Julien VAUBOURG / Valentin KELLER
✉ julien@vaubourg.com
✉ valentin.keller@esial.net
2A groupe GG1

Développement d'un extracteur d'archives *tar* durable et parallèle

Rapport

RS

Le 1er décembre 2011

Table des matières

1 Conception

1.1 Travail réalisé

La version de *ptar* que nous avons implémenté a les caractéristiques suivantes :

- Extraction d'archives *tar*
- Parallélisme (cf. section correspondante)
- Durabilité
- Possibilité de lister les fichiers contenus dans l'archive
- Préservation de l'ensemble des attributs fournis par le format *ustar* (*mode, uid, gid* et *mtime*)
- Support de tous les types de fichiers que peut représenter le type *ustar* (fichiers réguliers, liens physiques, liens symboliques, périphériques de caractères, périphériques de blocs, répertoires et tubes nommés)
- Gestion des options avec *getopt*
  - Liste des fichiers (*-t*)
  - Mode verbeux, avec le temps d'extraction (*-v*)
  - Nombre de *threads* paramétrable (*-p*)
  - Chemin d'extraction (*-C*)
  - Aide (*-h*)
- Support de la compression grâce à des fonctions d'abstraction et un chargement dynamique des librairies (*gzip* implémenté), avec une reconnaissance du *mime-type*
- Vérification des *checksums*
- Meilleure durabilité, en synchronisant tous les types de fichier, y compris les répertoires
- Récupération du nombre d'erreurs au travers de l'ensemble des *threads*
- Possibilité de tester facilement avec *make test* et *make ztest*

1.2 Parallélisation

1.2.1 Première stratégie

Nous avons essayé d'implémenter la parallélisation de la façon **la plus simple et la plus cohérente possible pour le système**.

Nous avons commencé par implémenter une version qui, pour chaque fichier, lance un *thread* qui le met en mémoire, déverrouille un *mutex* pour laisser le programme principal passer au fichier suivant, puis l'écrit et le synchronise.

Cette version fonctionnait mais posait deux problèmes :

- Le fichier était mis systématiquement en mémoire : si ce dernier fait 10Go, l'ordinateur est saturé, le *swap* est utilisé et les performances sont radicalement amoindries.
- Les verrous qui sont une source d'erreur ne sont pas justifiés. En effet, étant donné que la seule fonction de la boucle principale est de lire les entêtes dans l'archive pour lancer les extractions, et qu'elle ne peut pas lire dedans tant que le *thread* qu'elle vient de lancer n'a pas fini de lire le fichier, autant que ce soit directement elle qui le lise. Ainsi, le *thread* ne lit plus le fichier, et il n'y a plus de raison d'utiliser un mécanisme de blocage.
- Les *threads* étaient joints par une interruption déclenchée par le signal *SIGUSR1*, dès lors qu'ils étaient prêts à mourir. Ce mécanisme, qui utilisait aussi un verrou, compliquait le programme inutilement.

Suite à cette analyse, nous avons décidé de rationaliser cette stratégie :

- Le problème des gros fichiers est réglé par un tampon qui lit dans l'archive et écrit les fichiers au fur et à mesure, et qui n'encombre jamais la mémoire de façon abusive.
- Nous avons retiré tous les verrous et rendu la lecture des fichiers au programme principal.
- Nous avons supprimé le signal, pour le remplacer par un système de tableaux avec des drapeaux qui indiquent si l'emplacement est libre (il peut aussi correspondre à un *thread* mort qui attend donc d'être joint, ou bien à un *thread*

en cours d'activité). Ainsi, pour lancer un nouveau *thread*, nous cherchons un emplacement libre ou correspondant à un *thread* mort (que nous joignons, avant de prendre la place). En fin de programme, nous parcourons tous les emplacements pour joindre *threads* morts et en cours d'activité, pour une vraie durabilité.

Nous avons alors constaté que la solution des tampons était bonne pour les gros fichiers, mais pénalisait les petits fichiers : les *threads* se contentaient de synchroniser le disque dur, alors qu'ils auraient pu écrire ces petits fichiers depuis la mémoire.

Confirmés dans notre idée par un précieux ami, nous avons intégré la possibilité pour les petits fichiers (plus petits qu'un tampon complet) d'être stockés en mémoire et d'être écrits directement par le *thread*. Nous en avons profité pour généraliser cette pratique, en lui faisant également systématiquement écrire le dernier tampon des gros fichiers.

Cette stratégie est visible et est parfaitement fonctionnelle **en version 17**.

1.2.2 Seconde stratégie

Nous avons continué notre analyse en nous demandant s'il n'y avait pas une solution qui soit encore meilleure. Être plus performant consisterait à réussir à faire faire plus de choses aux *threads* pour encore mieux exploiter la parallélisation. À ce stade, ce qui empêche de les laisser écrire plus de données (sans bloquer la boucle principale, sinon ça n'a aucun intérêt), c'est qu'il faut chaque fois lire l'archive pour remplir le tampon. Nous avons donc cherché des solutions pour faire lire l'archive par plusieurs *threads* en même temps.

Puisqu'on ne peut pas avoir plusieurs vraies têtes de lecture pour un seul fichier sans le dupliquer en mémoire, il faut utiliser une stratégie de *multiplexage* pour alterner lectures et écritures, en déplaçant la tête de lecture chaque fois.

Quand un *thread* vient de lire un morceau et l'écrit dans le fichier extrait, un autre prend la main pour repositionner la tête là où il s'était arrêté, pour lire un nouveau morceau de son fichier à extraire, et ainsi de suite.

Après quelques recherches, nous sommes partis avec les a priori suivants :

1. Ce qui est le plus coûteux en terme de temps sur un système, c'est le disque dur : déplacer la tête de lecture physique du disque en permanence est à notre sens suicidaire pour les performances.
2. Le système est intelligent et compte sur la prédictibilité de nos actions. Plutôt que de lire par petits morceaux comme on le pense, il préférera souvent mettre en mémoire plus de données, pour servir plus rapidement ce qui sera probablement notre prochaine attente. Le système est très bon sur la lecture de fichiers séquentiels parce que c'est hautement prévisible. En déplaçant la tête de lecture de façon aléatoire, nous perdrons toutes les optimisations qu'il nous met en place pour permettre à notre programme d'aller plus vite.
3. Les écritures sur le disque utilisent le cache : elles utilisent donc naturellement la mémoire, et sont peu coûteuses par rapport aux synchronisations (tant que le cache n'est pas plein, ce qui peut entrainer une synchronisation).

Cependant, une meilleure utilisation de la parallélisation pourrait permettre d'obtenir des délais plus courts.

Afin de vérifier, nous avons implémenté cette stratégie dans les dernières versions, en utilisant un sémaphore pour partager l'accès alternatif à l'archive entre les *threads*.

1.2.3 Comparaison des deux stratégies

Nous avons effectué les tests sur une archive contenant des petits fichiers et une autre en contenant de très gros, une dizaine de fois.

Pour les petits fichiers :

```
# Premiere strategie
real    0m18.701s
```

```
user    0m0.288s
sys     0m2.428s

# Seconde strategie
real    0m18.618s
user    0m0.192s
sys     0m2.292s
```

Pour les gros fichiers :

```
# Premiere strategie
real    0m0.008s
user    0m0.000s
sys     0m0.004s

# Seconde strategie
real    0m0.008s
user    0m0.004s
sys     0m0.000s
```

Les résultats sont plutôt décevants, et ne prouvent aucune différence remarquable entre les deux stratégies.

Nous en concluons que les problèmes liés à la seconde stratégie ont compensé le gain obtenu grâce à la meilleure parallélisation.

Une solution à exploiter pour améliorer l'une et l'autre des stratégies pourrait être de fixer la taille du tampon (*BUFFER_SIZE*) automatiquement en fonction de la mémoire disponible sur la machine.

1.3 Les deux blocs de caractères *NUL*

La documentation indique : « *The end of the archive is indicated by two records consisting entirely of zero bytes* ».

Il a été compliqué de respecter cette condition en faisant en sorte que la boucle principale n'ait pas à relire plusieurs fois le même bloc. Ainsi, un système de variables et de conditions permet de naviguer dans la boucle pour être capable de détecter un bloc de *NUL* (censé être un entête) et de traiter le suivant. Soit il s'agit à nouveau d'un bloc de *NUL* et il faut donc stopper la lecture de l'archive, soit ce n'est pas un bloc de *NUL*, et il faut donc revenir en arrière pour traiter ces deux blocs normalement.

En respectant ainsi strictement cette spécificité (probablement héritée des enregistrements à bandes magnétiques), dans le cas où il n'y a qu'un bloc de caractères *NUL*, celui-ci sera traité en tant qu'entête. Étant donné que sur la plupart des systèmes de fichiers récents, les deux seuls caractères interdits dans les noms de fichier sont *NUL* et le slash, c'est stupide. Nous avons décidé de nous contenter d'ignorer ce bloc *NUL* et de ne revenir que d'un bloc en arrière dans ce cas.

1.4 Les arborescences mal ordonnées

Nous nous sommes aperçus par hasard que dans certains cas, les dossiers de l'archive étaient rencontrés à la lecture après les fichiers qu'ils contiennent. Il semblerait que ce soit une habitude de certains gestionnaires d'archives graphiques.

Pour régler ce problème, nous avons décidé de créer tous les dossiers de l'arborescence des fichiers à extraire, et d'attendre de rencontrer les dossiers en questions pour leur associer leurs différents attributs.

1.5 Gestion des compressions

Nous avons souhaité rendre les compressions les plus transparentes possibles pour le programme. Ainsi, nous avons réécrit toutes les fonctions de manipulation de fichiers : *pfopen*, *pfread*, *pfclose*, *pfseek*, *pfeof*, *pftell*.

Ces fonctions s'utilisent comme leurs équivalents sans le *p* initial. La fonction *pfopen* se charge de trouver toute seule le type de compression s'il y en a une, et de lancer dynamiquement la librairie avec la fonction correspondante. Une structure est alors renseignée, avec toutes les fonctions nécessaires à la suite des traitement, et le type de compression détecté.

Toutes les autres fonctions, selon le contenu de cette structure, modifieront leur comportement pour gérer la décompression à la volée de façon totalement transparente pour le programme.

Nous n'avons décidé d'implémenter que *gzip*, puisque les autres types de compression peuvent maintenant s'intégrer très facilement et de la même façon, simplement en ajoutant des alternatives supplémentaires aux conditions des fonctions d'abstraction (dans *extractTools.c*).

À noter que nous avons passé un certains temps à écrire une fonction de décompression à la volée, avec un système complexe de tampon et de manipulation de la mémoire. Avant de nous apercevoir que *gzread* fonctionnait de la même façon. Le code est en annexe (*RIP*).

Il semble que *dlopen* ait plus de mal à trouver les *.so* que ce qui est écrit dans la documentation. Nous n'avons pas pu plus creuser, mais il peut être nécessaire d'indiquer en dur le chemin (dans *extractTools.c*) lors de son appel.

2 Travail en binôme

Le travail de groupe a été très intéressant. Ayant tous deux un niveau très différent en C, nous avons dû trouver des solutions pour réellement travailler ensemble sur le projet, sans que l'un se sente frustré tandis que l'autre se sente perdu.

Nous avons décidé de découper le projet en beaucoup de fichiers pour mieux nous répartir les tâches. L'un s'est occupé de l'extraction de base et d'options comme la décompression avec des librairies dynamiques, tandis que l'autre implémentait le système d'options, quelques types de fichiers, etc.

Pour les plus gros morceaux (étude des *threads*, etc), nous nous sommes souvent retrouvés pour programmer ensemble. Cette façon de faire a souvent été plus efficace que de travailler chacun de notre côté, et a permis un échange de savoirs intéressant.

Au niveau temps, nous avons estimés :

- Julien : 20h
- Valentin : 15h
- Étude des différentes stratégies et pur échange de connaissances autour du C, ensemble : 10h

3 Licence

GNU GPL v3

This program is free software : you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY ; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

(consulter le fichier *LICENCE*)

4 Remerciements

- /usr/bin/man (et en particulier la page 5 de *tar*)
- http://en.wikibooks.org/wiki/C_Programming/Strings
- http://www.koders.com/c/fid96336B4591FD0C5D2C4DADA2D264D3A04F21A934.aspx
- http://stackoverflow.com/questions/3054442/is-it-possible-to-use-threads-to-speed-up-file-reading?answertab=votes#tab-top
- http://stackoverflow.com/questions/2882771/plugin-architecture-in-c-using-libdl
- http://en.wikipedia.org/wiki/Dynamic_loading
- http://zlib.net/zpipe.c
- http://www.dwheeler.com/program-library/Program-Library-HOWTO/x172.html
- http://www.gzip.org/zlib/zlib_faq.html
- http://www.gzip.org/zlib/manual.html
- http://www.ietf.org/rfc/rfc1952.txt

5 Annexe

```
size_t pfread(void *ptr, size_t size, size_t nmemb, FILE *stream, int compressed) {
    if(compressed) {
        unsigned char in[size * nmemb];
        static unsigned char *out;
        static int sizeOut;
        int i, toRead, recovery, isEnd;
        char *p_write, *tmpOut;
        z_stream strm;

        strm.zalloc = Z_NULL;
        strm.zfree = Z_NULL;
        strm.opaque = Z_NULL;
        strm.avail_in = 0;
        strm.next_in = Z_NULL;

        /* Taille de ce que reclame le pfread */
        toRead = size * nmemb;

        /* Il n'y a aucune donnee restante de la derniere lecture */
        if(out != NULL) {

            /* Si ce qu'on veut lire est plus grand que ce qui est
             * actuellement dans le buffer, on il faut tout
             * recuperer du buffer et demander la lecture de ce qui
             * manque. Sinon, il faut juste prendre dans le buffer
             * ce qui nous intéresse. */
            recovery = toRead > sizeOut ? sizeOut : toRead;

            /* Soit il reste le delta a lire dans l'archive, soit il
             * n'y a plus rien à lire (toRead = 0) */
            toRead -= recovery;

            /* Copie de ce qui est a recuperer en tete du pointer de
             * retour */
```

```
            memcpy(ptr, out, recovery);

            /* Le buffer doit etre reduit d'autant que ce qui a ete
             * recupere */
            sizeOut -= recovery;

            /* Si en etant reduit, le buffer n'est toujours pas
             * vide, on reconstruit le buffer en copiant ce qui
             * reste en tete d'un nouveau pointeur, qui substitura
             * au buffer actuel */
            if((tmpOut = malloc(sizeOut)) != NULL)
                memcpy(tmpOut, out + recovery, sizeOut);

            /* Switch avec le buffer mis a jour */
            free(out);
            out = tmpOut;
        } else
            recovery = 0;

        /* Si tout a ete recupere du buffer, on arrete */
        if(toRead == 0)
            return size * nmemb;

        /* Decompression gzip (16) et zlib */
        inflateInit2(&strm, 32+MAX_WBITS);

        /* Lecture dans l'archive de ce qui reste a lire (on recuperera
         * plus que ce qu'on demande, puisque qu'on recupere du
         * compresse */
        strm.avail_in = fread(in, 1, toRead, stream);
        strm.next_in = in;

        i = 0;

        /* Lecture dans le fichier compresse jusqu'a ce qu'on a ce qu'il
         * nous faut */
        do {
            /* Le buffer grandit en fonction de ce qu'on recupere,
             * puisqu'on ne peut pas prevoir quelle sera la taille
             * des donnees une fois decompressees */
            sizeOut = toRead * ++i;
            out = realloc(out, sizeOut);

            strm.avail_out = toRead;

            /* Lecture a la fin de ce qui est deja lu */
            strm.next_out = out + toRead * (i - 1);

            isEnd = inflate(&strm, Z_NO_FLUSH);

        } while (strm.avail_out == 0 && !isEnd);

        inflateEnd(&strm);

        /* Taille de ce qui est a recupere dans le buffer (puisqu'on a
         * forcement recupere plus que ce qu'il fallait, en decompresse */
        toRead = (i == 1) ? toRead - strm.avail_out : toRead;

        /* Copie a la suite de ce qui eventuellement deja ete recupere
         * du buffer, de ce qu'on vient de recuperer */
        memcpy(ptr + recovery, out, toRead);

        /* Taille du buffer actuel, sans ce qui n'a pas ete utilise */
        sizeOut -= strm.avail_out - toRead;

        /* Si il y a un reliquat, on recree le buffer avec uniquement ce
         * reliquat, a la bonne taille */
        if((tmpOut = malloc(sizeOut)) != NULL)
            memcpy(tmpOut, out + toRead, sizeOut);

        /* Switch des buffers */
        free(out);
        out = tmpOut;

        isTherePfreadStock = sizeOut > 0;

        return toRead;
    } else
        return fread(ptr, size, nmemb, stream);
}
```