SYSTÈME DE FICHIER CRYPTÉ
SHELL

Antoine VENANT
Christophe RIOLO

Sous la direction de Dominique LAVENIER et Kévin HUGUENIN

Remerciements à Vincent Picard pour la présentation du document.

Système de fichiers crypté et interpréteur de commandes

Antoine VENANT Christophe RIOLO

22 février 2009

Résumé

Nous présenterons dans ce rapport les solutions que nous avons apportées aux problèmes de la création d'un système de fichiers crypté, et de l'élaboration d'un interpréteur de commandes, nous familiarisant avec ces deux briques de base d'un ordinateur, que sont la gestion de la mémoire et celle des processi.

Table des matières

1 SYSTÈME DE FICHIERS CRYPTÉ ROM

1.1 Introduction

Cette section concerne le système de fichier crypté réalisé dans le cadre du projet système. Nous exposerons les différentes étapes par lesquels nous sommes passés, nos choix de représentation, notamment en ce qui concerne la structure de données et sa représentation physique ainsi que les limites du système réalisé et les problèmes rencontrés.

1.2 Structure de données et Fonctionnement général

1.2.1 Structure Arborescente, liste d'attributs

La première question que nous nous sommes posé et celle de la représentation logique du système de fichier : quelle structure serait adaptée pour réaliser les manipulations demandées ?

Un système de fichier a par définition une structure arborescente, il s'agit d'un répertoire source et de ses fils lesquels peuvent être d'autres noeuds (repertoires) ou des feuilles (fichiers). Cette structure intuitive nous a paru convenir pour représenter le système au sein du programme, toutes les opérations sur le système se traduisent immédiatement en opération sur les arbres, par exemple :

création d'un répertoire	insertion d'un noeud
suppression d'un repertoire ou d'un fichier	suppression d'un noeud
recherche d'un fichier	recherche d'une étiquette

On représente donc le système par un arbre. Les noeuds possèdent les informations suivantes, le type (repertoire ou fichier) ainsi le nom du fichier ou du repertoire (en vue d'une recherche). Nous nous sommes également donné la possibilité de stocker toute sorte d'information sur le fichier *via* une liste d'attributs, sous la forme clé/valeur (les deux étant des chaînes de caractères), le but étant de permettre de rajouter facilement des fonctionalités suplémentaires suivant les besoin, sans modifier fondamentalement la structure (en particulier des informations diverses sur le fichier : taille, dernière modification, date de création...).

1.2.2 représentation physique

Il faut également pouvoir stocker et récupérer cette structure. Nous avons opté pour une représentation sous forme d'arbre xml, *i.e* une succession de balises imbriquées de type `file` ou `dir`, possédant des attributs, avec la syntaxe suivantes :

```
<type_de_balise clé1=''valeur1'' clé2=''valeur2''...>
        .
        .
        .
<\type_de_balise>
```

Par exemple, la commande `cmkfs` devra générer un nouveau fichier contenant une balise `dir` :

```
<dir name=''root''>
</dir>
```

contenu fichiers Avant de pouvoir réaliser le système proprement dir, il restait à définir comment stocker le contenu des fichiers. Une première solution envisagée était de stocker ce contenu au sein des balises `file` :

```
<file name=''fichier''>
contenu_du_fichier
</file>
```

Cependant, cette façon de faire exigerait un parcours complet de tout le système de fichier avant de pouvoir construire l'arborescence, de plus comme le système est crypté, il faudrait tout décrypter et charger l'intégralité du système en mémoire vive.

Or pour réaliser une opération, on a en général besoin d'accéder à un seul fichier. il suffirait donc d'employer un cryptage par blocs, de connaitre l'emplacement du fichier puis de décrypter uniquement les blocs associés au fichier.

Pour ce faire nous avons placé l'arborescence à part dans le fichier (tout à la fin), de manière à ce qu'on puisse décrypter et accéder à l'arborescence seule et nous avons ajouter des attributs `position` et `size` aux balises `file`, pour accéder à un fichier on charge initialement l'arborescence, puis on décrypte les `size` octets situés à l'offset `position` dans le fichier.

Le cryptage choisit est le cryptage A.E.S, on code des blocs de 16 octets.

1.3 implémentation

modules nécessaires Avant de pouvoir réaliser les fonctions de lecture, construction et sauvegarde de l'arborescence, nous avions besoin de définir les arbres, les listes d'attributs, et les listes d'arbres, et c'est par la que nous avons commencé, en programant les modules attribut_liste, arbre et arbre_liste.

Conformément à ce qui a été dit plus haut, un arbre est pointeur sur un enregistrement d'un type, d'un nom, d'une liste d'attributs et de la liste de ses fils (on garde aussi un pointeur sur le père, de façon a pouvoir facilement remonter dans l'arborescence) :

```
typedef struct node *arbre;
struct node {
  arbre pere;
  char type[5];
  char *nom;
  attribut_liste attributs;
  arbre_liste fils;
};
```

Nous avons également écrit io.c, qui contient des fonctions se chargeant de lire en décryptant ou d'écrire et cryptant des blocs de taille multiple de 16 dans un fichier.

Enfin, le module crypto.c contient les fonctions de cryptage et décryptage par l'algortihme A.E.S. Pour le réaliser, nous nous sommes appuyés sur le document `http://www.progressive-coding.com/tutorial.php`.

1.3.1 construction de l'arborescence

construction de l'arborescence Nous avons réservé le dernier bloc de 16 octets du fichier pour indiquer le nombre de blocs occupés par l'arborescence. Pour construire l'arborescence,

on commence par récupérer cette information, puis on s'en sert pour récupérer le texte de l'arborescence (constitué des derniers blocs précédants ce bloc de fin). Il faut ensuite analyser ce texte pour construire l'arborescence.

L'algorithme est le suivant :

```
initialisation: regarder la première balise,
                si c'est bien une balise dir alors
                creer un arbre root.
                commencer l'algorithme après cette balise
en attendant une balise dir fermante et avec
root comme parent.


algorithme(arbre parent, balise_attendue):
tant que la balise_attendue n'a pas été trouvée
            regarder la première balise depuis la position courante.
    déplacer la position courante derièr cette balise.
              Si cette balise est la balise fermante attendue, alors
 retourner parent.
      Sinon si c'est une balise ouvrante
       créer un arbre fils du type de la balise,
       lire les attributs de la balise, les ajouter à repertoire
       ajouter repertoire comme fils de parent.
                      appliquer l'alghoritme à fils
       en attendant la balise fermante correspondante.
    sinon il y a eu erreur de syntaxe.
```

Pour ce faire, nous avons utilisé la bibliothèque d'expression régulière pcre, qui permet entre autres de trouver la première occurence d'une expression régulière dans une chaîne de caractères. Nous avons utilisé plusieurs expressions régulières :

- recherche d'une balise :
  `<[^<>]*>`
- syntaxe correcte d'une balise ouvrante :
  `<[^/ <>=]+ *( +[^/ =<>]+="[^/ =<>]+")* *>`
- recherche d'un attribut[1] :
  `(([^/ =<>]+=)("[^/ =<>]+"))`
- tester le type d'une balise :
  `<file ,<dir`
- balises fermantes :
  `</file>,</dir>`

On définit également une structure `tag` représentant une balise, et contenant l'offset de début et la taille de la balise dans le texte xml de l'arborescence.

On implémente l'algorithme ci-dessus à l'aide de trois fonctions principales :

1. une fonction récupérant tous les attributs d'une balise, en itérant des recherches de l'expression correspondantes.

[1] pcre permet de récupérer la position de chaque sous exession entre parenthèses dans la chaîne testée. On récupère donc à la fois le nom etla valeur de l'attribut.

2. une fonction appliquant l'algorithme.
3. une fonction l'initialisant.

1.3.2 sauvegarde de l'arborescence

pour sauvegarder l'arborescence on effectue un parcours infixe de l'arbre :

```
Sauvegarde de l'arborescence:
 écrire la balise ouvrante correspondant au noeud visité,
 appliquer l'algorithme aux fils
 écire la balise fermante.
```

Une fois qu'on dispose des fonctions de construction et suvegarde de l'arborescence, on peut programmer les fonctions de manipulation du système de fichier proprement dit.

1.4 manipulation du système de fichier

Nous avons condensé toutes les informations relatives au système de fichier dans une structure `filesystem` contenant

- le fichier contenant le système.
- le mot de passe du système.
- l'arborescence

Excepté cmkfs, toutes les manipulations effectués sur le système de fichiers s'appuient sur le même schema :

- créer un objet filesystem (récupérer l'arborescence)
- appliquer le traitement.
- si des modifications ont étés apportées, sauvegarder l'arborescence.

Nous avons donc créer un fichier fs.c rassemblant les divers traitement, ainsi que la récupération de l'objet fs. On crée ensuite un éxécutable suivant le modèle ci-dessus pour chaque commande à implémenter.

pratiquement toutes les opérations à implémenter requiert de rechercher un fichier dans l'arborescence, c'est pourquoi nous avons ajouté une fonction recherchant un chemin dans l'arborescence et renvoyant le sous-arbre correspondant, ou NULL si le chemin est incorrect.

Reste Enfin à implémenter les fonctions demandés :

cmkfs on crée simplement (s'il n'existe pas, sinon on lève une erreur) un fichier dans lequel on écrit la chaîne

```
"<dir name=\"root\"></dir>#########0000000000000002".
```

cmkdir on récupère le chemin parent du répertoire à créer, on cherche le repertoire parent dans l'arborescence, si il existe on ajoute un fils, puis on sauvegarde.

cls et clr les deux utilisent à la même fonction de fs.c : on cherche le répertoire demandé dans l'arborescence Si il existe et que c'est bien un répertoire, on affiche tous ces fils, si l'option récursive est demandé, on appelle ensuite la fonction sur chacun des fils.

cin on récupère le repertoire parent du fichier à créer. On cherche le répertoire parent dans l'arborescence ; On vérifie que ce répertoire n'a pas déjà un fils de même nom que celui du fichier à créer. On lit ensuite bloc par bloc le fichier à importer, et on le recopie bloc par bloc dans le système de fichier (on complète éventuellement le dernier bloc pour atteindre 16 octets). On sauvegarde l'arborescence.

cout et ccat si le fichier à créer n'existe pas, on le crée, sinon on lève une erreur. On cherche le fichier à exporter dans l'arborescence. On récupère les attributs size et position. On le recopie ensuite bloc par blocs. Dans le fichier externe (ou stdout pour ccat).

crm et crmdir On cherche dans l'arborescence le répertoire parent du fichier à supprimer. On remplace la liste des fils, par la liste des fils dans laquelle on a retiré le fichier ou le repertoire à supprimer. On détruit l'ancienne liste. On détruit le fichier. On sauvegarde l'arborescence.

cmv On cherche dans l'arborescence le fichier à déplacer, On procède comme pour crm pour le supprimer de son emplacement actuel, mais on ne détruit pas le sous arbre associé ! On cherche dans l'arborescence le répertoire parent du fichier à créer, On ajoute le fichier à la liste des fils de ce répertoire.

1.5 problèmes rencontrés

Le cryptage A.E.S autant que le système de fichier fonctionnent bien indépendemment. Cependant le lien entre les deux, et largement imparfait, dans la mesure où une partie des fichiers importés n'est pas affichée correctement à l'appel de cout ou ccat. Nous avons pu identifier l'origine du problème, mais nous n'avons pas eu le temps de le résoudre tout à fait, même si certaines améliorations on pu être faites immédiatement : Nous avions initialement programmé le système à l'aide de fonctions qui se basait sur l'écriture de chaînes de caractères, or le codage de ces chaînes par A.E.S peut introduire un octet nul avant la fin de la chaîne et donc l'amputer de ce qui suit, du moins vis à vis des fonctions classiques de manipulation des chaînes de caractères. Une modification de la fonction writeAtEnd de io.c a nettement amélioré les choses, sans résoudre le problème tout à fait.

2 LE SHELL CASH

2.1 Questions préliminaires

Question 1

La différence fondamentale entre les commandes "cd" et "ls" est que la première est une commande de base du shell, alors que la seconde est en réalité l'appel au programme "ls", correspondrait en réalité à la commande exec.

Ainsi un programmeur réalisant un interpréteur de commandes n'a-t-il à se soucier que d'un nombre restreint de commandes de base (cd, exec, ...), le reste pouvant être codé comme des programmes à part entière, à l'instar de ls et printenv.

Question 2

Si l'on veut fixer une taille maximale à une ligne de commande, on pourrait mettre cette information à la disponibilité de tous les programmes avec par exemple unes variable d'environnement. Bien entendu, il serait malvenu qu'un programme modifie cette variable, et il faudrait alors la rendre constante. Avoir une telle information à disposition serait très important pour tout programme qui génère automatiquement des lignes de commandes (tel un front-end d'un outil de compression, qui pourrait faire une ligne de commande trop longue si on lui demande de compresser trop de fichiers en même temps).

Question 3

La solution la plus naturelle pour *rediriger* la sortie d'un programme vers l'entrée d'un autre est les pipes. En effet, tout écrire dans un fichier puis lire ce fichier avec le second programme présente deux défauts majeurs.

Premièrement, si le premier programme ne s'achève pas, le second programme ne se lancera jamais, et on peut imaginer que le second programme pourrait avoir un effet de bord qui provoquerait l'arrêt du premier, auquel cas, il serait très dommage que les informations ne soient pas redirigées au fur et à mesure ! (j'ai dans l'idée notamment un premier programme qui renvoie le nom du premier fichier ne commençant pas par "toto", et un second programme qui ajoute "toto" à la fin du fichier dont on lui a passé le nom en argument)

Ensuite, si l'on manque cruellement de mémoire (TRÈS cruellement pour refuser de créer un fichier texte), envoyer les données au compte goutte peut permettre de ne pas la remplir totalement.

Question 4

La structure la plus adaptée pour la ligne de commande et un tableau de tableaux de chaînes de caractères : le premier tableau contient les descriptions des commandes qui chaînées, chaque commande étant représentée par le tableau des arguments (le nom de la commande étant l'argument zéro).

2.2 Implémentation

Le projet a été séparé en 6 fichiers .c, qui seront vus plus en détail ensuite :

shell.c qui effectue les initialisations principales (création de variables globales communes à tous les fichiers notamment), libère la variable directory, contenant le nom du répertoire actuel, allouée dynamiquement, et effectue la boucle principale prompt/read/eval

utils.c qui définit quelques fonction utiles, telles la fonction qui change de répertoire en vérifiant qu'il n'y a pas eu d'erreur, pour pouvoir utiliser des noms explicites quand je veux effectuer l'action en question.

commandes.c qui contient les fonctions manipulant la ligne de commande pour en extraire la substantifique moelle qui sera appelée par les fonctions built-in

builtins.c qui contient lesdites fonctions built-in, cd, exit et exec. Le fichier a été organisé de façon à rendre plus aisé l'ajout de nouvelles fonctions. Il est d'ailleurs à noter que les définitions de types ont été mises en commun avec commandes.c via le fichier builtins.h.

pidlist.c qui contient les fonctions de manipulation d'une liste de pid, pour gérer la liste des programme lancés dans le shell à tuer à la sortie.

liste.c qui est un fichier de manipulations de listes d'entiers que j'avais fait il y a quelques années et que j'ai modifié pour générer des listes de pid, que j'interface dans pidliste.

Maintenant que les présentations sont faites, rentrons un peu plus dans les détails.

Le shell

Le shell a été nommé CASH (Christophe & Antoine's SHell), car après tout, ils n'ont pas non plus étés sérieux pour BASH ! La fonction qui affiche le prompt affiche également l'utilisateur actuel et le dossier actuel. Pour éviter les débordements, le nom d'utilisateur est tronqué si il est trop long (ex : si l'utilisateur a eu la bonne idée de s'appeler abcdefghijklmnopqrstuvwxyz, alors sera affiché à la place "abcdefghijklmnopq..."). Ces informations sont récupérées à chaque prompt, si bien qu'après un cd, c'est le bon dossier qui s'affiche. En revanche, j'ai pu découvrir que lors de l'appel de su, c'est un autre shell qui se lance, et ce n'est pas le nôtre !

Lors de l'affichage du dossier, j'ai géré le remplacement du dossier personnel par $\sim$, pour que ce soit plus lisible. Pour cela j'ai récupéré le chemin du dossier personnel dans l'environnement avec getenv(), puis j'ai testé si le début du dossier courant (récupéré avec getcwd()) commençait par ce dossier avec strncmp().

Pour pouvoir avoir un chemin arbitrairement grand, la taille de directory est allouée dynamiquement, c'est l'intérêt de la fonction getdir(), qui me permet d'augmenter la taille de cette chaine de caractère si elle n'est pas suffisante (mais 100 caractères par défaut, c'est déjà bien).

Problème notable de la récupération de la ligne de commande : si la fenêtre est trop petite et que notre commande passe à la ligne, il est impossible de remonter dans la commande, ou plus exactement, on ne le voit pas.

Une fois la commande récupérée, on la stocke dans la variable globale "cmd" puis on appelle evalcmd(). L'avantage de la variable globale par rapport à un argument est la mise en commun à toutes les fonctions et surtout aux structures de commande de cette ligne de commande (structure que nous allons voir à la prochaine section).

Les commandes

Une commande est de la forme :
commande : := *commande-elem* (| *commande-elem*)*
commande-elem : := cd | | exit | | nokill *programme* | | *programme* (&)
programme : := nom (argument)*

On notera les espaces avant et après le caractère | dans les commandes chaînées, ce choix s'explique par le fait que "commande1 | commande2" est remplacé par "commande1 &\000commande2", sans les espaces on aurait un "commande1&\000ommande2" que le shell ne comprendrait pas.

Le mécanisme interne est de décomposer la chaîne cmd en sous chaînes en remplaçant les | par le caractère nul. Ainsi la chaîne se trouve décomposée en sous chaînes dont on stocke les adresses (l'adresse de cmd + le nombre de caractères de ce qui précède). On procède ensuite de même avec les espaces (et les tabulations) pour décomposer en arguments.

En mémoire, bien que cette représentation suffise a priori pour une commande élémentaire, comme on l'a vu dans les question préliminaires, j'ai choisi d'utiliser une structure définie dans builtins.h :

- Le nombre d'arguments est retenu notamment pour tester si le dernier est un & (c'est d'ailleurs ce pourquoi il faut un espace avant, mais ce choix n'était pas nécessairement judicieux), puisque l'on ne peut pas décider a priori si un caractère nul vaut pour un espace ou un |
- Les arguments, comme un tableau de pointeurs vers les différentes parties de cmd qui représentent les arguments. C'est ce tableau que l'on passe directement à execvp, d'où l'avantage de cette représentation.
- la commande built-in correspondant, que j'ai défini dans une énumération dans un souci de lisibilité.

Il me suffit ensuite de faire un switch sur la commande built-in (un intérêt supplémentaire de l'énumération !) pour savoir ce que je dois faire avec ma commande.

Lors de l'exécution d'une commande chaînée, pour chaque |, on crée un tube, et chaque commande (sauf les extrêmes bien sûr !) sera lancée en connaissance des deux tubes l'entourant, pour fermer les extrémités idoines.

Départ et nokill

Lorsque l'on quitte le shell, on voudrait faire le ménage dans les programmes qui ont étés lancés. Pour cela, on introduit une structure de liste de PIDs, qui contient la liste des processus fils à tuer en quittant. Lorsque l'on quitte avec la commande exit (lors d'un départ forcé, difficile de prévoir ce que l'on fait), on tue tous les fils dans cette liste.

Pour implémenter la fonction nokill, il suffit donc de ne pas mettre le processus dans cette liste. Pour ensuite dire explicitement que le fils ne doit pas être un zombie, je le rattache explicitement à init en utilisant _exit() au lieu de exit()

Principaux problèmes

Un problème est l'absence de commandes enchaînées par " ;" , mais ceci s'implémenterait par la même méthode que les commandes chaînées est la décomposition en arguments.

Aussi, tel que j'ai implémenté le &, on peut a priori le mettre sur chacune des commandes chaînées. Mais de toutes façons, pour ne pas que les commandes chaînées bloquent

en attendant la mort des premières (comme dans la commande “cat | cat”, même si elle ne sert à rien), il faut que les commandes soient “indépendantes”. Mais comme je me suis rendu compte de ce problème très tard, je pouvais difficilement modifier la structure de mon programme en peu de temps, donc j'ai utilisé la fonction de nokill, qui intègre le & automatiquement (à quoi bon empêcher la mort si le programme bloque le shell ?). Mais du coup dans les commandes chaînées, si je fais par exemple “emacs | true”, ce qui encore une fois est d'un intérêt limité, c'est équivalent à “nokill emacs”, ce qui n'est pas voulu.

Aussi, je confesse avoir retiré mes tests sur waitpid(), tester si la valeur était -1. J'affichais un texte dans ce cas, qui stipulait que le fils avait été mal enterré. Or ce texte s'affichait tout le temps pour les programmes lancés sans le & ou le nokill, bien que le fils était bien enterré. J'ai fini par comprendre (sauf erreur de ma part) que bien que le fils ait été enterré, le père interceptait le signal SIGCHLD et tenter d'enterrer le fils une deuxième fois, mais je n'ai pas encore trouvé comment adresser ce problème.

Intégration

Suite à un oubli de ma part, l'intégration du système de fichiers n'est pas encore terminé à l'heure actuelle, mais l'idée derrière la réalisation est la suivante :

- pour savoir si on se trouve ou non dans une archive, j'ai ajouté un drapeau mis initialement à 0, et que l'on met à 1 lorsque l'on fait cd sur le nom d'une archive (qui se termine en .arch pour cela).
- Une fois le drapeau à 1, l'évaluation de la ligne de commande passe par une autre fonction que evalcmd, qui appelle d'autres fonctions que les fonctions par défaut.
- lors des commandes cd successives, on garde en mémoire le chemin parcouru, et lors d'un cd .. à la racine de l'archive, on remet le drapeau à 0.

Normalement l'intégration devrait être finie pour la soutenance, mais pour l'instant, seule la nouvelle fonction d'évaluation est terminée (ou presque, je n'exclue pas d'éventuelles modifications plus tard).