mickael.hoerdt@hesge.ch 2018/2019

Programmation Système/Systèmes d'exploitation
Mini-projet

Un compresseur/décompresseur de fichiers au format Android Sparse File

1. Contexte

Les mises à jour du système Android se présentent sous forme d'images disque dont une grande partie du contenu est majoritairement composée de blocs **vides** de données, c'est à dire uniquement composés de bytes dont la valeur hexadécimale est `0x00`.

Ces mises à jour étant transmises le plus souvent par le réseau (Android Over The Air (OTA) updates), envoyer ces blocs vides à distance représente une perte d'efficacité très importante : l'information envoyée est toujours la même, tout en utilisant la majorité de la bande passante disponible. On pourrait zipper l'image avant de l'envoyer, mais sur des appareils disposant que de faibles ressources, décompresser une image disque au format zip avant de procéder à une mise à jour de partition est au mieux très coûteux en ressources, et au pire irréalisable sur la plupart des appareils qui exécutent Android. Une solution raisonnable doit autoriser la mise à jour à partir d'un flux de données plutôt qu'un fichier. C'est pour cette raison un système de compression d'images disques très simple appelée « Android Sparse File » a été conçu.

Le principe général est le suivant : Les données du fichier sont découpées en **blocs** de taille fixe, qui est indiquée dans l'entête de fichier préfixée au Sparse File. Ensuite, les données des blocs **non-vides** sont conservées tels quels dans le fichier, alors que les blocs consécutifs de données **vides** sont remplacés par une instruction qui indique combien de blocs **vides** suivent l'instruction. Les blocs **non-vides** mais composés de bytes ayant tous la même valeur sont aussi remplacés par une instruction, suivi de la valeur du byte qui devra être contenu dans ces blocs à la décompression (cf **annexe 1**)

Visuellement, le format d'un fichier Sparse se présente de la manière suivante :

Entête de fichier (28 bytes)	Entête de k blocs **Non vide** (12 bytes)	Bloc 1	...	Bloc k	Entête de m blocs **vide** (12 bytes)	Entête De n blocs **Non vide** (12 bytes)	Bloc 1	...	Bloc n

On y trouve en début d'image l'entête de fichier, qui contient notamment le nombre total de blocs du fichier décompressé, la taille d'un entête de bloc et un checksum de l'image au format non compressé. Viennent ensuite une série d'entête de blocs de 12 bytes, qui indiquent si il s'agit d'une entête de blocs vides ou non, le nombre de blocs qui suivent ainsi que leur longueur totale en bytes, en incluant la taille de l'entête. La définition exacte de l'entête de fichier ainsi que ceux des blocs est indiquée en annexe.

2. Cahier des charges

2.1 Utilitaires de compression/décompression en ligne de commande

L'objectif de ce projet est d'implémenter en langage C et sur la base des d'appels système Unix vus en cours, un outil de compression d'image Sparse avec son outil de décompression associé. **Vos outils n'auront pas besoin de gérer les CRC32 de l'image** (ils sont toujours à mis à zéro lors de la compression, et non interprétés lors de la décompression), mais devront gérer la lecture/l'écriture du reste de l'entête ainsi que les trois types d'entête de blocs indiqués en **Annexe 1.**

1. Votre interface au compresseur/décompresseur sparse doit avoir la forme de deux exécutables en ligne de commande dont l'utilisation sera la suivante :

   ```
   sparse2img <sparse_file> <img_file>
   img2sparse <img_file> <sparse_file>
   ```

2. La première commande prendra en entrée un fichier compressé au format sparse dont le path est indiqué par le paramètre `sparse_file`, et décompressera son contenu dans le path du fichier indiqué par le paramètre `img_file`. La deuxième commande fera l'inverse : elle compressera le contenu d'un fichier indiqué en paramètre dans un nouveau fichier sparse indiqué en deuxième paramètre.

Important : il est autorisé et même encouragé d'aller lire le code source en C de l'outil de compression/décompression implémenté par d'Anestis BetchSoudi[1] pour s'inspirer, par contre il est formellement interdit de copier les structures de données ou les fonctions utilisées par cet outil : vous devrez proposer vos propres structures de données et fonctions, en particulier pour l'outil de décompression.

2.1 Tests du bon fonctionnement de vos outils.

Vos outils devrons inclure un ensemble de tests qui vérifient le bon fonctionnement des outils de compression/décompression. Ces tests devront pouvoir être exécutés automatiquement à l'aide d'une cible de votre fichier `Makefile` appelée « tests ». L'utilisateur lancera donc la série de tests que vous aurez conçus par la commande « `make tests` ». Vos tests devront mettre en œuvre des shellscript bash vérifiant automatiquement le bon fonctionnement de votre implémentation dans des cas prédéfinis.

1 https://github.com/anestisb/android-simg2img

2.2 Modification des outils pour qu'ils gèrent aussi la compression/décompression depuis l'entrée standard.

Vos outils devront aussi pouvoir fonctionner en mode filtre, c'est à dire lire les données depuis l'entrée standard et produire leur résultat sur la sortie standard plutôt que dans un fichier donné en paramètre. Il devra donc être possible de les lancer de la manière suivante dans un shell, illustrant que le contenu d'un fichier sparse envoyé dans `sparse2img` puis dans `img2sparse` est équivalent à la fonction identité sur ce contenu, de même pour l'opération inverse.

```
# cat sparse_file | sparse2img | img2sparse > same_sparse_file
# diff sparse_file new_sparse_file
# echo $?
0

# cat img_file | img2sparse | sparse2img > newimg_file
# diff img_file newimg_file
# echo $?
0
```

3. Modalités de rendu

Le projet est à réaliser par groupe de 2 ou 3 personnes. **L'étape 2.2 est obligatoire pour les groupes de 3 personnes, optionnelle pour les binômes**.

Un rapport de maximum **5 pages** au format **pdf** décrivant les différents composants/blocs/functions de votre implémentation ainsi que leur interdépendances devra être rendu par mail avant le **23/12/2018**.

Un rapport au format **pdf** documentant votre implémentation devra être rendue par mail au plus tard le **23/1/2019** avant 23h59.

Votre code C doit être compilable via un fichier `Makefile` sous GNU/Linux et `gcc` sur les machines de l'école. Si votre code ne compile pas sur les machines de l'école, vous serez pénalisé dans l'évaluation. L'ensemble du projet devra être stocké sur le serveur git du département ITI de HEPIA sur https://githepia.hesge.ch. L'URL de votre projet devra être transmise au professeur avant le **1/12/2018**

Annexe 1 : format et types des entêtes de Sparse File

```
typedef struct sparse_header {
    uint32_t magic;               /* 0xed26ff3a */
    uint16_t major_version;       /* (0x1) - reject images with higher major versions */
    uint16_t minor_version;       /* (0x0) - allow images with higer minor versions */
    uint16_t file_hdr_sz;         /* 28 bytes for first revision of the file format */
    uint16_t chunk_hdr_sz;        /* 12 bytes for first revision of the file format */
    uint32_t blk_sz;            /* block size in bytes, must be a multiple of 4 (4096) */
    uint32_t total_blks;          /* total blocks in the non-sparse output image */
    uint32_t total_chunks;        /* total chunks in the sparse input image */
    uint32_t image_checksum;      /* CRC32 checksum of the original data, counting "don't
                                     care" */
                                /* as 0. Standard 802.3 polynomial, use a Public Domain */
                                /* table implementation */
} sparse_header_t;

#define SPARSE_HEADER_MAGIC 0xed26ff3a

#define CHUNK_TYPE_RAW          0xCAC1
#define CHUNK_TYPE_FILL         0xCAC2
#define CHUNK_TYPE_DONT_CARE    0xCAC3

typedef struct chunk_header {
    uint16_t chunk_type;        /* 0xCAC1 -> raw; 0xCAC2 -> fill; 0xCAC3 -> don't care */
    uint16_t reserved1;
    uint32_t chunk_sz;          /* in blocks in output image */
    uint32_t total_sz;          /* in bytes of chunk input file including chunk header
                                   and data */
} chunk_header_t;

/* Following a Raw or Fill or CRC32 chunk is data.
 *  For a Raw chunk, it's the data in chunk_sz * blk_sz.
 *  For a Fill chunk, it's 4 bytes of the fill data.
 *  For a CRC32 chunk, it's 4 bytes of CRC32
 */
```