ext3Viewer – Documentation développeur

Laurent Sebag Nathan Periana

5 avril 2007

Résumé

Cette documentation a pour but de faciliter la reprise de code par un autre programmeur afin qu'il puisse aisément comprendre le programme. Nous allons y expliquer l'accès aux structures du système de fichier ainsi que l'organisation mise en place dans notre programme.

Table des matières

Avant-propos

Pour une meilleure lisibilité, définissons quelques conventions pour ce document. Tout d'abord, le vocabulaire technique propre au système de fichier sera écrit en Anglais, pour la cohérence avec le code. De plus, chaque mot emprunté au vocabulaire du système de fichier ext3 sera mis en valeur.

Nous écrirons donc le *superblock*, et non le *superblock*. De même les champs des structures seront écrites en gras, exemple : **ext3_super_block**, **ext3_inode** ...

Première partie

Les structures du système de fichier ext3

Chapitre 1

Accès au structures

Afin de réaliser le projet ext3Viewer, nous avions besoin d'accéder aux structures ext3. Cependant il n'existe pas de fonctions dans LibC permettant de lire les "briques" du système de fichier. Il fallait donc construire nos propres fonctions de lecture. Chaque fonction fait un calcul pour se positionner au bon endroit sur la partition, puis lit les données.[1]

1.1 Lire le *superblock*

Lire le *superblock* est la première chose à faire lorsque l'on veut explorer une partition. On y trouve des informations propres à la partition comme la taille d'un *block*. Pour le lire, rien de plus simple : on se déplace de 1024 octets correspondant à la taille du *boot sector* puis on remplit une structure struct ext3_super_block de taille sizeof(struct ext3_super_block).

1.2 Lire un *group descriptor*

L'emplacement du *group descriptor* ext3_group_desc suit la politique suivante : il se trouve après le *boot sector* (le *boot sector* n'est pas utilisable par ext3, c'est un espace réservé), et après le *superblock*[2], cependant le décalage nécessaire pour atteindre le *group descriptor* est variable. En effet on stocke toujours le *superblock* sur la taille d'un *block* (1, 2, ou 4 Ko) et le *boot sector* est sur 1 Ko. Si la taille d'un *block* est supérieure à 1 Ko, on stocke le *boot sector* et le *superblock* dans un même *block*.

Pour lire le *group descriptor*, on utilise le champ s_first_data_block qui applique la politique expliquée ci dessus. Ce champ indique un nombre n. Il faut sauter $(n+1)$ *blocks* pour lire le *group descriptor*.

Exemples :

- si un *block* = 1024 oct, alors le descripteur de groupe est situé après les 2048 oct occupés par le *boot sector* et le *superblock*.
- si un *block* = 4096 oct, alors le *group descriptor* est situé après les 4096 oct partiellement occupés par le *boot sector* et le *superblock*. Il y a en fait un espace vide entre le *superblock* et les *group descriptors*.

[1] on se positionne avec lseek() ; la lecture s'effectue simplement avec la fonction read()

[2] le *superblock* est toujours situé après le 1er Ko occupé par le *boot sector*

1.3 Lire une *inode*

On lit le *group descriptor*, ce *group descriptor* contient un champ bg_inode_table qui contient le numero de *block* où commence l'*inode table*. Il nous reste plus qu'à faire un *lseek* de

$$\begin{array}{rl} & \textit{numero de block} \times \textit{taille d'un block} \\ + & \textit{numero de l'inode dans le groupe} \times \textit{taille d'un inode}. \end{array}$$

Evidemment, tous les *inodes* ne se trouvent pas sur la même *inode table*. Il faut calculer sur quel groupe se trouve notre *inode* avant de le lire. Pour cela nous disposons d'un champs dans le *superblock* qui donne le nombre d'*inodes* par groupes.

1.4 Lire une *inode bitmap*

Après avoir lu le *group descriptor* d'un groupe donné, il est facile de lire une *inode bitmap*. En effet il existe un champ bg_inode_bitmap dans la structure ext3_group_desc qui contient le numéro de *block* où se situe cette *inode bitmap*. Il suffit de se placer à ce numéro de *block* et de lire l'*inode bitmap*.

1.5 Lire une *block bitmap*

La lecture de la *block bitmap* s'effectue de la même façon que la lecture d'une *inode bitmap*. On regarde le champ bg_block_bitmap de la structure ext3_group_desc et on effectue une lecture avec read().

1.6 Allocation des *blocks* pour un *inode*

Les *blocks* sont les briques qui forment les données d'un fichier. Pour chaque *inode*, on alloue un certain nombre de *blocks*, ils sont pointés par les éléments du tableau i_block[EXT3_N_BLOCKS]. Ce tableau est composé de 15 éléments. On pourrait stocker les données directement dans ce tableau, cependant si on fait le calcul, on ne peut que stocker dans un fichier $15 \times \textit{taille d'un block} = 60\ Ko$[3], en appliquant cette méthode. En fait, certains *blocks* vont servir de table de *blocks* pour pouvoir adresser plus de place pour un *inode*.
Dans les exemples suivants, on utilise un block *de taille 4096 oct.*

1.6.1 Blocs directs

Les éléments du tableau i_block[] de 0 à EXT3_NDIR_BLOCKS $-$ 1 sont des pointeurs vers les *blocks* de données. Pour lire les données associées au fichier, on lit simplement un *block* n en sautant i_block[n] $\times$ (taille d'un *block*) octets.

[3]si on un utilise des *blocks* de 4 Ko

1.6.2 Blocs indirects

À partir du EXT3_IND_BLOCK - ième champ du tableau i_block[], on ne stocke plus les données directement à l'emplacement i_block[], on indexe. On considère le champ i_block[EXT3_IND_BLOCK] comme une table de pointeurs. Pour lire les données on lit sur i_block[EXT3_IND_BLOCK] 4 oct par 4 oct les adresses des *blocks* de données.

On peut donc stocker grâce à un *block* indirect[4] :

$$\frac{taille\ d'un\ block}{4\ oct} \times\ taille\ d'un\ block = 4096\ Ko \tag{1.1}$$

1.6.3 Blocs indirects doubles

Pour lire les *blocks* indirects doubles, le principe est le même que pour les *blocks* indirects. Cette fois-ci on considère i_block[EXT3_DIND_BLOCK] comme une table de table de *blocks* de données.

On peut donc stocker grâce à un *block* indirect double :

$$\left(\frac{taille\ d'un\ block}{4\ oct}\right)^2 \times\ taille\ d'un\ block = 4096\ Mo \tag{1.2}$$

1.6.4 Blocs indirects triples

La lecture des *blocks* indirects triples se fait comme suit : on lit i_block[EXT3_TIND_BLOCK] 4 oct par 4 oct pour obtenir l'adresse d'une table de tables qui contient les données. Puis on lit cette seconde table pour avoir les tables de *blocks* de données. En lisant cette dernière table on connait l'adresse des données et on peut les lire.

On peut donc stocker grâce à un *block* indirect triple :

$$\left(\frac{taille\ d'un\ block}{4\ oct}\right)^3 \times\ taille\ d'un\ block = 4096\ Go \tag{1.3}$$

CCL : Calculons la taille maximum d'un fichier

$$\begin{aligned} taille\ max\ fichier = &\ blocks\ directs + (1.2) + (1.3) + (1.4) \\ = &\ 4096 \times \left(12 + \left(\tfrac{4096}{4}\right) + \left(\tfrac{4096}{4}\right)^2 + \left(\tfrac{4096}{4}\right)^3\right) \\ \approx &\ 4\ To \end{aligned} \tag{1.4}$$

1.7 Lire un dossier

Sous ext3, un dossier est représenté par un inode qui contient (*blocks* alloués) un ensemble de *dir entries*. On doit tester le type de fichier de l'*inode* grâce au champ i_mode de l'*inode*. Puis après avoir vérifié qu'il s'agit bien d'un dossier, on lit les *blocks* alloués à l'*inode* pour remplir une structure ext3_dir_entry. Dans cette *dir entry*, on a plusieurs informations à propos du premier fichier contenu

[4] considérons qu'un block fait 4096 octets

dans le répertoire. Parmi celles-ci, on trouve rec_len, qui représente la taille d'un enregistrement d'un fichier dans le répertoire.

On aurait pu lire les *blocks* de données n oct par n oct, avec n = sizeof(struct ext3_dir_entry_2). Cependant comme il peut y avoir des espaces vides dans les *blocks*[5], on n'applique pas cette méthode. Le principe consiste à lire par taille de n oct mais on se déplace de rec_len oct entre chaque lecture de *directory entry*. rec_len pointe toujours vers le prochain ext3_dir_entry. On se déplace dans les *blocks* de données d'un répertoire comme on se déplacerait dans une liste chainée. Il faut néanmoins faire attention que le décalage effectué pour lire les ext3_dir_entry_2 ne soit pas supérieur à la taille d'un *block*. Dans ce cas il faut continuer la lecture sur le *block* suivant qui n'est pas forcément le *block* suivant sur le disque.

1.8 Lire un *symlink*

Le *symlink* est un lien vers un fichier. C'est en fait un *inode* qui a pour données une chaine de caractères : le chemin vers le fichier pointé. Cependant pour économiser de l'espace, ext3 n'alloue pas toujours de *blocks* pour un *symlink*. La règle est la suivante : si la taille du lien est inférieure à 60 octets, alors on peut lire ex3_inode.i_block comme une chaine de caractères. Sinon, on parcours l'arbre des *blocks* alloués à l'*inode* et on obtient la chaine de caractères.

1.9 Lecture du journal

La journalisation qui est utilisée avec le système de fichier ext3 n'est pas propre à ext3. En effet, c'est un mécanisme qui peut être utilisé en théorie par n'importe quel système de fichier. Le système de journalisation s'appelle jbd.

L'etude du journal fut l'étape la plus dure de notre projet car nous manquions de documentation technique sur le journal. Il existe une API qui permet de manipuler le journal, cependant celle-ci a un niveau d'abstraction trop haut, qui ne permet pas d'accéder aux données souhaitées. Nous avons tout de même réussi à extraire quelques données du journal interne au système de fichier.

Remarques :

Il est important de noter que toutes le structures du journal sont en big endian, il est donc necessaire pour afficher le contenu d'une structure de transformer l'ordre des octets sur la machine locale. Pour cela il existe des fonctions comme be32_to_cpu().

Comme dans ext3, il existe aussi une notion de *block* dans le journal, d'ailleur on trouve un champ *block size* dans le *superblock* du journal. Néanmoins cette taille de *block* semble coïncider avec la taille d'un *block* du système de fichier. Il semblerait que ce champ soit fourni pour des raisons de compatibilités avec les autres systèmes de fichiers.

Tout d'abord présentons les différentes structures de jbd définies dans /usr/-include/linux/jbd.h.

[5] à cause des suppressions de fichiers

1.9.1 Le *journal superblock*

Tout comme le système de fichier, le journal contient des informations générales qui sont stockées dans un *superblock*. Cette structure se situe sur les *blocks* alloués à l'inode du journal. L'inode du journal est defini dans le champ s_journal_inum du *superblock* du système de fichier (en général, il s'agit de l'*inode* 8). Le premier *block* alloué à cet *inode* contient le *journal superblock*.

Parmis les champs utiles à la lecture du *journal superblock* il y a s_header.h_magic qui permet de tester s'il s'agit bien d'un *superblock* valide. Le champ s_start permet de connaître l'état du journal : s'il vaut 0, alors le journal est vide. Sinon, le journal contient des entrées, et se termine lorsque les compteurs t_blocknr ne sont plus consécutifs entre deux *blocks*.

1.9.2 Les *journal headers*

Après le *block* contenant le journal, on trouve des *blocks* de descrtiption d'une action dans le journal. On peut vérifier la validité d'un *block* grâce au *magic number*. Il existe cinq différents types de *blocks* définis dans l'en-tête jbd.h. Un *block* JFS_DESCRIPTOR_BLOCK prévient du début de la description d'une transaction. La transaction se termine par un JFS_COMMIT_BLOCK. Après chaque *block* de description, on trouve une liste de *journal block tags*.

1.9.3 Les *journal block tags*

Une transaction semble pouvoir concerner plusieurs *blocks* du système de fichier. La liste des *journal block tags* donne une correspondance entre un numéro de *block* et un *flag* qui definit l'action. On indique la fin de cette liste par un *flag* JFS_FLAG_LAST_TAG.

Chapitre 2

Interprétation des champs des structures ext3

2.1 ext3_super_block

Il contient des informations administratives importantes, portant sur le système de fichiers, pour le lancement du système d'exploitation. Il existe des copies du *superblock* à plusieurs endroits sur un système de fichier. Il est représenté par la structure ext3_super_block, sur au plus un block.

Explication des champs du superblock :
__u32s_inodes_count ;
Nombre total d'inodes
__u32ss_blocks_count ;
Nombre de blocks
__u32ss_r_blocks_count ;
Nombre de blocks réservés
__u32ss_free_blocks_count ;
Nombre de blocks libres
__u32ss_free_inodes_count ;
Nombre d'inodes libres
__u32ss_first_data_block ;
Numéro du premier block contenant les données.
__u32ss_log_block_size ;
Taille des blocks, de la forme : $1024 * 2^{log\_block\_size}$. Donc si la valeur est 0, la taille d'un *block* sera de $1024 * 2^0 = 1024$ octets.
__s32s_log_frag_size ;
Taille des fragments.
__u32ss_blocks_per_group ;
Nombre de blocks par groupe
__u32ss_frags_per_group ;
Nombre de fragments par groupe
__u32ss_inodes_per_group ;
Nombre d'inodes par groupe
__u32ss_mtime ;

Timestamp de la dernière opération de montage. Le timestamp est au format UNIX (c'est à dire, le nombre de secondes écoulées depuis le 1er Janvier 1970, à 0h00mn00sec ; pour en savoir plus ,regarder man 3 ctime .)
__u32ss_wtime ;
Timestamp de la dernière opération d'écriture
__u16s_mnt_count ;
Compteur conservant le nombre de montages ;
__s16s_max_mnt_count ;
Nombre de montages maximum autorisé avant de vérifier le système de fichier
__u16ss_magic ;
Nombre magique, égal à 0xef53 pour ext2/3.
__u16ss_state ;
Flag de statut, indiquant si le système de fichier est propre ou non. La fonction void print_state(__u16 s_state) ; permet de traduire la valeur : elle est égale à 0 si le système de fichier n'a pas été démonté correctement ; 1 si il est propre, 2 si il contient des erreurs, 4 si des inodes orphelins sont en train d'être récupérés.
__u16ss_errors ;
Indique quel est le comportement qui doit être utilisé en cas de détection d'erreurs. Si la valeur est 1, on continue l'execution ; 2, on remonte le système de fichiers en lecture seule ; 3, on déclenche un kernel panic.
__u16ss_minor_rev_level ;
Version minimum du système de fichier.
__u32ss_lastcheck ;
Timestamp de la dernière vérification.
__u32ss_checkinterval ;
Durée maximale entre les vérifications.
__u32ss_creator_os ;
OS créateur.
__u32ss_rev_level ;
Version du système de fichiers.
__u16ss_def_resuid ;
UID par défaut pour les *blocks* réservés. Certains blocks sont réservés pour le superutilisateur ; on peut par contre modifier l'UID auquel ils seront affectés par défaut.
__u16ss_def_resgid ;
GID par défaut pour les *blocks* réservés.
Ces champs sont seulement valables pour les superblocks de ext2/3 version 2 (avec taille d'inode dynamique).
__u32ss_first_ino ;
Numéro du premier inode libre.
__u16 s_inode_size ;
Taille des inodes.
__u16ss_block_group_nr ;
Numéro de block du superblock lu .
__u32ss_feature_compat ;
Carte (bitmap) des fonctionnalités compatibles.
Les cartes des fonctionnalités sont lisibles bit par bit (une position de bit correspond à une fonctionnalité : 0 signifie non supportée, 1 signifie supportée.)
0x0001 La partition supporte la préallocation pour les dossiers.
0x0002 La partition supporte les inodes magiques

0x0004 La partition supporte la journalisation.
0x0008 La partition supporte les attributs utilisateurs étendus.
0x0010 La partition supporte les inodes de taille variable.
0x0020 La partition est compatible avec la création d'index de dossier.

__u32ss_feature_incompat ;
Carte des fonctionnalités incompatibles
0x0001 La partition n'est pas compatible avec la compression 'transparente'.
0x0002 La partition n'est pas compatible avec les types de fichiers.
0x0004 La partition a besoin d'être réparée.
0x0008 La partition n'est pas compatible avec un journal externe.
0x0010 La partition n'est pas compatible avec les métadonnées des block groups.

__u32ss_feature_ro_compat ;
Carte des fonctionnalités compatibles en lecture seule
Le système de fichier peut être lu sans problème, mais une écriture sur celui ci risquerait de le corrompre.
0x0001 La partition supporte en lecture seule les fichiers troués.
0x0002 La partition supporte en lecture seule les fichiers de grande taille.
0x0004 La partition supporte en lecture seule les arborescences de types B-Arbres
(Pour plus d'informations, voir http ://fr.wikipedia.org/wiki/Arbre_B)

__u8s_uuid[16] ;
uuid du le volume.
chars_volume_name[16] ;
Nom du volume.
charss_last_mounted[64] ;
Dossier dans lequel la partition a été montée pour la dernière fois.
__u32ss_algorithm_usage_bitmap ;
Sert pour la compression 'transparente', c'est à dire la compression intégrée directement au système de fichier.
__u8ss_prealloc_blocks ;
Nombre de blocks à préallouer.
__u8ss_prealloc_dir_blocks ;
Nombre de blocks à préallouer pour les dossiers.
__u16ss_padding1 ;
Non affiché, utilisé pour avoir un 'bon' nombre d'octets dans le superblock.
__u8ss_journal_uuid[16] ;
uuid du superblock du journal.
__u32ss_journal_inum ;
Numéro d'inode du journal.
__u32ss_journal_dev ;
Numéro du périphérique sur lequel se trouve le journal.
__u32ss_last_orphan ;
Debut de la liste des inodes orphelins à effacer
__u32ss_hash_seed[4] ;
Graine de hachage HTREE.

__u8ss_def_hash_version ;
Version de hachage à utiliser par défaut.
__u8ss_reserved_char_pad ;
__u16ss_reserved_word_pad ;
Non affichés, servent à 'aligner' la structure du superblock.
__u32ss_default_mount_opts ;
Options de montage par défaut.
Les options de montage par défaut sont également lisibles bit par bit (une position de bit correspond à une option : 0 signifie non activée, 1 signifie activée.)
0x0001 Affichage des informations de debug à chaque montage.
0x0002 Le groupe ID par défaut d'un nouveau fichier est celui du répertoure dans lequel il se trouve.
0x0004 Support pour les attributs étendus (xattr) sur les utilisateurs.
0x0008 Support des Access Control List (ACL).
0x0010 Désactivation du mode 32-bits pour les UIDs et les GIDs (rétrocompatibilité avec les anciens kernels).
0x0060 La journalisation est activée. Signature commune avec le mode writeback ; si ni l'un ni l'autre des autres modes ne sont activés, cela signifie que le journal est en mode writeback.

0x0020 Le journal est utilisé en mode journalisation complète (données+informations administratives).
0x0040 Le journal est utilisé en mode ORDERED : similaire au mode writeback décrit ci dessous, mais le contenu des fichiers est écrit avant de journaliser les données administratives. C'est la valeur par défaut.
0x0060 Le journal est utilisé en mode WRITEBACK : Les données administratives (metadata) sont journalisées, tandis que le contenu du fichier ne l'est pas. Pour plus de détails, voir la section concernant le journal.
__u32ss_first_meta_bg ;
Premier block group pour les informations administratives.
__u32ss_reserved[190] ;
Non affiché, sert à finir de remplir le superblock.

2.2 ext3_group_desc

Le group descriptor contient des informations administratives sur le block group. Il est représenté par la structure ext3_group_desc.

Explication des champs du group descriptor :

__u32sbg_block_bitmap ;
Numéro de block du block bitmap
__u32sbg_inode_bitmap ;
Numéro de block de l'inode bitmap
__u32sbg_inode_table ;
Numéro de block de la première inode table

__u16sbg_free_blocks_count ;
Nombre de blocks libres dans le groupe
__u16sbg_free_inodes_count ;
Nombre d'inodes libres dans le groupe
__u16sbg_used_dirs_count ;
Nombre de dossiers dans le groupe
__u16sbg_pad ;
__u32sbg_reserved[3] ;
Parties inutilisées de la structure.

2.3 ext3_inode

L'inode contient des informations administratives liées à un "fichier" (fichier régulier, dossier, block device...), comme sa taille, ses droits d'accès, les blocks qu'il occupe...
Elle est représentée sur le disque par la structure ext3_inode.

Explication des champs d'une inode :

__u16 i_mode ;
"Mode" du fichier, indiquant son type (dossier, block device, fichier...), ses permissions, ses modes spéciaux(setuid, sticky bit)...

__u16 i_uid ;
16 bits de poids faible de l'ID du propriétaire de l'inode

__u32 i_size ;
Taille du fichier

__u32 i_atime ;
__u32 i_ctime ;
__u32 i_mtime ;
__u32 i_dtime ;
"Timestamps" d'accès, de création, de modification et de suppression.
__u16 i_gid ;
16 bits de poids faible du GID de l'inode

__u16 i_links_count ;
Nombre de "liens durs" (hard links) pointant vers cette inode

__u32 i_blocks ;
Nombre de blocks occupés par le fichier

__u32 i_flags ;
"Flags" du fichier, indiquant comment l'implémentation de ext3 doit gérer ce fichier (écriture en concaténation uniquement, suppression sécurisée...).
0x00000001 Le fichier doit être effacé de façon sécurisée.

0x00000002 Ce fichier peut être récupéré après effacement.
0x00000004 Ce fichier est compressé.
0x00000008 Ce fichier doit être mis à jour de façon synchrone.
0x00000010 Ce fichier ne peut être modifié.
0x00000020 L'écriture sur ce fichier se fait par ajout.
0x00000040 Ce fichier ne doit pas être 'dumped' ou effacé.
0x00000080 Le champ "dernier accès" (atime) de ce fichier ne doit pas être mis à jour.
0x00000100 Le fichier est "sale" (en cours d'utilisation ?)
0x00000200 Ce fichier est composé d'un ou plusieurs 'clusters'/blocks compressés.
0x00000400 Ce fichier ne doit pas être compressé.
0x00000800 Ce fichier a subi une erreur lors de la compression.
0x00001000 Dossier indexé par hachage.
0x00002000 Dossier AFS.
0x00004000 Les données de ce fichier doivent être journalisées.
0x80000000 Valeur réservée pour la bibliothèque ext3.

union osd1 ;
Champ dépendant de l'OS

__u32 i_block[EXT3_N_BLOCKS] ;
Pointeurs vers les blocks (numéro des blocks dans lesquels se trouvent le fichier, blocks directs, indirects, doublement indirects, triplement indirects)

__u32 i_generation ;
Version du fichier, sert pour NFS

__u32 i_file_acl ;

Access Control List du fichier. Si le champ n'est pas à 0, il contient un pointeur (le numéro du block) vers le header de l'ACL associée au fichier.

__u32 i_dir_acl ;
ACL du dossier. Si l'inode est réellement celle d'un dossier, le champ contient pointeur (le numéro du block) vers le header de l'ACL associée au dossier. Sinon, il contient les 32 bits de poids faible de la taille du fichier.

__u32 i_faddr ;
Adresse du fragment.

union osd2 ;
pour linux :
__u8 l_i_frag ;
__u8 l_i_fsize ;
__u16 i_pad1 ;
__u16 l_i_uid_high ;
__u16 l_i_gid_high ;
__u32 l_i_reserved2 ;
pour hurd : __u8 h_i_frag ;

__u8 h_i_fsize ;
__u16 h_i_mode_high ;
__u16 h_i_uid_high ;
__u16 h_i_gid_high ;
__u32 h_i_author ;
pour masix :
__u8 m_i_frag ;
__u8 m_i_fsize ;
__u16 m_pad1 ;
__u32 m_i_reserved2[2] ;
Ces champs dépendent de l'OS :
Numéro de fragment
Taille du fragment
Réservé/bits de poids fort du mode du fichier/Reservé
Bits de poids fort de l'UID/Bits de poids fort de l'UID/Réservé
Bits de poids fort du GID/Bits de poids fort du GID/Réservé
Réservé/Auteur/Réservé

2.4 Les *bitmaps*

L'inode bitmap indique, dans chaque block group, pour chaque inode, si elle est utilisé ou non. Il est représenté par un tableau de bits, sur au plus 1 block, dont la position est indiquée dans le group descriptor.

Le data block bitmap indique, dans chaque block group, pour chaque block si il est utilisé ou non. Il est représenté par un tableau de bits, sur au plus 1 block, dont la position est indiquée dans le group descriptor.

2.5 Le journal

Sous ext3, le journal est géré par le JBD (journaling block device) ; c'est une "couche" du noyau, actuellement utilisée par ext3 (et ext4), mais conçue pour être utilisable par d'autres systèmes de fichiers. Le JBD est donc "indépendant" de ext3.

Le journal est constitué de plusieurs parties : Le superblock du journal, une structure qui reste en permanence, et conserve des informations sur le journal ; Une zone dédiée à un "descripteur" du journal ; Une "partie" constituée des "transactions" (copie des blocks de données administratives ou de données de fichiers, selon le mode de journalisation.)

Le manque de documentation technique sur le JBD rend difficile son étude.

2.5.1 Les modes de journalisation

Mode de journalisation complète :
Les données et les métadonnées (données administratives) sont enregistrées par le journal avant d'être insérées dans le système de fichiers.

Mode ORDERED :
Mode par défaut : les données dont écrites dans le système de fichiers avant de journaliser (inscrire dans le journal) les métadonnées.

Mode WRITEBACK :
Les données peuvent être écrites dans le système de fichier après l'inscription des métadonnées dans le journal ; l'ordre des données n'est pas préservé.

Deuxième partie

Organisation de ext3Viewer

Chapitre 1

Organisation du code

1.1 Organisation des fichiers sources

L'organisation des fichiers sources est décrite dans cette partie. Une description est faite de l'utilité de chaque fichier *.c*. Les fonctions importantes de chaque fichier sont présentées.

1.1.1 filesystem.c

Ce fichier est composé de deux fonctions qui permettent l'ouverture et la fermeture du système de fichier. Elles permettent l'affichage d'un message d'erreur en cas de problème.

1.1.2 superblock.c

Dans ce fichier il y a les fonctions de lectures et d'affichage du superblock.

read_superblock()

La lecture du *superblock* nécessite quelques vérifications : le magic number doit correspondre à celui de ext3, et le système de fichier doit être démonté de préférence (le flag EXT3_FEATURE_INCOMPAT_RECOVER doit être à 0). Cette fonction vérifie ces conditions puis remplis la structure ext3_super_block.

read_superblock_backup()

Cette fonction permet de lire une copie du *superblock* situé ailleur que dans le premier groupe. On doit donc lui preciser l'adresse du *block* à lire.

print_superblock()

Affiche tous les champs du *superblock* en faisant la correspondance entre le contenu de chaque champs et leur signification.

print_sb_copy()

Cette fonction affiche la localisation des copies du *superblock*. Il existe trois manières de sauvegarder les copies du *superblock* selon les versions de ext3 : soit il n'y a pas de sauvegarde, soit il en existe une sauvegarde dans chaque groupe, soit il y en a une sauvegarde sur les groupes 0, 1 et sur les puissances de 3, 5 et 7.

1.1.3 groups.c

Dans ce fichier se trouve des fonctions en rapport avec les groupes de *blocks*.

read_inodebitmap()

Rempli un tableau de char qui doit contenir l'*inode bitmap*. Celui-ci doit être alloué dynamiquement avant l'appel de la fonction.

read_datablockbitmap()

Idem pour le *block bitmap*.

read_group_desc()

Si on lui passe le numéro d'un groupe, cette fonction remplit la structure *ext3_group_desc*.

print_groupdesc()

Affiche les champs de la structure *ext3_group_desc* et leur significations.

print_inodebitmap()

Affiche l'*inode bitmap* d'un groupe, représentée par un tableau de bits qui indiquent si un *inode* est utilisé ou non, 0 signifiant que l'*inode* est libre et 1 signifiant qu'il est utilisé.

1.1.4 inode.c

read_inode() et print_inode()

Fonctions de lecture et d'affichage d'un *inode*.

print_file()

Cette fonction permet d'afficher le contenu d'un fichier, soit en ascii, soit en hexadecimal. Pout lire les données associées à un *inode*, la fonction récursive print_blocks() est utilisée. On lit les 13 premiers *blocks* associés à l'*inode* directement. Puis pour les indices 12, 13, et 14 du tableau i_block[] ; on doit les considérer comme des tableaux de *blocks* (*blocks* indirects, doublement indirect et triplement indirect).

print_symlink

Le contenu d'un *symlink* doit être considéré différemment de celui d'un fichier regulier. En effet, si le chemin pointé par le *symlink* fait moins de 60 caractères, le tableau i_block[] est entièrement utilisé pour stocker les données (on utilise pas de *blocks* indirects). Au contraire, si la taille du chemin pointé est supérieur à 60 caractères, on l'affiche de la même façon qu'un fichier régulier.

print_dir()

Pour afficher le contenu d'un répertoire, il faut lire les *dir entries* qui composent les données de l'*inode* du répertoire. Pour se déplacer d'une *dir entry* à l'autre, *rec_len* octets doivent êtres sautés. Cette fonction récursive est dédiée à ce travail.

1.1.5 block.c

read_block()

Cette fonction remplit un tableau de caractères avec le contenu d'un *block* de données. Le *block* peut être de différentes tailles suivant le système de fichier : 1024, 2048 ou 4096 octets.

read_block_pointer()

Même fonction que la précédente, cependant on y remplit un tableau de __u32. Ceci va nous permettre de lire ce tableau et de connaître les *blocks* de données à partir d'un *block* indirect.

print_blocks()

Fonction récursive qui affiche tout le contenu d'un fichier, c'est à dire tous les *blocks* alloués pour un *inode* donné.

1.1.6 search.c

ext3Viewer permet à l'utilisateur de rechercher un fichier, pour cela il peut procéder de deux manières : rechercher par le nom de fichier ou par son numéro d'inode.

Rechercher un fichier par son nom :
Nous utilisons pour la recherche par nom de fichier la syntaxe *extended regex*. La fonction search_file() cherche à partir du répertoire (donné comme argument) dans tous les dossiers enfants, un fichier qui correspond à un pattern regex.

Pour rechercher un fichier on utilise trois fonctions : search_file(), search_file_rec() et search_file_rec_block().

Des fonctions similaires existent pour chercher un fichier par son numéro d'*inode*.

1.1.7 path.c

Ce fichier contient un ensemble de fonctions utilisées pour empiler le chemin d'un fichier dans les fonctions de recherche récursives.

1.1.8 journal.c

Ce fichier contient les fonctions de lectures et d'affichage des informations liées au journal.

read_journal_superblock()

Fonction de lecture du *journal superblock*. Cette fonction prend comme un de ses arguments le numéro de *block* du premier *block* de données du journal (inode 8).

print_journal_superblock()

Cette fonction affiche les champs du *journal superblock* et leur correspondance avec les informations de */usr/include/linux/jbd.h*.

read_journal_header()

Lit un en-tête d'un *block* du journal.

dump_journal()

Affiche le contenu du journal : les *blocks* de la partitions en relation avec le journal, et les *flags* expliquant l'action effectuée.

1.1.9 acl.c

read_xattrh()

Fonction utilisée pour lire le header des *extended attributes*.

read_xattre()

Lecture d'une entrée d'*extended attribute*.

print_acl()

Affichage des *extended attributes* [1]

1.1.10 main.c

Ce fichier fait l'interface entre les demandes de l'utilisateur en mode console, et l'appel des fonctions correspondantes (affichage de la syntaxe, affichage de message d'erreur ...).

[1] La partie dédiée à la gestion des *Access Control Lists* n'est pas opérationnelle.

1.1.11 text.h

Cet en-tête contient les textes utilisés en mode console pour afficher le rappel des fonctionalités et l'aide qui explique leurs utilités respectives.

1.1.12 debug.h

Définition de quelques macros pour le débuggage du logiciel. Nous avons défini des macros DEBUG_PRINT et DEBUG_PRINT_V qui permettent de débugger le programme. Elles peuvent être activées à la compilation grâce aux options -DDEBUG et -DDEBUG_VERBOSE à rajouter dans le *Makefile* comme *CFLAGS*.

1.2 Divers

1.2.1 Gestion des erreurs

Pour le mode console du programme, les messages d'erreur sont écrits sur *stderr*, la sortie standard d'erreur sous Linux. De plus, lorsqu'un erreur survient, la valeur 1 est renvoyée.
Pour la partie graphique les erreurs sont signalées par un pop-up.

Annexe A

Bibliographie

Voici une liste non-exhaustive des livres et publications conseillées pour la compréhension du système de fichier ext3 :

Understanding the Linux Kernel, chapitre 17, *O'REILLY* : explication générale sur l'organisation du système de fichier ext2

`http://www.suse.de/~agruen/acl/linux-acls/online/` : documentation concernant les ACL.

`http://kerneltrap.org/node/6741` : informations sur le journal jbd.
`http://lxr.linux.no/` :
Sources du noyau pour plusieurs versions, avec des possibilités de comparaison.
`http://www.nongnu.org/ext2-doc/ext2.html` :
Documentation technique concernant ext2. Un bon nombre des informations restent valable pour ext3.