Projet du module de Réseaux et Systèmes : *ptar*, un extracteur d'archives *tar* durable et parallèle

GARCIA Guillaume et ZAMBAUX Gauthier

December 16, 2016

Remerciements

Nous avons réaliser ce projet en nous aidant des sites web mentionnés dans la section Bibliographie, avant l'annexe. Avec ces sites nous avons pu trouver des solutions à nos problèmes et des renseignements sur les fonctions standards et appels systèmes en C et comment les utiliser. Nous avons également trouver des renseignements sur la zlib et toute sa documentation (qui était suffisante) sur *zlib.net*.

Nous tenions également à remercier Alexandre Chichmanian (également élève en deuxième année) avec qui nous avons souvent discuté de nos problèmes respectifs et parfois communs, et des stratégies à employer pour les résoudre. Discuter avec quelqu'un d'extérieur au binôme nous a permis de prendre du recul sur ce que nous avions fait et de nous rendre compte de certaines erreurs aberrantes que nous n'avions pas remarquées.

Enfin, nous tenions à remercier M. Lucas Nussbaum pour nous avoir offert la possibilité de tester notre programme régulièrement sur des archives spécialement conçues pour les test, ce qui nous a grandement aidé dans les phases de débogage. Nous avons pû nous inspirer de ces archives pour créer les nôtres et mettre ne place nos propres tests (cf $archives\_test/vice.tar$ dans le dépôt github).

1 Conception

1.1 Page de manuel

Pour l'écriture de la page de manuel *ptar(1)*, il a fallu assimiler le language utilisé pour écrire ce type de documents. Nous avons ainsi été capable de reproduire la page donnée dans le sujet du projet. Pour l'édition et l'affichage de la page, nous avons utilisé l'utilitaire *groff-utf8* qui permet l'utilisation de caractères spéciaux comme les accents français.

Pour visualiser la page du manuel, il est possible d'utiliser la commande :

$$man \quad ./manpage/ptar.1.gz$$

depuis la racine du dépot. À noter que le format *.1.gz* est un format standard pour les pages (1) de manuels. Si toute fois on souhaite la consulter avec la commande :

$$man \quad ptar$$

il est nécessaire de d'abord suivre la procédure d'installation décrite dans le *README* (nécessite les droits administrateur).

1.2 Étape 1 : Listeur basique

Tout au long de la conception du programme, la page de manuel *tar(5)* nous a servi de référence. Nous y avons d'abord trouvée la structure d'une entête d'archive *POSIX USTAR* (nous avons donc choisi de gérer ce type d'archives).

Pour parcourir l'archive, nous avons utilisé les fonctions de *fcntl.h* (*open()*, *read()*, *lseek()*, *close()*) et une boucle *do...while* dont la sortie est assurée par la détection d'une taille nulle du champ *name* du *header*. En effet, la fin de l'archive (*End Of File*) est indiquée par deux blocs successifs d'octets à 0 (voir *tar(5)*).

Nous avons également dû omettre les données suivant le *header* dans le cas d'un fichier non vide. Pour cela, nous récupérions le champ *size* du *header* afin d'appeler la fonction *lseek()* pour déplacer la tête de lecture courante du *offset* égal à *size*. Deux problèmes se sont alors présentés :

- Nous devions convertir *size*, une chaine de caractère représentant la taille en octal, en un entier (*int*) en base décimale. Pour cela, nous avons utilisé la fonction :

  $$long \ int \ strtol(char * string, \ char **endptr, \ int \, base)$$

  Le deuxième argument ne nous intéresse pas ici et est mis à *NULL*.

- En appliquant simplement cela, nous nous sommes apperçu que le programme n'affichait pas ce qui était attendu. Cela était dû à la segmentation des archives *tar* en blocs de 512 octets et était particulièrement visible sur les données des fichiers (qui suivent le *header* concerné). Nous avons trouvé ce phénomène en utilisant la commande :

$$hexdump \ -C \ nomArchive.tar$$

  Nous avons donc dû déterminer le multiple de 512 supérieur à *size* après conversion qui lui était le plus proche, tout en excluant le cas où *size* en était déjà un multiple.

1.3 Étapes 2 et 3 : Options de lignes de commande et extraction

Pour gérer les options en lignes de commande, nous avons utilisé la page de manuel *getopt(3)*. Cette partie n'a pas posé de problème particulier.

À cette étape, nous avons voulu vérifier l'existance et la validité (extension correcte) de l'archive passée en paramètre (voir *checkfile.h* dans les sources du dépot). Nous avons aussi, dans la boucle principale, vérifié le champ *magic* du *header* pour nous assurer qu'il s'agissait bien d'une archive *USTAR*. Le cas échéant, *ptar* retourne immédiatement 1.

Pour ce qui est de l'extraction, nous avons créer une fonction

$$int \ extraction(headerTar \ * header, \ char \ * data)$$

où *data* sont les données éventuelles suivant le *header* récupérées à l'aide d'un appel à *read()* à la place de *lseek()*. La différenciation d'éléments se faisait sur le champ *typeflag* du *header* à l'aide d'un *switch...case* sur le premier caractère de ce champ. Pour extraire, nous avons utilisé diverses fonctions telles que *open()*, *write()*, *mkdir()* et *simlink()*. Il fallait faire attention à passer *size* converti non ramené au multiple de 512 calculé dans la boucle principale en argument au *write()* éventuel.

1.4 Étape 4 : Listing détaillé et application des attributs à l'extraction

Nous avons crée une fonction *listing()* qui prend en argument un *header*. Cette fonction affiche les informations comme le ferait la commande *ls -l*. Nous n'avons pas eu de problème particulier grâce à la fonction *strtol()* citée précédement, excepté dans le cas d'un lien symbolique pointant sur un élément dont le nom est un nombre qui produisait un faux *typeflag*. L'erreur venait du fait qu'on ne prenait pas le premier caractère du champ *typeflag* mais toute la chaîne qui parfois se concaténait avec le champ *linkname* qui contenait dans ce cas un nombre.

Pour l'application des attributs aux éléments pendant l'extraction, nous avons utilisé *strtol()*, *setuid()*, *setgid()*, *chmod()*, *utimes()* et *lutimes()* pour les liens symboliques. Cependant, nous nous sommes apperçu qu'il fallait changer la date de modification des dossiers en fin de traîtement car

le fait de créer un fichier à l'intérieur de ce même dossier met à jour la date de modification de ce dernier. Pour cela nous stockons les noms de dossiers dans un tableau de string (char[2048][255]) et les champs mtime converti en int dans un tableau d'int (int[]). Nous avons choisi par convention de considérer qu'une archive tar ne peut excéder 2048 dossiers différents.

Nous avons choisi de créer une option *-e* pour dévelopeur visant à créer un *logfile* des divers appels systèmes et fonctions utilisées dans le programme. Cela nous a été fortement utile pour déboguer le programme par la suite. Une information a été ajoutée à la page de manuel à cette occasion (cette fonction est désactivée lors de l'extraction dans le cas d'un multithreading).

1.5 Étape 6 : Changement dynamique, décompression et archives désordonnées

En ce qui concerne le changement dynamique de la *zlib* avec *dlopen* de *dlfcn.h*, nous avons rapidement trouvé la marche à suivre dans le cours. Nous avons donc chargé les cinq fonctions homologues à *open()*, *read()*, *lseek()* et *close()* de la *zlib*. Au début, nous avions tenté de charger la *zlib* téléchargée localement avant de nous rendre compte qu'il suffisait de ne pas spécifier de chemin en argument de *dlopen()* pour que celle-ci cherche automatiquement dans le système grâce à la variable d'environnement *LD_LIBREARY_PATH*. Une fonction permet de charger la bibliothèque dynamique avec un *void *handle* globale et c'est la fonction *main* qui appelle *dlcose()* sur *handle*. Les fonctions sont aussi déclarées globalement dans *utils.h*

Notre première stratégie, abandonnée par la suite, consistait à décompresser l'archive *tar.gz* dans un fichier temporaire en lui donnant un nom quelconque (pour éviter d'écraser un fichier portant potentiellement le même nom) à l'aide des fonctions de la *zlib*. Il suffisait de lire ensuite le fichier dans la fonction principale comme s'il s'agissait d'un tar normal. Cette solution peut élégante et "bricolée" ne nous a pas satisfaits. Nous avons eu l'idée par la suite de passer par un tube nommé en *forkant* (avec l'appel système *fork()*) notre processus, l'un écrivait dans le tube puis se tuait tandis que le principal lisait la sortie du tube (en attendant la mort du fils). Cette solution nous semblait plus adaptée au attentes du module de RS. Cependant, il s'avère que la taille maximale d'un tube sur notre système d'exploitation (Ubuntu) est de 64 Ko, et comme on était obligé de décompresser totalement avant d'y lire, la taille maximale d'une archive compressée pouvant être lue par *ptar* était de 64 Ko. A ce point, deux solution s'offrait à nous : soit nous retournions à l'idée de départ, soit nous procédions directement à la différenciation de *open*/*gzopen*, *read*/*gzread*, *close*/*gzclose*, dans le corps principal du programme en vérifiant à chaque fois si l'option -z était spécifiée. Nous avons donc choisi la deuxième option, beaucoup plus élégante et efficace en mémoire et vitesse d'exécution. Il a fallu passer un certain nombre de variables locales en variables globales (voir *utils.h*), et donc coder avec prudence. Avec cette solution, il est donc possible de décompresser en même temps d'extraire/lister.

C'est à partir de cette étape que nous avons été confrontés à un problème de taille : les archives désordonnées. Il s'agit d'archives dont, par exemple, le *header* d'un fichier contenu dans un dossier pouvait arriver avant celui du dossier en question, et donc il était impossible de créer le fichier. Bien que M. Nussbaum nous ait assuré que ce cas ne se produirait pas dans les tests blancs, nous avons tout de même voulu résoudre ce problème, notamment parce que nombre de nos archives de test étaient désordonnées.

Notre première idée, très grossière, gourmande en mémoire et peu efficace à l'exécution, consistait à d'abord lire l'archive et de ranger les *header* dans des structures spéciales créées pour l'occasion. Ces structures sont des tableaux de *header*, de *char*, et des *int* (voir *sorting.h* en annexe, ce code ne fait plus parti du code source). Il s'agissait ensuite de trier ces *header* et les données éventuelles suivant ce *header* (à côté mais dans la même structure), à l'aide de diverses fonctions et de notamment un tri à bulle simple. Cette stratégie a été un enfer à déboguer tant nous avions de problèmes à ce niveau : beaucoup de *core dump* étaient dénombrés dans nos tests dûs à de nombreux *malloc/free* hasardeux. Le tri se faisait sur la profondeur de l'élément dans l'archive en comptant le nombre de token séparés par le délimiteur '/'. Nous avons utilisé

$$char\ *strtok(char\ *string,\ const\ char\ *delim)$$

de *string.h* pour cela (et pour la fonction *bool checkfile(char *file)* de *checkfile.h* également, mais avec le délimiteur '.').

Nous avons fini par abandonner cette idée pour une bien meilleure : vérifier l'existence des dossiers parents grâce à des fonctions conçues pour l'occasion dans *checkfile.h*. Les deux premières fonctions, *bool existeDir(char *dir)* et *bool existeFile(char *file)* testent tout simplement l'existence d'un dossier ou d'un fichier. On en a besoin par la suite dans *extraction()* de *utils.h* et dans *int checkpath(char *file)* de *checkfile.h*. Dans cette dernière on utilise encore *strtok()* avec le délimiteur '/' en conjonction de *strcat()* en bouclant et en s'assurant à chaque boucle de l'existence du dossier dont le nom est constitué au moment de la boucle, si il n'existe pas on le créer avec *mkdir()* de *sys/stat.h* avec des droits drwxrwxr-x, qui sont temporaires puisque le header de ce même dossier va être lu plus tard par *ptar* et mettras à jour ses droits avec ses bonnes permissions. Ensuite on passe au dossier fils, et ainsi de suite jusqu'à arriver à l'élément pointé par le *header*. Sa création n'est plus un problème puisque son chemin de dossiers parents existe forcément (sauf si il y a eu une erreur, causée sans aucun doute par une problème de droits de création de dossier). Pour finir, la fonction *char *recoverpath(char *linkname, char *pathlink, char pathname[])* de *checkfile.h* était utilisée pour recréer un chemin complet relatif depuis l'endroit d'exécution de *ptar* pour les pathname des éléments pointés par un lien symboliques afin de les créer avant de leur appliquer *symlink()*. Cela créait des désagrément, il n'est en fait pas nécessaire de créer l'élément avant d'appeller *symlink()*. Cette fonction n'est donc plus utilisée puisqu'on ne teste plus l'existence de l'élément pointé par le lien.

1.5 Étape 7 : Checksum et header Pax tar, UTF-8

Cette étape fut plutôt facile à réaliser à condition d'être rigoureux. Nous avons donc créé une fonction *bool checksum(headerTar *head)* qui se charge de calculer le *checksum* du *header* et de le comparer au champs *checksum* du *header*. Elle renvoie vrai si l'archive est corrompue, faux sinon. Cette fonction est appelée pour chaque header. Si l'un des header est corrompu, un booléen global (global pour la compatibilité avec les threads) est mis à vrai ; cela permet de continuer d'exécuter ptar sur le reste de l'archive, et de renvoyer 1 à la fin. Seuls les éléments sains sont extraits.

Nous avons utilisé l'algorithme de calcul spécifié dans tar(5), à savoir une sommation de chaque byte du header en remplaçant les 8 bytes du champ *checksum* du header par 8 espaces de

valeurs ASCII 32 en décimal. Nous avons ensuite pratiqué un masque 0x3FFFF (à l'aide d'un ET bit-à-bit) sur cette somme d'*unsigned int* pour ne récupérer que les 18 bits de poids faibles comme indiqués dans tar(5).

Par soucis de compatibilité avec l'UTF-8, nous avons pris en compte que des éléments portant des caractères spéciaux propres à l'UTF-8 (comme des accents) dans leur nom générait un header supplémentaire de type Pax (POSIX 2008), et comme ce genre de header possède un typleflag particulier ('x' ou 'g'), il est aisé de les ignorer (nous omettons volontairement les informations potentiellement utiles qu'il contiennent, puisque ce n'est pas testé dans les tests blancs). Cependant, les noms accentués provoquaient une sortie anormal de *ptar* avec une erreur indiquant une archive corrumpue. En fait, les caractères accentués ont une valeur hexadécimale de la forme 'ffffffc3' pour 'é' dans notre *header* (codés probablement comme des *signed int*). Seuls la partie 'c3' (dans cet exemple) nous intéressait, nous avons donc appliqué un second masque 0xFF sur chaque octet du header pour ne récupérer que les 8 bits de poids faible. Cela a résolu nos sorties anormales.

1.5 Étape 5 : Durabilité, parallélisation et derniers débug

Nous avons gardé l'étape 5 pour la fin, car elle nous semblait être la plus compliquée. En réalité, bien qu'il a fallu être très prudent durant la conception de cette étape à cause du *multithreading*, cette étape a été plus rapide à réaliser que l'étape 6, qui nous a longuement posée problème. Il a fallu tout de même ingérer toute une documention sur les threads POSIX.

Pour rendre le code compatible avec les threads, nous avons dû changer la fonction principale traitement() de utils.h sur plusieurs point, et également passer quelques variables locales en variables globales, notamment le descripteur de fichier/gzFile du open()/gzopen() de l'archive, le *FILE * logfile*, un mutex pour la lecture et 3 booléen d'état de corruption et d'*End Of File*. Il a fallut changer le protoype de base de traitement en *void *traitement(void *arg)* comme conseillé dans la documentation sur les threads POSIX. La création et le kill ($pthread\_join$) des threads dans le programme ne fut pas difficile à implémenter, il a fallu cependant protéger en lecture/écriture nos variables globales ! Sinon chaque thread peut à sa guise lire dans l'archive et la tête de lecture serait déplacée de façon incontrolable et imprévisible.

Pour cela, nous avous avons utilisé des $pthread\_mutex$, un pour les $read()/lseek()$ $/gzread()/gzseek()$ dans l'archive une fois ouverte, et un pour l'ecriture des éléments. En fait, nous avons réalisé plus tard, à l'issu du dernier test blanc, que le mutex d'écriture était complètement inutile et explosait littéralement le temps d'exécution du programme en multithread, alors que le temps d'exécution devait être un peu plus faible qu'un séquentiel. C'est à partir de ce moment qu'il a fallut être très prudent dans l'implémentation des mutex. Après quelques core dump nous avons fini par faire une release très stable et rapide en exécution du programme (version 1.7.5).

Tests d'exécution sur une archive compressée gzip pesant 3,9Mo :

Sans threads :

real 0m0.304s
user 0m0.133s
sys 0m0.131s

Avec 8 threads :

real 0m0.356s

user 0m0.147s

sys 0m0.284s

Le débogage du programme étant une partie important du projet, nous avons créé toute une floppée d'archives différentes pour stresser et roder notre programme. Parmis l'une d'elles, l'archive vice.tar et son homologue compressée en gzip vice.tar.gz, présente dans le dossier archives_test du dépôt github, ont été particulièrement meurtières mais redoutablement efficaces pour pointer les problèmes de notre code.

Les diverses méthodes de débogage sont listées dans la partie Debug du README.

2 Répartition du travaill effectué

	Conception	Codage	Tests et débug	Rapport
GARCIA	25h	12h	18h	2h
ZAMBAUX	20h	8h	25h	3h

3 Bibliographie

https://www.freebsd.org/cgi/man.cgi?query=tar&sektion=5
http://homepages.loria.fr/nussbaum/RS/
https://openclassrooms.com/
https://linux.die.net/
http://stackoverflow.com/
https://www.tutorialspoint.com/c_standard_library/
https://www.freebsd.org/
http://zlib.net/
http://arche.univ-lorraine.fr/course/view.php?id=12478
http://manpagesfr.free.fr/

sorting.h

```
/*
Fonctions de tri de header

Utile seulement après la décompression.
Liste et comptent chaque élément de chaque types.
Sert à trier les éléments récupérés après la décompression, en effet ils so
Il arrive alors que certains dossiers arrivent avant leur dossiers parents
*/

#ifndef INCLUDE_SORTING_H
#define INCLUDE_SORTING_H


/* Structure comprenant les 3 tableaux de header de chaque type (fichiers,

struct header_table {
headerTar *headDir; //Tableau des header de dossiers.
headerTar *headFile; //Tableau des header de fichiers.
headerTar *headSymlink; //Tableau des header de liens symboliques.
char *datas; //Tableau des données suivant le header (si fichier non vide).
int nbDir; //Nombre de dossiers.
int nbFile; //Nombre de fichiers.
int nbSymlink; //Nombre de liens symboliques.
};

//On fait un alias de la structure.
typedef struct header_table gzHeadertype;


/*
Analyseur d'archive : compte le nombre d'élément de chaque type contenu dan
Retourne une structure adaptée aux traitements.
Le paramètre doit être le tube nommé créé après la décompression.
L'ordre des données est le même que celui des fichiers.
*/

gzHeadertype analyse(const char *folder, FILE *logfile);


/*
Trieur : trie les éléments (les dossiers) avec un tri à bulle classique.
Ne retourne rien vu la simplicité de la fonction.
*/

void tribulle(gzHeadertype composition);
```

```
/*
Compte le nombre de token séparés par le délimiteur passé en paramètre.
Retourne ce nombre.
*/

int getnbtoken(char *name, const char *delim);


/*
Boucle sur les headers, dans le bon ordre, afin d'extraire/lister tous les
Retourne 0 si tout s'est bien passé, 1 si il y a eu au moins 1 erreur.
*/

int traitepostdecomp(gzHeadertype composition, FILE *logfile);


#endif
```