Android Sparse File

Boutay Matthias, Nogueiras Blanco David, Paschoud Nicolas

Introduction

Le projet consiste à réaliser un compresseur/décompresseur de fichier au format Android Sparse File. Ces outils sont implémentés en C et sur la base des d'appels système Unix. Dans ce rapport, nous vous présentons différents aspects de l'outil de compression et de décompression que nous avons créé.

Le code en annexe permet d'effectuer la décompression ou la compression d'un fichier compressé au format Android Sparse File. De plus, nous gérons l'utilisation de l'entrée et de la sortie standard.
Le programme peut-être compilé et testé à l'aide du fichier makefile. Les tests sont effectués sur la base des fichiers fournis test.img et test.simg.

Architecture des fichiers

Nous avons organisé les fichiers de la façon suivante : à la racine du projet, nous pouvons retrouver les deux fichiers principaux permettant la compression et la décompression. Ils se nomment respectivement *img2sparse.c* et *sparse2img.c*. On y retrouve également le *makefile*, le *tests.sh* et le fichier *.gitlab-ci.yml* permettant de faire effectuer différents test pour l'intégration continue. Finalement, on y retrouve les dossiers *library* et *tests*. Le premier contient les fichiers d'extension .h utilisés dans les deux fichiers principaux. Ils définissent la structure de donnée, certaines fonctions implémentées dans la librairie ainsi que certaines constantes. Le dossier de tests finalement contient les 2 programmes de tests implémentés.

Structure de données

Pour comprendre notre structure de donnée, il faut savoir que l'Android Sparse File est un format de compression découpant un fichier en blocs de taille fixe. Ces blocs sont répartis dans des chunks contenant chacun une en-tête *header*. Le fichier possède également un en-tête contenant les informations permettant la décompression.
Notre structure de données principale se présente comme suit : Il y a deux structures de données qui servent à stocker l'en-tête de fichier et les headers de chaque chunk. Ces deux structures de données sont construite selon la norme qui définit les sparse file. En dehors de ces deux structures, les programmes de compression et de décompression ont chacun leur propre fonctionnement incluant des fonctions ou/et structures spécifiques. Leurs fonctionnements sont expliqués dans leurs parties respectives.

Décompression

Pour décompresser un fichier compressé au format Android Sparse File en utilisant notre programme, il faut lancer dans le dossier contenant notre programme la commande ./sparse2img <sparse_file> <target_file>

Notre programme est actuellement constitué des fonctions suivantes :

La fonction *main* est la fonction principale qui s'exécute au lancement du programme. Elle prend en paramètres les fichiers d'entrée <sparse_file> et de sortie <img_file>. Elle retourne le succès ou non du programme.

Les autres fonctions sont appelées par la fonction main et retournent toutes un buffer contenant les informations à traiter sauf une.
Les fonctions read_file_header et read_chunk_header lisent respectivement le header du fichier compressé et le header du prochain chunk à partir d'un file descriptor fd.
La fonction read_to_next_chunk_header_inclusive lit la suite du fichier en entrée jusqu'au suivant chunk header. Il décrémente une variable chunks2read en paramètre de la fonction. Il met également à jour le paramètre chunk_header contenant toutes les informations du courant chunk à traiter. Il prend également en paramètre la variable blk_sz indiquant la taille d'un bloc du fichier.
Finalement, la fonction retranscribe_blks retranscrit un certain nombre de blocs contenus dans le buffer dans le fichier de sortie. La fonction retourne ensuite le nombre de bytes écrit dans le fichier.

Notre programme fonctionne ainsi :
Il commence par lire dans le fichier d'entrée l'entête de fichier à l'aide de la fonction *read_file_header*. Il enregistre ces informations récupérées du buffer. Parmi ces informations retranscrite, il y a le nombre total de *chunks* dans le fichier. Une variable *chunks2read* est

initialisée avec cette valeur. Cette variable sera décrémentée au fur et à mesure que des en-têtes de *chunk_header* seront lues.

La suite du programme se déroule dans une boucle qui commence par identifier quel est le *chunk_type* et agit en conséquence. Dans les trois cas, nous allons définir une taille différente pour le buffer :

Dans le cas de 0xCAC1, le programme copie les blocs dans le buffer en ajoutant le prochain header, puis écrit dans le fichier les bytes à écrire et affecte les nouvelles valeurs du prochain header dans la structure *chunk_header*.

Pour 0xCAC2, la subtilité réside dans la création d'un buffer temporaire. En effet, vu qu'il y a un pattern sur 4 bytes, il faut réécrire celui-ci sur le nombre de blocs multiplié par la taille des blocs dans le fichier de destination. Grâce à un buffer temporaire, le programme peut ainsi écrire avec un seul appel système, le nombre d'occurrences du pattern. Il termine ensuite de la même manière que pour la méthode RAW.

Finalement, 0xCAC3 va quant à lui uniquement lire le prochain header (s'il y en a) et appliquer le même fonctionnement que 0xCAC2 avec un buffer rempli de 0.

Compression

Pour compresser un fichier non compressé au format Android Sparse File en utilisant notre programme, il faut lancer dans le dossier contenant notre programme la commande ./img2sparse <target_file> <sparse_file>

Pour implémenter la compression, nous avons choisi de définir la taille des blocs de manière fixe à 4096 octets. Nous avons créé une structure chunk permettant de retenir le nombre *n* de blocs contenus dans le chunk, le type *type* de chunk auquel nous avons à faire ainsi que le modèle de bloc *blk_model* contenu dans le chunk. Cette structure nous permet au fur et à mesure de la lecture d'un fichier de modifier le tronçons et de retenir les informations lues au préalable. Le type de chunk est une énumération qui peut prendre trois différentes valeurs : *UNDEFINED*, *SAME* et *DIFFERENT*. Dans le premier cas, cela signifie que nous ne savons pas encore à quoi ressemblera le chunk. Si l'attribut est *SAME*, alors le chunk est une succession de blocs similaires. Nous aurons donc un chunk_type de 0xCAC2 où 0xCAC3 dans le chunk_header en fonction de la valeur contenue dans le modèle de bloc. Pour le dernier cas, il s'agit d'une suite de blocs ne répondant à aucun pattern.
Finalement, nous avons créer une énumération de décision *Decision*. Elle peut prendre les valeurs *READ_NEXT*, *RETRANSCRIBE* et *SET_UP*. Cette énumération sera utilisée pour savoir quelles actions effectuer.

Notre implémentation fonctionne ainsi :

Au démarrage, nous écrivons dans le fichier de sortie un sparse_header vide suivi d'un chunk_header vide. Nous initialisons également un chunk.
Tant que nous n'avons pas lu le fichier en entrée en entier, nous le lisons avec un descripteur de fichier bloc par bloc. Nous retenons dans un buffer *read_buffer* ce que nous avons lu et dans une variable *nb_read* le nombre d'octets qui ont été lu. Nous prenons une décision à l'aide de la fonction *what_to_do*. Cette fonction prends en paramètre le chunk ainsi que le buffer de lecture pour retourner sa décision.
Dans le cas où la décision est *SET_UP*, le programme va regarder s'il a assez d'information pour mettre à jour le type du chunk. Si le chunk type devient *DIFFERENT*, le modèle de bloc va être copié dans le buffer des données *datas_buffer* contenant la suite de bloc différents. Ensuite, le programme va copier le buffer de lecture dans le modèle de bloc du chunk et ajouter 1 à son attribut *n*.
Dans le cas où la décision est *READ_NEXT*, si le chunk est de type *DIFFERENT*, il va mettre à jour le buffer des données avec le modèle de bloc et ce dernier avec le buffer de lecture. Dans le cas où le buffer de données a atteint sa taille maximale, il est entièrement retranscrit et vidé. Cependant, nous ne changeons pas de chunk tant que la décision de retranscription n'est pas prise.
Finalement, le programme ajoute 1 à l'attribut n du chunk.
La dernière décision possible est celle de la retranscription. Elle est prise lorsque le dernier bloc lu met fin à une série de blocs similaires ou différents. Les actions effectuées à la suite de cette décision est une retranscription du buffer de données s'il possède des données, une réécriture du chunk header à l'aide des informations du chunk ainsi qu'une redéfinition des informations du chunk pour qu'elles soient compatibles avec les premières informations connues. Un nouveau header vide est créé qui sera à son tour réécrit une fois les détails du suivant chunk connus.

Finalement, après avoir lu toutes les données, le programme s'occupe de finaliser le fichier compressé en écrivant les données du dernier chunk et en réécrivant le file header complet.

Entrée et sortie standard

Afin de pouvoir utiliser l'entrée et sortie standard, nous avons mis en place un système de reconnaissance des paramètres. Si un seul paramètre est détecté (l'exécutable), le programme va écouter l'entrée standard et créer deux fichier temporaire. Le premier, afin d'y stocker à l'aide d'un buffer d'une taille de 4 Ko (qui sera rempli puis vidé jusqu'à avoir lu toute l'entrée) et les commandes read() et write(). Le second, servira de destination au résultat. A la fin du programme, nous ouvrons le second fichier temporaire et réécrivons dans la sortie standard, à l'aide d'un buffer de 4 Ko (qui sera à nouveau rempli puis vidé jusqu'à avoir tout écrit dans la sortie standard), et supprimons les fichiers temporaires.

Tests

Lors du développement du programme, un ensemble de tests unitaires ont été réalisés afin de tester certaines fonctions de notre programme. Le système d'intégration continue à lui aussi été utilisé afin de s'assurer que lors de l'implémentation de nouvelles fonctions, le programme fonctionne toujours comme désiré.
Deux programmes de tests ont été créés, l'un pour tester les fonctions de lecture (reading_tests) et l'autre pour tester les fonctions d'écritures (writing_tests). Ces deux programmes de tests se basent sur les fichiers image fournis avec l'énoncé. Ceux-ci sont compiler chacun leur tour par un shell script test.sh. Celui-ci vérifie le bon déroulement de la compilation puis de l'exécution des programmes. Si quelque chose se passe mal dans les programmes de tests, le shell script informe que quelque chose s'est mal passé et s'arrête une fois les deux programmes exécutés.
Le shell script *test.sh* exécute aussi des tests fonctionnels. Il va tester si le fichier compresser / décompresser par le programme est le même que celui fourni. Une fois les tests unitaires terminés, le script va tester les deux programmes via les entrées et sorties standard :

```
cat sparse_file | sparse2img | img2sparse > same_sparse_file.simg
cat img_file | img2sparse | sparse2img > newimg_file.img
```

Conclusion

En conclusion, nous avons pu implémenter toutes les fonctionnalités nécessaires à la compression et la décompression de fichiers tout en gérant les entrées et sorties standard. Cependant, nous avons recensé quelques bugs ou améliorations qui pourraient être faites sur notre travail.
Actuellement, nous savons que si nous décompressons un fichier que nous avions au préalable compressé, nous n'obtenons pas le même fichier qu'au départ dans le cas où le fichier de base ne faisait pas une taille qui est un multiple de 4096, soit le multiple de la taille d'un bloc. En effet, le fichier résultant possède à la fin des octets supplémentaires jusqu'à atteindre la taille du bloc final.
Une amélioration majeure que nous aurions pu apporter est une gestion des entrées et sorties standard durant le processus de compression ou décompression plutôt que de le gérer avec des fichiers temporaires au début et à la fin du processus.
Finalement, nous pourrions améliorer la compression en ayant une taille de bloc variant selon la taille du fichier à compresser. En effet, un très petit fichier ne serait pas compressé par notre implémentation.