique ne contenant que des propriétés-valeurs (ce qui est souvent le cas au début) sera désormais très rapide et prendra le même temps qu'avec la logique par défaut[3]. On ne paye donc pas si on n'utilise pas. L'insertion d'une propriété-valeur dans un cache logique contenant des propriétés-formules sera plus coûteuse car elle nécessite un certain nombre de comparaisons car il faut «ranger» cette propriété. Cependant, on n'aura besoin de comparer cette propriété-valeur qu'avec les propriétés-formules du cache, et donc pas avec les autres propriétés-valeurs. Comme souvent le cache logique induit par une logique forme un arbre (par exemple avec la logique d'entiers les formules sont au nœuds et les valeurs aux feuilles), le nombre de nœuds est bien inférieur au nombre de feuilles et donc le nombre de formules est bien inférieur au nombre de valeurs. On passe donc en complexité de $\mathcal{O}(n)$ à $\mathcal{O}(f)$ avec $f \ll n$, où f représente le nombre de propriétés-formules déjà présentes dans le cache et n le nombre total de propriétés (formules et valeurs). Enfin, l'insertion d'une propriété-formule sera comme avant cette optimisation coûteuse et au pire cas en $\mathcal{O}(n)$. Cependant, comme nous le verrons à la Section 6.3.3 page 166, ce temps d'insertion est de l'ordre de la seconde.

Il faut noter que cette optimisation est très importante. En effet, on peut se permettre d'attendre une seconde lors de l'insertion d'une propriété-formule car c'est un cas qui est rare. Il n'a en général lieu qu'à la demande de l'utilisateur lors d'un `cd`, et ne concerne donc qu'une propriété à la fois. On ne pourrait cependant pas se le permettre pour l'insertion d'une propriété-valeur, comme c'était le cas avant cette optimisation, car c'est un cas bien plus fréquent. Par exemple, la copie d'un fichier sous LFS implique en général l'appel à un transducteur et donc l'insertion d'une dizaine de nouvelles propriétés[4]. C'est encore pire lors d'un `cd parts` qui peut impli-

[3] En fait, l'opération sera légèrement plus coûteuse car il aura fallu tout de même appeler le moteur de déduction une fois pour effectivement savoir si la propriété était bien une propriété-valeur.

[4] Du fait que les propriétés sont en général partagées par plusieurs fichiers, l'extraction de 10 propriétés par un transducteur n'induit pas forcément l'insertion de 10 propriétés. Par exemple, deux fichiers peuvent avoir la même taille. Lorsqu'une propriété existe déjà (qu'elle soit d'ailleurs une propriété-formule ou une propriété-valeur), la mention de cette propriété n'implique pas d'appel à insert_property. Cela réduit donc le nombre de nouvelles propriétés mais ce nombre reste tout de même important.

quer des milliers de propriétés, qui sont en général toutes des propriétés-valeurs.

3.1.2 Comment rechercher ?

Le cache logique, même si il permet de répondre rapidement à la question $x \models y?$, ne résoud pas tous les problèmes. Un problème restant important est de pouvoir répondre rapidement à l'exécution de la commande `ls`. En effet, cette commande est l'opération la plus originale de LFS, et aussi la plus coûteuse. Dans un système de fichier hiérarchique, cela consiste juste à lire le contenu d'un bloc sur le disque représentant le répertoire. Sous LFS, cela implique un vrai calcul. En effet, l'«algorithme `ls`» doit calculer les ensembles *Files* et *Dirs*, afin de pouvoir lister les fichiers et sous-répertoires d'un répertoire (voir la définition de *Files* et *Dirs* Section 2.5.2 page 81). Ce calcul implique, en plus de l'utilisation du cache logique, des opérations ensemblistes, de parcourir l'ensemble des objets et leurs descriptions, ou encore de parcourir l'ensemble des propriétés. Il faut donc trouver des structures de données supplémentaires complétant le cache logique pour rendre ces calculs efficaces.

Il faut noter qu'il y a tellement de combinaisons possibles de propriétés qu'il n'est pas faisable de créer et pré-calculer statiquement tous les répertoires correspondants. Par exemple, rien qu'avec les propriétés `a` et `b`, l'utilisateur peut former les répertoires `a/b`, `b/a`, `a|b`, `a/!b`, etc. De plus, même si nous pouvions tous les créer, chaque fois que l'utilisateur ajouterait ou effacerait un fichier, le contenu de tous ces répertoires devrait être recalculé car les parties *Files* ou *Dirs* pourraient avoir changées. Ainsi, les répertoires, qui nous le rappelons correspondent à des requêtes de l'utilisateur, sont créés *à la demande*.

Nous présentons d'abord un algorithme naïf afin d'identifier «les points chauds» du calcul. Puis nous présenterons les structures de données supplémentaires complétant le cache logique qui rendent possible la définition d'un meilleur algorithme qui évite ces points chauds.

L'algorithme naïf

L'algorithme naïf s'inspire de la spécification. Les formules définissant *Dirs* et *Files* sont on le rappelle :

$$\mathit{Dirs} = \mathit{max}_{\models}\{p \in \mathcal{P} \mid \emptyset \subset \mathit{ext}(p \wedge pwd) \subset \mathit{ext}(pwd)\}$$
$$\mathit{Files} = \mathit{ext}(pwd) - \bigcup_{p \in \mathit{Dirs}(pwd)} \mathit{ext}(p)$$

avec

$$\mathit{max}_{\models}(\mathcal{P}) = \{p \in \mathcal{P} \mid \neg\exists p' \in \mathcal{P}, p' \neq p, p \models p'\}$$

et

$$\mathit{ext}(f) = \{o \in \mathcal{O} \mid d(o) \models f\}$$

On représente chaque propriété de $\mathcal{P}$ par un *identifiant de propriété interne* p_i, et chaque objet de $\mathcal{O}$ par un *identifiant d'objet interne* o_i.

Une optimisation importante de LFS est la spécialisation de la logique du noyau LFS. En effet, la mention de conjonctions, de disjonctions ou négations de propriétés, qui représentent une partie importante des requêtes d'un utilisateur, n'entrainent pas l'insertion de formules dans le cache logique ni d'appel à des programmes externes. L'algorithme naïf que nous allons présenter n'est donc en fait pas l'algorithme le plus naïf. LFS reconnaît donc lorsqu'un répertoire à la forme d'une formule de la logique des propositions. Un répertoire (et donc une formule) est donc représentée en interne par une liste d'éléments, où chaque élément est soit un identifiant de propriété, soit un *tag* `Or` associé à une liste d'identifiants de propriétés (les opérandes de la disjonction), soit le *tag* `Not` associé à un identifiant de propriété. La liste joue le rôle de la conjonction. Cette représentation correspond à la forme normale conjonctive d'une formule de la logique des propositions.

Pour représenter la description $d(o)$ d'un objet, on utilise une table `obj->props` indexée par les identifiants d'objets, et qui retourne une liste d'identifiants de propriétés. Elle correspond aux lignes de la matrice *objet* $\times$ *propriété*.

Comme dit précédemment, LFS utilise le cache logique pour pouvoir comparer rapidement deux propriétés. Cela permet d'éviter d'appeler les moteurs de déduction logique qui peuvent être coûteux. Ce cache est un graphe de propriétés ; il permet de représenter l'ordre entre les propriétés, que ce soit l'ordre induit par les axiomes, ou l'ordre induit par les moteurs de déduction logique. On représente ce graphe à l'aide de deux tables : la table `prop->children` qui associe à un identifiant de propriété p_i une liste d'identifiants de propriétés correspondant aux successeurs de p_i dans le graphe, et la table `prop->parents` qui fait la même chose mais pour les prédécesseurs.

L'algorithme `ls` fera appel à la sous-fonction `ext` qui

permet de calculer l'extension d'une formule (voir la définition de *ext*). Cette fonction prend en paramètre une formule `f`, correspondant en général à un `PWD`, et retourne un ensemble d'identifiants d'objets, correspondant à l'extension de `f`. Le pseudo-code de cette fonction est présenté ci-dessous :

```
ext(f) =
 filter o from O
  such as
   let d = obj->props[o] in
   check that forall x in f
     if x is a single property   then
       check_downwards(x, d)
     if x is a Or with elements xs then
       check exist y from xs
        such as check_downwards(y, d)
     if x is a Not y  then
       check not check_downwards(y, d)

check_downwards(x, xs) =
  if member(x, xs)  then true
  else
   if empty(prop->children[x])
   then false
   else
    exist y from prop->children[x]
     such as check_downwards(y, xs)
```

La fonction ext, tout comme dans sa spécification formelle, parcourt l'ensemble des objets avec ($\{o \in \mathcal{O} \mid ...\}$) qui devient `filter o from O ...`. Cependant, grâce à la spécialisation de la logique sous LFS à une logique des propositions et l'utilisation d'un cache logique pour les autres logiques, on évite au moins la partie coûteuse $d(o) \models f$ (en la remplacant pour simplifier par $d(o) \supseteq f$ avec la fonction `member`). Ce n'est donc pas tout à fait l'algorithme naïf qui appellerait les moteurs de déduction logique.

La fonction ext parcourt aussi la formule `f`, c'est à dire la liste représentant une conjonction. La fonction check_downwards traverse récursivement le cache logique. En effet, si un objet x a dans sa description la propriété b, et si $b \models a$ (c'est à dire que b est un descendant de a dans le graphe), alors x est aussi dans l'extension de a.

Ainsi, l'algorithme ls est :

```
ls(pwd) =          # pwd is a formula
 let old  = ext(pwd)

 let ps  =        # possible increments
  filter p from P such as
   let new = inter(ext(p), old) in
   0 < card(new) < card(old)

 let maxps =  # maximal increments
    filter p from ps such as
      not check_upwards(p, minus(ps, p))

 let esubdirs =
     foreach p in maxps
      compute union of ext(p)

 in Dirs  = maxps
    Files = minus(old, esubdirs)



check_upwards(x,xs) =
 if member(x, xs)  then true
 else
  if is_top(x) then false
  else
   exist y from prop->parents[x]
    such as check_upwards(y,xs)
```

où les fonctions `minus`, `union`, `inter` et `card` (cardinalité) sont les fonctions usuelles d'une librairie sur les ensembles.

On retrouve dans la fonction ls des morceaux de la spécification formelle comme par exemple $\{p \in \mathcal{P} \mid ...\}$ qui devient `filter p from P ...`. Comme pour `ext`, la spécialisation de la logique sous LFS à une logique d'ensemble et l'utilisation d'un cache logique pour les autres logiques permet de remplacer les $\models$ présents dans la spécification formelle par du code plus efficace. Ainsi, le $p \models p'$ présent dans la définition de $max_{\models}$ est remplacé par l'utilisation de la fonction check_upwards qui utilise le cache logique pour comparer les propriétés.

Les points chauds et un meilleur algorithme

L'algorithme naïf, même si il réduit certains coûts grâce au cache logique, est encore trop coûteux. En effet, chaque appel à `ext` (1) parcourt tous les objets (voir ligne `filter o from O`), et (2) peut impliquer une exploration récursive de tout le graphe, et donc de toutes les propriétés, avec la fonction check_downwards. Ainsi, le temps de réponse au pire cas pour la fonction `ext` est au moins proportionnel à $\|\mathcal{O}\|.\|\mathcal{P}\|$. Enfin, chaque appel à ls (3) parcourt toutes les propriétés (voir ligne `filter p from P`), et calcule l'extension de chaque propriété p, ce qui implique encore de parcourir tous les objets. De plus, comme pour (2), chaque appel à ls peut impliquer une exploration récursive d'une grande partie du graphe avec la fonction check_upwards. Ainsi, le temps de réponse au pire cas pour la fonction ls est au moins proportionnel à $\|\mathcal{P}\|.(\|\mathcal{O}\|.\|\mathcal{P}\|)$.

1. **Parcourir tous les objets**. Comme nous venons de le voir, partir des objets et vérifier si leur description satisfait la formule est coûteux. Cela donne l'idée de faire l'opposé, c'est à dire de partir de la formule pour calculer les objets qui satisfont cette formule.

   Ainsi, sous LFS les extensions ne sont pas calculées à l'aide de la table `obj->props`, mais à l'aide de la table inverse `prop->objs`. On utilise une technique classique d'optimisation : l'*indexation*. On *indexe* les propriétés. Cette table correspond aux colonnes de la matrice *objet* $\times$ *propriété*. On appelle dans la litérature [BYRN99] ce type d'index des *inverted index*. L'extension de $a \wedge b$ est calculée en intersectant les extensions de a et de b, c'est à dire *ext*$(a \wedge b) =$ `prop->objs[a]` $\cap$ `prop->objs[b]`. De manière similaire, *ext*$(a \vee b) =$ `prop->objs[a]` $\cup$ `prop->objs[b]`, et *ext*$(\neg a)$ $=$`complement(prop->objs[a])`. L'hypothèse sous-jacente pour l'utilisation de cette table inverse est que le nombre de propriétés mis en jeu dans un chemin et dans la description d'un fichier est petit vis-à-vis du nombre total de fichiers ce qui est le cas. En effet, la plupart des propriétés sont extraites automatiquement par des transducteurs qui ne retournent en général qu'une vingtaine de propriétés par fichier (et non 100 000, ce qui pourrait être par contre le nombre de fichier sous LFS). Enfin, la formule du `PWD` est en général petite car elle est le résultat de la trace d'une navigation faite par un humain. La conjonction de quelques propriétés réduit déjà considérablement le nombre de fichier satisfaisant cette requête et donc l'utilisateur n'aura pas à aller plus loin.

   Cela dit, le calcul de l'extension d'une propriété a impliquerait toujours un processus récursif afin de fusionner `prop->objs[a]` avec les valeurs `prop->objs[x]` des sous-propriétés x de a.

2. **Le processus récursif**. La seconde idée, afin d'éviter un calcul récursif pour l'extension, est d'inclure les valeurs `prop->objs[x]` des sous-propriétés x d'une propriété a dans `prop->objs[a]`. On utilise une autre technique classique d'optimisation : l'*inlining*.

   Ainsi, la table `prop->objs` ne contient plus seulement pour chaque propriété les objets décrits par cette propriété, mais aussi les objets décrits par les sous-propriétés de cette propriété. Cela revient à pré-calculer l'extension de toutes les propriétés. On appelle donc cette table la *table des extensions*. Il faut noter que cela n'est pas la même chose que de pré-calculer le contenu des répertoires. En effet le nombre des propriétés est bien plus petit que le nombre des combinaisons possibles de propriétés, c'est à dire de répertoires-requêtes possibles.

   De ce fait, ajouter un fichier avec la propriété x requiert de mettre à jour (en utilisant la table `prop->parents`) l'entrée de x et de tous ses ancêtres dans `prop->objs`. Cela est coûteux, mais nous pensons que la principale opération à optimiser est la consultation. Nous préférons donc transférer la grosse partie du travail dans les commandes de création. De plus, la création ne requiert pas une mise à jour immédiate ; son temps de réponse peut être suffisamment bon en laissant se faire les calculs de mise à jour en tâche de fond.

   Les entrées de cette table peuvent contenir de nombreux objets et donc contenir des ensembles de tailles importantes. Une optimisation importante de LFS est de compresser ces extensions en utilisant une représentation en intervalle (où une liste d'éléments successifs est uniquement représentée par le

min et le *max*). Cela permet non seulement de gagner de la place et donc aussi du temps lors des lectures/écritures de ces extensions sur le disque, mais aussi d'accélérer encore un peu plus l'algorithme ls puisque l'on peut alors utiliser des algorithmes sur les ensembles spécialisés pour la représentation par intervalles. Les opérations d'intersection, d'union ou de différence utilisés dans l'algorithme ls sont alors plus rapides. Cette compression est particulièrement utile pour les propriétés les plus générales, car elles contiennent dans leurs extensions pratiquement tous les objets.

Ainsi, la table `prop->objs` retourne les extensions des propriétés en $\mathcal{O}(1)$. Calculer `ext(PWD)` coûte seulement quelques accès ($\|PWD\|$) à la table `prop->objs`. L'algorithme naïf causait $\|\mathcal{O}\|$ accès à la table `obj->props`, ce qui est trop coûteux. En effet, ces consultations induiraient de nombreux accès au disque, même si l'on arrivait à rassembler ensemble des entrées sur le même secteur d'un disque.

3. **Parcourir toutes les propriétés**. Un calcul rapide des extensions ne résoud pas le problème (3) ; la fonction ls a toujours besoin de parcourir toutes les propriétés. Comme pour le problème (1), à la place de parcourir toutes les propriétés pour les trier ensuite, la dernière idée est de faire l'opposé, c'est à dire de partir des propriétés triées directement. C'est justement l'un des services fournit par le cache logique.

L'algorithme final

Le pseudo-code de l'algorithme ls final qui utilise la nouvelle table `prop->objs` est décrit ci-dessous :

```
ext(f) =
 let set =
  foreach x in f
   compute intersection of
    either x is a single property
      then prop->objs[x]
    either x is a OR with elements xs
      then
       forall y in xs
         compute union of prop->objs[y]
    either x is a Not y
      then do nothing
 in ext := set
    foreach x in f
      if x is a Not y then
        ext := minus(ext, prop->objs[y])
       otherwise do nothing
    return ext

ls(pwd) =
 let old  = ext(pwd)
 let ps  =
   collect properties p
     from logic cache
     using a depth first search
     starting from the top node
     using prop->children
    until
     let new = inter(prop->objs[p], old) in
     0 < card(new) < card(old)
 let esubdirs =
    foreach p in ps
     compute union of prop->objs[p]
 in Dirs = ps
    Files = minus(old, esubdirs)
```

Ainsi, l'algorithme final localise d'abord les identifiants des propriétés qui sont présentes dans le PWD, et calcule l'extension de ce PWD. Puis il parcourt le cache logique à partir de la racine vers le bas, tant que l'extension des nœuds traversés contient l'extension du PWD. Les propriétés les plus hautes dont l'extension ne contient pas l'extension du PWD, mais qui a une intersection non vide avec elle, sont les incréments à lister dans *Dirs*. Ce sont les propriétés les plus générales permettant de raffiner la requête.

La Figure 3.7 illustre cet algorithme. Le contexte est le même que celui de la Figure 2.7 page 71. Chaque nœud du cache logique est représenté par le nom de la propriété correspondante, et son extension (par exemple, l'intervalle $[1..7]$ pour le nœud racine). Les lignes épaisses représentent les arcs du cache logique. Les deux lignes fines pointent vers les propriétés du PWD. Les flèches pointillées représentent le parcours fait par l'algorithme afin de répondre à la commande ls. Pour chaque nœud traversé, l'algorithme compare l'extension de ce nœud

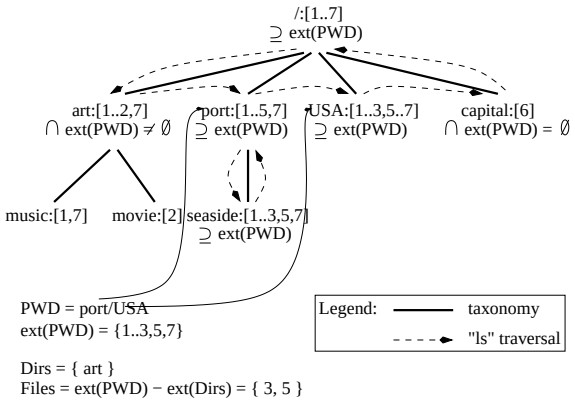

FIG. 3.7 – L'algorithme ls final

avec l'extension du PWD ; le résultat de cette comparaison est indiqué sous la description du nœud (par exemple $\supseteq\ ext(\texttt{PWD})$ pour le nœud racine). L'algorithme entre dans les sous-nœuds uniquement si le résultat est $\supseteq ext(\texttt{PWD})$. Les sous-répertoires sont ceux qui ne contiennent pas le PWD, mais qui ont une intersection non vide avec (par exemple art). $capital$ n'est pas un sous-répertoire car il n'intersecte pas le PWD ; il ne raffine pas la requête. Les fichiers à lister sont ceux qui font parties de l'extension du PWD mais pas de l'extension d'un sous-répertoire (c'est à dire les objets 3 et 5).

Cette technique évite d'avoir à parcourir toutes les propriétés. En effet, les sous-propriétés d'une propriété dont l'extension n'intersecte pas avec celle du PWD ne sont pas considérées, puisque si `prop->objs[`x`]` n'intersecte pas l'extension du PWD, aucun descendant de x ne le fera non plus car `prop->objs[`x`]` contient aussi les extensions («*inlinées*») de ses enfants. De la même manière, les sous-propriétés d'une propriété dont l'extension intersecte strictement avec celle du PWD ne sont pas non plus considérées, parce que ces sous-propriétés, même si leurs extensions intersectaient celle du PWD, ne seraient pas les propriétés les plus générales.

Les algorithmes ls et ext sont aussi décrits en Annexe B.2.1 page 202.

Optimisation

Les algorithmes précédents requièrent la création d'index (la table `prop->objs`). Cette indexation pour qu'elle ne soit pas trop coûteuse suppose que le nombre de propriétés par fichier reste petit (moins d'une centaine). Cela est effectivement vrai avec les propriétés extrinsèques puisque l'utilisateur n'est pas prêt à passer son temps à associer des centaines de propriétés à son fichier. C'est aussi vrai avec les propriétés intrinsèques puisque l'on a tendance à associer avec les transducteurs des propriétés de haut niveau («sémantiques») au fichier, qui sont donc souvent peu nombreuses.

Cependant, l'utilisateur aimerait aussi formuler des requêtes comme `cd contains:foo/` pour récupérer les fichiers contenant le mot `foo`. La recherche sur l'intégralité du texte des fichiers est une méthode populaire de recherche (*full-text retrieval*[BYRN99]) et offre un service appréciable. Écrire un transducteur pour extraire des propriétés de la forme `contains:`x serait facile, mais il en résulterait des milliers de propriétés par fichier et notre schéma d'indexation ne le supporterait pas.

L'utilisateur pourrait lancer la commande `grep foo` sur l'ensemble de ses fichiers mais c'est un processus trop long lorsque cet ensemble est très grand. Sous UNIX il existe un équivalent rapide à `grep` : Glimpse [MW94][5]. Cet outil construit comme LFS un index pour être efficace, mais il utilise une technique d'indexation adaptée au contexte de l'indexation *full-text* (un index à multiples niveaux ; voir [MW94] pour plus de détails).

Cependant, comme pour `grep`, Glimpse ne supporte que l'interrogation. Ainsi, si un grand nombre de fichier satisfait la requête, l'utilisateur n'est pas beaucoup plus avancé. On aimerait naviguer à partir de cette requête. Une des optimisations de LFS consiste donc à intégrer Glimpse dans LFS[6].

Le principe est de faire de `contains` : une propriété spéciale et de spécialiser l'algorithme ext. Lorsque la propriété en jeu commence par `contains` : alors ext

[5] Il existe aussi sous UNIX un équivalent rapide à `find` : `locate`[Woo83] (en fait à `find -name`).

[6] L'outil `locate` utilise aussi une technique d'indexation particulière avec un schéma de compression adapté aux noms de fichiers et chemins. Cependant l'extraction des propriétés systèmes `name` : alliée à l'indexation «naïve» de LFS est suffisante et nous n'avons pas donc besoin d'intégrer `locate` à LFS.

n'utilise pas la table `prop->obj` mais appelle Glimpse pour calculer l'extension de cette requête. Il faut noter que ls ne change pas. Après une requête mentionnant une propriété `contains`: LFS calculera aussi des incréments. L'utilisateur a donc la possibilité d'interroger aussi avec des propriétés de «bas niveau» et de naviguer ensuite avec des propriétés de «haut niveau».

3.1.3 Comment manipuler ?

Le dernier problème important pour LFS est celui du coût de l'indexation des fichiers par les transducteurs. LFS possède deux types de transducteurs, les «normaux» qui indexent des fichiers entiers, et qui extraient en général peu de propriétés par fichier, et les «avancés» qui indexent des parties de fichiers. Dans le dernier cas, cela implique aussi peu de propriétés par partie, mais du fait qu'un fichier possède beaucoup de parties cela peut impliquer au total un grand nombre de propriétés. Par exemple, dans le prototype de LFS les parties sont des lignes, et il est très fréquent qu'un fichier dépasse le millier de lignes, et donc le millier de parties.

Ainsi, l'indexation d'un fichier, ainsi que sa réindexation suite à une modification du même fichier, par un transducteur dit «normal» n'est pas très coûteuse. Cela implique juste l'accès à une entrée de la table `obj->props`, pour mettre à jour la description d du fichier, et quelques accès dans sa table inverse `prop->objs`.

Il n'en est pas de même pour les transducteurs avancés. Après un `cd parts` LFS manipule encore des objets, qui cette fois ne font plus référence à des fichiers sur le disque, mais à des parties d'un fichier. On utilisera donc encore des tables du même genre que `obj->props` et `prop->objs` pour représenter ce nouveau genre de contexte. Le problème est qu'avec des milliers de parties, et donc des milliers d'objets, une réindexation naïve d'une fichier suite à la modification d'une vue impliquerait cette fois des milliers d'accès aux tables `obj->props` et `prop->objs`.

Les sections suivantes présentent des solutions au problème du coût de la réindexation ainsi qu'aux autres aspects du problème de la gestion des vues et des transducteurs avancés.

Structure de données pour la gestion des vues

Les structures de données utilisées par LFS pour manipuler les parties des fichiers ne sont pas très différentes de celles utilisées pour manipuler les fichiers. Cela est normal puisque les services fournis par LFS sont les mêmes dans les deux cas. Comme on l'a vu précédemment, pour représenter la matrice *objet* $\times$ *propriété*, on utilise deux tables `obj->props` et `prop->objs`. Un objet peut représenter un fichier ou une partie de fichier. Pour des raisons de clarté d'implémentation, ces deux types d'objets sont en fait séparés. Ainsi, LFS maintient deux tables «globales» `obj->props` et `prop->objs`, correspondants à la matrice *fichier* $\times$ *propriété*, et deux autres tables «locales» `obj->props` et `prop->objs` correspondants à la matrice *ligne* $\times$ *propriété*. LFS indique aux algorithmes, comme ls, quel type de table utiliser, les globales ou les locales, en fonction de l'endroit où est lancée la commande. Ainsi, après un `cd parts`, ls utilisera les tables locales. Dans ce dernier cas, l'*identifiant d'objet interne* o_i représente une partie. Dans la suite des explications nous considérerons que les parties sont des lignes. Ainsi chaque objet représentera une ligne du fichier. L'appel aux transducteurs avancés permettra d'initialiser ces deux tables, et allouera de nouveaux objets, un par ligne. Le cache logique n'est pas dédoublé puisque les deux mondes, celui des fichiers et celui des parties de fichier, utilisent la même logique.

En plus de ces tables, la gestion des vues demande la création de structures de données supplémentaires. Ainsi, LFS utilise aussi une table `obj->contents` qui associe à chaque objet son contenu, c'est à dire le contenu de la partie en question. LFS utilise aussi une table `line->obj` qui associe à chaque numéro de ligne du fichier d'origine, son identifiant d'objet interne correspondant. En effet, l'ordre des objets n'est pas forcément le même que l'ordre des lignes. Par exemple, si une nouvelle ligne est insérée dans une vue, un nouvel objet sera créé, et l'on ne veut pas décaler toutes les entrées de `obj->contents` ou `obj->props`. Ces deux tables, `obj->contents` et `line->obj`, permettent de reconstituer le contenu du fichier d'origine.

La Figure 3.8 illustre ces structures de données avec le fichier `foo.c` (voir Section 2.3.2 page 75), où nous supposons que la fonction `f2` fut créée la première, puis la fonction `f`, puis le message d'erreur (la ligne

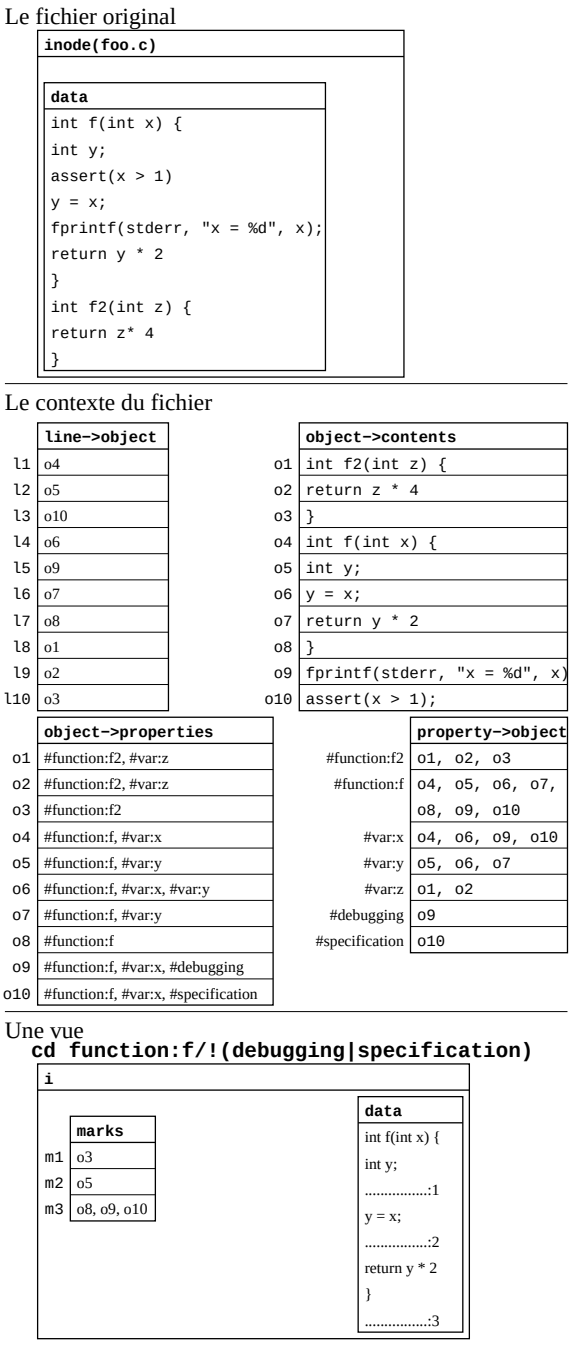

FIG. 3.8 – Structures de données pour la gestion des vues

avec le `fprintf`), et enfin l'assertion (la ligne avec le `assert`). Ce scénario explique la numérotation des objets du contexte.

Enfin, LFS stocke sur le disque le contenu des vues, sous la forme d'un fichier. Ces fichiers sont distingués des fichiers «normaux», comme les fichiers d'origine servant justement à calculer le contenu de ces vues, car les vues sont des fichiers temporaires. On associe à chacune de ces vues, un *identifiant de vue* v_i, qui permettra de localiser sur le disque le contenu de la vue. Chacune de ces vues contient comme on l'a vu Section 2.6.2 page 83 des marques spéciales, les *marques d'absence*, correspondants aux parties cachées du fichier d'origine. Pour être capable de propager les modifications d'une vue sur le fichier d'origine, il est nécessaire de pouvoir se rappeler à quoi correspondait une marque d'absence. Ainsi, LFS stocke en interne aussi sur le disque une table `marks` pour chaque vue. Cette table est indexée par un numéro de marque, et retourne un ensemble d'objets correspondant à l'ensemble des parties cachées par cette marque. L'identifiant de vue permet en interne aussi de localiser la table `marks` correspondant à cette vue.

Nous allons maintenant voir comment on utilise et modifie ces structures.

Le contenu des vues

Sous LFS, à chaque répertoire correspond une formule logique f. L'algorithme d'interrogation (ext(f)) calcule, en utilisant la table `prop->objs`, l'ensemble des objets `epwd` qui satisfont f. Chaque fois que l'utilisateur lit le contenu d'un nouveau répertoire, par exemple à l'aide de la commande `ls`, une nouvelle vue est créée. Concrètement, un nouvel identifiant de vue v_i, un nouveau fichier temporaire, et enfin une nouvelle table `marks` sont créés. Le contenu de ce fichier et de cette table sont déterminés par l'algorithme suivant :

```
compute_view(epwd) =
 mark   = 0
 marks  = empty_array
 inmark = false
 buffer = empty_string

 foreach l in line->obj
```

```
  let o = line->obj[l] in
  match(member(o, epwd), inmark) with
  (true, false) ->
    add obj->contents[o] to buffer;
  (true, true) ->
    inmark = false;
    add ".........:" and mark
      and obj->contents[o]
     to buffer;
  (false, true) ->
    add o to marks[mark]
  (false, false) ->
    mark++;
    inmark = true;
    marks[mark] = o;

 if inmark then
  add ".........:" and mark to buffer
 return (buffer, marks)
```

Indexation à la volée

Lorsque l'utilisateur modifie une vue, LFS doit tout d'abord mettre à jour le contenu du fichier d'origine. La fonction new_content permet de calculer ce nouveau contenu, en se servant du contenu de la vue modifiée `view_content`, ainsi que de la table des marques de cette vue `marks`. Le pseudo-code de cette fonction est présenté ci-dessous :

```
new_content(view_content, marks) =
 buffer = empty_string
 foreach line l in view_content
  if l contains a mark with number j
  then
   foreach o in marks[j]
    add obj->contents[o] to buffer
  else add l to buffer
 return buffer
```

La modification d'une vue entraîne aussi la mise à jour de la table `obj->props`, et donc aussi sa table inverse `prop->objs`, puisque les propriétés des lignes peuvent avoir changées. Comme nous l'avons dit au début de cette section, réindexer toutes les parties du fichier est coûteux.

Observation 1 : *un utilisateur modifie en général seulement quelques parties du fichier entre deux sauvegardes.*

L'action d'un utilisateur sur un fichier est donc souvent petite ; en général seulement quelques lignes du fichier sont affectées par cette action. Ainsi, en calculant la différence entre le contenu avant et après modification avec un algorithme à la `diff` [Mye86], on peut déterminer un ensemble beaucoup plus réduit de parties à réindexer. On va donc «*patcher*» les tables suivant les modifications de la vue. Ainsi, à chaque modification d'une vue, LFS changera uniquement quelques entrées dans la table `obj->props` (et `prop->objs`), au lieu de les effacer toutes du disque pour en recréer de nouvelles.

Le problème est que les transducteurs avancés peuvent posséder un *état*. C'est ce qui leur permet de pouvoir analyser des fichiers structurés comme les sources d'un programme. Cet état fait que la modification d'une ligne peut avoir un effet sur les propriétés des autres lignes, même si le contenu de celles-ci ne change pas. Par exemple, si l'utilisateur modifiait le nom de la fonction `f` à la ligne 1, cela modifierait aussi la description des lignes suivantes. Ainsi, limiter seulement la réindexation aux parties dont le contenu a été modifié ne serait pas correct. Même en sachant que les modifications sont petites, on devrait tout de même appliquer les transducteurs avancés sur l'intégralité du fichier d'origine et réindexer toutes les parties. Nous proposons un schéma dans lequel ces transducteurs avancés fournissent plus d'information à LFS sur le format du fichier afin d'éviter ensuite cet réindexation complète.

Observation 2 : *un fichier est souvent composé d'un ensemble d'unités relativement indépendantes dont les propriétés ne dépendent pas les unes des autres ; par exemple les fonctions dans un fichier C ou des entrées BibTeX.*

Dans beaucoup de cas, la modification d'une ligne n'a donc une influence que sur les propriétés d'une petite *zone* du fichier. Dans l'exemple précédent, la modification de `f` n'a pas d'impact sur les propriétés des lignes de la fonction `f2`. Ainsi, en calculant la différence entre le contenu avant et après modification, et en s'appuyant sur des indications de «*zone d'influence*» fournies par les transducteurs, on peut déterminer un ensemble plus réduit de parties à réindexer.

On peut voir le programme des transducteurs avancés comme une *machine à état*. À certains points du fichier, un transducteur peut revenir à son *état initial*. Par exemple, pour le fichier `foo.c`, chaque ligne introduisant une nouvelle fonction réinitialise l'état (chaque ligne introduisant une nouvelle «unité»). Cela veut dire que le contenu des parties avant ce point n'influencent pas les propriétés des parties après ce point. LFS va enregistrer ces points spéciaux comme des *points de synchronisation*. Ces points permettent d'établir les «zones d'influence».

Concrètement, un transducteur peut indiquer à LFS quels sont ces points en attribuant à la ligne qui contient ce point la propriété spéciale `synchro`. LFS peut retrouver ensuite rapidement cette information en stockant dans une nouvelle table `synchro` l'ensemble des numéros de lignes contenant un point de synchronisation.

Ainsi, si un fichier est modifié, et si il doit être réindexé, il est suffisant de réindexer ce fichier en partant du dernier point de synchronisation précédent la modification (marqué (a) dans la Figure 3.9), jusqu'au prochain point de synchronisation suivant la modification (marqué (b) dans la Figure 3.9).

Il faut noter qu'on ne réindexe pas en partant uniquement de l'endroit de la modification. En effet, on doit partir du dernier point de synchronisation afin de placer le transducteur dans le bon état lorsqu'il arrivera sur l'endroit de la modification. De plus il se peut que les propriétés des lignes précédents la modification changent aussi.

La Figure 3.9 illustre un exemple de réindexation en indiquant en noir les endroits d'une modification, en vert les points de synchronisation, et en rouge les zones (ici la zone) qui nécessitent de rappeler les transducteurs et qui nécessitent donc une réindexation.

Nous allons maintenant décrire plus précisément l'algorithme de réindexation reindex. Cet algorithme prend en paramètre l'ancien contenu du fichier d'origine `old_content` ainsi que le nouveau `new_content` (calculé par la fonction précédente new_content()). Il calcule ensuite la différence entre ces deux contenus. Il applique ensuite les transducteurs avancés sur des segments de `new_content` délimités par les points de synchronisation qui entourent la zone modifiée. Il modifie ensuite les différentes tables responsables de la gestion des vues comme `line->obj` ou `obj->props`.

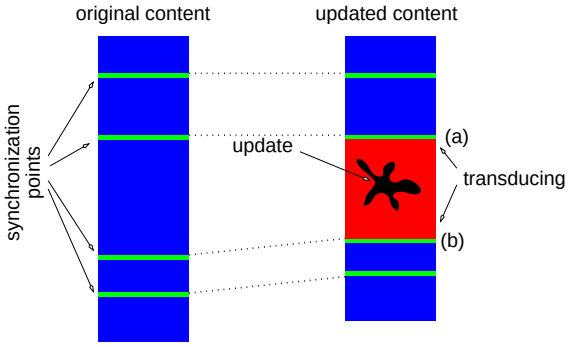

FIG. 3.9 – Réindexation entre points de synchronisation

Pour calculer la différence entre les deux contenus, LFS utilise la sortie de l'outil Unix `diff` [HM76]. Différentes actions sont ensuite exécutées en fonction du résultat de la comparaison des lignes des deux contenus. Dans l'algorithme qui suit, ces actions sont représentées par une fonction `f`, qui est passée en paramètre à `diff`, et qui est appelée après chaque comparaison de deux lignes. Cette fonction `f` prend en paramètre deux numéros de lignes (`newi` et `oldi` dans l'algorithme), ainsi que le résultat de la comparaison des contenus des lignes correspondantes (soit `Match`, soit `Old_not_in_New`, soit `New_not_in_Old`).

Le pseudo-code de reindex est présenté ci-dessous :

```
reindex(new_content, old_content) =

newline->obj = empty_array;
newsynchro   = empty_set;
must_index  = empty_list;
to_del     = empty_list;
is_in_clean = true;
may_index     = empty_list;

let f(newi, oldi, comparaison) =
 if comparaison = Match then
  if oldi is in synchro then
   is_in_clean = true;
   may_index = empty;
   add newi to newsynchro;
  if is_in_clean then
```

```
   newline->obj[newi] = line->obj[oldi];
   add newi to may_index;
  else
   add oldi to to_del;
   add newi to must_index;
 else   // not Match
  add may_index to must_index;
  is_in_clean = false;
  may_index = empty_list;
  if comparaison = New_not_in_Old
   then add newi to must_index;
  if comparaison = Old_not_in_New
   then add oldi to to_del;
in diff(new_content, old_content, f);

let (stdin, stdout) =
  pipe with transducer;

foreach i in must_index
  write to stdin new_contents[i];
foreach i in must_index
  let l = read_line from stdout in
  let props = parse l in
  let o = free_object() in
  foreach p in props
   add o in prop->objs[p];
  obj->props[o] = props;
  newline->obj[i] = o;
  obj->contents[o] = l;
  if 'synchro' is in props
    then add i to newsynchro;

foreach i in to_del
 let o = line->obj[i] in
 foreach p in obj->props[o]
   del o from prop->objs[p]
 free obj->props[o]
 free obj->contents[o]

line->obj = newline->obj
synchro = newsynchro
```

Condition : *le fait qu'une partie est un point de synchronisation ne doit pas dépendre des autres parties.*

Cette condition est très forte, mais elle rend possible l'algorithme de réindexation que nous venons de présenter. De plus, elle assure que l'effet d'une modification «ne traversera pas les compartiments étanches», un peu comme dans un sous-marin où une brèche d'un compartiment ne vient pas compromettre l'intégrité de l'ensemble du bateau. Dans notre cas, une parenthèse mal fermé dans la déclaration d'une fonction ne causera pas la réindexation coûteuse des autres fonctions.

Les différents algorithmes responsables de la gestion des vues sont aussi décrits en Annexe B.2.3 page 204.

Optimisation

Avec un gros fichier d'origine (comme un fichier BibTeX de 100 000 lignes), le temps de mise à jour suite à une petite modification sur une vue de ce fichier n'est plus alors dominé par l'appel au transducteur avancé et par l'indexation des propriétés mais par l'appel à la fonction new_content et le calcul du `diff` qui s'en suit. Cependant, comme la modification se fait sur la vue, l'utilisateur ne peut modifier que les parties du fichier d'origine rendues visibles par cette vue. Ainsi, toute les parties du document couvertes par une marque d'absence sont forcément les mêmes avant et après modification. En passant de la vue modifiée au nouveau contenu du fichier d'origine (avec new_content), l'algorithme reindex est plus simple car il n'a pas à se soucier lors du `diff` de la notion de marque d'absence (puisqu'il n'y en a plus), mais l'on a perdu de l'information qui aurait permis au `diff` d'être plus rapide.

Ainsi, afin d'être plus efficace l'algorithme reindex est en fait légèrement plus compliqué. Il n'applique pas `diff` entre l'intégralité de l'ancien et nouveau contenu du fichier d'origine mais uniquement entre l'ancien et nouveau contenu de la vue, qui a en général une taille bien plus petite. De plus, ce faisant on peut aussi économiser l'appel à new_content qui n'est plus nécessaire. En effet, la mise à jour des tables internes comme `obj->contents` est suffisante pour faire fonctionner les futures modifications des futures vues. On n'a pas besoin d'avoir un fichier d'origine à jour[7].

L'algorithme reindex est alors plus compliqué car il

[7] L'utilisateur aimerait cependant avoir un fichier d'origine à jour sur le disque, ce qui nécessite un appel à new_content ou une régénération du fichier d'origine à partir des tables internes `line->obj` et `obj->contents`. Cependant, on peut effectuer ce travail en tâche de fond et de manière paresseuse.

doit désormais prendre en compte la notion de marque d'absence puisque le `diff` est appliqué sur des vues. Une subtilité est qu'une marque d'absence peut cacher un point de synchronisation. En effet, la notion de marque d'absence n'a rien à voir avec la notion de point de synchronisation et une marque ne tombe donc pas forcément sur un point. Ainsi, la modification de la vue, qui concerne donc des parties du fichier d'origine rendues visibles par la vue, peut avoir comme impact la modification de la description de parties cachées par cette vue. L'algorithme doit donc parfois rentrer à l'intérieur de certaines marques d'absence et appeler ensuite le transducteur en lui passant aussi des parties cachées. Un autre difficulté est que la modification peut aussi concerner la marque d'absence elle-même, par exemple lorsque l'utilisateur déplace une marque ou l'efface. Ces cas particuliers compliqueraient énormément cette nouvelle version de l'algorithme reindex. Cependant, comme ces cas sont plus rares, on peut se permettre lorsqu'on les rencontre de repartir sur l'algorithme reindex de «base».

Synchroniser les vues

L'utilisateur est encouragé à ouvrir simultanément différentes vues du même fichier, à l'aide de différents répertoires, pour pouvoir travailler plus efficacement sur son fichier. Cela introduit un problème de concurrence. En effet, la mise à jour d'une vue modifie le fichier d'origine, et donc pourrait rendre invalide le contenu des autres vues. Nous avons choisi de déléguer la résolution de ces problèmes de concurrence aux applications qui utilisent LFS. Par exemple, un éditeur de texte comme Emacs envoie un message à l'utilisateur lorsque le contenu d'un fichier qu'il manipule vient d'être modifié par une autre application.

Cela requiert de la part du système d'exploitation un dispositif pour indiquer à une application que le fichier qu'il manipule (dans notre cas la vue) n'est pas assez «frais». Cela requiert aussi l'assurance que les applications vont utiliser et vérifier cette indication.

Pour ce faire, nous associons une *estampille* (*timestamp*) à chaque répertoire et nous gérons une estampille globale qui est incrémentée à chaque fois qu'une vue est modifiée. A chaque fois que l'utilisateur consulte une vue dans un répertoire, LFS vérifie que l'estampille de ce répertoire est égale à l'estampille globale. Si c'est le cas, la vue est donc assez «fraîche» et aucune action n'est requise. Dans le cas contraire, une nouvelle vue avec un nouvel identifiant est créée ; ce nouvel identifiant est lié à ce répertoire, et l'estampille de ce répertoire est mise à jour avec la valeur de l'estampille globale.

Cependant, certaines applications pourraient mal se comporter, par exemple en ne vérifiant pas si une vue est assez fraîche. LFS gère ces applications en interdisant simplement la mise à jour du fichier d'origine à l'aide d'une vue *périmée*. On obtient cela en associant aussi une estampille à chaque vue, et plus particulièrement à chaque identifiant de vue. Lorsqu'une application sauvegarde une vue, LFS vérifie que l'estampille de cette vue est égale à l'estampille globale. Si c'est le cas, l'estampille globale est incrémentée et les tables sont mises à jour comme décrit dans les sections précédentes. Dans le cas contraire, un code d'erreur est retourné à l'application.

C'est donc un système qui autorise des lecteurs multiples mais un écrivain unique.

Impacts sur les mécanismes traditionnels de gestion de fichier

Le mécanisme de gestion des vues de LFS a quelques conséquences inhabituelles sur les mécanismes traditionnels de gestion de fichier. Un des aspects «déroutant» de ces vues vient du fait qu'un utilisateur peut ajouter une ligne dans une vue où cette ligne ne devrait pas apparaître ; par exemple, ajouter un commentaire dans une vue qui réside dans un répertoire avec la propriété `!comment`. La sauvegarde d'une telle vue entraîne une réindexation et génère une vue avec un contenu mis à jour, qui ne contiendra donc pas cette nouvelle ligne (ici la ligne avec le commentaire) car elle n'appartient pas à cette vue ! Les applications, comme les éditeurs de texte, ne sont cependant pas toujours préparées au fait que le contenu d'un fichier après une action de sauvegarde peut différer du contenu juste avant la sauvegarde ; elles ne rafraîchissent pas le *buffer* de ce fichier avec le nouveau contenu. Cela veut dire que le commentaire restera dans cette vue, et qu'une deuxième opération de sauvegarde aura pour effet de ré-insérer cette ligne une deuxième fois dans le fichier d'origine. Pour éviter ce problème, ces applications doivent être configurées de tel façon que chaque sauvegarde implique aussi un rafraî-

chissement du contenu. Ces applications doivent relire la vue. Cette condition n'est pas excessive. Par exemple, il suffit sous Emacs de définir la macro suivante :

```
(global-set-key "^X^S"
  (lambda ()
    (interactive)
     (basic-save-buffer)
     (revert-buffer t t t)
))
```

3.2 Structures de données

Nous venons de voir les principaux algorithmes, et les principales structures de données LFS qui leur sont associées : le *cache logique*, la *table des extensions*, et la *table des points de synchronisation*. Dans cette section nous allons présenter l'ensemble complet des structures de données utilisées par LFS. En effet, nous nous sommes jusque là concentré principalement sur les structures rendant efficace la lecture d'un répertoire et la modification d'une vue. Il existe sous LFS d'autres opérations, comme par exemple la création ou destruction de fichiers, et donc d'autres structures de données, qui rendent possibles ces opérations. De plus, les structures précédentes n'étaient pas spécifiques aux systèmes de fichier. Elles pourraient tout aussi bien servir à implémenter les idées de LFS sous la forme d'une application classique. Ainsi, dans cette section nous présenterons aussi des structures de données supplémentaires spécifiques, comme par exemple des ajustements apportés à la notion d'*inode*, qui font de LFS un vrai système de fichier Unix.

3.2.1 Structures générales

Comme dit auparavant, les structures de données principales de LFS sont constituées de la table `prop->children` (le cache logique) et de la table `prop->objs` (la table des extensions), qui sont utilisées par l'algorithme ls. La table des extensions utilise une compression en intervalles pour représenter l'ensemble des objets. Ces entrées prennent ainsi moins de place sur le disque, ce qui améliore du coup aussi la vitesse pour lire ces entrées sur le disque. Les algorithmes de gestion du cache logique ainsi que l'algorithme ls n'ont besoin que de la partie `prop->children` du graphe pour fonctionner (les algorithmes de gestion du cache utilise aussi une table `prop->isformula` pour accélérer les insertions de propriétés, voir Section 3.1.1 page 93). Nous verrons plus loin que les autres opérations du système de fichier auront besoin aussi de la table inverse `prop->parents`. De même, certaines opérations auront besoin aussi de la table `obj->props` (la table des descriptions d).

Les structures de données internes de LFS utilisent des identifiants d'objets et de propriétés (o_i et p_i), qui sont en fait des entiers. C'est plus efficace à la fois en terme d'espace et de rapidité que l'utilisation de chaînes de caractères représentant le nom du fichier ou de la propriété. En effet, rien que pour la table des descriptions d (`obj->props`), on ne peut pas se permettre de dupliquer la chaîne de la propriété dans chaque entrée, surtout qu'une propriété est en général partagée par de nombreux objets.

Les tables précédentes, comme `obj->props`, sont donc indexées par un numéro d'identifiant. Il faut donc un moyen pour passer d'une chaîne à son identifiant et inversement. Ainsi, afin de retourner des noms de fichiers ou de sous-répertoires comme réponse à la commande `ls`, on introduit deux tables supplémentaires pour récupérer le nom d'un fichier ou d'une propriété en fonction de l'identifiant interne retourné par l'algorithme ls. Ces deux tables s'appellent `prop->propname` et `obj->filename`. De même, lorsque l'utilisateur lance par exemple la requête `cd USA/`, il faut pouvoir récupérer l'identifiant de propriété correspondant à USA pour pouvoir appeler ensuite les algorithmes comme ls qui eux manipulent des identifiants internes. On introduit donc la table `propname->prop`[8].

Afin de préserver la cohérence des extensions, l'exécution par exemple de la commande `touch b/file` doit en interne avoir comme effet non seulement d'ajouter l'objet `file` dans l'entrée `prop->objs[b]` (en fait ajouter l'identifiant interne de `file` dans l'entrée correspondant à l'identifiant interne de la propriété `b`), car `file` a la propriété `b`, mais aussi dans toutes les entrées des propriétés qui sont les parents de la propriété `b`

[8] Cette table joue aussi le role de table de *hash* sur les noms des propriétés pour savoir rapidement lors de la mention d'une propriété si il faut insérer cette propriété et donc déclencher insert_property.

de manière récursive. En effet, l'entrée d'une propriété dans la table des extensions contient aussi les objets de ses sous-propriétés du fait de l'*inlining*. C'est une des choses qui rend l'algorithme ls rapide. Il faut donc pouvoir récupérer les ancêtres d'une propriété, d'où le besoin de la table `prop->parents`. De manière similaire, l'exécution de la commande `rm file` doit en interne mettre à jour les extensions de toutes les propriétés que possèdent ce fichier, du fait que leurs extensions se verront amputées désormais d'un élément. De même, la modification du contenu du fichier peut avoir les mêmes conséquences puisque le fichier peut perdre certaines propriétés. Il faut donc pouvoir récupérer la description d'un fichier, d'où le besoin de la table `obj->props`, la table des descriptions appelée $d(o)$ dans les spécifications. Chaque valeur de cette table est décomposée en deux parties : une partie pour les propriétés extrinsèques et l'autre pour les intrinsèques.

Toutes ces tables sont présentées dans la Figure 3.10. Ces tables sont des *tables d'association* clef-valeur. Les clefs sont soit des identifiants internes, c'est à dire des entiers, soit des chaînes. Les valeurs sont elles aussi soit des identifiants ou ensemble d'identifiants, soit des chaînes. Les clefs et les valeurs sont donc souvent de taille variable. On ne peut donc pas utiliser simplement un tableau, avec des entrées de taille fixe, pour les représenter. Elles sont donc stockées sur le disque sous la forme de *B-tree* [Com79], ce qui rend l'accès associatif rapide.

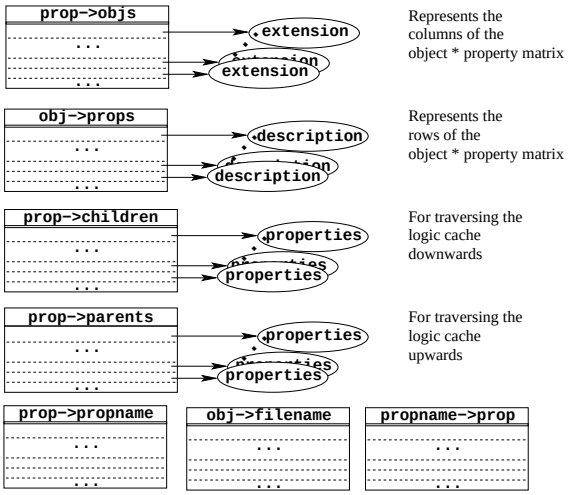

FIG. 3.10 – Les tables principales

3.2.2 Structures Unix

Nous avons jusque là parlé de tables contenant des identifiants, qui sont juste des entiers. Ces tables sont stockées sur le disque. Un système de fichier ne manipule cependant pas des objets «abstraits» mais des données «concrètes» : les fichiers. Ainsi, en plus de ces tables LFS stocke bien sûr aussi sur le disque le contenu des fichiers. On peut utiliser les mêmes techniques de stockage que celles employées dans les systèmes de fichier hiérarchiques, par exemple avec l'utilisation d'adressage direct et indirect de blocs (voir Section 1.6.1 page 38) pour pouvoir accéder ensuite rapidement au contenu d'un fichier.

De plus, même si sous un *shell* l'utilisateur désigne un fichier par son nom, le système d'exploitation Unix et notamment le VFS (voir Section 1.6.2 page 50) utilise un identifiant interne : *le numéro d'inode*. Les systèmes de fichier traditionnels stockent ainsi sur le disque une *table des inodes* `inode_table` (voir Section 1.6.1 page 38), indexée par un numéro d'*inode*, et où chaque entrée (c'est à dire l'*inode*) contient des informations de contrôle pour gérer le fichier ou répertoire correspondant : ses permissions, sa taille, ses adresses de blocs, etc. LFS n'échappe pas à cette règle et utilise donc aussi une table des *inodes*.

Le code de LFS s'interfaçant avec le VFS, les opérations LFS prennent en paramètre non pas des chemins mais des *inodes*. L'*inode* peut servir à désigner un fichier, un répertoire, ou bien encore une vue. Par exemple, l'opération LFS correspondant à la commande `ls` prend en paramètre un *inode* désignant un répertoire. Nous avons vu que LFS utilise déjà des identifiants internes d'objets et de propriétés (o_i et p_i) dans ses algorithmes. Ces identifiants servent comme clef ou comme valeur dans les tables. Pour les *inodes* de fichier, il est facile de faire correspondre un numéro d'*inode* à un numéro d'identifiant interne d'objet. Pour les répertoires la situation est cependant différente. Contrairement aux fichiers, il n'y a pas de correspondance directe entre un répertoire et un identifiant de propriété. En effet un répertoire désigne une formule composée d'un ensemble de propriétés. Sous un système de fichier traditionnel la taille de

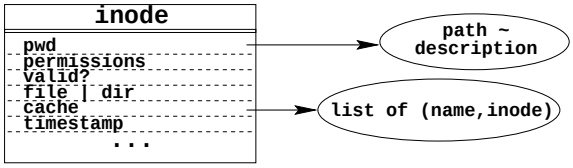

FIG. 3.11 – Un *inode* de répertoire LFS

la table des *inodes* est fixe, car il y a un nombre statiquement borné de fichiers et de répertoires. Sous LFS, le nombre des répertoires possibles est tellement grand qu'il n'est pas possible de calculer le contenu de ces répertoires statiquement. Ces répertoires, et donc les *inodes* correspondant, sont créés *à la demande*. Lorsqu'un utilisateur rentre dans un nouveau répertoire, un nouvel *inode* est alloué.

Classiquement l'*inode* d'un répertoire contient juste des informations sous la forme d'adresses de blocs pour retrouver sur le disque le contenu de ce répertoire. Lorsqu'un utilisateur modifie un répertoire, il modifie son contenu. Sous LFS la modification d'un répertoire peut avoir pour conséquence la modification d'un autre répertoire, comme par effets de bord. Il faut donc parfois recalculer le contenu d'un répertoire, et pour cela il faut savoir ce que l'on doit recalculer. Chaque *inode* de répertoire contient donc entre autre comme information supplémentaire le `PWD` de ce répertoire. La Figure 3.11 présente la structure d'un *inode* répertoire sous LFS avec l'ensemble des ajustements apportés, sous la forme de nouveaux champs, à un *inode* de répertoire classique, pour faire fonctionner les algorithmes LFS.

L'algorithme `ls` a besoin comme point de départ de la formule, le `PWD`, correspondant à un répertoire. Un numéro d'*inode* de répertoire ne lui est d'aucune utilité. Ainsi, nous avons ajouté un champ `pwd` dans l'*inode* d'un répertoire, Ce champ pointe sur une liste. Chaque élément de cette liste contient soit un entier, représentant l'identifiant interne d'une propriété, soit un *tag* `Or` avec une liste d'identifiants internes de propriétés (les opérandes de la disjonction), soit le *tag* `Not` associé à un numéro d'identifiant interne de propriété. La liste joue le rôle de la conjonction.

Les réponses de `ls` (c'est à dire le contenu des répertoires) sont placées dans un *cache*. Les *inodes* de répertoires contiennent donc des adresses de blocs où sont stockés le résultat calculé par l'algorithme `ls`, mimant ainsi le contenu d'un répertoire sous un système de fichier hiérarchique. Cependant, comme ce résultat doit être recalculé chaque fois que quelqu'un modifie le contenu de LFS (par exemple en ajoutant quelque part un fichier ou une propriété), l'*inode* d'un répertoire contient aussi une *estampille locale* indiquant le temps auquel ce résultat fut calculé. LFS maintient aussi une *estampille globale* qui est incrémentée chaque fois qu'un utilisateur modifie LFS, c'est à dire chaque fois qu'un utilisateur ajoute, enlève ou modifie un fichier ou une propriété. L'opération système appelée par la commande `ls` compare donc l'estampille global et la locale pour décider si il doit ou non recalculer les incréments et fichier appartenant au répertoire.

Tout ceci implique que à chaque fois que le contenu d'un répertoire change suite à la découverte de nouveaux incréments, de nouveaux *inodes* de répertoires sont alloués pour ces nouveaux sous-répertoires, et leurs estampilles sont positionnées à zéro (pour forcer le prochain appel de `ls` sur ces répertoires à calculer son contenu).

Pour finir, les *inodes* de vues ainsi que les vues sont eux aussi alloués et créés dynamiquement. Le contenu d'une vue est stocké sur le disque comme le contenu d'un fichier normal, mais dans une zone marquée comme temporaire, comme pour le contenu des répertoires. En effet, la place prise par ces données «dynamiques» peut être libéré suite à un *reboot* de la machine car plus aucun processus n'aura de référence sur ces données. Comme pour les *inodes* répertoires, les *inodes* des vues contiennent des informations supplémentaires pour faire fonctionner les algorithmes qui les utilisent. Chaque *inode* vue contient donc lui aussi un champ `pwd`, une estampille, ainsi que des adresses de blocs contenant les informations sur les marques d'absence.

3.3 Opérations concrètes

Nous basculons maintenant des commandes *shell* vers les véritables opérations d'un système de fichier pour rentrer plus dans les détails. Nous utiliserons la terminologie associée au VFS de Linux (voir Section 1.6.2 page 50).

Nous resterons cependant là encore, comme pour les structures de données, à un niveau de description d'as-

sez haut niveau. Nous allons abstraire la description des opérations afin de se concentrer sur les différences entre LFS et les systèmes de fichier hiérarchiques. Ainsi, nous ne parlerons pas de gestion de mémoire, de gestion de blocs, de concurrence et de synchronisation, ou de tolérance aux fautes, ce qui constitue traditionnellement la majeure partie de la description du codage d'un système de fichier. En fait, là encore comme pour les structures de données, la façon d'implémenter LFS fait que nous avons répondu à toutes ces questions en les évitant ou en réutilisant des techniques existantes.

Ainsi, pour la gestion de la mémoire nous nous basons sur un glaneur de cellule, pour le schéma d'allocation des blocs et leurs manipulations sur les services des systèmes de fichier hiérarchiques, pour l'indexation des tables sur une librairie de *B-tree*. Nous avons évité le problème de la concurrence en plaçant le code des opérations LFS dans des zones d'exclusion mutuelle. Une seule requête LFS est donc traitée à la fois. Pour plus d'informations, la Section 6.1 page 141 présente plus en détail l'architecture logicielle du prototype actuel LFS.

Un aspect important que nous n'avons pas évité est celui de la tolérance aux fautes. En effet, c'est l'une des fonctionnalités primordiales d'un système de fichier. Le système de fichier doit être exploitable même si la machine a subit une défaillance, par exemple un *reboot* brutal suite à une coupure d'électricité. Là encore nous avons réutilisé une technique existante : la *journalisation* (voir Section 1.6.1 page 43) qui permet d'implémenter la notion de *transaction*. Chaque fois que plusieurs tables doivent être mises à jour de manière atomique, une transaction est lancée, qui enregistre dans un *log* l'action de modification. En cas de panne, le prochain *reboot* relancera l'action (ou l'annulera) contenu dans le *log*. Par exemple, renommer une propriété a pour conséquence de modifier à la fois la table `prop->propname` et sa table inverse `propname->prop`. Ces actions doivent être effectuées dans une transaction pour s'assurer que les deux tables sont cohérentes entre elles. LFS est donc aussi un système de fichier journalisé.

Les opérations d'un système de fichier peuvent être divisées en 3 groupes : les opérations globales (*superblock operations*), les opérations sur répertoires (*inode operations*), et les opérations sur fichiers (*file operations*). Les opérations globales s'occupent de la gestion du système de fichier : (dé)montage et statistiques. Les opérations sur répertoires s'occupent de l'interrogation/navigation, et de la création/suppression de fichiers/propriétés. Enfin, les opérations sur fichiers s'occupent de la gestion du contenu des fichiers.

3.3.1 Opérations globales

`read_super` est appelée par le programme `mount`. Elle localise les différentes tables sur le disque, enregistre ces informations dans la structure du superbloc pour les utilisations futures, et enfin remplit la première entrée de la table des *inodes* qui correspond à l'*inode* de la racine (le `/`). Le champ `pwd` de cette entrée contient une liste d'un élément contenant l'identifiant interne de la propriété correspondant à la racine du cache logique, c'est à dire la propriété *vrai*. L'estampille locale de ce répertoire est fixée à la valeur 0 et l'estampille globale à la valeur 1.

`put_super` est appelée par le programme `umount`. Elle peut libérer les différentes structures temporaires utilisées par LFS comme les blocs utilisés par le contenu des vues ou les caches de répertoires.

3.3.2 Opérations sur répertoires

`readdir` est appelée par le programme `ls`. C'est cette fonction qui appellera l'algorithme `ls` décrit Section 3.1.2 page 98. Elle prend en paramètre l'*inode* d'un répertoire, et retourne une liste de paires `(nom, inode)` contenant une chaîne de caractère, correspondant à un nom de fichier ou de sous-répertoire, et son *inode* correspondant. Si l'estampille globale est égale à l'estampille locale de l'*inode*, la liste de paires est lue en se basant sur les adresses de blocs que contient cet *inode*. Dans le cas contraire, un nouveau calcul doit être lancé : l'algorithme `ls` est appliqué sur le champ `pwd` de l'*inode* passé en paramètre. Le résultat est une paire `Dirs` et `Files` contenant respectivement une liste d'identifiants internes de propriété (les incréments), et une liste d'identifiants internes d'objet (les fichiers). Ensuite, un nouveau *buffer* est alloué, et pour chaque `oi` de `Files` la paire `(obj->filename[oi], oi)` est stockée dans ce *buffer*. Pour chaque `pi` de `Dirs`, un nouveau numéro d'*inode* `ino` est alloué et la paire `(prop->propname[pi], ino)` est stockée dans le

buffer. De plus, pour chacun de ces nouveaux *inodes*, le champ `pwd` est rempli avec comme formule la conjonction de `pi` et du `PWD` courant, et son estampille est fixée à la valeur 0. Enfin, les blocs utilisés pour contenir le contenu du répertoire sont libérés ; de nouveaux blocs sont alloués et remplis avec le contenu du *buffer*. L'estampille de l'*inode* passé en paramètre est fixée à la valeur de l'estampille globale, car son contenu est désormais à jour, et la liste des paires contenues dans le *buffer* est retournée.

`lookup` peut être appelée via la commande `cd` f. C'est principalement cette fonction qui appellera l'algorithme insert_property décrit Section 3.1.1 page 92 et donc aussi les moteurs de déduction logique. En effet, la requête de l'utilisateur peut contenir des formules comme `size:>100` qu'il convient d'insérer au bon endroit dans le cache logique. Elle prend en paramètre l'*inode* d'un répertoire et une chaîne de caractères `str`, correspondant à un nom d'un fichier ou à une formule, et retourne l'*inode* correspondant ou un code d'erreur. L'opération `readdir` est tout d'abord appelée pour regarder si `str` est dans la liste des paires du répertoire, auquel cas l'*inode* correspondant est retourné. Dans le cas contraire, la chaîne doit correspondre soit à une propriété non listée comme incrément de ce répertoire, soit à une formule. Ainsi, si la chaîne a la forme d'une propriété unique, l'entrée `propname->prop[str]` retourne l'identifiant interne `pi` de la propriété correspondante. Si cette entrée est vide, et que la propriété a la forme d'une propriété atomique, c'est que l'utilisateur a mentionné une propriété qui n'existe pas, auquel cas un code d'erreur est retourné. Cependant, si la propriété a la forme d'un attribut valué, c'est que peut-être cette propriété n'existe pas encore mais qu'il faut la créer, car cette propriété correspond à une formule comme par exemple `size:>100`. Il faut donc vérifier d'abord que l'attribut correspondant existe, localiser ensuite le moteur de déduction logique approprié, et appeler l'algorithme insert_property avec en paramètre la formule `str` et le moteur logique. L'algorithme retourne alors, soit un identifiant de propriété `pi` correspondant à la propriété qui vient juste d'être créée, soit un code d'erreur car la propriété n'a pas la bonne syntaxe. À ce stade, en cas de succès, on a donc récupéré un identifiant de propriété `pi` (soit d'une propriété classique existante, soit d'une propriété qui vient juste d'être créée). Un nouvel *inode* `ino` est alors alloué et son champ `pwd` est rempli avec la conjonction de `pi` et du `PWD` du répertoire courant. Cet *inode* est alors retourné et l'opération `lookup` se termine. Si la chaîne `str` avait la forme d'une disjonction `x|y|z|...`, les noms de propriétés sont traduits en identifiants internes `xi`, `yi`, `zi`, selon le même schéma que précédemment. Si une seule de ces propriété n'a pas d'identifiant de propriété correspondant, alors un code d'erreur est retourné. Dans le cas contraire, un nouvel *inode* est alloué et son champ `pwd` est cette fois rempli avec la conjonction du `PWD` du répertoire courant et la disjonction (avec le *Tag* `Or`) des identifiants de propriétés `xi`, `yi`, `zi`. Si la chaîne avait la forme d'une négation `!x`, l'opération est alors similaire, mais plaçant cette fois dans le champ `pwd` le *Tag* `Not`.

`create` est appelée par exemple par la commande `touch`. Elle prend en paramètre l'*inode* d'un répertoire et une chaîne `str` représentant le nom d'un fichier, et retourne l'*inode* correspondant au fichier qui sera crée. Le `pwd` du répertoire doit correspondre à une conjonction `ps` de propriétés (du fait de la restriction apportée à la description d'un objet, voir Section 2.4 page 79). Cela est vérifié en premier. Puis, une vérification similaire est effectuée sur la chaîne `str` qui ne doit pas contenir les symboles réservés `|`, `&` ou `!`, sous peine de rentrer en conflit avec des noms possibles de requêtes. Si l'une de ces vérifications échoue, un code d'erreur est retourné. Dans le cas contraire, un nouvel identifiant interne d'objet `oi` est alloué, et `str` est ajouté dans l'entrée `object->filename[oi]`. La partie extrinsèque de l'entrée `obj->prop[oi]` contient la liste des propriétés `ps`, et pour chaque propriété `pi` de la liste `ps`, l'identifiant `oi` est ajouté dans la liste des objets de l'entrée `prop->obj[pi]`. En effet, l'extension de la propriété `pi` contient désormais un nouveau fichier. Cette dernière action est répétée récursivement pour toutes les propriétés ancêtres des propriétés `ps` dans le cache logique, en parcourant donc le cache logique vers le haut grâce à la table `prop->parents`, du fait du besoin d'*inlining*. Enfin, l'identifiant `oi` qui représente aussi le numéro d'*inode* du fichier est retourné.

mkdir sert à créer des axiomes. Elle prend en paramètre un *inode* de répertoire et une chaîne de caractère `str` correspondant au nom du répertoire (sous LFS la propriété) à créer. Elle alloue ainsi un nouvel identifiant de propriété `pi`, puis ajoute `pi` dans l'entrée `propname->prop[str]` et `str` dans l'entrée `prop->propname[pi]`. Elle positionne à vide les entrées `prop->children[pi]` et `prop->obj[pi]` puisque cette propriété n'a pas encore de sous-propriétés ni d'objets. Enfin, pour chaque propriété `p` composant le `PWD` de l'*inode* du répertoire passé en paramètre, elle ajoute `pi` dans l'entrée `prop->children[p]` et `p` dans l'entrée `prop->parents[pi]`. Elle ajuste donc le cache logique avec ces nouveaux axiomes.

unlink défait ce qui a été fait dans `create`.

rmdir se comporte de la même façon que `unlink` mais cette fois pour les propriétés.

3.3.3 Opérations sur fichiers

Les opérations sur fichiers sont `open`, `read`, `write`, `release`, `lseek` et `truncate`. Pour bien comprendre leur sens, on peut décomposer l'action d'édition d'un fichier par un utilisateur suite par exemple à la commande `emacs toto.c`. Cette commande implique en interne l'exécution de l'opération `open` suivie d'une suite de `read` ayant pour but de récupérer le contenu du fichier `toto.c` pour le placer en mémoire. Chaque opération `read` lit un morceau du fichier. La modification sous Emacs de ce fichier se fera tout d'abord en mémoire. Enfin, suite à une demande de sauvegarde, le contenu mémoire sera replacé sur le disque avec une suite d'opérations `write` suivie de l'opération de fermeture `release`. L'opération `lseek` est plus rarement utilisée. Elle sert à se déplacer rapidement dans un fichier, par exemple à la fin du fichier pour rendre rapide l'exécution de commandes comme `tail`. L'opération `truncate` sert à tronquer la fin d'un fichier, c'est donc une forme spécialisée de `write`.

Puisque sous LFS le contenu des fichiers (et des vues) est stocké sur le disque de la même manière que sous un système de fichier hiérarchique, les opérations `lseek`, `read`, `write`, `truncate` sont les mêmes sous LFS. Par exemple, l'opération `read` prend en paramètre l'*inode* d'un fichier (ou d'une vue), et l'adresse d'un *buffer* à remplir avec le contenu de ce fichier (ou de cette vue). Elle doit localiser sur le disque ces données en se basant sur les adresses de blocs contenues dans l'*inode*, et remplir ensuite de manière appropriée le *buffer* passé en paramètre.

L'originalité de LFS ne réside donc pas dans les opérations de lecture ou d'écritures de fichiers, mais dans les indexations ou réindexations qui s'en suivent. L'originalité réside dans les appels aux transducteurs, normaux et avancés. Ces transducteurs sont appelés après chaque modification d'un fichier ou d'une vue. En effet, lorsque le contenu change, ses propriétés changent aussi. Cette réindexation n'est faite qu'à la suite d'un appel système de fermeture, dans l'opération `release`. Elle n'est pas faite après un appel système d'écriture. En effet, appeler les transducteurs est coûteux ; nous limitons donc la réindexation à l'opération essentielle.

release est donc appelée suite à la fermeture d'un fichier, ce qui couvre notamment la sauvegarde d'un fichier[9]. C'est donc cette opération qui appellera les transducteurs normaux, ainsi que les transducteurs avancés et l'algorithme reindex décrit Section 3.1.3 page 102 lorsque l'opération concerne une vue. C'est cette opération qui permettra de propager la modification d'une vue sur le fichier d'origine. Elle prend en paramètre l'*inode* d'un fichier ou d'une vue et peut retourner un code d'erreur, par exemple si la vue n'était pas assez fraîche. Lorsque l'*inode* concerne un fichier normal, il faut juste mettre à jour la description intrinsèque du fichier. Comme dit Section 2.4.2 page 80, la description d'un objet est divisée en deux parties : une partie extrinsèque et une partie intrinsèque. Ainsi, chaque entrée de la table `obj->props` est aussi décomposée en deux parties. L'opération `release` appelle donc les transducteurs appropriés avec comme paramètre le contenu du fichier, récupère les propriétés et met à jour ensuite la partie intrinsèque, sans toucher à la partie extrin-

[9] En effet, même si pour l'éditeur de texte et l'utilisateur le fichier paraît encore ouvert suite à une sauvegarde, car l'utilisateur n'a pas fermé l'application, d'un point de vue système la sauvegarde d'un fichier est perçue comme une fermeture. La prochaine action de l'utilisateur sur le fichier impliquera une réouverture système et une refermeture.

sèque qui elle n'a pas changée. Elle doit mettre à jour aussi la table inverse `prop->objs` tout comme dans l'opération `create`. Si l'opération concerne une vue, `release` regarde d'abord l'estampille de l'*inode* pour déterminer si la vue est assez fraîche pour pouvoir mettre à jour le fichier d'origine. Si ce n'est pas le cas, un code d'erreur est retourné. Du point de vue de l'application cela se traduira par un message d'erreur lors de l'opération de sauvegarde. Si la vue est assez fraîche, `release` appelle l'algorithme new_content avec le contenu de la vue ainsi que la table de ses marques d'absence, puis l'algorithme reindex, tous deux décrit Section 3.1.3 page 102. Elle incrémente ensuite l'estampille globale car le contenu des répertoires doit être recalculé puisque les propriétés peuvent avoir changées.

open est appelée juste avant de lire le contenu d'un fichier. C'est un passage obligé mais cette opération ne fait traditionnellement quasiment rien. Sous LFS, si le fichier en question est une vue, elle calcule le contenu de cette vue et appelle donc l'algorithme compute_view décrit Section 3.1.3 page 101. L'opération `readdir` pourrait se charger de ce calcul, mais il est préférable d'agir de manière paresseuse en attendant le moment où le contenu devient effectivement utile, c'est à dire jusqu'à l'opération `open`. En effet, un utilisateur peut inférer, en se basant sur les noms sous-répertoires, qu'il a encore besoin de naviguer un pas de plus, sans pour cela avoir eu besoin de lire le contenu de la vue de ce répertoire. C'est pour cette raison que l'*inode* d'une vue contient aussi un champ pwd, qui est passé en paramètre de la fonction compute_view. L'opération `readdir` se charge donc juste d'allouer un nouvel *inode* pour cette vue, et de placer la bonne valeur dans le champ pwd.

3.4 Conclusion

Les fonctionnalités offertes par LFS impliquent des calculs qui peuvent paraître très coûteux du fait de l'utilisation à outrance de la logique $\models$ notamment via des moteurs de déduction logique externes, de l'utilisation de transducteurs, et du fait que ces calculs peuvent impliquer un grand nombre d'objets (fichiers ou parties de fichier) et donc aussi un grand nombre de propriétés.

Comme solution à ces problèmes nous avons proposé principalement 3 structures de données : le *cache logique*, la *table des extensions* (avec l'*inlining* et une représentation des ensembles sous forme d'*intervalles*), et la table des *points de synchronisation*. le cache logique mais aussi la *spécialisation* de la *logique du noyau* à une logique des propositions règlent le problème du coût logique, l'indexation des propriétés via la table des extensions et la représentantion en intervalles règlent le coût de la manipulation d'un grand nombre d'objets, et enfin les points de synchronisation règlent le coût des transducteurs et de l'indexation associée. Nous avons aussi décrit les algorithmes exploitant à bon escient ces structures comme insert_property, ls, et reindex.

Nous n'avons pas parlé dans ce chapitre des complexités théoriques des différents algorithmes LFS, car ces complexités sont difficiles à établir du fait que les opérations ne dépendent pas uniquement du nombre d'objets mais aussi du type des propriétés et de bien d'autres facteurs. Il est cependant important de savoir si LFS «passe à l'échelle» car les ambitions LFS sont importantes et le contexte des systèmes de fichier est un contexte exigeant : il n'est pas rare pour un utilisateur de posséder 100 000 fichiers, et l'utilisateur est amené à utiliser des commandes comme `ls` et à éditer des fichiers sans arrêt. Dans un tel contexte interactif, le temps de réponse des différentes opérations doivent être de l'ordre de la seconde. Pour répondre à ces inquiétudes, la Section 6.3 présente des benchmaks réalisés avec un prototype LFS démontrant la viabilité des algorithmes et structures de données décrits dans ce chapitre, avec notamment la Section 6.3.5 qui montre que LFS n'explose pas en complexité.

La Section 6.1 complète les explications de ce chapitre sur l'implémentation de LFS en présentant l'architecture logicielle du prototype LFS. L'Annexe B décrit l'un des composants de cette architecture, le composant principal, celui qui contient les algorithmes et structures de données principaux décrits Section 3.1 et Section 3.2.1. Seul manque les structures Unix et les spécificités d'interfacage au VFS décrites Section 3.2.2 et Section 3.3 qui ont été séparées du composant principal pour des raisons de génie logiciel.

Chapitre 4

Sécurité

Hacking is like sex. You get in, you get out, and hope that you didn't leave something that can be traced back to you.

Anonymous

Le rôle d'un système de fichier est bien sûr de stocker des données, mais aussi de les sécuriser puisque le système peut être utilisé par de multiples utilisateurs. Un *modèle de sécurité* doit fournir des moyens pour *protéger* l'information, mais aussi pour *partager* l'information. En effet, un modèle trop restrictif interdirait tout travail coopératif.

Nous avons déjà vu Section 1.5 page 30 deux modèles de sécurité pour les systèmes de fichier, celui d'Unix avec les *groupes* et celui des *liste de contrôle d'accès* (ACL). Cependant, du fait de la différence très importante entre LFS et les systèmes de fichier classiques, on ne peut pas ré-appliquer tel quel ce genre de modèle pour LFS.

De plus, ces modèles furent inventés en un temps, celui des systèmes de fichiers hiérarchiques, où le pouvoir d'organisation du répertoire était pauvre, et où il fallut donc étendre de manière importante le système pour gérer la sécurité. Il fallut par exemple rajouter aux fichiers des métas-données où seraient stockées les propriétés de sécurité. L'amélioration des moyens d'organisation sous LFS fournit donc l'occasion de revisiter ces anciens modèles sous un jour nouveau. Le modèle de sécurité pour LFS que nous allons présenter dans ce chapitre s'intègre naturellement avec le reste du système. Nous n'aurons pas à étendre LFS, ou pas beaucoup, pour gérer la sécurité. Ceci est rendu possible grâce aux fondations logiques de LFS : les *règles de sécurité* peuvent être modélisés par des *règles logiques*, et le *statut de protection* de l'information peut être modélisé par une *propriété*.

Traditionnellement, le modèle de sécurité est le même pour les fichiers et les répertoires. Cela paraît logique puisque sous les systèmes de fichiers hiérarchiques le répertoire n'est qu'une forme de fichier qui possède lui aussi un contenu statique. Ce n'est plus le cas sous LFS. Ainsi, sous LFS les mécanismes de sécurité pour un fichier et un répertoire sont différents. Nous allons tout d'abord décrire le modèle de sécurité pour les fichiers, qui sera proche du modèle des ACL. Nous verrons ensuite un modèle de sécurité pour les répertoires, qui sera proche du modèle Unix. Enfin, nous résumerons la sémantique du point de vue sécurité des différentes commandes *shell* vis à vis de ces deux modèles.

4.1 Les fichiers : un modèle à la ACL

Sous un système de fichier traditionnel, un fichier possède un chemin, le répertoire dans lequel il réside, et des propriétés spéciales stockées dans l'*inode* de ce fichier, avec notamment le *propriétaire* du fichier et des *droits*. Comme LFS sait manipuler depuis le début des propriétés, il est légitime de modéliser aussi les propriétés liées à la sécurité sous la forme de propriétés gérées par LFS. Cela n'est cependant pas suffisant pour produire un système de sécurité. Pour vérifier ensuite si un utilisateur a le droit de lire ou de modifier un fichier, il faudra vérifier comme sous les autres systèmes que les propriétés du fichier autorisent cette action. Cependant, comme nous

allons le voir, LFS fournira là aussi gratuitement la machinerie pour réaliser cette vérification.

Nous allons présenter trois modèles de sécurité d'expressivité croissante ce qui permettra de progresser doucement dans les explications. Cela permettra aussi de mieux apprécier les avantages du dernier modèle.

4.1.1 Un modèle plat

Un fichier possède classiquement 3 droits d'accès, les droits de *lecture* (`r`), d'*écriture* (`w`), et d'*exécution* (`x`). Le propriétaire du fichier peut ensuite varier ces droits en fonction des utilisateurs. On peut donc spécifier des droits différents pour le *propriétaire*, pour le *groupe*, et pour les *autres* utilisateurs. Comme on l'a vu Section 1.5.2 page 33, les ACL permettent de raffiner ce modèle en permettant même de donner des droits spécifiques en fonction d'un seul utilisateur.

Ainsi, sous LFS, pour que le propriétaire `bar` d'un fichier permette à un utilisateur `foo` de lire son fichier, il suffit que `bar` ajoute simplement la propriété spéciale `user:foo_r` à son fichier, par exemple avec la commande :

```
mv fichier.txt user:foo_r/
```

Les propriétés de LFS permettent de modéliser les propriétés de sécurité. On peut aussi assigner les propriétés `user:foo_w` et `user:foo_x` à ce fichier. Comme on veut souvent rendre visible un fichier à de nombreux utilisateurs, il serait fastidieux et sujet à erreurs de lister manuellement ces utilisateurs à l'aide de ces propriétés. Ainsi, on peut réutiliser la notion de groupe et représenter les droits pour ces groupes par exemple à l'aide de la propriété spéciale `group:admin_r` et pour l'ensemble des utilisateurs avec la propriété `other:r`. Il faut noter qu'on peut ainsi donner différents droits selon différents groupes, alors qu'Unix ne permet que de spécifier les droits pour un seul groupe, celui du fichier.

La vérification d'accès à un fichier par un utilisateur peut être implémentée de manière traditionnelle, c'est à dire en parcourant la description du fichier à la recherche de certaines propriétés de sécurité, tout comme on parcourait une ACL dans un système de fichier classique. On peut aussi utiliser le mécanisme du calcul d'*extension* pour résoudre ce problème de vérification. En effet, pour savoir si un utilisateur `foo` du groupe `foogr` a le droit de lire un fichier, il suffit de vérifier si ce fichier fait partie de l'extension de la formule $user{:}foo\_r \vee group{:}foogr\_r \vee other{:}r$. Ainsi, les mécanismes de LFS offrent toute la machinerie, ici la gestion des disjonctions, le principe des extensions et la notion de propriété, pour implémenter facilement un système de sécurité.

Concrètement, il faut modifier la fonction `open` dans le VFS. Cette fonction prend en paramètre l'*inode* d'un fichier et un mode d'accès. En fonction de ce mode, la lecture seule, l'écriture ou l'exécution, il faut vérifier si ce fichier fait partie respectivement de l'extension de la formule mentionnant des `r` ($user{:}foo\_r \vee other{:}r \vee \ldots$), de la formule mentionnant des `w` ($user{:}foo\_w \vee other{:}w \vee \ldots$), ou de celle mentionnant des `x` ($user{:}foo\_x \vee other{:}x \vee \ldots$).

C'est donc une manière d'ajouter des permissions à la ACL dans un système de fichier, tout en utilisant des commandes traditionnelles, c'est à dire `mv` et les répertoires. L'utilisateur n'a ni à éditer des fichiers de permissions spéciaux, ni à utiliser des commandes spéciales. Bien sûr, cela n'est rendu possible que parce que ce système de fichier est LFS. L'utilisateur sous LFS peut en effet assigner de multiples propriétés à un fichier.

Un point que l'on n'a pas abordé est qu'il pourrait être cependant fastidieux à chaque création de fichier d'avoir à spécifier manuellement les propriétés de sécurité pour ce fichier. L'utilisation, par exemple, de la commande `mv` pour ajouter des droits différents concernant la lecture, l'écriture, l'exécution, selon le propriétaire, le groupe et les autres concernerait un nombre important de propriétés. Elle serait donc longue à spécifier. En fait, ce problème existe déjà sous les systèmes de fichier classiques, et l'utilisateur n'utilise pas la commande `chmod` à chaque création d'un fichier. En effet, le mécanisme lié à la commande `umask` permet à l'utilisateur de spécifier une valeur par défaut. On peut réutiliser ce mécanisme sous LFS, et tout comme on extrait automatiquement à la création d'un fichier les propriétés concernant le nom, l'extension, la date et la taille d'un fichier (`name:`, `ext:`, `date:` et `size:`), on peut extraire automatiquement à partir du mode de création par défaut les propriétés de lecture, d'exécution et d'écriture (`user:foo_r`, `other:w`, etc). Le propriétaire peut ensuite ajuster ce premier jet en utilisant la commande `mv`, tout comme un utilisateur dans un système de fichier classique le ferait avec la commande `chmod`.

4.1.2 Un modèle taxinomique

Nous allons dans cette section présenter un modèle de sécurité qui ne sera pas plus expressif que le précédent mais plus élégant. Ce système permettra de plus de transiter plus doucement vers le modèle final qui sera aussi élégant et cette fois plus expressif.

Dans le modèle Unix, il y a implicitement un *ordre* entre les différentes propriétés de sécurité et dans la vérification. Ainsi, lors d'un accès à un fichier le noyau commence par vérifier si l'utilisateur est le propriétaire de ce fichier, on regarde dans ce cas les droits `u`. Si le test échoue on vérifie si il fait parti du groupe de ce fichier, on regarde dans ce cas les droits `g`. Sinon on finit enfin par regarder les droits `o`. On peut utiliser le mécanisme des taxinomies offert par LFS pour représenter cet ordre. Ainsi, les propriétés de sécurité peuvent être ordonnées selon les axiomes $other \models group \models user$, et ceci à la fois pour les propriétés de lecture, d'écriture, et d'exécution. La Figure 4.1 représente un exemple d'une telle hiérarchie.

Avec cette taxinomie, l'accès en lecture, en écriture ou en exécution peut être vérifié désormais en regardant plus simplement si ce fichier fait parti de l'extension respectivement des propriétés `user-foo_r`, `user-foo_w` et `user-foo_x`. Là encore, les mécanismes de LFS offrent toute la machinerie pour modéliser les différentes règles d'un système de sécurité.

En effet, si un fichier a la propriété `other_r`, cela veut dire que désormais du fait des axiomes ce fichier est aussi automatiquement dans l'extension de tous les parents de `other_r` (voir Section 2.4.1 page 80), et qu'il est donc aussi dans l'extension de tous les répertoires de la forme `user-username_r`. En effet, un fichier ayant la propriété spécialisée `other_r` satisfera aussi la requête `cd user-foo_r/`, tout comme un fichier ayant la propriété spécialisée `movie` satisfera aussi la requête `cd art/`. Ainsi, avec ce schéma, tous les utilisateurs pourront lire ce fichier, ce qui est bien l'effet attendu par l'assignation de la propriété `other_r`, d'où l'ordre $other \models group \models user$ pour la taxinomie.

4.1.3 Un modèle logique

Le modèle taxinomique est plus élégant que le modèle plat mais il souffre des mêmes limitations. Par exemple, un utilisateur aimerait donner l'accès à un fichier à tous les membres d'un groupe *excepté* un utilisateur considéré comme mal intentionné, ce qui n'est pas très pratique à spécifier dans les deux modèles précédents. On aimerait donc exprimer la *négation*.

On pourrait demander à l'administrateur du système de créer un groupe spécial qui exclurait cet utilisateur. Ce n'est cependant pas toujours possible. En effet, l'administrateur peut refuser cette requête car elle pourrait ouvrir la porte à de plus en plus de requêtes de la part des utilisateurs[1]. En fait, c'est le même genre de problème qui a conduit à la création des ACL du fait de la limitation des groupes Unix. Cette fois on tombe sur une limitation des ACL qui ne peuvent représenter des propriétés de sécurité que de manière *positive* ; la négation n'est pas gérée.

De plus, le contexte LFS est encore plus exigeant et rend encore moins souhaitable les requêtes d'un utilisateur pour créer un nouveau groupe. En effet, l'ajout d'un groupe entraîne aussi la modification de la taxinomie, ce qui requiert là aussi une intervention de l'administrateur système. En fait, l'ajout d'un utilisateur ou d'un groupe requiert d'ajuster 3 taxinomies, ce qui nécessite de l'attention de la part de l'administrateur pour ne pas rendre le système incohérent. Par exemple, la propriété `other` doit toujours impliquer logiquement tous les groupes et tous les utilisateurs.

Enfin, ces 3 taxinomies séparées ne sont pas très pratiques puisqu'elles obligent l'utilisateur à répéter 3 fois le nom d'un utilisateur pour lui confier tous les droits d'accès sur un fichier, une fois pour le `r`, une pour le `w` et une pour le `x`. Ce processus est sujet à erreurs puisqu'on peut facilement écorcher l'un de ces trois noms.

En fait, le modèle précédent, qui repose sur les axiomes et la taxinomie, est trop *manuel*. C'est le point commun qui rend la négation, la création de nouveaux groupes, et l'attribution de propriétés fastidieuses. Il faudrait un modèle où les propriétés sont gérées *automatiquement* par des règles précises plutôt que par un administrateur. C'est justement ce que permettent les *moteurs de déduction logique* de LFS. La gestion de la sécurité sous LFS est donc principalement gérée par un moteur

[1] Et les administrateurs ont tendance à ne pas aimer travailler.

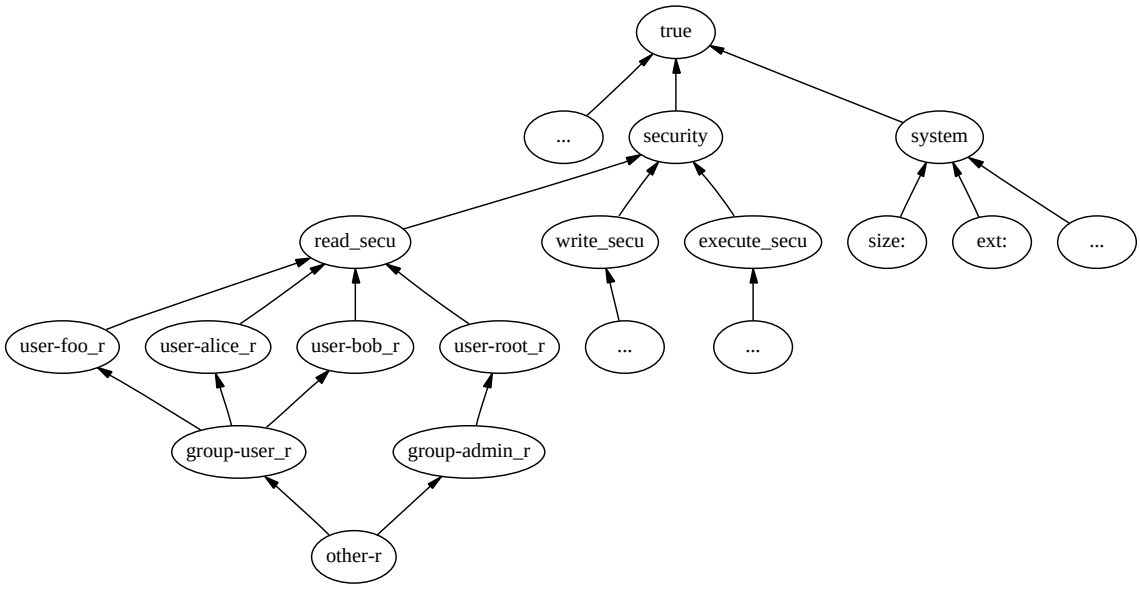

FIG. 4.1 – Taxinomie des propriétés de sécurité

de déduction spécialisé modélisant une *logique de sécurité* attaché à l'attribut `secu`:. Le fait de confier la gestion des propriétés à un programme augmente encore un peu plus la sécurité du système puisqu'un programme est plus fiable qu'un humain lorsqu'il s'agit de tâches mécaniques et automatiques.

L'une des premières limitations citées précédemment était l'incapacité de gérer la négation. On peut voir la négation comme une opération ensembliste. En effet, interdire l'accès à un fichier c'est l'autoriser pour tous moins une personne. La négation correspond donc à une différence ensembliste. Ainsi, la logique de sécurité sous LFS est basée sur une *logique d'ensemble*. La grammaire de cette logique est décrite ci dessous :

```
Prop --> Set ';' Rights

Set --> usr
      | 'group' '('  grp ')'
      | 'union' '(' Set ',' Set ')'
      | 'inter' '(' Set ',' Set ')'
      | 'minus' '(' Set ',' Set ')'
      | 'all'
      | 'not'   '(' Set ')'

Rights --> Read Write List Exec

Read  --> 'r' | empty
Write --> 'w' | empty
List  --> 'l' | empty
Exec  --> 'x' | empty
```

L'unité lexicale `usr` peut contenir le nom d'un utilisateur comme on le trouve dans le fichier `/etc/passwd`. L'unité lexicale `grp` peut contenir le nom d'un groupe comme on le trouve dans le fichier `/etc/group`. La propriété `all` correspond à l'ensemble qui inclut tous les utilisateurs du système. Elle peut être mise en relation avec la propriété `other` des systèmes classiques. Ainsi, l'ensemble `not(foo)` est en fait une sorte de sucre syntaxique correspondant à l'ensemble `minus(all, foo)`.

La propriété `l` pour *listing* contrôle la *visibilité* d'un fichier, ce qui est normalement la responsabilité du répertoire contenant ce fichier. Cependant, comme on va le voir plus loin Section 4.2.2 page 120 cette responsabilité a été déplacée vers le fichier.

Ainsi, avec ce dernier modèle, un utilisateur désirant donner les droits de lecture à tous les utilisateurs excepté Charlie pour les sources C présents dans son répertoire courant n'aura qu'à exécuter la commande :

`mv *.c secu:not(charlie);r/`

Si il veut de plus donner l'accès en écriture à Alice il n'a qu'à rajouter en plus la propriété `secu:alice;w`.

Il faut noter que comme les propriétés de sécurité sont des propriétés LFS et donc des répertoires, l'utilisateur peut naviguer parmi ces propriétés. Par exemple, l'utilisateur `foo` désirant contrôler qu'aucun fichier lui appartenant ne puisse être accédé en lecture par `charlie` peut se placer dans le répertoire `/lfs/owner:foo/secu:charlie;r/` pour vérifier qu'il ne contient aucun fichier. La négation peut aussi être utile dans le cadre du contrôle en examinant par exemple le répertoire `/lfs/owner:foo/secu:not(foo);r/` qui permet de voir qui a le droit d'accéder au fichier de `foo` à part bien sûr `foo` lui même.

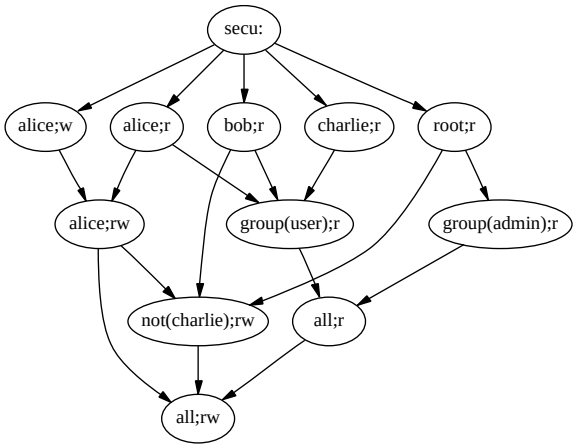

FIG. 4.2 – Cache logique pour la sécurité

Après sa *syntaxe*, un moteur de déduction logique est défini par sa *sémantique* avec la relation de déduction $\models$. Dans notre cas cette relation est simplement l'inclusion ensembliste :

$X; Y \models X'; Y'$ si $X' \subset X$ et $Y' \subset Y$.

Par exemple, $\mathit{foo}; rl \models \mathit{union}(\mathit{foo}, \mathit{group}(\mathit{admin})); rwl$ car on a à la fois $\{\mathit{foo}\} \subset \mathit{union}(\mathit{foo}, \mathit{group}(\mathit{admin}))$ et $\{r, l\} \subset \{r, w, l\}$.

Le mécanisme de vérification est sensiblement le même que pour les deux modèles précédents. Par exemple, l'accès en écriture à un fichier par un utilisateur `foo` entraînera le calcul de l'extension de la propriété `secu:foo;w` pour vérifier que ce fichier fait parti de cette extension. Il faut noter que cette propriété n'a pas besoin d'avoir été créée manuellement. À son premier accès, cette propriété sera automatiquement insérée dans le cache logique, et tous les fichiers ayant par exemple la propriété `secu:not(charlie);rwl` seront aussi dans l'extension de `secu:foo;w`.

La Figure 4.2 représente un exemple de cache logique (voir Section 3.1.1 page 88) induit par les règles d'ordonnancement des propriétés de la logique de sécurité.

Il faut noter que le moteur de déduction logique doit accéder et analyser le fichier `/etc/group` pour transformer un groupe en sa liste de membres pour que l'inclusion ensembliste puisse fonctionner. Cela n'est cependant pas gênant d'un point de vue efficacité. En effet, ce moteur n'est appelé que lors de l'insertion d'une nouvelle formule, or il est rare que l'on rajoute par exemple un nouveau groupe. Ainsi, l'ensemble des propriétés de sécurité utiles sera très vite couvert. L'ouverture d'un nouveau fichier et la vérification des droits d'accès est donc rapide sous LFS ; elle ne met en jeu que le calcul de l'extension d'une propriété qui sera certainement déjà présente dans le cache et déjà indexée.

Avec ce modèle, l'ajout d'un nouvel utilisateur ou d'un nouveau groupe est automatique. Par exemple, étant donné le graphe précédent, la mention d'un nouvel utilisateur `eve` et d'une nouvelle propriété `secu:eve;r` ajustera automatiquement ce graphe comme décrit Figure 4.3. On peut voir comment les propriétés mentionnant des négations et la propriété `all` sont automatiquement replacées au bon endroit à l'aide des algorithmes gérant le cache logique (voir Section 3.1.1 page 92) et la règle de déduction spécifiée par la logique de sécurité présentée précédemment.

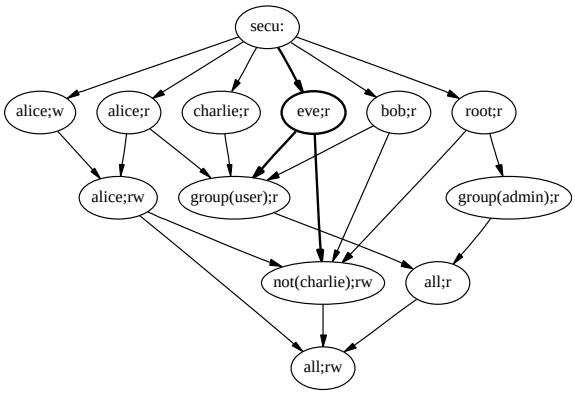

FIG. 4.3 – Cache logique après modification

4.2 Les répertoires : un modèle à la Unix

Nous venons de proposer un modèle de sécurité pour les fichiers sous LFS. Nous allons maintenant en proposer un pour les répertoires. Du fait de la différence très importante entre LFS et les systèmes de fichier classiques, on ne peut pas ré-appliquer tel quel les modèles de sécurité existants. Pour les fichiers, le modèle proposé précédemment est certes meilleur mais il n'est pas fondamentalement différent des modèles classiques du fait que si l'on fait abstraction de la notion de vue, les fichiers ont le même statut dans ces deux systèmes. Pour les répertoires, leurs statuts est vraiment très différent sous LFS.

Nous verrons ainsi que l'impossibilité de ré-appliquer les modèles existants conduit à déplacer certaines *responsabilités* du répertoire au fichier. Cela complétera ainsi le modèle de sécurité pour fichier présenté précédemment. Ce déplacement de responsabilité nous amènera alors à nous poser la question du besoin même de droits pour les répertoires. En effet, LFS ayant changé beaucoup de choses par rapport aux systèmes de fichier hiérarchiques, les raisons qui ont fait le besoin de propriétés de sécurité pour les répertoires dans les systèmes d'hier ne s'appliquent plus, ou du moins peuvent être résolues sans passer par des droits sur les répertoires.

En effet, comme nous allons le voir plus précisément après, la possibilité d'utiliser la négation, la conjonction, d'utiliser des droits avancés sur les fichiers, ou encore le fait d'avoir des propriétés intrinsèques systèmes rend les droits sur les répertoires presque inutiles. Cependant, comme nous le verrons aussi, LFS, même si il rend invalides les anciennes raisons pour avoir des droits sur les répertoires, en introduit de nouvelles. Ainsi, nous proposerons tout de même un modèle de sécurité pour les répertoires.

4.2.1 Répertoires et propriétés

Sous LFS on ne parle plus de répertoires mais de propriétés. Un répertoire est juste une formule mentionnant un ensemble de propriétés. Ainsi, sous LFS, l'assignation de droits par l'utilisateur se fait sur les propriétés et non sur les répertoires.

Il faut noter que du fait que désormais les propriétés LFS peuvent représenter des droits (par exemple la propriété `secu:foo;rw`), l'expression «assigner des droits aux propriétés» peut paraître confus. Cette section s'attache à définir un modèle de sécurité pour les répertoires, et donc pour les propriétés. Les droits des propriétés LFS ne peuvent cependant pas être des propriétés. On ne peut pas donner la propriété `secu:foo:rw` à la propriété `movie`, car `movie` n'est pas un objet. De plus il faut forcément un moment amorcer le processus puisque quelles seraient les droits de la propriété `secu:foo;rw` ? Ainsi, sous LFS les fichiers ont des permissions représentées sous la forme de propriété LFS (`secu:foo;rw`), et les propriétés LFS ont des permissions représentées sous forme classique Unix. Pour les fichiers on utilise donc par exemple la commande `mv fichier.txt secu:foo;rw/` et pour les propriétés la commande `chmod a+rw movie/` (ou même `chmod a+rw secu:foo;rw/`). Dans ce qui suit nous parlerons donc de permissions pour parler des droits sur les propriétés, et de propriété pour désigner une propriété LFS, quelle qu'elle soit.

Cependant, avec des permissions sur les propriétés, quelles sont alors les permissions d'un répertoire représentant une conjonction de propriétés comme `movie/port` ? Sous un système de fichier classique, ce serait les permissions du dernier élément du chemin, ici `port`, qui donneraient les permissions de l'ensemble du chemin. Cela paraît logique puisque la création d'un fichier dans `port` n'aurait pas d'influence sur le répertoire `movie`, et ne modifierait donc pas son contenu. Cepen-

dant, sous LFS, ajouter un fichier dans `movie/port` affectera le résultat des futures commandes `ls` sur le répertoire `movie`, et donc par effet de bord modifiera d'une certaine manière aussi le contenu de ce répertoire. Par exemple, la commande `ls /movie` affichera désormais `port` comme sous répertoire, puisque `port` sera maintenant un incrément.

L'ordre des propriétés dans le chemin ne veut rien dire sous LFS, et donc c'est logiquement que les permissions d'une conjonction de propriétés comme `movie/port` doivent être la conjonction des permissions de ces propriétés. Ainsi, en enlevant la permission `w` au répertoire `movie`, son propriétaire est assuré que personne ne pourra placer de fichiers dans `/movie` ou dans `/movie/port`.

Ainsi, en supposant qu'il y ait un intérêt à avoir des droits sur les propriétés, le fait de gérer les permissions des répertoires par une conjonction de permissions rend les commandes qui créent des fichiers comme `touch` sures. Cependant, qu'en est-il des autres commandes comme `rm`, `mv` ou `ls` et des autres droits `r` et `x` ?

La sémantique de LFS pour `ls` permet à l'utilisateur de voir un fichier sans pour autant spécifier toutes ses propriétés. Il suffit d'en spécifier assez pour pouvoir désigner ce fichier de manière non ambiguë. C'est comme si l'on pouvait dans un système de fichier hiérarchique voir le contenu d'un sous-répertoire directement depuis des répertoires plus haut dans la hiérarchie. Cela a des conséquences assez étrange sur la sécurité. En effet, un utilisateur pourrait voir un fichier qu'il ne possède pas depuis un répertoire où il aurait la permission `w`. Cela veut dire que même en utilisant la conjonction de permissions pour les répertoires, avec une sémantique de sécurité traditionnelle il pourrait effacer ce fichier avec `rm`, même si il n'avait pas la permission d'écriture sur toutes les propriétés décrivant ce fichier. De même, il pourrait ajouter ou enlever des propriétés à cette description avec la commande `mv`.

De manière similaire, la nouvelle sémantique de `ls` a aussi d'étranges répercussions sur la permission `r` d'une propriété. Sous un système de fichier hiérarchique, enlever la permission `r` à un répertoire implique que les autres utilisateurs ne pourront pas voir son contenu, c'est à dire le nom des fichiers et sous-répertoires composants ce répertoire. Cela permet de cacher un fichier. En effet, même le nom d'un fichier peut déjà contenir de l'information que l'on n'aimerait pas partager. Sous LFS, un fichier peut être vu depuis de nombreux répertoires, et `ls` peut même afficher un fichier dans un répertoire général alors que ce fichier possède des propriétés plus spécifiques, car ce fichier peut déjà être désigné de manière non ambiguë. Un utilisateur pourrait ainsi se placer dans un répertoire avec des droits de lecture mais contenant cependant des fichiers dont les propriétés, en fait plus spécifiques, ne permettent pas la lecture. On pourrait pour la commande `ls` filtrer en interne ces fichiers et vérifier que l'utilisateur a effectivement le droit de lecture sur les propriétés. Cependant faudrait il vérifier qu'il a le droit de lecture sur toutes ces propriétés ou suffirait-il qu'il ait le droit de lecture sur certaines d'entre elles, par exemple celles mentionnées dans le `pwd` ?

On rencontre les mêmes genres de problèmes avec la permission d'exécution `x`. Dans un système classique, si un répertoire n'a pas cette permission, ni celui-ci, ni ses sous-répertoires ne sont accessibles. C'est une autre manière de cacher un fichier en interdisant l'accès au répertoire contenant ce fichier. Sous LFS, un utilisateur peut mentionner une propriété de n'importe où ; c'est la faculté de pouvoir interroger en plus de simplement naviguer. Sous LFS, on peut ainsi «sauter» par dessus des propriétés hautes pour aller directement vers des propriétés plus basses dans la hiérarchie des concepts, des propriétés plus précises. Si l'on voulait appliquer une sémantique de sécurité classique pour les répertoires, il faudrait vérifier avant chaque requête, avant chaque commande `cd`, que l'utilisateur a les droits d'exécution non seulement sur la propriété mentionnée mais aussi sur toutes les propriétés plus générales liées à cette propriété. De plus, du fait que sous LFS une propriété peut avoir plusieurs parents, et donc peut être le sous-concept d'un ensemble de concepts, il faudrait en fait vérifier qu'il existe un chemin permettant cette exécution. Cela compliquerait les algorithmes.

En fait, vouloir appliquer la sémantique de sécurité traditionnelle aux répertoires sous LFS compliquerait non seulement les algorithmes, mais comme on va le voir dans la section suivante cela n'aurait pas de sens.

4.2.2 Un modèle centré sur les fichiers

Comme on vient de le voir, les permissions des propriétés présentes dans le `pwd` d'un répertoire ne suffisent

pas pour assurer le contrôle de l'*effacement*, du *déplacement*, et de la *visibilité* d'un fichier dans ce répertoire. On devrait regarder aussi les autres propriétés du fichier. D'une certaine manière ce n'est plus le répertoire, le `pwd`, qui guide la sécurité mais le fichier.

Dans les systèmes de fichier hiérarchiques, confier ces différentes responsabilités au répertoire peut paraître logique puisqu'un fichier ne fait parti que d'un répertoire. Cependant, dans le contexte LFS, un fichier peut faire parti de nombreux répertoires en même temps car il peut posséder de nombreuses propriétés. Ainsi, doit-on autoriser une action sur un fichier uniquement si l'utilisateur a le droit sur toutes les propriétés ou juste si il a le droit sur l'une d'entre elles ? En fait, confier ces responsabilités aux propriétés paraît finalement moins logique que de le confier au fichier lui même. En effet, sous LFS un fichier n'appartient plus vraiment à un répertoire, à une propriété, mais c'est plutôt l'inverse : c'est la propriété qui appartient au fichier.

LFS retourne donc la vision des choses et centre la sécurité sur le fichier. Nous avons donc déplacé ces différentes responsabilités, le contrôle de l'effacement, du déplacement et de la visibilité d'un fichier, du répertoire au fichier. Ainsi, toutes les propriétés de sécurité concernant un fichier sont représentées par des droits de ce fichier, et non plus éparpillées à la fois sur le fichier et le répertoire contenant ce fichier.

Ainsi, en ce qui concerne l'effacement et le déplacement sous LFS, seul le propriétaire du fichier peut exécuter les commande `rm` et `mv` sur ce fichier. Pour le déplacement, il faut de plus qu'il ait le droit d'écriture sur les propriétés qu'il rajoute et enlève.

Pour le contrôle de la visibilité, nous avons introduit sous LFS un quatrième droit, le droit de voir un fichier, c'est à dire de voir le fichier dans les listings retournés par la commande `ls`. Ainsi, en plus de `r`, `w`, et `x` vient donc se rajouter le `l` pour *Listing*. Ainsi, pour bénéficier des mêmes genres de service offerts pas les systèmes de fichiers hiérarchiques, un utilisateur sous LFS peut ajouter (ou ne pas mettre) par exemple la propriété `secu:all;l` à un fichier pour permettre aux autres utilisateurs de voir ce fichier. Cela permet d'offrir un contrôle d'accès discrétionnaire pour la visibilité du fichier. Sous un système de fichier hiérarchique le grain était celui du répertoire ; tous les fichiers d'un répertoire étaient malheureusement logés à la même enseigne.

Ajouter seulement une propriété à un fichier ne fournit cependant pas la machinerie nécessaire pour cacher un fichier. Il faut que le système utilise ensuite à bon escient cette connaissance. Ainsi, chaque requête d'un utilisateur, c'est-à-dire chaque répertoire issu de la navigation d'un utilisateur, aura une formule implicite représentant l'utilisateur en plus de la formule explicite représentant le `pwd` de ce répertoire. Par exemple, la formule complète correspondant au répertoire `/art/port`, pour un utilisateur `foo` sera en faite $art \wedge port \wedge secu{:}foo;l$. Ainsi, les fichiers n'ayant pas la propriété `secu:foo;l` seront invisibles pour l'utilisateur `foo`. Cela veut dire que deux utilisateurs exécutant la même commande `ls` dans le même répertoire pourront parfois voir deux choses différentes.

Concrètement, cette fonctionnalité est implémentée en modifiant juste l'opération `readdir` du VFS, qui appellera désormais la fonction interne de LFS `ls` avec la formule complète, et non plus juste le `pwd`.

4.2.3 Du besoin de droits sur les répertoires

Nous venons donc de déplacer vers le fichier certaines responsabilités gérées habituellement par les droits du répertoire. Ainsi, que reste-il comme responsabilités pour le répertoire ? Pourquoi utilisait-on les droits sur les répertoires dans les systèmes de fichier classiques ?

On utilisait le `r` et le `x` pour pouvoir cacher des fichiers. Cependant, comme on l'a vu dans la section précédente, LFS propose une nouvelle propriété pour les fichiers, le `l`, qui permet déjà de cacher des fichiers.

On utilisait le `w`, ou plutôt on n'utilisait pas le `w`, à la fois pour protéger les fichiers d'un répertoire en interdisant à quiconque de pouvoir les effacer, et à la fois pour interdire à quiconque de pouvoir en créer de nouveaux.

En ce qui concerne le contrôle de l'effacement, comme on l'a vu précédemment confier cette responsabilité à un répertoire a moins de sens sous LFS. Ainsi, sous LFS seul le propriétaire du fichier peut effacer ce fichier.

En ce qui concerne le contrôle de création, qui représente l'intérêt principale du droit `w` sur les répertoires, on l'utilise pour empêcher qu'un utilisateur malveillant puisse créer de trop nombreux fichiers dans un

répertoire. En effet, cela parasiterait la navigation. Pour un répertoire comme `/bin`, autoriser l'accès en écriture serait même dangereux à cause du risque d'introduction de virus de type cheval de Troie. Cependant, les possibilités d'organisation de LFS peuvent résoudre ces problèmes. Il suffit que chaque fichier ait comme propriété supplémentaire l'identité de son propriétaire, par exemple `owner:root`. Cette propriété sera automatiquement attribuée par le système, et ne pourra pas être enlevée ni forgée. Les programmes sûrs peuvent ensuite être accédés simplement à partir de `/bin/owner:root/`. De la même manière, l'utilisation de la négation combinée avec le nom d'un utilisateur malveillant comme par exemple dans la commande `cd /lfs/ !owner:charlie/`) permet de naviguer partout sans être dérangé par les fichiers parasites de Charlie.

Ainsi, sous LFS ni le `r`, ni le `x`, ni le `w` ne sont vraiment utiles. On peut permettre à tous de lire, de rentrer ou d'écrire dans tous les répertoires. Cela n'aura aucune conséquence néfaste pour l'utilisateur puisqu'il aura d'autres moyens pour s'assurer de la protection de ses fichiers.

Ce modèle permet de plus d'améliorer l'aspect travail coopératif et *partage* de données comme on va le voir maintenant. Comme on l'a déjà dit, les possibilités de partage forment un des aspects importants qui contribuent au succès d'un modèle de sécurité.

L'idée d'encourager le partage de données et de centrer plus la sécurité sur les fichiers est en fait déjà présente dans les systèmes de fichier Unix. En effet, on retrouve le même genre de décision avec le répertoire `/tmp`. On veut d'un côté que chaque utilisateur puisse créer un fichier dans `/tmp`, ce qui implique de donner la permission d'écriture à tous sur ce répertoire, et de l'autre côté on ne voudrait pas qu'un utilisateur puisse effacer le fichier d'un autre utilisateur, ce qui implique cette fois d'enlever la permission d'écriture. Le fait de rassembler avec la même propriété `w` les deux possibilités que sont la création et l'effacement d'un fichier a donc un inconvénient important. Pour résoudre ce problème, la sécurité dans les systèmes classiques a été étendue avec une nouvelle propriété, le *sticky-bit* ou `t` (voir Section 1.5.1 page 33). Dans un répertoire ayant cette propriété, lors d'un effacement de fichier le système ne regarde plus les permissions du répertoire mais l'identité du processus ayant demandé cet effacement. Le système vérifie que l'identité de ce processus est la même que l'identité du propriétaire du fichier. Sous LFS ce n'est plus l'exception mais la règle ; c'est comme si tous les répertoires avaient implicitement ce bit.

Ce «modèle *sticky*» tout comme le modèle LFS encourage plus le partage que le modèle du droit `w` sur le répertoire. En effet, tout le monde partage `/tmp` ce qui a l'avantage de fournir un point de référence pour pouvoir stocker des données temporaires, non seulement pour l'utilisateur mais aussi surtout pour les processus. C'est l'occasion aussi de bénéficier de services partagés comme par exemple le nettoyage automatique de ce répertoire par le système à chaque redémarrage.

Le répertoire `/tmp` est malheureusement à peu près le seul cas sous Unix de partage. En fait, la raison n'est pas le fait d'avoir un modèle de sécurité (celui d'Unix) trop restrictif, car le *sticky bit* permet le partage, mais le fait d'avoir des moyens d'organisation (ceux des systèmes de fichiers hiérarchiques) restreints.

Par exemple les répertoires `/bin` ou `/doc` ne sont par partagés sous ces systèmes. La première raison de ce choix a déjà été évoquée précédemment. Sous ces systèmes, l'envie de partager est tempérée par le besoin de protéger. En effet, autoriser le partage impliquerait ensuite des trous de sécurité ou pourrait amener avoir trop de fichiers dans ces répertoires ce qui rendrait la navigation pénible. Les possibilités offertes par LFS comme la possibilité de «tagger» chaque fichier avec une propriété supplémentaire `owner:` et d'interroger ou naviguer ensuite de manière plus flexible que sous un système de fichier hiérarchique résoud ce problème. Sous LFS on peut donc satisfaire et les besoins de partage et les besoins de protection.

De plus, même si on autorisait l'écriture sur `/bin` ou sur `/doc` sous les systèmes de fichier hiérarchiques, en faisant abstraction du besoin de protection, les utilisateurs n'y placeraient pas leurs données. En effet, les utilisateurs et administrateurs ne veulent pas mélanger leurs données, afin de pouvoir ensuite retrouver leurs propres documents plus facilement. Il y a d'autres critères d'organisation que seulement le fait d'être un binaire. Ainsi, il n'est pas rare que chaque utilisateur possède son propre répertoire `bin`, qu'il range ses musiques, ses documents selon sa propre classification dans son coin. Cependant

là encore ces problèmes n'existent plus sous LFS du fait que l'utilisateur peut placer son fichier dans `/doc` pour le partager, sans avoir à renoncer à sa propre classification du fichier puisqu'il peut lui assigner d'autres propriétés.

Le fait de pouvoir décrire un fichier par de multiples propriétés et d'utiliser ensuite des conjonctions dans ses recherches ouvrent désormais de nombreuses possibilités. Ainsi, nous espérons qu'en plus de permettre une meilleure organisation des données, les fonctionnalités de LFS auront un impact fort sur le partage de ces données ainsi que le travail coopératif autour de ces données. Sous LFS, un utilisateur devrait normalement avoir moins besoin de créer ses propres répertoires ; il pourra réutiliser ceux qui existent. Dans l'idéal, ces répertoires formeront des taxinomies qu'un expert d'un domaine aura créées, et les fichiers seront automatiquement rangés à l'aide de transducteurs réalisés aussi par des experts. Ainsi, on pourra avoir par exemple un seul répertoire `bin`, un seul répertoire `doc` partagés par tous.

Ainsi, comme on l'a vu le `r`, le `x` et le `w` sur les propriétés ne servent plus vraiment. Cependant, la propriété `w` sera quand même utilisée mais pour de nouvelles raisons. En effet, au moins pour amorcer le processus, on ne peut autoriser n'importe qui à effacer ou ajouter par exemple la propriété `owner:foo` à un fichier. Il faut donc au moins protéger cette propriété. De même, pour les autres propriétés intrinsèques, on s'attend à ce que par exemple le répertoire `ext:c` contienne uniquement des fichiers C. Bien sûr, si `owner:` était déjà protégée, les autres propriétés auraient moins besoin de l'être. Cependant, même si par l'usage combiné des propriétés `owner:` et de la négation un utilisateur peut écarter les utilisateurs malveillants qui auraient placé de faux fichiers C dans ce répertoire, cet utilisateur peut se retrouver sans le vouloir lui aussi malveillant. En effet, sans le vouloir, à cause d'une erreur d'inattention, un utilisateur pourrait placer un fichier non C dans le répertoire `ext:c`. Ainsi, pour son propre bien, il est important de protéger certains répertoires pour que le système le prévienne lors de son erreur (en faisant en sorte que seul le transducteur C ait le droit d'y écrire). Il faut noter que les besoins de protection sont tout de même différents de ceux des systèmes classiques. En général, les droits d'un répertoire protègent les fichiers de ce répertoire ; sous LFS ces droits protégeront en fait le répertoire lui-même.

On a donc décidé d'avoir tout de même des droits sur les répertoires, en fait sous LFS des droits sur les propriétés. Cependant, on ne se servira que du droit d'écriture et du fait d'avoir un propriétaire. De plus, le droit d'écriture ne conférera même pas le droit d'effacer un fichier de ce répertoire.

4.3 Sémantique des commandes *shell*

Nous allons maintenant résumer la sémantique finale vis a vis de la sécurité des commandes *shell* dans ce nouveau contexte. Nous allons aussi voir quelques détails de sécurité que l'on n'a pas encore évoqués qui affineront un peu plus notre modèle.

`touch` La sémantique de sécurité de la commande `touch` se base sur le fait que la permission `w` d'un chemin (c'est la seule qu'on utilise pour les propriétés) est la conjonction des permissions des propriétés de chaque composant de ce chemin. Il est important de rappeler que normalement un utilisateur ou administrateur aura moins besoin sous LFS de protéger ses propriétés, et sera encouragé au contraire à les partager (partage qui sera on le rappelle sans risque).

`mkdir` Il faut déterminer aussi la sémantique de sécurité pour la commande de création `mkdir`. Traditionnellement, le droit `w` sur un répertoire conditionne à la fois la création de fichier dans ce répertoire et la création de sous-répertoires. Ce n'est pas forcément souhaitable sous LFS. Ainsi, même si l'on veut permettre à tous de placer par exemple des fichiers dans le répertoire `secu:eve;rl`, ce n'est pas pour autant que l'on veut autoriser la création de sous-concepts liés à ce répertoire. Les fichiers et les répertoires, les objets et les propriétés, sont tout de même deux choses très différentes sous LFS. Ainsi, on autorise uniquement le propriétaire d'une propriété à créer des sous-propriétés. Le *root* aura bien sûr tous les droits, comme sous les systèmes classiques. C'est donc lui qui amorcera le processus en attribuant à chaque sous-concept de la propriété `home` (`/home/foo`, `/home/bar`) son proprié-

taire correspondant. Chaque utilisateur pourra ainsi développer sa propre taxinomie à partir de son *homedir*. La plupart des propriétés du *root* comme `secu:`, `bin` ou `doc` autoriseront l'écriture ce qui permettra à tous les utilisateurs d'ajouter ces propriétés à leurs fichiers (en plus de leurs propres propriétés).

`rm` On a déjà vu précédemment que pour encourager le partage sans risque, seul le propriétaire peut effacer son fichier.

`mv` Il en est de même sous LFS pour le renommage et le déplacement. Ainsi, seul le propriétaire peut renommer son fichier, ou encore ajouter et enlever des propriétés à son fichier avec la commande `mv`, à condition bien sûr que comme pour `touch` il ait les droits d'écriture sur ces propriétés. Cette dernière condition fait qu'en protégeant les propriétés `owner:` (possédées par le *root*), en ne leur conférant pas le droit d'écriture, le propriétaire d'un fichier ne pourra pas enlever cette propriété de son fichier ni en forger une autre. Cette condition n'empêche heureusement pas son propriétaire d'ajuster les autres propriétés, celles n'étant pas critique pour la sécurité du système.

Pour `chmod`, sous LFS comme sous les systèmes de fichier hiérarchiques, seul le propriétaire peut ajuster les droits de son fichier. En fait, cette commande n'est plus utile pour les fichiers car les propriétés de sécurité des fichiers sont des propriétés comme les autres. L'utilisateur peut donc utiliser la commande `mv` pour ajuster la sécurité d'un fichier. Ce choix rend d'ailleurs cohérent la décision précédente concernant le `mv` puisque comme certaines propriétés ont trait à la sécurité comme `secu:foo;rl`, et que le `mv` mime dans ce cas exactement la commande `chmod`, il est donc logique d'appliquer la même politique à `chmod` et à `mv`.

`rmdir` On procède de même pour que pour les fichiers, et seul le propriétaire d'une propriété peut l'effacer. Il faut rappeler que sous LFS la descruction d'une propriété n'entraine pas la destruction des fichiers ayant cette propriété. Elle entraine juste la mise à jour des descriptions de ces fichiers.

`mvdir` Il en est de même pour le renommage et le déplacement. Seul le propriétaire peut renommer ou ajuster les axiomes concernant sa propriété avec la commande `mv`, et changer ses permissions avec la commande `chmod`. On pourrait décider de confier la responsabilité du renommage d'un répertoire à son parent, comme c'est le cas sous Unix. Après tout, même si le répertoire `/home/foo` appartient à `foo`, on ne souhaiterait pas forcément que celui-ci puisse le renommer. Cependant, pour être plus cohérent avec la politique appliquée aux fichiers, on préfère confier ce droit au répertoire lui-même. Ainsi, dans chacun des cas, pour le fichier ou la propriété, chacun sera responsable de lui-même.

`ls` On a déjà vu précédemment que pour cacher un fichier au processus de navigation on préférait utiliser la propriété `l` d'un fichier, alliée au `pwd` implicite (chaque requête correspondant à un répertoire pour un utilisateur `foo` se voit rajouter implicitement via une conjonction la propriété `secu:foo;l`) plutôt que les propriétés `r` et `x` des répertoires. En effet, ce choix permet d'abord un contrôle plus fin pour la visibilité au niveau du fichier, et il est plus logique là encore dans le contexte de LFS. Les propriétés étant aux services des fichiers et non l'inverse, on préfère confier la responsabilité de la visibilité des fichiers aux fichiers eux-mêmes. De plus, la propriété `owner:` et la négation permettent à un utilisateur de naviguer sans risque de pollution.

`read/write` En ce qui concerne les commandes hypothétiques `read` et `write` qui permettent de lire et de modifier le contenu d'un fichier, on se repose sur les propriétés de la forme `secu:foo;r` et `secu:foo;w` et du calcul d'extension.

4.4 Conclusion

La première contribution de ce chapitre est déjà d'avoir proposé un modèle de sécurité pour LFS. En effet, les modèles de sécurité existants ont été conçu dans le contexte des systèmes de fichier hiérarchiques ; il a fallu donc en extraire les idées sous-jacentes pour les adapter ensuite au contexte LFS, très différent.

De plus, cette adaptation a conduit à une meilleure intégration de la sécurité dans le système de fichier. En ef-

fet, dans les systèmes de fichier hiérarchiques, les mécanismes internes et les interfaces pour gérer la sécurité sont très différents des autres mécanismes et interfaces utilisés par les autres services du système de fichier. Sous LFS, les propriétés de sécurité sont des propriétés comme les autres et les règles de sécurité sont des règles logiques comme les autres. Ainsi, en ce qui concerne l'aspect interface, l'utilisateur peut utiliser le même jeu de commande qu'il utilisait pour organiser ses fichiers pour aussi paramétrer la sécurité de ces fichiers. En ce qui concerne les mécanismes, ils s'intègrent bien avec les autres services offerts par LFS et l'esprit logique de LFS puisque c'est un moteur de déduction logique et le mécanisme d'extension qui permettent d'implémenter les vérifications.

Le modèle de sécurité LFS est aussi plus «orthogonal» que les modèles de sécurité des systèmes hiérarchiques. En effet, sous ces modèles certains choix peuvent être déstabilisant pour un nouvel utilisateur ; il existe des bizarreries. Par exemple, un utilisateur voulant protéger un de ses fichiers peut s'attendre en enlevant la propriété `w` à ce fichier à être sur que personne ne pourra modifier son contenu. Cependant, un cas pire que la simple modification peut arriver puisque si ce fichier fait parti d'un répertoire autorisant l'écriture, un autre utilisateur pourra carrément effacer tout son contenu en supprimant ce fichier. Avec le modèle LFS, qui recentre plus la sécurité sur les fichiers que sur les répertoires, les décisions concernant la sécurité d'un fichier sont toutes localisées dans les propriétés de ce fichier. Elles ne sont donc pas éparpillées à la fois sur le fichier et sur le répertoire contenant ce fichier. De même, le statut de protection d'un répertoire concerne uniquement ce répertoire, et n'a pas d'impact sur le statut de protection des fichiers de ce répertoire.

Enfin, le modèle de sécurité proposé exploite à bon escient les fonctionnalités offertes par LFS. L'utilisateur peut ainsi bénéficier de nouveaux services comme par exemple l'utilisation de la négation (on retrouve là encore l'esprit logique de LFS et son impact), la possiblité de naviguer parmi les propriétés de sécurité, ou le contrôle d'accès discrétionnaire pour la visibilité d'un fichier. Le modèle LFS est ainsi plus expressif que les modèles classiques pour protéger, tout en permettant de plus d'améliorer l'autre aspect important d'un modèle de sécurité : le partage. En effet, LFS rend plus facile que dans les systèmes de fichier hiérarchiques le partage de données et le travail coopératif.

Chapitre 5

Extensions

Beware of the Turing tarpit in which everything is possible but nothing of interest is easy.

Alan Perlis

Le Chapitre 2 a exposé les principaux mécanismes de LFS. Les fonctionnalités de LFS qui y ont été présentées se veulent un ensemble cohérent et forment le noyau dur de LFS. Il serait difficile d'éliminer de ce noyau une fonctionnalité sans que cela ait un impact sur les autres fonctionnalités du noyau. C'est l'interaction de ces fonctionnalités qui fait l'originalité de LFS. Nous présentons dans ce chapitre d'autres fonctionnalités qui ont été introduites dans le prototype de LFS, fonctionnalités appréciables mais moins nécessaires. Ces extensions peuvent aussi être classées selon les différents domaines de la gestion de l'information auxquelles elles s'attaquent. Ainsi, certaines extensions permettront une meilleure *organisation*, plus expressive, d'autres une meilleure *recherche*, plus rapide, et enfin d'autres une meilleure *manipulation*, plus flexible.

5.1 Organisation avancée

Le système de fichier à travers le *shell* fournit de nombreuses commandes comme `ls`, `cd`, `touch`, ou encore `mkdir`. Nous n'avons cependant pas encore parlé des commandes `ln` et `ln -s`, c'est à dire des liens, dans le contexte de LFS. On pourrait bien sûr mimer sous LFS le comportement qu'ils avaient dans les systèmes de fichier classiques. Cela n'aurait cependant pas grand intérêt puisque LFS permet déjà de fournir les services qu'offraient les liens dans les systèmes classiques, et même les améliore, en utilisant d'autres commandes. En effet, traditionnellement on créait des liens pour pouvoir référencer de plusieurs endroits le même fichier, pour pouvoir ensuite le retrouver plus facilement, par exemple en plaçant ce fichier dans différentes classifications en même temps. La Figure 1.2 page 15 montre un exemple d'utilisation de ces liens. C'est désormais possible sous LFS sans les liens puisque l'on peut associer à un fichier de nombreuses propriétés, en se servant par exemple de la conjonction à la création du fichier ou encore en utilisant la commande `mv` pour ajouter des propriétés à ce fichier. LFS va même plus loin que ces liens puisqu'il améliore, en même temps que les mécanismes d'organisation, les mécanismes de recherche ; les incréments permettent d'utiliser à bon escient ces multiples classifications. Cependant, au lieu de mimer, on peut changer cette sémantique des liens, tout comme on a déjà changé la sémantique des autres commandes. On va ainsi dans ce qui suit utiliser les liens d'une autre manière, pour améliorer le pouvoir d'organisation de LFS, tout comme dans un autre temps ils avaient amélioré le pouvoir d'organisation des systèmes hiérarchiques.

5.1.1 Lien propriété-fichier : réflexivité

Dans l'organisation actuelle, LFS force plus ou moins l'utilisateur à choisir une sorte de *schéma* pour ses informations puisque l'on a deux mondes bien distincts, le monde des objets avec les fichiers, et le monde des propriétés avec les répertoires. Par exemple, pour organiser ses fichiers musicaux, un utilisateur va considérer les chansons comme des objets, et les artistes comme

des propriétés. Ce choix permettra ensuite de facilement trouver une chanson en naviguant parmi les artistes. Mais cette séparation peut être trop contraignante. En effet, que se passerait-il si l'utilisateur voulait désormais gérer de l'information sur ces artistes ? On aimerait les classer selon leurs langues, leurs pays. Ensuite, un utilisateur aimerait non plus chercher des chansons, mais des artistes. Ces artistes deviendraient alors les objets de la recherche et non plus les propriétés de la recherche.

Il faut bien comprendre qu'utiliser la fonctionnalité de LFS qui permet de pouvoir classer les concepts dans une taxinomie, avec `mkdir`, et donc qui permettrait de classer les artistes, ne serait pas la solution. En effet, on pourrait faire de `artist:beatles` un sous-concept de `country:england`. Cependant, on lierait par un axiome logique deux concepts qui n'ont pourtant rien à voir entre eux. L'axiome $f \models g$ permet de dire que tout document parlant de f est aussi un document qui parle de g. Conceptuellement, le cinéma est en effet une forme d'art, et il est donc logique de créer l'axiome $movie \models art$ et donc de faire de $movie$ un sous-concept de art par le biais de la commande `mkdir` comme on l'a vu Section 2.3.1 page 70. Le concept `artist:beatles` n'a pas grand chose à voir avec le concept `country:england`. De plus, en imaginant que l'on veuille classer à leur tour les pays, on ferait ainsi de `country:england` un sous-concept de `beaupays`, ce qui induirait par transitivité que `artist:beatles` serait forcément lui aussi un sous-concept de `beaupays`, ce qui parait encore moins logique. En fait, ordonner des concepts orthogonaux n'a pas de sens, et l'on répéterait d'une certaine manière les erreurs que l'on avait commises dans les systèmes de fichier hiérarchiques. Cet illogisme dans l'organisation aurait ensuite des répercussions sur la recherche qui donnerait à son tour des résultats illogiques. En effet, à cause des axiomes, un fichier musical ayant la propriété `artist:beatles` aura forcément aussi les propriétés `country:england` et `beaupays`. Il sera donc aussi dans l'extension de requêtes mentionnant ces propriétés. Ainsi, la requête `cd beaupays/` contiendra des fichiers musicaux, ce qui n'est pas vraiment logique, et il sera même proposé comme complément de requête des propriétés musicales. Les propriétés et les objets sont intrinsèquement deux choses différentes et l'on ne peut pas se servir de l'un pour représenter l'autre.

Bien sûr, au lieu d'utiliser la taxinomie, un utilisateur pourrait créer de manière indépendante un fichier `beatles.info`, puis classer ce fichier, par exemple en lui donnant la propriété `country:england`. Cependant, ce fichier et la propriété `artist:beatles` n'auront aucune *connexion* entre eux, et même si l'on en créait une en affectant la propriété au fichier, on n'aurait tout de même aucun moyen pour formuler des requêtes intéressantes comme la recherche de fichiers musicaux fait par un artiste anglais.

Un utilisateur de LFS aimerait gérer de *multiples types d'information*, de multiple bases d'information, de l'information sur des fichiers musicaux, de l'information sur les artistes de ces fichiers musicaux, de l'information sur les pays de ces artistes. Mais un utilisateur aimerait aussi faire coopérer ces informations pour pouvoir exprimer des requêtes avancées.

Les *bases de données relationnelles* fournissent une solution à ces besoins. Elles n'ont pas deux mondes séparés ; un objet peut jouer le rôle d'une propriété, et une propriété le rôle d'un objet. Les bases de données gèrent de multiple tables, une table peut réunir de l'information sur des artistes, une autre sur des chansons, et l'expressivité de ces bases de données vient de la faculté d'établir des *liens* entre ces tables. Les Tableaux 5.1, 5.2, et 5.3 montrent un exemple de schéma d'organisation pour gérer les informations vues précédemment. Une notion importante est celle d'*identifiant*, qui est un numéro dont on se sert pour référencer une *entité*. Cet identifiant peut être présent dans plusieurs tables et c'est précisément lui qui établit des liens. Par exemple, dans la table des chansons le champ artiste contient des références pointant vers la table des artistes. Au début un champ contient généralement une chaîne de caractère, par exemple le champ artiste contiendra uniquement le nom de l'artiste, puis plus les données évoluent, plus on veut donner de l'information aux choses, plus on passe alors par les intermédiaires que sont les identifiants. Ces tables et avec elles un langage d'interrogation comme SQL permettent de formuler des requêtes avancées comme : «la recherche de chansons fait par des artistes habitant dans un beau pays».

```
SELECT Titre
```

```
FROM Chansons AS C,
     Artistes AS A,
     Pays AS P
WHERE P.Beau = "Oui" AND
      A.Pays = P.Id AND
      C.Artist = A.Id
```

En général on prend la requête à l'envers, on cherche donc tout d'abord des beaux pays. Les objets de la recherche sont donc des identifiants de pays. En utilisant la table des pays, une *sélection* et une *projection* on obtient ces identifiants. De là, on cherche des artistes habitant ces pays, les objets de la recherche deviennent donc des identifiants d'artistes, et cette fois les identifiants de pays vont servir comme propriétés de la recherche ; on a changé de *point de vue*. En effectuant une *jointure* entre la table des artistes et les identifiants des beaux pays, puis une projection, on obtient les identifiants d'artistes que l'on cherchait. Là encore, on va changer de point de vue, les objets de la recherche devenant désormais des identifiants de chansons et les propriétés de cette recherche devenant les identifiants d'artistes, ce qui permettra de finalement répondre à la requête. Nous allons sous LFS utiliser les mêmes genres de principes avec cette notion de *lien* et cette faculté de changer son *point de vue*.

Ainsi, pour rendre LFS réflexif, c'est à dire pour pouvoir considérer une propriété comme un objet, nous allons commencer comme précédemment par créer de manière séparée une propriété et un fichier, mais cette fois nous allons *connecter* avec un lien ces deux entités. Concrètement, la commande `ln artist:beatles/ beatles.info` connecte la propriété `artist:beatles` et le fichier `beatles.info`. La propriété `artist:beatles` devient une sorte de *propriété-objet*. Comme d'habitude, l'utilisateur peut classer le fichier `beatles.info` comme tous les autres fichiers, par exemple en assignant à ce fichier la propriété `country:england`. Ensuite, l'utilisateur peut comme d'habitude avec la commande `cd country:england/` sélectionner tous les fichiers partageant cette propriété. L'utilisateur aimerait cependant désormais avoir un moyen d'aller des fichiers vers leurs propriétés correspondantes, il aimerait changer son point de vue sur le type d'objets qu'il recherche. Nous

Id	Titre	Artist	Année	Genre
c1	staying alive	a3	1970	Disco
c2	yesterday	a1	1967	Pop
c3	été indien	a5	1975	Slow
c4	beach	a2	1967	Rock
c5	love to love	a4	1966	Disco

TAB. 5.1 – Table des chansons

Id	Nom	Pays	Type
a1	Beatles	p3	Groupe
a2	Beach Boys	p3	Groupe
a3	Bee Gees	p1	Groupe
a4	Donna Summer	p1	Solo
a5	Joe dassin	p2	Solo

TAB. 5.2 – Table des artistes

Id	Nom	Beau	Grand
p1	USA	Non	Oui
p2	France	Oui	Non
p3	Angleterre	Non	Non

TAB. 5.3 – Table des pays

ajoutons à LFS le mot-clef spécial `meta` pour pouvoir exprimer l'intention de passer du monde des objets au monde des propriétés, un peu comme on avait ajouté le mot-clef `parts` pour passer du monde des fichiers au monde des parties de fichiers. La commande `cd meta` transforme ainsi l'extension de la requête courante en une requête disjonctive dont les opérandes sont les propriétés associées, par le moyen des liens, aux objets de cette extension. On transforme d'une certaine manière une *extension* en une *intention*. Ainsi, la commande `cd country:england/meta` mènera au nouveau `pwd` $artist : beatles \vee artist : blur \vee \ldots$. On peut bien sûr enchaîner plusieurs `meta` pour formuler des requêtes complexes, et l'on pourra ainsi utiliser toutes les connaissances présentes sous LFS pour pouvoir précisément décrire ce que l'on cherche, et donc pour trouver plus rapidement ce que l'on cherche.

En reprenant les données et l'exemple de la Section 2.3.1 page 75, une suite possible de commandes utilisant les liens pourraient être :

```
[1] % cd /lfs/music/
[2] % cd artist:/genre:/
artist:Beatles/ Artist:BeachBoys/ ...
genre:Disco/ genre:Rock/ ...
note:excellent/ note:good/
staying_alive.mp3 yesterday.mp3
...
[3] % cd /lfs
[4] % mkdir artist ; cd artist/
[5] % mkdir country: type:
[6] % touch country:england/type:group/
          note:excellent/beatles.info
[7] % touch country:england/type:group/
          note:good/beachboys.info
...
[8] % ln artist:Beatles/ beatles.info
[9] % ln artist:BeachBoys/ beachb.info
...
[10] % cd /lfs
[11] % mkdir country ; cd country/
[12] % mkdir beau grand
[13] % touch beau/petit/france.info
[14] % touch grand/usa.info
...
[15] % ln country:france/ france.info
[16] % ln country:usa/ usa.info
...
```

Une fois les connections établies, on peut s'en servir pour formuler des requêtes avancées :

```
[17] % cd /lfs; ls
music/ artist/ country/
[18] % cd country/; ls
beau/ grand/
[19] % cd grand/; ls
usa.info
[20] % cd meta;  ls
artist/ poets/ ...
[21] % cd artist/; ls
type:group/ type:solo/
note:excellent/ note:good/
[22] % cd type:solo/; ls
DonnaSummer.info  Madonna.info
...
[23] % cd meta; ls
artist:DonnaSummer/ artist:Madonna/ ...
genre:Disco/ genre:Pop/ ...
year:[1970..1980]/ year:[1980..1990]/ ...
note:excellent/ note:good/
...
love_to-love.mp3 holiday.mp3
...
```

Il faut noter que nous ne défendons cependant pas l'usage des bases de données relationnelles pour gérer les données d'un utilisateur même si nous faisons se rapprocher LFS par certain côté de ces bases de données. En effet, nous pensons que l'interface d'un système de fichier est plus facile d'accès pour un utilisateur qu'un langage comme SQL. De plus, les tables de ces bases de données sont rigides ce qui n'est pas très pratique lorsque l'on veut classer des informations rapidement et que l'on ne possède pas les valeurs de tous les champs d'une table (*semi-structured data*). De plus, elles rendent difficile pour un utilisateur de faire évoluer ses informations ; par exemple l'ajout d'un nouveau type de propriété à une seule de ces entrées oblige à rajouter un champ dans une table et donc à changer le schéma de la base de données. Sans doute plus important, les bases de données ne sont basées que sur l'interrogation et ne supporte pas la navigation, qui comme on l'a vu, est un paradigme de

recherche aussi important. Ainsi, nous avons juste «emprunté» aux bases de données un de leurs concepts d'organisation : le lien. Ce concept ne vient pas brider les autres fonctionnalités présentes dans LFS, par exemple la navigation est ainsi toujours présente.

D'une certaine manière, LFS rend facile les choses faciles (par le biais de la simplicité du couple `cd`, `ls`), et désormais possible les choses complexes (par le biais de `ln` et `cd meta`). Au contraire les bases de données forcent tout de suite la complexité. Comme le disent Borenstein et Gosling dans [BG88] : *"The key principle is that there is simple syntax for simple operations. This does not mean that a more complex syntax should not be available for more complex operations, only that the complexity should not be forced on the programmer in a simple context."*.

Un système de fichier fournit des liens *hard* avec la commande `ln` mais aussi des liens *soft* avec la commande `ln -s`. Quelle peut être l'utilité de ces liens *soft* dans le contexte de LFS ? Nous pouvons les utiliser aussi comme précédemment pour établir un lien entre un fichier et une propriété, mais ce sera un lien un peu différent. Ainsi, au lieu de créer un fichier `beatles.info` et de le lier à la propriété `artist:beatles` avec un lien *hard*, on peut aussi créer un lien *soft* s'appelant `beatles.info` qui pointe vers la propriété `artist:beatles` avec la commande `ln -s /lfs/artist:beatles/ beatles.info`. Ce lien *soft* est en fait considéré comme un fichier, certes un peu spécial mais un fichier quand même, comme c'était déjà le cas dans les systèmes classiques. Ainsi, on peut classer ce fichier, ce lien, par exemple en lui donnant la propriété `country:england`. Ainsi, l'utilisateur peut lancer la commande `cd /country:england/beatles.info/` et se retrouvera alors dans le `pwd` $artist{:}beatles$. Cependant, contrairement à la technique précédente avec les liens *hard*, cette fois ce fichier, ce lien, n'a pas de contenu. On ne peut donc pas stocker de l'information sur les Beatles autrement que par le biais des propriétés. De plus, on ne pourra formuler des requêtes mettant en jeu un ensemble d'artistes et passer de cet ensemble d'objets à l'ensemble des propriétés correspondantes avec la commande `cd meta`. Avec les liens *soft*, on ne peut traiter qu'un lien à la fois, et donc qu'un artiste à la fois. C'est donc une solution plus «soft» aux problèmes précédents, mais qui a l'avantage d'être plus simple. Nous verrons plus loin d'autres usages de ces liens *soft* plus intéressants.

5.1.2 Lien fichier-fichier : relation

Cette façon de créer un fichier lorsque l'on crée une nouvelle propriété (en créant par exemple le fichier `beatles.info` suite à la création de la propriété `artist:beatles`, ce qui on le rappelle permettra de donner de l'information sur cette propriété), et de les lier ensemble peut être rendue symétrique. En effet, on peut aussi suite à la création d'un fichier créer une propriété lui correspondant et les lier ensemble. Cependant, à quoi servirait par exemple la propriété `toto.c:` qui correspondrait à un fichier `toto.c` ? quel fichier voudrait cette propriété ?

En fait, certains fichiers ont un lien implicite entre eux, par exemple le fichier objet `toto.o` est le résultat de la compilation du fichier source `toto.c`. De même, dans le cadre des pages `man`, certaines commandes ont un lien entre elles et sont mêmes référencées à la fin de la page `man` d'une commande dans la section `see also`. Une autre grande forme de recherche, en plus de l'interrogation et de la navigation, est la *recherche par similarité* (dit aussi *par l'exemple*, de proche en proche).

Ainsi, un utilisateur cherchant la commande pour changer de propriétaire, qu'il ne connaît donc pas, peut se dire qu'il y a des chances qu'elle soit liée à la commande pour changer de mode, qu'il connaît. Suite à la commande `man chmod`, il pourra en effet retrouver le nom de la commande qu'il cherchait, `chown`, dans la section `see also`.

Ainsi, on aimerait rendre ces liens plus explicites pour pouvoir ensuite s'en servir dans les recherches. Ces liens, dans le cadre des pages `man`, sont bien sûrs explicites puisqu'il suffit de regarder la section adéquate dans le contenu de ces pages, cependant il est préférable d'offrir là encore ce service au niveau du système de fichier afin que tous les fichiers puissent en bénéficier. Il n'y a cependant cette fois rien à rajouter dans LFS, pas de mot-clef spécial, pour pouvoir représenter ces liens. En effet, il suffit de réutiliser les mécanismes vus dans la section précédente. Pour lier les fichiers `toto.c` et `toto.o`, il suffit de créer les propriétés `toto.c:` et `toto.o:`, de lier comme vu précédemment chaque fichier à sa

propriété, puis d'assigner à chaque fichier la propriété correspondant à l'autre fichier, par exemple avec les commandes `mv toto.c toto.o:/` et `mv toto.o toto.c:/`.

On aimerait parfois affiner cette *relation* qui unit ces deux fichiers en y affectant une *étiquette*. On aimerait par exemple indiquer que ces deux fichiers sont liées par la notion de compilation. Il suffit pour cela d'utiliser la possibilité offerte par LFS d'avoir des attributs valués. Ainsi, le fichier `toto.o` pourrait par exemple être mis dans l'extension de la propriété plus précise `toto.c:compilation`, de même que `toto.c` dans l'extension de `toto.o:compilation`. Une fois ces liens et assignations de propriétés créés, d'un répertoire contenant le fichier `toto.c` (ou la page `man` de `chmod`), l'utilisateur pourra avec la commande `cd meta` passer dans un répertoire dont le `pwd` logique sera $toto.c$: pour ainsi découvrir les fichiers liés à ce fichier, par exemple `toto.o` (ou la page `man` de `chown`). Il pourra ainsi de proche en proche naviguer de fichier en fichier pour trouver celui qu'il désire finalement. Il faut noter qu'on peut lier plusieurs fichiers à un fichier. En effet, grâce au fait que les attributs valués sont des sous-concepts de l'attribut, suite à la commande `cd meta` LFS proposera comme complément de requête des propriétés mentionnant des étiquettes, par exemple `toto.c:compilation` et `toto.c:mainfile` (pour lier le source d'un module au source du programme principal dont il dépend). Cela permettra de retrouver un de ces fichiers encore plus facilement.

Comme il peut être fastidieux pour un utilisateur d'avoir à créer ces liens, d'assigner les propriétés correspondantes de manière symétrique aux deux fichiers, on peut surcharger la commande `ln` et fournir ainsi un peu de sucre syntaxique pour alléger ce processus. Ainsi, la commande `ln toto.c '<compilation>' toto.o` se charge de faire toutes ces opérations en une seule commande.

5.1.3 Lien propriété-propriété : raccourci

Nous venons de voir qu'il est désormais possible sous LFS de créer des liens entre des fichiers et des propriétés, des fichiers et des fichiers, qu'en est il des liens entre propriétés et propriétés ? A quoi pourrait servir ce genre de liens ?

On a déjà vu un genre de lien entre les propriétés par le biais des axiomes. Cependant, parfois deux propriétés peuvent avoir un lien entre elles sans pour autant que l'on souhaite les ordonner. Par exemple, deux propriétés peuvent être synonymes, ou l'une peut être l'abréviation de l'autre, par exemple `fs` pour `filesystem`, ou encore on peut avoir deux propriétés exprimées dans deux langages différents, par exemple `sea` et `mer`.

On pourrait se satisfaire d'une seule des deux propriétés. Après tout, un des principes de la navigation est de proposer à l'utilisateur des propriétés, ce qui fait que si l'utilisateur ne se rappelle plus le mot qu'il avait choisi à la création du fichier, `fs` ou `filesystem`, le système le lui rappellera. Cependant, les utilisateurs peuvent avoir différentes préférences pour ces noms et il est important d'offrir un support pour pouvoir associer différents noms à la même propriété.

Ainsi, on a besoin d'un moyen pour lier deux propriétés. Nous proposons là encore de surcharger la commande `ln`, qui lorsque elle mettra en jeu deux propriétés, unira ces deux propriétés. Concrètement, suite aux commandes `mkdir filesystem`; `mkdir fs`; `ln filesystem/ fs/`, l'utilisateur pourra librement utiliser l'une ou l'autre propriété dans ses requêtes. Avec ce lien, l'ajout de la propriété `fs` à un fichier se traduira automatiquement par l'ajout aussi de la propriété `filesystem`. Il faut noter que l'on verra ainsi ces deux propriétés proposées comme incréments. De plus, ces deux propriétés n'ont pas forcément besoin d'être des sœurs, c'est à dire d'être deux sous-concepts du même concept. Ainsi, dans le contexte des langues, on pourrait avoir deux taxinomies, une pour chaque langue, reliées entre elles par des liens. Une fonctionnalité supplémentaire de LFS que l'on verra plus loin permettra même à l'utilisateur d'exprimer la sélection d'une langue, ce qui évitera d'avoir à chaque fois les deux types d'incréments proposés, un dans chaque langue.

5.2 Recherche avancée

5.2.1 Interrogation Prolog

Le modèle d'interrogation de LFS, $ext(q) = \{o \in \mathcal{O} \mid d(o) \models q\}$, fait que entre autre,

on voit et traite chaque objet de manière indépendante. En effet, LFS parcourt tous les objets et compare la description de chaque objet, et uniquement celle là, avec la requête. Cette vision des choses a l'avantage d'être facile à comprendre et suffisante pour la plupart des cas. Cependant, les requêtes mettant en jeu plusieurs objets en même temps ne sont pas possibles. Ainsi, la requête `find -newer toto.c` qui permet de chercher les fichiers créés après la date de création du fichier `toto.c` n'a pas d'équivalent sous LFS. Bien sûr, l'utilisateur pourrait récupérer la date de création du fichier `toto.c` manuellement, par exemple le `11/03/2004`, puis lancer la requête `cd ctime:>(11/03/2004)` (en utilisant le moteur logique pour les dates). Cependant, c'est justement ce travail manuel que l'on aimerait éviter. Dans ce cas précis, on aimerait pouvoir récupérer et nommer une valeur d'attribut d'un objet pour ensuite s'en servir pour chercher d'autres objets ; on aimerait faire un hypothétique `cd 'let X = ctime(toto.c) in ctime:>X'`.

Un autre exemple de requête que l'on ne peut pas spécifier sous LFS est la recherche de fichiers musicaux faits par deux artistes, c'est à dire les duos. On peut bien sûr savoir quels sont les fichiers possédant deux artistes précis, en les mentionnant manuellement dans la requête, par exemple `cd artist:beatles/artist:beachboys/`, mais si l'on ne connaît justement pas ces artistes et leurs collaborations, on ne pourra pas exprimer avec LFS la requête.

Le processus d'interrogation de LFS a donc une vison locale et n'a pas connaissance lorsqu'il traite un objet des autres objets, voir même des autres propriétés. Ainsi, en plus du processus de calcul d'extension classique $\{o \in \mathcal{O} \mid d(o) \models q\}$, LFS propose aussi à l'utilisateur une autre forme d'interrogation avec une vision plus globale, plus générale, mais plus complexe, $\{q(\mathcal{O}, \mathcal{P}, d)\}$, à réserver pour les cas plus complexes. En prenant en paramètre le monde, c'est à dire l'ensemble des objets $\mathcal{O}$, l'ensemble des propriétés existantes $\mathcal{P}$, et l'ensemble des descriptions d'objets d, la requête q aura ainsi toute les libertés pour exprimer toutes les requêtes possibles puisque l'on met à sa disposition toutes les connaissances présentes sous LFS.

Cependant, quel sera le langage pour ce nouveau genre de requête ? En effet, une requête ne sera plus comme auparavant une simple conjonction ou disjonction de propriétés ; il faudra entre autre nommer ou compter. Sous quelle forme va-t'on passer en paramètre le monde à ce nouveau genre de requête ? Comment la requête pourra exploiter ce monde ? Au lieu d'inventer un nouveau langage, nous proposons de réutiliser un langage existant : Prolog. Grâce à l'*unification*, Prolog permet facilement de nommer et comparer. Il autorise n'importe quelle forme de traitement étant Turing complet. De plus, le monde passé en paramètre peut être représenté facilement par des faits, et le processus de résolution de Prolog permet de facilement l'exploiter. Enfin, Prolog a une longue histoire en tant que langage d'interrogation de base de données (voir [RU93]).

Concrètement, l'utilisateur par le biais du préfixe spécial `prolog:` pourra formuler des requêtes Prolog. Les requêtes correspondant ainsi aux exemples précédents sont `cd prolog:(name(toto.c,TOTO), ctime(V,TOTO), ctime(W,X), W > V)` et `cd prolog:(artist(V,X), artist(W,X),V!= W)`. La variable `X` est considérée comme spéciale par LFS puisque toutes les valeurs pouvant s'unifier avec ce `X` seront les objets formant la nouvelle extension de ce nouveau `pwd`.

Pour ce qui est du monde, l'idée est donc de transformer par exemple l'assignation d'un attribut valué $x : y$ à un objet o en un fait Prolog $x(y, o)$. Ce schéma de translation peut être étendu aux autres informations et structures présentes sous LFS. On va ainsi réifier ces structures internes en les exposant aux requêtes. Ainsi, l'assignation d'une propriété atomique comme `excellent` à un objet o se traduira par le fait $excellent(o)$, l'axiome $movie \models art$ par la clause `art(X) :- movie(X)`. On peut aussi représenter sous forme de faits le `pwd`. Ainsi, le prédicat unaire `pwd` permet de récupérer les objets du répertoire courant. Ainsi, dans la requête Prolog précédente, l'ajout des conditions `pwd(TOTO), name(toto.c, TOTO)` permet de restreindre la recherche aux fichiers plus récents que le fichier `toto.c` de ce répertoire, et non plus de tous les fichiers `toto.c` présents sous LFS. Enfin, les liens entre propriétés et fichiers vus dans la section précédente peuvent aussi être représentés par des faits. Ainsi, le lien entre le fichier `beatles.info` et la propriété `artist:beatles`, en supposant que l'identifiant interne du fichier soit l'entier `45`, sera représenté par le fait `link(45, artist, beatles)`.

En fait, l'utilisation de l'interrogation Prolog prend tout son sens avec la présence de ces liens. En effet, même si le mot clef `meta` introduit précédemment fournit déjà un premier pas pour formuler des requêtes avancées, il reste limité. Prolog quand à lui permet vraiment un éventail très riche de requêtes. La requête vue dans la section précédente illustrant l'usage de `meta` peut être traduite en Prolog par la commande `cd prolog:(country(england,ArtObj), link(ArtObj,_,ArtProp) , artist(ArtProp, X))`.

En ce qui concerne l'implémentation de cette nouvelle fonctionnalité, nous utilisons un moteur Prolog existant (GNU Prolog). On reproduit un peu la même architecture logicielle choisie pour communiquer avec les transducteurs avancées et les moteurs logique. Ainsi, suite à une requête commençant par `prolog:`, on commence par créer un *pipe* avec un interpréteur Prolog. On analyse la chaîne de caractère représentant la requête pour inférer le type de faits qui seront nécessaires à sa résolution, par exemple tout ce qui touche à l'attribut `name:` ou `ctime:`. On utilise ensuite les structures de données internes de LFS, par exemple la table des extensions `prop->obj`, la table des noms `prop->name`, pour traduire les informations contenues dans ces tables sous la forme de fait selon le schéma de traduction vu précédemment. On passe ensuite sous forme textuelle ces faits au processus Prolog par l'intermédiaire du *pipe* ainsi que pour finir la requête de l'utilisateur. Enfin, on récupère du *pipe* la sortie standard du processus Prolog qui contiendra l'ensemble des valeurs pouvant s'unifier avec la variable `X`, c'est à dire un ensemble d'entiers correspondant à des identifiants internes d'objets. Cet ensemble sert ensuite de nouveau `pwd` pour le nouveau répertoire. L'*inode* de ce répertoire contiendra ainsi dans son champ `pwd` le tag spécial `ExtDyn` avec comme opérande la liste de ces objets.

Il faut noter que suite à une requête Prolog, les autres mécanismes de LFS fonctionnent toujours. Ainsi LFS proposera encore des incréments pour affiner cette requête. C'est uniquement le calcul du `pwd` qui a été temporairement altéré par cette nouvelle fonctionnalité. Il faut noter aussi que c'est encore un mécanisme de déduction logique qui est en jeu. Il s'est juste étoffé puisque l'on passe plus ou moins d'une logique des propositions vers une logique des prédicats. Ainsi, cette nouvelle fonctionnalité s'intègre assez bien avec les autres fonctionnalités présentes sous LFS et l'esprit logique de LFS.

Il faut noter aussi que là encore nous ne défendons pas l'usage des bases de données, et plus précisément ici des base de données déductives, pour gérer les données d'un utilisateur, même si l'on fait se rapprocher LFS encore un peu plus de ces systèmes. Là encore, nous avons juste adapté, voir même intégré certaines des idées présentes dans ces systèmes. Les fonctionnalités qui font l'originalité de LFS comme l'intégration de l'interrogation et la navigation manquent à ces systèmes.

5.2.2 Navigation contrôlée

Même si le principe de navigation par incréments, c'est à dire de proposer des propriétés qui affinent la requête, est souvent le mode idéal pour trouver un fichier, il a parfois quelques inconvénients. Le Chapitre 2 et notamment le long exemple d'utilisation de LFS pointait déjà du doigt certains de ces inconvénients Section 2.3.1 page 73. Nous présentons donc dans cette section de légères variations pour la sémantique de `ls` qui vont toutes agir sur la notion d'incrément.

Des incréments moins stricts

Les transducteurs, et notamment les transducteurs systèmes fournis de base par LFS, permettent d'ajouter automatiquement aux fichiers de nombreuses propriétés dites intrinsèques. Cette profusion de propriétés, même si elle permettra justement ensuite de pouvoir retrouver un fichier à l'aide de nombreux critères, a des effets négatifs sur le résultat de la commande `ls`. Chaque fichier a désormais des propriétés systèmes, sous la forme d'attributs valués. La Figure 5.1 montre un exemple typique d'organisation des fichiers résultant sous LFS. Cela veut dire que le premier `ls` de la racine de LFS devrait lister malheureusement toutes les propriétés de la forme `size:`x, `name:`x, etc. En effet, le système ne répondra pas avec les concepts plus généraux comme `size:`, `name:` ou même `system`, car ces concepts ne raffinent pas la requête puisque tous les fichiers ont ces propriétés ; la seule chose qui différencie ces fichiers sont les valeurs de ces attributs.

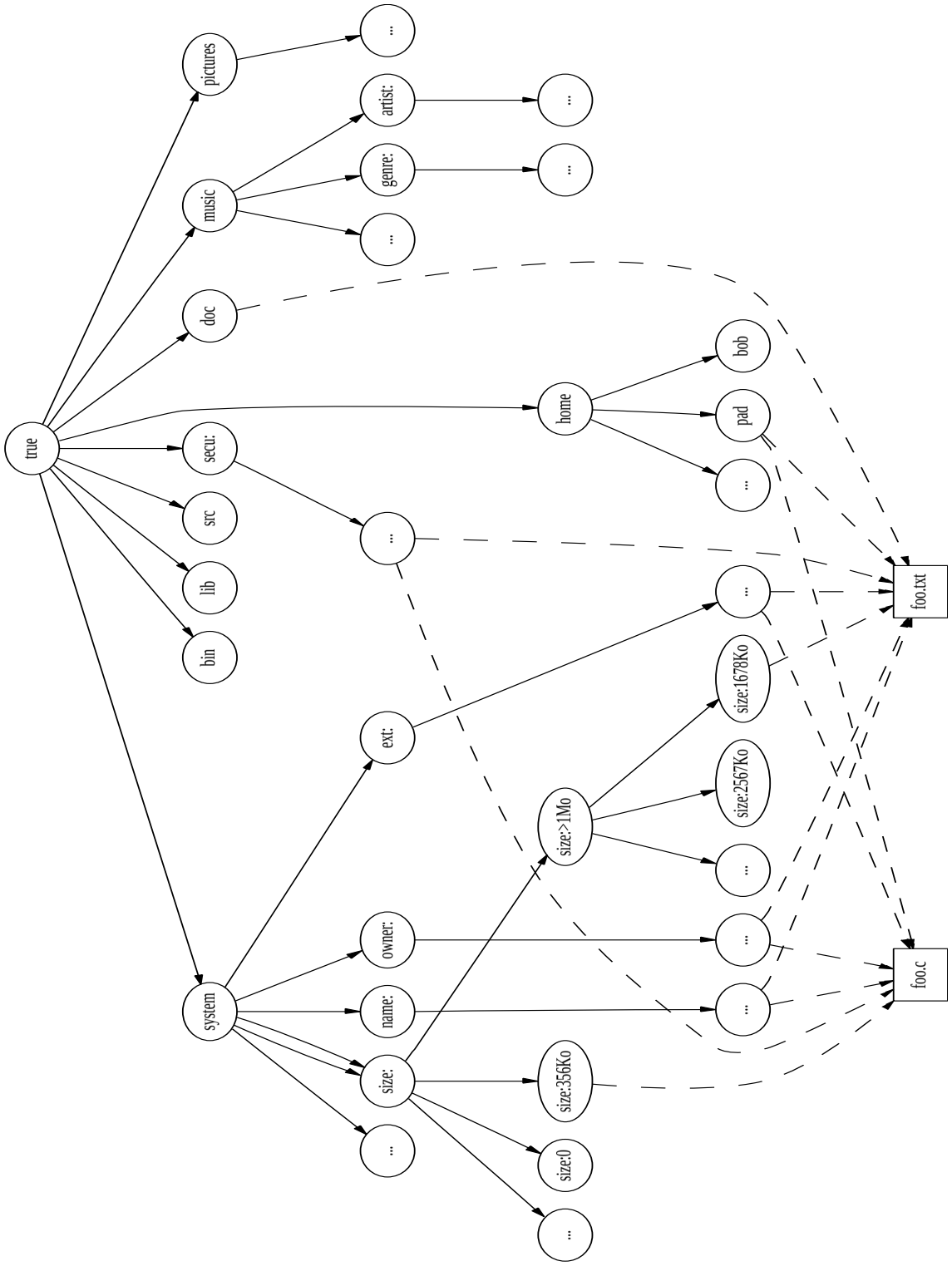

FIG. 5.1 – Une hiérarchie typique de propriétés LFS

Un utilisateur préférerait cependant la deuxième solution afin d'éviter d'être noyé sous un trop grand nombre de propriétés. Pour cela, l'utilisateur peut décider de «relaxer» la condition pour être un incrément avec la commande `cd .relaxed`. La sémantique de `ls` pour le calcul d'incréments, c'est à dire des sous-répertoires, devient alors :

$Dirs = max_{\models}(\{p \in \mathcal{P} \mid \neg(pwd \models p), \emptyset \subset ext(p \wedge pwd) \subseteq ext(pwd)\})$.

La principale nouveauté est que l'inégalité la plus à droite n'est plus stricte. Avec ce schéma, la commande `ls` à la racine listera donc uniquement les grosses catégories. La contrainte $\neg(pwd \models p)$ permet d'avancer en évitant de reproposer indéfiniment les mêmes répertoires. En effet, sans cette contrainte le système proposerait encore la propriété `size`: dans le répertoire `size`:. Avec cette contrainte, le système proposera désormais les sous-concepts de `size`: comme `size`:`0`, ce qui permet d'une certaine manière de développer une branche.

Les deux sémantiques sont proposées à l'utilisateur puisque chacune d'elles peuvent être la plus appropriée selon la situation. Concrètement, on rajoute un champ dans la structure de l'*inode* qui spécifiera la sémantique choisie. Elle pourra être changée par l'utilisateur en exécutant soit la commande `cd .relaxed`, soit la commande `cd .strict`.

Sélection d'incréments

Même si la fonctionnalité précédente permet de mieux contrôler les incréments et le nombre des sous-répertoires proposées, et même si elle permet de ne développer que certaines branches, il reste que l'utilisateur voit tout de même les départs de toutes les branches. C'est certes souvent pratique, puisque l'on ne se ferme pas de possibilités, mais il arrive parfois qu'un utilisateur sache précisément le type de propriétés qu'il veut mentionner dans sa requête, par exemple des propriétés de type musicale. Le danger avec la profusion de propriétés est que beaucoup d'entre elles sont simplement du bruit lorsque l'on recherche un certain type de fichier en ayant déjà en tête un point de vue sur ce fichier. On aimerait donc contrôler encore un peu plus ce bruit.

Une nouvelle fonctionnalité, la *sélection d'incréments*, permet de contraindre l'ensemble des propriétés que l'algorithme `ls` peut retourner comme incréments possibles. L'utilisateur déclare une sélection en ajoutant le symbole spécial `^` (un chapeau) à la fin du nom d'une propriété. Cela indique que les futurs `ls` listeront uniquement comme incréments des propriétés qui sont des sous-concepts de cette sélection. Par exemple, un utilisateur cherchant un fichier musical et intéressé uniquement par des propriétés liées à la musique, peut utiliser la commande `cd music/^`. Ainsi, dans le reste de la navigation, ne seront proposés comme sous-répertoires que des propriétés comme `artist`:`Beatles` ou `genre`:`Disco`. Les autres concepts, comme `size`:x ne viendront donc pas polluer cette recherche. D'une certaine manière, le `pwd` sélectionne le genre d'objets que l'on recherche, et la sélection le genre d'incréments que l'on veut utiliser dans la navigation.

Concrètement, on rajoute encore un champ dans l'*inode* qui contiendra une référence vers la sélection courante, c'est à dire vers une propriété. Cette sélection peut être changée avec la commande `cd` et donc dans l'opération système `lookup`, qui lorsque la chaîne passée en paramètre correspondra à l'expression régulière `.*^` pourra ajuster ce champ en conséquence. L'algorithme `ls` est aussi affecté, et prend tout simplement en paramètre cette propriété qui représentera le nouveau point de départ pour la recherche dans le graphe. Ainsi, par défaut, on passera a l'algorithme la propriété `true` correspondant à la racine lorsqu'aucune sélection n'aura encore été faite, ce qui permettra donc de retrouver le comportement habituel pour `ls`.

Pas d'incrément : réponse extensionnelle

Sous un système de fichier classique, un utilisateur peut récupérer l'ensemble des fichiers présents dans un répertoire et ses sous-répertoires (c'est à dire son *extension*) à l'aide de la commande `ls -R` où avec la commande `find`. Ces commandes traversent récursivement tous les répertoires d'un sous-ensemble de la hiérarchie. On peut utiliser ces commandes parce qu'on estime que l'on a suffisamment navigué et que l'on sera capable d'identifier le fichier que l'on désire à partir de cette longue liste de fichiers. On peut aussi utiliser ce processus récursif dans d'autres commandes pour pouvoir effectuer le même traitement sur un ensemble de fichiers, par exemple les effacer avec la commande `rm -R`. On

retrouve ces mêmes besoins sous LFS et l'on préférait donc parfois récupérer un ensemble d'objets plutôt que des incréments. Cependant, sous LFS, même si un utilisateur peut lancer les commandes `ls -R` ou `rm -R`, leurs temps d'exécution pourront être très long. En effet, la traversée récursive sera coûteuse car il y a de très nombreux chemins pouvant mener au même endroit, au même fichier, du fait de toutes les combinaisons possibles de propriétés. Par exemple, avec un fichier ayant 5 propriétés, il y a déjà 120 chemins possibles menant à cet unique fichier. La commande `ls -R` est impraticable et de plus aurait des résultats dupliqués.

Paradoxalement, en interne, calculer l'extension d'un répertoire est très rapide du fait de la présence de la table `prop->obj`, la table des extensions. Ainsi, pour être capable de récupérer l'extension d'une requête, on étend une nouvelle fois le langage d'interrogation en y ajoutant un nouveau mot-clef spécial `.ext`. L'utilisateur exécutant la commande `cd .ext/; ls` récupérera ainsi l'extension du répertoire courant.

Concrètement, on utilise la même technique d'implémentation que précédemment en ajoutant un champ dans l'*inode* indiquant si l'on est ou pas dans «le mode extension».

5.2.3 Navigation hypertexte

Lorsque l'on navigue à l'intérieur d'un fichier, suite à la commande `cd parts`, les vues de ce fichier peuvent contenir des marques d'absence qui cachent certaines parties du fichier. Ces marques sont utiles comme on l'a vu Section 2.6.2 page 83 car ce sont elles qui rendent possible la modification de ces vues. De plus, cacher certaines parties du fichier est utile puisque cela permet entre autre à l'utilisateur de se concentrer plus facilement sur sa tâche. Cependant, un utilisateur aimerait parfois voir ce qu'il y a à l'intérieur de ces parties cachées. Ainsi, «être la partie cachée numéro x» est considéré comme une propriété. L'utilisateur est autorisé à spécifier une vue avec la requête `cd mark:`x`/` pourvu que x soit un numéro de marque présent dans la vue courante. Ces numéros, en plus de permettre de déplacer ou d'effacer des parties cachées, permettent aussi de pouvoir les référencer. Ainsi, dans l'exemple de la Section 2.3.2 page 77, une autre suite de commandes *shell* utilisant LFS pourrait être :

```
[111] % cd mark:1
[112] % ls
foo.c
[113] % cat foo.c
................:1
fprintf(stderr, "x = %d", x);
................:2
[114] % cd mark:1
[115] % ls
specification/
var:x/ var:y/
foo.c
[116] % cat foo.c
int f(int x) {
int y;
assert(x > 1);
y = x;
................:1
```

Cela permet de naviguer encore plus dans le contenu du fichier, et ajoute à LFS les avantages des systèmes hypertexte. On implémente cette fonctionnalité en utilisant la table `i.marks` pour calculer l'ensemble des objets `ois` qui appartiendront à la nouvelle vue. On shunte ainsi temporairement le mécanisme d'interrogation qui est normalement le responsable de ce calcul d'objets. Il faut noter que le répertoire `mark:1/` n'a rien de spécial comme l'indique le résultat de la commande `14`. En effet, comme dans tous les autres répertoires de LFS, le système propose des incréments. Cette fonctionnalité hypertexte s'intègre ainsi très bien avec le reste des fonctionnalités présentes dans LFS.

5.3 Manipulation avancée

5.3.1 Manipuler des groupes de fichiers

Certaines applications requièrent que l'information soit placée dans différents fichiers. C'est le cas par exemple avec les environnements Java qui forcent le programmeur à placer le code de chaque classe dans un fichier différent à chaque fois. En fait, même si l'exemple du Chapitre 2 n'illustre pas cet aspect, l'utilisateur peut «rentrer» dans un ensemble de fichiers suite à la commande `cd parts`. Si le répertoire dans lequel l'utilisateur a lancé la commande `cd parts` contenait un en-

semble de fichiers, par exemple `foo.c` et `bar.c`, LFS indexera avec les transducteurs avancés l'ensemble de ces fichiers. LFS générera ensuite dans chaque répertoire une vue par fichier d'origine, ici des vues ayant pour nom `foo.c` et `bar.c`. En fait, LFS ne listera une vue avec `ls` que si elle n'est pas vide. Ce mode favorise ainsi la découverte du rôle de ces différents fichiers ; par exemple qui définit quoi ? qui utilise quoi ? Ainsi, le répertoire `.../parts/function:f` ne contiendra que la vue `foo.c` car seul ce fichier possède une fonction `f`.

Un des objectifs de LFS était de pouvoir manipuler facilement les différents aspects de l'information que contient un fichier, aspects pouvant être disséminés un peu partout dans le fichier. C'est ainsi que LFS permet entre autre à l'utilisateur de regrouper ensemble toutes les conclusions d'un document pour pouvoir mieux les éditer. Cependant, il arrive aussi qu'un aspect de l'information soit disséminé dans différents fichiers, par exemple lorsque des applications obligent l'utilisateur à séparer son projet en différents fichiers. Ainsi, différentes fonctions dans différents modules peuvent utilisées la même variable globale et l'on aimerait avoir une vue regroupant toutes ces fonctions. Ce n'est cependant pas possible sous LFS suite à la commande `cd parts`.

Ainsi, nous avons ajouté à LFS la possibilité de vraiment pouvoir manipuler un groupe de fichiers avec la commande `cd globalparts`. Dans ce mode, LFS calcule des incréments comme d'habitude, mais ne génère qu'une seule vue `globalview` qui contient toutes les parties de tous les fichiers qui satisfont la requête. Un nouveau genre de marque est requis pour indiquer à l'utilisateur quel fichier sera modifié, pour éviter là encore toute ambiguïté. Ainsi, dans l'exemple de la Section 2.3.2 page 75, en supposant l'existence des fichiers `bar.c` et `foo2.c`, une autre suite de commandes *shell* utilisant LFS pourrait être :

```
[102] % ls
foo.c bar.c foo2.c
[103] % cat bar.c
 int f(int o) {
  return o + 2;
 }
[103] % cat foo2.c
 int f2(int o) {
  return o + 3;
 }
[104] % cd globalparts
[105] % cd function:f/!debugging/
[106] % cat globalview
.................:foo.c
int f(int x) {
int y;
assert(x > 1);
y = x;
.................:1
return y * 2
}
.................:2
.................:bar.c
int f(int o) {
  return o + 2
}
```

Cette fonctionnalité va aussi loin que possible dans la virtualisation des fichiers puisqu'elle abolit même la notion de frontière entre fichier.

5.3.2 Manipuler les propriétés

On aimerait parfois voir toutes les propriétés que possède un fichier, tout comme on aime parfois sous un système de fichier classique voir les propriétés systèmes d'un fichier avec la commande `ls -l`. Ce besoin est d'autant plus important que des propriétés systèmes supplémentaires ont été ajoutées sous LFS avec les propriétés représentant le statut de protection d'un fichier.

Bien sûr, le chemin qui mène à un fichier traduit déjà la description du fichier puisque les propriétés présentes dans ce chemin impliquent forcément la description du fichier ; mais il ne le traduit qu'en partie. En effet, il n'est pas utile sous LFS de mentionner toutes les propriétés d'un fichier pour spécifier de manière non ambiguë ce fichier. Ainsi, la seule manière pour inférer la description d'un fichier est de tester si ce fichier est présent ou non dans l'extension d'une propriété en lançant à chaque fois les commandes `cd prop/; ls`.

Nous ajoutons à LFS un nouveau mot clef `.int` qui permet de voir la description d'un fichier. Suite à la commande `cd .int/` dans un répertoire contenant un fi-

chier, LFS proposera non plus des incréments, mais des répertoires dont les noms correspondent aux propriétés de ce fichier.

Nous proposons aussi que dans un répertoire `.int/`, un utilisateur puisse utiliser les commandes `mkdir` et `rmdir` avec une sémantique différente. Le but ne serait non plus d'ajouter ou d'effacer un concept ou sous-concept, mais d'ajouter ou d'enlever une propriété à la description du fichier, ce qui peut s'avérer parfois plus pratique que d'utiliser la commande `mv`.

En fait, un utilisateur peut utiliser la commande `cd .int/` de n'importe quel répertoire, pas uniquement dans un répertoire contenant un seul fichier. La sémantique de cette commande est en fait d'afficher l'intention, d'où le nom `.int`, la plus spécifique d'un répertoire, c'est à dire l'ensemble des propriétés partagées par les fichiers de ce répertoire. Dans le cas ou ce répertoire ne contient qu'un fichier, cela revient en effet à afficher la description du fichier. C'est un peu donc le pendant de la commande `cd .ext/` qui permettait de voir l'extension d'un répertoire. Cette fonctionnalité permet de bénéficier un peu des avantages des systèmes de *fouilles de données* (*data mining*) puisque l'on peut inférer des relations entre les propriétés. Par exemple, les commandes `cd /artist:Beatles/.int/; ls` montreront que l'ensemble des fichiers musicaux fait par les Beatles appartiennent tous au même genre musical qui est le Rock, ce que `cd /artist:Beatles/; ls` ne montrera pas puisque la propriété `genre:Rock` ne sera même pas proposée puisqu'elle ne raffinerait pas la requête.

5.4 Conclusion

Une des contributions de ce chapitre est d'avoir encore une fois, comme précédemment avec la sécurité, su adapter un concept des systèmes de fichier, le lien, et même su exploiter à bon escient ce concept dans le nouveau contexte LFS. La possibilité d'établir des liens entres fichiers et propriétés, et d'établir des relations entre fichiers, permet d'augmenter l'expressivité de LFS de manière importante. L'utilisateur a désormais la possibilité d'effectuer des recherches par l'exemple et peut tirer profit de la variété des données qui peuvent se trouver ensembles sous LFS.

Troisième partie

Évaluation

Chapitre 6

Expérimentations

No one believes an hypothesis except its originator, but everyone believes an experiment except the experimenter.

W. I. B. Beveridge

There's more to science than just hurting small animals, but it's the part that's the most fun.

Scott Adams (Dilbert)

Nous allons dans ce chapitre présenter de nombreuses expériences faites avec un prototype de LFS, visant à démontrer la viabilité de l'approche LFS pour la gestion de l'information, à la fois en terme d'efficacité et d'utilité.

Nous allons tout d'abord décrire plus précisément l'implémentation concrète de ce prototype. Cela complétera ainsi les explications données au Chapitre 3 sur les algorithmes et structure de données utilisés par LFS, en se focalisant cette fois sur l'*architecture logicielle* du prototype LFS.

Nous présenterons ensuite une suite d'expériences où l'on aura utilisé LFS pour gérer des données de tous les jours (fichiers musicaux, e-mails, documents). Nous montrerons à travers des exemples d'utilisation, des exemples de requêtes, la viabilité de LFS en terme d'utilité. Nous montrerons ensuite à l'aide de benchmarks la viabilité de LFS en terme d'efficacité, à la fois en espace disque et en vitesse de réponse. Comme il n'y pas d'autres systèmes de fichier offrant l'ensemble des services que propose LFS, nous évaluerons donc surtout le surcoût qu'apporte LFS par rapport aux systèmes de fichier classiques. Nous présenterons aussi des expériences où l'on évaluera la performance de LFS sur des tâches pour lesquelles LFS a été conçu, comme par exemple la recherche d'information. Dans ce cas, nous comparerons LFS avec des applications offrant le même genre de service, par exemple la commande `find`.

Nous présenterons aussi des expériences plus «scientifiques», où l'on fera varier isolément différents paramètres, comme le nombre de fichiers ou le nombre des propriétés par fichier. Ces expériences permettront de se faire une meilleure idée sur la complexité des opérations de LFS : comment leurs coûts évoluent, de manière quadratique ? linéaire ? en fonction de ces paramètres, et quelles sont les limites actuelles du prototype LFS.

Enfin, nous finirons en présentant une suite d'expériences permettant d'évaluer la pertinence des optimisations présentées dans cette thèse, comme l'utilisation d'un cache logique ou la représentation d'ensembles par intervalles. Nous présenterons donc des benchmarks réalisés avec des prototypes LFS ne bénéficiant pas de toutes les optimisations. En effet, certaines optimisations peuvent parfois paraître sur le papier intéressantes mais en pratique se révéler inutiles, ou pas suffisantes pour justifier leur implémentation puis leur maintenance. Ce n'est pas le cas pour les optimisations présentes dans LFS, qui comme nous le verrons améliorent à chaque fois de manière significative les performances de LFS.

6.1 Implémentation du prototype

Il existe de nombreuses façons d'implémenter un système de fichier (voir Section 1.6.2 page 47). Le prototype actuel LFS est un système de fichier tournant au *niveau*

utilisateur. Nous avons donc choisi de ne pas placer le code de LFS directement dans le noyau. LFS est donc constitué essentiellement d'un processus «normal» qui agit comme une sorte de *serveur*. Une petite partie de code placée dans le noyau est chargée de rediriger toutes les requêtes systèmes concernant le système de fichier à ce serveur. Cette communication entre le noyau et ce serveur se fait à travers une sorte de *pipe* via un pseudo-périphérique dédié.

Ce style d'implémentation est pratique pour prototyper. En effet, le déboggage d'un programme «normal» est plus facile que le déboggage d'un noyau. De plus, ce programme peut être codé dans un langage de plus haut niveau que le C, peut réutiliser d'autres programmes, et comme on le verra plus loin peut même réutiliser d'autres systèmes de fichier. Une autre raison pour ce choix vient du fait que comme LFS permet à l'utilisateur de fournir des *plug-ins*, il faut de toute façon un moment ou à un autre faire communiquer le noyau avec des programmes externes.

6.1.1 Organisation du code

Il existe de nombreux systèmes aidant à l'implémentation d'un système de fichier tournant au niveau utilisateur. On peut par exemple modifier un serveur NFS ou encore fournir un module C++ au système UserFS [Fit96]. Nous avons choisi d'utiliser plutôt le système PerlFS [Cal01] qui possède une interface d'utilisation plus simple, et qui permet de coder en Perl, un meilleur langage que le C ou le C++ pour le prototypage. Comme pour UserFS, il suffit de fournir un module, cette fois en Perl, implémentant un certain nombre de fonctions. Ces fonctions portent les mêmes noms que les opérations du VFS.

Le tout premier prototype LFS était donc codé en Perl. Dans un second temps nous avons décidé de réécrire le prototype dans un langage compilé et plus sûr avec un typage statique : O'Caml [LDG+04]. Comme c'est souvent le cas dans la réalisation d'un logiciel (voir le livre de Brooks [Bro75]), la réécriture permet de repartir sur de meilleures bases grâce à l'expérience acquise sur le premier prototype. Le code de ce second prototype est ainsi plus clair. De plus, l'utilisation de O'Caml a permis non seulement grâce au typage d'avoir une confiance dans le code beaucoup plus grande avec une meilleure stabilité, elle a permis aussi d'améliorer de manière significative les performances. En effet, O'Caml étant un langage compilé, et ce en mode natif, la vitesse du prototype O'Caml est à peu près 10 fois supérieure au prototype Perl sur l'ensemble de nos expériences (avec un code O'Caml plus clair mais au fond sensiblement identique au code Perl du premier prototype ; cette amélioration n'est donc pas due à de meilleurs algorithmes mais bien juste à l'utilisation d'un langage plus efficace).

Cependant, même si il existe un équivalent à PerlFS pour O'Caml (MLFUSE pour *ML Filesystem in User-SpacE*), ce système ne possède pas encore toutes les fonctionnalités et la stabilité de PerlFS. Nous avons donc décidé de garder le système PerlFS. Le module Perl se charge juste de rediriger à travers un *pipe* (encore une fois) les requêtes au processus O'Caml. Il y a donc une double indirection puisque les requêtes viennent du noyau vers le serveur Perl par une sorte de *pipe*, puis du serveur Perl au serveur O'Caml encore une fois par un *pipe*.

Le *pipe* est donc le moyen de communication privilégié. Il sert aussi pour la communication entre LFS et les différents *plug-ins*. La Figure 6.1 illustre l'interaction de l'ensemble de ces acteurs : le noyau, PerlFS, le programme LFS en O'Caml, et les *plug-ins*.

Le système peut paraître compliqué, et même pour de mauvaises raisons, du fait de l'interaction de l'ensemble de ces acteurs. On pourrait penser qu'il serait préférable «d'éliminer de l'équation» PerlFS, ou de le remplacer par un CamlFS pour ne garder plus qu'un seul *pipe*. Cependant, le choix de garder PerlFS, un peu comme le choix de réécrire le premier prototype, s'est avéré payant. En effet, le fait d'avoir deux serveurs nous a forcé à séparer clairement différentes préoccupations. Ainsi, le serveur O'Caml contient les algorithmes (Section 3.1) et structures de données principales (Section 3.2.1), et le serveur Perl contient lui les détails comme les spécificités d'interface liées au système Unix et au VFS (Section 3.2.2). D'un point de vue génie logiciel, le programme O'Caml est ainsi plus propre. C'est d'ailleurs cette partie, la plus intéressante, qui est exposée en Annexe B. Par exemple, la partie Perl implémente une fonction `readdir` prenant en paramètre l'*inode* d'un répertoire et un offset `i` et doit retourner la ième entrée de ce répertoire. La partie O'Caml ne possède que la simple fonction `ls` qui prend en paramètre une chaîne de carac-

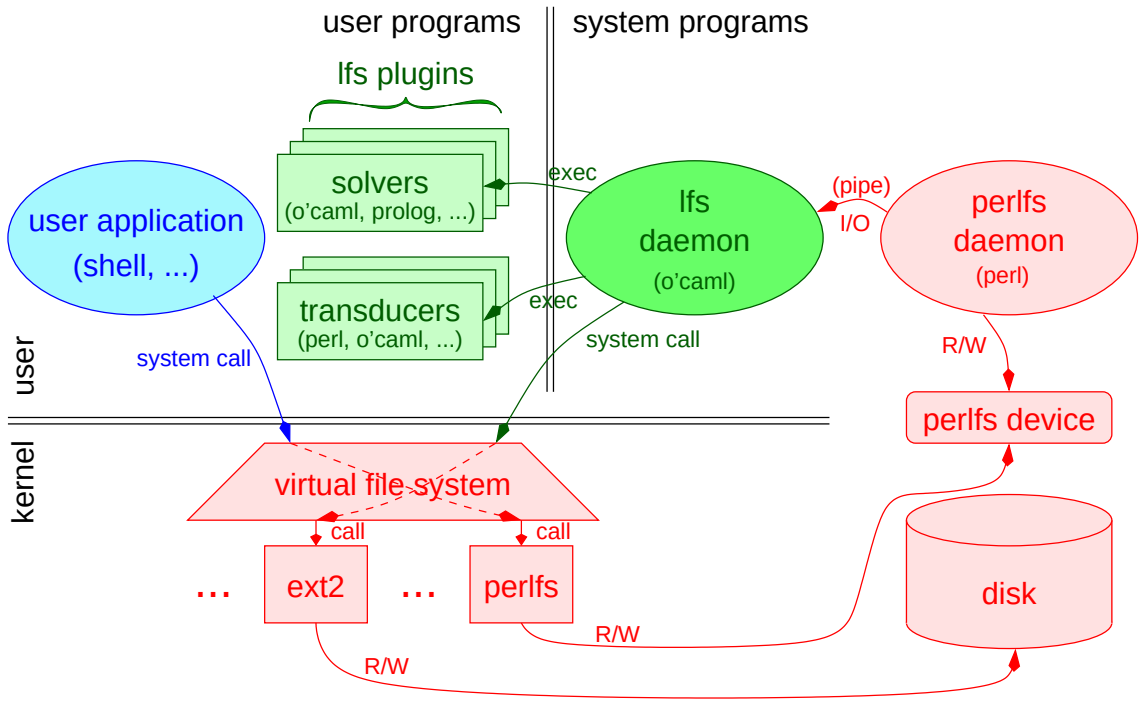

FIG. 6.1 – Architecture logicielle LFS

tère représentant le chemin d'une requête et qui retourne la liste des sous-répertoires et fichiers de ce répertoire. La partie O'Caml est plus «carrée» et la partie Perl permet de faire rentrer ce truc carré dans un truc rond, le VFS. Cette organisation rend encore plus facile le déboggage et le test puisque l'on peut débogguer ou tester isolément la partie O'Caml dans un monde là aussi plus propre. De plus, même si le prototype actuel LFS ne tourne que sous Linux, le fait que la partie O'Caml ne contienne aucun détail lié au VFS, et le fait que cette partie constitue l'essentiel du code, nous laisse à penser qu'il serait très facile de porter LFS sur un autre système d'exploitation. Ainsi, nous pensons que le système est paradoxalement plus simple du fait de ces différents acteurs.

Il faut noter que même si le style d'implémentation choisi pour LFS est différent des autres systèmes de fichier classiques comme EXT2, cela reste un vrai système de fichier. On peut ainsi par exemple l'exporter par NFS. Cela permet du coup de rendre accessible LFS à d'autres systèmes d'exploitation, comme Windows ou Mac, à travers le réseau. On voit là encore l'intérêt de placer un service au plus bas niveau, le niveau système, puisque l'on peut bénéficier ensuite de tous les services du système gratuitement sans rien avoir à coder.

6.1.2 Organisation des structures de données

En ce qui concerne la gestion des données, le choix d'un système de fichier tournant en mode utilisateur permet là encore de prototyper plus facilement : nous utilisons les services offerts par les systèmes de fichiers existants, dans notre cas EXT2 [CTT94]. En effet, même si les algorithmes et structures de données utilisés par LFS sont très différents de ceux utilisés dans les systèmes de fichier classiques, il n'en reste pas moins qu'il faut stocker ces données sur un disque, gérer des contenus de fichiers ou de vues, et donc gérer des blocs. Cette partie n'est pas différente des autres systèmes et il serait fastidieux de la recoder sous LFS. Nous avons donc *délégué* toute la partie gestion de blocs au système de fichier sous-jacent.

La Figure 6.2 montre l'organisation des données LFS internes sur le disque. Ainsi, plutôt que d'adresser les données par numéro de secteur, LFS utilise le service d'adressage par chemin qu'offre les systèmes de fichier hiérarchiques. Plutôt que d'allouer des blocs, LFS crée tout simplement des fichiers. Par exemple, lorsque LFS a besoin d'accéder en interne au fichier ayant 23 comme numéro d'identifiant, il lui suffit d'accéder au fichier `/meta_lfs/files/23/data`. Un peu comme les systèmes de fichiers hiérarchiques apportent de nouveaux services à l'utilisateur tout en se basant en interne sur les services plus bas niveau offert par un disque (comme l'adressage de blocs), LFS apporte aussi de nouveaux services à l'utilisateur tout en se basant en interne sur les services de plus bas niveau offerts par les systèmes de fichier hiérarchiques.

Pour la gestion des différentes tables (les méta-données) nous utilisons la librairie Berkeley DB [OBS99] [1]. Cette librairie permet des accès *associatifs* (clef-valeur) rapides sur des données stockées sur le disque. Selon les tables, les clefs correspondent à des identifiants internes d'objets ou de propriétés, ou bien à des noms de propriétés. Ces données *persistentes* sont stockées sous la forme de *B-tree* [Com79]. En effet, on ne peut se permettre de stocker toutes ces associations en mémoire ; Berkeley DB allie à la fois l'efficacité mémoire, l'efficacité CPU, et la persistence.

On pourrait utiliser la même technique que pour les fichiers, en créant de manière similaire au répertoire `/files/22` un répertoire `/properties/22`. Cependant, ces méta-données et leurs accès sont plus critiques pour l'efficacité du système que ne l'est l'accès au contenu des fichiers. De plus, le module *transactionnel* de Berkeley DB fournit l'outillage nécessaire pour éviter la corruption de ces méta-données suite à un crash du système (voir Section 3.3 page 109). C'est une fonctionnalité essentielle du système puisque c'est ce que l'on attend avant tout d'un système de fichier. Le prototype LFS est donc aussi un système de fichier *journalisé* (voir Section 1.6.1 page 43). Il faut noter que bien que EXT2 soit lui même un système de fichier journalisé, cela ne suffit pas pour faire de LFS un système de fichier journalisé. En effet, EXT2 ne permet que d'éviter la corruption de sa propre structure interne. Berkeley DB permet d'éviter la corruption des structures internes LFS avec le contenu

[1] Et la librairie `ocamlbdb` [Ste02] qui permet d'y accéder à partir d'un programme O'Caml.

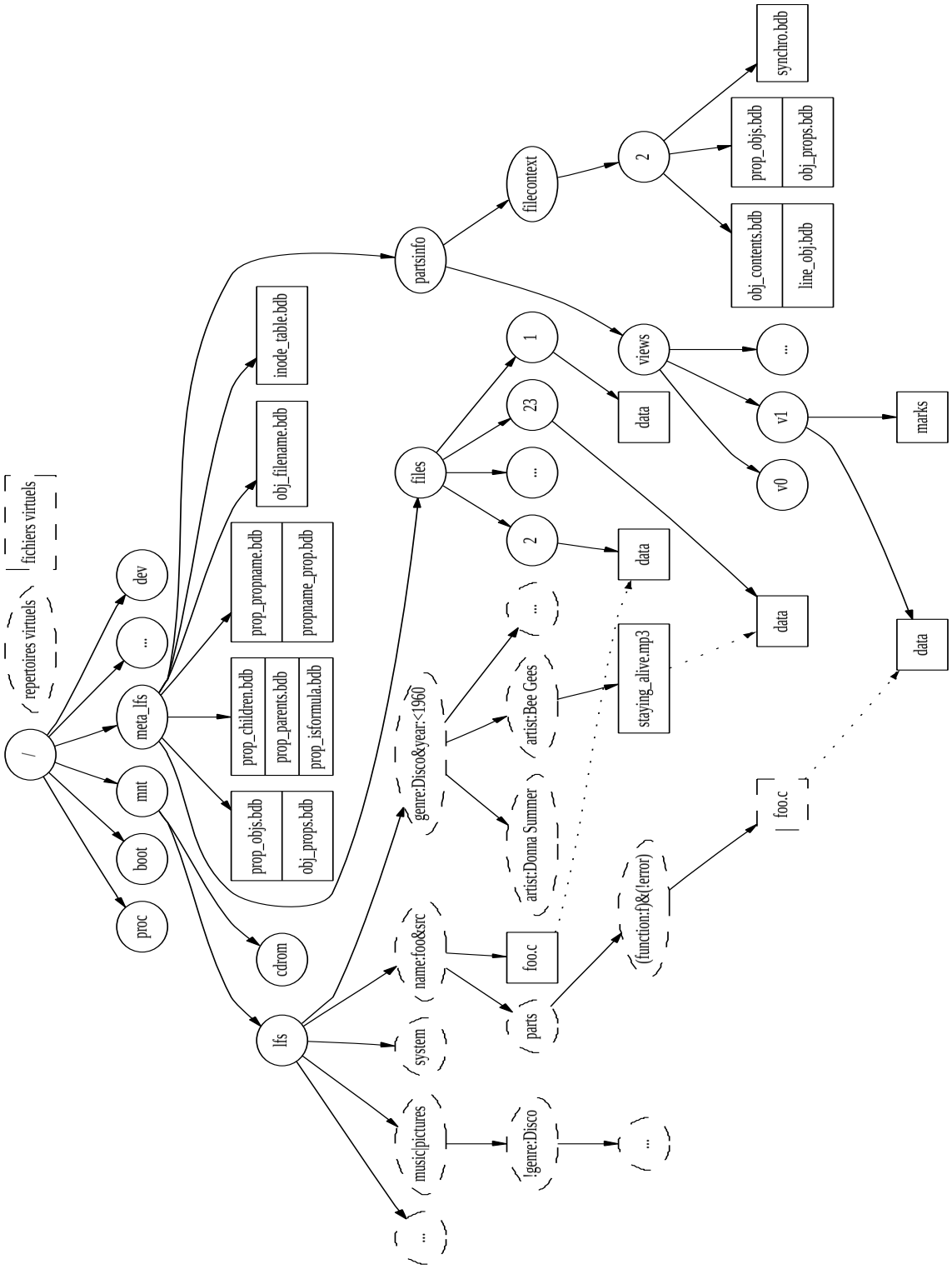

FIG. 6.2 – Organisation interne des données

des tables et les liens qui unissent ces tables[2].

La Figure 6.2 montre aussi l'ensemble des fichiers Berkeley DB utilisés par LFS. Chaque fichier correspond à une table vue Section 3.2 page 106 qui présentait les structures de données LFS. Par exemple, le fichier `/meta_lfs/obj_props.bdb` correspond à la table `obj->props`. Cette table est indexée par un numéro d'identifiant interne d'objet et retourne une structure avec les propriétés extrinsèques et intrinsèques de ce fichier. De même, pour récupérer par exemple l'extension de la propriété `size:100Mo` LFS récupère d'abord l'identifiant interne de cette propriété en cherchant la valeur associée à la clef `size:100` dans la table `/meta_lfs/propname_prop.bdb`. Il ne reste plus qu'à se servir de cet identifiant comme clef dans la table `/meta_lfs/prop_objs.bdb`.

C'est donc sur EXT2 et Berkeley DB que repose LFS pour la gestion des données. Il faut noter que la partie valeur dans ces tables est presque tout le temps de taille variable, par exemple l'extension d'une propriété peut être plus ou moins grande. Comme nous l'avons vu dans la partie bibliographique, ces problèmes de variation de taille, par exemple de taille de fichiers, de taille d'*inode*, de taille de blocs d'adresse, sont importants pour l'implémentation d'un système de fichier et ont conduit par exemple à l'invention de notions comme les blocs d'adresse indirects (voir Section 1.6.1 page 38). Nous n'avons cependant pas à nous en soucier puisque c'est Berkeley DB et EXT2 qui gèrent automatiquement pour LFS tous ces problèmes d'allocation de blocs.

Discussions

Comme déjà dit auparavant, certains choix ont été guidés par le besoin de prototypage, c'est à dire de pouvoir coder rapidement en réutilisant au maximum les services déjà présents dans le système. Certains choix introduisent cependant de nouveaux surcoûts qui viennent se rajouter aux surcoûts déjà apportés par les nouveaux services et calculs de LFS. Ainsi, le choix d'un système de fichier tournant en mode utilisateur implique le surcoût lié à la communication entre le noyau et un processus à travers un *pipe*. Le choix d'utiliser EXT2 pour stocker les données implique le surcoût lié à l'interprétation des chemins, et une utilisation de l'espace disque qui n'est pas forcément optimale. Dans les deux cas, pour le code et pour les données, l'ajout d'intermédiaires dans la chaîne introduit forcément des coûts supplémentaires. On aimerait ainsi pour améliorer encore les performances placer le code LFS directement dans le noyau et utiliser directement le disque. Cependant, nous pensons que cela n'aurait que peu d'impact sur les performances du système.

En effet, en ce qui concerne les données, une implémentation de plus bas niveau, avec une politique d'allocation de blocs plus élaborée, avec des optimisations comme le rapprochement des méta-données des données sur le disque (voir Section 1.6.1 page 40), n'améliorent souvent les performances du système uniquement que de quelques pourcents. Ces optimisations jouent essentiellement sur la minimisation des temps de déplacement de la tête de lecture du disque dur. Pour LFS, le *goulot* n'est pas tellement sur le disque mais sur le CPU (voir Section 6.3.5 page 171) car l'originalité de LFS se trouve dans les calculs qu'impliquent les nouveaux services de LFS (comme le calcul d'incréments). Il y a donc de nombreuses optimisations algorithmiques à faire avant d'arriver à ces optimisations visant à gagner les derniers pourcents.

Pour ce qui concerne le code, et plus précisément les surcoûts liés à la communication à travers les *pipe*, là encore nous pensons que le *pipe* n'est pas le goulot mais que c'est bien le CPU. En effet, si la fréquence des appels systèmes est faible par rapport au temps mis par LFS pour répondre à chacun de ces appels, alors le surcoût lié à la communication des questions-réponses est négligeable sur le temps total. Or c'est précisément le cas puisqu'à la fois l'utilisateur ne lance pas des millions de commandes `ls` à la seconde, et LFS et le CPU mettent effectivement du temps à répondre à chacune de ces commandes.

En fait, comme on le verra plus loin Section 6.3.2 page 164, le *pipe* peut s'avérer un frein important à la performance uniquement lors d'opérations mettant en jeu des entrées-sorties intensives (comme la copie de gros fichiers avec des temps en moyenne 2 fois plus long). En effet, cela implique un nombre important de suites de `read` et `write`, avec en paramètre à chaque fois un petit morceau du fichier. Cependant, le

[2] Pour une fois une fonctionnalité du système de fichier n'irradie pas sur les applications.

code LFS de ces opérations n'a rien d'original ; il appelle juste les mêmes opérations sur EXT2. Seules les opérations encadrant ces appels sont originales (avec `open` qui peut créer une vue et `release` qui implique la réindexation du fichier, voir Section 3.3). Ainsi, en spécialisant légèrement la partie PerlFS présente dans le noyau (voir Figure 6.1 page 143), on pourrait *inliner* ce code directement dans le noyau et éviter ainsi le surcoût lors d'entrées-sorties intensives.

Pour conclure, même si nous avons utilisé des techniques de prototypage, nous pensons qu'une implémentation plus fine ne serait pas beaucoup plus rapide. De plus, notre expérience montre que ce «prototype» est déjà rapide et stable.

6.2 Expériences

Nous allons présenter dans cette section de nombreuses expérimentations faites avec différents types de données : des fichiers musicaux, des pages `man`, des e-mails, et des documents plus «professionnels» avec des sources de programmes, des sources LaTeX d'articles, et des référence bibliographiques avec des fichiers BibTeX. Nous pensons que l'approche LFS bénéficierait aussi aux autres types de fichiers, mais nous nous sommes restreints à certains types de documents, ceux que nous rencontrons le plus souvent. En effet, LFS est un vrai système de fichier et nous l'utilisons tous les jours. La plupart des expériences sont donc avant tout le fruit du besoin de mieux gérer nos propres documents grâce à LFS.

Certaines expériences stresseront la partie gestion de fichiers de LFS, d'autres la partie gestion de contenu de fichiers avec le fait de naviguer à l'intérieur même d'un ou plusieurs fichiers suite à la commande `cd parts` (voir Section 2.6.2 page 83).

La plate-forme utilisée pour l'ensemble des expérimentations est un PC Linux avec un noyau 2.4 muni d'un Pentium 4 à 2Ghz, de 750Mo de RAM, et d'un disque dur IDE de 40Go.

Lors de ces expériences, les fichiers posséderont tous des propriétés de sécurité (`secu:foo:rwl`, voir Chapitre 4), quelques propriétés extrinsèques, des propriétés intrinsèques systèmes comme la taille du fichier (`size:10ko`) ou sa date de modification (`date:01-Feb-2003:18h03`), et enfin des propriétés issues de transducteurs spécifiques à chaque type de fichier. Ce sont surtout ces dernières propriétés qui feront l'intérêt de LFS. Ces transducteurs sont pour la plupart de petits scripts Perl. Le code du transducteur de fichiers musicaux est présenté en Annexe C.2 page 219.

Pour ce qui concerne les parties de fichier, c'est à peu près le même schéma. Dans ces expériences, nous avons utilisé à chaque fois deux transducteurs avancés, un général et un spécifique. Le premier effectue une indexation complète du fichier. Il consiste en un script Perl de deux lignes, qui extrait une propriété par mot, par exemple la propriété `contains:foo` pour une ligne contenant le mot `foo`. Le deuxième extrait des propriétés spécifiques au type de fichier, par exemple `function:foo` pour des fichiers de type source de programme. Ce sont là encore ces dernières propriétés qui montreront l'intérêt de LFS. Ces transducteurs avancés sont de petits scripts Perl ou O'Caml. Le code du transducteur avancé de fichier BibTeX est présenté en Annexe C.3 page 220.

La plupart des propriétés sont gérées par des moteurs de déduction logique adaptés aux documents manipulés. Ainsi, l'utilisateur en plus de pouvoir formuler des requêtes de la logique des propositions (avec `/`, `&`, `|` ou `!`) peut aussi utiliser une logique d'entier (`year:>1910`), de taille `size:[3..4]Mo`, de durée `time:<7min`, de date, de chaîne (`function:.*fooa+`), de sécurité (voir Section 4.1.3 page 115), ou de type pour les fonctions (`type:?bool` pour les fonctions prenant en paramètre un booléen). Ces moteurs de déduction sont codés en O'Caml, Perl ou λProlog [MN86, HRB96]. Le code du moteur de déduction sur la logique des entiers est présenté en Annexe C.1 page 218.

Nous allons tout d'abord décrire plus précisément les données sur lesquelles seront menées les expériences et notamment décrire les propriétés et transducteurs spécifiques à ces différents types de données. Nous y verrons aussi des exemples de requêtes LFS montrant l'intérêt d'utiliser LFS pour gérer ces données. Ces exemples seront présentés encore sous la forme d'interaction avec le *shell*. Cependant, du fait que LFS est implémenté au plus bas niveau, l'utilisateur peut aussi entre autre utiliser un explorateur de fichier graphique. Nous décrirons ensuite les résultats du benchmark Andrew [KMN+88] qui est un benchmark classique utilisé pour tester des systèmes de fichier. Cependant, même si ce benchmark teste à la

fois la vitesse pour organiser, chercher, et manipuler des fichiers, il ne stressera pas vraiment ce pour quoi LFS a été conçu du fait qu'il a été inventé surtout pour tester des systèmes de fichier hiérarchiques. La composante recherche est par exemple testée en appliquant l'utilitaire `find` sur le système de fichier alors qu'avec LFS cette commande est justement devenue inutile. Nous décrirons ainsi ensuite nos propres benchmarks qui eux testeront plus précisément les capacités de LFS.

6.2.1 Fichiers musicaux

Les expériences sur les fichiers musicaux concerneront plus de 1000 fichiers de type MP3[3]. Le format MP3 prévoit le stockage de méta-données dans le fichier. Le fichier contient donc lui même des informations sur le morceau musical. Ainsi, les propriétés extraites du fichier par le transducteur MP3 sont le genre (`genre:Disco`), le nom de l'artiste (`artist:BeeGees`), l'année de production (`year:1983`) ou encore la durée de la chanson (`time:3m45s`).

Grâce au transducteur, le premier intérêt de LFS pour la gestion de fichiers musicaux est d'éviter à l'utilisateur d'avoir à classer lui même ces fichiers. Tout est automatique. De plus, si l'on compare LFS à un système de fichier classique, l'utilisateur, en plus d'être forcé de classer lui même ces fichiers, serait aussi forcé de choisir un seul type de propriété, c'est à dire de classer ces fichiers selon une seule hiérarchie, par exemple par artiste. Comme on l'a répété déjà plusieurs fois dans cette thèse, avec LFS tous les types de propriétés peuvent être pris en compte et l'utilisateur peut ensuite utiliser différents critères dans ses recherches, et peut naviguer dans différentes classifications.

La possibilité d'utiliser la négation, la disjonction ou des opérations sur les attributs valués sur des propriétés comme le genre, l'année ou le nom d'un artiste rend LFS très pratique pour sélectionner un ensemble de fichier musicaux à jouer [4] :

```
[1] % cd /lfs/music/year:[1980..1990]/
[2] % cd !genre:Comedy/
[3] % cd time:>7min
[4] % cd .ext/
[5] % playmp3 *
...
[6] % ls /lfs/music/genre:Disco/artist:/
artist:Bee Gees/ artist:DonnaSummer/
artist:Cerrone/ artist:KoolGang/
...
```

Les commandes précédentes permettent ainsi de jouer (grâce à LFS et l'utilitaire `playmp3`) uniquement l'ensemble des fichiers musicaux, non pas des extraits de sketchs comiques, produits avant les années 80, et durant plus de 7 minutes. Utilisant des propriétés extrinsèques, l'utilisateur peut même donner ensuite des notes à ces fichiers (par exemple en ajoutant avec la commande `mv` à certains fichiers la propriété `excellent`), ou encore créer des compilations virtuelles (en ajoutant par exemple la propriété `superdisco`). Toutes ces propriétés pourront d'ailleurs servir ensuite comme incréments pour permettre à l'utilisateur de retrouver plus facilement certains fichiers.

Il existe de nombreux logiciels spécialisés dans la gestion de fichier musicaux comme Winamp ou iTunes. Ces logiciels utilisent aussi les méta-données présentes dans les fichiers MP3 et évitent ainsi aussi à l'utilisateur de classer ces fichiers sous le système de fichier hiérarchique ; tous ces fichiers peuvent être mis en vrac dans le même répertoire. Le logiciel iTunes permet à l'utilisateur de naviguer selon différentes hiérarchies, par genre, par artiste, et propose même une certaine forme d'incrément : après la sélection du genre, le logiciel propose automatiquement comme complément de requête uniquement des noms d'artistes ayant fait des musiques appartenant à ce genre (comme la commande `6`).

Cependant, ces logiciels imposent de nombreuses restrictions et n'offrent pas autant de possibilités que LFS. Par exemple, même si il est permis de naviguer par genre puis par artiste, l'inverse n'est pas possible. Il n'est pas non plus permis de naviguer par année, ou encore de formuler des requêtes évoluées. Ainsi, même en étant spécialisés, ils n'arrivent pas à offrir l'ensemble des services proposés par LFS (qui est un système général). Nous n'avons rien eu à coder pour bénéficier de tous ces services, si ce n'est le transducteur MP3 qui ne contient que 30 lignes de code Perl (qui font essentiellement appel

[3] Fichiers que possède l'auteur, tous légaux bien sûr (ou presque).

[4] La commande `cd .ext` permet de récupérer l'extension d'une requête et a été décrite Section 5.2.2 page 134.

à une librairie sachant manipuler le format MP3). Au contraire, les logiciels iTunes et Winamp, bien que moins puissants ont nécessité beaucoup plus d'investissements et contiennent beaucoup de lignes de code.

Ainsi, un peu comme dans la tradition Unix, il est préférable d'avoir un petit système bénéficiant d'un petit nombre de concepts *orthogonaux* pouvant se combiner, que d'avoir un logiciel ad-hoc, avec certes des éléments graphiques pouvant paraître alléchants mais qui au fond limitent l'utilisateur.

Cette approche peut se généraliser aux autres types de données multimédia comme les images ou les vidéos. Si le format du fichier ne prévoit pas le stockage de méta-données comme c'était le cas avec les MP3, l'utilisateur pourra quand même avec des propriétés extrinsèques associer de multiples propriétés à ces fichiers. Par exemple, pour le rangement de photographies de vacances on pourrait associer aux photos le nom du pays, les noms des personnes présentes sur la photo, etc.

En fait, on pourrait même aller plus loin en naviguant à l'intérieur de ces données multimédia avec la commande `cd parts`, tout comme on peut déjà naviguer par exemple à l'intérieur d'un source de programme. En effet, même si actuellement la notion de *partie de fichier* est fortement associée à la notion de ligne, les principes exposés Section 2.6.2 page 83 peuvent s'appliquer aussi à d'autres notions de parties, comme par exemple des morceaux de vidéos. Le standard MPEG7 définit comment des méta-données peuvent être attachées à des *segments* d'un flux vidéo. En supposant l'existence d'un outil de segmentation que l'on pourrait utiliser comme un transducteur avancé, l'utilisateur pourrait naviguer dans un film en exprimant de simples requêtes comme `cd scene:kung-fu|amour/; ls character:/`. Cette requête permettrait d'afficher seulement les noms de personnages apparaissant dans des scènes de Kung-Fu ou d'amour, et la commande `playvideo character:neo/matrix.mpg7` permettrait ensuite de jouer certaines de ces scènes, celles où le personnage `neo` apparaît. On pourrait aussi naviguer à l'intérieur d'un fichier musical. La requête `playmp3 !instrument:guitar/yesterday.mp3` permettrait par exemple de jouer le morceau sans la partie guitare.

6.2.2 Pages `man`

Les expériences sur les pages `man` concerneront plus de 10000 fichiers. Le but est de fournir grâce à LFS un meilleur système de documentation avec un meilleur outil de recherche que la commande Unix `apropos`. Les propriétés extraites du fichier par le transducteur de page `man` sont sa section (les pages `man` sont divisées en différentes sections, par exemple la première section correspond aux commandes standards, la troisième aux API), les mots-clefs composant sa description (`keyword:change`), et enfin les options acceptées par cette commande si la page `man` correspond effectivement à une commande (`option:-R`).

Trouver de l'aide avec l'aide de LFS semble plus rapide qu'avec `apropos`. En effet, classiquement un utilisateur doit d'abord lancer la commande `apropos`, puis localiser l'entrée correspondante, ce qui peut prendre du temps puisque le résultat d'une telle requête peut être long. On retrouve l'inconvénient classique des systèmes basés sur l'interrogation. Avec LFS, le système propose cette fois automatiquement des compléments de requête (les incréments) et le processus de recherche converge rapidement vers la page `man` recherchée. La suite de commandes *shell* suivante montre un exemple de session[5] :

```
[1] % cd lfs/man^/
[2] % ls
keyword:/ section:/ option:/
[3] % cd section:/; ls
section:Command/
section:API/
...
keyword:/ option:/
[4] % cd section:Command
[5] % cd keyword:^/
[6] % cd keyword:change; ls
keyword:directory/
keyword:rights/
keyword:owner/
...
[7] % cd keyword:owner/; ls
```

[5] Les commandes `cd man^/` et `cd keyword:^/` permettent de restreindre les incréments respectivement à des sous-concepts de la propriété `man` et `keyword:`. Ce type de commande a été décrit Section 5.2.2 page 134.

```
chown.man
[8] % cat chown.man
...
[9] % cd /lfs/man/keyword:flush; ls
fflush.man
```

En plus du manque de navigation, la commande `apropos` ne propose qu'un seul critère de recherche : la recherche par mot-clef. LFS en propose trois, et ces trois critères peuvent se combiner les uns avec les autres afin d'accélérer la recherche. Nous aurions pu aussi extraire par exemple le nom de l'auteur de cette commande, ou encore le type des paramètres attendus (un fichier, un répertoire, un fichier musical) afin de bénéficier d'un nombre de critères encore plus grand. On pourrait aussi mieux classer les commandes par exemple en les rangeant dans différentes catégories, les programmes multimédia, les programmes bureautiques, ce qui permettrait avec la navigation de trouver plus facilement le nom d'une commande.

Nous pensons que cette approche peut se généraliser aux autres types de systèmes de documentation, par exemple la documentation d'un langage de programmation, de son API, de ses modules, ou encore la documentation d'un outil comme Emacs. Ce dernier possède de nombreuses commandes pas toujours évidentes à trouver ou retrouver. On pourrait extraire de nombreuses propriétés de ces commandes pour faciliter ces recherches. Emacs propose d'ailleurs lui aussi un système d'aide, avec des recherches par mot clef, une navigation hypertexte thématique, des recherches par touche. Cependant, là encore toutes ces méthodes de recherche ne peuvent pas se combiner.

6.2.3 E-mails

> *Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.*
>
> *Jamie Zawinski*

Nous mènerons des expériences sur une collection de plus de 40000 e-mails où LFS sera utilisé pour concurrencer les fonctionnalités d'organisation et de recherche des logiciels de messagerie. Dans ces expériences, chaque e-mail correspondra à un fichier, avec comme contenu de fichier le contenu de l'e-mail et comme nom de fichier le sujet de cet e-mail [6] suivi du suffixe `.mail`.

Les propriétés extraites de chaque e-mail par le transducteur d'e-mail sont son sujet (`subject:Re:foo killed bar`), son *thread* (`thread:foo killed bar`) afin que l'e-mail original et ses réponses partagent une propriété pour pouvoir ensuite retrouver l'ensemble de la discussion, ses thèmes (`thema:foo`) qui correspondent aux différents mots présents dans le sujet, l'expéditeur (`from:foo@irisa.fr`) ainsi que lorsque l'adresse e-mail de l'expéditeur suit une norme particulière les nom et prénom de cet émissaire (`from_name:Yoann Padioleau`), et enfin la date (`date_send:01-Jun-2003:18h01`). Comme dans les autres expériences, l'utilisateur a aussi la possibilité d'utiliser les propriétés `contains:` gérées en interne par le logiciel Glimpse [MW94] (voir Section 3.1.2 page 99) qui permet une indexation complète du contenu de ces e-mails. De même, chaque e-mail possède des propriétés systèmes intrinsèques comme sa taille ce qui permet par exemple à l'utilisateur ensuite de sélectionner plus facilement les e-mails prenant le plus de place sur le disque pour les effacer.

Nous avons codé un petit script qui est chargé de récupérer la boîte aux lettres de l'utilisateur dans `/var/spool/mail`, d'analyser ce fichier pour en extraire les différents e-mails, pour enfin créer les fichiers correspondant sous LFS. Ces fichiers n'ont au début pas la propriété `read`, qui représente le statut de lecture d'un e-mail. Ainsi, à chaque fois qu'un utilisateur souhaite consulter ses nouveaux e-mails, il lui suffit d'appeler ce script et de se placer ensuite dans le répertoire `!read`. Lorsqu'il aura fini de lire ces nouveaux e-mails, il lui suffira avec la commande `mv` de leur ajouter la propriété `read`. L'utilisateur organise et recherche ses e-mails avec un *shell* ou un *browser*, utilise `cat` ou Emacs pour voir le contenu de ces e-mails, et peut ensuite utiliser la commande `mail` pour répondre à ces messages.

Cela peut paraître un peu rudimentaire mais l'on pour-

[6] Sous LFS deux fichiers peuvent partager le même nom du moment qu'ils n'ont pas exactement la même description, c'est à dire les mêmes propriétés. Ainsi, deux e-mails peuvent avoir le même sujet sans que cela pose probème

rait facilement imaginer une interface graphique plus conviviale qui par exemple lorsque l'on clique sur un e-mail lui associe ensuite automatiquement la propriété `read`, ou qui présente les fonctionnalités de recherche et d'organisation de LFS de manière plus graphique. Il faut noter que cette application serait beaucoup plus facile à construire qu'un logiciel de messagerie adhoc puisque elle pourra se reposer sur les services offerts par LFS et n'aura donc à s'occuper que de l'aspect graphique. Même si LFS ne concurrence pas complètement un logiciel de messagerie, et requiert certains logiciels comme `mail`, il concurrence tout de même une grande partie de leurs fonctionnalités avec leurs capacités d'organisation et de recherche d'e-mails.

L'exemple suivant présente des commandes possibles avec LFS concernant la gestion d'e-mails. Le sens de certaines des commandes est expliqué dans les paragraphes suivants.

```
[1] % cd /lfs/e-mail/
[2] % get_newmail
[3] % cd !read/.ext; ls
Vacation.mail  Re:buy books.mail
Enlarge Your Penis.mail
...
[4] % cat Vacation.mail
From:maman@maison.fr
Subject: Vacation
----------------------
ready to go ?
[5] % mail maman@maison.fr
       "i am ready to go"
[6] % mv * ../read
[7] % cd /lfs/e-mail^/; ls
subject:/ thema:/ thread:/
from:/ from_name:/ mailing_list:/
SPAM/ boite:/
date_send:/ year:/ month:/
[8] % cd !SPAM/thema:LIS|thema:LFS/; ls
from_name:Sebastien Ferre/
from_name:Olivier Ridoux/
...
thema:Erreur/ thema:mkfs/ thema:release/
...
year:2003/ year:2004/
month:Jul/ month:Mar/
...
[9] % ls from:Olivier Ridoux/.ext
Erreur dans mkfs.mail
release 2.mail
...
[10] % cd /lfs/e-mail/
[11] % mv from:ridoux@irisa.fr/
    boite:encadreur/
[12] % mv from:.*@irisa.fr/ boite:irisa/
[13] % mv thema:sex/ SPAM/
[15] % ls boite:/
boite:encadreur/ boite:irisa/
boite:friends/
boite:badminton/ boite:boulot/
boite:girls/
```

Un problème récurrent auquel est confronté un utilisateur est de retrouver un ancien e-mail. C'est pour cette raison que l'utilisateur les garde en archive et les classe. Le classement en différents *thread* où l'on regroupe les e-mails appartenant à la même discussion n'est cependant pas la seule façon de classer ses e-mails. La plupart des logiciels de messagerie, par exemple Netscape, permettent ainsi à l'utilisateur de classer manuellement ses e-mails dans différents *boîtes* (*folder*), en faite des sortes de répertoires. On peut ainsi les ranger par thème, par auteur. Cependant, on retombe sur les problèmes des systèmes de fichier hiérarchiques puisqu'il est impossible d'associer plusieurs propriétés à un e-mail, c'est à dire de le placer dans différentes boîtes en même temps. On voit bien que les problèmes des structures hiérarchiques sont omni-présents. L'idée de répertoire (ici appelé boîte) et le besoin de classer et d'organiser ne se retrouvent pas uniquement au niveau d'un système de fichier. Avec LFS il suffit à l'utilisateur d'utiliser la commande `mv` pour associer différentes propriétés à ces fichiers pour mieux les retrouver ensuite.

Une autre fonctionnalité importante de LFS dans le cadre de la recherche d'e-mails est l'utilisation des moteurs logiques. En effet, l'utilisateur n'a très souvent qu'une idée très partielle de ce qu'il recherche, par exemple que l'e-mail qu'il cherche fut posté pendant l'été. L'utilisation d'un moteur logique sur les dates permet tout de même d'exploiter ces informations. De plus, les incréments peuvent ensuite aider l'utilisateur pour compléter sa recherche. En effet, même si il ne se

rappelle plus précisément le nom de son expéditeur, il pourra peut-être le reconnaître lorsque il lui sera présenté sous la forme d'un répertoire. Reconnaître ce que l'on cherche est souvent plus facile que d'exprimer ce que l'on cherche.

Une fonctionnalité apparue récemment dans certains logiciels de messagerie est la possibilité de classer automatiquement les e-mails arrivant. Ainsi, il est possible de créer des sortes de *boîtes virtuelles* à qui l'on associe une requête qui spécifie quel genre d'e-mail peut appartenir à cette boîte. Par exemple, la boîte `friends` pour tous les e-mails dont l'expéditeur appartient à une liste spécifiée par l'utilisateur. On retrouve sous une autre forme la notion de répertoire virtuel, en fait dans ce cas la notion de répertoire-vue[7] fournie par le système de fichier Nebula [BDB+94] décrit Section 1.4.3 page 27. Cependant LFS fait mieux. Contrairement à ces systèmes, sous LFS les propriétés comme `from:fubar@irisa.fr` peuvent être utilisées pour interroger mais aussi pour naviguer. Les propriétés extraites par les transducteurs jouent ainsi déjà le rôle de boîtes puisque le système les propose. On peut même classer ces propriétés en utilisant les axiomes, par exemple avec la commande `mv from:ridoux@irisa.fr/ boite:friends/` en faisant de la propriété `from:ridoux@irisa.fr` désormais un sous-concept de `boite:friends` (en plus bien sûr d'être déjà un sous concept de `from:`). Les e-mails (ainsi que les futurs e-mails) ayant la propriété `from:ridoux@irisa.fr` feront désormais aussi parties de la «boîte virtuelle» `boite:friends`. On peut même aller plus loin puisque comme les propriétés-formules sont des propriétés comme les autres sous LFS, grâce aux moteurs logique, on peut généraliser la notion de boîte. Par exemple, la commande `mv from:.*@irisa.fr/ boite:irisa/` permet grâce à un moteur logique sur les chaînes de caractères de classer automatiquement les e-mails existants et à venir venant du domaine `irisa.fr` dans la boîte adaptée. De plus, sous LFS du fait des incréments toutes ces boîtes peuvent coopérer ensemble pour aider l'utilisateur à trouver ce qu'il cherche, ce qui n'est pas le cas avec les boîtes-vues de ces logiciels de messagerie (ou les répertoires-vues de Nebula).

[7] Le mot vue n'a ici rien à voir avec la notion de vue sous LFS.

Cette approche peut se généraliser aux autres formes de communication présentes sous Internet comme les forums de discussions, les *weblogs*, les *news*. De nombreux sites fonctionnent sous la forme de *dépêches*. Il est par exemple possible de consulter un ensemble d'articles du New York Times, ou encore de regarder les nouvelles concernant l'actualité financière, informatique, etc. Un utilisateur peut s'enregistrer sur ces sites et se voir présenter à chaque visite les nouveaux articles, un peu comme des nouveaux e-mails. On pourrait donc là encore appliquer LFS pour gérer ces nouveaux types de données. L'Internet de manière plus générale et ses moteurs de recherche pourraient aussi bénéficier de LFS. Les sites seraient indexés par de nombreuses propriétés et il serait ensuite possible pour l'internaute de retrouver un site en utilisant de nombreux critères. En attendant ces indexations, un utilisateur pourrait déjà utiliser LFS pour gérer ses bookmarks et classer ainsi les sites qu'il fréquente le plus. En effet, un bookmark est actuellement une structure hiérarchique qu'il faudrait mieux remplacer par une structure LFS.

6.2.4 Sources de programmes

Nous utiliserons LFS pour gérer des sources de programmes, des projets de logiciels. Ces sources seront tout d'abord les données du benchmark d'Andrew [KMN+88] qui sont censées représenter un projet typique C de taille moyenne, et enfin les sources du noyau Linux. Les propriétés extraites du fichier par le transducteur C sont pour l'instant uniquement les noms de fonctions composant ce module (`function:fork`), comme produit par la commande `ctags`. Les fichiers possèdent aussi des propriétés extrinsèques correspondant au répertoire dans lequel ils étaient situés dans le système de fichier hiérarchique (pour Linux, les propriétés `driver`, `fs`, `include`, etc).

On aimerait classer ces sources selon différentes classifications, selon le type du fichier (un fichier d'interface `.h`, un fichier d'implémentation `.c`, de la documentation `.txt`, etc), pour le noyau Linux selon l'architecture (`x86`, `ppc`, etc) ou selon le module (les sources liés aux périphériques `driver`, ceux liés à la gestion de la mémoire virtuelle `mm`). L'organisation actuelle des sources du noyau souffre des limitations des systèmes de fichier hiérarchiques. Ainsi, sous ces systèmes les program-

meurs sont forcés de favoriser une de ces classifications, et donc de ranger par exemple d'abord selon le type, puis selon le module, puis selon l'architecture. On arrive ainsi à une profusion de répertoires portant le même nom mais éparpillés dans la hiérarchie, tout comme il existe sous Unix une profusion de répertoires `bin`. Le répertoire `sound` par exemple apparaît 4 fois. Il n'est ainsi pas possible de récupérer facilement tous les fichiers liés au son. L'organisation n'est d'ailleurs pas toujours cohérente, et est finalement difficile à comprendre ce qui rend la manipulation de l'ensemble de ces fichiers difficile. Avec LFS tous ces problèmes disparaissent. Ainsi, il est possible d'avoir en même temps toutes ces classifications, de récupérer par exemple l'ensemble des fichiers liés à la gestion de la mémoire, que ce soit de la documentation, des sources ou des fichiers d'interface, ou inversement de récupérer tous les fichiers d'interfaces. Les exemples suivants montrent des requêtes possibles ainsi que l'organisation sous LFS des sources du noyau.

```
[1] % cd /lfs/kernel; ls
architecture:/ module/ kind:/
ext:/ name:/ size:/
[2] % ls architecture:^/
architecture:x86/ architecture:alpha/
architecture:ppc/ architecture:ia64/
[3] % ls kind:^/
kind:source/ kind:header/
kind:documentation/
kind:objects/
kind:script/ kind:make-related/
[4] % ls module^/
boot/ init/
drivers/ mm/ fs/
ipc/ net/ sound/
lib/
[5] % ls function:fork/.ext/
fork.c syscall.h
[6] % ls sound/kind:documentation/
ultrasound.txt
soundblaster.txt
...
[7] % ls ext:asm/module^/
boot/ mm/
drivers/
```

Nous mènerons des expériences où l'on naviguera parfois parmi ces fichiers, parfois à l'intérieur des fichiers. En effet, même si la requête `cd function:fork/` retourne uniquement l'ensemble des fichiers possédant une fonction `fork`, ces fichiers peuvent eux mêmes être longs et la commande `cd parts/function:fork/` permet alors de naviguer à l'intérieur de ces fichiers et de sélectionner uniquement la partie du code implémentant vraiment cette fonction.

Le nombre des propriétés extraites par fichier C étant petit, ces expériences ne montrent pas toutes les possibilités que pourraient offrir LFS. Le transducteur C n'est pour l'instant pas très avancé. Nous parlerons plus loin d'expériences faites sur les sources de LFS avec des sources O'Caml et Perl bénéficiant de transducteurs beaucoup plus avancés et donc présentant des requêtes plus intéressantes. Nous présentons tout de même des expériences sur des projets C car tout d'abord le benchmark Andrew est un benchmark classique pour les systèmes de fichiers, et car les sources du noyau offrent la possibilité d'expérimenter LFS sur un grand projet, ce que n'est pas LFS du point de vue taille de code. Cela permettra ainsi de tester les capacités algorithmiques de LFS.

Pour le noyau on pourrait imaginer isoler et assigner des propriétés aux aspects de synchronisation (avec les `lock`/`unlock`), aux aspects de gestion d'erreur, de mémoires, etc. Cela permettrait ensuite de mieux manipuler et gérer ces aspects et la complexité du noyau.

6.2.5 Un *homedir*

Nous venons de voir différents types de données, des fichiers musicaux, des e-mails, de la documentation et des sources de programmes, et nous avons vu comment LFS concurrence et même rend plus de services que des logiciels spécialisés comme Winamp, `apropos` ou Netscape. Dans ces expériences, on utilise les mêmes principes d'organisation et de recherche : les principes de LFS. Cela évite ainsi à l'utilisateur d'avoir à réapprendre à chaque fois à utiliser un nouveau logiciel, de nouveaux principes, et de nouvelles interfaces (et de nouvelles limitations). Un autre intérêt de LFS par rapport à ces logiciels spécialisés est qu'il est possible avec LFS de lancer des requêtes *transversales* sur ces différents types

de données du fait que les services de LFS sont implémentés au plus bas niveau et du fait que toutes ces données peuvent être placées sous la responsabilité de LFS. Ainsi, nous mènerons des expériences où l'on aura stocké toutes ces données (e-mails, fichiers musicaux, etc) ainsi que toutes les autres données d'un utilisateur, c'est à dire son *homedir*, sous LFS.

Un utilisateur cherchant un fichier ou tous les fichiers contenant le mot `synchronisation` mais ne se rappelant plus exactement leurs types, leurs emplacements, pourra de son *homedir* ainsi lancer la requête `cd contains:synchronisation/`. Il pourra récupérer ainsi l'ensemble des données quel que soit son type, des e-mails, de la documentation (quel que soit son format), des sources de programme (quel que soit le langage), mentionnant le mot `synchronisation`. Le système proposera même à l'utilisateur comme complément de requête le type des fichiers contenant ce mot avec par exemple comme incrément `e-mail` ou `documentation`. Une fois choisi le type, l'utilisateur pourra continuer sa requête en utilisant les propriétés propres à ce type de données.

En fait, de nombreuses propriétés peuvent être partagées par ces données, en plus des propriétés systèmes. Dans le cadre des données multimédia, l'utilisateur peut posséder des extraits de films sous forme sonore. Ces fichiers, bien que contenant des données sonores, pourront posséder des propriétés propres aux vidéos, et être classés manuellement selon des propriétés cinématographiques. De même, la notion de genre se retrouve aussi bien dans la musique que dans le cinéma. Ainsi, l'utilisateur pourra par exemple récupérer l'ensemble des données propres à la comédie avec la requête `cd genre:Comedy/`. Le système pourra proposer comme incréments `movie` et `music`, voir même `e-mail` car certains e-mails contenant une blague auront reçus la propriété `genre:Comedy`. On pourra de même faire partager à toutes les données d'un projet, les e-mails, les sources, la documentation, une même propriété comme par exemple le nom du projet. Actuellement ces données sont souvent éparpillées dans le `homedir` de l'utilisateur car on a préféré favoriser la séparation en type de fichier que la séparation en projet. On peut espérer qu'avec LFS les données d'un utilisateur seront plus faciles à organiser. De même, dans le cadre de la programmation, des sources codés dans différents langages pourront posséder des propriétés communes. Ainsi, même si une fonction `foo` n'est pas actuellement disponible dans un langage x, la requête `cd function:foo/` permettra au programmeur de rapidement trouver une source d'inspiration en s'appuyant sur des sources écrits dans des langages différents mais implémentant cette même fonction.

Les exemples suivants montrent des requêtes possibles[8] :

```
[1] % cd /lfs/; ls
music/ e-mails/ publi/ code/ project/
todo/
system/ contains:/
[2] % ls -l
1250  music/
11560 e-mails/
10    publi/
100   code/
500   project/
40    todo/
13290 system/
13290 contains:/
[3] % ls system^/
ext:/ name:/ size:/ date:/ secu:/
[4] % ls music^/
genre:/ year:/ time:/ artist:/
[5] % ls e-mails^/
subject:/ thema:/ thread:/
from:/ from_name:/
SPAM/ boite:/
[6] % ls project^/
LFS/ CHR/ ICFP-contest/
[7] % cd LFS; ls -l
20  e-mails/
3 publi/
20 code/
4 todo/
43 system/
43 contains:/
[8] % cd code/ext:^/; ls -l
15 ext:ml/
5  ext:pl/
```

[8] La taille d'un répertoire sous LFS (visible suite à une commande comme `ls -l`) correspond au nombre de fichiers satisfaisant ce répertoire, c'est à dire la taille de l'extension de ce répertoire.

```
[9] % cd /lfs/contains:synchronisation
[10] % ls
e-mails/ publi/ code/
system/ contains:/
```

Cette approche peut se généraliser au système entier en faisant de LFS le système de fichier par défaut gérant tout depuis la racine `/` du système. Nous avons vu précédemment comment LFS offrait une meilleure organisation pour les sources du noyau Linux en passant d'une structure hiérarchique à une structure LFS. Il pourrait en être de même pour toutes les données d'une distribution Linux. Le répertoire `bin` ne serait ainsi plus éparpillé partout dans la hiérarchie. On pourrait ainsi classer un fichier à la fois selon son type (`kind:bin`, `kind:lib`, `kind:doc`, etc), selon le logiciel auquel il se rattache (`soft:emacs`, `soft:kde`, `soft:X11`, etc), selon la cible à laquelle s'adresse ce fichier (`target:root`, `target:usr`, `target:net-admin`, etc), ou encore selon le type d'installation (`install:local`, `install:opt`). Le répertoire traditionnel `sbin` deviendrait par exemple sous LFS la conjonction `kind:bin/target:root`.

Actuellement, de nombreuses choses sont proches dans l'esprit mais très distantes dans la hiérarchie. Par exemple, la documentation d'Emacs se retrouve à la fois sous forme de pages `man`, sous forme de fichiers `info`, sous forme de fichiers `.txt`, sous la forme d'une FAQ, etc. Tous ces fichiers sont éparpillés dans le système et bénéficient de systèmes d'indexation différents, ce qui fait qu'il est très difficile de récupérer l'ensemble de la documentation ayant trait à Emacs. Du fait de cette variété de documentations, de programmes, de langages de programmation, de systèmes de fenêtrages, le système Linux ne paraît pas uniforme et donc compliqué.

Parfois l'inverse peut se produire et des données peuvent se retrouver trop proches dans la hiérarchie. Ainsi, les exécutables des programmes sont rangés en vrac dans le même répertoire `/usr/bin`. Cette profusion de programmes fait qu'il est parfois difficile pour un utilisateur de trouver le bon programme. On aimerait ainsi les classer par catégories, avec la catégorie logiciel de messagerie, compilateur, etc. On retrouve d'ailleurs ces catégories dans le format de description des *packages* du système, mais pas dans le système de fichier.

Il est bien sûr difficile d'organiser un grand ensemble de fichiers comme une distribution Linux. Les initiatives comme la FHS [RQY94] (*Filesystem Hierarchy Standard*) ont justement pour but de standardiser et mieux organiser le système. Cependant, comme son nom l'indique, la FHS vise à trouver une *hiérarchie*, et non une *organisation*. Les problèmes arrivent lorsque l'on veut hiérarchiser des concepts orthogonaux qui ne peuvent donc pas être hiérarchisés. La FHS est donc le fruit d'un compromis. Nous pensons ainsi que l'organisation complète du système d'exploitation Linux serait non seulement meilleure sous LFS, mais aussi plus facile à concevoir. Paradoxalement même si les fichiers auront plus de propriétés, le concepteur d'un *package* aura moins de question à se poser du fait qu'il pourra assigner librement des propriétés à ces fichiers, sans se poser la question de l'ordonnancement entre ces propriétés. Nous pensons que le système sera plus cohérent et uniforme à tous les niveaux, pour la documentation, les librairies, les données multimédia. L'administration système et les fichiers de configurations et de préférences pourraient eux aussi bénéficier de l'approche LFS.

6.2.6 Fichiers BibTeX

Nous allons maintenant décrire dans cette section et les deux suivantes des expériences où l'on naviguera à l'intérieur des fichiers et non plus parmi les fichiers.

Nous utiliserons LFS pour naviguer à l'intérieur de fichiers BibTeX de différentes tailles (10 000 et 100 000 lignes). Le transducteur BibTeX extrait des propriétés qui correspondent aux différents champs d'une entrée BibTeX : le titre (`title:Logic File System`), les auteurs (`author:Y.padioleau`), l'année, le type (`type:article`), etc.

Ces propriétés sont dupliquées sur toute les lignes d'une entrée BibTeX pour gérer le fait que la vraie notion de partie est l'entrée et non la ligne. Par exemple, la propriété `title:LFS` sera aussi assignée à la ligne mentionnant l'auteur de cet article. En effet, suite à la requête `cd title:LFS/` l'utilisateur aimerait éditer l'entrée BibTeX complète, pas juste la ligne du titre. Ainsi, même avec une gestion orientée ligne, le schéma d'assignation des propriétés et la notion d'*état* dans un transducteur permettent d'avoir en fait une gestion plus fine. On fait de plus correspondre à chaque nouvelle entrée un nouveau *point de synchronisation*, ce qui permet de rendre

la modification d'un fichier BibTeX rapide (voir Section 3.1.3 page 102).

Naviguer dans un fichier BibTeX donne une vision du fichier qui ressemble à ce que l'on peut obtenir avec du *data-mining* [HMS01, AIS93]. En effet, l'utilisateur par le biais des incréments peut savoir quels sont par exemple les co-auteurs les plus fréquents d'un auteur, les dates les plus importantes pour un sujet.

Les exemples suivants montrent des requêtes possibles[9] :

```
[1] % cd /lfs/ext:bib/; ls
ref.bib
[2] % cd parts; ls
title:/ type:/ ref:/
author:/ year:/ title:/
institution:/ contains:/
domain:/
ref.bib
[3] % cd .relaxed
[4] % cd author:/; ls
author:A.Aho/ author:Wadler/
author:O.Ridoux/ author:S.Ferre/
...
year:/ title:/ type:/ ref:/ ...
ref.bib
[5] % cd author:O.Ridoux/year:>1990/; ls
author:S.Ferre/ author:Y.Padioleau/
author:P.Brisset/ author:Y.Bekkers/
...
year:2003/ year:2002/ year:2000/
year:1998/ year:1996/
...
title:/ type:/ ref:/ ...
ref.bib
[6] % ls domain:^/
domain:logic/ domain:compilation/
domain:langage/ domain:formal-methods/
domain:filesystem/
[7] % cat author:Y.Padioleau/ref.bib
.....................:1
@inproceedings{lfs-article,
author = "Y. Padioleau and O. Ridoux",
title = "A Logic File System",
year = 2003,
booktitle = "USENIX",
keywords = "filesystem, logic",
}
.....................:2
```

Il existe de nombreux outils pour manipuler des fichiers BibTeX, mais aucun n'offre autant de possibilités que LFS. L'outil Bib2html [Mar], qui génère automatiquement un ensemble de fichiers HTML à partir d'un fichier BibTeX, permet ensuite via un navigateur Web de naviguer dans ces données selon différentes classifications : par année, par sujet, par auteur comme sous LFS. Cependant, la modification de ces fichiers n'aura aucun effet sur le fichier d'origine et l'utilisateur ne peut ainsi pas par exemple corriger certaines données suite à une de ses navigations. On ne peut pas non plus combiner une navigation par année et une navigation par auteur, ou encore formuler des requêtes avancées comme la recherche de documents faits avant les années 1960. Enfin, aucun des outils BibTeX ne propose une forme d'incrément qui permettrait à l'utilisateur de retrouver plus facilement ce qu'il cherche.

Cette approche se généralise à toutes les formes de bases de données. De nombreux sites Web regroupent par exemple un ensemble d'informations sur des films, des livres, des musiques. Chaque entité possède de nombreuses propriétés comme l'année de production, le titre, ses acteurs pour un film, sa note. Ces catalogues, soit commerciaux soit issus de la collaboration d'internautes, fournissent aussi des outils de recherche mais qui sont basés uniquement sur l'interrogation, qui de plus est souvent très pauvre. Les moteurs logiques de LFS et les incréments aideraient grandement l'utilisateur à trouver par exemple le film ou le livre qu'il cherche.

6.2.7 Publications LFS

Nous utiliserons LFS pour naviguer à l'intérieur de fichiers LaTeX. En fait, ces fichiers seront les sources des articles de recherche exposant LFS ainsi que la thèse LFS, c'est à dire ce document. En effet, comme il a déjà

[9] La commande `cd .relaxed` permet de limiter le nombre d'incréments. Elle a été décrite Section 5.2.2 page 132. Dans cet exemple, l'effet est de limiter les incréments aux propriétés descendant de `author:` et `year:`. Elle permet de «développer une branche», tout en gardant la possibilité d'ouvrir les autres plus tard.

été dit nous utilisons LFS pour gérer nos propres documents. Cette section présente donc la gestion de la thèse de LFS par LFS, tandis que la section suivante présentera la gestion du code de LFS par LFS.

Le transducteur LaTeX extrait comme propriétés les noms et types de section (une partie, un chapitre, une section, un paragraphe), des propriétés correspondant à des commandes LaTeX comme l'indexation (`index`:`SFS`) ou les environnements (`env`:`verbatim`), et enfin des propriétés pour les commentaires. Certaines propriétés, par exemple `chapter`:`Principes` sont assignées à un ensemble de lignes consécutives, ici toutes les lignes du chapitre, d'autres juste à une ligne. Le transducteur LaTeX propose aussi une indexation plus fine que le transducteur générique qui permettait d'indexer complètement le fichier (avec les propriétés de la forme `contains`:x). Par exemple, la nouvelle propriété `paracontains`:`foo` est assignée à l'ensemble des lignes d'un paragraphe contenant le mot `foo`, et non pas seulement à la ligne mentionnant le mot `foo`. En effet, souvent l'utilisateur veut aussi voir le contexte de cette ligne.

Avec toutes ces propriétés, l'utilisateur peut naviguer dans son document selon la structure du document, avec un grain pouvant aller jusqu'au simple paragraphe. Il peut utiliser différentes vues sur ce document, par exemple une vue regroupant toutes les introductions, toutes les conclusions. L'exemple présenté plus bas illustre un ensemble de requêtes possibles. On peut par exemple avoir une vue ne présentant pas toutes lignes ayant des commandes LaTeX liées à l'indexation (commande `12`), ou une vue où tous les exemples de session *shell* utilisant LFS sont réunis (commande `11`). On peut aussi avoir une vue où seulement les entêtes des titres sont présentés (commande `7`), pour des titres ayant une profondeur inférieure à 3 (les parties, chapitres et sections). Cela permet ainsi d'avoir une vue sur une sorte de table des matières. La navigation hypertexte présentée Section 5.2.3 page 135 permet même ensuite de rentrer dans certaines sections. Les répertoires présentent aussi une forme de table des matières puisque des propriétés comme `chapter`:`Principes` sont proposées comme incréments. On peut encore avoir une vue où tous les paragraphes mentionnant le mot `SFS` sont réunis (commande `13`). Ainsi, de nombreuses notions transversales au document où des thèmes pouvant revenir peuvent être manipulées plus facilement. Il est même possible avec les incréments de mieux voir globalement les endroits où l'on parle de telle ou telle notion puisque l'on pourra voir entre autre l'ensemble des chapitres où l'on parle du mot `SFS` (commande `10`) sous la forme de répertoires. On peut ainsi mieux se rendre compte si l'on a oublié de parler d'une certaine notion dans un chapitre.

```
[1] % cd /lfs/ext:tex/; ls
thesis.tex
[2] % cd parts; ls
part:/ section:/ subsection:/
note:/ comment:/ aspect:/
environment/ latex:/
contains:/ titledepth:/
[3] % ls part:^/
part:Contexte/ part:Contributions/
part:Evaluation
[4] % ls part:Contributions/chapter:^/
chapter:Principes/ chapter:Algorithmes/
chapter:Sécurité/ chapter:Extensions/
[5] % cat chapter:Résumé/thesis.tex
 ......................:1
 \chapter*{Résumé}
 %overview: définition domaine %dup: intro
 Les systèmes d'information offrent ...
 %overview: définition problème
 Pour rechercher ces documents, ...
 %overview: définition solution
 Nous proposons un nouveau ...
 %plan:
 Nous présentons les principes ...
 ......................:2
[6] > cat chapter:Résumé/
    (!comment)|aspect:structure/thesis.tex
 ...................:1
 \chapter*{Résumé}
 ...................:2
 Les systèmes d'information offrent ...
 ...................:3
 Pour rechercher ces documents, ...
 ...................:4
 Nous proposons un nouveau ...
 %plan:
 Nous présentons les principes ...
 ......................:5
```

```
[7] > cd titledepth:<3/; cat thesis.tex
 ...................:1
 \chapter*{Résumé}
 ...................:2
 \part{Contexte}
 ...................:3
 \chapter{Les systèmes de fichier}
 ...................:4
 \part{Contributions}
 ...................:5
 ....
[8] > sed -e s/Résumé/Abstract/
          thesis.tex
[9] > ls chapter^:/
chapter:Abstract/
chapter:Les systèmes de fichier/
...
[10] > ls contains:SFS/chapter:^/
chapter:Les systèmes de fichier/
chapter:Principes/
chapter:Expérimentations/
[11] > emacs env:shellsession/
        thesis.tex
[12] > emacs !index:/!comment:/
        thesis.tex
[13] > emacs paracontains:SFS/
        thesis.tex
```

Les commentaires LaTeX pour la thèse bénéficie d'un traitement spécial. La rédaction de ces commentaires suit une certaine discipline et il existe ainsi différentes sortes de commentaires. Certains commentaires sont préfixés par un mot-clef suivi du symbole `':'`, par exemple `%overview:`. Chacun de ces mots-clefs permet de structurer ces commentaires, de documenter un aspect du document. En effet, une thèse ou un livre est un objet digital très complexe et il est important pour l'auteur, afin de mieux s'y retrouver, de documenter aussi la structure du document, de documenter toutes les choses implicites qu'il a dans sa tête et qui lui permettent de produire un meilleur document, plus cohérent, mieux structuré. Par exemple, le *tag* `overview` mis au début de certains paragraphes est suivi ensuite d'un commentaire qui résume en une ligne le contenu de ce paragraphe. Cela permet d'avoir une vue un peu plus générale du document, de mieux suivre le flot des idées, la structure du document. Le *tag* `plan` indique que le paragraphe qui suit ne propose pas de nouvelles idées mais est juste là pour guider le lecteur dans le document.

Ces annotations, en plus de servir à l'auteur du document pourraient servir aussi au lecteur. Certains commentaires (ce qui est le cas pour ce document de thèse) peuvent par exemple contenir des explications supplémentaires, une alternative, une autre piste, mais l'on ne les place pas dans le texte directement car elle emmènerait le lecteur trop loin. Le texte final d'un document est le résultat d'un choix où l'on privilégie un axe et où l'on évite certaines digressions qui pollueraient l'idée principale à faire passer, qui noieraient le lecteur. Cependant, on pourrait proposer ces discussions supplémentaires un peu comme des «bonus». Certains livres contiennent d'ailleurs dans le texte certaines de ces digressions dans un encart spécial afin de prévenir le lecteur. Les vues LFS fournissent le moyen d'accéder à ces bonus.

Dans un registre très différent, LaTeX permet à l'utilisateur de diviser son document en différents fichiers, par exemple un fichier par chapitre, et offre des commandes pour ensuite réassembler tous ces fichiers dans un seul document. Cela permet à l'utilisateur de naviguer plus rapidement dans son document ; plutôt que de *scroller* il suffit d'ouvrir le fichier adéquat. Cependant, on favorise un axe, la division en chapitres, et il devient ensuite impossible par exemple d'avoir un fichier contenant l'ensemble des introductions. De plus, cette division est souvent handicapante car de nombreux outils ou opérations d'un éditeur de fichiers ne traversent pas ou mal les frontières de fichiers. Ainsi, suite à la division en chapitres, on ne peut par lancer une recherche globale ou une opération de substitution avec son éditeur sur l'ensemble du document. Cela est très pénalisant pour par exemple maintenir une cohérence typographique. En fait ce problème de tension entre garder ensemble ou diviser se retrouve même avec le chapitre puisque l'utilisateur pourrait aussi diviser ce chapitre en différents fichiers, un par section cette fois. La division en fichiers est souvent le résultat d'un compromis. Ainsi, nous pensons qu'il est préférable de garder tout le document dans un seul fichier pour ensuite sélectionner les tranches que l'on veut éditer. On bénéficie ainsi de tous les avantages sans les inconvénients à la fois de la division et de la réunification.

Comme les vues sont des objets systèmes standards (des fichiers), elles peuvent être utilisées par de nombreuses applications en même temps, applications qui peuvent même ne pas se connaître les unes les autres. Par exemple, il devient facile avec LFS de faire coopérer LaTeX et un vérificateur orthographique comme `spell` dans Emacs. Le vérificateur ne comprend ni les commandes LaTeX ni la notion de commentaire, et peut donc afficher malheureusement de nombreux messages d'erreur lorsque on l'applique sur un fichier LaTeX. On pourrait bien sûr «delatexiser» un fichier LaTeX en filtrant par exemple avec `grep` toutes ces commandes, puis passer ensuite ce résultat à travers le vérificateur. Cependant, la correction de ce fichier résultat ne corrigera pas le fichier original, celui sur lequel on veut justement travailler. Si à la place on utilise un transducteur qui permet à l'utilisateur de cacher ces commandes LaTeX, par exemple avec la commande `emacs !latex/thesis.tex`, la vue résultante pourra aussi être passée à travers le vérificateur et la modification de cette vue corrigera aussi le fichier original ; LFS maintiendra la cohérence automatiquement.

Nous pensons que LFS favorise ainsi la combinaison d'outils. Dans le cas précédent, l'utilisateur n'a pas à coder un mode spécial Emacs permettant de combiner `spell` et LaTeX ; il suffit juste d'utiliser la bonne vue et la combinaison devient alors possible. Dans le passé, considérer les fichiers comme un ensemble plat de caractères a été une abstraction fructueuse permettant la combinaison d'outils via les *pipe*, les redirections. LFS permet de retrouver de la structure tout en permettant encore la combinaison fructueuse d'outils et en permettra certainement de nouvelles.

Nous pensons aussi que LFS peut rendre plus rapide le processus d'écriture. En effet, la possibilité de mieux voir la structure du document, de pouvoir se concentrer plus facilement sur une tâche en sélectionnant la vue adéquate ou encore de pouvoir facilement modifier de manière cohérente le document peut faire gagner un temps précieux. La thèse est un objet digital très complexe et il est important d'avoir des outils puissants.

Si cependant ce document n'est pas cohérent, c'est surtout le fait d'un défaut de l'auteur et pas de l'outil. LFS ne rend malheureusement pas plus intelligent, mais aide et soulage un peu l'écrivain. L'approche LFS se généralise à d'autres types de documents digitaux complexes ; la section suivante en montre un autre, le source d'un programme avec le code de LFS.

6.2.8 Le code LFS

La dernière expérimention consiste à naviguer à l'intérieur du source LFS. Une partie de ce source est présentée en Annexe B. Un peu comme dans le domaine des compilateurs, il y a une sorte de *bootstrapping* puisque l'on utilise LFS pour coder LFS.

Les propriétés extraites des fichiers par le transducteur avancé O'Caml et Perl sont comme pour le transducteur C décrit précédemment le nom des fonctions composant ce fichier (`function:dirs`), mais aussi les définitions de types (`deftype:context`) et de variables (`var:default_world`). Le transducteur O'Caml extrait aussi le type des fonctions, par exemple `type:formula -> (bool * int)` pour une fonction prenant en paramètre une formule et retournant une paire composée d'un booléen et d'un entier. Ces propriétés sont gérées par un moteur de déduction logique sur les types qui permet ensuite par exemple les requêtes `cd type:?formula/` ou `cd type:!bool/` qui permettent respectivement de récupérer l'ensemble des fonctions prenant un paramètre (avec le symbole `?`) une formule ou les fonctions retournant (avec le symbole `!`) un booléen. Le codage de LFS suit une discipline proche de la programmation littéraire et les fonctions sont ainsi groupées en sections comme le montre la table des matières de l'Annexe B. Le transducteur O'Caml extrait donc, comme pour le transducteur LaTeX, le nom des sections (`section:Plug-ins`) et sous-sections (`subsection:Transducteur`) auquelles appartient la fonction ou variable. Les commentaires suivent tout comme la thèse une certaine discipline. Le code des différentes optimisations sont de plus *taggées* afin que l'on puisse voir l'ensemble du code mis en jeu pour cette optimisation mais aussi pour pouvoir parfois cacher ce code pour se concentrer sur le code d'origine plus simple. La plupart de ces commentaires et certains aspects ont été cachés dans l'Annexe B, qui présente donc en fait une vue légèrement simplifiée du source.

Certaines propriétés, par exemple `function:dirs` sont assignées à un ensemble de lignes consécutives, ici toutes les lignes de la fonction, d'autres juste à une ligne, par exemple les propriétés liées aux aspects.

Les exemples suivants montrent des requêtes possibles :

```
[1] % cd /lfs/.ext; ls
lfs.ml spec.ml lfsperl.pm
string_logic.ml
int_logic.ml
...
[2] % cd parts; ls
function:/ deftype:/ defvar:/
type:/ aspect:/ var:/ field:/
documentation:/ header/
section:/ subsection:/
lfs.ml spec.ml lfsperl.pm
...
[3] % ls aspect:^/
aspect:error/ aspect:fault-tolerance/
aspect:security/ aspect:specification/
aspect:debugging/ aspect:logging/
aspect:optimisation/ aspect:version/
lfs.ml lfsperl.pm ...
[4] % ls section:^/
section:Structure de données/
section:Algorithmes/
section:Plug-ins/
section:Le shell/
lfs.ml lfsperl.pm ...
[5] % ls type:?formula/
type:?context/
type:!bool/
type:!property set/
...
function:ext/ function:dirs/
function:files/ function:view/
...
[5] % cat type:!formula/lfs.ml
.....................:1
let (pwd: unit -> formula) = fun () ->
  fst3 (top !w.pwd_history)
.....................:2
[6] % ls field:pwd_history/function:^;
function:pwd/ function:cd/
function:dopath/ function:ls/
[7] % cat function:mkdir/!aspect:/
        lfs.ml
.....................:1
let (mkdir: string -> unit)= fun name->
  let p = Prop name in
  let pwd = pwd () in
.....................:2
  let ps = properties_of_formula pwd ...
  let ip = new_iprop p in
  w := {!w with
    graphp = (!w.graphp #add_node ip)
              +> (fun g -> ps ...
    iprop_prop =
       !w.iprop_prop#add (ip, p);
    prop_iprop =
       !w.prop_iprop#add (p, ip);
    extfiles =
       !w.extfiles#add (ip, empty());
       }
.....................:3
[8] % cd header&section:Shell/
[9] % cat lfs.ml
.....................:1
let (pwd: unit -> formula) = fun () ->
.....................:2
let rec (cd: path_element -> unit) = ...
.....................:3
let (ls: unit -> ((property * file) set) ...
.....................:4
...
[10] % sed -e s/formula/Formule/g
        lfs.ml > lfs.ml
[11] % cat ../type:!Formule/lfs.ml
.....................:1
let (pwd: unit -> Formule) = fun () ->
  fst3 (top !w.pwd_history)
.....................:2
```

Nous avons aussi appliqué LFS aux fichiers d'interface de la librairie standard O'Caml. Cela permet de plus facilement retrouver une fonction. Prenons l'exemple, d'un utilisateur qui aimerait trouver une fonction retournant une sous-chaîne d'une chaine mais qui se rappelerait juste qu'elle commence par `sub`. Avec LFS, la requête `cd function:/` affiche l'ensemble des fonctions disponibles. Avec la *completion* il peut se rendre compte que 20 fonctions commencent par `sub` (en tapant `TAB` sous le *shell* apres le début de requête `cd function:sub`). Même si cette liste de résultat est pe-

tite, l'utilisateur peut préférer avec la logique de chaîne taper la requête `cd function:sub.*/` pour pouvoir affiner sa rechercher selon d'autres critères tout en exprimant ce qu'il sait déjà. De là, LFS propose un ensemble d'incréments comme par exemple le type de ces différentes fonctions. Se rendant alors compte que la fonction qu'il cherche prend forcément une chaîne de caractère en paramètre, l'utilisateur peut avec la requête `cd type:?string/` réduire les possibilités et voir que seul deux fichiers de la librairie satisfont cette requête dont le fichier `str.ml`. Enfin, la commande `cat str.ml` permet d'afficher la vue d'un de ces deux fichiers ne contenant bien sûr que la partie satisfaisant la requête c'est à dire l'interface et la documentation de la fonction O'Caml `substring` (CQFT, ce qu'il fallait trouver).

Discussions

Nous venons de présenter différents types de données et l'intérêt d'utiliser à chaque fois LFS pour gérer ces données. Un point important est que si les fonctionnalités de LFS peuvent paraître intéressantes en théorie, comme le fait d'avoir la négation, la disjonction, des axiomes, d'assigner des formules à des fichiers, d'avoir des moteurs logiques, ou encore de voir des formules comme incréments, elles le sont aussi en pratique.

Nous avons vu aussi différents logiciels et vu comment LFS concurrençait et rendait même plus de services que ces logiciels à l'utilisateur. Nous avons pu voir aussi l'omniprésence des structures hiérarchiques, dans les livres, dans les logiciels de messagerie, dans la programmation. La gestion de ces données souffrait ainsi des mêmes problèmes. Ces logiciels possèdent des fonctionnalités communes, et l'on retrouvait des besoins d'organisation, de recherche, d'interroger et naviguer, et de mettre à jour, avec cependant à chaque fois de nombreuses limitations. Cela illustre bien que ces concepts sont fondamentaux et c'est en cela qu'il est important d'en faire l'ossature d'un système d'exploitation. Il est important ainsi de régler ces problèmes une fois pour toute, et de fournir et factoriser les réponses à ces problèmes au plus bas niveau, sinon, on se condamne à réimplémenter à chaque fois une partie des fonctionnalités de LFS dans chaque logiciel, avec très certainement des limitations. De plus, le jour où l'on rajoutera une autre fonctionnalité à LFS, toutes ces données pourront ainsi en bénéficier. Au contraire, l'ajout de cette fonctionnalité dans un logiciel comme Netscape n'aura aucun impact sur la commande `apropos` et sur la gestion des pages `man`.

Nous pensons que LFS en plus de permettre une meilleure organisation des fichiers peut aussi grandement influencer la façon d'écrire, de programmer, de lire, ou encore d'organiser son travail.

Cette démonstration d'utilité, montrant des exemples de requêtes intéressantes sur des données très différentes, reste cependant très modeste. Une véritable démonstration nécessiterait un panel d'utilisateurs confrontés à LFS à qui l'on aurait soumis un ensemble de tâches, considérées comme typiques. Trouve-t-on mieux un fichier avec LFS ? Est-ce plus facile ? On pourrait par exemple observer si par le biais des incréments un utilisateur converge vite lors de sa recherche d'une page `man`. Est ce que du fait de la navigation, et donc de la possibilité de voir des propriétés complexes contenant des formules, l'utilisateur devine plus facilement des types de requêtes possibles acceptés par un moteur logique ? Est ce que l'utilisateur n'est pas noyé par trop d'incréments ? Il n'y a pas actuellement de réponses à ces questions si ce n'est notre expérience puisque nous utilisons LFS de façon journalière et nous trouvons effectivement LFS très pratique.

6.3 Benchmarks

Dans ce qui suit, nous allons désormais nous intéresser non plus à l'aspect qualitatif mais à l'aspect quantitatif. La Table 6.1 présente des statistiques sur les contextes qui serviront dans les expériences présentées après[10].

Pour ces différents types de données, nous avons essayé tant que possible d'avoir à la fois une petite et une grande collection de ces données (selon les expériences un petit et grand nombre de fichiers ou un petit et grand nombre de lignes dans un fichier). Pour LaTeX, nous avons ainsi un article de recherche dévrivant une partie de LFS, petit, et la thèse LFS, grande. Pour les sources

[10] Pour certaines expériences comme avec le noyau Linux, nous naviguerons à la fois parmi et à l'intérieur des fichiers, comme décrit Section 5.3.1 page 135, d'où un nombre d'objets supérieur au nombre de fichiers dans la table.

de programmes, le projet Andrew et les sources du noyau Linux. Pour le BibTeX, des références bibliographiques collectées par l'auteur et des références bibliographiques collectées par la communauté scientifique travaillant autour des langages de programmation. Le but est qu'avec des expériences menées sur les mêmes genres de données, on puisse se faire une première idée de la complexité des opérations LFS en observant comment leurs coûts évoluent lorsque la taille de la collection augmente.

	Ext3	Hierar/PerlFS	LFS
Mkdir	0.7s	0.6s	0.6s
Copy	1.5s	3.6s	20.3s
Scan	2.3s	4.1s	7.4s
Read	5.3s	9.1s	17.0s
Make	20.0s	27.0s	33.7s
Total	30.0s	44.4s	79.0s

TAB. 6.2 – Benchmark Andrew

6.3.1 Benchmark Andrew

Le benchmark Andrew [KMN+88] est considéré comme le benchmark classique pour tester les systèmes de fichier. Ce benchmark a 5 phases : *Mkdir* construit une hiérarchie de répertoires, *Copy* copie des fichiers, *Scan* traverse récursivement toute la hiérarchie, *Read* lit tous les octets d'un fichier, et finalement *Make* compile des fichiers. Le but du benchmark Andrew est qu'avec ces 5 phases, considérées comme représentatives des activités d'un utilisateur Unix, on puisse tester la rapidité de l'ensemble des opérations d'un système de fichier.

Les fichiers mis en jeu sont les sources C d'une application Unix, là aussi considérée comme un projet représentatif. En fait, afin que les temps de calcul ne restent pas négligeables, et donc pour pouvoir évaluer plus précisément LFS, nous avons copié 10 fois les données du benchmark. Il en résulte alors un ensemble de 860 fichiers. Ainsi, on ne lancera par exemple pas une mais 10 compilations, une sur chaque instance. Cette variante du benchmark est connu sous le nom de benchmark Andrew modifié.

Il faut noter que nous n'avons rien eu à modifier pour faire fonctionner le benchmark sous LFS. Les données de ce benchmark, bien qu'organisées dans une hiérarchie, ont été copiées tel quel. En effet, LFS ne change que la sémantique des commandes *shell*, par leur interface Ainsi, l'utilisation de `make` ou `gcc` ainsi que la création de fichiers qui résulte de l'utilisation de ces commandes marchent parfaitement sous LFS.

Puisqu'il n'y a pas de système de fichier similaire à qui se comparer, nous opposerons LFS aux systèmes de fichier hiérarchiques. Puisque le prototype LFS se base sur EXT2 et PerlFS, nous évaluerons donc le surcoût apporté par LFS par rapport à ces systèmes de fichier. Ainsi, nous avons lancé le benchmark Andrew modifé d'abord sur le système de fichier natif (EXT2), puis sur un système de fichier hiérarchique implémenté via PerlFS (Hierar/PerlFS), puis enfin sur LFS. La Table 6.2 présente les résultats de ces expériences.

Nous aurions pu aussi comparer LFS aux systèmes de fichier avancés décrit Section 1.4 comme SFS [GJSJ91], même si ils ne sont pas non plus vraiment comparables à LFS puisqu'ils n'offrent qu'un sous-ensemble des fonctionnalités de LFS. Cependant, aucun de ces systèmes de fichier n'est disponible. Ils sont tous restés au stade de maquette privée.

Bien sûr, comme le montre la Table 6.2, LFS est plus lent qu'un système de fichier hiérarchique sur ce genre d'expérience. Ce dernier ne fait quasiment rien alors que LFS doit indexer les propriétés (`function:`, `size:`, `ext:`, etc) et calculer les incréments dans les répertoires. La question n'est donc forcément pas de combien de pourcents LFS améliore l'efficacité des systèmes de fichier mais plutôt si le surcoût pour pouvoir bénéficier des nouveaux services de LFS est tolérable. Les systèmes de fichier hiérarchiques sont certes très rapides mais ils n'offrent que très peu de services. Les nouveaux services de LFS devraient justement permettre plus tard de faire gagner du temps à l'utilisateur lors de ses recherches avec les incréments ou lors de son travail sur un document avec les vues. Ce gain là est plus difficile à quantifier.

De plus, si l'on n'omet la phase de création de fichier, même sur cette expérience LFS n'est pas beaucoup plus lent. En fait, le «goulot» n'est même pas sur LFS mais sur le processeur puisque c'est le temps de compilation qui domine le temps total. Sur cette phase EXT2 n'est

	NbObj	Size	Loc	Prop	Prop/Obj
MP3	1108f	2968Mo	31	3256	11
Pages `man`	11743f	249Mo	20	15204	9
E-mails	42816f	358Mo	55	192707	16
Andrew (petit)	76f/12814l	560Ko	6/38	347/2259	7/3
Linux (grand)	214f/127912l	3620Ko	6/38	3459/26467	18/4
Code LFS	69f/13045l	972Ko	0/174	174/8003	4/6
BibTeX (petit)	8184l	280Ko	33	7015	9
BibTeX (grand)	100000l	3716Ko	33	37077	10
Article LFS (petit)	1856l	88Ko	114	2038	8
Thèse LFS (grand)	22260l	1188Ko	114	9779	15

NbObj : Nombre d'objets (nombre de fichiers et/ou de lignes) **LoC** : Taille du transducteur et/ou du transducteur avancé en lignes de code ; **Size** : Taille du ou des fichiers ; **Prop** : Nombre total d'attributs ; **Prop/obj** : Nombre d'attributs moyen par objet (fichier et/ou partie de fichier).

TAB. 6.1 – Statistiques sur les contextes des expériences

pas beaucoup plus rapide que LFS. La phase de création prend beaucoup plus de temps car elle implique l'appel au transducteur C. Enfin, même si certaines opérations sont 10 fois plus lentes sous LFS, on passe en général d'un temps individuel pour chaque commande de 0.01 seconde à 0.1 seconde. Dans un contexte interactif, ces différences n'ont donc aucune importance.

Cependant, ce benchmark fut inventé pour tester les systèmes de fichier hiérarchiques. Ces systèmes de fichier sont trop différents de LFS pour que ce benchmark ait encore un sens. Le benchmark ne stresse donc pas ce pour quoi LFS a été conçu. Par exemple, le nombre de fichiers considérés dans cette expérience reste petit. Sous un système de fichier hiérarchique, le nombre de fichiers total n'a pas d'impact sur le coût des opérations. Seul peut compter le nombre de fichiers dans le répertoire où sont lancés les commandes ; les «calculs» sont localisés. Ce n'est plus le cas sous LFS. De plus, même si nous avons utilisé un transducteur C, le benchmark ne connaît pas la notion de propriété ou de logique. Ainsi, il n'affecte pas de multiples propriétés à un fichier, et se contente de créer une simple hiérarchie. Enfin, soit le benchmark ne stresse pas LFS, soit il le stresse mal. Ainsi, la phase *Scan* utilise l'utilitaire `find` alors qu'il est devenu coûteux et inutile sous LFS. Ces chiffres ne sont donc pas suffisants pour analyser les performances de LFS.

6.3.2 Benchmark sur l'organisation

Cette section ainsi que les deux suivantes présentent des benchmarks qui stresseront vraiment LFS. Dans cette section nous évaluerons les coûts LFS en ce qui concerne l'organisation, c'est à dire les temps de création de fichiers et l'espace disque utilisé par ces fichiers. Les deux sections suivantes évalueront les coûts LFS en ce qui concerne la recherche et la manipulation.

Espace disque

La première expérience évalue le surcoût en espace disque dû au stockage des métas-données LFS (voir Table 6.3). En effet, les fichiers ont désormais sous LFS de nombreuses propriétés et l'efficacité de LFS dépend des structures d'indexation autour de ces propriétés, comme le *cache logique* ou la *table des extensions* (voir Section 3.2.1 page 106). Ces structures prennent de la place sur le disque.

Pour les expériences concernant les parties de fichier, le surcoût LFS est important. Cependant, les tables créées pour gérer ces données (la matrice *ligne* $\times$ *propriété*) ne sont que temporaires. En effet elles sont calculées à partir du contenu du fichier, et peuvent donc être recalculées. De plus, un utilisateur ne naviguera pas à l'intérieur de tous ses fichiers. Il peut donc se permettre de stocker pour quelques fichiers le contexte de ces fichiers afin de bénéficier de l'ensemble des services offerts par LFS, notamment les vues. La taille de ces

	NbObj	DataSize	MetaSize	SpOverH%
MP3	1108	2968Mo	1.6Mo	0.05%
Pages `man`	11743	249Mo	8.8Mo	3%
E-mails	42816	351Mo	103.4Mo	29%
Andrew (petit)	76/12814	560Ko	48Ko/3248Ko	8%/× 6
Linux (grand)	214/127912	3620Ko	1484Ko/30704Ko	40%/× 8
Code LFS	69/13045	972Ko	48Ko/6640Ko	4%/× 7
BibTeX (petit)	8184	280Ko	4532Ko	× 16
BibTeX (grand)	100000	3716Ko	35064Ko	× 9
Article LFS (petit)	1856	88Ko	1320Ko	× 15
Thèse LFS (grand)	22260	1188Ko	9588Ko	× 8

NbObj = Nombre d'objets (nombre de fichiers et/ou lignes), **DataSize** = Taille total du ou des fichiers, **MetaSize** = Taille total des tables LFS, **SpOverH%** = Surcoût (pourcent).

TAB. 6.3 – Benchmark pour l'espace disque

contextes reste de plus négligeable si l'on considère l'espace disque disponible sur les disques durs modernes.

L'important dans la Table 6.3 se trouve donc surtout dans les surcoûts liés aux fichiers, avec des tables qui doivent être persistentes (la matrice *fichier* × *propriété*). La table montre dans ce cas que le surcoût en espace disque reste tolérable. De plus, selon notre expérience le nombre de propriétés par fichier reste globalement constant quel que soit la taille du fichier et le type de ce fichier. Ainsi, plus la taille d'un fichier augmente, plus la place prise par les métas-données devient négligeable par rapport à la taille du fichier. La tendance actuelle faisant qu'un utilisateur a des fichiers de plus en plus gros (en particulier depuis l'explosion des données multimédia), le surcoût apporté par LFS deviendra dans le futur de plus en plus négligeable.

Temps de création

La deuxième expérience évalue le surcoût en temps lié aux appels aux transducteurs et à l'indexation des propriétés qui résulte de la création de nouveaux fichiers (voir Table 6.4). Comme pour le benchmark d'Andrew, nous comparons LFS à EXT2 et PerlFS. L'expérience consiste à copier un ensemble de fichiers sous LFS.

Pour les données concernant des parties de fichier (par exemple les fichiers BibTeX), nous évaluons le temps mis par LFS pour rentrer la première fois à l'intérieur du fichier suite à la commande `cd parts` (voir Table 6.5). En effet, là aussi il en résulte un appel à des transducteurs (avancés) ainsi qu'une indexation ; les opérations mises en jeu sont donc similaires aux opérations ayant lieu lors de la copie d'un grand nombre de fichiers.

	NbObj	`cd parts`
Andrew (petit)	12814	12.3s
Linux (grand)	127912	213.1s
Code LFS	13045	30.0s
BibTeX (petit)	8184	9.7s
BibTeX (grand)	100000	328.2s
Article LFS (petit)	1856	2.5s
Thèse LFS (grand)	22260	57.3s

NbObj = Nombre d'objets (nombre de lignes).

TAB. 6.5 – Benchmark pour les temps de création d'un contexte de fichier(s)

Pour les expériences concernant les parties de fichier, le surcoût LFS, ou plutôt le coût LFS puisqu'aucun autre système n'offre ce type de service, peut être très important. Par exemple, le temps de réponse de la commande `cd parts` dans un répertoire contenant un fichier Bib-

	NbFile	EXT2	Hierar/PerlFS	LFS	TimeOverH%
MP3	1108	4min36	7min30	9min30	$\times$ 2
Pages `man`	11743	1min29	2min33	17min20	$\times$ 11
E-mails	42816	10min20	12min08	193min	$\times$ 18
Andrew	76	0.2sec	0.4sec	5.1sec	$\times$ 25
Linux	214	0.4sec	0.9sec	14.8sec	$\times$ 37
Code LFS	69	0.2sec	0.5sec	1.50sec	$\times$ 7

NbFile = Nombre de fichiers), **TimeOverH%** = Surcoût total (pourcent).

TAB. 6.4 – Benchmark pour les temps de création de fichiers

TeX de 100 000 lignes est ainsi de 5min30. En effet, cette commande implique en interne la création d'un nouveau contexte avec un grand nombre d'objets ainsi que l'indexation d'un grand nombre de propriétés. De plus, contrairement aux expériences qui créent des fichiers, et donc aussi des objets, cette indexation est «pure» puisqu'il n'y a même pas en jeu la copie d'un contenu et donc d'entrées/sorties sur le disque comme lors de la copie des fichiers MP3. C'est donc le type de situation qui stresse le plus LFS.

Cependant, un point important est que les contextes de ces fichiers sont sauvegardés sur le disque[11]. Ainsi, seul le premier `cd parts` prends du temps. Si l'utilisateur désire un peu plus tard renaviguer dans le même fichier BibTeX, le temps de réponse de `cd parts` sera quasi nul. De plus, un utilisateur ne naviguera pas à l'intérieur de tous ses fichiers. Il peut donc se permettre afin de bénéficier de l'ensemble des services offerts par LFS d'attendre un peu la première fois. Nous pensons de plus que cette perte de temps sera comblée ensuite par les gains de temps qu'apportent les vues, l'interrogation et la navigation. Les services concernant les parties de fichier sont essentiellement là pour aider le travail de longue haleine comme la programmation ou l'écriture d'un document. Ils sont là pour aider l'utilisateur à mieux comprendre, à mieux se concentrer, et à modifier de manière cohérente un document. Ce premier contretemps est donc négligeable si l'on considère le temps total que l'on passera sur ce document.

En ce qui concerne la copie de fichiers, l'expérience représente un cas extrême puisque normalement les fichiers (e-mails, musiques) arrivent au fur et à mesure sur la machine. Ce surcoût devrait donc s'étaler sur une période beaucoup plus grande. Cependant, même sur ce cas extrême, il reste tolérable. C'était l'une de nos plus grandes inquiétudes puisque LFS doit indexer au fur et à mesure de plus en plus d'objets et de propriétés. La compression en intervalle alliée au fait que sur des données réelles les fichiers partagent de nombreuses propriétés rendent cette indexation rapide. Nous verrons plus loin que l'évolution en temps est presque constante. Ainsi, l'insertion du 1000ème fichier est plus lente que l'insertion du 100ème mais pas de beaucoup. LFS n'explose pas en complexité.

Il faut noter que cette indexation est faite en *temps-réel*. Le contenu des répertoires est à chaque fois cohérent avec le contenu des fichiers ainsi que l'organisation de ces fichiers. Au contraire, les systèmes de fichier avancés décrit Section 1.4 supportant l'idée de transducteurs (comme SFS) proposaient tous une indexation journalière. Ainsi, le contenu des répertoires virtuels sous ces systèmes ne traduisait pas forcément la réalité du moment. La copie de nouveaux fichiers n'était pas répercutée sur l'organisation. Ce choix traduisait peut-être aussi une limitation de ces systèmes qui n'autorisaient pas la création de nouveaux fichiers dans ces répertoires virtuels. Nous pensons cependant que même si cette cohérence a un prix, c'est une caractéristique importante de LFS. Sous LFS, l'utilisateur peut assigner aussi manuellement des propriétés à un fichier. Cet utilisateur s'attend à ce que le contenu du répertoire où l'on vient de rajouter un fichier soit mis à jour. Les propriétés assignées automatiquement ne sont pas différentes du point de vue LFS

[11] Dans une zone certes temporaire, mais sauvegardés tout de même.

des propriétés assignées manuellement. C'est d'ailleurs une des contributions de LFS. Il n'y a donc pas de raison que les répertoires concernant de telles propriétés ne soient pas mis à jour après chaque modification. .

La copie de fichiers MP3 montre que lors d'expériences mettant en jeu des entrées sorties de données massives, le goulot est plus dans PerlFS que dans LFS. En effet, les données doivent passer à travers le *pipe* (voir Section 6.1.1 page 142). Cependant, comme expliqué Section 6.1.2 page 146, ce surcoût pourrait être évité en spécialisant légèrement PerlFS en plaçant le code des opérations d'entrées sorties directement dans le noyau. Dans ce cas, le temps de copie devrait être de 6min36 ce qui ne présenterait plus qu'un surcoût de 43% par rapport à EXT2, le surcoût dû à l'indexation pure. Ainsi, plus les données sont grosses, plus le surcoût LFS baisse puisque le temps d'indexation devient négligeable sur le temps global de copie. Le goulot passe de LFS au disque dur.

6.3.3 Benchmark sur la recherche

Dans cette section nous évaluerons les coûts LFS en ce qui concerne la recherche, c'est à dire les opérations d'interrogation et de navigation. Nous évaluerons donc le temps de réponse des commandes `cd` et `ls`.

Temps de réponse de `ls`

Pour évaluer correctement le temps de réponse moyen de `ls` il faudrait un modèle d'usage traduisant le type de commande typique qu'effectue un utilisateur, tout comme il faudrait un modèle de donnée traduisant le type de donnée typique. Nous n'avons cependant pas ces modèles. C'est pour cette raison que nous testons LFS sur les différents types de données présentés précédemment, et dans différentes configurations. La commande `ls` est ainsi à chaque fois utilisée dans deux situations différentes. La première pour formuler des requêtes très vagues retournant de nombreux incréments. Par exemple, pour les MP3 `ls artist`: retourne plus de 200 incréments. La deuxième pour formuler des requêtes plus précises retournant donc moins d'incréments. Par exemple, `ls year:1998/ !genre:Disco/artist:/` retourne 40 incréments. La Table 6.6 montre les résultats des temps moyen que met `ls` pour ces deux situations avec différentes commandes `ls` sur l'ensemble des données.

Les résultats sont satisfaisants puisqu'ils sont la plupart du temps de l'ordre de la seconde. Il faut noter que seule la première exécution de la commande `ls` dans un nouveau répertoire prend du temps. Une fois calculés les incréments et fichiers de ce répertoire, ce résultat est caché en interne et contrôlé par une *estampille* (voir Section 3.2.2 page 107). Ainsi, si aucun fichier n'est ajouté ou supprimé entre deux exécutions de `ls`, le deuxième `ls` dans ce même répertoire est instantané.

Temps de réponse de `cd`

Dans les expériences précédentes, le temps pour aller dans le répertoire voulu avec `cd` est à chaque fois quasi nul. Seul `ls` prend du temps. En effet, les requêtes ne concernaient que des propriétés déjà existantes, et des formules conjonctives, disjonctives ou négatives sur des propriétés ou attributs valués déjà présents. La spécialisation de la logique du *noyau* LFS à une logique d'ensemble (avec `|`, `!`, `&` ou `/`) ainsi que l'utilisation du cache logique pour les axiomes et autres logiques évitent tout appel aux moteurs de déduction.

Si l'utilisateur mentionne cependant une nouvelle *propriété-formule*, le temps d'exécution de `cd` est forcément plus long car il implique cette fois d'appeler un moteur de déduction logique et d'insérer cette propriété dans le cache (voir Section 3.1.1 page 92). Cependant, ce temps est la plupart du temps négligeable. La commande `cd year:<1980/` pour les fichiers MP3 prend ainsi 0.06 secondes, et `cd size:[4..5]Mo/` prend 1.66 secondes. Il faut noter que seule la première mention d'une nouvelle propriété-formule prend du temps. Une fois insérée dans le cache logique, cette propriété devient une propriété comme une autre. En effet, un deuxième `cd` mentionnant cette propriété impliquera en interne un appel à `lookup` qui vérifiera, comme pour n'importe quelle propriété, si cette propriété existe déjà avec la table de *hash* `propname->prop` (voir Section 3.2.1 page 106), ce qui est très rapide. Le temps d'exécution de `ls` dans un tel répertoire est ainsi le même que dans les autres, car cette propriété est en interne une propriété comme un autre.

Ce premier `cd` peut cependant être problématique dans certains répertoires comme `keyword`: pour les

	NbObj	`ls`
MP3	1108	0.1s
Pages `man`	11743	1.2s
E-mails	42816	4.1s
Andrew (petit)	76/12814	0.1s/0.1s
Linux (grand)	214/127912	0.1s/1.9s
Code LFS	1637	0.1s
BibTeX (petit)	8184	0.1s
BibTeX (grand)	100000	0.9s
Article LFS (petit)	1856	0.1s
Thèse LFS (grand)	22260	0.1s

TAB. 6.6 – Temps de réponse de `ls`

pages `man` où l'on aura de très nombreuses sous-propriétés. La commande `cd keyword:.*flush.*/` avec 10 000 fichiers (et donc 3785 propriétés de la forme `keyword:`x) ne prend cependant que 1.04 secondes. De plus, plus l'utilisateur mentionnera de formules, moins ces commandes prendront du temps. En effet, le cache logique deviendra de plus en plus riche formant une structure ressemblant à un *arbre de décision*. Ainsi, si ce cache contient par exemple déjà la formule `size:<5ko`, l'algorithme d'insertion (voir Section 3.1.1 page 92) évitera déjà toutes les comparaisons avec les propriétés représentant une taille plus petite que 5ko lorsque l'utilisateur voudra insérer la formule `size:[4..5]Mo`. L'algorithme comparera surtout la nouvelle formule avec les formules du haut du cache (les formules les plus générales) et élaguera la plupart des comparaisons avec les formules du bas (les formules les plus spécifiques). D'une certaine manière, le système s'améliore au fur et à mesure et profite de son interaction avec l'utilisateur. De plus, en plus de rendre les futures interrogations plus rapides, la mention de nouvelles formules améliorera aussi la navigation puisque les propriétés seront mieux classées.

Comparaisons

Comme les systèmes de fichier hiérarchiques ne sont pas vraiment conçus pour la recherche d'information, nous comparons maintenant la vitesse du couple `cd/ls` sous LFS avec les outils de recherche comme `find`, `grep` ou `apropos`. Ces outils n'offrent pas la composante navigationnelle de LFS mais ils fournissent une première base pour pouvoir évaluer LFS. En fait, ils n'offrent pas complètement non plus la composante interrogation puisque par exemple la requête `cd year:1998/author:O.Ridoux/` n'est pas exprimable avec `grep`. En effet, l'année et l'auteur d'une entrée BibTeX sont en général placés sur deux lignes différentes.

Avec les fichiers Andrew, trouver le fichier implémentant la fonction `GXfind` prend 2.899 secondes avec `grep` et 0.081 secondes avec LFS. De manière similaire, avec les fichiers MP3 trouver les fichiers musicaux de type Disco prend 3.292 secondes avec un utilitaire à la `grep` et est immédiat sous LFS. L'exécution de `ls keyword:change` avec les pages `man` prend 0.2 secondes et retourne 110 incréments. L'exécution de `apropos change` prend 0.145 secondes mais retourne 288 items. En effet, les propriétés sont souvent partagées par plusieurs fichiers ; elles sont donc moins nombreuses. Les incréments révèlent ainsi déjà un peu l'organisation de ces items. L'exécution de `ls keyword:change/keyword:directory/.ext` prend 0.026 secondes, et retourne 4 entrées. `apropos change | grep directory` prend 0.030 secondes, et retourne les mêmes 4 items. Pour les 40000 e-mails, la commande `ls size:>2Mo` prend 1.3 secondes sous LFS et la commande `find -size`

`+2048k` prend 0.5 secondes sous EXT2.

Pour conclure, sur l'ensemble de nos expériences, avec donc des contextes très différents et des requêtes assez variées, la navigation et l'interrogation sont assez rapides, surtout lorsque l'on considère le travail qu'elles effectuent en interne. C'était l'un de nos plus grands doutes puisque la spécification de *Dirs* et *Files* implique tout de même des calculs assez importants. Les algorithmes et optimisations présentés Section 3.1, ainsi que la puissance des CPU modernes ont permis de rendre ces temps de calcul compatibles avec un usage interactif puisqu'ils sont la plupart du temps de l'ordre de la seconde. De plus, les opérations `cd/ls` se comparent de manière favorable avec des outils de recherche au niveau application comme `grep`.

6.3.4 Benchmark sur la manipulation

La dernière expérience consiste à évaluer le temps que met LFS pour répondre à une commande de sauvegarde qui suit la modification d'une vue. En effet, LFS doit mettre à jour les tables pour que la navigation et les vues restent cohérentes avec ce nouveau contenu (voir Section 2.6.2 page 83). Nous avons à chaque fois fait une petite et une grosse modification. La Table 6.7 montre les temps de mise à jour pour les différents types de données.

Il faut noter que ces temps de sauvegarde comptabilisent à la fois le temps de réindexation et le temps pour regénérer la vue. En effet, suite à une modification la vue doit être raffraichie car son contenu peut avoir changé (voir Section 3.1.3 page 105). Cependant, c'est essentiellement la réindexation qui prend du temps puisque le calcul du contenu d'une vue implique juste le calcul d'une extension, celle du `PWD`, ainsi que la génération du contenu de la vue sur le disque. Cette génération est en général très rapide.

Ces temps de sauvegarde sont tout à fait tolérables. La plupart du temps l'utilisateur ne sera pas perturbé par le travail interne de LFS puisque ce temps de sauvegarde est souvent inférieur au temps que met cet utilisateur pour appuyer sur le bouton de sauvegarde. C'est donc là encore tout à fait compatible avec un usage interactif.

Nous avons aussi indiqué dans la Table 6.7 les temps mis pour rentrer la première fois dans le fichier, ce qui permettait de construire le contexte de ce fichier. L'impact de notre algorithme de réindexation avec les points de synchronisation (voir Section 3.1.3 page 102), peut être ainsi mesuré en comparant les temps pour rentrer la première fois dans le fichier (indexation complète), et les temps de mise à jour (réindexation partielle). La réindexation partielle est 10 à 100 fois plus rapide que l'indexation complète.

Conclusion

Ces expériences sur des vrais données sont encourageantes. Elles montrent que LFS peut être utilisé en pratique. Ses temps de réponse sont compatibles avec un usage interactif.

6.3.5 Évaluation de la complexité

Il est difficile d'établir avec précision les temps de réponse de LFS dans le cas général car il n'y a pas de cas général. En effet, la complexité des opérations de LFS dépend non seulement du nombre des fichiers, mais aussi de leurs types et de leurs transducteurs associés puisque ces fichiers peuvent avoir plus ou moins de propriétés. Cela dépend même du type de ces propriétés qui peuvent être plus ou moins partagés par l'ensemble des fichiers. Chacun de ces critères a un impact sur les temps de réponse de LFS. Il est donc difficile d'établir la complexité théorique des opérations LFS. Nous préférons donc une évaluation plus pratique de cette complexité.

Les expériences précédentes montrent que sur des données classiques, avec un nombre de propriétés par fichier ainsi qu'un taux de partage de ces propriétés considérées comme typique, LFS est efficace. Nous allons dans cette section essayer d'établir plus précisément la complexité des opérations ainsi que les limites du système en faisant varier isolément les différents paramètres qui contrôlent le temps de réponse des opérations LFS.

Les Figures 6.3, 6.4 et 6.5 développent le temps de création pour chaque fichier lors des expériences sur les fichiers MP3, les pages `man` et e-mails. Elles montrent que le temps de création augmente mais n'explose pas en complexité. La Figure 6.6 présente l'évolution du temps de création pour des e-mails de la liste de messagerie Linux (disponible à partir de `lkml.org`) contenant plus

	NbObj	`cd parts`	mise à jour
Andrew (petit)	12814	12.3s	0.2s
Linux (grand)	127912	213.1s	1.1s
Code LFS	1637	30.0s	1.3s
BibTeX (petit)	8184	9.7s	0.3s
BibTeX (grand)	100000	328.2s	2.8s
Article LFS (petit)	1856	2.5s	0.3s
Thèse LFS (grand)	22260	57.3s	1.2s

TAB. 6.7 – Benchmark pour les temps de mises à jour

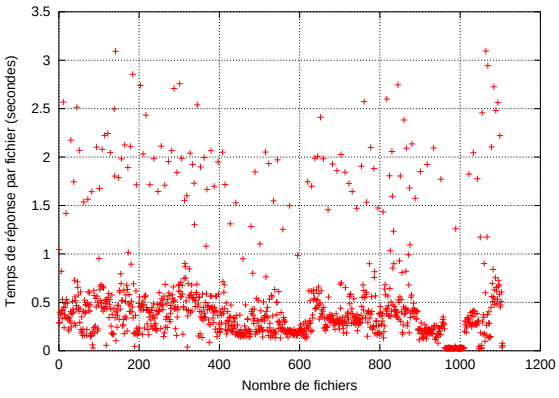

FIG. 6.3 – Temps de création (sec) MP3

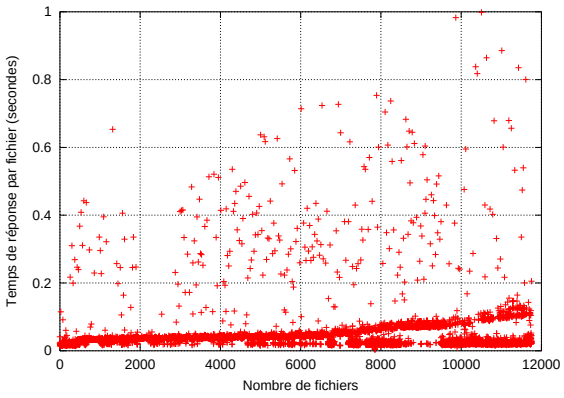

FIG. 6.4 – Temps de création (sec) pages `man`

de 170 000 fichiers. La copie complète et donc l'indexation de ces fichiers sous LFS prend alors 28 heures mais la figure montre là aussi que même jusqu'à 170000 fichiers, LFS n'explose pas en complexité.

Varier le nombre d'objets

Nous allons maintenant développer un peu plus l'analyse précédente en regardant l'évolution non seulement des temps de création, mais aussi des temps de recherche et de modification. Nous n'allons cependant tracer l'évolution de ces opérations que pour les données BibTeX. Nous pensons qu'elles sont représentatives des autres contextes. Cela simplifiera ainsi la présentation des résultats[12]. Nous allons donc mener des expériences avec des BibTeX de plus en plus gros, avec de plus en plus de lignes et donc d'objets.

Nous allons tout d'abord tracer l'évolution du temps de réponse de la commande `cd parts`[13]. Plus il y a d'objets, plus cette commande prend du temps. Sa complexité est donc forcément au moins linéaire en nombre d'objets. Cependant, du fait que cette commande doit calculer les index, elle dépend aussi du nombre de propriétés. Plus il y a d'objets, plus il y a de propriétés (même si elles sont souvent partagées), et plus il faut donc en indexer. De plus, les extensions de ces propriétés contiendront de plus en plus d'objets ce qui implique normalement des opérations d'insertion ensembliste de

[12] Cela réduira aussi mon temps de travail.

[13] Le transducteur avancé générique (avec les propriétés `contains`:) n'est cependant pas appliqué au fichier, d'où un temps final de 115 secondes au lieu des 300 secondes affichées dans les expériences précédentes.

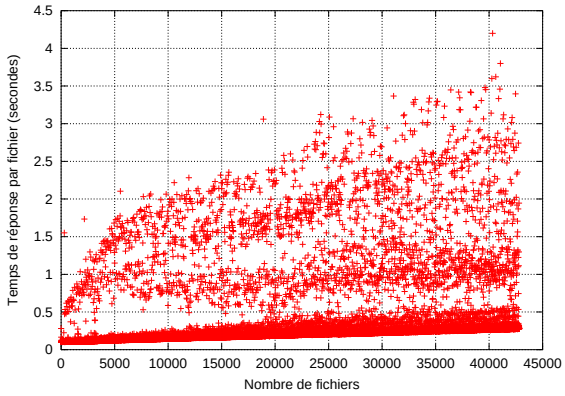

FIG. 6.5 – Temps de création (sec) e-mails

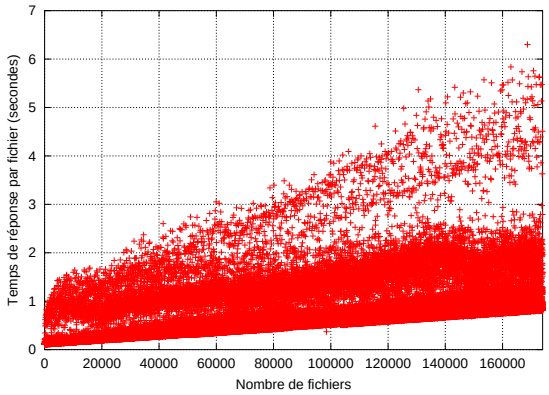

FIG. 6.6 – Temps de création (sec) e-mails liste de messagerie Linux

plus en plus coûteuses et des lectures/écritures de ces extensions sur le disque de plus en plus longues.

Nous tracerons l'évolution du temps moyen de réponse de la commande `ls` dans différents répertoires (avec des requêtes vagues et précises) comme `ls year:1998/author:`. Là encore, plus il y a d'objets, plus il y a de propriétés (même si les objets en partagent un certain nombre), et donc plus il y a d'incréments possibles et de calculs d'intersection d'ensembles. Ces ensembles sont aussi de plus en plus gros.

Enfin, nous tracerons l'évolution du temps moyen de sauvegarde de différentes vues (avec des petites et grosses modifications), par exemple celle du répertoire

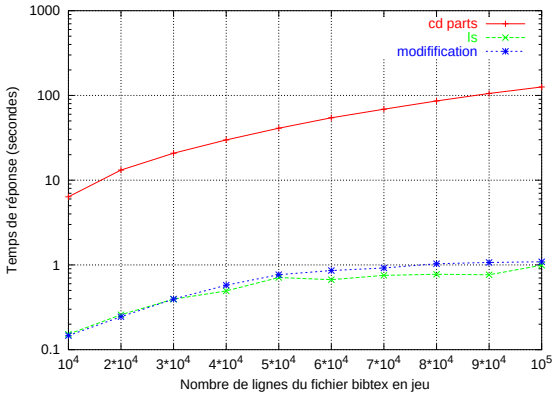

FIG. 6.7 – Corrélation nombre d'objets/temps de réponse (sec)

`year:1998/`. Cette opération implique la suppression et l'ajout d'objets et donc une réindexation de certaines propriétés. Les points de synchronisation réduisent le nombre d'objets à réindexer qui restera donc constant. Cependant, cela n'empêche pas l'insertion et la suppression d'objets dans des ensembles d'objets de plus en plus grand.

La Figure 6.7 présente l'ensemble de ces traces. La ligne rouge correspond à l'évolution des coûts d'organisation (`cd parts`), la bleue des coûts de recherche (`ls year:1998/author:`), et la verte des coûts de modification (`sed s/.../... year:1998/ref.bib`).

Varier le nombre de propriétés

Nous allons maintenant faire varier non plus le nombre d'objets mais le nombre de propriétés par objet. La taille du fichier BibTeX sera ainsi fixe (100 000 lignes) et nous lancerons les expériences avec des transducteurs BibTeX de plus en plus complets. Celui-ci ne renverra au début comme propriétés que l'année et le noms des auteurs d'une entrée (ainsi que les points de synchronisation). Il renverra ensuite en plus le titre, le type, la référence, le domaine et enfin l'institution. La Figure 6.8 présente l'ensemble des traces. Le code couleur est le même que précédemment.

Le temps de création devient plus important lorsque l'on indexe plus de propriétés. Cependant, le goulot étant essentiellement dans la communication et le travail du

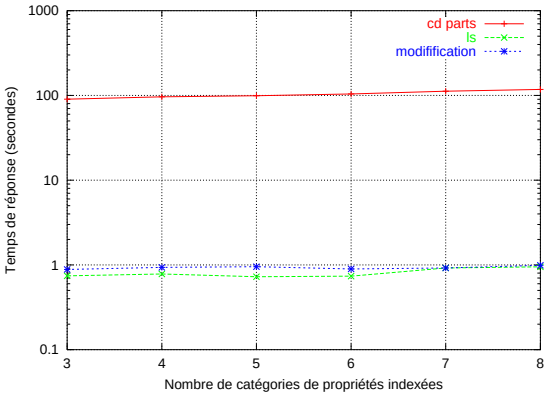

FIG. 6.8 – Corrélation nombre de propriétés/temps de réponse (sec)

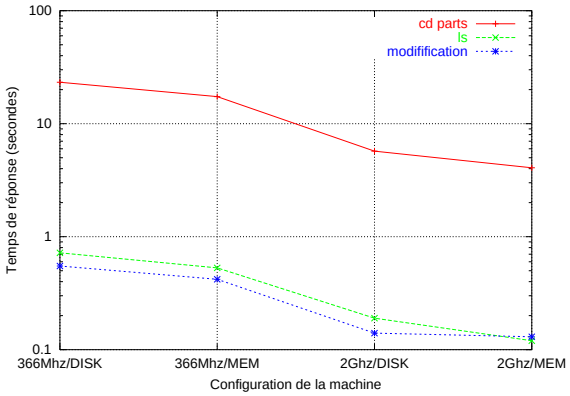

FIG. 6.9 – Impact du processeur et du disque (sec)

transducteur, l'ajout de propriétés n'a que peu d'incidence pour LFS. Le point important est que les propriétés supplémentaires ne viennent pas ralentir la recherche. En effet, LFS propose les propriétés les plus générales lors des recherches. Ainsi, même si une nouvelle propriété est définit, LFS n'affichera d'abord que la catégorie générale (par exemple `domain`:) et donc ne fera qu'un calcul d'intersection supplémentaire.

Varier la machine

On peut aussi évaluer quel est l'impact du processeur et du disque sur les temps de réponse de LFS. Où est le goulot d'étranglement ? Nous avons donc lancé les mêmes expériences sur des disques durs différents, et des CPU différents. Pour les disques, nous avons pris d'abord un disque dur IDE. Nous avons ensuite mené les mêmes expériences en plaçant les tables LFS en mémoire, ce qui peut être assimilé à une expérience avec un disque dur idéal. Pour les processeurs, nous avons pris une machine munie d'un processeur à 366Mhz et la machine utilisée jusqu'à présent dans nos expériences qui possède un processeur à 2Ghz. Comme précédemment, nous n'avons mené ces expériences que sur les données BibTeX, avec cette fois un transducteur complet et un nombre de lignes fixé à 10 000[14]. La Figure 6.9 présente les résultats de ces expériences.

Comme on peut le voir, même si le disque ralentit les opérations LFS, le goulot reste essentiellement sur le processeur.

6.3.6 Évaluation des optimisations

Nous allons dans cette section évaluer la pertinence des algorithmes et des optimisations présentés au Chapitre 3. Nous présentons donc des benchmarks réalisés avec des prototypes LFS ne bénéficiant pas de toutes les optimisations. Nous irons de l'implémentation la plus naïve vers l'implémentation finale de LFS :

1. La spécification formelle de LFS (voir Annexe A) constitue l'implémentation la plus naïve. Elle ne possède ni cache logique, ni table des extensions et n'utilise pas les points de synchronisation.
2. Nous ajouterons donc ensuite à cette implémentation un cache logique (voir Section 3.1.1 page 88), ce qui évite de nombreux appels aux moteurs de déduction, mais en gardant toujours un algorithme ls naïf qui part des objets (voir Section 3.1.2 page 95).
3. Nous ajouterons donc ensuite un algorithme ls plus efficace qui part des propriétés, avec une première table des extensions. On utilise donc l'indexation

[14]Les capacités mémoire de la machine moins puissante étant plus faibles (64 Mo), on ne peut pas lancer l'expérience ou les données sont en mémoire avec le fichier de 100 000 lignes. Cependant, si les données sont sur le disque, comme c'est normalement le cas, LFS marche alors parfaitement sur cette machine avec de gros fichiers.

des propriétés, mais cette indexation reste partielle.

4. Nous ajouterons ensuite l'algorithme `ls` final (voir Section 3.1.2 page 98) avec une table des extensions complète et donc une indexation complète. Chaque entrée de la table des extensions contiendra les objets ayant dans leur description la propriété correspondant à cette entrée mais aussi les objets ayant les sous-concepts de cette propriété (dû à l'*inlining*).
5. Nous ajouterons ensuite la représentation en intervalles pour les extensions. Précédemment les ensembles d'objets (les extensions) étaient représentés naïvement par une liste d'identifiants d'objets. Les intervalles permettent de compresser ces extensions. On arrive alors à ce point à une implémentation efficace pour gérer des fichiers.
6. Enfin, nous ajouterons la gestion des points de synchronisation (voir Section 3.1.3 page 102) ce qui évitera la réindexation naïve et complète d'un fichier lors de la modification d'une partie d'un fichier. On arrive alors à ce point à une implémentation efficace pour gérer les parties d'un fichier, et donc à l'implémentation finale de LFS.

La Figure 6.10 présente les résultats de ces expériences menées sur ces différents prototypes. Nous nous sommes contenté comme dans la section précédente de mener ces expériences sur le fichier BibTeX de 100 000 lignes. Pour les implémentations `1` (la spécification) et `2`, les commandes `ls` et de modifications ne finissant pas sur le fichier de 100 000 lignes, nous les avons lancé respectivement sur un fichier de 1000 et 20 000 lignes.

Comme on peut le voir, chaque optimisation améliore de manière importante la performance de LFS, parfois pour la création, parfois pour la recherche, parfois pour la modification. La seule exception est l'utilisation de l'indexation. En effet, la phase d'organisation prend alors plus de temps qu'avant. C'est un choix conscient puisque nous pensions que la principale opération à optimiser est la consultation. Nous préférons donc transférer la grosse partie du travail dans les commandes de création. La Figure 6.10 nous donne raison puisque le temps perdu à la création du contexte et des index est largement comblé par le temps gagné lors des consultations.

Il faut noter que la compression permet non seulement de gagner en vitesse, puisque les opérations ensemblistes et entrées sorties sont plus rapides, mais aussi en espace disque, puisque la taille des extensions est réduite sur le disque.

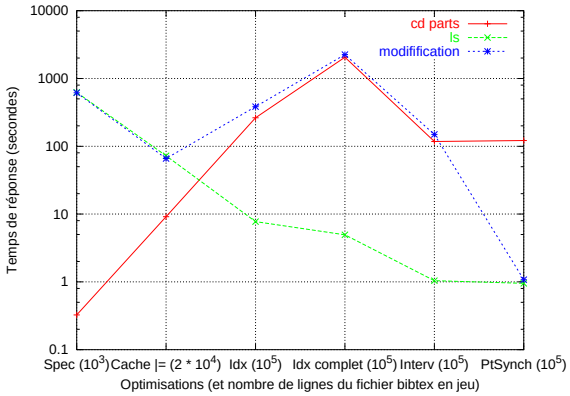

FIG. 6.10 – Impact des optimisations (sec)

6.4 Conclusion

Nous avons utilisé LFS pour gérer une grande variété de données (musiques, e-mails, programmes, etc) et envisagé son utilisation sur un grand nombre d'autres cas. À chaque fois LFS améliore la gestion de ces données en offrant plus de services que les systèmes de fichier hiérarchiques et les outils spécialisés sur ces données.

Nous avons menés de nombreuses expériences stressant les capacités de LFS. Les résultats sont encourageants. LFS peut gérer un grand nombre de fichiers, et des fichiers de grande taille, avec des temps raisonnables. Les opérations ont la plupart du temps des temps de réponse inférieurs à la seconde, ce qui est compatible avec un usage interactif. La limite actuelle de LFS est approximativement de 100 000 objets ; LFS peut gérer efficacement 100 000 fichiers et des fichiers pouvant aller jusqu'à 100 000 lignes.

Conclusion

All resistance is futile, you will be assimilated.

un Borg

LFS was what makes Windows 2030 one step ahead its competitors.

Bill Gates

Résumé

Nous avons présenté au Chapitre 1 les nombreux travaux existants autour des systèmes de fichier. L'une des idées force était que placer une fonctionnalité dans le système de fichier était préférable à la placer dans une application spécialisée car dans le système de fichier cette facilité a plus d'impact et bénéficie plus à l'utilisateur et aux applications.

Nous avons aussi vu dans ce chapitre les premiers signes de faiblesse des systèmes de fichier hiérarchiques. Nous avons présenté des systèmes de fichier avancés qui comblaient certaines de ces faiblesses mais qui en apportaient de nouvelles. Aucun de ces systèmes de fichier n'était vraiment meilleur qu'un autre.

Nous avons ensuite identifié au Chapitre 2 les problèmes sous-jacents de tous ces systèmes de fichier et les raisons profondes qui font qu'ils sont tous condamnés à être limités.

L'une de ces raisons est que tout d'abord certains systèmes, ou plutôt les choix d'organisation de certains systèmes font qu'ils privilégient le principe d'*interrogation*, tandis que d'autres systèmes privilégient le principe de *navigation*. Cependant, chacun de ces deux modes de recherche a ses avantages. Ils sont de plus complémentaires l'un de l'autre puisqu'ils représentent les deux phases dualles d'un dialogue homme-machine. L'utilisateur voudrait donc bénéficier de tous ces avantages dans le même système pour pouvoir retrouver encore plus rapidement un fichier.

L'autre raison est que certains systèmes privilégient l'*assignation manuelle* de propriétés à un fichier tandis que d'autres privilégient l'*assignation automatique* avec une indexation basée sur le contenu de ce fichier. Cependant, là encore chacun de ces deux modes d'assignation a ses avantages et ils sont complémentaires l'un de l'autre. L'utilisateur voudrait donc bénéficier de tous ces avantages dans le même système pour avoir à sa disposition plus de critères possibles dans ses recherches.

Nous avons vu ensuite qu'à un autre niveau, non plus pour la gestion d'un ensemble de fichiers mais pour la gestion du contenu de ces fichiers, les mêmes genres de limitations s'appliquaient dû là encore à la présence d'organisations hiérarchiques. Ainsi certains outils privilégient la navigation et d'autres l'interrogation. Chacun de ces outils proposent de plus une navigation dans une taxinomie particulière ou une interrogation selon des critères particuliers et il n'est même pas possible de combiner deux outils de navigation ou deux outils d'interrogation entre eux. De plus, ce nouveau monde apporte avec lui un nouveau besoin avec la notion de *vue* qui permet de *comprendre* plus facilement un document, de mieux *se concentrer* sur une tâche, et de *modifier* de manière plus cohérente un document. Cependant, même si il existe de nombreux outils concernant ces vues, chacun de ces outils offre une forme de vue particulière, sur un type de fichier particulier et il n'est là non plus pas possible de combiner ces vues, ni même très souvent de modifier ces vues.

Là encore, comme pour les systèmes de fichier le problème avec tous ces outils est qu'ils manquent de principes généraux. Il manque un formalisme commun sous-tendant tous ces outils qui rendrait possible d'incorporer

facilement de nouveaux outils, supportant de nouvelles formes de navigation, de requêtes, de vues (autorisant des mises à jour), ce qui permettrait de plus ensuite de combiner ces outils de manière fructueuse.

Nous avons donc proposé ensuite dans le même chapitre un nouveau paradigme pour les systèmes d'information basé sur la logique qui résoud ces problèmes : le *Logic File System* (LFS). Nous avons décrit de manière intuitive sous la forme d'un tutoriel les principes de LFS et la nouvelle sémantique des commandes *shell* sous LFS. Nous avons décrit ensuite de manière plus formelle la sémantique de ces commandes (sémantique développée complètement dans l'Annexe A).

LFS offre une organisation expressive, une recherche combinant interrogation et navigation, et une facilité de manipulation, à la fois des fichiers et de leurs contenus ; le tout d'une façon intégrée et mis en œuvre au niveau système de fichier. Pour atteindre cette intégration, ce paradigme associe à des *objets* o des *propriétés* logiques, formant une description $d(o)$, et la *déduction logique* $\models$ sert de base pour naviguer et interroger. En effet, en plus de voir $\models$ comme une relation de déduction avec la question $d \models q?$, ce qui représente formellement le processus d'interrogation, nous la voyons aussi comme une relation d'ordre avec la question $f \models g?$, ce qui permet d'ordonner les propriétés et donc de modéliser les notions de générale et spécifique, caractéristiques clef de la navigation.

L'état d'une organisation sous LFS appelé *contexte* est donc formé d'un triplet :

$\mathit{contexte} = (\mathcal{P}, \mathcal{O}, \mathcal{L})$.

Ce triplet est composé d'un ensemble de propriétés $\mathcal{P}$, d'un ensemble d'objets $\mathcal{O}$ avec des descriptions contenant des propriétés de $\mathcal{P}$, et enfin d'une logique $\mathcal{L}$ gouvernant l'ordre entre les propriétés de $\mathcal{P}$.

Bien que LFS soit un système de fichier, nous parlons d'objets et non de fichiers car les principes de LFS s'appliquent aussi au contenu des fichiers. Un objet o peut donc désigner un fichier ou une partie de fichier. La commande `cd parts` permet à un utilisateur d'exprimer son intention de passer de la manipulation de fichier à la manipulation de parties de fichier.

Sous LFS *Les chemins sont des formules*. Les répertoires représentent des requêtes logiques et déterminent des ensembles d'objets (fichiers ou parties de fichiers) dont les propriétés satisfont la requête. La notion importante associée est celle d'*extension* d'une formule f :

$\mathit{ext}(f) = \{o \in \mathcal{O} \mid d(o) \models f\}$.

Cette formule f représente en général un répertoire pwd. Le répertoire racine représente la formule *vrai*, et les sous-répertoires d'un répertoire sont déterminés par les propriétés les plus générales permettant de raffiner la requête : les *incréments*. En effet, le principe de la navigation est de progresser, d'où la condition de raffinement, et de progresser doucement, d'où la recherche des propriétés les plus générales. Cela permet de combiner interrogation et navigation. Plus formellement :

$\mathit{Dirs}(pwd) = \mathit{max}_{\models}\{p \in \mathcal{P} \mid \emptyset \subset \mathit{ext}(p \wedge pwd) \subset \mathit{ext}(pwd)\}$

$\mathit{Files}(pwd) = \mathit{ext}(pwd) - \bigcup_{p \in \mathit{Dirs}(pwd)} \mathit{ext}(p)$

Le formalisme commun mentionné précédemment est donc la *logique* et les principes généraux sont la *déduction logique* et les opérations ensemblistes avec la notion de *raffinement*.

Suite à la commande `cd parts`, le contenu des fichiers est déterminé par les parties du fichier original qui satisfont la requête. Cela permet des accès simultanés en lecture et écriture sur différentes *vues* d'un même fichier, afin d'aider l'utilisateur à séparer et discerner les différents *aspects* de l'information qui sont contenus dans ce fichier. Ces vues contiennent des *marques d'absence* qui rendent possible la mise à jour de ces vues (ainsi que des *points de synchronisation* en interne qui rendent efficace cette mise à jour).

Les propriétés peuvent être attachées aux fichiers manuellement par l'utilisateur, formant la partie *extrinsèque* de la description d d'un fichier, et automatiquement par des programmes appelés *transducteurs*, formant la partie *intrinsèque* de d.

Ces propriétés peuvent être ordonnées afin d'améliorer encore un peu plus la navigation en permettant à l'utilisateur de passer de propriétés générales vers des propriétés plus spécifiques Ces propriétés peuvent être ordonnées manuellement par l'utilisateur pour former des taxinomies par le biais d'*axiomes*, ou automatiquement par des moteurs de déduction logique par le biais de *règles d'inférence*. Les utilisateurs peuvent étendre dynamiquement le système en fournissant leurs propres moteurs de déduction logique et transducteurs par le biais d'un système de *plug-ins*.

Nous avons décrit ensuite au Chapitre 3 les algorithmes et structures de données qui rendent possible cette vision. La nouvelle sémantique des commandes *shell* implique des calculs qui peuvent paraître très coûteux du fait de l'utilisation à outrance de la logique $\models$ et de moteurs de déduction logique, de l'utilisation de transducteurs, et du fait que ces calculs peuvent impliquer un grand nombre d'objets. Comme solution à ces problèmes nous avons proposé principalement 3 structures de données : le *cache logique*, la *table des extensions* (avec l'*inlining* et une représentation des ensembles sous forme d'*intervalles*) et la table des *points de synchronisation*. La *spécialisation* de la *logique du noyau* à une logique des propositions et le cache logique règlent le problème du coût logique, les points de synchronisation le coût des transducteurs et de l'indexation associée, et enfin la table des extensions et la représentation en intervalles le coût de la manipulation d'un grand nombre d'objets. Nous avons décrit aussi les algorithmes exploitant à bon escient ces structures comme insert_property, ls, et reindex.

Nous avons proposé au Chapitre 4 un modèle de sécurité pour LFS basé sur la logique, et qui s'intègre donc naturellement avec «l'esprit logique» de LFS. Ce modèle recentre la sécurité sur les fichiers puisque contrairement aux systèmes de fichier hiérarchiques ce n'est plus désormais les répertoires qui contiennent des fichiers mais les fichiers qui possèdent des propriétés. Ce modèle permet de plus d'exploitation de la négation, permet à l'utilisateur de naviguer parmi les propriétés de sécurité, et encourage beaucoup plus qu'auparavant le partage et le travail coopératif (tout en s'assurant que ce partage sera tout de même sécurisé).

Nous avons présenté au Chapitre 5 de nombreuses extensions. Une extension importante est la possibilité d'établir des liens entre les fichiers ce qui permet d'augmenter de manière importante l'expressivité du modèle LFS.

Nous avons décrit au Chapitre 6 l'implémentation concrète du prototype LFS actuel. Nous avons aussi présenté des expérimentations concrètes utilisant ce prototype démontrant la viabilité de l'approche LFS et du prototype actuel pour la gestion de l'information, à la fois en terme d'efficacité et d'utilité. En effet, nous avons utilisé LFS pour gérer une grande variété et un grand nombre de données (fichiers musicaux, e-mails, programmes, etc) et nous avons présenté de nombreux benchmarks confirmant que LFS n'était pas une vision utopique d'un système d'information puisque les temps de réponse des différentes opérations LFS critiques (notamment la lecture d'un répertoire avec `ls`, la mention de propriétés-formules avec `cd`, et la sauvegarde d'une vue sous un éditeur) sont de l'ordre de la seconde.

Contributions

> *Most papers in computer science describe how their author learned what someone else already knew.*
>
> *Peter Landin*

Nous avons jusque là dans cette thèse parlé de et comparé à LFS des outils que l'on peut qualifier de «première génération» comme les systèmes de fichier classiques (hiérarchiques), les outils comme `find` et `grep`, ainsi que des outils de «deuxième génération» avec les systèmes de fichier avancés «concurrents» déjà cités comme SFS [GJSJ91], CATFS [Gia93], Nebula [BDB+94], ou encore HAC [GM99], les bases de données, et des outils avancés comme les explorateurs de classes à la Smalltalk [Gol84], les environnements de développement intégré récents comme Eclipse [com00], ou les éditeurs avancés comme Emacs [Sta81, Sta02].

Il existe cependant bien d'autres travaux, définissant des outils que l'on peut qualifier de «troisième génération», dont nous n'avons pas encore parlé, et donc auxquels nous ne nous sommes pas encore comparé.

Ainsi, beaucoup d'idées ou constats développés dans cette thèse ne sont pas nouveaux :

- Les problèmes d'organisation des fichiers et les limitations des systèmes de fichiers hiérarchiques sont bien connus [Kis95, Nie96, SG03].
- Les problèmes d'organisation du contenu des fichiers sont aussi connues, surtout pour les sources de programmes, avec l'identification de la *tyrannie de la décomposition dominante* [TOHS99], et de la présence de *structures transversales*, mis (ou

remis) en évidence par l'explosion de la *programmation par aspect* [KLM+97] (AOP pour *Aspect Oriented Programming*). On parle aussi parfois au lieu de structures transversales (*crosscutting concern*) ou d'aspects de *facettes*, de *vues*, ou du problème de *séparation des préoccupations* (*separation of concern* [Par72]), ou encore de *multi-dimensionnalité*.

– L'idée d'utiliser la logique pour la recherche d'information, d'utiliser la logique comme langage à la fois de requête et de description, et donc de voir $\models$ comme un moteur de recherche n'est pas nouvelle. Nous n'avons pas inventé le point de vue logique sur l'interrogation ; que la réponse à une requête q soit l'ensemble des objets satisfaisant q, $\{o \mid d(o) \models q\}$ est une idée bien connue [vR86] (mais le point de vue logique sur la navigation, avec $f \models g$, n'a à notre connaissance été mentionné que par Ferré et Ridoux [FR04]).
– Le besoin de mélanger interrogation et navigation a déjà été reconnu dans [BDMS94, Chi97].
– L'idée de combiner l'interrogation et la navigation par le biais d'*incréments* a déjà été exploré. Dans la littérature, et notamment dans le domaine de la recherche d'information, les incréments sont aussi appelés *liste de co-occurences*, *termes suggérés*, *informations pertinentes*, *mots-clefs significatifs*, *raffinements de requêtes*, *expansions de requêtes*, ou encore *compléments de requêtes*. Ainsi, de nombreux systèmes comme Scatter/Gather [CPKT92, HKP95] (qui permet, pour la recherche d'articles de journaux en ligne, suite à une requête de l'utilisateur de regrouper en différentes catégories par une technique de *clustering* des articles «proches» satisfaisant aussi la requête, et de proposer ces catégories à l'utilisateur comme complément de requête[15]), Flamenco [YSLH03] (qui permet, pour la recherche d'images, à l'utilisateur de naviguer de manière conjointe à travers différentes taxinomies en même temps), StarZoom [BEH99], Cat-A-Cone [HK97], Braque [Ped93], et d'autres [SJCC96, MTW95] permettent de naviguer après une interrogation.
– L'idée de proposer des services avancés d'interrogation et navigation, voir même de les combiner, pour la recherche sur le contenu des fichiers a déjà été proposé avec les systèmes CIA [CNR90] (qui place les informations d'un programme dans une base de données pour que l'utilisateur puisse formuler des requêtes avancées), JQuery [MV04] (qui ajoute l'hypertexte à CIA en l'intégrant dans un IDE pour que du résultat de la requête, l'utilisateur puisse aller directement à l'endroit concerné dans le programme, et qui remplace SQL par Prolog comme langage d'interrogation), ou encore SuperBook [RGL87] (qui permet à l'utilisateur de naviguer sur un document à l'aide d'une table des matières hypertexte, de formuler des recherches de mots dans le document, et surtout suite à une telle recherche de faire clignoter dans la table des matières les sections du document contenant effectivement un mot de la requête).
– L'idée de proposer de multiples vues d'un même document ou d'un même projet pour pouvoir mieux travailler dessus est une vieille idée. Emacs permet d'ouvrir différents *buffers* en même temps sur le même fichier[16] et permet de cacher certaines parties du document avec des marques d'absence[17] de manière différente dans chacun de ces *buffers*. Emacs offre aussi des fonctionnalités avancés de recherche. Emacs s'inspire de nombreux travaux comme ceux des pionniers des environnements de développement et de travail comme NLS [EE68] (système révolutionnaire qui inventa et implémenta dés 1968 l'hypertexte, l'e-mail, la vidéo-conférence, la souris, le traitement de texte, les systèmes de fenêtrage et en ce qui nous concerne la possibilité d'avoir des vues hiérarchiques sur un document, c'est à dire de pouvoir plier et déplier des parties du document, appelées les *outlines* sous Emacs), Interlisp [TM81] (dont l'un des programmes appelé MasterScope fut le premier à proposer l'idée de placer les informations concernant un programme dans une base de donnée pour per-

[15] Un travail accessible similaire est disponible à travers Google à partir de `http://labs.google.com/sets`, et à travers des moteurs de recherche de dernières génération comme Clusty à partir de `http://www.clusty.com`.

[16] Fonctionnalité peu connue accessible avec la fonction `clone-indirect-buffer` et à ne pas confondre avec la possibilité d'avoir différentes *windows* sur le même fichier, qui est moins puissante.

[17] Fonctionnalité peu connue appelée mode *outline*.

mettre à l'utilisateur des requêtes comme «qui appelle la fonction `foo`», et dont les *tags* sous Emacs sont une version très simplifiée) Smalltalk [Gol84], Unix [KM81, Ray04] (avec l'idée de pouvoir combiner des petits outils) ou encore Cedar [Tei84], ainsi que des travaux autour des éditeurs structurés comme Mentor [DGHK+75], Cornell Program Synthesizer [TR81] ou Gandalf [HN86].

– Emacs ainsi que la plupart des outils dont il s'inspire ne comprennent cependant que la structure hiérarchique du document. De nombreux travaux comprennent aussi la structure transversale d'un document et proposent donc aussi l'accès à des vues transversales : MasterScope [TM81] (qui permet d'avoir une vue où par exemple toutes les parties du programme concernant une fonction `foo`, ou utilisant une variable `bar`, sont réunies et peuvent être éditées), Omega [Lin84], TuringTool [CER90] (permet d'appliquer des *masques* sur un fichier pour sélectionner uniquement certaines parties et qui permet de combiner par disjonction, conjonction ou négation ces masques pour former d'autres vues plus complexes), les techniques de *slicing* [Tip95], Live Text [FK90] (permet d'éditer le résultat de certains programmes comme `grep` rendant ainsi son résultat de «première classe»), HyperSpace [TOHS99] (plus récent mais en fait moins avancé qu'un outil comme TuringTool puisqu'il est spécialisé sur la possibilité d'avoir une vue «orientée objet», où les classes dominent, et une vue «orientée fonctionnelle», où les opérations dominent, et dont de nombreux successeurs comme Perspective [BJ00] ou Decal [JV04], tous des *plug-ins* Eclipse, sont eux aussi plus limités que TuringTool, même si ils proposent tous aussi la notion de *virtual source file* comme sous LFS), ainsi que certains modes non standards d'Emacs (comme le *all-mode* [Abr94]). La plupart de ces travaux autorisent la modification de ces vues.

– Certains travaux [Lin95, CS00] autour de l'analyse de concepts (AC) [GW99] décrivent des formules proches de *ext* et *Dirs*. Ainsi, dans [Lin95] un corpus de mots-clefs est extrait de pages `man` et des services proches de ceux offerts par LFS dans la Section 6.2.2 page 149 sont décrits. De même, [CS00] décrit pour la gestion d'e-mails des services proches de ceux offerts par LFS dans la Section 6.2.3 page 150. Dans ces travaux, l'organisation sous-jacente est identique au modèle booléen décrit Section 2.1.1 page 60. Chaque objet possède comme description d une liste de mots-clefs. Une requête q est formée d'une liste de mots-clefs. Ces travaux définissent aussi la notion d'extension avec $\mathit{ext}(q) = \{o \in \mathcal{O} \mid d(o) \supseteq q\}$ et d'incréments (appelés mots-clefs significatifs) avec $\mathit{Incr}(q) = \{p \in \mathcal{P} \mid \emptyset \subset \mathit{ext}(\{p\} \cup q) \subset \mathit{ext}(q)\}$.

Cependant, même si chacun de ces travaux apporte une «brique» intéressante, aucun de ces systèmes n'a toutes ces briques en même temps. L'originalité de LFS est d'avoir su rassembler toutes ces briques, et d'avoir formé un tout cohérent qui n'est pas juste un empilement de fonctionnalités[18]. Le Tableau page 183 dresse une comparaison entre LFS et les travaux que nous estimons les plus importants.

De plus, les fonctionnalités LFS sont comme des briques orthogonales qui bénéficient les unes aux autres. Ainsi, *l'ensemble est plus que simplement la somme de ses parties*. Ainsi, l'amélioration de la «brique interrogation» par l'ajout d'un moteur de déduction améliore aussi le pouvoir d'organisation puisqu'une propriété-formule est une propriété comme une autre pour LFS et que l'on peut assigner aussi bien manuellement que automatiquement une propriété à un fichier (un utilisateur peut ainsi assigner la propriété `year:1985..1990` à une image de vacances si il ne se rappelle plus exactement la date précise). Cette amélioration améliore aussi la navigation puisque les propriétés-formules permettront de classer les autres propriétés pour aller encore plus du général vers le spécifique. Enfin, cette amélioration permettra à l'utilisateur de spécifier des vues encore plus élaborées, améliorant alors la «brique manipulation». De même, l'amélioration de la «brique organisation» par l'ajout d'un transducteur (ou par l'ajout manuel de propriétés par l'utilisateur) améliore aussi l'interrogation, puisque d'autres critères de recherche peuvent être utilisés, et la navigation, puisque LFS proposera de nouveaux incréments.

Les principes de LFS et ses algorithmes ont été présenté à USENIX dans [PR03a] et [PR05].

[18] En fait, nous avons eu connaissance de la plupart des travaux précédents bien après avoir conçu LFS. Il n'a ainsi jamais été question de réunir les fonctionnalités offerts par ces travaux.

LFS et LIS

The best theory is inspired by practice. The best practice is inspired by theory.

Donald Knuth

Comme dit dans l'introduction de cette thèse, LFS se base sur les idées développées par Ferré et Ridoux pour les *systèmes d'information logique* dans [Fer99, Fer02, FR00, FR04] (LIS pour *Logical Information Systems*)[19]. La contribution du DEA de Sébastien Ferré [Fer99] est d'avoir défini ce qu'est un LIS, ce qui généralise les idées développées pour l'analyse de concept [Lin95]. LIS remplace pour la description d d'un objet une liste de mots-clefs par une formule f, pour *ext* l'inclusion $\supseteq$ par la déduction $\models$, et rajoute $max_{\models}$ devant *Incr*. C'est une importante contribution. La beauté du LIS réside justement dans cette apparente simplicité. Les choses les plus simples sont souvent les plus complexes à trouver. Ainsi, la notion d'*objet*, de *formule*, de *contexte*, et les définitions de *ext* et *Dirs* sont en fait des contributions du LIS. Les fondements du LIS, et donc de LFS, sont les notions de *logique*, d'*ensemble* et de *raffinement*, et les opérations de déduction logique et de manipulation d'ensembles qui vont avec. La thèse de Sébastien Ferré [Fer02] et ses autres travaux étendent ensuite dans de nombreuses directions le modèle théorique du LIS.

Notre contribution est donc surtout d'ordre pratique. Nous avons rendu ces objets et formules plus concrets en les adaptant au monde des systèmes de fichier pour former LFS. Ainsi, nous avons fait de ces objets des fichiers et des parties de fichiers. Implémenter et utiliser un vrai système de fichier a fait apparaître de nouveaux problèmes et de nouveaux besoins, et donc de nouvelles fonctionnalités. Pour les fichiers, cela nous a amené à réintroduire la notion de *transducteur* (comme sous SFS [GJSJ91]) afin de gérer l'hétérogénéité des fichiers. L'idée d'appliquer et d'adapter les principes du LIS pour gérer le contenu des fichiers (ce qui en soit est déjà une contribution) nous a amené à définir la notion de *partie de fichier*, de *vue*, de *transducteur avancé* et de *marque d'absence* (comme sous Emacs [Sta02] et d'autres *folding editors*). La logique est elle aussi plus concrète sous LFS. Ainsi, LFS est fait d'un *noyau* composé d'une logique des propositions. Cette logique peut ensuite être étendue par l'utilisateur. Le besoin de taxinomie a été réglé par l'introduction d'*axiomes*, et le besoin de gérer des domaines particuliers, pour pouvoir formuler des requêtes avancées, par l'introduction de la notion de *moteur de déduction logique*, que l'on attache à un attribut. Cet attachement se fait par un système de *plug-ins*, comme pour les transducteurs.

Parmi les autres contributions de LFS sur LIS il y a la gestion de la sécurité (question qui n'a pas ou peu de sens dans le modèle théorique du LIS), et la plupart des extensions dont la possibilité d'établir des *relations* entre les objets qui était citée dans la thèse de Sébastien Ferré [Fer02] comme l'une des plus fortes limitations du LIS.

L'autre contribution de LFS sur LIS, sans doute la plus importante, est d'avoir fait de LIS une réalité *efficace*.

Le DEA de Sébastien Ferré [Fer99] décrit rapidement l'implémentation d'un *shell conceptuel*[20] qui fut utilisé pour expérimenter des idées autour du LIS (et qui fut étendu dans sa thèse [Fer02]). L'idée du cache logique, ainsi que ses algorithmes associés comme insert_property, les premières versions de l'algorithme ls, dont l'idée de partir des propriétés et non des objets, quoique présentés de façon très différente, étaient déjà présents dans [Fer99].

Cependant, ce prototype gérait avec difficulté que quelques centaines d'objets «abstraits» et explosait en complexité. Notre contribution est d'avoir ajouté de nombreuses optimisations comme la spécialisation de la logique du noyau, la spécialisation des algorithmes de gestion du cache logique lorsque la propriété à insérer est une propriété-valeur (voir Section 3.1.1 page 93), l'indexation «agressive» des propriétés avec l'*inlining*, ou encore la compression en intervalles, qui ont améliorés de manière très importante les performances en augmentant de 2 ordres de grandeur le nombre d'objets gérables

[19] Et avant cela sur les travaux effectués par les étudiants de maîtrise d'Olivier Ridoux sur les *systèmes de fichier relationnel* (RFS), qui malgré le nom n'avaient rien de relationnel, mais qui offraient une forme d'incrément et qui offraient des services proches de ceux offerts par LFS dans la Section 6.2.2 page 149.

[20] En fait ce *shell* invente des nouveaux jeux de commande et n'est donc pas un *shell* standard. Il fut décrit aussi dans [FR00], *a filesystem based on concept analysis*, qui malgré le nom n'est en fait pas un système de fichier, mais qui posa tout de même les premières idées pour développer un LFS.

(de 1 000 à 100 000)[21].

Certaines optimisations comme l'*inlining* sont liées au contexte de LFS, le contexte des systèmes de fichier, puisque contrairement au prototype du *shell conceptuel* toutes les données et méta-données ne peuvent pas (ou l'on ne veut pas) tenir en même temps en mémoire ; on ne peut donc pas se permettre des parcours récursifs et des lectures de nombreux secteurs du disque. D'autres optimisations comme l'algorithme `reindex` avec les points de synchronisation sont liées au fait que LFS contrairement au LIS gère des vrais objets, ici des parties de fichier.

Certains choix nous ont amené à perdre une certaine généricité vis-à-vis du modèle théorique du LIS. Par exemple, le fait de restreindre la description d'un objet à une conjonction (là ou LIS autorise n'importe quelle formule) et le fait d'avoir une logique noyau, fait que l'on ne peut pas matérialiser des répertoires représentant une disjonction ou négation. Même si la commande `cd a|b` est autorisé, la commande `mkdir a|b` ne l'est pas sous LFS. Ces choix rendent cependant possible de nombreuses optimisations et des algorithmes spécialisés. Il faut noter aussi que LFS reste relativement générique, puisque même si une logique est plus ou moins *hard-codé* dans le noyau, LFS permet à l'utilisateur d'étendre en certaines points (via les attributs) cette logique avec les moteurs de déduction externes (ce qui permet alors d'associer des propriétés-formules au fichier).

La thèse de Sébastien Ferré [Fer02] présente de nombreuses extensions à [Fer99], et présente des fonctionnalités dont certaines rentrent en «concurrence» avec des fonctionnalités présentées dans cette thèse. Certains choix faits dans [Fer02] sont plus élégants d'un point de vue théorique que ceux faits dans cette thèse. Par exemple, la gestion de la négation est traitée avec une logique plus évoluée, la logique *All I Know*, la combinaison de logiques est traitée de manière plus complète que notre système de *plug-ins* à l'aide de *foncteurs logiques*, enfin nos axiomes sont remplacés par des *logiques à théorie*.

Cependant, leurs interfaces sont souvent plus complexes, et le gain apporté par ces solutions théoriquement plus évoluées n'est pas si évident dans la pratique. Par exemple, les foncteurs logiques sont statiques, ce qui rend impossible pour l'utilisateur d'étendre dynamiquement le système (ce que permet nos *plug-ins*). La possibilité d'avoir dans la description d'un fichier une disjonction n'est pas très intéressante (et peut même aboutir à des surprises désagréable comme décrit Section 2.4 page 79). Parfois, des solutions plus simples et théoriquement moins élégantes sont meilleures ; parfois *worse is better* [Gab91].

Ainsi, d'un point de vue théorique LFS n'est pas plus riche que LIS[22], mais il l'est d'un point de vue pratique. Pour résumer, LFS est un peu au LIS ce que O'Caml est au lambda-calcul.

Perspectives

Vers l'infini et au delà.

Buzz l'éclair

Extension du point de vue

Dans notre introduction, nous posions comme thèse que le traitement des informations aux niveaux Web, site, fichier et ligne partageait les mêmes problèmes, chaque niveau étant considéré comme une collection d'éléments du niveau inférieur. Notre propos a été de montrer comment traiter uniformément un ensemble d'objets, qu'ils soient des fichiers ou des parties de fichier. Ce faisant nous répondons à la préoccupation initiale en ce qui concerne le site et le fichier (considéré comme un ensemble de lignes). Cela ouvre deux perspectives : étendre le procédé vers le haut en direction du Web, et vers le bas en direction de structures plus fines que la ligne.

Côté Web, les challenges sont le volume, la distribution des données et l'absence de structure. Il faudra donc gagner en performance et la solution qui s'impose est la distribution des calculs et des indexes de LFS. Concernant, l'absence de structure, la solution est à rechercher du côté des transducteurs ; leur capacité à découvrir des traits intrinsèques doit être augmentée, par

[21] En fait, LFS peut peut être gérer 1 000 000 d'objets (ou plus) car nous n'avons pas eu l'occasion de mener des expériences avec autant de fichiers car nous n'avons pas autant de fichiers sur notre compte.

[22] LFS est en fait plus riche que LIS sur certains points comme les relations, mais est plus pauvre sur d'autres points.

exemple en utilisant des méthodes du *Semantic Web*. Ce sujet de recherche rejoint des préoccupations très vives actuellement qui concernent la combinaison de la navigation et de l'interrogation sur le Web. Malheureusement, beaucoup de ces solutions manquent de généralité, par exemple en ne traitant pas les descriptions intrinsèques et les descriptions extrinsèques sur le même plan.

Dans une variante plus simple, il s'agirait de distribuer LFS sur un réseau de telle sorte que la réponse à une requête serait la combinaison, à définir, des réponses à des sous-requêtes, elles-mêmes à définir. On devra considérer que les différents sites décrivent des ensembles d'objets différents selon des logiques différentes. De la même façon que NFS rend transparent l'accès à des données hiérarchiques distantes en ne formant plus qu'une hiérarchie, nous souhaitons rendre transparent l'accès à plusieurs contextes LFS. Notons que là où NFS ne permet pas de faire `cd bin` pour atteindre les répertoires `bin` de toutes les machines distantes, LFS distribué le permettra.

Côté structures plus fines que la ligne, on veut considérer comme objet des parties de fichiers qui ne sont pas nécessairement des lignes entières. Par exemple, lorsqu'on manipule des programmes, on peut vouloir masquer les commentaires, même si ils ne font qu'une portion de ligne. Le volume est alors un challenge car si l'on reconnaît comme granularité d'information des éléments plus fins qu'une ligne (ex. le mot), il y aura beaucoup plus d'objets. On peut vouloir aller plus loin et traiter des parties de fichiers de formats quelconques, même binaires. Par exemple, un fichier vidéo pourrait être segmenté et l'utilisateur pourrait naviguer dans un film en exprimant de simples requêtes comme `cd scene:kung-fu|amour/`; `ls character:/`. Cette requête permettrait d'afficher seulement les noms de personnages apparaissant dans des scènes de Kung-Fu ou d'amour. Un des problèmes est de s'assurer que les vues extraites par LFS d'un fichier d'un certain format sont encore de ce format.

Preuves

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

Rich Cook

La spécification de LFS est relativement petite puisque l'essence de LFS tient en une page (voir Section A.1 page 186) avec les définitions des notions fondamentales LFS dont les formules décrivant *Dirs* et *Files*. Les algorithmes ls, insert_property, et reindex, même si ils sont aussi relativement petits, possèdent de nombreuses subtilités. Il n'est pas si évident de dire si ils ont bien la même sémantique que leurs spécifications (surtout que de nombreuses optimisations ont compliqué encore un peu plus ces algorithmes, comme l'intégration de Glimpse dans la Section 3.1.2 page 99, la gestion particulière des propriétés-valeurs dans la Section 3.1.1 page 93, ou le `diff` sur les vues dans la Section 3.1.3 page 104). La spécification de LFS étant exécutable, nous avons pu confronter les résultats retournés par la spécification et l'implémentation sur des jeux de données de petites tailles. Cela a permis de découvrir certains bugs et de mettre en évidence des subtilités.

Cependant, *le test ne peut montrer que la présence de bugs, jamais son absence*[23]. Nous aimerions ainsi des *preuves de correction* des différents algorithmes LFS clefs pour avoir une plus grande confiance dans le code LFS. En fait, nous aimerions même aller plus loin en ayant des *preuves exécutables* à l'aide par exemple d'outils comme ACL2 [KMM00][24]. Il faut noter qu'une partie importante de ce genre de travail a déjà été réalisée puisque la partie spécification, qui a déjà été écrite, représente souvent une étape importante lorsqu'on utilise ce genre d'outils.

De plus, les preuves sont des objets digitaux complexes et donc des objets d'étude intéressants pour LFS (qui nous donnerons peut être des idées de nouvelles fonctionnalités). Les preuves sont proches des programmes : les lemmes sont des sortes de fonctions, les arbres de preuves des sortes de trace d'exécution, et le problème de la recherche de théorème (qui est important dû à la profusion de théorèmes) est proche de la re-

[23] Dijkstra.

[24] ACL2 est le successeur de NQTHM, aussi appelé *Boyer-Moore theorem prover*.

cherche de fonctions dans une librairie.

Enfin, ACL2 étant un démonstrateur de théorème, c'est un moteur de déduction logique (très avancé). On pourrait donc l'utiliser comme un *plug-in* LFS pour formuler des requêtes avancées : par exemple la recherche de fonction commutatives (un travail proche est celui d'AlgoVista [CP00]).

Ergonomie et interface graphique

It said "Insert disk 3..." but only 2 fit.

Anonymous

Nous avons partout dans ce document présenté l'utilisation de LFS via un *shell*. C'est le moyen de s'approcher le plus des fonctions du système de fichiers sans en voir les détails les plus technique. Cependant, rien n'empêche d'utiliser des interfaces élaborées, graphiques comme un explorateur de fichier. D'ailleurs, parce que LFS met en œuvre l'interface standard des systèmes de fichiers les interfaces graphiques actuelles marchent déjà avec LFS.

Mais ces interfaces qui furent conçues pour les systèmes hiérarchiques ne «comprennent pas» LFS. Il reste donc à définir des interfaces qui exploitent complètement les possibilités offertes par LFS.

Applications

En tant que système d'information, LFS a la même cible que les autres systèmes d'information. Cependant, nous pensons que certains types de données sont particulièrement propices à une exploitation originale par LFS. Par exemple, les flux, qu'il s'agisse de log, de trace, de mail, de journaux comptables, demandent à être traités séquentiellement, mais aussi à être explorés selon des critères très variés afin de leur donner de la structure. On peut ainsi localiser des bogues en explorant des traces d'exécution de programme, et détecter des attaques en filtrant des fichiers de log. Dans toutes ces structures, des éléments d'informations arrivent en vrac au cours du temps, mais interfèrent entre eux à des distances arbitraires. LFS permet de retrouver ces relations.

Assistant personnel numérique

I have always wished that my computer would be as easy to use as my telephone. My wish has come true . . .I no longer know how to use my telephone.

Bjarne Stroustrup

Dans le même ordre d'idée, les assistants personnels (PDA) et autres «prothèses mémoire» sont destinés à recevoir des éléments d'information en vrac (contacts, rendez-vous, photos) et de les restituer sous différents points de vue. Nous pensons que LFS peut rendre de grands services dans ce domaine.

Programmation orientée-vue

Software engineering is programming when you can't.

E. W . Dijkstra

Un autre champs d'application est celui du génie logiciel. Il concerne des objets de natures très variées, programmes, spécifications, données (jeux de test), qui entretiennent des relations complexes. En particulier, ils s'organisent en versions et configurations, on peut les décrire dans des diagrammes UML, et on peut y reconnaître des vues, des facettes et des aspects. Trop souvent, ces organisations sont considérées séparément. Nous pensons qu'il vaut mieux les considérer comme des vues alternatives du système d'information du programmeur. LFS peut rendre ce service en exploitant conjointement des données hétérogènes comme les sources, les spécifications et les conversations des programmeurs, et en en extrayant les vues appropriées à chaque situation : recherche de composants, localisation de bogue, vérification d'exigences, etc.

Disponibilité

Only wimps use tape backup : real men just upload their important stuff on ftp and let the rest of the world mirror it.

Linus Torvalds

Le prototype LFS peut être téléchargé à partir de l'URL suivante :

`http://www.irisa.fr/LIS/LFS`

LFS est un projet *open source*. Les sources de LFS et de ses *plug-ins* peuvent ainsi être aussi téléchargés à partir de cette URL. Nous espérons que LFS pourra être utile à la communauté et qu'il apportera sa petite pierre dans l'édifice GNU [Sta85].

	Mot-clef	Attribut valué	Taxinomie	Formule	Entité-Relation	Multiple propriétés	Manuelle	Automatique	Automatique extensible	Et	Ou, Non	Entier, chaîne	Formule (extensible)	Interrogation après Navigation	Proposer	Proposer utile	Général, spécifique	Navigation après Interrogation	Fichiers	Enregistrements	Parties de fichiers	Modification	Vue hiérarchique (*outline*)	Vue transversale	Modification vue	Multiples vues	Généricité, extensibilité	Simplicité	Système de fichier
	Formalisme					Assignation				Interrogation					Navigation				Monde				Vues						
Systèmes	Organisation									Recherche									Manipulation								Divers		
FS classique	●		○			○	●	○		○					●	●	○		●			●					○	●	●
Liens	●		●			●	●								●	●	●		●			○						●	●
`find`	●	●								●	●	●							●										
`grep`	●									○	●	○											○					●	
Emacs	●											●		●	●		●				●	●	○	●	●		●	○	
Total	●	●	●			○	●	●		●	●	●		●	●	●	○		●		●	○	●	○	●	●	○	○	○
SFS [GJSJ91]	●	●				●		●	●	●					●			○	●								●	●	●
CATFS [Gia93]	●	●				●	●			●	●	●	○	●	●	●		○	●			●					○	●	●
Nebula [BDB+94]	●	●				●	○	○		●	●	●	●	●	●	●	○		●			○					○	●	●
HAC [GM99]	●	●	●	●		○	●	●	○	●	●	●		●	●	●	○		●			●					○	○	●
Scatter/Gather [CPKT92]	●					●		●		●	●			●	●	●		●	●									●	
Flamenco [YSLH03]	●		●			●	●			●					●	●	●	●	●									●	
AC [Lin95]	●					●		●		●				●	●	●		●	●									●	
SuperBook [RGL87]	●		●			○		●				●			●	●	●	●			●		○					●	
Omega [Lin84]	○	○			●	●		●		●	●	●									●	○		●	○	○			
CIA [CNR90]	○	○			●	●		●		●	●	●									●								
TuringTool [CER90]	●					●		●		●	●										●	●		●	●	●	●	●	
Base de données	○	○			●	●	●			●	●	●								●		●							
LIS [FR04]	●	●	●	●		●	●	●		●	●	●	●	●	●	●	●	●	○		○	○					●	○	
LFS	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	●

TAB. 6.8 – Résumé comparatif

Annexe A

Spécification

You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program.

Alan Perlis

Programs must be written for people to read, and only incidentally for machine to execute.

Abelson et Sussman

Cette section présente la spécification formelle de LFS. Elle décrit la sémantique des commandes *shell* comme `cd` ou `ls` sous LFS.

Cette spécification n'est pas décrite en langage mathématique mais en langage informatique, plus précisément en O'Caml [LDG+04]. Elle est donc exécutable. Cela fournit ainsi un premier moyen pour s'assurer de la correction de l'implémentation en comparant automatiquement les résultats retournés par la spécification et l'implémentation sur des jeux de données de petites tailles.

Le texte de ce code a été pré-traité pour remplacer le nom de certaines fonctions O'Caml par le symbole mathématique lui correspondant comme ∪ pour `union_set`. Cela permet de renforcer la ressemblance entre ce code et les formules qui figurent dans la thèse.

A.1 L'abstraction

```
type property = PROP of string

type object = ⟨
      id : identity;
    content : content;
    description : property set;
  ⟩
  and identity = int
  and content = string

type context = ⟨
     P : property set;
     O : object set;
     L : logic;
   ⟩
and logic = (property → property → bool) (* mean : a ⊨ b ? *)

(*————————————————————————————————————————*)
type formula =
  | SINGLE of property
  | AND of formula × formula
  | OR of formula × formula
  | NOT of formula

let rec (ext : formula → context → object set) = fun f ctx →
  let (⊨) = ctx.L in
  match f with
  | SINGLE q → {o ∈ ctx.O | ∃ d ∈ o.description (d ⊨ q) }
  | AND (f1, f2) → (ext f1 ctx) ∩ (ext f2 ctx)
  | OR (f1, f2) → (ext f1 ctx) ∪ (ext f2 ctx)
  | NOT f → (ctx.O) \ (ext f ctx)

let (max⊨ : logic → property set → property set) = fun (⊨) P →
  {p ∈ P | ¬∃ p2 ∈ P (p2 ≠ p ∧ p ⊨ p2) }

let (dirs : formula → context → property set) = fun f ctx →
  max⊨ (ctx.L) ({p ∈ ctx.P | ∅ ⊂ ext (AND (f, SINGLE p)) ctx ⊂ ext f ctx })

let (objects : formula → context → object set) = fun f ctx →
  (ext f ctx) \ ( ⋃p∈(dirs f ctx) (ext (AND (f, SINGLE p)) ctx))
```

A.2 Le concret

A.2.1 Les vrais objets

```
type file = ⟨
      filename : filename ;
      extrinsic : property set ;
       intrinsic : property set ;
       fcontent : filecontent ;
   ⟩
   and filename = string
   and filecontent = content

 type idfile = identity

type transducer = (filecontent →  property set)
```

```
(*————————————————————————————————————————-*)
type part = ⟨
      fromfile : idfile ;
      pnumber : int ;
      pdescription : property set ;
      pcontent : partcontent ;
   ⟩
   and partcontent = string

type idpart = identity

type adv_transducer = (partcontent list →  (property set) list)
```

A.2.2 Le vrai contexte

```
(* the identities are used to make link between the real world and the *)
(* abstraction world (the context), and to avoid some redundancies in the structure *)
(* (for instance parts structure refer to files) *)
type world = ⟨
    properties : property set;
    axioms : (property, property set) assoc; (* a ⊨ b ∧ c ∧ d *)

    files : (idfile, file) assoc;
    parts : (idpart, part) assoc;
      filesparts : idfile set;

    plugins : (idfile, plugin) assoc;

    pwd_history : (formula × whichmode) stack;
  ⟩
  and whichmode = FILES | PARTS
  and plugin =
        | TRANS of transducer
        | ADVTRANS of adv_transducer
        | LOGIC of logic

let root = PROP "true"

let default_world = ⟨
  properties = { root };
  axioms = ∅;
  files = ∅;
  parts = ∅; filesparts = ∅;
  plugins = ∅;
  pwd_history = push ((SINGLE root), FILES) ∅
⟩

let (is_files_mode : world → bool) = fun w → (w.pwd_history ▷ top ▷ snd) = FILES

let (new_object : unit → identity) = fun () → counter ()
```

A.2.3 La vraie logique

```
(* valued attribute (vattr) are important in LFS, used to provide advanced logic *)
(* here we define some accessor to a vattr *)
let regexp_attr = "^[a-zA-Z_]+ :"

let (is_attr : property → bool) = fun (PROP s) → s ≈ (regexp_attr ^ "$")
let (is_vattr : property → bool) = fun (PROP s) → s ≈ (regexp_attr ^ ".+$")

let (value_vattr : property → property) = fun (PROP s) →
   PROP (regexp_match s (regexp_attr ^ "(.+)$"))

let (attr_vattr : property → property) = fun (PROP s) →
   PROP (regexp_match s ("(" ^ regexp_attr ^ ")"))

let _ = example (is_attr (PROP "toto :"))
let _ = example (is_vattr (PROP "toto :tata"))
let _ = example (value_vattr (PROP "toto :tata") = (PROP "tata"))
let _ = example (attr_vattr (PROP "toto :tata") = (PROP "toto :"))

(*————————————————————————————————————————*)
 let (find_alogic : world → property → logic) = fun w attr →
   let plugins = {(id, file) ∈ w.files | { PROP "logic"; attr } ⊆ file.extrinsic } in
    let _ = assert (is_attr attr) in
   let _ = assert (‖ plugins ‖ ≤ 1) in
   if plugins = ∅
   then (=) (* default logic when no plugins, flat attr *)
   else
     let (id, file) = top_set plugins in
     match (assoc id w.plugins) with
     | LOGIC (⊨) → (⊨)
     | _ → failwith "not a logic plugin"

(*————————————————————————————————————————*)
let (valid_prop : world → property → bool) = fun w p →
   (if is_vattr p then attr_vattr p else p) ∈ w.properties
```

A.3 Du concret à l'abstrait

```
let (context : world → context) = fun w →
  ⟨
    𝒫 = w.properties;

    ℒ =
          (let rec (⊨) = fun p1 p2 →
            match (p1, p2) with
            | (x, PROP "true") → true (* x ⊨ true *)
            | (PROP "true", x) → false (* true ⊨ x only when x = true *)
            | (x, y) when x = y → true (* x ⊨ x *)

            (* attr :x ⊨ attr :y if x ⊨ y with dedicated logic engine *)
            | (ax, ay) when (is_vattr ax ∧ is_vattr ay) ∧ (attr_vattr ax = attr_vattr ay) →
                let (⊨) = find_alogic w (attr_vattr ax) in
                (value_vattr ax) ⊨ (value_vattr ay)

             (* x ⊨ z if there is an axiom x ⊨ ... ∧ y ∧ ... and y ⊨ z *)
            | (x, z) when x ∈ (keys w.axioms) →
                 ∃ y ∈ assoc x w.axioms (y ⊨ z)

            (* attr :x ⊨ attr : is an axiom so attr :x ⊨ y if attr : ⊨ y *)
            | (ax, y) when is_vattr ax →
                (attr_vattr ax) ⊨ y

            | _ → failwith "internal error"
          in (⊨)
          );

    𝒪 =
          if is_files_mode w
          then w.files ▷ map (fun (idf, f) → ⟨
            id = idf;
            content = f.fcontent;
            description = f.intrinsic ∪ f.extrinsic;
          ⟩)
          else w.parts ▷ map (fun (idp, p) → ⟨
            id = idp;
            content = p.pcontent;
            description = p.pdescription;
          ⟩)

  ⟩
```

A.4 Les transducteurs

```
let transducer_system filename =
  (fun content → set [ PROP ("name :" ^ (fileprefix filename));
                       PROP ("ext :" ^ (filesuffix filename));
                       PROP ("size :" ^ (slength content ▷ i_to_s))
                     ] )


let adv_transducer_system =
  (fun contents → index_list contents ▷ map (fun (c, n) → { PROP ("part :" ^ (i_to_s n)) } ))


(*————————————————————————————————————————*)
let (find_trans : world → property → string → plugin set) = fun w prop suffix →
  {(id, file) ∈ w.files | { prop; PROP ("ext :" ^ suffix) } ⊆ file.extrinsic }
    ▷ map (fun (id, file) → assoc id w.plugins)


let (transducer : world → filename → transducer) = fun w filename →
  let transducers =
    { transducer_system filename } ∪
    (find_trans w (PROP "transducer") (filesuffix filename)
       ▷ map (function (TRANS t) → t | _ → failwith "not a transducer plugin"))
  in
  fun content →
    ⋃_{trans∈(transducers)} (trans content) ▷ filter (valid_prop w)


let (adv_transducer : world → filename → adv_transducer) = fun w filename →
  let transducers =
    { adv_transducer_system } ∪
    (find_trans w (PROP "adv_transducer") (filesuffix filename)
      ▷ map (function (ADVTRANS t) → t | _ → failwith "not an advanced transducer plugin"))
  in
  fun parts →
    transducers ▷ fold (fun props trans →
      let newprops = trans parts in
      zip props newprops ▷ map (fun (s1, s2) → (s1 ∪ (s2 ▷ filter (valid_prop w))))
    ) (parts ▷ map (fun part → ∅))


let (index_parts : world → idfile set → (idpart, part) assoc) = fun w idfiles →
  idfiles ▷ fold
```

```
(fun acc id →
  let file = assoc id w.files in
  let parts = lines file.fcontent in (* or tokens *)
  let part_prop = parts ▷ (adv_transducer w file.filename) ▷ zip parts ▷ index_list in
  part_prop ▷ fold (fun acc ((partcontent, props), partnumber) →
    insert_assoc (new_object (),
                  ⟨ fromfile = id ;
                    pnumber = partnumber ;
                    pdescription = props ;
                    pcontent = partcontent ;
                  ⟩) acc
   ) acc
) ∅
```

A.5 Le *shell*

```
type path_element = SLASH | DOTDOT | ELEMENT of formula
type path = path_element list

let (w : world ref) = ref default_world

let (pwd : unit → formula) = fun () → fst (top !w.pwd_history)

(*————————————————————————————————————————————*)
(* many commands mention valued attributes which do ¬ have to be created first, kind *)
(* of sugar, but they must nevertheless as with mkdir be added to the property set *)

let (check_and_add_properties : property set → unit) = fun ps →
  let _ = ps ▷ iter (fun p → assert (valid_prop !w p)) in
  w := ⟨ !w with properties = !w.properties ∪ ps ⟩

let rec (properties_of_formula : formula → property set) = function
  | SINGLE x → { x }
  | (AND (f1, f2) | OR (f1, f2)) → (properties_of_formula f1) ∪ (properties_of_formula f2)
  | NOT f → properties_of_formula f

let rec (is_conjunction : formula → bool) = function
  | SINGLE _ → true
  | AND (f1, f2) → is_conjunction f1 ∧ is_conjunction f2
  | ((OR _)|(NOT _)) → false
```

A.5.1 `cd/ls`

```
let (cd : path_element → unit) = function
   | SLASH → w := ⟨ !w with pwd_history = push ((SINGLE root), FILES) ∅⟩
   | DOTDOT → w := ⟨ !w with pwd_history = pop !w.pwd_history⟩
   | ELEMENT (SINGLE (PROP "parts")) →
       let _ = assert (is_files_mode !w) in
       let files_here = ext (pwd ()) (context !w)
                          ▷ map (fun obj → obj.id) in
       (w := ⟨ !w with
                pwd_history = push ((SINGLE root), PARTS) !w.pwd_history;
                filesparts = files_here;
                parts = index_parts !w files_here;
              ⟩;
        check_and_add_properties (⋃(id,p)∈(!w.parts) (p.pdescription))
       )
   | ELEMENT f →
       (w := ⟨ !w with pwd_history =
                let (oldf, whichmode) = top !w.pwd_history in
                push (AND (f, oldf), whichmode) !w.pwd_history;
              ⟩;
       check_and_add_properties (properties_of_formula f)
       )

let (dopath : path → (unit → 'a) → 'a) = fun path op →
   let old_pwd = !w.pwd_history in
   let _ = path ▷ iter (fun p → cd p) in
   let x = op () in
   (w := ⟨ !w with pwd_history = old_pwd⟩ ; x)

(*————————————————————————————————————————-*)
let (ls : unit → (property set × idfile set)) = fun () →
    let pwd = pwd () in
   let ctx = context !w in
   (dirs pwd ctx,
    if is_files_mode !w
    then objects pwd ctx ▷ map (fun o → o.id)
    else ⋃o∈(ext pwd ctx) ({ (assoc o.id !w.parts).fromfile } )
   )

let ls_filenames() = ls() ▷ snd ▷ map (fun id → (assoc id !w.files).filename)
let ls_id_of_name s = ls() ▷ snd ▷ find (fun id → (assoc id !w.files).filename = s)
```

A.5.2 `mkdir/mkfile`

```
let (mkdir : string → unit) = fun name →
    let pwd = pwd () in
   let ps = properties_of_formula pwd in

   let _ = assert (¬ ((PROP name) ∈ !w.properties)) in
   let _ = assert (¬ (name = "parts")) in
   let _ = assert (is_conjunction pwd) in (* TODO check simple atom ?, check no special sym *)

   w := ⟨ !w with
          properties = { PROP name } ∪ !w.properties;
          axioms = insert_assoc (PROP name, ps) !w.axioms;
        ⟩
```

```
(*————————————————————————————————————————————————-*)
let (mkfile : filename → filecontent → plugin option → unit) = fun name content plugin →
    let pwd = pwd () in
   let ps = properties_of_formula pwd in

   let _ = assert (is_files_mode !w) in (* TODO ? could *)
   let _ = assert (is_conjunction pwd) in (* TODO ? or filter ¬/or *)
   let _ = assert (not (name ∈ ls_filenames() )) in (* TODO need assert name ¬ in P *)

   let o = new_object () in
   let file = ⟨ filename = name;
                fcontent = content;
                extrinsic = ps;
                intrinsic = content ▷ (transducer !w name)
              ⟩ in

   (w := ⟨ !w with files = insert_assoc (o, file) !w.files ⟩;
    (match plugin with
    | SOME x → w := ⟨ !w with plugins = insert_assoc (o, x) !w.plugins ⟩
    | _ → ());
    check_and_add_properties file.intrinsic
   )
```

A.5.3 `rm/mv`

```
let (rm : filename → unit) = fun s →
    let _ = assert (is_files_mode !w) in (* TODO ? could *)
   let idfile = ls_id_of_name s in

   w := ⟨ !w with
         files = del_assoc idfile !w.files ;
         plugins = del_assoc idfile !w.plugins ;
       ⟩
```

```
(*————————————————————————————————————————————*)
(* TODO minp pb : cd a/b/b2/c mv ../../ ca va lui ajouter b, alors que en fait *)
(* mais bon l'utilisateur a qu'a faire gaffe (fleche forte/fleche faible) *)
let (mv : filename → path → filename → unit) = fun oldname newpath newname →
    let _ = assert (is_files_mode !w) in
   let idfile = ls_id_of_name oldname in
   let file = assoc idfile !w.files in

   let oldpwd = pwd() in
   let oldprops = properties_of_formula oldpwd in

   dopath newpath (fun () →
     let newpwd = pwd() in
     let newprops = properties_of_formula newpwd in

     let _ = assert (is_files_mode !w) in
     let _ = assert (¬ (newname ∈ ls_filenames() )) in (* in fact should erase content *)
     let _ = assert (is_conjunction oldpwd) in
     let _ = assert (is_conjunction newpwd) in

     w := ⟨ !w with files = !w.files ▷ replace_assoc
           (idfile, ⟨file with
                    extrinsic = (file.extrinsic \ oldprops) ∪ newprops ;
                    filename = newname ; (* TODO : call transducer_system *)
                   ⟩)⟩
   )
```

A.5.4 `rmdir/mvdir`

```
let (rmdir : string → unit) = fun s →
    let p = PROP s in
   let _ = assert (p ∈ !w.properties) in
   let _ = assert (is_files_mode !w) in (* TODO ? could *)
   (* TODO check have no child or adjust axioms *)

   w := ⟨ !w with
         properties = !w.properties \ { p } ;
         axioms = del_assoc p !w.axioms ;
         files = !w.files ▷ map (fun (id, file) →
           (id, ⟨file with
                  extrinsic = file.extrinsic \ { p } ;
                  intrinsic = file.intrinsic \ { p } ;
                ⟩ ))
       ⟩
(*————————————————————————————————————————————*)
let (mvdir : string → path → string → unit) = fun oldname newpath newname →
    let oldp = PROP oldname in
   let newp = PROP newname in

   let _ = assert (is_files_mode !w) in (* TODO ? could *)
   let _ = assert (oldp ∈ !w.properties) in
   let _ = if newp ≠ oldp then assert (¬ (newp ∈ !w.properties)) in

   let oldpwd = pwd() in
   let oldprops = properties_of_formula oldpwd in

   dopath newpath (fun () →
     let newpwd = pwd() in
     let newprops = properties_of_formula newpwd in

     let _ = assert (is_files_mode !w) in
     let _ = assert (is_conjunction oldpwd) in
     let _ = assert (is_conjunction newpwd) in

     let newaxioms = ((assoc oldp !w.axioms) \ oldprops) ∪ newprops in
     let (⊨) = (context !w).ℒ in
     let _ = assert (¬∃ p ∈ newaxioms (p ⊨ oldp) ) in (* avoid cycle *)

     (* TODO change name *)
     w := ⟨ !w with axioms = !w.axioms ▷ replace_assoc (oldp, newaxioms) ⟩
     )
```

A.5.5 `read/write`

```
let mark_string mark = ("..........  :" ^ (i_to_s mark))

let (view : formula → idfile → (filecontent × ((int, idpart list) assoc))) = fun pwd idfile →
    let parts_pwd = ext pwd (context !w)
                        ▷ map (fun obj → obj.id) in
   let all_parts = {(id, part) ∈ !w.parts | part.fromfile = idfile }
                              ▷ sort (fun (id1, p1) (id2, p2) → p1.pnumber <=> p2.pnumber) in

   let content = ref "" in
   let marks = ref ∅ in
   let mark = ref 1 in
   let pending = ref ∅ in

   all_parts ▷ iter (fun (id, part) →
     if id ∈ parts_pwd
     then
       if !pending = ∅
       then (content := !content ^ part.pcontent)
       else (content := !content ^ (mark_string !mark) ^ "\n" ^ part.pcontent;
             marks ▷ insert_assoc ( !mark, !pending);
             mark := !mark + 1 ;
             pending := ∅ ;)
     else pending := !pending ∪ { id }
    );
   if !pending = ∅
   then ( !content, !marks)
   else ( !content ^ (mark_string !mark), insert_assoc ( !mark, !pending) !marks)

(*————————————————————————————————————————-*)
let (read : filename → filecontent) = fun name →
    let idfile = ls_id_of_name name in
   let file = assoc idfile !w.files in
   if is_files_mode !w
   then file.fcontent
   else view (pwd ()) idfile ▷ fst

(*————————————————————————————————————————-*)
let (write : filename → filecontent → unit) = fun name newcontent →
    let idfile = ls_id_of_name name in
   let file = assoc idfile !w.files in
```

```
let finalcontent =
  if is_files_mode !w then newcontent
  else
    let marks = view (pwd ()) idfile ▷ snd in
    lines newcontent ▷ map (fun s →
      if s ≈ ".*\.\.\.\.  :"
      then let mark = regexp_match s ".*  :([0-9]+)" ▷ s_to_i in
        assoc mark marks ▷ map (fun idpart →  (assoc idpart !w.parts).pcontent) ▷ unwords
      else s
      ) ▷ unwords in

let newfile = ⟨file with
                fcontent = finalcontent;
                intrinsic = finalcontent ▷ (transducer !w file.filename);
              ⟩ in

(w := ⟨ !w with files = !w.files ▷ replace_assoc (idfile, newfile)⟩ ;
 check_and_add_properties newfile.intrinsic;

 if ¬ (is_files_mode !w) then
    (w := ⟨ !w with parts = index_parts !w !w.filesparts; ⟩ ;
     check_and_add_properties (⋃_{(id,p)∈(!w.parts)} (p.pdescription)));
)
```

Annexe B

Implémentation

Everyone can be taught to sculpt : Michelangelo would have had to be taught how not to. So it is with the good programmers.

Alan Perlis

It takes more than a license to make source code open.

Hans Reiser

I am an humble programmer [Dij87]. Cette section présente donc le source du programme LFS. Nous ne présentons cependant que la partie O'Caml du prototype (voir Section 6.1.1 page 142) qui forme la partie la plus intéressante. Cette partie contient les structures de données et les algorithmes principaux. Les autres parties du prototype se chargent de faire le lien avec le monde Unix et le VFS, et de faire le lien avec Berkeley DB [OBS99] afin de rendre les structures de données persistentes.

B.1 Structures de données

```
type property = PROP of string
type object = OBJ of identity
      and identity = int

type logic = (property → property → bool) (* mean : a ⊨ b ? *)

type context = ⟨
     logic_cache : property graph ;
      extensions : (property, object set) assoc ;
   ⟩

type formula =
   | SINGLE of property
   | AND of formula × formula
   | OR of formula × formula
   | NOT of formula

(*———————————————————————————————————-*)
type file = ⟨
     filename : filename ;
     extrinsic : property set ;
     intrinsic : property set ;
     fcontent : filecontent ;
   ⟩
   and filename = string
   and filecontent = string

type idfile = object

type transducer = (filecontent → property set)

(*———————————————————————————————————-*)
type part = ⟨
     fromfile : idfile ;
     pdescription : property set ;
     pcontent : partcontent ;
   ⟩
   and partcontent = string

type idpart = object

type adv_transducer = (partcontent list → (property set) list)
```

```
type parts_info = ⟨
    parts_info : (idpart, part) assoc;
    line_to_part : (int, idpart) assoc;
    lines_synchro : int set;
  ⟩

(*————————————————————————————————————————-*)
type world = ⟨
    graphp : property graph;

    files : (idfile, file) assoc;
    extfiles : (property, idfile set) assoc;

    parts : (idfile, parts_info) assoc;
    extparts : (property, idpart set) assoc;

    plugins : (idfile, plugin) assoc;
    pwd_history : (formula × whichmode) stack;
  ⟩
  and whichmode = FILES | PARTS
  and plugin =
        | TRANS of transducer
        | ADVTRANS of adv_transducer
        | LOGIC of logic

let root = PROP "true"

let default_world = ⟨
  graphp = empty_graph ▷ add_node root;

  files = ∅;
  extfiles = ∅ ▷ insert_assoc (root, ∅);
  parts = ∅;
  extparts = ∅ ▷ insert_assoc (root, ∅);

  plugins = ∅;
  pwd_history = push ((SINGLE root), FILES) ∅
⟩

(*————————————————————————————————————————-*)
let (is_files_mode : world → bool) = fun w → (w.pwd_history ▷ top ▷ snd) = FILES
let (new_object : unit → object) = fun () → OBJ (counter ())
let (context : world → context) = fun w →
  ⟨ logic_cache = w.graphp;
    extensions = if is_files_mode w then w.extfiles else w.extparts;
  ⟩
```

B.2 Algorithmes

B.2.1 ls

```
let rec (ext : formula → context → object set) = fun f ctx →
  match f with
  | SINGLE q → assoc q ctx.extensions
  | AND (f1, f2) → (ext f1 ctx) ∩ (ext f2 ctx)
  | OR (f1, f2) → (ext f1 ctx) ∪ (ext f2 ctx)
  | NOT f → (assoc root ctx.extensions) \ (ext f ctx)

let (dirs : formula → context → property set) = fun f ctx →
   let ois = ext f ctx in
  let graph = ctx.logic_cache in

  let visited = hcreate () in
  let dirs = ref ∅ in

  let rec dfs = fun from ps →
    visited ▷ hadd (from, true) ;
    ps ▷ iter (fun p →
      if ¬ (visited ▷ hmem p) then
        (* we do a try cos when cd parts, some props are no more defined *)
        let newois = (try (assoc p ctx.extensions) with _ → ∅) ∩ ois in

        match (length newois, length ois) with
        | (newl, oldl) when newl = oldl → dfs p (graph ▷ successors p)
        | (newl, oldl) when 0 < newl ∧ newl < oldl →
            (* we want the max, the next parent will do the job (if he has to do the job) *)
            if ((graph ▷ predecessors p) ▷ filter (fun p → ¬ (visited ▷ hmem p))) = ∅
            then (visited ▷ hadd (p, true) ;
                  dirs := !dirs ∪ { p } )
        | _ → visited ▷ hadd (p, true)
    )
  in
  dfs root (graph ▷ successors root) ;
 !dirs

let (objects : formula → context → object set) = fun f ctx →
  (ext f ctx) \ ( ⋃_{p∈(dirs f ctx)} (ext (AND (f, SINGLE p)) ctx))
```

B.2.2 Le cache logique

```
let rec (find_parents : property → logic → property graph → property → property set) = fun f (⊨) graph parent
→
  let children = graph ▷ successors parent in
  let implied = {child ∈ children | f ⊨ child } in
  if implied = ∅
  then { parent }
  else ⋃_{newparent∈(implied)} (find_parents f (⊨) graph newparent)

let rec (find_children : property → logic → property graph → property set → property set) = fun f (⊨)
graph children →
  if children = ∅ then ∅
  else
    let (child, others) = (head children, tail children) in
    if (child ⊨ f)
    then find_children f (⊨) graph others ▷ add_child child (⊨)
    else find_children f (⊨) graph (others ∪ graph ▷ successors child)

and (add_child : property → logic → property set → property set) = fun child (⊨) children →
  if (∃ other ∈ children (child ⊨ other))
  then children
  else { child } ∪ {other ∈ children | ¬ (other ⊨ child) }

(*————————————————————————————————————————-*)
let (insert_property : property → logic → property graph → property → property graph) = fun f (⊨) graph top
→
  let parents = find_parents f (⊨) graph top in
  let candidates = ⋃_{parent∈(parents)} (graph ▷ successors parent) in
  let children = find_children f (⊨) graph candidates in

  if (is_singleton parents) ∧ (head parents ⊨ f) then graph
  else
    let g = ref graph in
    g ▷ add_node f ;
    parents ▷ iter (fun parent →
      g ▷ add_arc (parent, f) ;
      (children ∩ (graph ▷ successors parent)) ▷ iter (fun child →
        g ▷ del_arc (parent, child)) ;
    ) ;
    children ▷ iter (fun child → g ▷ add_arc (f, child)) ;
!g
```

B.2.3 Les vues

```
let mark_string mark = (". . . . . .  :" ^ (i_to_s mark))

(*————————————————————————————————————————*)
 let (view : formula → context → parts_info → (filecontent × ((int, idpart list) assoc))) = fun pwd ctx info →
   let linesp = info.line_to_part in
   let parts_pwd = ext pwd ctx in
   let all_parts = enum 0 (length linesp -1) ▷ map (fun i → assoc i linesp) in

   let content = ref "" in
   let marks = ref ∅ in
   let mark = ref 1 in
   let pending = ref ∅ in

   all_parts ▷ iter (fun id →
     if id ∈ parts_pwd
     then
       let part = assoc id info.parts_info in
       if !pending = ∅
       then (content := !content ^ part.pcontent)
       else (content := !content ^ (mark_string !mark) ^ "\n" ^ part.pcontent;
             marks ▷ insert_assoc ( !mark, !pending);
             mark := !mark + 1 ;
             pending := ∅ ;)
     else pending := !pending ∪ { id }
    ) ;
   if !pending = ∅
   then ( !content, !marks)
   else ( !content ^ (mark_string !mark), insert_assoc ( !mark, !pending) !marks)


(*————————————————————————————————————————*)
 let (update_view : filecontent → parts_info → ((int, idpart list) assoc) → filecontent) = fun newcontent info marks
→
   lines newcontent ▷ map (fun s →
     if s ≈ ".*\.\.\.\.  :"
     then let mark = regexp_match s ".*  :([0-9]+)" ▷ s_to_i in
       assoc mark marks ▷ map (fun id → (assoc id info.parts_info).pcontent) ▷ unwords
     else s
    ) ▷ unwords
```

```
(*————————————————————————————————————————————*)
let (create_parts : idfile → file → adv_transducer → parts_info) = fun id file trans →
   let parts = lines file.fcontent in (* or tokens *)
   let part_prop = parts ▷ trans ▷ zip parts ▷ index_list in

   part_prop ▷ fold (fun info ((s, ps), i) →
     let o = new_object () in
     ⟨ parts_info = insert_assoc (o,
                                  ⟨ fromfile = id ;
                                    pdescription = ps ;
                                    pcontent = s ;
                                  ⟩) info.parts_info ;
       line_to_part = insert_assoc (i, o) info.line_to_part ;

       lines_synchro = (if (PROP "synchro") ∈ ps then { i } else ∅) ∪ info.lines_synchro
     ⟩
    ) ⟨ parts_info = ∅ ; line_to_part = ∅ ; lines_synchro = ∅⟩
```

```
(*————————————————————————————————————————————*)
(* return new part_info + todel + toadd) *)
(* quite complicated *)
 let (reindex_parts : idfile → filecontent → (file × parts_info) → adv_transducer → (parts_info × idpart set ×
idpart set)) = fun id newc (oldf, oldinfo) trans →

   let newlines = lines newc in
   let oldlines = lines oldf.fcontent in

   let (newinfo, newline, newsynchro) = (ref ∅, ref ∅, ref ∅) in

   let to_recall = ref ∅ in
   let to_reindex = ref ∅ in
   let to_del = ref ∅ in

   let pending = ref ∅ in
   let is_in_clean = ref true in


   let _ = (newlines, oldlines) ▷ diff
        (fun ai bi comparaison →
          match comparaison with
```

```
        | MATCH →
            let id = assoc bi oldinfo.line_to_part in
            let p = assoc id oldinfo.parts_info in
            let synchro = bi ∈ oldinfo.lines_synchro in
            if synchro then
              (is_in_clean := true ;
               pending := ∅ ;
               newsynchro ▷ (fun xs → xs ∪ { ai } ) ;
              ) ;
            if !is_in_clean
            then (newinfo ▷ insert_assoc (id, p) ;
                  newline ▷ insert_assoc (ai, id) ;
                  pending ▷ snoc ai ;
                 )
            else (to_del ▷ snoc bi ;
                  to_recall ▷ snoc ai ; to_reindex ▷ snoc ai ;
                 )
        | (ANOTINB | BNOTINA) →
            to_recall ▷ (fun xs → xs @ !pending ) ;
            is_in_clean := false ;
            pending := ∅ ;
            if comparaison = ANOTINB
            then (to_recall ▷ snoc ai ; to_reindex ▷ snoc ai ;)
            else to_del ▷ snoc bi
      ) in

  let parts = !to_recall ▷ map (fun i → LIST.nth newlines i) in
  let part_prop = parts ▷ trans ▷ zip parts ▷ (fun x → zip x !to_recall) in

  let info = part_prop ▷ fold (fun info ((s, ps), i) →
    if i ∈ !to_reindex then
      let o = new_object () in
      ⟨ parts_info = insert_assoc (o,
                                   ⟨ fromfile = id ;
                                     pdescription = ps ;
                                     pcontent = s ;
                                   ⟩) info.parts_info ;
        line_to_part = insert_assoc (i, o) info.line_to_part ;
        lines_synchro = (if (PROP "synchro") ∈ ps then { i } else ∅) ∪ info.lines_synchro
      ⟩
    else info
  ) ⟨ parts_info = !newinfo ; line_to_part = !newline ; lines_synchro = !newsynchro⟩
  in
  (info,
!to_reindex ▷ map (fun i → assoc i info.line_to_part),
```

```
 !to_del ▷ map (fun i → assoc i oldinfo.line_to_part)
  )
```

B.3 *Plug-ins*

B.3.1 Logiques

```
(* valued attribute (vattr) are important in LFS, used to provide advanced logic *)
(* here we define some accessor to a vattr *)
let regexp_attr = "^[a-zA-Z_]+ :"

let (is_attr : property → bool) = fun (PROP s) → s ≈ (regexp_attr ^ "$")
let (is_vattr : property → bool) = fun (PROP s) → s ≈ (regexp_attr ^ ".+$")

let (value_vattr : property → property) = fun (PROP s) →
   PROP (regexp_match s (regexp_attr ^ "(.+)$"))
let (attr_vattr : property → property) = fun (PROP s) →
   PROP (regexp_match s ("(" ^ regexp_attr ^ ")"))

let _ = example (is_attr (PROP "toto :"))
let _ = example (is_vattr (PROP "toto :tata"))
let _ = example (value_vattr (PROP "toto :tata") = (PROP "tata"))
let _ = example (attr_vattr (PROP "toto :tata") = (PROP "toto :"))

let (find_alogic : world → property → logic) = fun w attr →
   let plugins = try (assoc (PROP "logic") w.extfiles) ∩ (assoc attr w.extfiles) with _ → ∅ in
   let _ = assert (is_attr attr) in
   let _ = assert (‖ plugins ‖ ≤ 1) in
   if plugins = ∅
   then (=) (* default logic when no plugins, flat attr *)
   else
     let id = top_set plugins in
     match (assoc id w.plugins) with
      | LOGIC (⊨) → (⊨)
     | _ → failwith "not a logic plugin"

(*————————————————————————————————————————-*)
let (valid_prop : world → property → bool) = fun w p →
```

(if is_vattr p then attr_vattr p else p) ∈ w.graphp ▷ *nodes*

B.3.2 Transducteurs

```
let transducer_system filename =
  (fun content → set [ PROP ("name :" ^ (fileprefix filename));
                       PROP ("ext :" ^ (filesuffix filename));
                       PROP ("size :" ^ (slength content ▷ i_to_s))
                     ])
let adv_transducer_system =
  (fun contents → index_list contents ▷ map (fun (c, n) → { PROP ("part :" ^ (i_to_s n)) }))


(*————————————————————————————————————————*)
let (find_trans : world → property → string → plugin set) = fun w prop suffix →
  if suffix ≠ "" then
    (try assoc (PROP ("ext :" ^ suffix)) w.extfiles ∩ assoc prop w.extfiles with _ → ∅)
      ▷ map (fun id → assoc id w.plugins)
  else ∅


let (transducer : world → filename → transducer) = fun w filename →
  let transducers =
    { transducer_system filename } ∪
    (find_trans w (PROP "transducer") (filesuffix filename)
       ▷ map (function (TRANS t) → t | _ → failwith "not a transducer plugin"))
  in
  fun content →
    ⋃_{trans∈(transducers)} (trans content) ▷ filter (valid_prop w)


let (adv_transducer : world → filename → adv_transducer) = fun w filename →
  let transducers =
    { adv_transducer_system } ∪
    (find_trans w (PROP "adv_transducer") (filesuffix filename)
     ▷ map (function (ADVTRANS t) → t | _ → failwith "not an advanced transducer plugin"))
  in
  fun parts →
    transducers ▷ fold (fun props trans →
      let newprops = trans parts in
      zip props newprops ▷ map (fun (s1, s2) → (s1 ∪ (s2 ▷ filter (valid_prop w))))
```

```
) (parts ▷ map (fun part → ∅))
```

B.4 Le *shell*

```
type path_element = SLASH | DOTDOT | ELEMENT of formula
type path = path_element list

let (w : world ref) = ref default_world

let (pwd : unit → formula) = fun () → fst (top !w.pwd_history)

(*————————————————————————————————————————-*)
(* many commands mention valued attributes which do ¬ have to be created first, kind *)
(* of sugar, but they must nevertheless as with mkdir be added to the property set *)
let (check_and_add_properties : property set → unit) = fun ps →
  let _ = ps ▷ iter (fun p → assert (valid_prop !w p)) in
  let _ = ps ▷ iter (fun p →
    if is_vattr p then
      let attr = attr_vattr p in
      let (⊨) = find_alogic !w attr in
      let (⊨) = (fun p1 p2 →
        match (is_attr p1, is_attr p2) with
        | (true, true) → true (* attr : ⊨ attr : *)
        | (true, false) → false (* attr : ⊨ attr :v is false *)
        | (false, true) → true (* attr :v ⊨ attr : *)
        | (false, false) → (value_vattr p1) ⊨ (value_vattr p2) (* attr :x ⊨ attr :y if x ⊨ y *)
        ) in
      if ¬ (p ∈ ( !w.graphp ▷ nodes)) then
        (w := ⟨ !w with graphp = insert_property p (⊨) !w.graphp attr⟩ ;
          w := ⟨ !w with extfiles =
                insert_assoc (p, !w.graphp ▷ successors ⋃_{p∈(p)} (assoc p !w.extfiles))
 !w.extfiles⟩ ;
        )
  ) in ()

let rec (properties_of_formula : formula → property set) = function
  | SINGLE x → { x }
  | (AND (f1, f2) | OR (f1, f2)) → (properties_of_formula f1) ∪ (properties_of_formula f2)
  | NOT f → properties_of_formula f

let rec (is_conjunction : formula → bool) = function
```

```
  | SINGLE _ → true
  | AND (f1, f2) → is_conjunction f1 ∧ is_conjunction f2
  | ((OR _)|(NOT _)) → false

let (upward_props : property set → property graph → property set) = fun xs graph →
  graph ▷ fold_upward (fun props p → props ∪ { p } ) xs xs
```

B.4.1 `cd/ls`

```
(*-------------------------------------------------------------*)
let (cd : path_element → unit) = function
  | SLASH → w := ⟨ !w with pwd_history = push ((SINGLE root), FILES) ∅⟩
  | DOTDOT → w := ⟨ !w with pwd_history = pop !w.pwd_history⟩
  | ELEMENT (SINGLE (PROP "parts")) →
      let _ = assert (is_files_mode !w) in
      let files_here = ext (pwd ()) (context !w) in
      (w := ⟨ !w with
               pwd_history = push ((SINGLE root), PARTS) !w.pwd_history;
               parts = files_here ▷ map (fun idfile → (* in fact a fold insert_assoc *)
                 let file = (assoc idfile !w.files) in
                 (idfile, create_parts idfile file (adv_transducer !w file.filename))
                 )
           ⟩;
       let all_parts = ⋃(id,info)∈(!w.parts) (info.parts_info) in (* union assoc *)
       (check_and_add_properties (⋃(id,p)∈(all_parts) (p.pdescription));
        let extparts =
          all_parts ▷ fold (fun extparts (o, p) →
            upward_props p.pdescription !w.graphp
              ▷ fold (fun extparts p → (* apply_assoc p (fun x → x ∪ { o } ) extp *)
                match (try SOME (assoc p extparts) with _ → NONE) with
                | NONE → insert_assoc (p, { o } ) extparts
                | SOME x → replace_assoc (p, x ∪ { o } ) extparts
                ) extparts
           ) ∅ in
        w := ⟨ !w with extparts = extparts⟩
        )
      )
  | ELEMENT f →
      (w := ⟨ !w with pwd_history =
               let (oldf, whichmode) = top !w.pwd_history in
               push ((if oldf = SINGLE root then f else AND (f, oldf)), whichmode) !w.pwd_history;
           ⟩;
```

```
        check_and_add_properties (properties_of_formula f)
        )

let (dopath : path → (unit → 'a) → 'a) = fun path op →
   let old_pwd = !w.pwd_history in
   let _ = path ▷ iter (fun p → cd p) in
   let x = op () in
   (w := ⟨ !w with pwd_history = old_pwd⟩ ; x)

(*————————————————————————————————————————*)
let (ls : unit → (property set × idfile set)) = fun () →
   let pwd = pwd () in
   let ctx = context !w in
   (dirs pwd ctx,
     if is_files_mode !w
    then objects pwd ctx
    else
      let all_parts = ⋃_{(id,info)∈(!w.parts)} (info.parts_info) in (* union assoc *)
       ⋃_{o∈(ext pwd ctx)} ({ (assoc o all_parts).fromfile } )
   )

let ls_filenames() = ls() ▷ snd ▷ map (fun id → (assoc id !w.files).filename)
let ls_id_of_name s = ls() ▷ snd ▷ find (fun id → (assoc id !w.files).filename = s)
```

B.4.2 `mkdir/mkfile`

```
let (mkdir : string → unit) = fun name →
   let p = PROP name in
   let pwd = pwd () in
   let _ = assert (¬ (p ∈ ( !w.graphp ▷ nodes))) in
   let _ = assert (¬ (name = "parts")) in
   let _ = assert (is_conjunction pwd) in (* TODO check simple atom ?, check no special sym *)
   let ps = properties_of_formula pwd in
   w := ⟨ !w with
          graphp = !w.graphp
                     ▷ add_node p
                     ▷ (fun g → ps ▷ fold (fun g prop → g ▷ add_arc (prop, p)) g) ;
          extfiles = insert_assoc (p, ∅) !w.extfiles ;
        ⟩

(*————————————————————————————————————————*)
let (mkfile : filename → filecontent → plugin option → unit) = fun name content plugin →
   let pwd = pwd () in
```

```
  let _ = assert (is_files_mode !w) in (* TODO ? could *)
  let _ = assert (is_conjunction pwd) in (* TODO ? or filter ¬/or *)
  let _ = assert (not (name ∈ ls_filenames() )) in (* TODO need assert name ¬ in P *)
  let props = properties_of_formula pwd in
  let intr = content ▷ (transducer !w name) in

  let o = new_object () in
  let file = ⟨ filename = name;
               fcontent = content;
               extrinsic = props;
               intrinsic = intr;
             ⟩ in

  (w := ⟨ !w with files = insert_assoc (o, file) !w.files ⟩;
   (match plugin with
   | SOME x → w := ⟨ !w with plugins = insert_assoc (o, x) !w.plugins ⟩
   | _ → ());
   check_and_add_properties file.intrinsic;
   w := ⟨ !w with extfiles =
            upward_props (file.extrinsic ∪ file.intrinsic) !w.graphp
            ▷ fold (fun extfile p → apply_assoc p (fun x → x ∪ { o } ) extfile) !w.extfiles
        ⟩;
  )
```

B.4.3 `rm/mv`

```
let (rm : filename → unit) = fun s →
  let _ = assert (is_files_mode !w) in (* TODO ? could *)
  let _ = assert (s ∈ ls_filenames() ) in
  let o = ls_id_of_name s in
  let file = assoc o !w.files in
  w := ⟨ !w with
       files = del_assoc o !w.files;
       plugins = del_assoc o !w.plugins;
       extfiles =
         upward_props (file.extrinsic ∪ file.intrinsic) !w.graphp
           ▷ fold (fun extfile p → apply_assoc p (fun x → x \ { o } ) extfile) !w.extfiles
     ⟩

(*————————————————————————————————————————————*)
let (mv : filename → path → filename → unit) = fun oldname newpath newname →
```

```
  let _ = assert (oldname ∈ ls_filenames() ) in
  let _ = assert (is_files_mode !w) in
  let o = ls_id_of_name oldname in
  let file = assoc o !w.files in
  let oldpwd = pwd() in
  let oldprops = properties_of_formula oldpwd in
  dopath newpath (fun () →
    let _ = assert (¬ (newname ∈ ls_filenames() )) in (* in fact should erase content *)
    let _ = assert (is_files_mode !w) in
    let newpwd = pwd() in
    let newprops = properties_of_formula newpwd in
    let _ = assert (is_conjunction oldpwd) in
    let _ = assert (is_conjunction newpwd) in
    let newfile = ⟨file with
                    extrinsic = (file.extrinsic \ oldprops) ∪ newprops;
                    filename = newname;
                  ⟩ in
    (* TODO strange case when oldprops ¬ in file.extrinsic ? *)
    let oldprops = upward_props oldprops !w.graphp in
    let newprops = upward_props newprops !w.graphp in
    let common = oldprops ∩ newprops in
    let toadd = newprops \ common in
    let todel = oldprops \ common in
    (w := ⟨ !w with files = !w.files ▷ replace_assoc (o, newfile)⟩;
     w := ⟨ !w with extfiles = todel ▷ fold (fun extf p → apply_assoc p (fun x → x \ { o } ) extf) !w.extfiles⟩;
     w := ⟨ !w with extfiles = toadd ▷ fold (fun extf p → apply_assoc p (fun x → x ∪ { o } ) extf) !w.extfiles⟩;
    )
   )
```

B.4.4 `rmdir/mvdir`

```
let (rmdir : string → unit) = fun name →
  let p = PROP name in
  let _ = assert (p ∈ ( !w.graphp ▷ nodes)) in
  let _ = assert (is_files_mode !w) in (* TODO ? could *)
  (* TODO check have no child or adjust axioms *)
  let extprop = assoc p !w.extfiles in
  let parents = !w.graphp ▷ predecessors p in
  let children = !w.graphp ▷ successors p in (* assert no child *)
  w := ⟨ !w with
       graphp = !w.graphp
                  ▷ (fun g → parents ▷ fold (fun g prop → g ▷ del_arc (prop, p)) g)
                  ▷ del_node p;
```

```
        extfiles = del_assoc p !w.extfiles ;
        files = extprop ▷ fold (fun files id →
          let file = assoc id !w.files in
          files ▷ replace_assoc (id, ⟨file with
                                       extrinsic = file.extrinsic \ { p } ;
                                       intrinsic = file.intrinsic \ { p } ;
                                     ⟩ )
         ) !w.files
      ⟩

(*————————————————————————————————————————*)
let (mvdir : string → path → string → unit) = fun oldname newpath newname →
  let oldp = PROP oldname in
  let newp = PROP newname in

  let _ = assert (is_files_mode !w) in (* TODO ? could *)
  let _ = assert (oldp ∈ ( !w.graphp ▷ nodes)) in
  let _ = if newp ≠ oldp then assert (¬ (newp ∈ ( !w.graphp ▷ nodes))) in

  let oldpwd = pwd() in
  let oldprops = properties_of_formula oldpwd in

  dopath newpath (fun () →

    let newpwd = pwd() in
    let newprops = properties_of_formula newpwd in

    let _ = assert (is_files_mode !w) in
    let _ = assert (is_conjunction oldpwd) in
    let _ = assert (is_conjunction newpwd) in

    let oldaxioms = !w.graphp ▷ predecessors oldp in
    let newaxioms = (oldaxioms \ oldprops) ∪ newprops in
    (* let _ = assert we add stuff in (* TODO when del stuff *) *)
    let newprops = upward_props newprops !w.graphp in
    let extprop = assoc oldp !w.extfiles in

    w := ⟨ !w with extfiles = newprops ▷ fold (fun extf p → apply_assoc p (fun x → x ∪ extprop) extf) !w.extfiles⟩ ;
    w := ⟨ !w with graphp = newaxioms ▷ fold (fun g prop → g ▷ add_arc (prop, oldp)) !w.graphp⟩ ;
    w := ⟨ !w with graphp = oldaxioms ▷ fold (fun g prop → g ▷ del_arc (prop, oldp)) !w.graphp⟩ ;
    (* TODO ¬ good *)
    (* avoid cycle TODO *)
    (* TODO change name *)
    )
```

B.4.5 `read/write`

```
let (read : filename → filecontent) = fun name →
  let idfile = ls_id_of_name name in
  let file = assoc idfile !w.files in

  if is_files_mode !w
  then file.fcontent
  else view (pwd ()) (context !w) (assoc idfile !w.parts) ▷ fst

(*————————————————————————————————————————*)
let (write : filename → filecontent → 'a) = fun name newcontent →
  let idfile = ls_id_of_name name in
  let file = assoc idfile !w.files in

  if is_files_mode !w
  then todo()
  else
    let info = assoc idfile !w.parts in
    let (_, marks) = view (pwd ()) (context !w) info in
    let newcontent = update_view newcontent info marks in
    let newfile = ⟨file with
                    fcontent = newcontent;
                    (* intrinsic = finalcontent ▷ (transducer !w file.filename) ; *)
                  ⟩ in

    let (newinfo, toadd, todel) =
      reindex_parts idfile newcontent (file, info) (adv_transducer !w file.filename) in

    let extparts = !w.extparts in
    let extparts =
      todel ▷ fold (fun extparts o →
        upward_props (assoc o info.parts_info).pdescription !w.graphp
          ▷ fold (fun extparts p → apply_assoc p (fun x → x \ { o } ) extparts) extparts
      ) extparts in
    let extparts =
      toadd ▷ fold (fun extparts o →
        upward_props (assoc o newinfo.parts_info).pdescription !w.graphp
          ▷ fold (fun extparts p → apply_assoc p (fun x → x ∪ { o } ) extparts) extparts
      ) extparts in
    (w := ⟨ !w with files = !w.files ▷ replace_assoc (idfile, newfile)⟩ ;
     check_and_add_properties (⋃_{id∈(toadd)} ((assoc id newinfo.parts_info).pdescription));
     w := ⟨ !w with
             extparts = extparts;
             parts = !w.parts ▷ replace_assoc (idfile, newinfo);
```

〉
)

(need index intrinsic *)*

Annexe C

Exemples de *plug-ins*

f u cn rd ths, u cn gt a gd jb n cmptr prgmmng.

Anonymous

Real programmers don't comment their code. If it was hard to write, it should be hard to understand.

Anonymous

Nous présentons dans cette section le source de quelques *plug-ins* LFS. Nous nous sommes bornés à ne présenter qu'un seul source par type de *plug-ins*. Ainsi, pour les moteurs de déduction logique nous ne présentons que le source O'Caml du moteur gérant la logique sur les entiers. Le prototype LFS actuel contient aussi des moteurs pour gérer des logiques sur des chaînes de caractères, des dates et durées, des types, et sur des propriétés de sécurité (voir Section 4.1.3 page 115). Pour les transducteurs nous présentons le source Perl du transducteur de fichiers MP3 (voir Section 6.2.1 page 148). Pour les transducteurs avancés nous présentons le source O'Caml du transducteur de fichiers BibTeX (voir Section 6.2.6 page 155). Le prototype LFS contient d'autres transducteurs, notamment ceux mentionnés Section 6.2, avec donc des transducteurs pour gérer des pages `man`, des e-mails, et des transducteurs avancés pour gérer des fichiers C, Perl, LaTeX, et O'Caml.

Le transducteur avancé pour O'Caml permet ainsi d'avoir différentes vues sur la spécification et l'implémentation de LFS présentées dans les deux annexes précédentes. Le transducteur avancé LaTeX permet d'avoir différentes vues sur cette thèse.

C.1 Un moteur de déduction logique

```
(* INT logic (syntax = <,>, [x..y], sugar for ≤, ≥ *)
type property = PROP of string
type logic = (property → property → bool) (* mean : a ⊨ b ? *)

type interval = VAL of int | SUP of int | INF of int | IN of (int × int)

let parse_interval = fun s →
  match s with
  | s when s ≈ "^[0-9]+$" → VAL (s_to_i s)
  | s when s ≈ "^>([0-9]+)$" → SUP (s_to_i (matched1 s))
  | s when s ≈ "^<([0-9]+)$" → INF (s_to_i (matched1 s))
  | s when s ≈ "^\[([0-9]+)\.\.([0-9]+)\]$" →
      let (x1, x2) = matched2 s ▷ pair s_to_i in
      let _ = assert(x1 < x2) in
      IN (x1, x2)
  (* sugar *)
  | s when s ≈ "^>=([0-9]+)$" → SUP (s_to_i (matched1 s) - 1)
  | s when s ≈ "^<=([0-9]+)$" → INF (s_to_i (matched1 s) + 1)
  | _ → failwith "parsing error on interval"

let (interval_logic : logic) = fun (PROP s1) (PROP s2) →
  let (x1, x2) = (parse_interval s1, parse_interval s2) in
  (match (x1, x2) with
  | (VAL x, VAL y) → x = y (* 2 ⊨ 2 *)
  | (VAL x, SUP y) → x > y (* 2 ⊨ >1 *)
  | (VAL x, INF y) → x < y (* 2 ⊨ <3 *)
  | (VAL x, IN (y, z)) → x ≤ z ∧ x ≥ y (* 2 ⊨ [0..3] *)
  | (SUP x, SUP y) → x ≥ y (* >3 ⊨ >2 *)
  | (INF x, INF y) → x ≤ y (* <2 ⊨ <3 *)
  | (IN (x1,y1), IN (x2, y2)) → x1 ≥ x2 ∧ y1 ≤ y2 (* [2..3] ⊨ [0..4] *)
  | (IN (x,y), SUP z) → x > z (* [1..4] ⊨ >0 *)
  | (IN (x,y), INF z) → y < z (* [1..4] ⊨ <5 *)
  | _ → false)
(* make ml solver stdin(argv)/stdout compliant *)
let (interact_logic : logic → unit) = fun (⊨) →
  match (ARRAY.to_list SYS.argv ) with
  | [_;x;y] → if (PROP x) ⊨ (PROP y)
              then print_endline "yes"
              else print_endline "no"
  | _ : :xs → failwith ("give me 2 formulas, not "^(i_to_s (length xs))^"\n"
                        "I was given :\n"^(xs ▷ join "\n")^"\n")
  | _ → raise IMPOSSIBLE

let main = interact_logic interval_logic
```

C.2 Un transducteur

```
#!/usr/bin/perl
#################
# MP3 transducer
#################

use MP3::Info;
use MP3::Tag;

my $path = $ARGV[0];

my $tag = MP3::Tag->new($path);
if($tag) {
    $tag->get_tags();
    if (exists $tag->{ID3v1}) {
my $last = $tag->{ID3v1};
push @aux,
  ("title:"  .  $last->song,
   "artist:" .  $last->artist,
   "album:"  .  $last->album,
   "comment:" . $last->comment,
   "year:"   .  $last->year,
   "genre:" .   $last->genre,
  );
    } else {
my $tag = get_mp3tag($path);
if($tag) { map { my $e = $_; push @aux, (lc $e . ":$tag->{$e}") }
                         qw(TITLE ARTIST ALBUM YEAR COMMENT GENRE) }
    }
}

my $tag2 = get_mp3info($path);
if($tag2) { map { my $e = $_; push @aux, (lc $e . ":$tag2->{$e}") }
    qw(TIME BITRATE FREQUENCY)
}

print (join "/", @aux); print "\n";
```

C.3 Un transducteur avancé

```
(* BibTeX advanced transducer *)
type property = Prop of string
type adv_transducer = (partcontent list → (property set) list)
      and partcontent = string

let bibtex_adv_transducer = fun xs →
  let state = ref ∅ in (* THE INTERNAL STATE *)
  let rec aux = function
    | [] → []
    | s : :xs →
        let props =
          if s ≈ "[ \t]*@(.*)⟨(.*),"  then (* @ symbol => we've got a new entry, such as @book⟨... *)
            let (typ, ref) = matched2 s in
            let xs' = take_until (fun s → s ≈ "[ \t]*⟩") xs in
            let props = [Prop ("type :" ^ typ); Prop ("ref :" ^ ref)] in
            let _ = state :=
              xs' ▷ fold (fun a s →
                (
                 match () with
                 | _ when s ≈ "[ \t]*author[ \t]*=[ \t]*⟨(.*)⟩," →
                     matched 1 s ▷ split "[ \t]*and[ \t]*" ▷ map (fun s → Prop ("author :" ^ s))
                 | _ when s ≈ "[ \t]*year[ \t]*=[ \t]*([0-9]+)" →
                     [Prop ("year :" ^ matched 1 s)]
                 | _ when s ≈ "[ \t]*institution[ \t]*=[ \t]*(.*)," →
                     [Prop ("institution :" ^ matched 1 s)]
                 | _ when s ≈ "[ \t]*title[ \t]*=[ \t]*⟨(.*)⟩," →
                     [Prop ("title :" ^ matched 1 s)]
                 | _ when s ≈ "[ \t]*keywords[ \t]*=[ \t]*⟨(.*)⟩," →
                     matched 1 s ▷ split "[ \t]*[,;][ \t]*" ▷ map (fun s → Prop ("domain :" ^ s))
                 | _ → []
                ) ++ a
                        ) props in
            (Prop "synchro") : : !state (* SYNCHRONISATION POINT, a new bibtex entry => a new unit *)
          else !state in
        props : :aux xs
  in aux xs
(* make ml adv_transducer stdin(argv)/stdout compliant *)
let (interact_adv_transducer : adv_transducer → unit) = fun advtrans →
  let rec aux () =
    try let x = read_line () in x : :aux()
    with End_of_file → [] in
  aux () ▷ advtrans ▷ iter (fun ps → ps ▷ map (fun (Prop s) → s) ▷ join "/" ▷ print_endline)

let main = interact_adv_transducer bibtex_adv_transducer
```

Bibliographie

[Abr94] P. Abrahamsen. *All Mode,* 1994. Emacs Lisp Archive. cité page(s) 177

[AIS93] R. Agrawal, T. Imielinski, and A. N. Swami. *Mining Association Rules between Sets of Items in Large Databases.* In ACM SIGMOD Int. Conf. Management of Data, 1993. cité page(s) 156

[AISS98] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. *Ufo : A Personal Global File System Based on User-Level Extensions to the Operating System.* ACM Transactions on Computer Systems, 1998. cité page(s) 47

[AL92] P. W. Abrahams and B. R. Larson. *UNIX for the Impatient*. Addison Wesley, 1992. cité page(s) 11, 82

[App04] Apple. *Spotlight*, 2004. `http://www.apple.com/macosx/tiger/spotlight.html`. cité page(s) 30

[AU92] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. Computer Science Press, W.H. Freeman and Co., 1992. cité page(s) 89

[BDB+94] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. *A File System for Information Management.* In Int. Conf. Intelligent Information Management Systems (ISMM), 1994. cité page(s) 27, 62, 152, 175, 183

[BDMS94] C. M. Bowman, Peter B. Danzig, U. Manber, and M. F. Schwartz. *Scalable Internet Resource Discovery : Research Problems and Approaches.* Communications of the ACM, 1994. cité page(s) 58, 176

[BEH99] P. Bruno, V. Ehrenberg, and L. E. Holmquist. *STARzoom - An Interactive Visual Interface to a Semantic Database.* In ACM Intelligent User Interfaces (IUI), 1999. cité page(s) 176

[Ben85] J. L. Bentley. *Programming Pearls : A Spelling Checker.* Communications of the ACM, 1985. cité page(s) 17

[Ben86] J. L. Bentley. *Programming Pearls : Little Languages.* Communications of the ACM, 1986. cité page(s) 50

[BG88] N. S. Borenstein and J. Gosling. *UNIX Emacs : a Retrospective. Lessons for Flexible System Design.* In ACM SIGGRAPH Symp. on User Interface Software, 1988. cité page(s) 129

[BJ00] A. P. Black and M. P. Jones. *Perspectives On Software.* In OOPLSA Workshop on Advanced Separation of Concerns, 2000. cité page(s) 177

[Bla93] M. Blaze. *A Cryptographic File System for UNIX*. In ACM Conf. on Computer and Communications Security, 1993. cité page(s) 35

[BMR82] D. Brownbridge, L. Marshall, and B. Randell. *The Newcastle connection, or UNIXes of the World Unite !* Software — Practice and Experience, 1982.
disponible dans [Han01]. cité page(s) 21

[Bou78] S. R. Bourne. *The UNIX Shell.* The Bell System Technical Journal, 1978. cité page(s) 11

[BP98] S. Brin and L. Page. *The anatomy of a large-scale hypertextual Web search engine.* Computer Networks and ISDN Systems, 1998. cité page(s) 60

[Bro75] F. P. Brooks, Jr. *The Mythical Man Month : Essays on Software Engineering*. Addison Wesley, 1975. cité page(s) 142

[Bro00] J. G. Brookshear. *Computer Science, An Overview*. Addison Wesley, 2000. cité page(s) 44

[BYRN99] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison Wesley, 1999. cité page(s) 58, 97, 99, 222, 224, 226

[Cal01] C. Calvelli. *PerlFS*, 2001. `http://perlfs.sourceforge.net/`. cité page(s) 49, 142

[Cat92] V. Cate. *Alex – A Global File System.* In USENIX File System Workshop, 1992. cité page(s) 22

[CER90] J. R. Cordy, N. L. Eliot, and M. G. Robertson. *TuringTool : A User Interface to Aid in the Software Maintenance Task.* IEEE Transactions on Software Engineering, 1990. cité page(s) 177, 183

[CG91] V. Cate and T. Gross. *Combining the Concepts of Compression and Caching for a two-level filesystem.* In Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1991. cité page(s) 35

[Cha96] J. Chazarain. *Programmer avec Scheme - De la pratique à la théorie*. International Thomson Publishing France, 1996. cité page(s) 4, 78

[Chi97] Y. Chiaramella. *Browsing and Querying : two complementary approaches for Multimedia Information Retrieval.* In Int. Conf. Hypertext Information Retrieval Multimedia (HIM), 1997. cité page(s) 176

[CLG+94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. *RAID : High-Performance, Reliable Secondary Storage.* ACM Computing Surveys, 1994. cité page(s) 44

[CNR90] Y.F. Chen, M. Nishimoto, and C. V. Ramamoorthy. *The C Information Abstraction System.* IEEE Transactions on Software Engineering, 1990. cité page(s) 176, 183

[Cod70] E. F. Codd. *A Relational Model for Large Shared Data Banks.* Communications of the ACM, 1970. cité page(s) 61

[Com79] D. Comer. *The Ubiquitous B-Tree.* ACM Computing Surveys, 1979. cité page(s) 42, 51, 107, 144

[Com84] D. Comer. *Operating System Design, the XINU approach*. Prentice-Hall, 1984. cité page(s) 47

[com00] Eclipse community. *The Eclipse Homepage*, 2000. `http://www.eclipse.org`. cité page(s) 67, 175

[Cos98] R. Di Cosmo. *Piège dans le cyberespace*, 1998. `http://www.pps.jussieu.fr/~dicosmo/Piege/cybersnare/`. cité page(s) 42

[CP00] C. Collberg and T. Proebsting. *AlgoVista - A Search Engine for Computer Scientists.* Technical report, University of Arizona, 2000. cité page(s) 181

[CPKT92] D. R. Cutting, J. O. Pedersen, D. Karger, and J. W. Tukey. *Scatter/Gather : A Cluster-based Approach to Browsing Large Document Collections.* In Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1992.
décrit en partie dans [BYRN99]. cité page(s) 176, 183

[CQ83] J. Chambers and J. Quarterman. *UNIX System III and 4.1BSD ; a Practical comparison.* In USENIX Winter Conf., 1983. cité page(s) 18

[CS00] R. Cole and G. Stumme. *CEM - a Conceptual Email Manager.* In Int. Conf. on Conceptual Structures (ICCS), 2000. cité page(s) 177

[CTT94] R. Card, T. Ts'o, and S. Tweedie. *Design and Implementation of the Second Extended filesystem.* In Dutch Int. Symp. on Linux, 1994. cité page(s) 40, 144

[Dat99] C. J. Date. *Introduction to Database Systems.* Addison Wesley, 1999. cité page(s) 61

[DGHK+75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. Levy. *A Structure Oriented Program Editor : A First Step Towards Computer Assisted Programming.* Rapport de Recherche 114, INRIA, Rocquencourt, France, 1975. cité page(s) 177

[Dij87] E. W. Dijkstra. *The Humble Programmer (*1972). In ACM Turing Award Lectures : The First Twenty Years. ACM Press Anthology Series, Addison-Wesley, 1987. cité page(s) 199

[DN65] R. C. Daley and P. G. Neumann. *A General-Purpose File System for Secondary Storage.* In AFIPS, 1965.
disponible dans [Han01]. cité page(s) 11

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990. cité page(s) 89

[EE68] D. C. Engelbart and W. K. English. *A Research Center for Augmenting Human Intellect.* AFIPS, 1968. cité page(s) 176

[EP92] P. R. Eggert and D. S. Parker. *An Intensional File System.* In USENIX File Systems Workshop, 1992. cité page(s) 19, 22

[Fel79] S. I. Feldman. *Make-a Program for Maintaining Computer Programs.* Software — Practice and Experience, 1979. cité page(s) 20

[Fer99] S. Ferré. *Systèmes d'Information Logiques.* DEA, Université de Rennes 1, 1999. cité page(s) 3, 178, 179

[Fer02] S. Ferré. *Systèmes d'Information Logiques : un paradigme Logico-Contextuel pour Interroger, Naviguer et Apprendre.* Thèse d'université, Université de Rennes 1, 2002. cité page(s) 3, 178, 179

[Fit96] J. Fitzhardinge. *Userfs : A file system for Linux,* 1996. `ftp://sunsite.unc.edu/pub/Linux/ALPHA/userfs/`. cité page(s) 22, 49, 142

[FK90] C. W. Fraser and B. Krishnamurthy. *Live Text (editing).* Software — Practice and Experience, 1990. cité page(s) 177

[FR00] S. Ferré and O. Ridoux. *A File System Based on Concept Analysis.* In Int. Conf. Rules and Objects in Databases, LNCS 1861. Springer, 2000. cité page(s) 178

[FR04] S. Ferré and O. Ridoux. *An Introduction to Logical Information Systems.* Information Processing & Management, 2004. cité page(s) 3, 176, 178, 183

[FY94] D. Filo and J. Y. Yang. *Yahoo - Yet Another Hierarchical Officious Oracle,* 1994. `http://www.yahoo.com`. cité page(s) 58

[Gab91] R. P. Gabriel. *Worse Is Better,* 1991. `http://www.dreamsongs.com/WorseIsBetter.html`. cité page(s) 179

[Gia93] D. Giampaolo. *CAT-FS, a Content Addressable, Typed File System.* Master's thesis, Worcester Polytechnic Institute, 1993. cité page(s) 26, 62, 175, 183

[Gia99] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, 1999. cité page(s) 30

[GJSJ91] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole Jr. *Semantic File Systems.* In ACM Symp. Operating Systems Principles (SOSP), 1991. cité page(s) 23, 24, 60, 62, 162, 175, 178, 183

[GM99] B. Gopal and U. Manber. *Integrating Content-Based Access Mechanisms with Hierarchical File Systems.* In ACM Symp. Operating Systems Design and Implementation (OSDI), 1999. cité page(s) 29, 62, 175, 183

[Gol84] A. Goldberg. *Smalltalk - the Interactive Programming Environment*. Addison-Wesley, 1984. cité page(s) 64, 175, 177

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis : Mathematical Foundations*. Springer, 1999. cité page(s) 177

[Han01] P. B. Hansen. *Classic Operating Systems*. Springer, 2001. cité page(s) 221, 223

[HK97] M. A. Hearst and C. Karadi. *Cat-a-Cone : An Interactive Interface for Specifying Searches and Viewing Retrieval Results Using a Large Category Hierarchy.* In Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1997.
décrit en partie dans [BYRN99]. cité page(s) 176

[HKP95] M. A. Hearst, D. R. Karger, and J. O. Pedersen. *Scatter/Gather as a Tool for the Navigation of Retrieval Results.* In Working Notes AAAI Fall Symp. AI Applications in Knowledge Navigation, 1995.
décrit en partie dans [BYRN99]. cité page(s) 176

[HM76] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison.* Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, 1976. cité page(s) 103

[HMS01] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001. cité page(s) 156

[HN86] A. N. Habermann and D. Notkin. *Gandalf : Software Developement Environments.* IEEE Transactions on Software Engineering, 1986. cité page(s) 177

[HP94] J. S. Heidemann and G. J. Popek. *File-system Development with Stackable Layers.* ACM Transactions on Computer Systems, 1994. cité page(s) 49

[HRB96] S. Le Huitouze, O. Ridoux, and P. Brisset. *Prolog/Mali Reference Manual.* Technical report, IRISA, 1996. cité page(s) 147

[JV04] D. Janzen and K. De Volder. *Programming with Crosscutting Effective Views.* In European Conf. on Object-Oriented Programming (ECOOP), 2004. cité page(s) 177

[Kel96] J. J. Kelly. *The Essence of logic*. Prentice Hall, 1996. cité page(s) 78

[Kil84] T. J. Killian. *Processes as Files.* In USENIX Summer Conf., 1984. cité page(s) 16

[Kis95] O. Kiselyov. *A dream of an ultimate OS*, 1995. `http://okmij.org/ftp/DreamOS.html`. cité page(s) 175

[Kle86] S. R. Kleiman. *Vnodes : An Architecture for Multiple File System Types in Sun UNIX*. In USENIX Summer Conf., 1986. cité page(s) 20, 45

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, L. Loingtier, and J. Irwin. *Aspect-Oriented Programming.* In Object-Oriented Programming European Conference (ECOOP), 1997. cité page(s) 176

[KM81] B. W. Kernighan and J. R. Mashey. *The Unix Programming Environment.* IEEE Computer, 1981. cité page(s) 177

[KM91] B. Kahle and A. Medlar. *An Information System for Corporate Users : Wide Area Information Servers.* Technical Report TR-199, Thinking Machines Corporation, 1991. cité page(s) 60

[KMM00] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning : An Approach*. Kluwer Academic Publishers, 2000. cité page(s) 180

[KMN+88] H. J. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, N. Sidebotham, and M. West. *Scale and Performance in a Distributed File system.* ACM Transactions on Computer Systems, 1988. cité page(s) 147, 152, 162

[KP76] B. W. Kernighan and P. J. Plauger. *Software Tools*. Addison Wesley, 1976. cité page(s) 17

[Kru95] P.B. Kruchten. *The $4+1$ view model of architecture.* IEEE Software, 1995. cité page(s) 66

[LDG+04] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.08*, 2004. `http://caml.inria.fr/ocaml/htmlman/index.html`. cité page(s) 4, 142, 185

[LFJ83] S. J. Leffler, R. S. Fabry, and W. N. Joy. *a 4.2BSD Interprocess Communication Primer.* Technical Report CSD-83-145, University of California, Berkeley, 1983. cité page(s) 18

[Lin84] M. A. Linton. *Implementing Relational Views of Programs.* In ACM Software Engineering Symp. on Practical Software Development Environments, 1984. cité page(s) 177, 183

[Lin95] C. Lindig. *Concept-Based Component Retrieval.* In IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs, 1995. cité page(s) 177, 178, 183

[Mar] E. Marchand. *Bib2html.* `http://www.irisa.fr/prive/marchand/bib2html/`. cité page(s) 156

[Mic06] Microsoft. *WinFS*, 2006. `http://longhorn.msdn.microsoft.com/lhsdk/winfs/daovrwelcometowinfs.aspx`. cité page(s) 30

[MJLF84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. *A Fast File System for UNIX*. ACM Transactions on Computer Systems, 1984. cité page(s) 35, 40

[MN86] D. A. Miller and G. Nadathur. *Higher-order Logic Programming.* In Int. Conf. Logic Programming (ICLP), 1986. cité page(s) 147

[MTW95] R. Miller, O. Tsatalos, and J. Williams. *Integrating Hierarchical Navigation and Querying : A User Customizable Solution.* In ACM Multimedia Workshop on Effective Abstractions in Multimedia Layout, Presentation, and Interaction, 1995. cité page(s) 176

[MV04] E. McCormick and K. De Volder. *JQuery : Finding your way through Tangled Code.* ACM SIGPLAN Notices, 2004. cité page(s) 176

[MW94] U. Manber and S. Wu. *GLIMPSE : A Tool to Search Through Entire File Systems.* In USENIX Winter Conf., 1994. cité page(s) 99, 150

[Mye86] E. W. Myers. *An $O(ND)$ Difference Algorithm and its Variations.* Algorithmica, 1986. cité page(s) 102

[Neu92] B. C. Neuman. *The Prospero File System : A Global File System Based on the Virtual System Model.* Computing Systems, 1992. cité page(s) 22

[Nie96] J. Nielsen. *The Impending Demise of File Systems (or The Death of File Systems).* IEEE Software, 1996. cité page(s) 175

[OBS99] M. A. Olson, K. Bostic, and M. Seltzer. *Berkeley DB.* In FREENIX Track : USENIX Annual Technical Conf., 1999. cité page(s) 144, 199

[Org72] E. I. Organick. *The Multics System : An Examination of Its Structure*. MIT Press, 1972. cité page(s) 11

[Par72] D. L. Parnas. *On the Criteria to be Used in Decomposing Systems into Modules.* Communications of the ACM, 1972. cité page(s) 65, 176

[Pau90] R. Pausch. *Stage3 research,* 1990. `http://alice.etc.cmu.edu/stage3/whystage3.html`. cité page(s) 85

[PBMW98] L. Page, S. Brin, R. Motwani, and T. Winograd. *The Pagerank Citation Ranking : Bringing Order to the Web.* Technical report, Stanford Digital Library Technologies Project, Stanford University, 1998. cité page(s) 1

[PD90] R. Prieto-Díaz. *Implementing Faceted Classification for Software Reuse.* In Int. Conf. on Software Engineering (ICSE), 1990. cité page(s) 65

[Ped93] G. S. Pedersen. *A Browser for Bibliographic Information retrieval based on an Application of Lattice Theory.* In Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1993. cité page(s) 176

[Pik91] R. Pike. $8^{1/2}$*, the Plan 9 Window System.* In USENIX Summer Conf., 1991. cité page(s) 19

[PM95] J. Pendry and M. K. McKusick. *Union Mounts in 4.4BSD-Lite.* In USENIX Technical Conf., 1995. cité page(s) 22

[PPD+95] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. *Plan 9 from Bell Labs.* Computing Systems, 1995. cité page(s) 14, 19, 30

[PR03a] Y. Padioleau and O. Ridoux. *A Logic File System.* In USENIX Annual Technical Conf., 2003. cité page(s) 177

[PR03b] Y. Padioleau and O. Ridoux. *A Parts-of-File File System.* In Conférence Française sur les Systèmes d'Exploitation, 2003. cité page(s) 226

[PR05] Y. Padioleau and O. Ridoux. *A Parts-of-File File System.* In USENIX Annual Technical Conf. (short paper), 2005.
une version étendue est disponible dans [PR03b]. cité page(s) 177

[Ray04] E. S. Raymond. *The Art of UNIX Programming*. Addison Wesley, 2004. cité page(s) 177

[Rei84] S. P. Reiss. *Graphical Program Development with PECAN Program Development Systems.* In ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments, 1984. cité page(s) 66

[RGL87] J. R. Remde, L. M. Gomez, and T. K. Landauer. *SuperBook : An Automatic Tool for Information Exploration - Hypertext ?* In ACM Hypertext, 1987.
décrit en partie dans [BYRN99]. cité page(s) 176, 183

[Rit89] M. Rittri. *Using Types as Search Keys in Function Libraries.* In Int. Conf. on Functional Programming and Computer Architecture (FPCA). 1989. cité page(s) 65

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language, Reference Manual*. Addison-Wesley, 1999. cité page(s) 66

[RO92] M. Rosenblum and J. K. Ousterhout. *The Design and Implementation of a Log-Structured File System.* ACM Transactions on Computer Systems, 1992. cité page(s) 43

[Roo92] W. D. Roome. *3DFS : A Time-Oriented File Server.* In USENIX Winter Conf., 1992. cité page(s) 30

[RQY94] R. Russell, D. Quinlan, and C. Yeoh. *Filesystem Hierarchy Standard,* 1994. `http://www.pathname.com/fhs/`. cité page(s) 155

[RT74] D. M. Ritchie and K. Thompson. *The UNIX Time-Sharing System.* Communications of the ACM, 1974. cité page(s) 11, 14, 58

[RU93] R. Ramakrishan and J. D. Ullman. *A Survey of Research on Deductive Database Systems.* Journal of Logic Programming, 1993. cité page(s) 131

[SFHV99] D. Santry, M. Feeley, N. Hutchinson, and Veitch. *Elephant : The File System That Never Forgets.* In Workshop on Hot Topics in Operating Systems (HotOS), 1999. cité page(s) 30

[SG86] R. W. Scheifler and J. Gettys. *The X Window System.* ACM Transactions on Graphics, 1986. cité page(s) 19

[SG94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts.* Addison Wesley, 1994. cité page(s) 47

[SG03] C. A. N. Soules and G. R. Ganger. *Why can't I find my files ?* In Workshop on Hot Topics in Operating Systems (HotOS), 2003. cité page(s) 175

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. *Design and Implementation of the Sun Network Filesystem.* In USENIX Summer Conf., 1985. cité page(s) 21

[SJCC96] B. R. Schatz, E. H. Johnson, Cochrane, and H. Chen. *Interactive Term Suggestion for Users of Digital Libraries : Using Subject Thesauri and Co-occurrence Lists for Information Retrieval.* In ACM Digital Library Conf., 1996. cité page(s) 176

[SRR97] R. Seltzer, D. S. Ray, and E. J. Ray. *The AltaVista Search Revolution : How to find anything on the Internet.* McGraw-Hill, 1997. cité page(s) 60

[Sta81] R. M. Stallman. *EMACS : The Extensible, Customizable, Self-Documenting Display Editor.* Technical Report AIM-519A, MIT Artificial Intelligence Laboratory, 1981. cité page(s) 13, 67, 175

[Sta85] R. M. Stallman. *The GNU Manifesto.* Dr. Dobb's Journal of Software Tools, 1985. `http://www.gnu.org/gnu/manifesto.html`. cité page(s) 182

[Sta02] R. M. Stallman. *GNU Emacs Manual.* GNU Press, 2002. cité page(s) 47, 65, 175, 178

[Ste02] L. Stein. *`ocamlbdb`,* 2002. `http://www.eecs.harvard.edu/~stein/bdbfs/`. cité page(s) 144

[Sun90] V. S. Sunderam. *PVM : a Framework for Parallel Distributed Computing.* Concurrency, Practice and Experience, 1990. cité page(s) 21

[Tei84] W. Teitelman. *A Tour Through Cedar.* In Int. Conf. on Software Engineering (ICSE), 1984. cité page(s) 177

[Tic85] W. F. Tichy. *RCS — a System for Version Control.* Software — Practice and Experience, 1985. cité page(s) 30

[Tip95] F. Tip. *A Survey of Program Slicing Techniques.* Journal of programming languages, 1995. cité page(s) 177

[TM81] W. Teitelman and L. Masinter. *The Interlisp Programming Environment.* IEEE Computer, 1981. cité page(s) 176, 177

[TOHS99] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. *N Degrees of Separation : Multi-Dimentional Separation of Concerns.* In Int. Conf. on Software Engineering (ICSE), 1999. cité page(s) 175, 177

[TR81] T. Teitelbaum and T. W. Reps. *The Cornell Program Synthesizer : A Syntax-Directed Programming Environment.* Communications of the ACM, 1981. cité page(s) 177

[vR86] C. J. van Rijsbergen. *A New Theoretical Framework for Information Retrieval.* In Int. ACM SIGIR Conf. on Research and Development in Information Retrieval. ACM, 1986. cité page(s) 176

[WO88] B. B. Welch and J. K. Ousterhout. *Pseudo Devices : User-Level Extensions to the Sprite File System.* In USENIX Summer Conf., 1988. cité page(s) 18

[Woo83] J. A. Woods. *Finding Files Fast.* ;login : the USENIX Association newsletter, 1983. cité page(s) 99

[YSLH03] K-P. Yee, K. Swearingen, K. Li, and M. Hearst. *Faceted Metadata for Image Search and Browsing.* In ACM CHI Conf. on Human Factors in Computing Systems, Searching and organizing, 2003. cité page(s) 176, 183

[Zim95] P. R. Zimmermann. *The Official PGP User's Guide.* MIT Press, 1995. cité page(s) 34

[ZN00] E. Zadok and J. Nieh. *FiST : A Language for Stackable File Systems.* In USENIX Annual Technical Conf., 2000. cité page(s) 50

Index

M

Q

R

S

T

U

V

W

X

Y

Z

Résumé

Les systèmes d'information offrent des moyens pour *organiser*, *chercher*, et *manipuler* l'information. Ils deviennent de plus en plus important avec l'avènement de l'âge numérique et l'augmentation de la variété et du nombre des documents digitaux (fichiers musicaux, images, vidéos, e-mails, pages Web, sources de programmes, documents XML). Pour rechercher ces documents, les systèmes d'information traditionnels, comme les systèmes de fichiers ou le Web avec ses moteurs de recherche, fournissent chacun des outils de *navigation* et d'*interrogation*, mais ne permettent pas de les combiner. D'un côté, la navigation est intuitive et progressive mais elle implique une classification des données rigide et unique. De l'autre côté, l'interrogation apporte flexibilité et expressivité, mais ne possède pas les avantages de la navigation. Pour manipuler le contenu de ces documents, d'autres outils existent, comme les éditeurs de texte ou les environnements de développement, mais ils souffrent des mêmes limitations. Nous proposons un nouveau paradigme pour les systèmes d'information basé sur la *logique* : le *Logic File System* (LFS), qui offre une organisation expressive, une recherche combinant interrogation et navigation, et une facilité de manipulation, à la fois des fichiers et de leurs contenus ; le tout d'une façon intégrée et mis en œuvre au niveau système de fichier.
Pour atteindre cette intégration, ce paradigme associe des *propriétés logiques* aux fichiers et *parties* de fichiers, et la *déduction logique* sert de base pour naviguer et interroger. *Les chemins sont des formules*. Les répertoires représentent des requêtes logiques et déterminent des ensembles de fichiers et de parties de fichiers dont les propriétés *satisfont* la requête. Le répertoire racine représente la formule *vrai*, et les sous-répertoires d'un répertoire sont déterminés par les propriétés les plus générales permettant de *raffiner* la requête, combinant ainsi navigation et interrogation. Le contenu des fichiers est déterminé par les parties du fichier original qui satisfont la requête. Cela permet des accès simultanés en lecture et écriture sur différentes *vues* d'un même fichier, afin d'aider l'utilisateur à séparer et discerner les différents *aspects* de l'information qui sont contenus dans ce fichier. Les propriétés peuvent être attachées aux fichiers manuellement par l'utilisateur et automatiquement par des programmes appelés *transducteurs*, et peuvent être ordonnées manuellement par l'utilisateur pour former des taxinomies ou automatiquement par des *moteurs de déduction logique*. Les utilisateurs peuvent étendre dynamiquement le système en fournissant leurs propres moteurs de déduction logique et transducteurs.
Nous présentons les principes, usages, algorithmes, structures de données et détails d'implémentation de LFS. Nous présentons aussi un *modèle de sécurité* pour ce système de fichier, basé sur la logique, et qui s'intègre naturellement avec «l'esprit logique» de LFS. Nous présentons des extensions à cet esprit logique comme la possibilité de formuler des requêtes Prolog, et des extensions réflexives unifiant fichiers et propriétés. Finalement, des expérimentations concrètes utilisant un prototype de LFS sont présentées, démontrant la viabilité de l'approche LFS pour la gestion de l'information, à la fois en terme d'efficacité et d'utilité.
Mots-clefs : système de fichier, logique appliquée, interrogation, navigation, vue