KFS : le systeme de fichiers de Kitlog

C. Fouque

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: +33(1)39 63 55 11

Rapports de Recherche

N° 1809

Programme 1

Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

KFS : LE SYSTÈME DE FICHIERS DE KITLOG

KFS : KITLOG FILE SYSTEM

Carole FOUQUÉ

Juillet 1992

Université Pierre et Marie Curie – Paris VI

DEA Systèmes Informatiques

Rapport de stage

KFS : Le système de fichiers de $K_i^tL^O_g$

KFS : $K_i^tL^O_g$ File System

Carole Fouqué

Juillet 1992

Directeur de stage : Marc Shapiro

Lieu du stage : projet SOR, INRIA, Rocquencourt

Résumé

Ce rapport décrit la conception de KFS ($K_I^T L^O_G$ File System). KFS est un système de fichiers entièrement journalisé, bati sur le service de journalisation d'usage général $K_I^T L^O_G$. Ce travail valide le modèle de $K_I^T L^O_G$, en montrant l'adéquation des services offerts par un outil de journalisation générique aux besoins spécifiques du système de fichiers. L'architecture du système de fichiers KFS est simplifiée par le fait qu'une grande part des mécanismes nécessaires (gestion des espaces libres, recherche de données, etc.) sont déjà assurés par le service de journalisation $K_I^T L^O_G$.

Abstract

This report describes the design of KFS ($K_I^T L^O_G$ File System). KFS is a log-structured file system implemented on top of the general purpose logging service $K_I^T L^O_G$. This work validates the $K_I^T L^O_G$ model by showing that services provided by a generic logging tool adequatly support the specific needs of the file system. As a result, the design of the KFS file system is simplified, because a number of usefull mechanisms such as free space management or data lookup are already supported by the $K_I^T L^O_G$ logging service.

Table des matières

Chapitre I

Introduction

L'objectif de ce stage est l'utilisation de l'outil générique $K_I{}^T L^O{}_G$. En tant que générique, tous types d'applications peuvent se servir de $K_I{}^T L^O{}_G$, tout en payant que le coût de ce qui est utilisé. Après avoir examiné plusieurs types d'applications possibles, la réalisation d'un système de fichiers, ayant uniquement comme support $K_I{}^T L^O{}_G$, a été choisi.

Ce choix a été motivé par plusieurs facteurs :

- Il permet de tester la généricité de $K_I{}^T L^O{}_G$ puisque cet outil n'a pas été pensé plus particulièrement pour ce type d'application.
- La technologie du disque (support de stockage traditionnel d'un système de fichiers) dépend de trois facteurs : son coût, sa capacité et ses performances en lecture/écriture. Les performances, et plus particulièrement le temps d'accès en lecture/écriture, constitue le goulot d'étranglement de la technologie du disque.

  L'utilisation de la journalisation, comme structure de stockage, constitue un gain de temps important lors d'écriture puisque la tête de lecture ne se déplasse plus aléatoirement mais séquentiellement. Grâce à l'utilisation importante de structures de contrôles le temps de lecture d'un fichier reste raisonable. Donc, la journalisation permet d'avoir des performance meilleurs.
- L'utilisation de la journalisation, comme seule structure de stockage pour un système de fichiers, a déjà fait l'objet de recherche. C'est donc à partir d'existant que j'ai pu travailler même si, comme nous allons le voir, les recherches déjà effectuées n'etaient pas toujours adaptées aux objectifs fixés.

Ce document s'articule autour de deux chapitres. Le premier chapitre constituant un état de l'art sur la journalisation et le deuxième chapitre étant une description du système de fichiers réalisé : KFS.

Chapitre II

Etat de l'art

1 Introduction

1.1 Définition

Un journal peut être défini comme étant un système de sauvegarde et de récupération de données écrites de manière séquentielle sur un support fiable [Daniels et al. 87, Daniels 88]. L'unité de stockage d'un journal est l'enregistrement, qui peut être de taille fixe (facile à gérer) ou variable (optimisation du stockage).

Il faut distinguer les notions de journaux logique et physique. Un journal logique est un journal tel qu'il est vu par une application. Il est composé d'enregistrements ayant un lien logique entre eux. Il est, en général, symbolisé par un identificateur unique de journal (par exemple, appelé Log-IDentifier dans KITLOG [Ruffin 92a, Ruffin 92b]). La nature du lien entre enregistrements n'est connu que par l'application. Les journaux logiques peuvent être partagés par un ou plusieurs processus d'une même application ou même par différentes applications.

Un journal physique correspond à un ensemble d'enregistrements appartenant à différents journaux logiques qui ont été multiplexés sur un même support de stockage.

Les notions de journaux physiques et logiques coïncident si le journal physique ne contient que des enregistrements d'un même journal logique et s'il les contient tous.

1.1.1 Ecriture séquentielle

La caractéristique principale d'un journal est contenue dans la notion d'écriture séquentielle : celle-ci signifie que, au moins théoriquement, les seuls accès en écriture sur le journal consistent à ajouter des enregistrements à la fin du journal. Une fois écrites sur le support physique, les informations ne sont jamais modifiées sauf en cas de compactage. De plus, l'écriture séquentielle est la principale opération effectuée en marche normale.

Généralement, la lecture est également effectuée de manière séquentielle, soit dans l'ordre selon lequel les enregistrements ont été écrits, auquel cas on parle de lecture avant, soit dans l'ordre inverse, auquel cas on parle de lecture arrière. La lecture avant est souvent utilisée pour rejouer des actions, tandis que la lecture arrière permet de défaire des actions n'ayant pas abouti. Les informations lues sur le journal peuvent également être recherchées de manière aléatoire, mais une telle lecture est une opération relativement coûteuse et réalisée plus rarement.

1.1.2 Support fiable

La seconde caractéristique d'un journal est la notion de support fiable. Par « support fiable » on entend enregistrement fiable des données, indépendamment des pannes logicielles ou matérielles qui peuvent survenir et auxquelles le système doit, par conception, résister. Il existe bien sûr toujours des pannes catastrophiques après lesquelles le système ne pourra pas redémarrer sans intervention humaine, ou qui auront entraîné des pertes de données, mais il est possible de concevoir un système de manière à ce qu'il soit capable de supporter sans dommage les pannes les plus courantes. En particulier, les données stockées sur le journal ne doivent pas être corrompues en cas de coupure de courant, de panne d'une machine ou d'atterrissage de tête de lecture sur un disque par exemple.

Pour réaliser effectivement un support d'enregistrement fiable, plusieurs techniques peuvent être mises en œuvre. On peut tout d'abord utiliser un support qui ne permette effectivement qu'une écriture définitive et inaltérable, comme un disque optique numérique non réinscriptible; il est clair que cette solution est sûre mais manque de souplesse. Une autre technique consiste à utiliser deux disques, gérés de préférence par deux contrôleurs indépendants, et utilisés « en mirroir » : toute écriture sur le journal est effectuée indépendamment sur les deux disques, qui sont donc à tout instant copies exactes l'un de l'autre; en cas de perte de données sur l'un des deux, l'autre reste disponible pour les récupérer.

On peut aussi utiliser une technique de réplication : la gestion du journal est prise en charge par plusieurs serveurs répartis sur différents sites, chacun possédant une fraction du journal total; le répartition de la charge entre les différents serveurs est effectuée de manière à ce que chaque enregistrement soit répliqué un moins un certain nombre de fois. On est ainsi assuré qu'un enregistrement sera toujours accessible, même en cas de panne d'un des sites serveurs. Cette méthode est donc intéressante du point de vue de la disponibilité du journal; elle a de plus l'avantage de distribuer géographiquement le support physique, ce qui rend le système plus résistant à des tentatives de dégradation malveillante [Deswarte 91].

Dans Camelot [Daniels et al. 87, Daniels 88] la réplication de journal est réalise en utilisant un ensemble de `N` sites. Chaque site dispose d'un gestionnaire de journal. Un enregistrement est écrit sur `W` sites (avec $W < N$). Chaque site dispose d'un morceau de journal. Il y a réplication partielle. Ceci implique que l'on tolère (`W - 1`) pannes simultanées. La demande de lecture d'un enregistrement est diffusée à l'ensemble des gestionnaires de journaux. (`N - W`) + 1 réponses des serveurs sont collectées, donc au moins un serveur possédant l'enregistrement recherché a répondu. Plus le nombre de

serveur est grand (N), plus on risque d'avoir de surcoût lors de la mise en place de l'algorithme de réplication des messages (W). D'un autre côté, le fait de disposer de nombreux serveurs diminue la charge respective, augmente la vitesse de traitement et la disponibilité des données.

Pour qu'un système soit opérationnel en cas de partitionnement du réseau [Davidson et al. 85], il faut qu'il soit fiable pour que les données restent cohérentes. La cohérence des données peut facilement être obtenue en traitant toutes les réplications de façon synchrone (le système est en attente pendant la réalisation de toutes les réplications). Dans bien des cas, cette solution n'est pas possible car elle est bien trop coûteuse en temps de traitement. Dans le cas de réplication asynchrone, une reprise sur panne après partionnement du réseau doit prendre en compte un ensemble de réplications qui ne vont pas avoir été exécutées dans leur totalité, sans que l'on sache vraiment lesquels.

La reprise sur panne de journaux centralisés est connue et efficace. Dans le domaine du répartie, les solutions sont compliquées et la prise en compte de différents types de reprise sur panne se fait au détriment de l'efficacité.

1.2 Intérêt de la notion

Des caractéristiques mises en évidence précédemment découlent les deux intérêts principaux de la technique de journalisation, à savoir la fiabilité des informations stockées sur le journal et les bonnes performances du système en écriture.

1.2.1 Fiabilité

Le premier point intéressant de la notion de journal est lié à la sûreté de fontionnement d'un système : comme les données enregistrées sur le journal ne sont jamais modifiées, elles ne courent pas le risque d'être inopinément perdues ou détruites et restent donc toujours disponibles (en cas de compactage, un surcroît de traitement est nécessaire pour garder cette propriété). Ainsi le journal offre un support de stockage d'informations plus sûr qu'un serveur de fichiers classique; il sera donc bien adapté pour sauvegarder des informations critiques pour la sûreté de fonctionnement d'un système, c'est-à-dire des informations qui ne doivent en aucun cas être perdues, comme par exemple des modifications à apporter à des fichiers de bases de données. Cette caractéristique sera utilisée lors de la mise en place des transactions atomiques en particulier, comme on le verra plus loin.

1.2.2 Performances

Le second intérêt d'un journal est relatif à ses bonnes performances en écriture, dues à une bonne utilisation des capacités du support physique : l'écriture séquentielle ne nécessite pas de mouvements de positionnement des têtes de lecture; or ce sont ces mouvements qui, dans le cas d'un système de gestion de fichiers classique par exemple, représentent la plus grande partie du coût en temps d'une écriture (par exemple dans

un système de fichiers comme Unix, chaque modification d'un fichier nécessite au moins deux écritures : la modification du fichier et de son inode). Une structure journalisée peut ainsi utiliser au mieux la bande passante disponible au niveau du support physique. Pour donner un ordre d'idée, le système de gestion de fichiers Sprite LFS [Rosenblum et Ousterhout 91] utilise jusqu'à 70 % de la bande passante en écriture des disques alors qu'un système classique tel que UNIX NFS n'en utilise au plus que 10 %.

1.3 Implémentation

La mise en œuvre effective d'un système de gestion de journal pose certains problèmes spécifiques, en particulier ceux liés au compactage des données obsolètes et à l'archivage des données anciennes. De plus, il faut offrir à l'utilisateur une interface autant que possible simple d'utilisation et transparente vis-à-vis de la distribution du journal ou du partage du support physique.

1.3.1 Compactage/archivage

Le premier problème posé par l'implémentation d'un système utilisant la journalisation est lié à cette notion d'écriture uniquement séquentielle : en théorie, la taille d'un journal est infinie, celui-ci croissant au fur et à mesure que des enregistrements lui sont ajoutés; comme les supports physiques ont une capacité de stockage finie, il faut archiver régulièrement les données les plus anciennes, sur bande magnétique par exemple. De plus, certaines applications comme les systèmes transactionnels utilisent la structure de journal pour stocker de manière rapide et fiable des informations temporaires, qui au bout d'un certain temps sont invalides et donc indésirables; il faut dans ce cas-là être capable de compacter le journal, c'est-à-dire de déterminer quelles sont les enregistrements qui correspondent à des données effectivement utilisables et de ne conserver que celles-ci, libérant ainsi de la place disponible pour l'utilisateur. Dans le cas du compactage comme dans celui de l'archivage se pose les problèmes de la disponibilité du journal au cours de la phase de mise à jour, et de la nécessité pour le gestionnaire du journal de déterminer quels enregistrements sont valides pour pouvoir les compacter. Ces problèmes, qui font l'objet de la seconde partie de ce rapport, représentent quelques-uns des points difficiles de l'implémentation d'un gestionnaire de journal.

1.3.2 Interface d'utilisation

Les enregistrements stockés sur le journal sont repérés par un numéro d'enregistrement ou « Log Sequence Number » (LSN), formant une suite strictement croissante d'entiers, attribués par le gestionnaire du journal au fur et à mesure que des enregistrements lui sont transmis pour écriture. Cette propriété permet à une application utilisatrice de retrouver les enregistrements dans l'ordre chronologique où ils ont été écrits. Une interface minimale de fonctions de manipulation d'un journal peut être par exemple

définie par les cinq appels suivants [Daniels et al. 87, Daniels 88] :

```
append   (record) -> lsn    : écrit un enregistrement en fin de journal.
read     (lsn)    -> record : lit l'enregistrement indiqué.
previous (lsn)    -> lsn    : renvoie le numéro de l'enregistrement précédent.
next     (lsn)    -> lsn    : renvoie le numéro de l'enregistrement suivant.
end-of-log        -> lsn    : renvoie le numéro du dernier enregistrement écrit.
```

Dans cet exemple simplifié, les commandes `append` et `read` donnent accès aux données stockées sur le journal; `next` et `previous` vont permettre de parcourir le journal pour une lecture avant ou une lecture arrière respectivement. L'appel `end_of_log` permet de retrouver la fin du journal.

1.3.3 Implémentation réelle

Une interface d'utilisation d'un journal doit permettre en outre, afin d'optimiser l'utilisation du support physique et les coûts de communication, de partager un même journal physique entre plusieurs journaux logiques entrelacés; ceci implique la possibilité de distinguer plusieurs sessions d'écriture à l'aide de commandes du type `open` et `close`. De même, les performances en lecture du système peuvent être améliorées grâce à l'utilisation de curseurs pour des lectures séquentielles, qui permettent d'anticiper l'accès aux données.

Par defaut, un application utilisera de écritures asynchrone. En effet, une écriture synchrone oblige l'application à attendre l'écriture effective des données sur le support physique, ce qui coûte cher, en particulier dans le cadre de systèmes transactionnels qui ont à effectuer plusieurs centaines d'écritures par seconde. Mais les applications doivent avoir la possibilité de demander explicitement qu'une écriture soit effectuée de manière synchrone, par exemple à l'aide de la primitives `flush` qui force l'écriture des tampons sur le support physique.

Les écritures synchrones posent un problème supplémentaire, en forçant potentiellement à écrire un tampon non plein, ce qui entraîne une fragmentation des données sur le support et donc une mauvaise utilisation de ce dernier. Ces problèmes peuvent être résolus en utilisant deux tampons utilisant des mémoires vives non volatiles, par exemple alimentées par une alimentation séparée [Banâtre et al. 91]. Lorsqu'un enregistrement doit être écrit, le contenu du tampon courant est recopié dans son jumeau, ce dernier devenant le nouveau tampon courant, et l'écriture est effectuée sur celui-ci; ainsi à tout moment l'état exact du tampon est connu. Cette technique permet d'utiliser des tampons de taille plus grande, par exemple de la taille d'une piste de disque, et augmente beaucoup les performances du système.

2 Utilisation des journaux

La notion de journal peut être utilisée dans le cadre de nombreuses applications, en particulier celles touchant à la sécurité et plus particulièrement à la sûreté de fonctionnement d'un système. La mise en place d'un protocole de validation de transactions à deux phases [Oki et al. 85] et le recouvrement [Haskin et al. 87, Kistler and Satyanarayanan 91] en sont des exemples. La journalisation est particulièrement bien adaptée aux traitements qui demandent le maintien d'un historique, et à la mise en œuvre du concept de transaction atomique.

2.1 Historiques

De par cette notion d'écriture séquentielle qui caractérise les journaux, ceux-ci sont naturellement utilisés pour conserver une succession chronologique d'actions jouées ou d'évènements subis par un système, ce que l'on nomme un historique.

2.1.1 Historique des accès

Une première utilisation possible d'un journal consiste à conserver l'historique des accès à un système, soit afin d'établir des statistiques, soit afin de détecter des comportements suspects de la part de certains utilisateurs. Ces comportements pourraient correspondre à des tentatives de violation de la sécurité du système. Par exemple, le système UNIX utilise le fichier `/usr/adm/messages` à la manière d'un journal.

2.1.2 Etat d'un système

Une autre utilisation classique de la journalisation consiste à sauvegarder l'état d'un système en enregistrant sur le journal, au fur et à mesure de son exécution, la succession des opérations qu'il effectue. On peut ainsi après une panne retrouver un état donné en rejouant toutes les actions qui se sont déroulées depuis un instant où l'état du système était parfaitement connu. Cet état peut être tout simplement celui du système après initialisation. Ce peut être aussi un état précis qui aura été sauvegardé, éventuellement lui aussi sur le journal, sous la forme d'un point de sauvegarde (checkpoint). Ceci permet de limiter la taille du journal et de diminuer le temps nécessaire pour rejouer les actions. Cette technique permet par exemple à un site tombé en panne de redémarrer et de retrouver un état cohérent et bien défini; on parle alors de reprise sur panne [Johnson et Zwaenepoel 88]. Elle permet aussi à une application de redémarrer après que les structures de données associées aient été modifiées; cette technique est couramment utilisée pour passer d'une version d'un système de gestion de fichiers à une autre version ultérieure incompatible. Elle peut être mise en œuvre à l'aide du concept de «transaction atomique», comme on le verra plus loin.

2.2 Applications spécifiques

Enfin certaines applications utilisent la notion de journal d'une manière tout-à-fait originale. Ainsi le système de fichiers Sprite LFS [Rosenblum et Ousterhout 91] enregistre toutes ses données sur disque sous forme d'une structure journalisée. Il tire ainsi parti de l'écriture séquentielle pour obtenir de très bonnes performances en écriture. Le surcoût engendré par la gestion du journal est compensé par le fait que les écritures sur disques représentent de plus en plus le goulet d'étranglement des systèmes de gestion de fichiers actuels, en particulier du fait de l'utilisation en lecture de caches de plus en plus grands.

Instant Replay [Leblanc et Mellor-Crummey 87] offre un mécanisme de deboguage de programme parallèle permettant de reproduire une exécution d'un programme parallèle. Les évènements caractéristiques d'une exécution sont sauvegardés pendant le déroulement du programme. Aucune donnée relative à ces événements n'est gardée. La journalisation est alors largement utilisée afin de prendre en compte la sérialisation des évènements.

2.3 Validation atomique de transactions

Nous allons nous intéresser ici à l'utilisation des journaux pour la réalisation des transactions atomique, et en particulier pour la mise en œuvre du protocole de validation à deux phases [Oki et al. 85, Baer et al. 81].

2.3.1 Notion de transaction

Syntaxiquement, une transaction est un morceau de code délimité par les instructions `Begin_Transaction` et `End_Transaction`. Elle contient une instruction `Abort_Transaction` en cas d'erreur et peut être interrompue par le système car la transaction peut accéder à des données permanentes et globales (cf figure II.1). Une transaction garantit quatre propriétés importantes dans le cadre de la sûreté de fonctionnement d'une application, à savoir la cohérence, l'atomicité face aux pannes, la sérialisabilité et la permanence. On retrouve parfois ces propriétés sous le sigle ACID : Atomicite, Cohérence, Indépendance (pour sérialisabilité) et Durabilité (pour permanence) [[Haerder et Reuter 83, Weihl 85, Papadimitriou 79, Ruffin 89] .

Certains auteurs étendent les propriétés des transactions à l'atomicité. L'ensemble des propriétés d'une transaction est alors appelé ACID (Atomicité, Cohérence, Indépendance, Durabilité).

2.3.2 Algorithme de validation à deux phases

La validation à deux phases (two-phase commitment) [Gray 78, Bernstein et Goodman 80, Baer et al. 81, Oki et al. 85] est un algorithme permettant d'effectuer la validation répartie d'une transaction distribuée sur différents sites malgré les différents risques de

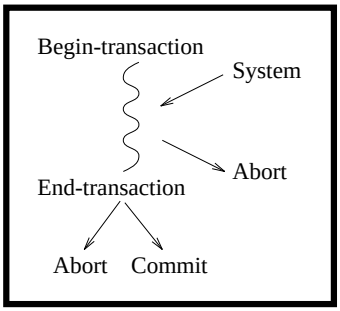

Figure II.1 : Déroulement d'une transaction

panne qui peuvent intervenir avant et en cours de validation. Cet algorithme permet donc d'assurer les trois propriétés citées précédemment. Il distingue un site maître, appelé le coordinateur, qui supervise le déroulement du protocole, et un certain nombre de sites esclaves, les participants, qui effectuent les mises-à-jour sur les données sous l'initiative du coordinateur. Ce protocole se décompose en deux phases distinctes consécutives, une première lors de laquelle chaque participant prépare les mises à jour à effectuer sur les objets de sa liste, et une seconde lors de laquelle ces mises à jour sont effectivement effectuées :

- Le maître envoie une requête de mise à jour, les esclaves préparent les nouvelles valeurs des données (images après) et renvoient au maître un message positif signifiant qu'ils sont près à valider.
- Lorsque tous les esclaves ont répondu positivement, le maître envoie l'ordre de valider (les images après remplacent les images avants). Si l'un des esclaves a répondu négativement, le maître envoie l'ordre d'abandonner à tous les esclaves.

Le but de cet algorithme est d'assurer une atomicité répartie d'une transaction.

Le maître et les esclaves disposent chacun d'un journal logique. Sur ce journal ils consignent les différentes étapes du protocole. Ainsi, pour le maître, les indications suivantes sont consignées (se reporter à la figure II.2) :

Début Le maître va émettre une requête R.

Succès Le maître à reçu une réponse positive (R+) de chacun de ses esclaves.

Echec Une réponse négative (R-) à été reçue.

Validation L'ordre de valider (C) à été envoyé à chacun.

Abandon L'ordre d'abandonner (A) à été envoyé à chacun.

De leur côté, les esclaves inscrivent sur leur journal les marques suivantes :

Début L'esclave est prêt à la réception d'une requête.

Dépendant L'esclave a recu une requête, il va alors préparer ses données et sa réponse au maître.

Validation L'ordre de valider à été recu, l'esclave à mis en place les nouvelles données.

Abandon L'ordre d'abandonner à été reçu.

Il est clair qu'en cas de panne d'un esclave ou du maître, il peut par lecture de son journal et par consultation des journaux des autres participants, connaître l'état d'avancement du protocole et en continuer l'exécution.

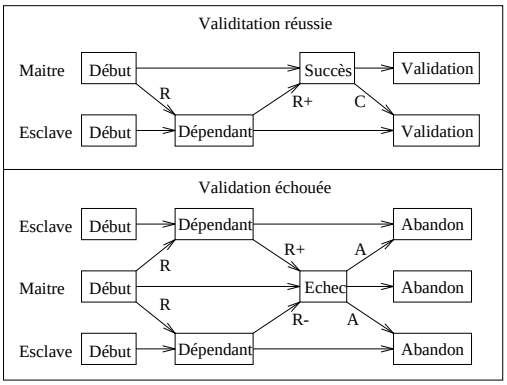

Figure II.2 : Journalisation lors de la validation à deux phases

3 Gestion physique du journal

Nous allons maintenant traiter trois points important de la journalisation : retrouver la fin du journal en cas de redémarrage ou de reprise sur panne, retrouver les données écrites sur le journal le plus rapidement possible et compacter le journal.

3.1 Trouver la fin du journal

Trouver la fin du journal est un problème traditionnel lors de la conception d'une méthode de journalisation. En fonctionnement normal, la fin du journal est toujours en mémoire, la retrouver n'est donc pas un problème. Par contre, en cas de redémmarage ou de reprise sur panne, retrouver la fin du journal est moins simple. En effet, le repérage de la fin du journal ne peut être écrit à un endroit fixe du support de stockage car cela signifierait qu'à chaque fois que des données sont concaténées en fin de journal, une seconde écriture serait nécessaire pour repérer la nouvelle fin du journal. Ce qui dégraderait notoirement la vitesse d'écriture sur le support de stockage, chaque écriture nécessitant deux déplacement de la tête d'écriture. Des supports comme la bande, le disque optique numérique ne s'y prêtent pas.

D'autre part, écrire des informations indispensables à la compréhension du journal physique à un endroit fixe d'un support de stockage implique en cas de dommage de cette partie fixe, la perte de la totalité du journal physique où au moins d'une partie non endommagée du journal. On ne peut donc écrire où se trouve la fin du journal à un endroit fixe d'un support de stockage indépendant.

Dans le cas d'un redémarrage, la fin du journal doit être exacte puisque l'application a été quittée proprement. En revanche, en cas de reprise sur panne, certaines applications ne garantissent qu'une fin de journal approximative mais normallement un état des données cohérents.

3.1.1 Espace non utilisé rempli de zéro

La solution traditionnelle consiste à initialiser le support de stockage en le remplissant de zéros [Daniels et al. 87, Daniels 88, Haskin et al. 87, Finlayson et Cheriton 87], et à la fin de chaque compactage de réinitialiser à zéro la partie nouvellement libérée. Par une recherche binaire, on trouve ensuite la fin du journal en cherchant le dernier secteur du support non rempli de zéros. Cette méthode a pour principaux inconvénients : la nécessité d'initialiser les supports de stockage, et un allongement du temps de compactage. De plus, la présence d'eventuel enregistrements remplis de zéros complexifient l'algorithme de recherche.

3.1.2 Point de sauvegarde à un endroit fixe du support

Le système de fichier Sprite LFS [Douglis and Ousterhout 89], sauvegarde la fin du journal dans une structure (« checkpoint ») qui se trouve à un endroit fixe du support. Cette sauvegarde n'est pas faite à chaque écriture, en cas de panne le système de fichiers peut donner une fin de journal qui est approximative. Cette structure étant critique (c'est à partir de cette structure que le journal peut être réutilisé), deux exemplaires sont mis à jour en alternance. Cette structure contient, entres autre, la fin et la dernière date de modification du journal. La date de dernière modification est mise à jour en dernier dans cette structure, donc si il y a panne pendant l'écriture du journal, la dernière écriture du journal qui est incomplète ne sera pas prise en compte. Lors du redémmarage ou de reprise sur panne, le système lit les deux structures et utilise la plus récente.

3.1.3 Marqueur de fin de secteur

$K_I{}^T L^O{}_G$ [Ruffin 92b] utilise un marqueur de fin de secteur pour retrouver la fin du journal. Le marqueur est constitué de la date de sa création, du numéro du site auquel il est relié et d'un compteur de compactage. L'ensemble des marqueurs d'un journal est suffisant pour retrouver la fin du journal exact. Mais afin de limiter le temps de recherche de la fin du journal, à intervalles réguliers (et en fin de compactage) le marqueur du dernier secteur du journal physique et le numéro de ce secteur sont écrits à une position fixe du support de stockage. Cela permet de réduire le champ de recherche, mais aussi de connaître la valeur du compteur de compactage courant.

3.2 Retrouver les données

Les données écrites sur un journal sont lues de différentes façon en fonction du type de service que rend l'application.

3.2.1 Lecture avant, lecture arrière

Beaucoup d'application utilisant un journal se contentent d'une simple lecture avant ou arrière d'un journal logique ou physique.

Une lecture avant d'un journal est typiquement utilisée lors d'une reprise sur panne. Elle se fait en deux etapes : (i) déterminer le point de départ (par exemple, dans le cas d'une reprise sur panne d'un gestionnaire de transaction [Daniels et al. 87, Daniels 88], le point de départ sera le début des transactions qui n'ont pas été terminées à cause de la panne), (ii) lire le journal, à partir de ce point de départ jusqu'à la fin du journal.

La lecture arrière d'un journal se fait à partir de la fin du journal en le lisant vers du début. Ce type de lecture est, par exemple, utilisé dans la commande `UNDO` de l'editeur Emacs, puisque cette commande défait des actions à partir de la fin du journal.

3.2.2 Structures de contrôles journalisées

Pour des applications qui lisent le journal de façon aléatoire des structures de contrôles (meta-data) permettent de connaitre directement la localisation de données recherchées. Cela évite une lecture totale du journal. Ces structures de contrôles forment des indirections qui permettent de retrouver les données recherchées.

Sprite LFS [Rosenblum et Ousterhout 91] utilise de nombreuses structures de contrôles pour retrouver les données des fichiers. Ces structures sont journalisées sur le même journal que les données d'un fichier. Prenons l'exemple de deux structures de contrôles : les `inodes` et les `inode-map` (pour plus de détail voir III.3).

Il existe un `inode` pour chaque fichier. Cette structure contient des attributs du fichier tels que son type, son propriétaire, ses droits d'accès, les adresses des blocs du fichier.

La structure `inode-map` permet de connaître la position de chaque `inode`. A partir d'un numéro d'`inode`, l'`inode-map` détermine la position de l'inode dans le journal par simple indexation. L'`inode-map` est divisé en morceaux qui sont écrits sur le journal. En effet, un morceau de l'`inode-map` correspond à un intervalle de 30 secondes pendant lesquelles des inodes ont été créés ou modifiés.

Un exemple de configuration est donné dans la figure II.3. Cette figure représente deux morceaux de l'`inode-map` qui pointent tous les deux sur deux `inodes` (différentes) qui référencent respectivement 1, 1, 1 et 2 blocs de données.

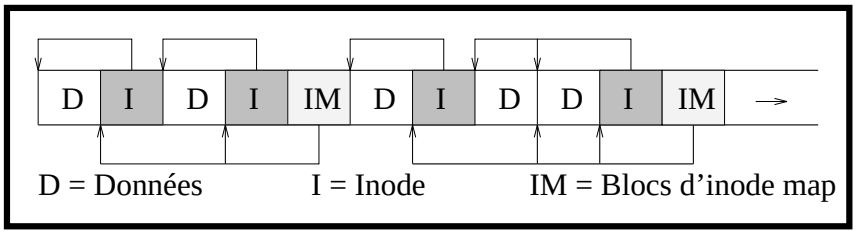

Figure II.3 : Structures d'inodes et d'inode-map dans LFS, système de fichier de Sprite

3.2.3 Arborescence

Dans Clio [Finlayson et al. 87] une arborescence est utilisée pour retrouver tous les enregistrements d'un même journal logique car tous les enregistrements des journaux logiques sont multipléxés sur un même support physique. Il existe un arbre par journal logique.

Une feuille de l'arbre décrit un ensemble de secteurs. Elle contient un nombre binaire. Chaque chiffre, de ce nombre binaire, décrit un secteur en terme de présence ou d'abscence de au moins un enregistrement du journal logique recherché. Un exemple est donné dans la figure II.4. L'arbre de cette figure nous indique que cinq secteurs du journal physique contiennent des enregistrements du journal logique recherché.

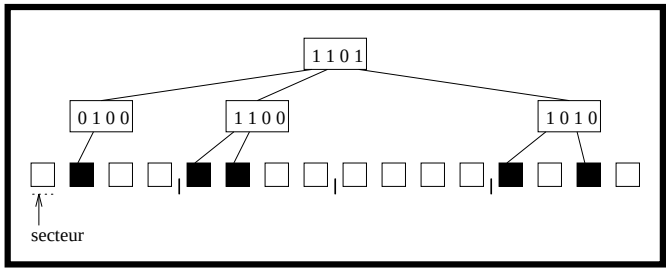

Figure II.4 : Recherche des enregistrements d'un journal logique sur un journal physique

L'idée de l'utilisation d'un arbre peut être étendue à d'autre type de recherche [Daniels et al. 87, Daniels 88, Ruffin 92b]. Par exemple sur une recherche de numéro d'enregistrement. En effet, dans certaine application les enregistrements constituant un journal sont de tailles variables. La position d'un enregistrement n'est donc pas calculable en fonction de la taille d'un enregistrement (de plus il faut être sûr que les numéros d'enregistrement soient augmentés à chaque fois du même incrément).

3.3 Le compactage

Un problème important lié au journal est le besoin d'avoir toujours de l'espace disque libre à la fin du journal pour continuer à écrire. Il faut donc gérer les espaces libérés par des enregistrements pour qu'ils puissent être réutilisés. Ces enregistrements obsolètes se trouvent à l'intérieur du journal. L'espace libéré ne peut donc pas être utilisé pour l'écriture, puisque par définition, les écritures sur un journal se font toujours à la fin. Le compactage consiste à retirer les enregistrements obsolètes en déplacant les enregistrements valides vers le début du journal. Ce déplacement ce fait en gardant l'ordre d'écriture. Le compactage sert à limiter la fragmentation du journal et à avoir le maximum de place à la fin du journal pour écrire.

Le compactage peut aussi avoir une autre forme qui s'apparente plus à de l'archivage. Par exemple, pour un journal stocké sur un support non réinscriptible (par exemple, le disque optique non réinscriptible), lorsque le support est plein, il est nécessaire d'en extraire certaine données pour les recopier sur le nouveau disque. Cela permet de lier les données et de passer d'un disque à l'autre en lecture. Le gestionnaire du journal à la possibilité de passer d'un disque à l'autre de façon cohérente. Les différentes étapes de ce procédé font appel aux même étapes que le compactage.

3.3.1 Quand compacter

Le compactage d'un journal ne se fait pas en continu mais de temps en temps. La fréquence est déterminée par des facteurs qui dépendent des applications. Cependant toutes les applications sont évidemment d'accord de compacter dès que le support est plein ou tend à l'être. Mais pour éviter cette situation peu confortable, deux options peuvent être prises :

- Compacter à intervalle régulier. Le problème est de choisir le bon intervalle. L'idéal est de le choisir un moment au l'activité est faible, typiquement pendant la nuit. Mais, par exemple, pour un journal de système de fichier cette fréquence de compactage est trop faible.

- Compacter lorsqu'il y a peu d'activité. Cette option est tout aussi délicate que la première car la fréquence d'utilisation du journal est rarement prévisible dans le temps. Donc, même si le compactage commence dans une phase de faible activité, rien ne permet d'affirmer que cette faible activité va être constante pendant toute la durée du compactage.

De plus, ces deux options peuvent être utilisées alternativement.

La plupart des applications interdisent la journalisation pendant le compactage. Une des exeption à cette règle est $K_I{}^T L^O{}_G$ [Ruffin 92b] qui permet de continuer la journalisation mais avec des performence très diminuées.

3.3.2 Comment trouver les données non valides

Le journal n'est pas compacté à chaque invalidation d'enregistrements mais suivant différentes politiques. De ce fait, un enregistrement invalide doit être connu ou retrouvé comme tel lors du déclanchement du compactage. Pour être retrouvé il peut être soit stocké (par exemple dans un journal logique) soit déductible à partir de données valides du journal.

3.3.2.1 Journal logique des enregistrements invalides. Grâce à l'utilisation d'un journal logique $K_I^T L_G^O$ [Ruffin 92a, Ruffin92b] stocke les numéros d'enregistrements invalides. Ce sont les applications, qui utilisent $K_I^T L_G^O$, qui déterminent quels sont les enregistrements dont elle n'a plus besoin. En effet, les applications ont une fonction d'invalidation d'enregistrements qui constitue une partie de l'interface entre l'application et $K_I^T L_G^O$. A la réception de cette fonction, $K_I^T L_G^O$ stocke dans un journal logique les numéros d'enregistrements passés en paramètre (ces numéros d'enregistrement sont des numéros d'enregistrement invalides). Lorsque le compactage est déclenché, $K_I^T L_G^O$ enlève tous les enregistrements qui ont été déclaré invalide par l'application à partir du journal logique de numéros d'enregistrements invalides.

3.3.2.2 Validation de données par rapport aux données de contrôle. Le système de fichier LFS [Rosenblum et Ousterhout 91] n'utilise qu'un seul journal. Il n'est donc pas possible de sauvegarder dans un journal logique particulier les enregistrements obsolètes.

Pour retrouver les enregistrements invalides, LFS utilise les données de contrôles qu'il journalise avec les données des fichiers. A partir de numéro de version du fichier, il arrive à déterminer si un bloc est obsolète.

3.3.3 Comment compacter

Différentes techniques de compactage sont utilisées. Leur utilisation est avant tout fonction de la nature de l'application. En effet, des critères tels que le taux moyen de données obsolètes, la longeur du journal, les critères de choix d'une structure de journal (tel que la sériabilité, l'efficacité, la fiabilité) ont une influence sur les techniques de compactages.

3.3.3.1 Copier les données aux début du journal. Certaine application n'ont pas de données obsolètes. Par exemple, l'editeur Emacs utilise un journal pour mettre en place la commande `UNDO`. Cette commande permet à l'utilisateur de défaire ce qu'il vient de faire. L'utilisateur pouvant défaire ce qu'il a fait depuis la création du journal (à condition que la taille du support le permette), toutes les données doivent être gardées. Si la taille du support n'est pas suffisante pour sauvegarder toutes les modifications de l'utilisateur, le support peut être considéré comme un support circulaire. C'est-à-dire que lorsqu'il n'y a plus de place, les nouvelles modifications écrasent les premieres...

3.3.3.1.1 Compactage par vague. Cette méthode de compactage divise le journal en `N` parties. Une partie du journal va être traitée en une vague et de façon indépendante par rapport au autres. Pour chaque vague du journal, on détermine qu'elles sont les données obsolètes à partir des données qu'elle contient. Une fois que les données valides sont repérées elles sont recopiées au début du journal.

Cette méthode présente l'avantage d'avoir besoin d'un intervalle de temps court pour compacter. En effet, le journal peut ne pas être compacté dans sa totalité en une seule fois mais en plusieurs étapes. En contre partie, cette méthode considère des données valides alors qu'elles peuvent être invalidées par les données des vagues suivantes.

Le choix de la longeur d'une vague est un compromis à trouver entre le temps minimum de compactage et l'invalidité des données. En effet, si les vagues sont longues, beaucoup de données pourront être invalidées mais le temps de compactage minimum est important alors que si les vagues sont petites, le temps de compactage minimum est petit mais moins de données pourront être invalidées.

Une heuristique peut compléter cette technique: ne pas toujours compacter par le début du journal si il a déjà été compacté plusieurs fois. En effet, dans le cas d'un système de fichiers, par exemple, un fichier a souvent une durée de vie soit très courte (il serat rammassé pendant les première vagues de compactage), soit très longue (il est inutile de chercher à le récupérer à chaque compactage)

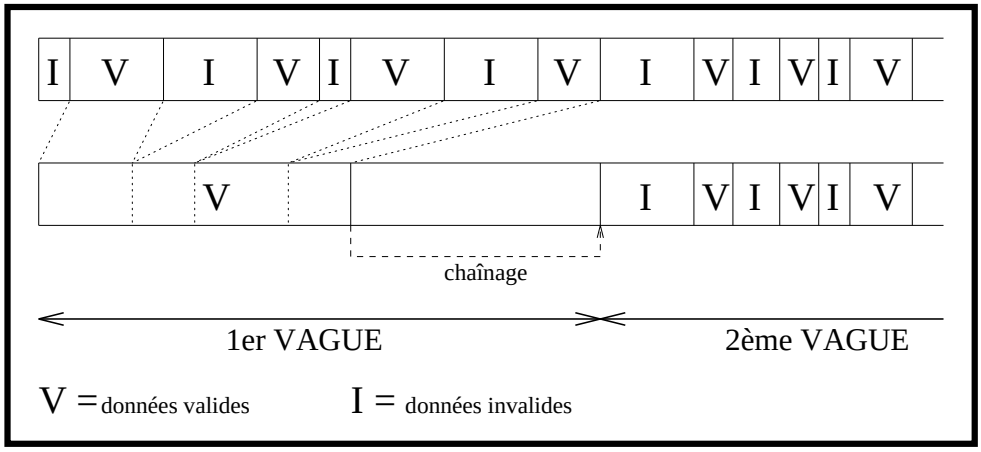

Figure II.5 : Compactage par vagues

La figure II.5 montre la réalisation de la première vague : toutes les données valides sont écrites au début du journal. Les autres vagues se font sur le même principe. Les données valides de la deuxième vague seront écrite à la suite des données de la première vague.

3.3.3.1.2 Double lecture : le compactage en deux phases. Cette méthode de compactage va compacter le journal en une seule fois. Deux lectures complètes du journal sont effectuées. La première lecture permet de déterminer l'ensemble des

enregistrements non valides. La deuxième lecture permet de recopier l'ensemble des données valides au début du journal.

Cette méthode est très coûteuse en temps car deux lectures complètes du journal sont nécessaires. Cependant, elle n'est pas toujours à écarter car la lecture complète du journal peut être nécessaire pour retrouver l'ensemble des données invalides

3.3.3.1.3 Ne pas écraser les données avant de les avoir réécrites. Quelle que soit la méthode de compactage, il ne faut pas écraser les données avant de les avoir réécrite sinon en cas de panne le journal se retrouve dans un état incohérent. La figure II.6 propose une solution utilisée par KITLOG [Ruffin 92b]. Cette solution se déroule en deux étapes :

- Ecrire les données en mémoire et à la fin du journal si il n'y a pas assez de place dans la zone de compactage,
- Ecrire les données au début du journal, puis invalider ce qui à été écrit à la fin du journal.

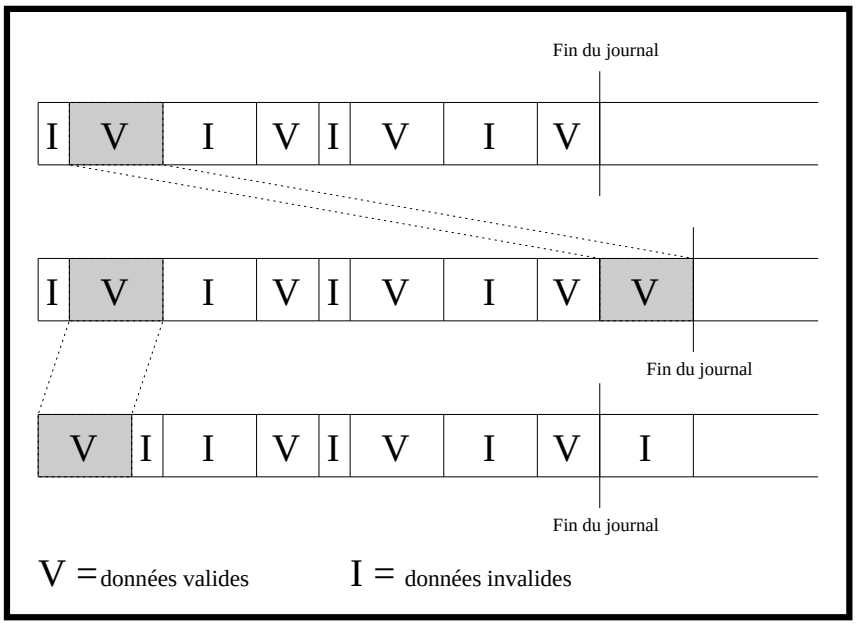

Figure II.6 : Compactage de données en début de journal

3.3.3.2 Utiliser un nouveau support. Une autre méthode de compactage peut être l'utilisation d'un autre support. Dans ce cas là, lorsqu'un support est plein, l'ensemble des données valides de ce support sont recopiées sur le support vide. Cette méthode semble moins coûteuse en ce qui concerne le compactage. Cependant elle est chère financièrement puisqu'il faut doubler le nombre de supports (cela s'ajoute à la grande consommation de place que demande un journal).

3.3.3.3 Regrouper les données en paquets et compacter les paquets. Sprite LFS [Douglis and Ousterhout 89] utilise un seul journal comme structure de stockage. Le support est découpé en sections de taille fixe (en général, de 512 Kilo-octets ou 1 Méga-octets) appelées segments. Le journal est formé de segments qui ne sont pas forcément contigüs. Des structures, sur le journal, au début et en fin de segment, permettent de lier les segments pour former le journal. Cette décomposition en segments du journal permet de mettre en place une politique efficace de compactage. En effet, lors du compactage seul les segments peu pleins sont compactés, ce qui permet de récupérer un maximum de place tout en déplacant un minimum de données.

La figure II.7 donne un exemple de compactage de trois segments. Le premier et le deuxième segment sont peu pleins, ils sont regroupés dans le premier segment. Le troisième segment est composé de beaucoup de données valides, il n'est pas compacté. Au fur et à mesure du compactage, l'ordre des segments composant le journal a changé.

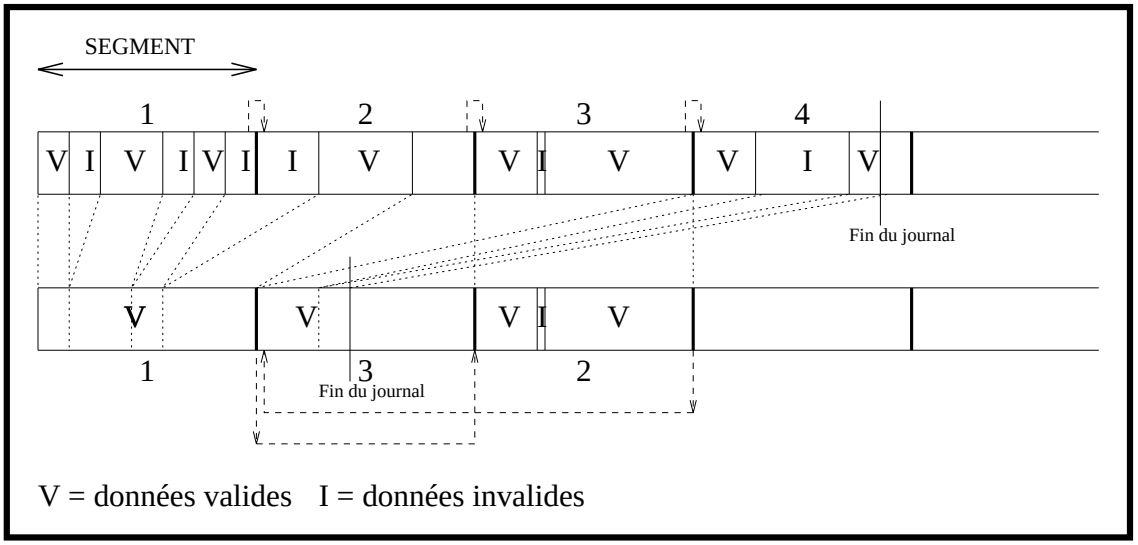

Figure II.7 : Compactage dans LFS, système de fichier de Sprite

4 Conclusion

Le concept de journal est simple et efficace. Dans sa mise en œuvre, la journalisation est complexe mais aide à règler de nombreux problèmes tels que la cohérence, la durabilité et l'indépendance.

Un nombre croissant d'applications ont besoin de journaux pour des raisons bien distinctes et disposant de caratéristiques spécifiques. Par exemple, Camelot [Daniels et al. 87, Daniels 88] et QuickSilver [Haskin et al. 87] disposent d'un journal pour les transactions et la reprise après panne. Argus [Oki et al. 85] base son stockage fiable d'objets permanents sur un journal fiable. Un éditeur de textes, tel qu'Emacs, utilise un journal en mémoire afin de défaire de manière intéractive (ou de refaire) des modifications aux fichiers édités. Instant Replay [Leblanc et Mellor-Crummey 87] a

besoin d'un journal dans le but de rejouer des sessions de débogage. Isis [Isis 91] base la fiabilité de ses communications sur la journalisation des messages envoyés. Sprite [Rosenblum et Ousterhout 91] assure le stockage de tout son système de fichiers sous la forme d'un journal.

L'ensemble de ces applications n'ont pas été référencé dans cette synthèse, qui ne se veut pas être un catalogue des applications exitentes mais qui veut donner un aperçu des différentes techniques utilisées. De plus, de nombreuses techniques sont reprises d'une application à une autre.

La journalisation étant de plus en plus utilisée, la recherche se tourne vers des outils de journalisation générique comme $\mathrm{KI^{T}LO_{G}}$. Cet outil ne répond pas encore à toutes les demandes des applications, mais est sur la bonne voie.

Chapitre III

KFS : $K_I^T L^O_G$ File System

1 Introduction

Ce chapitre concerne la réalisation d'un système de fichiers appelé KFS. Il s'appuie sur l'outil de journalisation $K_I^T L^O_G$ [Ruffin 92b] et est construit sur le modèle du système de fichiers de Sprite : Log-Structured File System (LFS) [Rosenblum and Ousterhout 91].

Le plan de ce chapitre est organisé comme suit :

Dans un premier temps, une description rapide de l'outil $K_I^T L^O_G$ et du système de fichiers LFS est effectuée.

Dans un deuxième temps, les différentes caractéristique de KFS sont exposées.

Pour finir, les interfaces utilisées et fournies par KFS sont explicités.

2 Description de $K_I^T L^O_G$

$K_I^T L^O_G$ est un outil générique de journalisation. Il offre la possibilité de gérer un ensemble de journaux logiques sur un ou plusieurs supports physiques. Dans un premier temps l'utilisateur construit un graphe de composition qui définit les journaux et leurs environnements. Dans un deuxième temps, l'utilisateur dispose d'une interface qui lui permet de les manipuler.

2.1 Graphe de composition

Le modèle, utilisé par $K_I^T L^O_G$, découpe les différentes caractéristiques des journaux en cinq sortes d'objets :

- la gestion de la mémoire,

- ❒ la distribution du journal,
- ❒ la réplication des enregistrements,
- ❒ le partage des journaux,
- ❒ la gestion des journaux sur les ressources physiques.

Ces caractéristiques sont encapsulées dans des classes d'objets. Le programmeur d'application compose les objets de ces classes en un graphe orienté acyclique.

A travers un fichier de configuration l'utilisateur va définir ce graphe de composition pour chaque type de journal. Ce fichier de configuration permet de préciser à $K_I{}^T L^O{}_G$:

- ❒ les éléments qui composent le graphe,
- ❒ les liens qui relient les différents éléments du graphe,
- ❒ les machines sur lesquels sont localisées les éléments constituant le graphe,
- ❒ les supports.

2.2 Interface de $K_I{}^T L^O{}_G$

Lorsque le ou les graphes de compositions sont définits, des fonctions de création et de destructions de journaux sont utilisables. L'interface présentée en Figure 1 permet de manipuler les journaux après les avoir ouvert.

Ecriture d'enregistrements	`put (Lid, Rid, D, S)` `put (Lid, &Rid, D, S)`
Signalisation au système de la liste des enregistrements obsolètes	`invalidate_records (...)`
Lectures en arrière	`get_last (Lid, &Rid, &D, &S)` `get_prev (Lid, &Rid, &D, &S)`
Lectures en avant	`get (Lid, &Rid, &D, &S)` `get_next (Lid, &Rid, &D, &S)`
Action de forcer les écritures sur les supports de stockage	`flush ()`

Figure III.1 : Exemple d'interface d'un journal

`Lid`: Identificateur de journal (log identifier), `Rid`: Identificateur d'enregistrement (record identifier), `D`: Bloc opaque de données de l'enregistrement (Data), `S`: Taille de l'enregistrement (Size), `&`: Passage de paramètre par référence. En plus de l'écriture, la première opération `put()` permet aux applications de choisir la valeur de leur `Rid`; alors que la seconde, demande au système de s'en charger. L'opération `invalidate_record()` possède plusieurs formes correspondant à différentes méthodes de désignation des enregistrements effaçables. L'opération `get()` appelée avec une valeur de `Rid` nulle, retourne le premier enregistrement du journal.

3 Présentation de LFS

LFS est un sytème de fichiers qui utilise que la journalisation : toutes les modifications sont écrites séquentiellement sur le disque. Il existe donc un seul journal logique qui coïncide donc avec le journal physique. Le journal est constitué d'enregistrements de tailles fixes : le bloc (512 octets). Des structures de contrôles sont journalisées sur le même journal afin de relire les données de façon efficace.

3.1 Description des structures de LFS

LFS utilise différentes structures de données :

- **inode** : structure de contrôle pour la lecture de données.

  Cette structure est la structure de base de LFS. Elle est identique à celle utilisée par le sytème de fichiers d'UNIX. Il existe un `inode` pour chaque fichier. Cette structure contient les attributs d'un fichier : son type, son propriétaire, ses droits d'accès... et les adresses des dix premier enregistrements du fichier. Pour les fichiers supérieurs à dix enregistrements, l'`inode` contient l'adresse disque d'un autre enregistrement appelé **indirect-block**. Un `indirect-block` contient à son tour l'adresse d'enregistrements de données et éventuellement l'adresse d'autre `indirect-block`. Les `inodes` ne sont pas écrits à un endroit fixe, comme sous UNIX, mais écrits sur le journal.

- `inode-map` : structure de contrôle pour la lecture de données.

  Cette structure permet de connaître la position de chaque `inode` sur le journal. A partir d'un numéro `d'inode`, l'`inode-map` détermine la position de son `inode` par simple indexation. L'`inode-map` est divisé en morceaux (ayant la même taille que tous les autres enregistrements, c'est-à-dire celle d'un bloc) qui sont écrits sur le journal. Un morceau de l'`inode-map` contient les numéros des `inodes` qui ont été créés ou modifiés dans un intervalle de 30 secondes. Cette technique s'inscrit dans une politique de cache pour augmenter les performances du système de fichiers : les modifications du système de fichiers, qui sont stockées dans le cache, sont écrites toutes les 30 secondes sur le disque. En effet, de nombreuses études ont montré que beaucoup de fichiers avaient une durée de vie très courte [Mary G. Baker and al. 91]. Dans ce cas, ils n'ont pas besoin d'être écrit sur support stable, leur maintient en mémoire est suffisant. En cas de reprise sur panne, le système de fichiers pert au maximum 30 secondes de fonctionnement.

- `checkpoint` : structure de contrôle pour la reprise sur panne et la lecture de données.

  C'est une structure qui se trouve à un endroit fixe du support. Elle contient :

  - L'adresse de tous les morceaux de l'`inode-map`.
  - La fin exacte (en cas d'arrêt « propre ») ou la fin approximative (en cas de reprise sur panne) du journal.

- ☆ La structure `segment-usage-table` (cette structure est définie plus loin).
- ☆ La date de la dernière écriture du journal. Cela est utile lors de la reprise sur panne.

Le `checkpoint` est une structure critique car c'est à partir de cette structure que va se faire la reprise sur panne. De ce fait, deux `checkpoints` sont mis à jour en alternance. La date de modification du `checkpoint` est mise à jour en dernier, donc si il y a panne pendant la mise à jour du `checkpoint` cette dernière mise à jour incomplète ne sera pas prise en compte. Lors de la reprise sur panne, le système lit les deux `checkpoints` et utilise le plus récent.

- ❒ **segment** : découpage du journal en sections de taille fixe (en générale de 512 Kilo-octets ou 1 Mega-octets) afin de mettre en place une politique de nettoyage efficace. Si l'on prend un `segment` de 512 Kilo-octets, il contient 1 000 enregistrements, chaque enregistrement étant un bloc de 512 octets.
- ❒ `segment-summary-block` : structure de contrôle décrivant un `segment`.

  Cette structure identifie toutes informations écrites dans un `segment`. Par exemple, pour chaque enregistrement de données de fichiers, la structure `segment-summary-block` contient son numéro d'`inode` et l'adresse de l'enregistrement de ce fichier. Cette structure est utile aussi bien pour la reprise sur panne que pour le compactage (elle permet de déterminer les données invalides). Elle est écrite au début des `segments` qu'elle décrit.
- ❒ `segment-usage-table` : structure de contrôle décrivant l'ensemble des `segments`.

  Pour chaque `segment`, cette structure maintient le nombre d'octets valides et la date de modification des enregistrements. Elle sert lors du compactage à choisir les `segments` qu'il est le plus avantageux de compacter : les `segments` les moins pleins sont compactés, ce qui permet de récupérer un maximum de place tout en déplacant un minimum de données. Les `segments` composant le journal ne sont pas contiguës physiquement mais reliés entre eux pour former le journal. La `segment-usage-table` sert à lier ces `segments` entre eux car elle contient l'adresse du `segment` suivant. Cette structure ce trouve dans le `checkpoint`.

3.2 Organisation des différentes structures de LFS

Le support est divisé en `segments`. Les `segments` sont reliés grâce à la structure `segment-usage-table` pour former le journal. La structure `segment-summary-block` est écrite au début du `segment` qu'elle décrit.

Dans LFS, l'`inode-map` est une structure qui permet de connaître la localisation de chaque `inode` dans le journal. A partir d'un numéro d'identification d'un fichier, l'`inode-map` donne la localisation de l'`inode`. L'`inode-map` est une structure qui est répartie en plusieurs morceaux sur le journal. Chaque morceaux correspond à 30 secondes de création ou de modification d'`inode`. La structure `checkpoint` permet de retrouver l'ensemble des morceaux de l'`inode-map`.

La figure III.2 montre un exemple de configuration qui représente un système de fichiers composé de cinq fichiers. Ils sont stockées sur trois `segments` qui ne sont pas contigüs physiquement puisque le journal commence sur le premier `segment` puis continu sur le troisième et enfin sur le deuxième. La structure `segment-summary-block` (ssb) permet, à partir de la fin d'un `segment`, de savoir sur quel autre `segment` continu le journal. Les fichiers ont été créés ou modifiés pendant trois tranches de 30 secondes puisqu'il y a trois morceaux d'`inode-map` (`IM1`, `IM1` et `IM2` - les deux premiers `inode-map` sont les mêmes car, comme nous allons le voir, le deuxième est une nouvelle version du premier -). Nous allons maintenant décrire les cinq fichiers :

- Un fichier d'`inodes I1` qui a trois enregistrements de données : `D1`, `D2` et `D3`. L'adresse de `I1` est référencée par `IM1`. `D1`, `D2`, `D3` et `I1` sont dans le premier `segment`. `IM1` est dans le deuxième `segment` logique et dans le troisième `segment` physique.

- Un fichier d'`inodes I2` qui a deux enregistrements de données : `D4` et `D5`. L'adresse de `I2` est référencée par `IM1`. `D4` est dans le premier `segment` et `D5`, `I2` et IM1 dans le deuxième `segment` logique et dans le troisième `segment` physique. Le deuxième enregistrement de ce fichier a été modifié depuis sa création. Dans le premier `segment`, trois enregistrements de sa première version ont été invalidés (`D5`, `I2` et `IM1`) puisqu'ils ont été modifiés. Ces trois enregistrements sont donc réécrit à la fin du journal.

- Un fichier d'`inodes I3` qui a deux enregistrements de données : `D6` et `D7`. L'adresse de `I3` est référencée par `IM1`. Ce fichier a été créé dans la même tranche de temps que la modification du fichier d'`inode I2` et que la création du fichier d'`inode I4`.

- Un fichier d'`inodes I4` qui a trois enregistrements de données : `D8`, `D9` et `D10`. L'adresse de `I4` est référencée par `IM1`.

- Un fichier d'`inodes I5` qui a un enregistrement de données : `D11`. L'adresse de `I5` est référencée par `IM2`.

Deux `checkpoints` sont représentés. `Checkpoint1` est le dernier `checkpoint` mis à jour.

3.3 Utilisation du modèle de LFS

Les structures `checkpoint`, `segment`, `segment-summary-block` et `segment-usage--table` sont mises en place dans LFS pour assurer la reprise sur panne et le compactage. Or, ces deux concepts sont gérés par $K_I{}^T L^O{}_G$, ils n'ont donc pas besoin d'être implémentés par KFS.

Dans KFS la structure de `checkpoint` est abandonnée. En effet, dans LFS, la structure `checkpoint` est une structure qui se trouve à un endroit fixe du disque. Il n'est pas souhaitable de garder cette caractéristique dans KFS car cela voudrait dire qu'une ressource de stockage fiable ne servirait qu'à cela. En effet, $K_I{}^T L^O{}_G$ gère l'ensemble d'une ressource, il n'est pas possible d'écrire sans journaliser et donc d'écrire à un

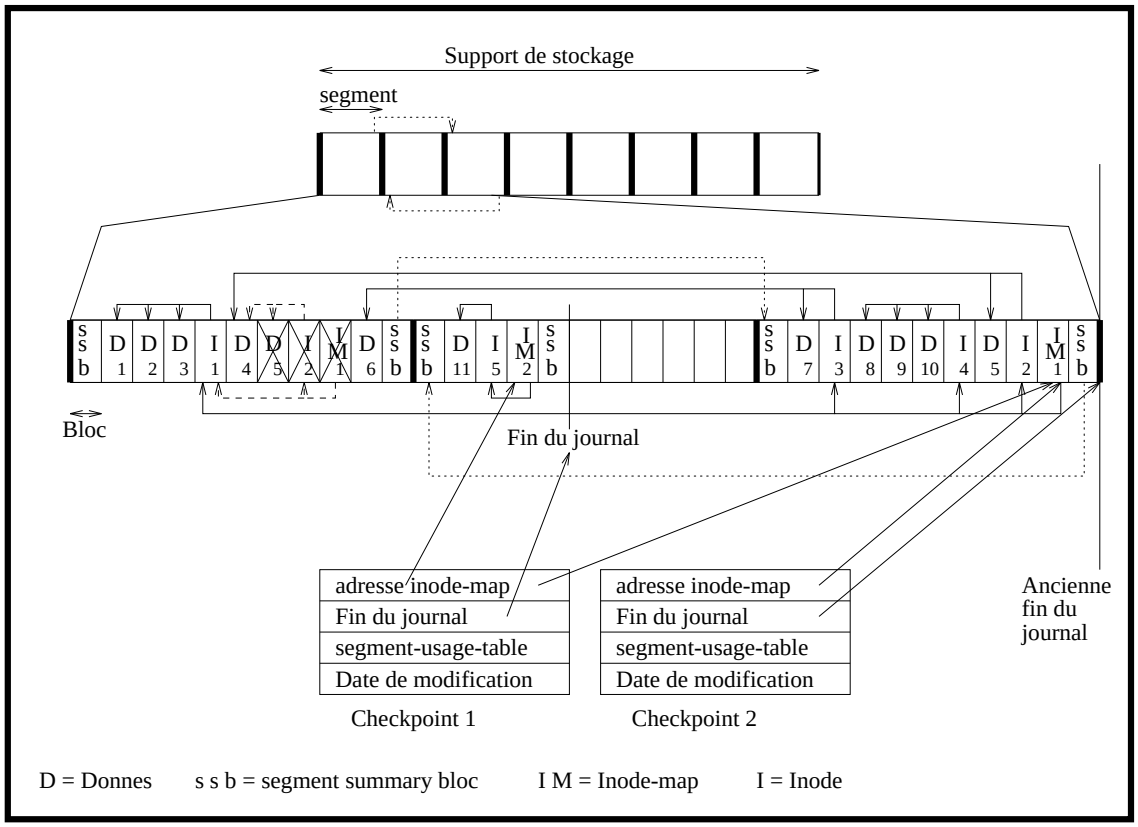

Figure III.2 : Structures d'inodes et d'inode-map dans LFS, système de fichier de Sprite

endroit fixe. Une alternative serait d'écrire sur un autre support sans passer par KITLOG mais cela n'est pas souhaitable pour des raisons de coût et de politique car KITLOG doit être suffisant.

La structure `checkpoint` sert à repérer la fin du journal en fonctionnement normal et à l'approximer en cas de panne. Or le problème de la fin du journal (qu'elle soit approximative, en cas de panne ou exacte, en cas de redémarrage) est géré par KITLOG.

Quant à la structure d'`inode-map`, elle est nécessaire pour des raisons d'efficacités dans LFS. En effet, LFS dispose d'un seul journal. Les `inodes` et les données des fichiers se trouvent donc sur le même journal. Pour pouvoir retrouver un `inode` plus facilement, leurs adresses sont regroupées dans des morceaux d'`inode-map`. C'est une sorte d'indirection qui permet de retrouver les `inodes` plus rapidement.

Avec un seul journal il est difficile de faire la différence entre des structures de contrôles et des données. Or, avec plusieurs journaux, les différents types de données peuvent être regoupés dans des journaux logiques différents. C'est cette propriété, qui est fournie par KITLOG qui va nous permettre de supprimer la structure d'`inode-map`.

4 Description de KFS

Après avoir décrit le graphe de composition et la structure d'`inode` utilisés par KFS, l'organisation du système de fichiers est proposée.

4.1 Graphe de composition de KFS

KFS est un sytème de fichier utilisable par un seul utilisateur sur un seul disque centralisé. Cette solution a été choisie pour, dans un premier temps, avoir la configuration la plus simple possible. La figure III.3 représente le graphe des journaux utilisés.

Ce graphe est valable pour tous les journaux logiques qui définissent le système de fichiers. Il est définit par KFS pour KITLOG à travers un fichier de configuration.

4.2 L'inode

L'`inode` décrit un fichier. Il permet de connaître les attributs et la localisation des données d'un fichier. Ce fichier peut être un fichier de données ou un répertoire.

La structure `inode` a la forme suivante :

```
#define NB_RECORD 10              /* Identique à UNIX */
typedef uid {
        ...                       /* Type définit par KITLOG comme étant
                                     un identificateur unique d'enregistrement */
}
typedef time_t {
```

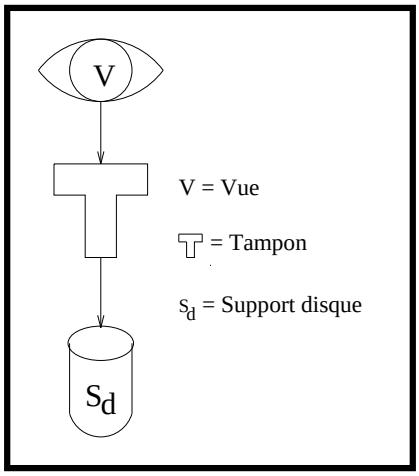

Figure III.3 : Arbre de composition

```
        ...                         /* Type définit par UNIX pour manipuler des dates */
}
typedef Inode {
        int num_inode;              /* Numéro de l'inode */
        enum type {file, dir};      /* Type du fichier que l'inode référence */
        uid num_record[NB_ENREG]; /* NB_ENREG premier enregistrement de données
                                     du fichier */
        uid num_record_ind;         /* Numéro d'enregistrement où sont stockés
                                     les autres numéros d'enregistrement */
        time_t last_acces;          /* Date du dernier accès au fichier */
        time_t last_modif;          /* Date de dernière modification du fichier */
        time_t last_modif_inode;    /* Date de dernière modification de l'inode */
        long size;                  /* Taille du fichier en octets */
        long link;                  /* Nombre de liens vers le fichier */
        short owner_user;           /* Nom du proprietaire */
        short owner_group;          /* Groupe du proprietaire */
        short perms;                /* Permissions en lecture, écriture et exécution
                                     du fichier pour le propriétaire, son groupe et
                                     les autres*/
}
```

Le champ `type` d'un `inode` permet de savoir si l'`inode` référence un fichier ou un répertoire. Dans le deuxième cas, la forme des données des enregistrements composant le fichier du répertoire est alors connue. En effet, il contient une référence sur le répertoire qui le contient (et sur lui même). Chaque autre éléments de ce fichier est composé d'un couple : nom de fichier, numéro d'`inode` qui décrit les fichiers du répertoire.

Le champ `num_record_ind` d'un `inode` est utilisé si le nombre d'enregistrements d'un fichier est supérieur à `NB_ENREG`. Si c'est le cas, ce champ contient le numéro d'enregistrement ou sont stockés les numéros d'enregistrements suivants du fichier.

Remarques :

- Dans LFS, ce sont les adresses physiques des enregistrements qui sont référencées, il n'y a pas de notion de numéro d'enregistrement. Cette solution a été possible facilement car LFS travail avec des enregistrements de tailles fixes (blocs), quelque soit la nature des données (données de contrôle du journal, `inode`, données...). Cette solution provoque un sur-coût important lors du compactage : il faut mettre à jour toutes les références. D'un autre côté, les accès en fonctionnement courant sont beaucoup plus directes et donc rapides. KITLOG ne permet pas d'avoir l'adresse physique d'un enregistrement. Les enregistrements sont référencés uniquement par un numéros d'enregistrement cela permet une gestion des enregistrements de tailles variables plus facile et des coût de compactage moins important.

- Si le numéro d'enregistrements de `l'inode` d'un fichier est stocké dans le répertoire à la place du numéro d'`inode`, la recherche d'un fichier est immédiate (sinon, une étape supplémentaire est nécésaire puisqu'a partir du numéro `d'inode` il faut trouver l'enregistrement qui stocke l'`inode`). Mais cela demande des mises à jour trop importantes en cas de modification ou de destruction du fichier. En effet, dans ce cas, l'`inode` du fichier est alors modifié (taille, date de dernière modification...). L'`inode` du répertoire est donc réécrit à la fin du journal pour référencer la nouvelle adresse de l'`inode` du fichier. De proche en proche, il va falloir réécrire tous les répertoires jusqu'à la racine. Il n'est donc pas souhaitable de chaîner les répertoires (et leurs fichiers) en écrivant le numéro d'enregistrement d'un `inode` dans le répertoire qui le contient.

4.3 Comment retrouver les `inodes` ?

Dans un premier temps, pour coller le plus au modèle de Sprite LFS, il a été envisagé la construction d'une table de hashage pour retrouver les `inodes`. Mais pour différentes raisons, qui sont exposées après la description cette organisation, il a été décidé d'utiliser aux maximum les possibilités offerte par KITLOG : créer un journal logique par fichier (cf 4.3.2).

Quelque soit l'organisation choisie, la politique de détermination de la taille des enregistrements est identique. Les enregistrements de données d'un fichier sont de tailles fixe, c'est-à-dire un bloc (512 octets). Les enregistrements des `inodes` sont de tailles fixes correspondant à la taille de la structure.

4.3.1 Table de hashage

Voyons maintenant la première solution pour retrouver les `inodes`. L'outil KITLOG permet de gérer plusieurs journaux logiques. Il est donc facile de mettre toutes les `inodes` dans un journal logique ce qui permet de les localiser dans leur ensemble. Le problème est alors un problème d'efficacité. Il ne faut pas être contraint de lire à chaque fois tout le journal pour trouver un `inode`. Une table de hashage est utilisée pour optimiser cette recherche. Cette table de hashage prend en entrée un numéro

d'`inode` et retourne le numéro d'enregistrement de l'`inode` recherchée, dans le journal logique. Ce journal logique contient uniquement et toutes les `inodes` du système de fichiers. Un autre journal logique contient les autres données. Pour des raisons de simplicité, le nom `J1` sera utilisé pour désigner le journal logique des données et `J2` pour celui des `inodes`.

La table de hashage est construite lors du démarrage ou d'une reprise sur panne. Si elle est construite au fur et à mesure des demandes, à chaque fois `J2` est parcouru, donc autant qu'il le soit une fois pour toute à l'initialisation. La figure III.5[1] donne un exemple de configuration de l'arborescence donné en figure III.4.

La racine a toujours le numéro d'`inode` numéro 1 et son père a le numéro d'inode 0 (qui est en fait lui même puisque la racine n'a pas de père par définition), comme sous UNIX. La racine est donc facile à repérer lorsque la table de hashage est constituée.

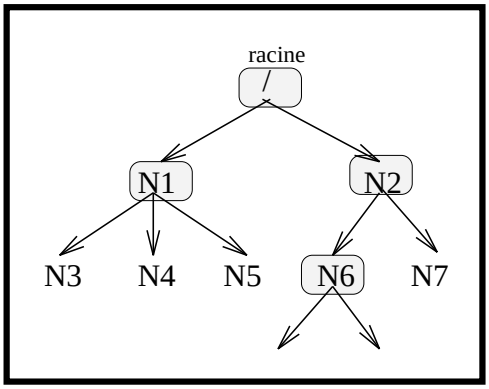

Figure III.4 : Configuration de l'arbre du système de fichier

Dans la figure III.5 prenons comme exemple la recherche du fichier de nom `/N1/N3` :

Le numéro d'enregistrement de l'`inode` de la racine est recherché : `I1` est entrée dans la table de hashage, qui donne en sortie `R0` dans `J2` (action 1).

L'`inode` de la racine est lu : elle fait référence à `R0` dans `J1` (action 2).

`R0` est lu dans `J1` : cet enregistremnt précise qu'il y a deux fichiers dans la racine de nom `/N1` et `/N2` avec leur numéro d'`inode` respectif. Le nom du fichier recherché est décomposé et l'arbre est alors parcouru du coté de la branche `N1` : `I2` est entrée dans la table de hashage (action 3).

La sortie de la table de hashage donne `R1` dans `J2` (action 4). Cet enregistrement est lu.

Le contenu du repertoire `/N1` est décrit dans `R1` de `J1` (action 5). Cette enregistrement est lu.

Cet enregistrement nous dit qu'il y a trois fichiers dans le répertoire `/N1` de nom

[1] La fonction de hashage est la fonction `modulo 7`. Cette fonction a été choisie pour limiter la taille de la figure. Dans l'implémentation, elle sera `modulo 17`.

`/N1/N3`, `/N1/N4` et `/N1/N5` avec leur numéro d'`inode` respectif. Le fichier est trouvé: `I4` est entré dans la table de hashage (action 6).

La position de l'`inode` (`R3` dans `J2`) est donnée grâce à la table de hashage (action 7).

L'`inode` est lu afin de trouver les enregistrements de données du fichier `/N1/N3` dans `J1` (action 8)

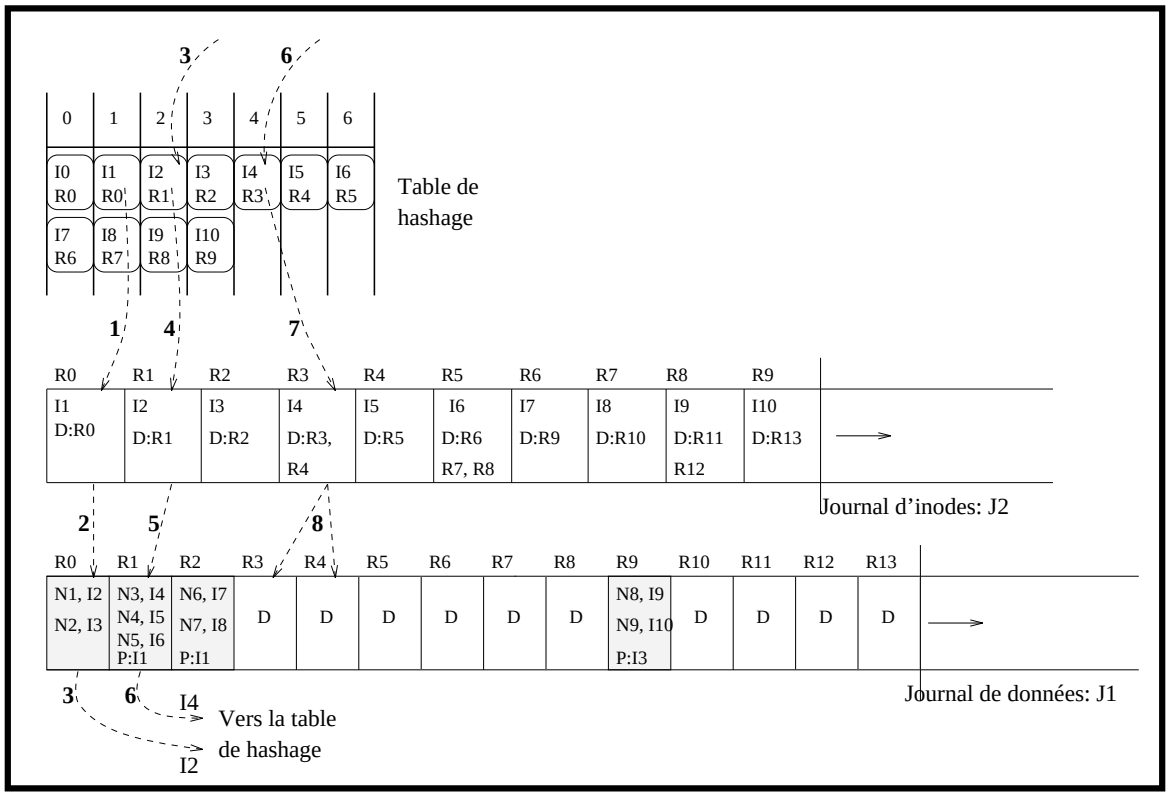

Figure III.5 : Recherche du fichier de nom /N1/N3

Trois modifications du système de fichiers, représentant des opérations types de ce qu'il est possible de faire, vont être exécutées pas à pas :

- Modification du fichier `/N1/N3`,
- Suppression du fichier `/N1/N4`,
- Création d'un répertoire `/N2/N10`.

Le résultat de ces trois opérations donne la configuration présentée dans la figure III.9. Cette nouvelle configuration donne l'arbre de la figure III.6. Le détail de ces trois étapes est maintenant décrit.

La modification du fichier `/N1/N3` a été choisie la plus simple possible. Cette modification est du type ajout d'un caractère à la fin du fichier. Le dernier enregistrement

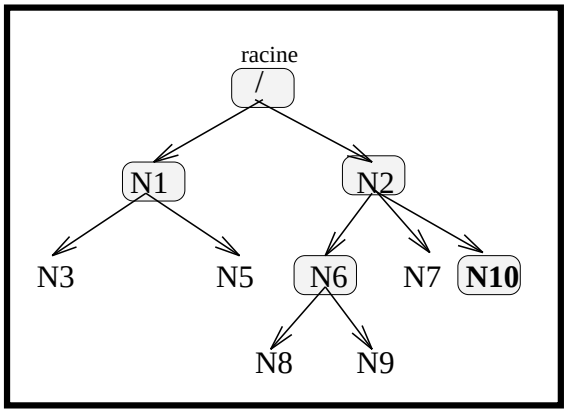

Figure III.6 : Configuration du nouvel arbre du système de fichier

de ce fichier n'étant pas plein de données, l'ajout d'un caractère ne modifie pas le nombre d'enregistrement de ce fichier. L'ensemble des actions décrites maintenant est représenté dans la figure III.7.

Le nouvel enregistrement de données modifié est écrit à la fin de `J1` (action 1). L'`inode` a besoin d'être modifiée, elle est donc écrite à la fin de `J2` (action 2). L'entrée dans la table de hashage de l'`inode` du fichier est modifiée (action 3) puis l'ancien enregistrement de données (`R3` dans `J1`) et l'ancien `inode` (`R4` dans `J2`) sont invalidés (action 4 et 5). Chaque écriture ou invalidation d'enregistrements est atomique, en cas de panne au moment de la modification du fichier, l'ancienne version du fichier est toujours valide.

La suppression du fichier `/N1/N4` est représentée dans la figure III.8 et se fait de la manière suivante : le fichier de données du répertoire dans lequel se trouve `/N1/N4` et son inode `I2` sont réécrits à la fin de `J1` et de `J2` (action 1 et 2). La table de hashage est mise à jour : `I2` ne se trouve plus en `R1` mais en `R11` (action 3) et `I5` est supprimé (action 4). Puis les enregistrements `R1` et `R4` dans `J2` et `R4` dans `J1` sont déclarés invalides (actions 5 à 7).

La création du répertoire `/N2/N10` est représentée dans la figure III.9 et se fait de la manière suivante : l'enregistrement de données décrivant le contenu du répertoire est écrit dans `J1` (`R15` - action 1), puis l'`inode` qui le référence dans `J2` (`R12` - action 2). Une nouvelle entrée est créée dans la table de hashage faisant correspondre `I11` à `R12` (action 3). Le répertoire père de `/N1/N10` est modifié : les données du répertoire (`R16` dans `J1` - action 4) et son `inode` (`R13` dans `J2` - action 5). La table de hashage est modifiée pour référencer le nouveau numéro d'enregistrement de l'inode modifié (action 6). Enfin, `R2` dans `J2` et `R2` dans `J1` sont invalidés (actions 7 et 8).

Si une panne intervient au moment ou une action est effectuée sur un fichier (modification, suppression ou destruction) aucune donnée n'est perdue. Cependant des actions créant des enregistrements peuvent avoir été faites. Ces enregistrements doivent être invalidés car ils ne sont pas référencés dans KFS et considérés par KITLOG comme

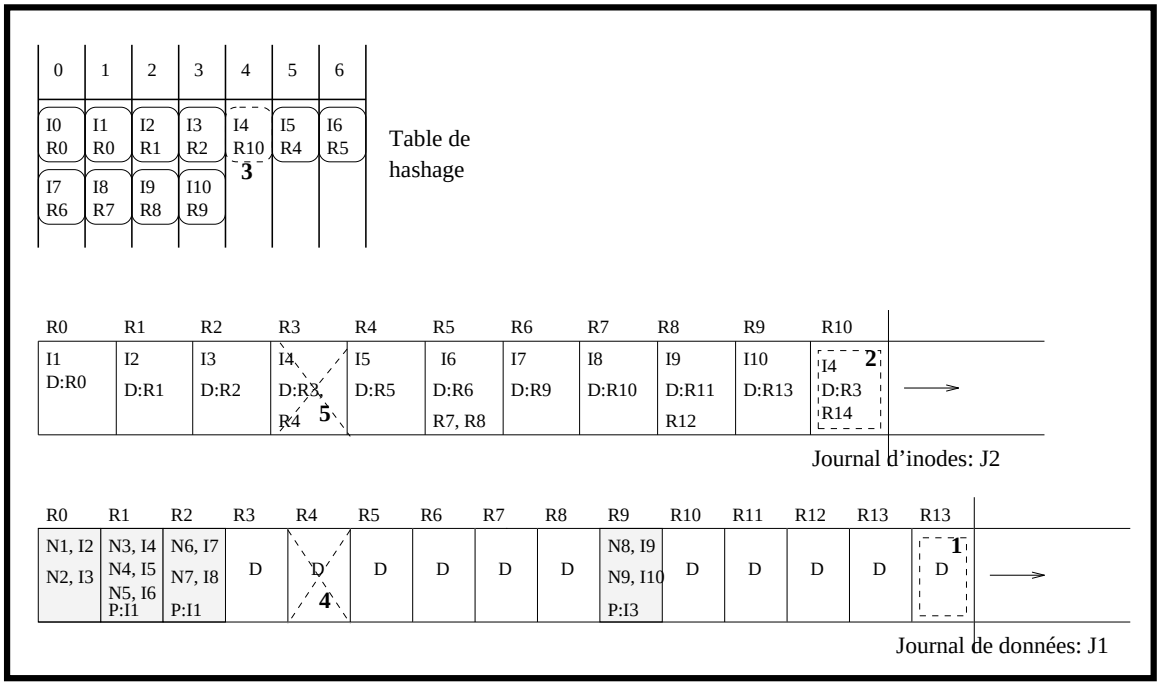

Figure III.7 : Augmentation de la taille du fichier /N1/N3

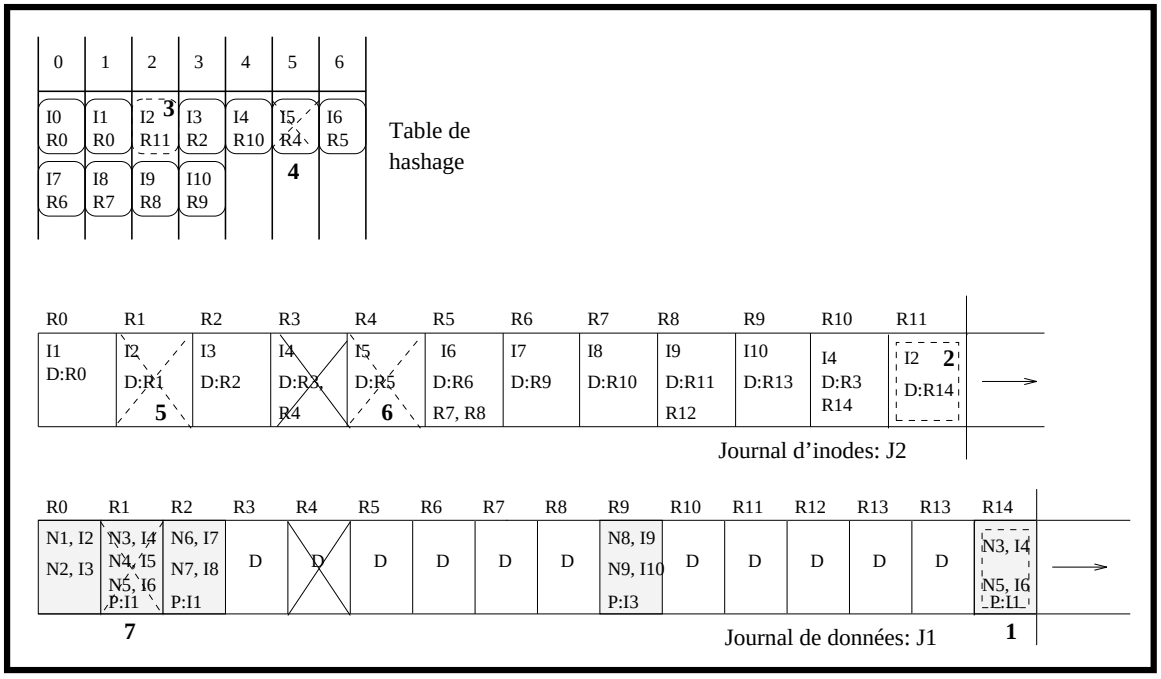

Figure III.8 : Suppression du fichier /N1/N4

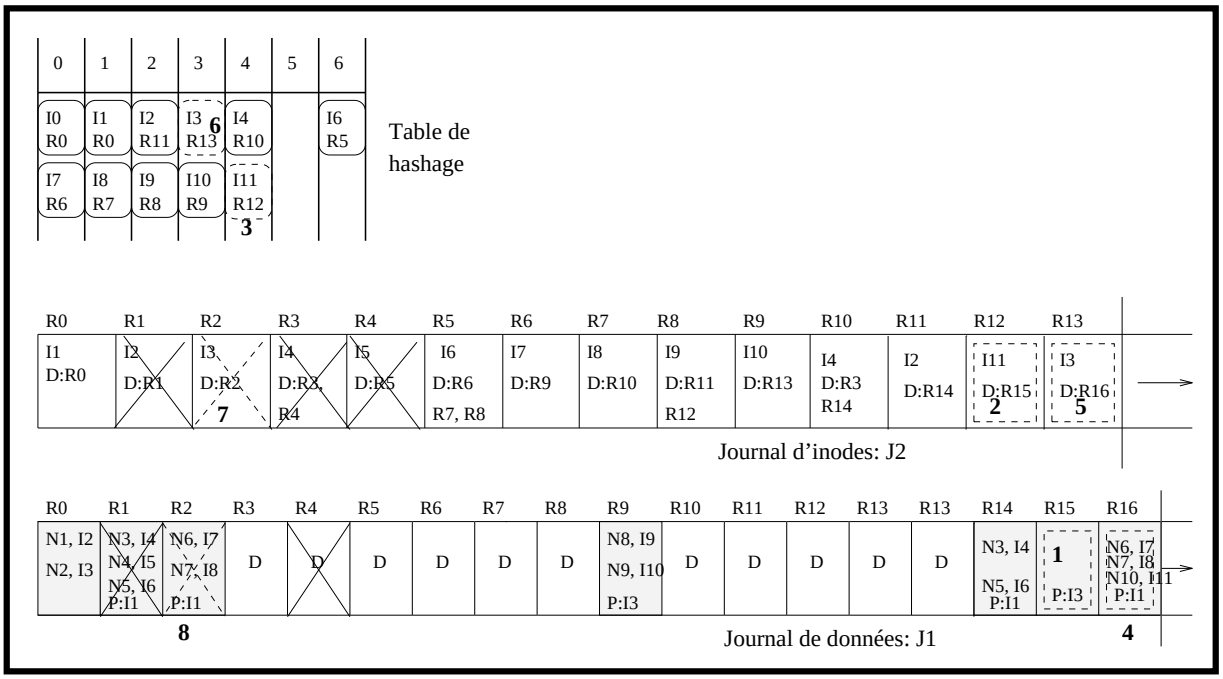

Figure III.9 : Création du répertoire /N1/N10

valides. Pour résoudre ce problème, un journal logique supplémentaire est créé pour journaliser ce type d'enregistrements et dès que l'action est terminée, ils sont invalidés. En cas de reprise sur panne l'ensemble des enregistrements référencés dans ce journal seront détruits.

Cette solution a été édifié pour coller le plus au modèle choisi, Sprite LFS. La notion d'`inode map` est remplacée par la table de hashage, pour palier à l'impossibilité d'écrire à un endroit fixe du support. La notion de journal logique, fournit par $K_I{}^T L^O{}_G$, est exploitée pour permettre un redémarrage plus rapide en différenciant les structures de contrôles et les données. Cependant cette solution n'ait pas très satisfaisante car au fur et à mesure de la manipulation des journaux et de $K_I{}^T L^O{}_G$, il a semble que $K_I{}^T L^O{}_G$ n'était pas utilisé de manière efficace. D'autre part, le nombre de lecture aléatoire et d'indirection semble beaucoup trop complexe. Suite à cela, il a été envisagé une autre solution qui est décrite maintenant.

4.3.2 Un journal logique par fichier

Maintenant, nous allons détailler une autre solution pour retrouver les `inodes`. Un journal logique est créé à chaque création de fichier, qu'il soit répertoire ou fichier de données. La modification d'un fichier entraîne des écritures et des invalidations d'enregistrements dans son journal mais ne modifie pas son nom de journal. Un nom de journal est donc alloué définitivement à un fichier au même titre qu'un numéro d'`inode` dans le système UNIX. Le nom de journal d'un fichier peut donc être gardé dans le fichier de son répertoire à la place du numéro d'`inode`. Un fichier est modifié,

créé ou détruit en allant de répertoire en répertoire (de journal en journal) à partir du journal de la racine.

Dans chaque journal il faut une structure pour définir le fichier qu'il contient. Cette structure est de la forme de l'`inode` et doit être en permanence à la fin du journal pour pouvoir être retrouvée et modifiée lorsque le fichier qu'elle décrit est modifié, créé ou détruit.

La localisation et la manière dont l'`inode` est retrouvé change. Un `inode` n'est plus retrouvé à partir de données écrites à un endroit fixe mais dans des journaux logiques. Le temps de recherche d'un fichier est maintenant principalement fonction du temps de recherche et de lecture par $K_I{}^T L^O{}_G$ d'un nouveau journal logique.

La figure III.10 donne la configuration de l'arbre de la figure III.4.

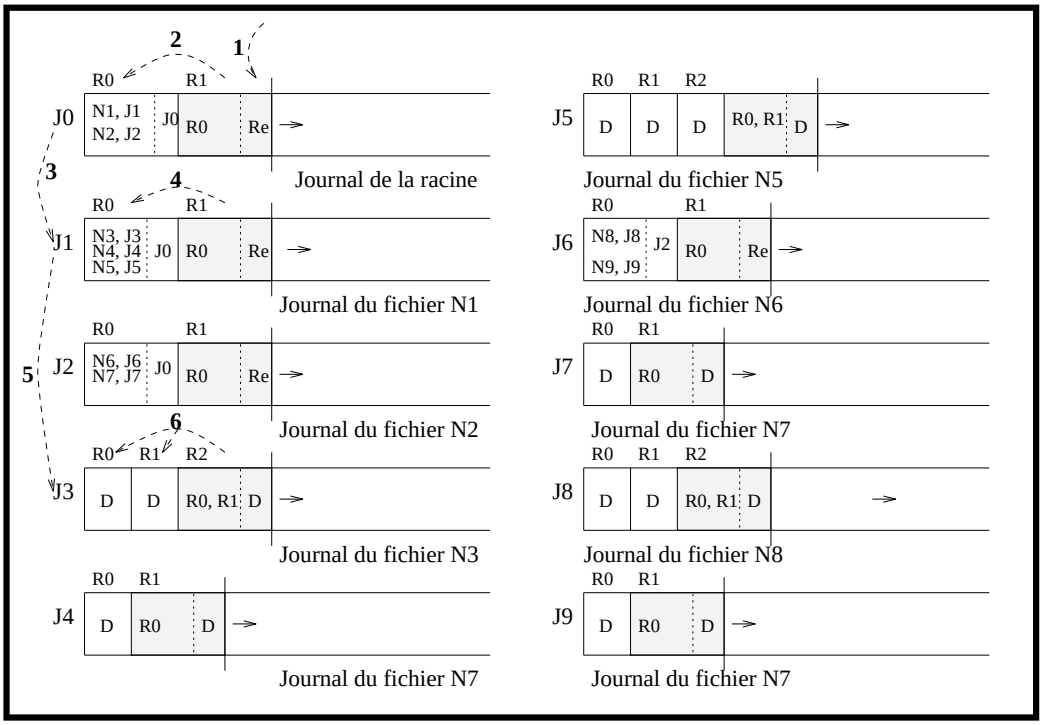

Figure III.10 : Recherche du fichier /N1/N3

Prenons comme exemple la recherche du fichier de nom `/N1/N3` comme dans la section précédente (4.3.1). Cette recherche est illustrée dans la figure III.10.

Il faut, dans un premier temps, connaître le nom de journal de la racine (dans l'exemple c'est `J0`). Le dernier enregistrement du fichier de la racine (`R1` dans `J0`) est lu (action 1). Cet enregistrement donne le numéro d'enregistrement, dans ce journal, qui décrit le contenu du répertoire (`R0`).

Ce dernier enregistrement est lu (action 2) et indique que la racine contient deux fichiers de nom `/N1` et `/N2` qui sont respectivement décrit dans `J1` et `J2`. Le nom

du fichier recherché est décomposé et l'arbre est alors parcouru du coté de la branche `/N1` qui se trouve dans `J1`.

Lecture du dernier enregistrement de `J1` (action 3). Le dernier enregistrement de `J1` indique le numéro d'enregistrement dans lequel est décrit le répertoire `/N1` (`R0`).

Cet enregistrement est lu (action 4), il donne l'information suivante : trois fichiers dans le répertoire `/N1` de nom `/N1/N3`, `/N1/N4` et `/N1/N5` avec leur nom de journal respectif (`J3`, `J4` et `J5`). Le nom du fichier recherché est décomposé et `J3` est lu.

Le dernier enregistrement de `J3` (`R2` - action 5) est lu pour savoir quel sont les enregistrements de données qui compose le fichier et l'ordre dans lequel ils doivent être concaténer.

Les enregistrements `R0` et `R1` dans `J3` composent le fichier `/N1/N3` (action 6).

Les trois modifications faites dans la section précédente (4.3.1) vont être exécutées pas à pas.

La modification du fichier `/N1/N3` est présentée dans la figure III.11. Elle est réalisée en écrivant le nouvel enregistrement de données (`R3` - action 1) suivit du nouvel `inode` (`R4` - action 2) dans `J3`. Ensuite, l'ancien enregistrement de données et l'ancien `inode` sont invalidés (actions 3 et 4).

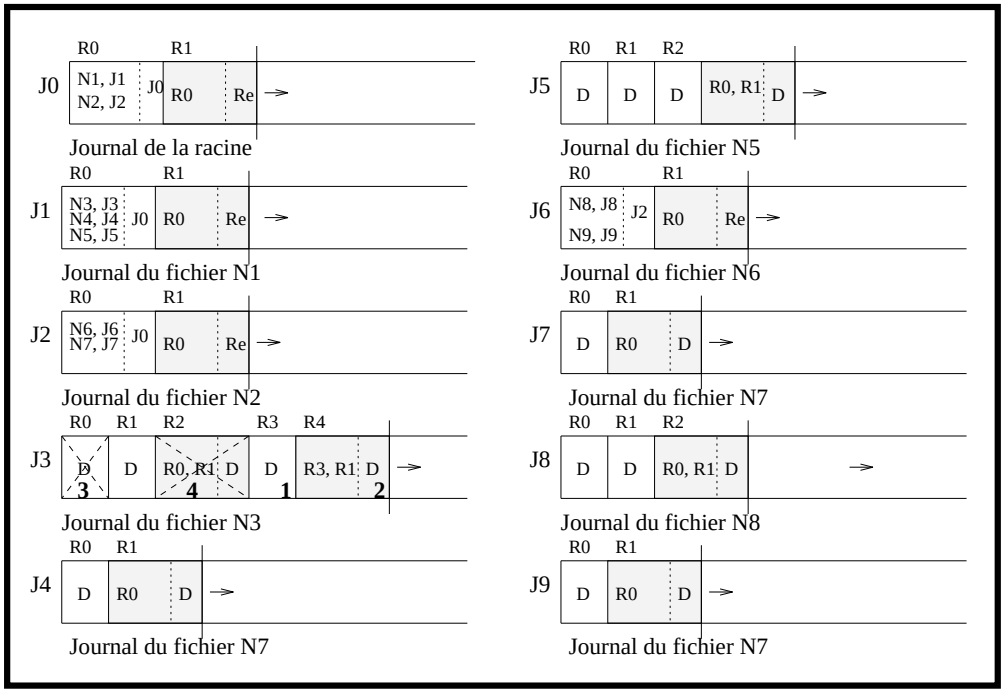

Figure III.11 : Modification du fichier /N1/N3

La suppression du fichier `/N1/N4` est présentée dans la figure III.12 et se réalise en deux étapes. Dans un premier temps, la suppression des références dans le

répertoire auquel il appartient est effectuée: modification de `R0` et de `R1` dans `J1`. `R0` est réécrit en `R2` (action 1) et `R1` en `R3` (action 2) toujours dans `J1`. Les anciens enregistrements peuvent être invalidé (actions 3 et 4). Dans un deuxième temps, la demande de suppression du journal `J4` à KITLOG est réalisée (action 5).

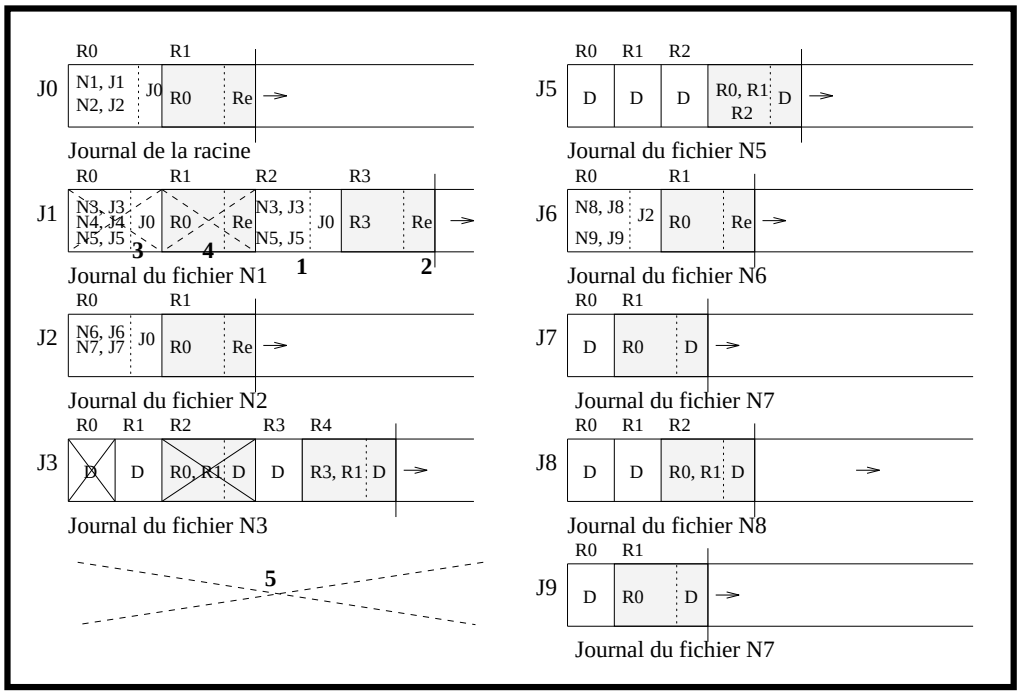

Figure III.12 : Suppression du fichier /N1/N4

La création du répertoire `/N2/N10` se fait de la manière suivante. Une demande de création de journal est envoyée à KITLOG, qui fournit en retour un nom de journal (dans l'exemple ce numéro est 10 - action 1). Dans ce nouveau journal, un premier enregistrement est écrit (action 2). Cet enregistrement représente le fichier qui décrit le répertoire (dans le cas d'un fichier de données, ce sont l'ensemble des enregistrements de données composant le fichier qui seraient écrit). Cet enregistrement contient pour l'instant que le numéro de journal de son répertoire père (dans ce cas là, c'est `J0`), puisqu'aucun répertoire ou fichier n'a encore été créé dans `/N2/N10`. Puis `l'inode` de ce fichier est écrit (`R1` sur `J10` - action 3). L'`inode` est donc bien le dernier enregistrement du journal. Pour finir, le numéro du nouveau journal est notifié dans le répertoire dans lequel il est (`J2`). `R0` et L'`inode` (`R1`) sont modifiées, ils sont donc réécrit en fin de `J2` (`R2` et `R3` - actions 4 et 5) puis invalidés (actions 6 et 7).

Pour prévenir des cas de reprise sur panne, des informations doivent être sauvegardées dans un autre journal pour permettre de défaire des écritures d'enregistrements qui font partie d'actions qui n'ont pas pu être réalisées dans leur totalité.

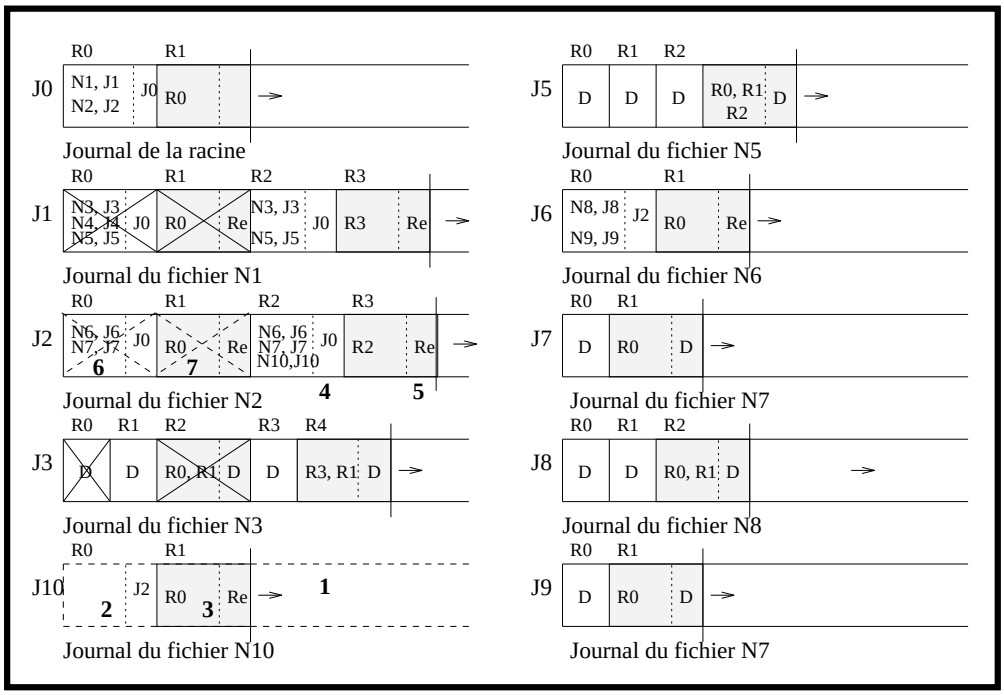

Figure III.13 : Création de répertoire /N2/N10

Ici les commentaires

4.3.3 Retrouver les journaux

Quelque soit les solutions choisies, il faut, lors de redémarrage ou de reprise sur panne connaître le nom du ou des journaux qui composent le système de fichiers. Or, KITLOG n'a pas de service de nommage. Le nom d'un journal est définit par KITLOG et est contitué d'un identificateur unique, composé en autre de la date. Lors de redémarrage ou de reprise sur panne, les noms des journaux n'étant plus en mémoire, ils ne peuvent pas être retrouvés. Seul le nom d'un journal suffit pour retrouver les autres. En effet, ce journal est alors un journal des noms des autres journaux.

Pour résoudre ce problème, KITLOG peut nommer un journal spécifique. Cela implique une écriture du nom d'un journal en dur dans le code de KITLOG. Cette solution est simple et ressoud le problème mais n'est pas une solution très « propre ». Si cette solution n'est pas possible, il faut trouver un moyen de stocker un nom de journal...

5 Interfaces

5.1 Interface avec le système d'exploitation

La librairie d'interface entre KFS et le système d'exploitation est à définir mais elle sera certainement de la forme de NFS.

5.2 Interface utilisateur

L'interface utilisateur sur les fichiers peut être composée de :

- `fd = open(*path, flag, mode)` /* Création ou ouverture d'un fichier, retourne le descripteur du fichier */
- `int = write(fd, *buffer, size)` /* Ecriture dans un fichier de descripteur `fd` le tampon `buffer` de taille `size` */
- `int = read(fd, *buffer, size)` /* Lecture dans un fichier de descripteur `fd` à partir du pointeur `pt` d'une longeur `size` */
- `off_t = lseek(fd, offset, whence)` /* Déplacement du pointeur sur le fichier */
- `int = close(fd)` /* Fermeture du fichier de descripteur `fd` */
- `int = rename(*path1, *path2)` /* Renommer ou déplacer un fichier */

L'interface utilisateur sur les répertoires est composée de :

- `dir = opendir(nom)` /* Création ou ouverture d'un répertoire */
- `closedir(dir)` /* Fermeture d'un répertoire */
- `readdir(dir)` /* Lecture d'un répertoire */
- `seekdir(dir, loc)` /* Fournit la position du pointeur sur le répertoire */

L'interface utilisateur sur les répertoires et les fichiers de données est composée de :

- `int = chmod(*path, mode)` /* Changement des droits d'un fichier */
- `int = stat(*path, *buffer)` /* Liste le contenu de l'`inode` */
- `int = link(*path1, *path2)` /* Lien physique */
- `int = unlink(*path)` /* Détruit un lien ou détruit le fichier si aucun lien n'existe */

6 Conclusion

La première partie de mon stage ést donc consacrée à la définition d'un système de fichiers ayant comme seul outil $K_I^T L^O{}_G$. Pour une mise en œuvre plus rapide ce système de fichiers devait s'appuyer sur le modèle du système de fichiers LFS du système Sprite. La réalisation de Sprite LFS ne permettait pas d'utiliser au maximum toutes les caractéristiques de $K_I^T L^O{}_G$. De ce fait une nouvelle réalisation a été pensée, et c'est qui est exposé dans ce chapitre.

Chapitre IV

conclusion

L'objectif de ce stage était l'utilisation de l'outil générique $K_I^T L^O_G$. Cette objectif devait être atteind en réalisant un système de fichiers s'appuyant uniquement sur $K_I^T L^O_G$. Ce système de fichiers devait avoir comme modèle le système de fichiers de Sprite LFS, mais comme nous l'avons vu, de nombreuses modifications on été apportées à ce modèle pour permettre une meilleurs utilisation de $K_I^T L^O_G$.

Après avoir fait une recherche approfondit sur la journalisation, des spécifications ont été définit pour permettre de réaliser ce système de fichiers appelé KFS. Puis, à partir de ses spécifications une réalisation est maintenant envisagée.

La deuxième partie de mon stage que j'effectue actuellement est consacrée à la réalisation de KFS. Dans un premier temps il va falloir définir de manière complète l'interface du système de fichiers avec le système d'exploitation pour qu'elle soit identique au système de fichiers d'UNIX en terme de fonctions et de paramètres. Cela va permettre de remplacer les points de montage de NFS par KFS de façon transparente.

Dans un deuxième temps l'ensemble de ces fonctions seront implémentées de manière à constituer le système de fichiers au dessus de $K_I^T L^O_G$.

Annexe A : Description rapide des projets référencés

Argus [Oki et al. 85] Argus est un langage et un environnement système permettant l'utilisation de transactions. Pour chaque verrou posé en écriture, une duplication des données est faite. Le système repose sur un mécanisme d'empilement de versions. Le journal ne contient que le minimum (transactions qui ont été validé ou échoué) et ne sert qu'au recouvrement du système, et non à défaire une transaction qui a échoué.

Camelot [Daniels et al. 87] est un gestionnaire de transaction qui utilise les journaux pour réaliser les propriétés intrasèques des transactions : atomicité face aux pannes, sérialisabilité et permanence.

Clio [Finlayson et al. 87] est un service de journalisation qui permet d'accéder à des fichiers de façon classique alors qu'ils sont en mode concaténation et lecture seulement.

Emacs est un editeur de texte. Il utilise la journalisation pour la mise en place de la commande `UNDO` qui permet de défaire des actions faites par l'utilisateur.

Instant Replay [Leblanc et Mellor-Crummey 87] Solution de deboguage de programme parallèle. Utilisation des journaux pour rejouer des session de deboguage.

Isis [Isis 91] Isis est un système qui par sa structure même facilite la construction d'applications sûres. Isis n'assure pas la sûreté de fonctionnement des applications mais donne le moyen de la réaliser à faible coût. Ses protocoles de diffusions évitent des interblocages, facilitent le recouvrement. La notion d'objet réparti permet une grande souplesse dans la construction d'applications réparties.

$K_I^T L^O{}_G$ [Ruffin 92a, Ruffin 92b] est un outil de journalisation générique. Il permet à toute application qui veut utiliser un journal de l'utiliser a travers une interface simple, $K_I^T L^O{}_G$ se chargeant de le gérer en terme, par exemple, de compactage, de reprise sur panne.

QuickSilver [Haskin et al. 87] QuickSilver est un système réparti, qui offre un certain nombre de services transactionnels : un gestionnaire de transactions, un gestionnaire de recouvrement et un gestionnaire de journal. QuickSilver est un système ouvert, qui fournit des outils minimaux permettant la construction d'applications

fiables.

Sprite LFS [Douglis and Ousterhout 89] est le système de fichier du système répari Sprite. Ce système de fichier utilise qu'un seul journal dans lequel sont journalisés les données des fichiers et les données de contrôle. Tous les enregistrements de ce journal sont de taille fixe : un bloc.

Annexe B : Glossaire

Action (action) Opération modifiant des données, ou agissant sur des ressources (impression, affichage à l'écran...).

Application (application) Vue d'un service système, une application est un programme ou un ensemble de programmes, utilisant ce service. Un service système utilisant d'autres services systèmes sera vu de leur part comme une application.

Atomicité (Atomicity) Assure que l'ensemble des sous-actions d'une action vue de l'extérieur est indivisible. C'est-à-dire, que soit toutes les sous-actions d'une action sont effectuées, soit aucune ne l'est. En pratique, si une panne survient, alors que seule une partie des sous-actions a été effectuée, toutes ces actions seront défaites.

Cohérence des transactions (transaction consistency) La propriété de cohérence des transactions spécifie qu'une transaction appliquée à un ensemble de données cohérentes, rend cet ensemble dans un état cohérent.

Communication Asynchrone (asynchronous communication) Une communication asynchrone entre deux processus A et B implique que le processus émetteur A ne soit pas bloqué à partir du moment où le système de communication a pris en charge le message. Toutefois le processus A pourra être bloqué durant certaines phases du traitement de la communication. Une communication asynchrone fiable est susceptible soit de bloquer le processus émetteur A en attente d'un accusé de réception ou d'un message d'erreur en provenance du service de communication, soit de lui permettre de consulter l'état de son envoi par la suite.

Durabilité ou permanence (persistency) Propriété qui assure qu'une action faite le sera pour toujours quoiqu'il advienne au système.

Enregistrement (Record). L'unité d'écriture sur le journal. Il peut être de taille variable ou de taille fixe suivant les applications. Un enregistrement peut contenir n'importe quel type de données.

Fiabilité (Reliability) Aptitude d'un système, d'un matériel à fonctionner sans incident pendant un temps donné (Petit Robert). En fait il faut apporter une nuance à cette définition. On considère que les causes d'incidents possibles ne sont pas le fruit de la malveillance d'usagers (programmeurs ou autres). Le degré de la fiabilité d'un système est fonction de la durée probable et des conditions de fonctionnement sans incident de ce système.

Générique (Generic). Se dit d'un système (ou d'un service) non-dédié à une utilisation particulière. Un système générique offre un ensemble de fonctions permettant de satisfaire les besoins de la majorité des différentes variétés d'applications en limitant le surcoût dû à l'existance de fonctions non utilisées.

Indépendance ou sériabilité (Serialisability) Technique qui ordonne les accès aux données, en cas de parallélisme. Bien que les actions peuvent s'exécuter concurremment, les effets d'une action doivent être perçus comme si elle s'était exécutée en série avec les autres actions en cours d'exécution. Une exécution d'un ensemble d'actions est sérialisable, s'il existe une exécution en série de ces mêmes actions, qui produirait le même effet.

Journal (log). Un **journal** est une sématique de stockage particulière. Les écritures se font par concaténation d'enregistrements en fin de journal. Certains enregistrements du journal peuvent devenir obsolètes. Ils sont alors effacés par compactage. Aucun autre type d'écriture n'est possible. La lecture est généralement séquentielle. La **lecture en arrière** (backward read) consiste à lire le dernier enregistrement du journal et de proche en proche à lire les enregistrements précédents. La **lecture en avant** (forward read) commence par la lecture d'un enregistrement quelconque du journal et continue par la lecture des enregistrements suivants. De plus, une opération particulière `flush` guarantit, une fois exécutée, que tous les enregistrements ont été effectivement écrit sur le support physique stockant le journal.

Journal logique (logical log) Le journal tel qu'il est vu par une application. Il est composé d'enregistrements ayant un lien logique entre eux. Dans KITLOG, il souvent symbolisé par un identificateur unique de journal. Ceci de manière à ne pas confondre avec ce qui est stocké sur les différents supports qui stockent un multiplexage d'enregistrements de différents journaux logiques. Si un support comprend un et un seul journal logique dans sa totalité alors le stockage du journal peut être confondu avec la notion de journal logique.

Journal physique (physical log) Un **journal physique** (physical log) correspond à un ensemble d'enregistrements appartenant à différents journaux logiques qui ont été multiplexé sur un même support de stockage.

Réplication (replication) Comme duplication excepté que le nombre d'exemplaires n'est pas limité à deux.

Reprise après panne Actions à entreprendre au redémarrage à la suite d'une panne afin de restaurer un état cohérent.

Support de stockage (storage medium) Ressource capable de stocker des données tels que la mémoire, le disque, la bande magnétique.

Sûreté de fonctionnement La sûreté de fonctionnement est ce qui permet dans un système réparti de palier aux comportements indésirables résultants des pannes, des erreurs, du parallélisme et de la malveillance.

Transaction (transaction) Collection d'opérations commenc,cant par une marque indiquant son début et finissant par une marque de fin correspondant à succès ou

un echec. Une transaction verifie les propriété de cohérence, d'atomicité face aux panne, de sérialisabilité et de permanence.

Références

[**Baer et al. 81**] J. L. Baer, G. Gardarin, C. Girault, G. Roucairol : « The Two-Step Commitment Protocol: Modeling Specification and Proof Methodology". IEEE, Proceedings of the 5^{th} International Conference on Software Engineering, San Diego (CA, USA). 9–12 mars 1981.
Une modélisation par réseaux de Petri d'une variante de la validation à deux phases.

[**Banâtre et al. 91**] Michel Banâtre, Gilles Muller, Brunot Rochat, Patrick Sanchez : « Design decisions for the FTM: A General Purpose Fault Tolerant Machine ». Rapport de Recherche INRIA N°1400. Mars 1991.
Description d'une solution matérielle pour la réalisation d'une machine tolérante aux pannes.

[**Isis 91**] Isis Distributed Systems : « The Isis Distributed Toolkit, Version 3.0, User Reference Manuel ». Chapitre 15 : « Logging and spooling ». 1991. *Manuel d'utilisation et de programmation du service de journalisation d'ISIS.*

[**Daniels et al. 87** Dean S. Daniels, Alfred Z. Spector, Dean S. Thompson : « Distributed Logging for Transaction Processing ». Proceedings of ACM Special Interest Group on Management of Data. San Francisco. May 27-29, 1987. SIGMOD Record, Vol. 16, N°3, pages 82–96. Decembre 1987.
Algorithme de gestion de la réplication répartie du journal de Camelot.

[**Daniels 88**] Dean Spencer Daniels : « Distributed Logging for Transaction Processing ». Phd Thesis, CMU-CS-89-114. Décembre 1988.
Etude des différents choix de conception et réalisation des journaux, étude du journal de Camelot.

[**Davidson et al. 85**] Susan B. Davidson, Hector Garcia-Molina, Dale Skeen : « Consistency in Partitioned Networks ». ACM Computing surveys, Vol 17, N°3, Pages 341–370. Septembre 1985.
Une synthèse sur les techniques de maintien de la cohérence en cas de partitionment de réseau.

[**Deswarte 91**] Yves Deswarte : « Fragmentation-Redundancy-Scatering : a means to tolerate accidental faults and intrusions in distributed systems ». Proceedings of the Ercim Workshop on Distributed systems, pages 31–34. INESC, Lisbon, Portugal. 14–15 Novembre 1991.
Un très court article, qui explique bien les problèmes de sécurité et propose une solution.

[**Finlayson et Cheriton 87**] Ross S. Finlayson, David R. Cheriton : « Log Files: An Extended File Service Write-Once Storage ». ACM 11^{th} Symposium on Operating System Principles, Austin (TX); Pages 139–148. Novembre 1987.
Etude d'une représentation physique de journal.

[**Haskin et al. 87**] Roger Haskin, Yoni Malachi, Wayne Sawdon, Gregory Chan : « Recovery Management in QuickSilver ». Proceedings of the Eleventh ACM symposium on Operating Systems Principles; Austin, Texas; pages 75–86. 8-11 Novembre 1987.

Description d'une approche de recouvrement originale : celle de QuickSilver où le minimum est fait par le système et le reste est écrit par l'usager.

[Haerder et Reuter 83] Theo Haerder, Andreas Reuter : « Principles of Transaction-Oriented Database Recovery ». ACM Computing Surveys, Vol. 15, N°4. Decembre 1983.
Une synthèse des techniques de recouvrement.

[Johnson et Zwaenepoel 88] David B. Johnson, Willy Zwaenepoel : « Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing ». Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing. Toronto (Canada), pages 171–181. Août 1988.
Une méthode optimiste pour la reprise après panne basée sur la journalisation de message et le calcul d'état global cohérent.

[Kistler and Satyanarayanan 91] James J. Kistler and M. Satyanarayanan : « Disconnected Operation in the Coda File System ». Proceedings of the 13th ACM Symposium on Operating Systems Principles; Pacific-Grove, USA. Operating Systems Review, Vol 25, N°5, pages 213–225. Octobre 1991.

[Leblanc et Mellor-Crummey 87] Thomas J. Leblanc and John M. Mellor-Crummey: « Debugging Parallel Programs with Instant Replay ». IEEE Transactions on Computers, Vol. C-36, N°4, pages 471–482. Avril 1987.
Description d'un débogueur qui permet de rejouer des sessions de deboguage.

[Lorie 77] Raymond A. Lorie : « Physical Integrity in a Large Segmented Database ». ACM Transaction on Database Systems, Vol. 2, *N*°1, pages 91–104. March 1977.
Description détaillée du mécanisme des pages-ombre.

[Mary G. Baker and al. 91] Mary G. Baker and John H. Hartman an Michael D. Kupfer and Ken W. Shirriff and John K. Ousterhout : « Measurements of a Distributed File System ». 13^{th} Symposium on Operating System Principles, Pacific-Grove. Operating System Review, Vol 25, N°5, pages 198–212. October 13–16 1991.

[Oki et al. 85] Brian M. Oki, Barbara H. Liskov, Robert W. Scheifler : « Reliable Object Storage to Support Atomic Actions ». Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, USA. Operating Systems Review, Vol. 19, N°5, pages 147–159. 1–4 Decembred 1985.
Gestion du stockage d'objets dans Argus, algorithmes pour l'écriture atomique répartie d'objets en stockage fiable.

[Papadimitriou 79] Christos H. Papadimitriou: « The serialisability of Concurrent Database Updates ». Journal of the ACM, Vol. 26 N°4, pages 631–653. Octobre 1979.
Etude du problème de la sérialisabilité, proposition d'un formalisme pour le contrôle de concurrence.

[Rosenblum and Ousterhout 91] Mendel Rosenblum and John K. Ousterhout : « The Design and I¡mplementation of a Log-Structured File System ». 13^{th} Symposium on Operating System Principles, Pacific-Grove. Operating System Review, Vol 25, N°5, pages 1–15. October 13–16 1991.

[Ruffin 89] Michel Ruffin : « Transactions : synthèse ». Rapport technique N°53, projet SOR, INRIA, Rocquencourt. Mars 89.
Synthèse sur la notion de transaction atomique.

[Ruffin 92a] Michel Ruffin: « Kitlog: a Generic Logging Service ». Proceedings of the Eleventhenth Symposium on Reliable Distributed Systems, Houston, TX (USA), pages 139–146. 5–7 Octobre 1992.
Description rapide du modèle et de la méthode de journalisation de Kitlog.

[Ruffin 92b] Michel Ruffin : « Kitlog : Un Service de Journalisation Générique ». Thèse de l'université Pierre et Marie Curie, Paris VI. Septembre 1992.
Description détaillée de la conception et de la réalisation d'un service de journalisation à but général.

[Weihl 85] William E. Weihl : « Atomic Data Types ». MIT Laboratory for computer Science. Ref ???. 1985.
Article assez théorique sur l'atomicité et les objets.