EPFL CS212 : ImgStore – Système de fichiers orienté images — 09 : finalisation de `read` et `insert`

E. Bugnion & J.-C. Chappelier EPFL

Rev. 2021.04.22 / 1

Table des matières

Projet de Programmation Système – CS212 – 2021

ImgStore – Système de fichier orienté images

09 : finalisation de `read` et `insert`

Introduction

Cette semaine, vous implémentez les commandes `read` (extraction d'une image depuis la base d'images) et `insert` (insertion d'une image dans la base). Pour

ceci, vous utilisez des fonctionnalités développées les deux semaines précédentes.

Attention :

1. le travail demandé cette semaine est plus complexe que les semaines précédentes, et vous prendra probablement plus de temps que d'habitude ; commencez tôt et venez avec des questions le mercredi matin ; pensez également à bien vous répartir le travail ;
2. le travail de cette semaine constitue par ailleurs, avec celui des semaines 7 et 9 et la correction de votre travail des semaines 4 à 6, le second rendu de ce projet (cf barème). Tout le code du « *command line manager* » que vous avez produit jusqu'ici sera évalué. Veillez donc bien à mettre à jour tous vos fichiers (en particulier suite aux remarques de la correction précédente) et à bien rendre le tout.

Pour ces raisons, vous disposez d'une semaine supplémentaire pour rendre votre projet (dimanche 16 mai) ; cela *ne* veut *pas* dire de commencer plus tard, mais simplement d'étaler un peu plus le travail si nécessaire !

Matériel fourni

Comme les semaines passées, le travail de cette semaine se construit sur tout votre travail des semaines précédentes, mais nous vous fournissons du matériel de test : 2 scripts et 4 images.

Travail à faire

Rajouter les déclarations de fonctions de la semaine

Tout d'abord, rajouter dans `imgStore.h` les déclarations manquantes en remplacement des `TODO WEEK 09` :

1. une fonction `resolution_atoi()` prenant en argument une chaîne de caractères (inchangée) et retournant un entier ;
2. une fonction `do_read()` prenant en argument :
   — un identifiant d'image (chaîne de caractères) ;
   — un entier représentant (le code d')une résolution d'image ;
   — un pointeur de pointeur sur des « caractères » (utilisés en tant qu'octets en fait), c'est l'adresse d'un « tableau » d'octets ;
   — un entier non signé sur 32 bits, représentant la taille de l'image ;
   — une structure `imgst_file` (de laquelle on lira l'image, mais elle pourra être modifiée en raison de la stratégie de construction « paresseuse » (« *lazy* ») des images « *small* » et « *thumbnail* ») ;
   cette fonction retourne un entier (code d'erreur) ;
3. une fonction `do_insert()` prenant en argument :
   — une image sous forme d'un pointeur (« tableau ») de « caractères » (utilisés en tant qu'octets en fait), non modifiés par cette fonction ;

— la taille de l'image, de type `size_t` ;
— un identifiant d'image (chaîne de caractères) ;
— et une structure `imgst_file` (dans laquelle on ajoutera l'image) ;
cette fonction retourne un entier (code d'erreur).

Note : ce dernier paramètre manque, par erreur, dans le commentaire Doxygen dans `imgStore.h` ; vous pouvez bien sûr l'ajouter (à la doc) si vous voulez.

Implémenter des fonctions utilitaires

Avant d'implémenter réellement les fonctions `do_read()` et `do_insert()`, il vous sera utile de commencer par quelques fonctions utilitaires :

`resolution_atoi()` L'objectif de cette fonction est de transformer une chaîne de caractères spécifiant une résolution d'image dans une des énumérations spécifiant un type de résolution, à savoir :

— retourner `RES_THUMB` si l'argument est soit `"thumb"` ou soit `"thumbnail"` ;
— retourner `RES_SMALL` si l'argument est `"small"` ;
— retourner `RES_ORIG` si l'argument est soit `"orig"` ou soit `"original"` ;
— retourner `-1` dans tous les autres cas, y compris si l'argument est `NULL`.

Cette fonction doit être implémentée dans `tools.c`.

Elle sera nécessaire pour traiter les arguments de ligne de commande du programme `imgStoreMgr`.

`get_resolution()` Ensuite, la fonction `get_resolution` dont le but est de récupérer la résolution d'une image JPEG. Elle a la signature suivante :

```
int get_resolution(uint32_t* height         ,
                   uint32_t* width          ,
                   const char* image_buffer ,
                   size_t image_size        );
```

Protoypez la fonction dans `image_content.h` et implémentez la dans `image_content.c`.

Cette fonction prend en entrée `image_buffer`, qui est un pointeur sur une région de mémoire contenant une image JPEG que l'on lira avec la fonction `vips_jpegload_buffer()`, et `image_size` qui est la taille de cette région.

Utilisez la bibliothèque VIPS (cf semaines 2 et 7) pour récupérer la résolution (longueur et largeur) de l'image et la stocker dans les deux paramètres `height` et `width`.

La fonction retourne un code d'erreur : `ERR_NONE` s'il n'y a pas de problème, ou `ERR_IMGLIB` en cas d'erreur de VIPS.

NOTE : le prototype de `vips_jpegload_buffer()` est erroné en ce sens que son premier argument devrait être `const void*` (au lieu de `void *` ; on *lit* depuis ces données !). En effet, en allant voir le code, cette fonction ne fait appel qu'à `vips_blob_new()` dont le second argument est correctement qualifié de `const void *` (bon, même si il y a cet horrible casting). Vous pouvez donc sans aucun risque (j'espère !) passer `image_buffer` à `vips_jpegload_buffer()` (en le castant, malheureusement... `:-( >:-o`).

`do_insert()`

La fonction `do_insert()` rajoute une image dans le « imgStore ». Créez un nouveau fichier `imgst_insert.c` pour l'implémenter.

La logique d'implémentation contient plusieurs étapes, dans un ordre à respecter.

Trouver une position de libre dans l'index Avant tout, vérifier que le nombre actuel d'images est moins que `max_files`. Retourner `ERR_FULL_IMGSTORE` si ce n'est pas le cas.

Vous devez ensuite trouver une entrée vide dans la table `metadata`. Lorsque c'est le cas, vous devez :

- placer la valeur hash SHA256 de l'image dans le champ `SHA` (revoir si nécessaire la semaine 1 pour le calcul des SHA256) ;
- copier la chaîne de caractères `img_id` dans le champs correspondant ;
- stocker la taille de l'image (passée en paramètre) dans le champs `RES_ORIG` correspondant (attention au changement de type).

De-duplication de l'image Appeler la fonction `do_name_and_content_dedup()` de la semaine passée en utilisant les bons paramètres. En cas d'erreur, `do_insert()` retourne le même code d'erreur.

Ecriture de l'image sur le disque Tout d'abord, vérifier si l'étape de déduplication a trouvé (ou non) une autre copie de la même image. Pour ce faire, tester si l'`offset` de la résolution d'origine est nul (revoir si nécessaire la fonction `do_name_and_content_dedup()`).

Si l'image n'existait pas, écrire son contenu à la fin du fichier. Pensez également à terminer proprement l'initialisation de ses méta-données.

Mise à jour des données de la base d'images Utiliser la fonction `get_resolution()` (voir plus haut) pour déterminer la largeur et hauteur de l'image. Copiez ces valeurs dans les champs `res_orig` du `metadata`.

Mettre à jour les champs du `header` de la base d'images.

Enfin, il ne vous reste plus qu'à écrire le `header`, puis l'entrée `metadata` *correspondante* sur le disque (votre code **ne** doit **pas** écrire toutes les méta-données

sur disque à chaque opération !).

`imgst_read`

La deuxième fonction principale de la semaine est `do_read()`, à implémenter dans `imgst_read.c`.

Cette fonction doit tout d'abord retrouver dans la table des méta-données l'entrée correspondant à l'identifiant fourni.

En cas de succès, déterminer tout d'abord si l'image existe déjà dans la résolution demandée (`offset` ou `size` nul). Si ce n'est pas le cas, appeler la fonction `lazily_resize()` des semaines précédentes pour créer l'image à la résolution voulue. (*Note : a priori*, cela ne devrait jamais être le cas pour `RES_ORIG`).

A ce moment, la position de l'image (dans la bonne résolution) dans le fichier est connue, ainsi que sa taille ; il vous est possible de lire le contenu de l'image du fichier dans une région de mémoire allouée dynamiquement.

En cas de succès, les paramètres de sortie `image_buffer` et `image_size` doivent contenir l'adresse en mémoire et la taille de l'image.

Faites bien attention à traiter les cas d'erreur possibles : * retourner le code d'erreur reçu en cas d'erreur d'une fonction interne ; * retourner `ERR_IO` en cas d'erreur de lecture ; * retourner `ERR_OUT_OF_MEMORY` en cas d'erreur d'allocation de mémoire ; * et retourner `ERR_FILE_NOT_FOUND` si l'identifiant demandé n'a pas pu être trouvé (au sens : le fichier image recherché dans notre « système de fichiers » à nous, la imgStore, n'a pas pu être trouvé).

Remarque : au cas où certain(e)s d'entre vous se poseraient la question : notez bien que le `read` sur une image dupliquée n'entraîne aucune modification de ses duplicata. En effet `lazily_resize()` n'a aucun impact sur les *autres* images que celle considérée (et a été écrite avant `do_name_and_content_dedup()`). Un tel comportement (qui **doit** être celui de votre programme) n'est pas bien grave en pratique car :

1. rechercher tous les duplicatas à chaque opération serait trop coûteux en général ;
2. ce sera justement le rôle du « garbage collector » que de faire ce type de travail, une fois de temps en temps (p.ex. toutes les nuits) et sur *toute* la base d'images ;
3. et *a priori* ce devrait souvent être le cas qu'une image dupliquée soit, quand elle arrive, le duplicata d'une image déjà utilisée, donc d'une image qui a *déjà* ses versions « small » et « thumb » ; dans ce cas, la nouvelle image insérée (duplicata) partagera déjà bel et bien ses versions « small » et « thumb » avec son original déjà présente dans la base.

Intégration dans le programme principal

Dans `imgStoreMgr.c`, implémenter les deux nouvelles commandes en suivant la même logique que pour les commandes déjà existantes.

Nous vous recommandons d'écrire des fonctions utilitaires pour vous simplifier la taĉhe, comme par exemple `read_disk_image()`, `write_disk_image()` ou encore `create_name()` (pour la convention de nommage ci-dessous).

Pour l'insertion (`do_insert_cmd()`), vérifier que le nombre actuel d'images est moins que `max_files`. Retourner `ERR_FULL_IMGSTORE` si ce n'est pas le cas (comme dans `do_insert()` ; les deux se justifient).

Pour l'extraction des images (lecture, `do_read_cmd()`), les fichiers créés doivent suivre la convention de nommage suivante :

```
image_id + resolution_suffix + '.jpg'
```

où :

- `image_id` est l'identifiant de l'image ;
- et `resolution_suffix` correspond à « `_orig` », « `_small` » ou « `_thumb` ».

Par exemple, la lecture de l'image « `pic1` » en résolution « *small* » créera le fichier `pic1_small.jpg`.

Mettre `help` à jour

Modifiez la commande `help` afin de refléter les nouvelles commandes :

```
> ./imgStoreMgr help
imgStoreMgr [COMMAND] [ARGUMENTS]
  help: displays this help.
  list <imgstore_filename>: list imgStore content.
  create <imgstore_filename> [options]: create a new imgStore.
      options are:
          -max_files <MAX_FILES>: maximum number of files.
                                  default value is 10
                                  maximum value is 100000
          -thumb_res <X_RES> <Y_RES>: resolution for thumbnail images.
                                  default value is 64x64
                                  maximum value is 128x128
          -small_res <X_RES> <Y_RES>: resolution for small images.
                                  default value is 256x256
                                  maximum value is 512x512
  read   <imgstore_filename> <imgID> [original|orig|thumbnail|thumb|small]:
      read an image from the imgStore and save it to a file.
      default resolution is "original".
  insert <imgstore_filename> <imgID> <filename>: insert a new image in the imgStore.
  delete <imgstore_filename> <imgID>: delete image imgID from imgStore.
```

Pour vos tests, pensez à « rapatrier » `provided/tests/helptext_week09.sh` comme le nouveau `helptext.sh` dans votre `done/tests`.

Tests

En utilisant le matériel fourni, vous pouvez tester les commandes d'insertion et de lecture. Vous pouvez aussi (comme toujours), faire vos propres tests à la main et utiliser l'application de visualisation d'images de votre choix pour vérifier que les images ont été correctement transformées (et extraites).

Nous vous recommandons par ailleurs de penser à tester toutes les commandes sur tous les cas particuliers.

ATTENTION ! Pour celles et ceux qui travaillent sur les VM `IN_SC`, utilisez les versions fournies dans `tests-VIPS.8.4.5/`, lorsqu'il y en a, plutôt que celles dans `tests` ; ceci en raison d'une version plus ancienne (8.4.5) de la libvips sur ces VM.

Rendu

Comme dit en introduction, le travail de cette semaine constitue, avec la révision du travail des semaines 4 à 6 et le (nouveau) travail des semaines 7 et 8, le **second rendu** du projet.

Le délai pour effectuer ce rendu est fixé au **dimanche 16 mai 23:59** ; mais veillez à ne pas accumuler de retard et bien vous répartir le travail.

Comme pour le premier rendu du projet (revoir si nécessaire la fin du descriptif de la semaine 7), le plus simple pour effectuer ce rendu est de faire

```
make submit2
```

(**attention** au **2** ici !) depuis votre répertoire `done/`, ou sinon d'ajouter vous-même un tag `projet2_1` à votre commit de rendu.