Croisière au cœur d'un OS*
Étape "8" : Système de fichiers virtuel (*VFS*)

Résumé

Jusqu'à maintenant, nous nous sommes concentrés sur les éléments fondamentaux et internes d'un OS, ce qui nous permet de lancer nos premiers programmes utilisateur. À partir de cet article, nous nous tournons davantage vers les applications utilisateur en leur permettant de disposer d'un environnement riche pour interagir avec le disque dur, le clavier, etc... Sous Unix, la plupart de ces interactions a lieu dans un cadre unique : le Système de Fichiers Virtuel (VFS). C'est ce que nous étudions ce mois ci.

Introduction

Les précédents articles de la série SOS ont permis de détailler l'implémentation d'un système d'exploitation minimal. En terme de gestion de la mémoire et de gestion des tâches, ce petit système dispose des fonctionnalités de base d'un système d'exploitation : gestion de la mémoire physique, gestion de la pagination, *threads* noyau et utilisateur, processus, gestion de la mémoire virtuelle et appels système minimaux.

Toutefois, nous n'avons pas évoqué jusqu'ici un sous-système fondamental d'un système d'exploitation complet : le sous-système s'occupant des fichiers et des systèmes de fichiers. Cet article s'intéresse en particulier au *Virtual File System*, l'infrastructure permettant l'implémentation des pilotes des différents systèmes de fichiers et la représentation des fichiers de manière uniforme au sein d'une arborescence unique. L'implémentation que nous proposons dans SOS est assez similaire à celle du noyau Linux [1] [2].

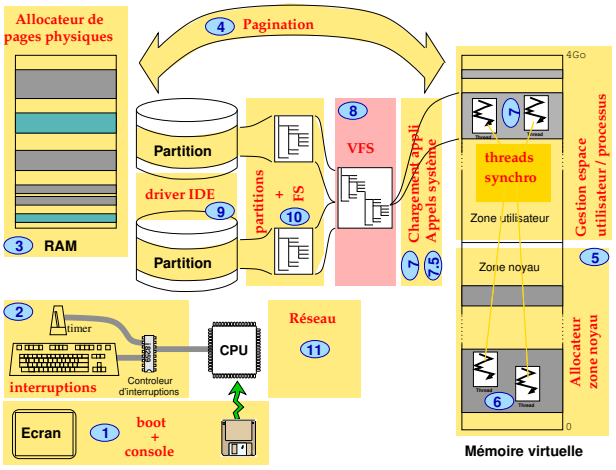

FIG. 1 – Programme des articles

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 74 – Juillet/Août 2005 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

Comme lors du précédent article, vous ne trouverez pas le code de SOS sur le CD accompagnant le magazine. Il sera disponible début Juillet sur le site de SOS : `http://sos.enix.org`.

1 Présentation générale

1.1 Fichier, point de vue utilisateur

Un des concepts fondateurs des systèmes de type Unix est *" Tout est fichier "*. L'idée de base est de représenter toutes les ressources du système sous la forme de fichiers, de manière à pouvoir leur appliquer des traitements similaires à l'aide de programmes simples.

En général (exception faite de *sockets* réseau sur la plupart des Unix), un fichier est identifié à l'aide d'un (ou plusieurs) *chemin(s)* au sein d'une arborescence unifiée (par exemple `/home/toto/fichier` sous Unix, ou `C:\Documents and settings\toto` sous Microsoft Windows). L'ensemble des systèmes de fichiers est accessible au sein de cette arborescence. Un programme utilisateur peut *ouvrir* des fichiers à l'aide de ce chemin, puis peut ensuite manipuler celui-ci en utilisant pour identifiant le *descripteur de fichier* qui lui a été fourni lors de cette ouverture.

1.2 Système de fichiers

On distingue sous Unix trois types de systèmes de fichiers :

- les systèmes de fichiers *normaux*, dans lesquels un fichier correspond à des données stockées sur un support de masse comme un disque dur, une disquette, une clé USB ou un CD-ROM. Ext2, Ext3, ReiserFS, VFAT, NTFS sont des systèmes de fichiers de ce type. Ils définissent chacun une organisation particulière des données sur le support de masse qui leur permet de créer, lire, écrire et détruire des fichiers ou répertoires ;
- les systèmes de fichiers *virtuels*, dans lesquels chaque fichier correspond à une abstraction spécifique du système : une liste d'informations, un périphérique, etc. Le fichier ne contient alors pas de données stockées sur un support de masse. `/proc` ou *sysfs* sous Linux sont des exemples de systèmes de fichiers de ce type.
- les systèmes de fichiers *réseau*, intermédiaires entre les deux types précédents. En effet, dans ces systèmes de fichiers, les fichiers ne sont pas stockés sur un support de masse local (ce qui en fait une forme de système de fichiers virtuels), mais sur des supports de masses dans des ordinateurs distants. NFS, CIFS, AFS ou Coda sont de tels systèmes de fichiers. Nous oublierons ce type de systèmes de fichiers dans toute la suite.

Chaque système de fichiers implémente un jeu d'opérations standardisées telles que l'ouverture, la

création, l'écriture, la lecture, la suppression d'un fichier ou d'un répertoire. Ce jeu d'opérations cache le fonctionnement interne du système de fichiers : lorsqu'on se place au dessus de ces derniers, rien ne ressemble plus à un système de fichiers qu'un autre système de fichiers.

1.3 Objectifs du *VFS*

Pour permettre le fonctionnement de ces multiples systèmes de fichiers, les systèmes Unix modernes (à partir de System V Release 4), tout comme SOS, utilisent un sous-système appelé *Virtual Filesystem* ou *VFS*. Ce dernier a plusieurs objectifs :

- fournir aux applications utilisateur une interface de programmation simple et uniforme, quel que soit le système de fichiers gérant les fichiers accédés. Cette interface est fournie sous la forme d'appels système tels que `open`, `read`, `write`, `close`, `seek`, etc ;
- factoriser l'implémentation des différents systèmes de fichiers. En effet, tous les systèmes de fichiers utilisent des abstractions similaires et fonctionnent souvent de manière proche. Le *VFS* factorise l'ensemble du code commun pour ne laisser dans les pilotes de système de fichiers que le code propre au système de fichiers considéré ;
- unifier l'arborescence, pour représenter au sein d'un même arbre tous les fichiers du système, quels que soient les systèmes de fichiers sur lesquels ils se trouvent.

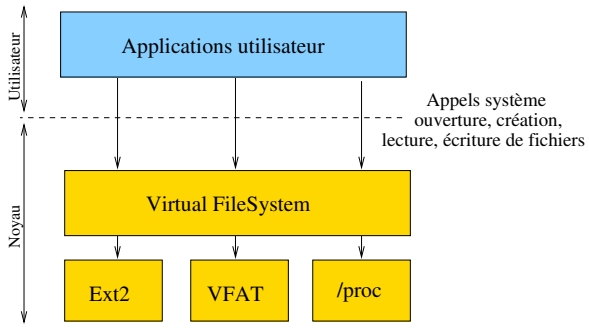

FIG. 2 – Le *Virtual FileSystem* implémente les différents appels système d'accès aux fichiers et fait l'interface avec les différents systèmes de fichiers

1.4 Une implémentation *"orientée objet"*

L'implémentation d'un sous-système tel que le *VFS* repose sur des mécanismes plus simples que ceux mis en œuvre dans les articles précédents autour de la gestion des tâches et de la mémoire. Dans le cas du *VFS*, le plus important est de définir les structures de données utilisées, l'implémentation de ce dernier étant ensuite relativement simple.

Le *VFS* de SOS, comme celui du noyau Linux, est architecturé à l'aide d'objets, constitués de données et de méthodes, comme en programmation orientée objet. Bien entendu, nous n'utilisons pas de langage prévu pour l'objet, mais nous utilisons le C avec quelques idées de ce paradigme de programmation. La structure fondamentale de ces objets est définie par le VFS et est donc identique quel que soit le système de fichiers utilisé. Mais elle peut être étendue par chaque implémentation de système de fichiers (relation objet type héritage ou association).

Le *VFS* implémente tous les appels système classiques de gestion des fichiers (`mount`, `umount`, `read`, `write`, `seek`, etc.) en manipulant ces objets. Chaque objet est lié à un jeu d'opérations qui diffère en fonction du système de fichiers sous-jacent. Dans l'esprit de la programmation orientée objet, le premier paramètre de chaque opération est toujours un pointeur sur l'objet sur lequel celle-ci s'applique. On peut assimiler ceci au pointeur `this` des langages orientés objet.

Par exemple, de manière simplifiée, lorsqu'une application utilisateur réalise un appel système `read` sur un système de fichiers *ext2*, le *VFS* recherche le fichier correspondant, puis appelle l'opération `read` de celui-ci. Le fichier appartenant à un système de fichiers *ext2*, celle-ci sera implémentée par le pilote de ce système de fichiers.

Pour permettre le fonctionnement du *VFS* avec les systèmes de fichiers sous-jacents, cinq objets importants sont définis, aussi bien dans SOS que Linux :

- un objet qui décrit un type de système de fichiers (*ext2*, */proc*, *vfat*). Cet objet contient notamment deux opérations permettant de *monter* et *démonter* un système de fichiers. Sous Linux, cette structure s'appelle `struct file_system`. Sous SOS, ce sera `sos_fs_manager_type` ;
- un objet qui décrit un "montage" particulier d'un système de fichiers, c'est-à-dire une instance particulière d'un système de fichiers. Un objet de ce type est créé à chaque *montage* et détruit à chaque *démontage*. Sous Linux, cette structure s'appelle `struct superblock`. Sous SOS, ce sera `sos_fs_manager_instance` ;
- un objet qui représente un fichier, que ce soit un fichier réel, un répertoire, un lien symbolique ou un périphérique. Cette structure ne contient pas d'information sur la localisation du fichier dans l'arborescence, elle contient surtout les informations relatives à ce fichier telles qu'elles sont stockées sur le disque. Sous Linux, cette structure s'appelle `struct inode`. Sous SOS, ce sera `sos_fs_node` ;
- un objet qui représente un fichier au sein de l'arborescence globale, c'est-à-dire associant un nom à un fichier. En général, un objet `sos_fs_node` est associé à un unique objet de ce type, sauf dans le cas des liens durs (*hard links*). En effet, les liens durs permettent précisément d'avoir un même fichier présent en différents endroits de l'arborescence, c'est-à-dire identifié par plusieurs chemins différents. Cet objet contient le nom du fichier, un pointeur vers le fichier associé, une référence à son parent et une liste de ses fils (pour les répertoires). Sous Linux, cette structure s'appelle `struct dentry`[1]. Sous SOS, ce sera `sos_fs_namespace_node` ;
- un objet qui contient les informations propres à l'ouverture d'un fichier donné par un processus donné. Par exemple, pour l'ouverture d'un même fichier, chaque processus dispose d'un pointeur courant de lecture/écriture (manipulé par `seek`/`read`/`write`). Chaque processus dispose d'un tableau de pointeurs vers de tels objets (tableau `sos_process::fds[]`

[1] En réalité, sous Linux, le lien vers le parent n'est pas distingué des autres liens vers les fils (c'est un fils dont le nom est toujours "..."). C'est d'ailleurs une des différences importantes avec le modèle de SOS.

dans `sos/process.h`). L'index dans ce tableau forme le *descripteur de fichier*, manipulé par les applications utilisateur. Cette structure contient donc le "pointeur courant" du processus dans le fichier ainsi qu'une référence vers le `struct sos_fs_nscache_node` associé. Sous Linux, elle s'appelle `struct file`. Sous SOS, ce sera `sos_fs_opened_file`.

L'ensemble du *VFS* s'architecture autour de ces cinq objets, en s'appuyant sur l'implémentation de chaque système de fichiers pour les manipuler, comme décrit en figure 3. Cette architecture est très proche de celle de Solaris et Linux.

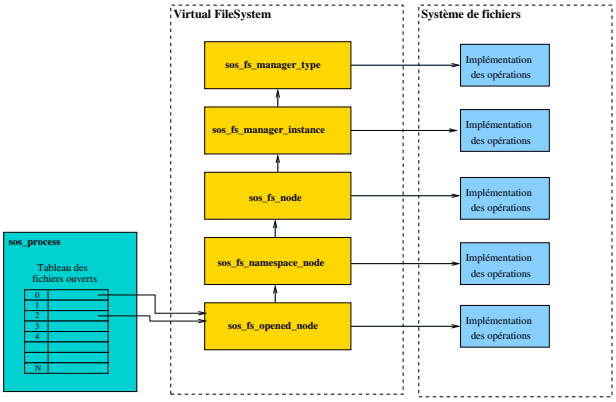

FIG. 3 – Le *Virtual FileSystem* gère les différents objets et effectue les opérations demandées par les applications utilisateur en utilisant les opérations implémentées par chaque pilote de système de fichiers.

2 Implémentation dans *SOS*

La figure 4 présente les structures de données du *VFS* que nous décrivons dans la suite. Les structures de données en jaune sont celles directement gérées par le cœur du *VFS* (`sos/fs.h` et `sos/fs.c`). Celle en jaune foncé est gérée par le sous-système *namespace cache* (`sos/fs_nscache.h` et `sos/fs_nscache.c`). Enfin, les structures en vert sont les opérations liées à chaque objet, qui sont implémentées dans chaque pilote de système de fichiers (par exemple `drivers/fs_virtfs.h` et `drivers/fs_virtfs.c`).

2.1 Cache de l'espace de nommage

2.1.1 Principe

Un des rôles du *VFS* est d'unifier l'espace de nommage, c'est-à-dire l'arborescence des fichiers, de manière à ce que tous les systèmes de fichiers soient représentés dans un espace unique. Ainsi, dans les systèmes Unix, un système de fichiers principal est monté à la racine `/` et les autres systèmes de fichiers sont montés dans des sous-répertoires, le tout formant une unique arborescence.

Dans SOS comme dans Linux, l'objet du système de fichiers (fichier, répertoire, périphérique...) est découplé de sa position dans l'arborescence. En effet, un même nœud peut être présent à plusieurs endroits de l'arborescence dans le cas de liens *durs* (*hard links*). C'est la raison pour laquelle on dispose dans SOS des objets `sos_fs_node` (représentant un objet du système de fichiers, en général stocké sur disque) et `sos_fs_nscache_node` (représentant un objet du système de fichiers au sein de l'arborescence).

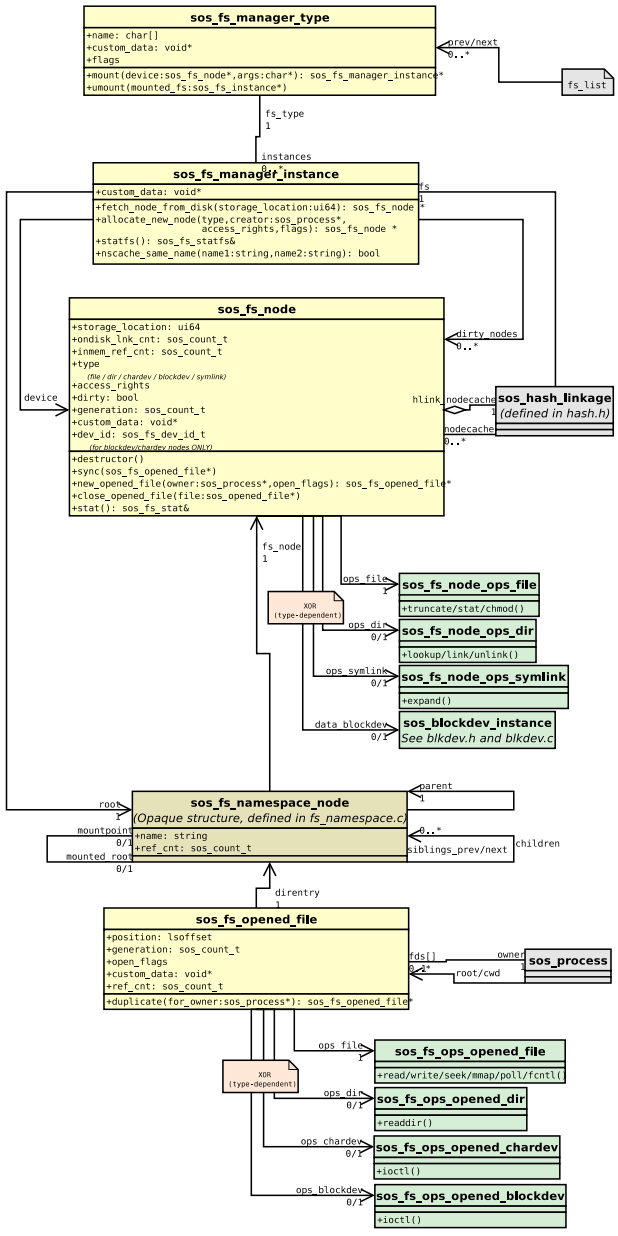

FIG. 4 – Représentation UML des structures de données du em VFS

Les objets `sos_fs_nscache_node` sont liés entre eux comme dans une arborescence : chaque nœud possède un parent et peut posséder plusieurs fils. Ainsi, l'arborescence de ces objets constitue une représentation **partielle** de la hiérarchie réelle des fichiers et répertoires. Au fur et à mesure de l'ouverture de nouveaux fichiers, cette représentation partielle de l'arborescence s'enrichira. Cette représentation permet notamment d'accélérer l'ouverture de fichiers précédemment ouverts, car il n'est pas nécessaire d'utiliser à chaque fois le disque pour retrouver un fichier. L'arborescence des `sos_fs_nscache_node` joue donc un rôle de *cache*, le *nscache* ou *namespace cache*. Dans Linux, les `struct dentry` analogues servent aussi de cache. Mais ils ne "cachent" pas une arborescence partielle, ils cachent des nœuds pris individuellement sans se soucier de conserver les liens entre ces nœuds.

2.1.2 Structure de données

Ce *cache* est géré par un sous-système séparé du cœur du *VFS*, implémenté dans le fichier `sos/fs_nscache.c`. La structure `sos_fs_nscache_node` y est définie : son contenu est invisible depuis le cœur du *VFS*. Ce dernier doit utiliser un jeu de fonctions limité, défini dans `sos/fs_nscache.h` pour maintenir et utiliser ce cache.

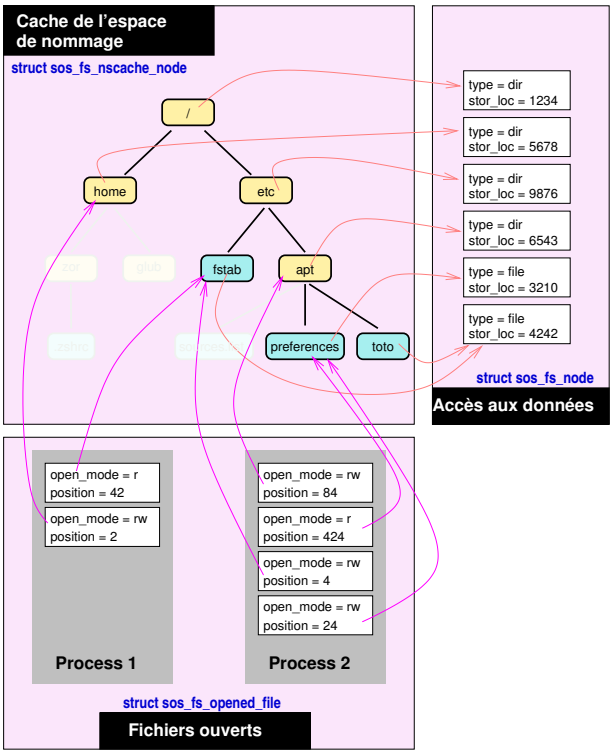

FIG. 5 – Les `sos_fs_nscache_node` forment une arborescence qui est une représentation partielle en mémoire de l'arborescence des fichiers. Ici, les objets du système de fichiers en clair ne sont pas dans le cache. Chaque `sos_fs_nscache_node` est lié à un `sos_fs_node` qui représente le fichier ou répertoire sous-jacent. De plus, au sein des différents processus, le tableau des fichiers ouverts contient des structures `sos_fs_opened_file` qui font référence aux `sos_fs_nscache_node`.

```
struct sos_fs_nscache_node
{
  struct sos_fs_node     *fs_node;
  struct sos_fs_pathname name;

  sos_count_t ref_cnt;

  struct sos_fs_nscache_node *mounted_root;
  struct sos_fs_nscache_node *mountpoint;

  struct sos_fs_nscache_node *parent;

  struct sos_fs_nscache_node *children;

  struct sos_fs_nscache_node *siblings_prev, *siblings_next;
};
```

Cette structure `sos_fs_nscache_node` comporte tout d'abord une référence vers le `sos_fs_node` qu'elle représente dans l'arborescence (champ `fs_node`). Le champ `name` contient le nom du nœud dans l'arborescence, par exemple `toto` si le nœud `/home/test/toto` est représenté. Pour stocker les chemins, nous utilisons une structure spécifique dans l'ensemble de SOS, la structure `sos_fs_pathname` qui contient un pointeur vers la chaîne de caractères ainsi que sa taille. Cette structure est définie dans `sos/fs_fsnscache.h`.

Le compteur de références `ref_cnt` permet de compter le nombre d'utilisations d'un *nœud nscache* et ainsi de le libérer lorsqu'il devient inutile. Enfin, le champ `parent` pointe tout naturellement vers le nœud parent dans l'arborescence, et `children` est la liste des nœuds fils dans l'arborescence. Les champs `siblings_prev` et `siblings_next` permettent de chaîner les nœuds voisins dans la liste des fils du nœud parent.

Les champs `mountpoint` et `mounted_root` permettent de gérer les points de montage. Le champ `mounted_root` d'un nœud pointe sur le nœud racine d'un sous-système de fichiers monté. Le champ `mountpoint` du nœud racine d'un sous-système de fichiers pointe sur le nœud sur lequel ce sous-système de fichiers est monté (voir figure 6).

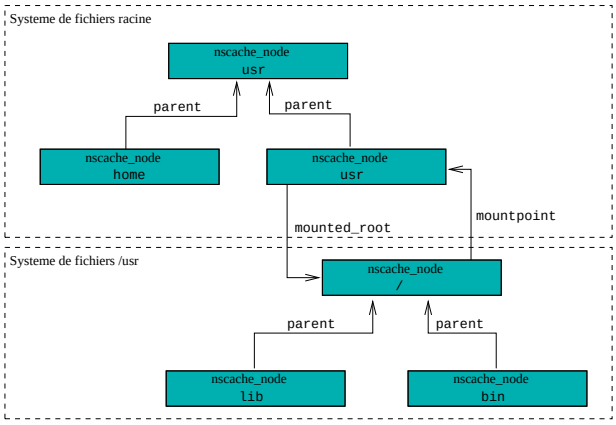

FIG. 6 – Le système de fichiers contenant `/usr` est monté dans le répertoire `usr` du système de fichiers racine.

On peut éventuellement monter plusieurs systèmes de fichiers sur le même point de montage, auquel cas le dernier système de fichiers monté recouvre tous les autres. Dans ce cas, plusieurs `mounted_root` se chaînent, et c'est le dernier de la chaîne qui est le système de fichiers auquel il faut accéder.

[2]Les liens durs sur les répertoires sont interdits sous Linux, le lecteur linuxien transposera facilement cette illustration au cas des liens durs sur des fichiers.

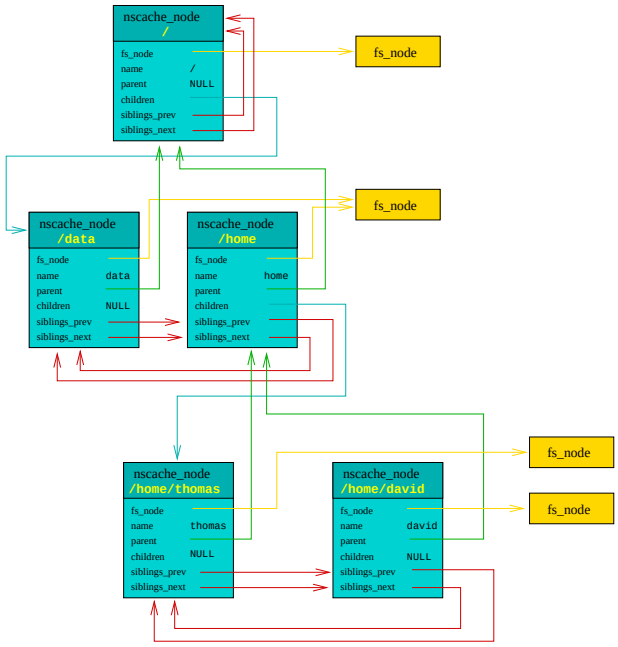

FIG. 7 – Un exemple de *nscache*, avec les nœuds `/`, `/data`, `/home`, `/home/thomas`, `/home/david`. `data` et `home` pointent sur le même : l'un est donc un lien *dur* sur l'autre [2].

2.1.3 Fonctions

Manipulation des chemins. Tout d'abord, quelques fonctions de manipulation des chemins de fichiers (représentés par la structure `sos_fs_pathname`) sont proposées dans le fichier `sos/fs_nscache.c`. Elles permettent de factoriser les traitements laborieux sur les chaînes de caractères. La structure `struct sos_fs_pathname` se rapproche d'une chaîne de caractères dans le langage `Pascal`. Elle stocke simultanément l'adresse de la chaîne de caractères et sa longueur. Cette représentation permet d'éviter d'avoir à faire des recopies de chaînes de caractères quand il s'agit de les découper en composantes. Par exemple, pour isoler la première composante "`home`" dans la chaîne "`/home/toto`", il suffit de dire que cette composante commence au deuxième caractère dans "`/home/toto`" et mesure 4 caractères. Ainsi plusieurs `sos_fs_pathname` peuvent pointer sur des sous-ensembles d'une même chaîne de caractères en mémoire.

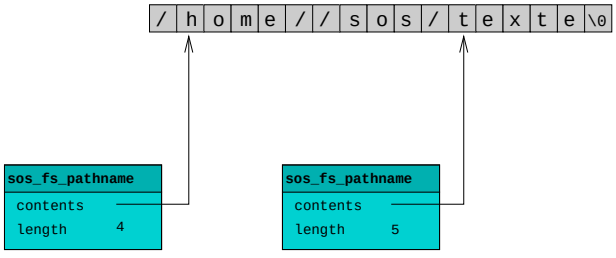

FIG. 8 – Deux `sos_fs_pathname` pointent sur deux sous-chaînes d'une même chaîne de caractères en mémoire. Ainsi, il n'est pas nécessaire d'effectuer de multiples allocations mémoire pour chaque sous-chaîne

La première fonction, `sos_fs_pathname_eat_slashes` a pour objectif de *manger* tous les `/` au début du chemin de fichier. Ainsi, elle transforme `/////home////sos` en `home////sos`. La deuxième fonction, `sos_fs_pathname_eat_non_slashes` réalise l'opération symétrique : retirer tous les caractères jusqu'au prochain `/`. Ainsi, elle transforme `home////sos` en `////sos`. La fonction `sos_fs_pathname_split_path` découpe un chemin en deux parties : le premier élément du chemin et le reste du chemin. Ainsi, le chemin `home///sos//texte.txt` sera découpé en `home` et `sos//texte.txt`. Enfin, la dernière fonction, `sos_fs_pathname_iseq` permet de tester si deux chemins sont égaux.

Accesseurs. La structure `sos_fs_nscache_node` n'étant définie que dans le fichier `sos/fs_nscache.c`, il est nécessaire de disposer de quelques accesseurs que le cœur du *VFS* pourra utiliser. Ainsi, `sos_fs_nscache_get_fs_node`, `sos_fs_nscache_get_parent` et `sos_fsnscache_get_name` permettent respectivement d'accéder au `fs_node` correspondant au `nscache_node`, au parent d'un `nscache_node` et au nom d'un `nscache_node`. La fonction `sos_fs_nscache_has_children` permet de savoir si un `nscache_node` a des fils.

Recherche dans l'arborescence. La fonction `sos_fs_nscache_lookup` est la fonction la plus importante du *namespace cache*. C'est en effet celle qui permet de retrouver des `nscache_node` dans l'arborescence. Elle recherche un `nscache_node` à partir d'un `nscache_node` donné et d'un nom donné : la recherche est donc limitée à un seul niveau dans l'arborescence.

Cette fonction traite trois cas :

- Le nœud recherché a le nom `''.''`, c'est-à-dire qu'il correspond au nœud courant. Ce dernier est donc tout simplement retourné ;
- Le nœud recherché a le nom `''..''`, c'est-à-dire qu'il correspond au nœud parent. Deux cas sont possibles ici : **1)** le nœud courant est la "sous-racine" indiquée par le paramètre `root_nsnode` passé à la fonction, auquel cas on ne remonte pas au parent, on retourne tout simplement le nœud courant. Cette sous-racine peut être soit la racine globale du système de fichiers global, soit une racine propre au processus appelant (i.e. celui-ci est "chroot"é : il avait appelé `chroot`). Ou **2)**, le nœud a un parent, auquel cas on remonte au parent, puis on "remonte" la chaîne des points de montage si ce nœud est un point de montage. Ceci permet de revenir au système de fichiers parent lorsqu'on se trouve au niveau d'un point de montage ;
- Dans les autres cas, on parcourt la liste des enfants du nœud courant à la recherche d'un nœud ayant le nom spécifié. Si le système de fichiers propose une opération de comparaison des noms `nsnode_same_name` alors elle est utilisée (certains systèmes de fichiers ne font par exemple pas la distinction entre majuscules et minuscules), sinon, c'est la fonction `fs_pathname_iseq` étudiée plus haut qui est utilisée.

Une fois le nœud trouvé, on "descend" les points de montage pour trouver le nœud réel. Ainsi, si l'on est au niveau d'un point de montage, on aura le nœud principal du système de fichiers fils, et non le répertoire du système de fichiers parent dans lequel est monté le système de fichiers fils.

Comptage de références. Chaque `nscache_node` possède un compteur de références permettant de savoir quand il peut être libéré.

La fonction `sos_fs_nscache_ref_node` incrémente simplement ce compteur de références. Elle est appelée par le sous-système *namespace cache* ou par le cœur du *VFS* dès qu'un `nscache_node` est utilisé.

La fonction `sos_fs_nscache_unref_node` est un peu plus complexe. En effet, en plus de décrémenter le compteur de références du `nscache_node` donné, elle essaie de libérer les `nscache_node` inutilisés. Pour cela, elle remonte de parent en parent, et si le compteur de références de ces nœuds tombe à zéro, ils seront libérés. Cette procédure se passe en deux étapes : **1)** construction de la liste des `nscache_node` à libérer en remontant de parent en parent, **2)** libération effective de tous les nœuds de la liste. L'utilisation de la liste intermédiaire des `nscache_node` à supprimer est nécessaire pour éviter toute récursivité de la fonction, dangereuse dans du code noyau (voir l'article 6).

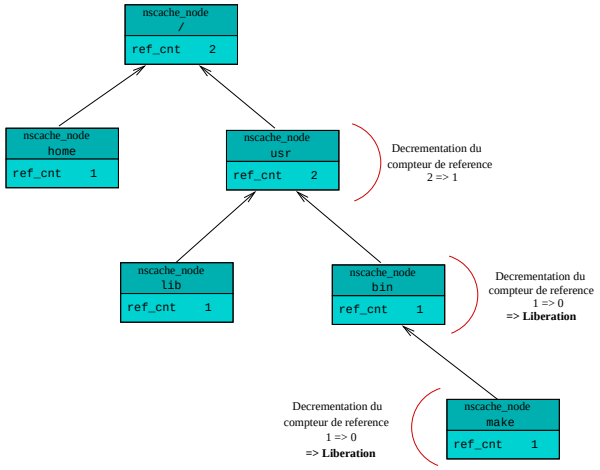

FIG. 9 – L'appel de `sos_fs_nscache_unref_node` sur le nœud `make` entraîne la décrémentation des compteurs de références des parents et la libération des nœuds devenus inutiles

Autres fonctions. La fonction `sos_fs_nscache_add_new_child_node` permet de créer un nouvel `nscache_node` en tant que fils d'un `nscache_node` donné. Ce nouveau `nscache_node` est créé avec le nom et le `fs_node` passés en paramètre. À l'inverse, la fonction `sos_fs_nscache_disconnect_node` permet de retirer un nœud de l'arborescence.

La fonction `sos_fs_nscache_register_opened_file` permet d'enregistrer un nouveau fichier ouvert pour un `nscache_node` donné. Elle est appelée à chaque fois qu'un fichier est ouvert.

Enfin, les fonctions `sos_fs_nscache_mount` et `sos_fs_nscache_umount` permettent de manipuler les champs `mounted_root` et `mountpoint` que nous avons vus plus haut. Avant de procéder au démontage, la fonction `sos_fs_nscache_umount` vérifie que le compteur de références de la racine est bien à 0, c'est-à-dire que tous les nœuds du système de fichiers ne sont plus utilisés. En effet, si un nœud était utilisé, même dans un sous-répertoire, alors une arborescence partielle existerait de la racine du système de fichiers vers ce nœud. Tous les nœuds de cette arborescence, y compris le nœud racine, auraient donc un compteur de références non nul puisqu'ils posséderaient tous au moins un nœud fils.

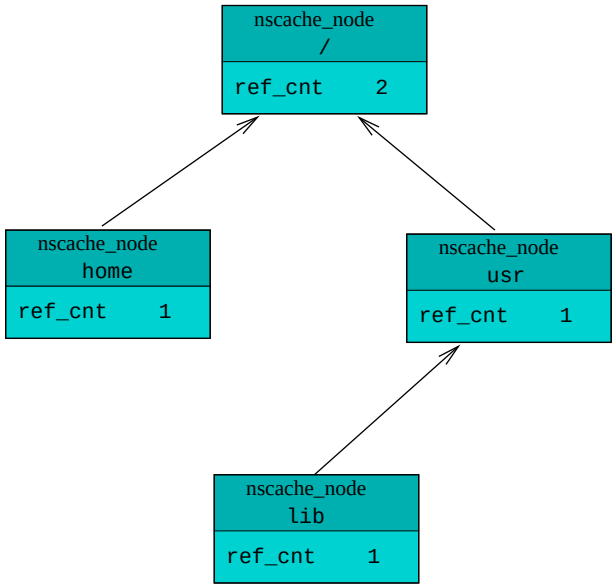

FIG. 10 – État de l'arbre des `nscache_node` après l'appel de `sos_fs_nscache_unref_node` sur le nœud `make` dans la figure 9.

2.2 Cœur du *VFS*

2.2.1 Pilote de système de fichiers

Chaque pilote de système de fichiers de SOS est défini par un objet de type `sos_fs_manager_type` et doit s'enregistrer auprès du *VFS* afin de créer un système de fichiers "monté" `struct sos_fs_manager_instance`. La structure `sos_fs_manager_type` renseigne le nom du système de fichiers, une opération *mount* et une opération *umount*. En plus de ceci, le pilote de système de fichiers peut stocker dans cette structure un pointeur vers des données privées (champ `custom_data`). Il peut l'utiliser comme il le souhaite dans l'implémentation des opérations *mount* et *umount*.

```
struct sos_fs_manager_type
{
  char name[SOS_FS_MANAGER_NAME_MAXLEN];

  sos_ret_t (*mount)(struct sos_fs_manager_type * this,
                     struct sos_fs_node * device,
                     const char * args,
                     struct sos_fs_manager_instance ** mounted_fs);

  sos_ret_t (*umount)(struct sos_fs_manager_type * this,
                      struct sos_fs_manager_instance * mounted_fs);

  void * custom_data;

  struct sos_fs_manager_instance * instances;

  struct sos_fs_manager_type *prev, *next;
};
```

La structure `sos_fs_manager` contient également des informations qui sont gérées uniquement par le *VFS* en interne. Ainsi, le *VFS* maintient la liste des montages du système de fichiers considéré, c'est-à-dire une liste de structures `sos_fs_manager_instance`. D'autre part, des liens `next` et `prev` permettent au *VFS* de maintenir une liste globale des pilotes de systèmes de fichiers disponibles.

Pour s'enregistrer auprès du *VFS*, le pilote de système de fichiers doit utiliser la fonction

`sos_fs_register_fs_type`. Le *VFS* ajoute simplement ce nouveau pilote à la liste globale (`fs_list` dans `sos/fs.c`) des pilotes de systèmes de fichiers disponibles. Symétriquement, un pilote de système de fichiers peut se "désenregistrer" du *VFS* en utilisant `sos_fs_unregister_fs_type`.

Dès lors qu'un pilote de système de fichiers est enregistré dans le *VFS*, on peut *monter* un système de fichiers géré par ce pilote.

2.2.2 Montage et démontage d'un système de fichiers

Pour monter un système de fichiers, le *VFS* propose la fonction `sos_fs_mount`. Celle-ci prend notamment un chemin *source*, un chemin *destination* et un nom de système de fichiers en paramètre (`"ext2"`, ...). Le *chemin source* désigne simplement un nœud de l'arborescence : le *VFS* ne se préoccupe pas de sa nature. Le *chemin destination* désigne l'endroit où sera monté le système de fichiers et doit nécessairement être un répertoire. Pour le montage, le *VFS* cherche tout d'abord, dans la liste des systèmes de fichiers enregistrés, la structure `sos_fs_manager_type` dont le nom est passé en paramètre. Ensuite, il recherche les chemins *source* et *destination*, avant d'utiliser l'opération *mount* du pilote de système de fichiers pour réaliser le montage proprement dit.

Cette opération prend notamment en paramètre le nœud du système de fichiers correspondant au chemin *source* et retourne une instance de système de fichiers, c'est-à-dire une structure `sos_fs_manager_instance`. Le nœud correspondant au chemin *source* peut être un périphérique de type "bloc" (dans le cas des systèmes de fichiers stockant leurs données sur un support de masse ; p.ex `"/dev/hda1"`), un fichier, un répertoire ou rien du tout (dans le cas des systèmes de fichiers purement virtuels). Dans cette opération *mount*, le pilote de système de fichiers doit réaliser le nécessaire pour créer une structure `sos_fs_manager_instance` dûment remplie.

Pour terminer, la fonction `sos_fs_mount` informe le sous-système *namespace cache* du nouveau système de fichiers monté en utilisant la fonction `sos_fs_nscache_mount` étudiée plus haut.

Symétriquement, la fonction `sos_fs_umount` du *VFS* permet de démonter un système de fichier, en donnant son point de montage.

2.2.3 Structure `sos_fs_manager_instance` et opérations associées

Comme nous l'avons vu dans la section précédente, la structure `sos_fs_manager_instance` est créée par l'opération `mount` du pilote de système de fichiers, et détruite par l'opération `umount` de ce même pilote. Elle est donc unique pour chaque montage de système de fichiers.

```
struct sos_fs_manager_instance
{
  struct sos_fs_manager_type * fs_type;
  struct sos_fs_node * device;
  sos_ui32_t flags;
  struct sos_fs_nscache_node * root;
  struct sos_fs_node * dirty_nodes;

  sos_ret_t (*fetch_node_from_disk)(struct sos_fs_manager_instance * this,
                                    sos_ui64_t storage_location,
                                    struct sos_fs_node ** result);

  sos_ret_t (*allocate_new_node)(struct sos_fs_manager_instance * this,
                                 sos_fs_node_type_t type,
                                 const struct sos_process * creator,
                                 sos_ui32_t access_rights,
                                 sos_ui32_t flags,
                                 struct sos_fs_node ** result);

  sos_ret_t (*statfs)(struct sos_fs_manager_instance * this,
                      struct sos_fs_statfs * result);

  sos_bool_t (*nsnode_same_name)(const char * name1, sos_ui16_t namele
                                 const char * name2, sos_ui16_t namele

  struct sos_hash_table * nodecache;
  void * custom_data;
  struct sos_fs_manager_instance * prev, * next;
};
```

Elle contient notamment une référence vers le pilote du système de fichiers, vers le périphérique monté (dans le cas d'un système de fichiers reposant sur un support de stockage), ainsi qu'une référence vers le `nscache` racine du système de fichiers. Elle contient également une liste des nœuds *sales* (*dirty*) du système de fichiers, c'est-à-dire ceux qui ont été modifiés mais qui n'ont pas encore été enregistrés sur disque (si le système de fichiers autorise l'écriture différée pour des raisons d'efficacité). Il sera nécessaire avant de démonter le système de fichiers de s'assurer que les modifications apportées à ces nœuds soient bien enregistrées sur le support de stockage (opération `sync`).

La partie la plus intéressante de cette structure est formée des opérations `fetch_node_from_disk` et `allocate_new_node`.

`fetch_node_from_disk` permet de récupérer un nœud existant sur disque (ou en mémoire pour les systèmes de fichiers virtuels) à partir d'un identifiant unique que nous avons appelé `storage_location`. Cet identifiant est un entier sur 64 bits qui n'a de signification que pour le pilote de système de fichiers. Du point de vue du *VFS*, cet identifiant doit simplement être différent pour chaque nœud d'un même système de fichiers. En pratique, `storage_location` correspondra le plus souvent à un numéro de secteur ou de bloc (systèmes de fichiers normaux), ou à une adresse en mémoire (systèmes de fichiers virtuels). L'opération `fetch_node_from_disk` retourne donc une structure `sos_fs_node` décrivant le nœud.

L'opération `allocate_new_node` permet de créer un nouveau nœud dans le système de fichiers. Elle prend en paramètre diverses informations sur le nœud à créer, et retourne une nouvelle structure `sos_fs_node`. Cette dernière contiendra notamment l'identifiant unique `storage_location` de ce nouveau nœud. Cette opération est uniquement utilisée lors de la création d'un nouveau fichier, elle permet notamment de réserver de la place sur disque (ou en mémoire pour les systèmes de fichiers virtuels) pour contenir au moins les informations liées au fichiers (on parle parfois de "méta-données").

Les deux autres opérations de cette structure sont moins importantes. `statfs` permet de récupérer des statistiques sur le système de fichiers, tandis que `nsnode_same_name` est une opération de comparaison de chaîne de caractères (voir la section 2.1.3). Chaque pilote de système de fichiers peut ainsi gérer différemment la comparaison des noms de fichiers (par exemple pour être ou non sensible à la casse).

2.2.4 Structure `sos_fs_node`

Comme nous l'avons vu précédemment, la structure `sos_fs_node` représente un unique objet du système de fichiers. Avant de détailler son utilisation dans les différentes

opérations du *VFS*, nous allons étudier quelques champs importants de cette structure.

Une structure `sos_fs_node` peut représenter un fichier, un répertoire, un lien symbolique ou un périphérique bloc ou caractères. Ces différents types d'objets sont listés dans le type `sos_fs_node_type_t`.

Pour de nombreux systèmes de fichiers, un répertoire n'est qu'un fichier un peu spécifique qui contient une liste d'autres fichiers et d'autres répertoires, et un lien symbolique n'est qu'un fichier contenant le chemin d'un autre fichier.

```
typedef enum {
  SOS_FS_NODE_REGULAR_FILE = 0x42,
  SOS_FS_NODE_DIRECTORY    = 0x24,
  SOS_FS_NODE_SYMLINK      = 0x84,
  SOS_FS_NODE_DEVICE_CHAR  = 0x48,
  SOS_FS_NODE_DEVICE_BLOCK = 0x12
} sos_fs_node_type_t;
```

Un `sos_fs_node` fait partie d'un système de fichiers monté, auquel il fait référence par l'intermédiaire du champ `fs`. Par ailleurs, le champ `storage_location` contient l'identifiant unique manipulé par les opérations `allocate_new_node` et `fetch_node_from_disk` et permet de savoir où sont situées les informations concernant le fichier sur le disque. Le couple $(fs; storage\_location)$ permet donc de localiser sur disque un unique objet "fichier" d'un système de fichiers.

```
struct sos_fs_node
{
  struct sos_fs_manager_instance * fs;
  sos_ui64_t storage_location;
  sos_count_t ondisk_lnk_cnt;
  sos_count_t inmem_ref_cnt;
  sos_fs_node_type_t type;
  sos_ui32_t access_rights;
  sos_ui32_t flags;
  sos_bool_t dirty;
  sos_count_t generation;

  struct sos_fs_dev_id_t
  {
    sos_ui32_t major;
    sos_ui32_t minor;
  } dev_id;

  struct sos_fs_node_ops_file  *ops_file;

  union
  {
    struct sos_fs_node_ops_dir      *ops_dir;
    struct sos_fs_node_ops_blockdev *ops_blockdev;
    struct sos_fs_node_ops_symlink  *ops_symlink;
  };

  sos_ret_t (*destructor)(struct sos_fs_node * this);

  sos_ret_t (*new_opened_file)(struct sos_fs_node * this,
                               const struct sos_process * owner,
                               sos_ui32_t open_flags,
                               struct sos_fs_opened_file ** result_of);

  sos_ret_t (*close_opened_file)(struct sos_fs_node * this,
                                 struct sos_fs_opened_file * of);

  void * custom_data;
  struct sos_hash_linkage hlink_nodecache;
  struct sos_fs_node *prev_dirty, *next_dirty;
};
```

La structure `sos_fs_node` contient d'autres informations telles que le nombre de chemins dans l'arborescence stockée sur disque pointant sur ce nœud (`ondisk_lnk_cnt` : permet de gérer les liens "hard"), le nombre de `nscache_node` actuellement en mémoire pointant sur ce nœud (`inmem_ref_cnt`), le type d'objet, ses droits d'accès, un booléen indiquant si le fichier a été modifié ou non (`dirty`).

Toutefois, l'aspect le plus important de la structure `sos_fs_node`, ce sont les opérations, implémentées par le pilote du système de fichiers. Tout d'abord, quelques opérations sont communes à tous les types d'objets, à savoir `stat` et `sync`, dans la structure `sos_fs_node_ops_file`.

D'autres opérations sont spécifiques au type de l'objet `sos_fs_node`. Pour les répertoires, la liste des opérations est `sos_fs_node_ops_dir`, pour les liens symboliques, il s'agit de `sos_fs_node_ops_symlink` et pour les périphériques en mode bloc, il s'agit de `sos_fs_node_ops_blockdev`. Ces opérations sont toutes les opérations qui s'appliquent sur un `sos_fs_node`, c'est-à-dire qui ne dépendent pas de l'ouverture de l'objet (représentée par une structure `sos_fs_opened_file`).

Les dernières opérations de `sos_fs_node` permettent la destruction de l'objet (`destructor`), la création d'un nouveau fichier ouvert (`new_opened_file`) ou la fermeture d'un fichier ouvert (`close_opened_file`).

2.2.5 Ouverture et création d'un fichier ou répertoire

L'ouverture d'un fichier ou répertoire est implémentée dans la fonction `sos_fs_open` et se déroule en quatre étapes :

- Calcul de la racine de la recherche. Si le chemin est absolu (commence par `/`) alors la racine de la recherche est la racine du processus (racine globale, ou racine du "chroot" pour les processus "chroot"é), récupérée par `sos_process_get_root`. Sinon, le chemin est relatif et la racine de la recherche est le répertoire courant du processus, récupéré grâce à `sos_process_get_cwd` ;
- La recherche ou la création du `sos_fs_node` correspondant au fichier ou répertoire à ouvrir. Pour la recherche, la fonction `sos_lookup_node` est utilisée. Si l'objet n'est pas trouvé mais que la création a été demandée par le flag `SOS_FS_OPEN_CREATE` alors, la fonction `fs_create_child_node` est utilisée pour créer ce nouvel objet ;
- Création de la structure `sos_fs_opened_file` correspondant à l'ouverture de ce fichier ou répertoire en utilisant l'opération `new_opened_file` du `sos_fs_node` ;
- Enregistrement de cette nouvelle ouverture de fichier auprès du *namespace cache* pour la mise à jour des compteurs de références (`sos_fs_nscache_register_opened_file`).

La structure `sos_fs_opened_file` contient les informations propres à une ouverture particulière de l'objet fichier. Elle est relative au processus qui a ouvert le fichier et contient par exemple le mode d'ouverture du fichier et la position courante dans la lecture/écriture du fichier ou du répertoire. Elle contient également de nombreuses opérations de manipulation de l'objet.

Certaines de ces opérations sont communes à tous les types d'objet (`sos_fs_ops_opened_file`), d'autres sont propres à certains types d'objet : `sos_fs_ops_opened_dir`, `sos_fs_ops_opened_chardev`, `sos_fs_ops_opened_blockdev`.

Recherche d'un nœud. La fonction de recherche `sos_lookup_node` permet de récupérer un `sos_fs_node` à partir d'un chemin. La recherche s'effectue en descendant au fur et à mesure dans les répertoires, à l'aide d'une boucle traitant à chaque itération une partie

du chemin du fichier recherché. Ainsi, pour le chemin `/home/sos/code/fichier.c`, il y aura des itérations pour `home`, `sos`, `code` et `fichier.c`.

À chaque itération, on cherche d'abord à suivre un éventuel lien symbolique en utilisant la fonction `fs_resolve_symlink`. Cette dernière fait appel à l'opération `expand` du `sos_fs_node` (opération disponible uniquement pour les nœuds qui sont des liens symboliques), puis fait de nouveau appel à `fs_lookup_node` pour récupérer la destination du lien symbolique. Il y a donc une récursivité entre `fs_lookup_node` et `fs_resolve_symlink` lors de la résolution des liens symboliques. Pour éviter d'entrer dans une récursivité infinie, on introduit un paramètre `lookup_recursion_level`, incrémenté à chaque début de `fs_lookup_node`. Si il dépasse un certain seuil, alors on retourne une erreur.

Une fois que l'éventuel lien symbolique a été résolu, on regarde si le chemin entier a été traité (condition de sortie de la boucle). Si il n'a pas été traité en entier, il faut nécessairement que l'objet en cours de traitement soit un répertoire, pour qu'il puisse contenir d'autres objets.

Une fois cette vérification effectuée, on cherche dans le répertoire courant un objet ayant le nom de la prochaine sous-partie du chemin. Par exemple, si l'on se trouve dans le répertoire `/home/sos`, on cherchera une entrée avec le nom "`code`". La recherche est d'abord effectuée dans le *namespace cache*, qui contient peut-être déjà l'objet recherché. Si c'est le cas, alors on peut poursuivre la recherche à l'itération suivante.

En revanche, si l'objet n'est pas présent dans le *namespace cache*, alors il faut le récupérer à partir du système de fichiers (i.e. sur le disque). Pour cela, l'opération `lookup` du répertoire courant est utilisée. À partir du répertoire courant et d'un nom, elle renvoie l'identifiant unique `storage_location` de l'objet ayant ce nom, si il existe. Une fois cet identifiant récupéré, la fonction `fs_fetch_node` permet de créer effectivement la structure `sos_fs_node` en utilisant l'opération `fetch_node_from_disk` du système de fichiers monté. Une fois l'objet trouvé, il est ajouté en tant qu'enfant du répertoire courant dans le *namespace cache*, ce qui évitera lors des prochaines ouvertures du même objet d'aller chercher des informations dans le système de fichiers, et donc sur le support de stockage.

L'algorithme de cette recherche peut se résumer ainsi :

ALGO 1 Principe de la recherche d'un nœud.

```
SI le niveau de récursivité maximum de recherche a été atteint ALORS
  renvoyer une erreur
FIN SI
BOUCLE
  extraire la sous-partie du chemin à traiter
  SI le nœud courant est un lien symbolique ALORS
    résoudre ce lien symbolique
  FIN SI
  SI la sous-partie à traiter est vide ALORS
    renvoyer le nœud courant, car la recherche est terminée
  FIN SI
  SI le nœud courant n'est pas un répertoire ALORS
    retourner une erreur
  FIN SI
  rechercher la sous-partie du chemin dans le nœud courant au sein du
  namespace cache
  SI pas trouvé dans le namespace cache ALORS
    utiliser l'opération lookup du répertoire courant pour rechercher
    l'objet
    récupérer le nœud correspondant à cet objet (opération
    fetch_node_from_disk)
    l'insérer dans le namespace cache
  FIN SI
  reboucler en utilisant le nouveau nœud comme nœud courant
FIN BOUCLE
```

Création d'un nœud. La création d'un nœud est réalisée par la fonction `fs_create_child_node`, utilisée à la fois par `sos_fs_open` si le drapeau `SOS_FS_OPEN_CREATE` est positionné, et par `sos_fs_create`.

Cette fonction de création commence par allouer un nouveau nœud dans le système de fichiers via la fonction `fs_allocate_node`, qui elle-même utilise l'opération `allocate_new_node` du système de fichiers. Un nouveau `sos_fs_node` avec un identifiant `storage_location` unique est donc créé dans le système de fichiers et disponible.

Ensuite, la fonction `fs_register_child_node` permet d'enregistrer ce nouveau nœud en tant que fils d'un répertoire. L'opération `link` du répertoire parent est utilisée pour cela. Son rôle est de créer l'association, dans le répertoire parent, entre le nom du nouveau nœud et le nœud lui-même. Enfin, le *namespace cache* est informé de l'existence de ce nouveau nœud.

Les deux étapes sont bien séparées : tout d'abord il y a création d'un nœud dans le système de fichiers, nœud qui n'est attaché à aucun autre nœud, puis il y a enregistrement de ce nœud dans un répertoire.

2.2.6 Opérations sur les nœuds

Toutes les opérations de manipulation des nœuds sont implémentées en utilisant les opérations de la structure `sos_fs_opened_file`. Ces opérations sont relatives à une ouverture particulière d'un fichier ou répertoire car elles utilisent ou modifient des paramètres liés à cette ouverture là du fichier (principalement : le curseur de lecture/écriture dans le fichier).

Ainsi, la fonction `sos_fs_read` permettant de lire un fichier utilise l'opération `read`, la fonction `sos_fs_write` utilise l'opération `write`, etc. Il en va de même pour toutes les autres fonctions de manipulation des objets : `sos_fs_seek`, `sos_fs_truncate`, `sos_fs_mmap`, `sos_fs_poll`, `sos_fs_fcntl`, `sos_fs_ioctl`. Le code de cet article n'implémentera pas `sos_fs_poll`. Quant à `sos_fs_fcntl`, son implémentation dépend des standards qu'on veut respecter (Posix, SVR4, Linux, ...), nous la laissons à titre d'exercice.

En revanche, les fonctions de création et de destruction d'objets ne sont pas relatives à une ouverture particulière d'un fichier ou répertoire. Elles utilisent donc des opérations de la structure `sos_fs_node`. Par exemple, `sos_fs_create`, discutée plus haut, permet de créer un nouveau fichier en utilisant l'opération `link`. `sos_fs_link` permet de créer un nouveau lien dur, `sos_fs_symlink` de créer un nouveau lien symbolique, `sos_fs_mkdir` de créer un nouveau répertoire, `sos_fs_rmdir` de supprimer un répertoire et `sos_fs_unlink` de supprimer un fichier ou lien.

La création de fichiers (`creat` ou `open` avec le flag `SOS_FS_OPEN_CREAT`) et de répertoire (`mkdir`) sont deux opérations similaires. Dans les deux cas il s'agit de rajouter une "entrée" dans un répertoire parent, entrée qui fait référence au fichier ou au répertoire. Ceci est réalisé par `link`. Rien n'empêche plusieurs répertoires de conte-

nir des références vers le même fichier ou répertoire. Toutes ces références sont des “liens durs” vers le fichier, le nombre de ces liens durs étant donné par le champ `ondisk_lnk_cnt` de la structure `sos_fs_node`. La seule contrainte étant que le système de fichiers soit le même pour le(s) répertoire(s) parent(s) et pour le fichier référencé. Sous Unix, lorsqu'on “enlève” un fichier ou répertoire d'un répertoire parent, en réalité on n'enlève pas le fichier du disque. On ne fait que supprimer l'entrée correspondante du répertoire parent, ce qui a pour autre effet de décrémenter le champ `ondisk_lnk_cnt` dans SOS. Cette opération est réalisée par `unlink` (l'équivalent de la commande *shell* `rm`) pour les fichiers, et `rmdir` pour les répertoires. Le fichier ou répertoire disparaîtra réellement du disque lorsqu'aucune entrée de répertoire n'y fera plus référence (i.e. `ondisk_lnk_cnt` passe à 0). On remarquera en passant qu'il n'y a pas d'opération “`rename`” dans SOS : c'est qu'il suffit, pour renommer (ou déplacer) un fichier de `/a/b/c` vers `/e/f/g`, de faire `link(fichier, "/e/f/g"); unlink("/a/b/c")`.

Enfin, de même que l'opération `read` permet de lire séquentiellement un fichier, l'opération `readdir` permet de lire séquentiellement un répertoire en remplissant pour chaque élément du répertoire une structure de type `sos_fs_dirent`. Cela signifie que les appels successifs à `readdir` renverront successivement la liste de tous les fichiers présents dans le répertoire. C'est une fonction assez délicate à réaliser dans la mesure où, entre deux appels à `readdir`, des fichiers peuvent être supprimés ou créés dans le répertoire. On trouvera un exemple d'implémentation de cette opération dans `drivers/fs_virtfs.c`.

3 Implémentation d'un système de fichiers minimal

Pour illustrer le fonctionnement du *VFS*, nous vous proposons l'implémentation d'un système de fichiers simple. Les données du système de fichiers seront simplement stockées en mémoire.

Avant de détailler l'implémentation proprement dite, reprenons la liste des opérations principales à implémenter dans un pilote de système de fichiers.

Structure	Opération	Type de nœud	Description
sos_fs_manager_type	**mount**		Monte le système de fichiers et alloue la structure sos_fs_manager_instance
	umount		Démonte le système de fichiers et libère la structure sos_fs_manager_instance
sos_fs_manager_instance	**fetch_node_from_disk**		Récupère le nœud correspondant à une storage_location depuis le disque et retourne une structure sos_fs_node
	allocate_new_node		Alloue un nouveau nœud sur le disque et retourne une structure sos_fs_node avec une nouvelle storage_location unique
	statsfs		Donne des statistiques sur le système de fichiers en remplissant une structure sos_fs_statsfs
	nsnode_same_name		Fonction de comparaison de chemins spécifique au système de fichiers (*optionel*)
sos_fs_node	**stat**	Tous	Donne des informations sur le nœud en remplissant une structure sos_fs_stat
	sync	Tous	Synchronise le contenu du nœud sur le support de masse sous-jacent, si le nœud a été modifié
	expand	Lien symbolique	Retourne la cible d'un lien symbolique
	lookup	Répertoire	Recherche un nœud dans un répertoire à partir d'un nom donné
	link	Répertoire	Créé une nouvelle entrée référençant un nœud dans un répertoire
	unlink	Répertoire	Supprime une entrée référençant un nœud dans un répertoire
	kread	Périphérique bloc	Lecture d'un périphérique bloc. Réservée au noyau (articles 9 et 10)
	kwrite	Périphérique bloc	Écriture d'un périphérique bloc. Réservée au noyau (articles 9 et 10)
	destructor	Tous	Détruit la structure sos_fs_node
	new_opened_file	Tous	Créé une nouvelle structure sos_fs_opened_file pour le nœud courant
	close_opened_file	Tous	Détruit une structure sos_fs_opened_file du nœud courant
sos_fs_opened_file	seek	Tous	Déplace le curseur de lecture/écriture dans le fichier
	truncate	Tous	Tronque le fichier à une taille donnée
	read	Tous	Lit depuis le fichier
	write	Tous	Écrit dans le fichier
	mmap	Tous	Informe le système de fichiers que le contenu d'un de ces fichiers va être projeté en mémoire dans une région virtuelle
	poll	Tous	Attend que le fichier soit disponible en lecture ou en écriture (article 9)
	fcntl	Tous	Effectue une opération spécifique sur le fichier
	ioctl	Périphérique bloc et caractère	Effectue une opération spécifique sur le périphérique
	readdir	Répertoire	Lit une entrée de répertoire en remplissant une structure sos_fs_dirent
	duplicate	Tous	Duplique une structure sos_fs_opened_file. Opération utilisée lors du fork pour recopier les descripteurs de fichiers

TAB. 1 – Récapitulatif des opérations à implémenter dans un pilote de système de fichiers. L'implémentation des opérations en gras est obligatoire.

Le système de fichiers proposé est implémenté dans le fichier `drivers/fs_virtfs.c`. Comme nous l'avons dit, il stocke ses données en mémoire, dans la zone noyau. Pour conserver une représentation de l'arborescence de ses fichiers, il utilise une structure `virtfs_node` pouvant être soit un fichier, soit un répertoire. Cette structure contient directement la structure `sos_fs_node` (on pourrait parler d'"héritage structurel") qui sera retournée au VFS lors de l'ouverture d'un fichier ou répertoire du *virtfs*.

Dans le cas où le `virtfs_node` représente un fichier, on trouve un pointeur vers le contenu du fichier et sa taille. Dans le cas d'un répertoire, on trouve une liste de `virtfs_direntry`, c'est à dire une liste d'entrées de répertoire. Cette structure associe simplement un nom (le nom du fichier ou répertoire) à la structure `virtfs_node` représentant ce nœud.

Le pilote de système de fichiers implémente toutes les opérations obligatoires du tableau 1, de manière très simple.

On peut préciser qu'une fonction interne `virtfs_resize` est utilisée à chaque fois que la taille du fichier change. Dans un vrai système de fichiers stocké sur disque, lorsque la taille d'un fichier augmente, on alloue plus de blocs pour ce fichier. Dans le cas le `virtfs`, les données sont stockées en mémoire. À un instant t, l'espace minimum requis pour un fichier est alloué en mémoire. Lorsque la taille change, il faut ré-allouer de la mémoire et recopier les données dans cette zone nouvellement allouée. Cette fonction, très largement sous-optimale (elle ne prévoit pas de marge pour éviter d'appeler trop souvent `sos_kmalloc/sos_kfree`), est utilisée dans `virtfs_write` et `virtfs_truncate`.

Le champ `storage_location` du `sos_fs_node` est utilisé pour stocker directement une référence vers le `virtfs_node`. Ainsi, l'implémentation de `fetch_node_from_disk` n'est qu'un jeu de *transtypage* : la `storage_location` donne le `virtfs_node` qui contient le `sos_fs_node` à retourner. D'autre part, l'opération `destructor` (implémentée dans `virtfs_node_destructor`) ne fait rien. En effet, le `sos_fs_node` étant intégré dans le `virtfs_node`, il n'est pas possible de détruire le premier sans détruire le second, et si l'on détruit le `virtfs_node`, le fichier n'existe plus dans le système de fichiers. Il s'agit d'un simple choix d'implémentation, le `sos_fs_node` aurait très bien pu ne pas être intégré au `virtfs_node`.

4 Implémentation des appels système

À partir des fonctions du *VFS* sont implémentés les appels système disponibles aux applications utilisateur. Les appels système correspondent directement à des fonctions du *VFS*, leur implémentation est donc aisée.

La seule opération qui leur incombe est la gestion des descripteurs de fichiers ouverts. En effet, le *VFS* travaille uniquement avec des `sos_fs_opened_node` et les applications utilisateur avec des entiers, les descripteurs de fichiers ouverts. Pour cela, en plus des champs `root` (racine locale du processus courant) et `cwd` (répertoire courant du processus) a été ajouté à la structure `sos_process` un tableau de pointeurs vers des `sos_fs_opened_file` : le tableau `fds`. Lorsqu'on ouvre un fichier, l'appel système `open` utilise la fonction `sos_fs_open` détaillée précédemment, puis cherche un emplacement libre dans le tableau `fds` pour y enregistrer le nouveau `sos_fs_opened_file`. L'index dans ce tableau forme le *descripteur de fichier* renvoyé à l'utilisateur.

Pour tous les autres appels systèmes, il est nécessaire d'effectuer la traduction *descripteur de fichier* → `sos_fs_opened_file`. En effet, l'utilisateur donne un entier, et les fonctions du *VFS* attendent un `sos_fs_opened_file`. C'est le rôle du tableau `fds[]`.

Enfin, l'appel système `fork` a également été modifié pour recopier le tableau des descripteurs de fichiers du processus père dans le nouveau processus fils. Pour cela, il créé pour chaque fichier ouvert du tableau `fds` de la structure `sos_process` un nouveau `sos_fs_opened_file` en utilisant l'opération `duplicate`.

5 Résumé et bilan

Le VFS de SOS permet de supporter la plupart des fonctionnalités des Unix classiques, notamment : arborescence de fichiers/répertoires, points de montage, liens symboliques, fichiers spéciaux caractère ou bloc (nous y revenons dans le prochain article), liens "durs", descripteurs de fichiers, `chroot`, possibilité de supprimer ou déplacer un fichier ou un répertoire en cours d'utilisation.

Par rapport aux implémentations complètes des systèmes de fichiers sous Unix, les limitations de SOS sont les suivantes :

- Pas de gestion des droits d'accès par utilisateur (pas de notion d'uid/gid)
- Pas de mise à jour des dates d'accès/modification des fichiers
- Pas de fichiers spéciaux type `Fifo` ou *socket Unix*
- Pas de bibliothèque de fonctions générique pour faciliter l'implémentation de la signalisation asynchrone, le verrouillage total ou partiel, la surveillance de modification ou d'utilisation de fichiers (resp. drapeaux `F_SETOWN`, `F_SETLK`, `F_GETLEASE` et `F_NOTIFY` de `fcntl`)
- Pas de fonction type `flock`

Toutes ces fonctionnalités restent malgré tout compatibles avec le modèle proposé. Elles peuvent être implémentées à titre d'exercice.

D'autre part, bien que nous l'interdisions dans la fonction `link`, ce modèle permet même de supporter la notion de "lien dur" sur les répertoires, interdite sous Linux par exemple.

Cependant, une limitation profonde du modèle proposé ici risque d'apparaître lors de l'implémentation de systèmes de fichiers réseau dits "sans état" ou "*stateless*" (NFS par exemple). En effet, dans SOS, l'arborescence de fichiers est partiellement cachée en dehors du pilote du système de fichiers (`fs_nscache.c`), ce qui pose un problème classique de cohérence. Car le pilote du système de fichiers devra assurer le maintien de la cohérence entre cette arborescence cachée localement, et l'arborescence sur le serveur de fichiers distant. Ceci complique considérablement la réalisation du pilote de systèmes de fichiers réseau. Mais cette complication peut être levée en modifiant l'implémentation de `fs_nscache.c`. On pourra par exemple se rapprocher de celle de Linux (cache de *dentry*), qui prend la forme d'un cache des nœuds relativement isolés, sans cacher *tous* les liens de parenté entre tous ces

6 Démonstration

La petite démo du mois est dans la même lignée que les deux précédentes. Elle n'apporte aucun effet visuel impressionnant puisqu'elle se contente de tester les fonctions les unes après les autres. Par contre, elle devrait réjouir les habitués de la programmation Unix qui se retrouveront (enfin) en terrain connu (fonctions `open/read/write/seek/readdir`, ...).

Le code de la démo du mois se trouve dans le répertoire `userland/` et est constitué d'applications utilisateur qui utilisent la batterie de nouveaux appels système accompagnant cet article. Parmi les tests effectués, on trouvera bien sûr les tests des fonctions habituelles `open/read/write/seek/readdir`, mais aussi de `mmap`, `link`, `unlink`, `mknod`, `mkdir`, `stat`, `statfs`, `chdir`, `chroot`, ... La seule fonction qui n'est pas testée est `poll` (nous y reviendrons dans l'article 9). Si certaines de ces fonctions ne vous sont pas connues, il suffit de consulter le manuel Unix pour savoir à quoi elles correspondent.

Conclusion

Cet article a permis de présenter l'infrastructure du *Virtual Filesystem* ainsi que l'implémentation d'un système de fichiers virtuel simple. Nous vous proposerons dans un prochain article d'étudier un système de fichiers réel, reposant sur un support de stockage de masse. Toutefois, pour cela, il est nécessaire de disposer d'un pilote de périphérique pour disque dur et d'une couche de gestion des périphériques blocs.

L'article 9 sera donc centré sur l'implémentation de pilotes de périphériques : pilote de clavier, pilote de ligne série, et pilote de disque dur IDE. Ce sera l'occasion de présenter l'implémentation de pilotes de périphériques bloc et caractère, ainsi que le fonctionnement de quelques éléments matériels standards d'un ordinateur PC.

Pour terminer, signalons que nous présenterons SOS aux *Rencontres Mondiales du Logiciel Libre* qui auront lieu du 5 au 9 juillet à Dijon. Xavier Grave sera également présent et nous parlera de TOY LOVELACE, l'adaptation de SOS en Ada 95. En plus de ces deux présentations, le thème *Conception et développement des systèmes d'exploitation* propose 17 autres conférences auxquelles nous vous invitons à assister ! Enfin, ces *RMLL* seront également l'occasion de nous rencontrer et de discuter autour d'un verre !

Pour plus d'informations sur les *RMLL*, vous pouvez vous rendre sur le site `http://www.rencontresmondiales.org`. En ce qui concerne le thème sur les systèmes d'exploitation, le programme est disponible sur `http://www.rencontresmondiales.org/sections/conference/noyau_et_systeme`.

The end.

Thomas Petazzoni et David Decotigny
`thomas.petazzoni@enix.org` et `d2@enix.org`
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`

Références

[1] Neil Brown et al. The linux virtual file-system layer. `http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html`, 1999.

[2] Tigran Aivazian. Linux kernel internals, the vfs. `http://www.faqs.org/docs/kernel_2_4/lki-3.html`, 2002.