Rapport du projet de programmation système

Yasin SENEL & Grégory EMIRIAN & Salomé COAVOUX

8 décembre 2012

Table des matières

1 Nos choix dans le code

1.1 Les headers

Au tout début de notre projet, après avoir essayé la vraie commande tar de plusieurs manières, et réfléchi sur comment définir un fichier dans une archive, nous avons décidé que la meilleure manière pour nous d'implémenter des archives allait être d'utiliser des headers, tout comme la commande qui nous servait de modèle. Nous avons donc cherché les headers de tar, et essayé de les comprendre. Nous avons ensuite créé une structure header avec le minimum requis, c'est-à-dire le nom du fichier, sa taille, sa date de dernière modification et ses droits. Bien qu'il y ait beaucoup de perte du tar originel au notre, nous n'avons pas trouvé utile d'avoir plus de champs dans notre structure, les nôtres suffisant pour implémenter les fonctions demandées. Cependant, nous avions commencé avec des attributs du header de type char *. Il en a résulté beaucoup de problèmes, des segfaults, et des difficultés pour implémenter les fonctions lire et écrire. Nous avons donc refait notre structure, en s'inspirant de celle de Thomas PIZON, Aymeric MIVIELLE et Matthieu DESPLAT, qu'ils avaient mis à notre disposition. Le code était plus clair, et bien plus simple, bien que nous ayons gardé notre format de headers de base.

1.2 Lecture et écriture

Les premières fonctions qui nous ont posé problème ont été celles de lecture et d'écriture d'une archive. Nous cherchions à écrire champ par champ nos headers avant d'écrire le fichier correspondant. Cela a ensuite posé des problèmes pour la lecture, car chaque champ était séparé des autres par un saut de ligne. Cette première implémentation était due au fait que nous testions toutes nos fonctions, soit avec la commande cat, soit en ouvrant l'archive dans un éditeur de texte, et que nous n'obtenions jamais le résultat désiré. Sur les conseils de Jérémy TURON, nous avons abandonné notre écriture champ par champ pour écrire directement la structure entière. La lecture s'en est trouvée grandement simplifiée, et nous avons ensuite pu nous pencher sur les autres fonctions restantes.

1.3 Les fonctions manquantes

Nos principales fonctions manquantes sont la gestion des dossiers, et l'option sparse. En effet, si la création, la lecture et l'écriture de fichiers nous paraissait assez intuitives, car en relation étroite avec le cours, la question de la gestion des dossiers n'a été posée qu'à la toute fin, et le temps nous a fait défaut. Nous n'avions aucune idée de comment les définir à la base, et encore moins de comment transformer notre code pour ajouter la gestion des fichiers aux fonctions déjà écrites. Nous avons tenté, infructueusement, une ébauche de la fonction sparse, en utilisant le format yale sparse matrix, mais il aurait fallu se

pencher dessus plus tôt pour avoir pu comprendre entièrement le principe et la coder correctement.

2 Nos principaux problèmes

2.1 La duplication de code

Le plus grand problème que nous n'avons pas pu régler à temps est la duplication de code. En effet, plusieurs de nos fonctions parcourent les fichiers ou les archives de la même manière ou presque, mais nous n'avons pas réussi à les rassembler, c'est le cas par exemple des fonctions lireEntetes et nbEntetes. La conséquence de cette duplication est la lourdeur du code : nous n'avons pas pu limiter les appels, notamment des *subroutines*. Comme certaines fonctions sont liées, nous aurions dû coder une fonction d'ouverture de l'archive ou des fichiers dans le mode voulu, et éviter ainsi d'ouvrir et de fermer nos fichiers dans chaque fonction. Bien que nous ayons pris conscience de nos erreurs, nous avons décidé au maximum de rendre un projet fonctionnel. Notre code aurait pû être beaucoup plus beau et optimisé que cela, et si nous n'avions pas été pris par le temps, c'est ce que nous aurions fait.

2.2 Les fonctions fausses

Les fonctions compresser et décompresser ont été implémentées très rapidement, et elles fonctionnent en temps normal. Cependant, nous les avons implémenté à l'aide de la fonction system, ce qui est fortement déprécié. Le code qui aurait dû être choisi à la place aurait utilisé un fork/exec ainsi qu'un pipe.Il en est de même pour la fonction difference, un fork/exec en utilisant la commande Unix *diff* aurait beaucoup mieux optimisé notre code. A la place, nous avons préféré ouvrir fichier et archive à la main et comparer les deux ligne par ligne. Les différences sont donc affichées quelques soient les fichiers, y compris les fichiers binaires. Enfin, notre grande erreur a été de ne pas penser à implémenter l'option -f, ou plutôt de ne pas penser à ne pas l'utiliser dans notre code. Nous n'avons pas redirigé l'écriture sur la sortie standard en cas de non-utilisation de cette option. Il faut donc préciser -f avec un nom d'archive à chaque utilisation de notre programme.

2.3 La personnalisation du code

Il est très dur de faire un code personnel. D'abord, parce que nous cherchons à réimplémenter une commande qui existe déjà. Notre code réutilise cependant très peu le code de tar : nous nous somme inspirés uniquement des headers. Nos connaissances étant limités, nos recherches internet nous ont souvent amenés sur des pages de codes avec des explications, que nous avons utilisé après les avoir adaptés à nos besoins. Enfin, très couramment, nous échangions nos idées ou nos manières d'implémenter avec les autres groupes. Nous avons donc été constamment influencés par les autres élèves, même si ce n'était qu'oralement.

Il est fort possible que notre façon de coder ait été influencée par ce partage d'idées, en bien comme en mal, d'ailleurs.

3 Nos résultats

3.1 Le main

Nous n'avons passé que très peu de temps sur le main. Ainsi, notre switch case est bancal, et ne fonctionne que si les options sont données exactement dans le bon ordre. Par exemple, pour créer une archive d'un nom donné, il faut d'abord appeler l'option -f avec le nom de l'archive voulu, puis l'option -c avec tous les fichiers à mettre dans l'archive. De même, pour afficher les fichiers contenus dans l'archive, il faut d'abord appeler -f avec l'archive, puis -t. Notre fonction principale est comme cela parce que nous ne nous sommes malheureusement pas penché sur la question plus tôt, nous faisions nos tests en appelant directement les fonctions à la main, et en affichant dans le terminal des données spécifiques aux fonctions appelées pour savoir si elles avaient fonctionné.

3.2 Les fonctions implémentées

Chaque fonction a été testé plusieurs fois avec des arguments différents. Il n'est malgré tout pas garanti que le code soit sans bug. Comme décrit précédemment, la duplication du code aurait pu être évitée, en rajoutant par exemple des fonctions intermédiaires. Ainsi, nous avons découpé le code, mais pas autant que voulu. On distingue tout de même la lecture, l'écriture, les options agissant sur un fichier à la fois, notre structure de base (les headers), mais aussi notre fonction d'aide. Il aurait d'ailleurs été plus judicieux de faire un fichier .c par option sur les fichiers, ou de rassembler la fonction de manuel avec ces options.

3.3 Le partage de code

Ce qui a été fait	Auteur
Ajouter un fichier	Yasin
Compresser une archive	Yasin
Décompresser une archive	Yasin
Mettre à jour une archive	Yasin & Grégory
Main (fonction principale)	Yasin & Grégory
Ecrire dans l'archive	Grégory
Sparser les fichiers	Grégory
Différence entre fichier et archive	Grégory & Salomé
Lire l'archive	Salomé
Effacer un fichier	Salomé
Extraire les fichiers	Salomé & Yasin