EPFL CS212 : ImgStore – Système de fichiers orienté images — le format imgStore

E. Bugnion & J.-C. Chappelier EPFL

Rev. 2021.02.26 / 2

Table des matières

Le but du travail de cette semaine est :

1. de prendre pleine connaissance du cadre et des concepts du projet ;
2. puis d'implémenter la fonctionnalité la plus simple, qui liste le contenu des métadonnées.

Commencez donc par lire le fichier de description principal du projet afin de bien comprendre le cadre général du projet.

Une fois ceci fait, vous pouvez continuer ci-dessous.

Matériel fourni

Dans votre dépôt GitHub de groupe, vous trouverez des fichiers suivants :

- `imgStore.h` - qui a pour rôle de définir les structures de données nécessaires pour la solution `imgStore`, et en particulier les structures de donnés en C qui définissent le format du fichier ;
- `imgStoreMgr.c` - le cœur du « *filesystem manager* », l'utilitaire en ligne de commande pour gérer des `imgStore` ; il lit une commande et appelle les fonctions correspondantes pour manipuler la base de données ;
- `tools.c` - les « fonctions outils » pour les `imgStore`, telles que par exemple afficher les structures de données utilisées ;
- `util.h` et `util.c` qui fournissent des macros et fonctions pratiques en général pour ce cours ; il n'est pas nécessaire de les utiliser (regardez si elles vous intéressent ; `b2l_16()` et `l2b_16()` sont en tout cas inutiles pour cette année) ;
- `error.h` - le « fichier d'en-tête » pour l'utilisation des codes et des messages d'erreurs ;
- `error.c` - les messages d'erreurs associés ;
- la bibliothèque locale `libmongoose` comme en semaine 2, pour le futur server Web (semaines 11 et suivantes) ;
- un `Makefile` qu'il faudra bien sûr compléter, mais qui contient déjà quelques cibles que l'on espère utiles pour vous ;
- et du matériel de test dont deux `imgStore` pour vos prochains tests : `test01.imgst_static` et `test02.imgst_static`.

Description : format des données

Les trois parties décrites dans le fichier de description principal du projet (« *header* », métadonnées et images) sont plus précisément constituées des informations suivantes :

1. `struct imgst_header` : le « *header* » qui contient les informations de configuration de la `imgStore`, à savoir :
   - `imgst_name` : un tableau de `MAX_IMGST_NAME` plus un caractères, contenant le nom de la base d'images ;
   - `imgst_version`: un `unsigned int` sur 32 bits ; version de la base de données ; elle est augmentée après chaque modification de la base ;
   - `num_files`: un `unsigned int` sur 32 bits ; nombre d'images (valides) présentes dans la base ;
   - `max_files`: un `unsigned int` sur 32 bits ; nombre maximal d'images possibles dans la base ; ce champ est spécifié lors de la création de la base, mais ne doit pas être modifié par la suite ;
   - `res_resized`: un tableau de 2 fois (`NB_RES`-1) `unsigned int` sur 16 bits ; tableaux des résolutions maximales des images « thumbnail » et « small » (dans l'ordre: « thumbnail X », « thumbnail Y », « small

X », « small Y ») ; tout comme le nombre maximum d'images, ces valeurs sont également spécifiées lors de création de la base d'image et ne doivent pas être modifiées par la suite ; à noter : les images originales ont chacune leur résolution propre, décrite plus bas ;
   - `unused_32`: un `unsigned int` sur 32 bits ; non utilisé (mais prévu pour des évolutions futures ou des informations temporaires) ;
   - `unused_64`: un `unsigned int` sur 64 bits ; non utilisé (mais prévu pour des évolutions futures ou des informations temporaires).
2. `struct img_metadata` : métadonnées d'une image :
   - `img_id` : un tableau de `MAX_IMG_ID` plus un caractères, contenant un identificateur unique (nom) de l'image ;
   - `SHA`: un tableau de `SHA256_DIGEST_LENGTH` `unsigned char` ; le « *hash code* » de l'image, comme expliqué ci-dessus ;
   - `res_orig`: un tableau de 2 `unsigned int` sur 32 bits ; la résolution de l'image d'origine ;
   - `size`: un tableau de `NB_RES` `unsigned int` sur 32 bits ; les tailles mémoire (en octets) des images aux différentes résolutions (« *thumbnail* », « *small* » et « *original* » ; dans cet ordre, donné par les indices `RES_X` définis dans `imgStore.h`) ;
   - `offset`: un tableau de `NB_RES` `unsigned int` sur 64 bits ; les positions dans le fichier « base de données d'images » des images aux différentes résolutions possibles (dans le même ordre que pour `size` ; utilisez aussi les indices `RES_X` définis dans `imgStore.h` pour accéder aux éléments de ce tableau) ;
   - `is_valid`: un `unsigned int` sur 16 bits ; indique si l'image est encore utilisée (valeur `NON_EMPTY`) ou a été effacée (valeur `EMPTY`) ;
   - `unused_16`: un `unsigned int` sur 16 bits ; non utilisé (mais prévu pour des évolutions futures ou des informations temporaires).
3. `struct imgst_file` :
   - `file`: un `FILE*` indiquant le fichier contenant tout (sur le disque) ;
   - `header`: une `struct imgst_header` ; les informations générales (« *header* ») de la base d'images ;
   - `metadata`: pour le moment (ce sera révisé plus tard) : un tableau de `MAX_MAX_FILES` `struct img_metadata` ; les « métadata » des images dans la base.

Remarques :

1. la taille du tableau des métadonnées ne change jamais ; elle est spécifiée dans le `header`. Pour le moment, elle est fixée à `MAX_MAX_FILES` ; plus tard (semaine 6), elle sera allouée dynamiquement à `max_files` ;
2. pour effacer une image, il suffira de changer `is_valid` ; il peut donc y avoir d'une part des « trous » dans le tableau des métadonnées, et d'autre part des parties inutilisées dans le fichier (puisque les images elles-mêmes ne sont pas effacées) ; l'idée de fond derrière tout ça est d'être prêt à

perdre un peu de place pour gagner du temps. A un niveau plus complexe, on peut imaginer un « *garbage collector* » (ou un « *defrag* ») qui en parallèle, quand « on a le temps », supprime effectivement les images qui ne sont plus utilisées, réorganise les métadonnées pour diminuer les trous, etc.
Nous n'entrerons pas dans ce genre de considérations dans ce projet.

3. sur les architectures 32 bits, si vous voulez tester avec les fichiers que nous fournissons, il faut ajouter deux champs à la structure `struct img_metadata` (vous pouvez les laisser dans votre code lors de la soumission ; cela ne pose pas de souci) :
   — un champ `padding0`, `unsigned int` sur 32 bits, juste après le champ `size` ;
   — un champ `padding1`, `unsigned int` sur 32 bits, juste après le chamap `unused_16`.
   (Pour vérifier, quelque soit l'architecture `sizeof(struct img_metadata)` doit donner 216).

Travail à faire

Le code fourni ne compile pas en l'état. L'objectif de la semaine consiste en les étapes suivantes, décrites en plus de détails plus bas :

1. définir (= compléter) les structures de données nécessaires aux `imgStore` ;
2. prototyper et définir une fonction `print_header()` permettant d'afficher le header d'une `imgStore` ;
3. compléter la définition de la fonction `print_metadata()` permettant d'afficher les métadonnées d'une `imgStore` ;
4. définir la fonction `do_list()` permettant d'afficher les informations d'une `imgStore` ;
5. compléter le `Makefile` pour compiler le tout (création d'un exécutable nommé `imgStoreMgr`).

A ce moment là, le code devrait compiler sans erreurs. Il vous restera alors à **tester** le résultat.

1. Définir les structures de données

Le format exact, à respecter scrupuleusement, du « *header* » et des « *metadata* » de `imgStore` est donné par la description ci-dessus. Les types

— `struct imgst_header`
— `struct img_metadata`
— `struct imgst_file`

sont à définir en remplacement de « `TODO WEEK 04: DEFINE YOUR STRUCTS HERE.` » dans le fichier `imgStore.h`.

Veuillez suivre les noms exacts fournis dans ce document. Et bien évidemment, veuillez vous assurer de respecter le format demandé.

2. Prototyper et définir la fonction `print_header()`

Replacer « `TODO WEEK 04: ADD THE PROTOTYPE OF print_header HERE` » dans le fichier `imgStore.h` afin d'y définir le prototype de la fonction `print_header()` qui a les caractéristiques suivantes:

1. elle ne retourne rien ;
2. elle prend comme seul argument une structure décrivant le `header` (et ne la modifiera pas).

Remplacer ensuite « `TODO: WRITE YOUR print_header CODE HERE` » dans `tools.c` avec votre implémentation de la fonction qui imprime le contenu du header.

Ces informations sont à afficher en respectant **strictement** le format suivant :

```
****************************************
**********IMGSTORE HEADER START**********
TYPE:            EPFL ImgStore binary
VERSION: 0
IMAGE COUNT: 0          MAX IMAGES: 10
THUMBNAIL: 64 x 64      SMALL: 256 x 256
***********IMGSTORE HEADER END***********
****************************************
```

Notes :

1. On utilisera le format `"%31s"` pour afficher `imgst_name`.
2. C99 définit (dans `inttypes.h`) des identifiants spécifiques `PRI...`pour les types `uint16_t` et similaires, à utiliser comme par exemple :

   ```
   printf("La valeur est %" PRIu16 " unités.\n", un_uint16_t);
   ```

   Notez bien la fermeture puis la réouverture des guillemets autour de `PRIu16`.
3. Les informations des deux dernières lignes sont séparées par respectivement deux puis une tabulation(s) (`\t`).

3. Compléter la définition de la fonction `print_metadata()`

Implémenter la fonction `print_metadata()` dans `tools.c`. Comme son nom le suggère, cette fonction imprime le contenu des métadonnées d'une image. Ces informations sont à afficher en respectant **strictement** le format suivant (où les informations `OFFSET` et `SIZE` sont séparées par deux tabulations (`\t`)) :

```
IMAGE ID: pic2
SHA: 1183f8ef10dcb4d87a1857bd16f9b5f8728a8d1ea6c9c7eb37ddfa1da01bff52
VALID: 1
UNUSED: 0
OFFSET ORIG. : 75100            SIZE ORIG. : 369911
OFFSET THUMB.: 0                SIZE THUMB.: 0
OFFSET SMALL : 0                SIZE SMALL : 0
ORIGINAL: 1200 x 800
****************************************
```

4. Définir la fonction `do_list()`

Pour commencer, définir le prototype de la fonction `do_list()` à l'endroit indiqué dans le fichier `imgStore.h`.

Note : comment trouver le prototype lorsqu'on ne vous l'explicite pas ?
Cherchez des appels à cette fonction...
Le but est ici de vous apprendre à entrer, à vous approprier, du code qui vous est donné.

Créer ensuite un *nouveau* fichier `imgst_list.c` pour y implémenter la fonction `do_list()`. L'objectif de `do_list` est tout d'abord d'imprimer le contenu du « *header* », et ensuite d'imprimer (exemples ci-dessous)

— soit
```
<< empty imgStore >>
```

si la base ne contient aucune image ;

— soit les métadonnées de toutes les images valides.

Pour ce faire, vous utiliserez les fonctions définies dans les étapes 2 et 3 précédentes.

Attention : il est possible d'avoir des « trous » dans le tableau des images : une/des image(s) invalide(s) peu(ven)t se trouver entre des images valides.

5. Compilation

Ecrivez un `Makefile` pour compiler et construire un exécutable nommé `imgStoreMgr`. Si vous utilisez les *warnings* du compilateur, vous pouvez ignorer les 5 « *unused variable* » (ou commenter la partie correspondante) ; ces variables seront utilisées plus tard dans le projet.

Exemples et tests

Pour faciliter la compréhension des différentes fonctions décrites ci-dessus, quelques exemples d'utilisation sont donnés ici. Ces exemples sont repris dans les tests fournis (voir plus bas).

Tests globaux (« *black-box testing* »)

Vous pouvez tester votre code avec les fichiers « `.imgst_static` » fournis : la commande

```
./imgStoreMgr list tests/data/test01.imgst_static
```

(modifier si nécessaire le chemin vers le dernier argument, nom de fichier) devrait afficher (fichier exact `test01-w04.txt` ici) :

```
****************************************
**********IMGSTORE HEADER START**********
TYPE:            EPFL ImgStore binary
VERSION: 0
IMAGE COUNT: 0          MAX IMAGES: 10
THUMBNAIL: 64 x 64      SMALL: 256 x 256
***********IMGSTORE HEADER END***********
****************************************
<< empty imgStore >>
```

et la commande

```
./imgStoreMgr list tests/data/test02.imgst_static
```

devrait afficher (fichier exact `test02-w04.txt` ici) :

```
****************************************
**********IMGSTORE HEADER START**********
TYPE:            EPFL ImgStore binary
VERSION: 2
IMAGE COUNT: 2          MAX IMAGES: 10
THUMBNAIL: 64 x 64      SMALL: 256 x 256
***********IMGSTORE HEADER END***********
****************************************
IMAGE ID: pic1
SHA: 66ac648b32a8268ed0b350b184cfa04c00c6236af3a2aa4411c01518f6061af8
VALID: 1
UNUSED: 0
OFFSET ORIG. : 2224             SIZE ORIG. : 72876
OFFSET THUMB.: 0                SIZE THUMB.: 0
OFFSET SMALL : 0                SIZE SMALL : 0
ORIGINAL: 1200 x 800
****************************************
IMAGE ID: pic2
SHA: 1183f8ef10dcb4d87a1857bd16f9b5f8728a8d1ea6c9c7eb37ddfa1da01bff52
VALID: 1
UNUSED: 0
OFFSET ORIG. : 75100            SIZE ORIG. : 369911
OFFSET THUMB.: 0                SIZE THUMB.: 0
```

```
OFFSET SMALL : 0                SIZE SMALL : 0
ORIGINAL: 1200 x 800
****************************************
```

NOTE : vous pouvez contrôler vos résultats en faisant par exemple :

```
./imgStoreMgr list tests/data/test02.imgst_static > mon_res_02.txt
diff -w test02-w04.txt mon_res_02.txt
```

avec le fichier fourni ci-dessus que vous aurez p.ex. sauvegardé dans `test02-w04.txt` (mais ne l'ajoutez (`add`) pas à `git` SVP).

Pour plus de détails : `man diff`.

[fin de note.]

Nous vous fournissons aussi dans le répertoire `tests` un Shell script qui fait automatiquement les tests ci-dessus, ainsi qu'un fichier C qui teste l'implémentation des principales structures. Ces deux tests sont ceux que nous tournons dans `make feedback`. Pour les faire tourner en local (largement recommandé **avant** de faire des `make feedback` !), faites :

```
make check
```

Et donc, comme d'habitude, nous fournissons également, *à bien plaire*, un `make feedback` (`make feedback-VM-CO` si vous travaillez sur les VM de l'Ecole) qui donne un retour *partiel* sur votre travail. Ceci est normalement à utiliser pour une vérification *minimale* **finale** de votre travail, avant de rendre. Préférez auparavant faire des tests **locaux** directement sur votre machine (et y compris plus de tests que vous aurez vous-même ajouté si nécessaire).

L'image Docker utilisé par `make feedback` sera chaque semaine marquée de l'étiquette `latest`, mais si vous souhaitez faire tourner le feedback d'une semaine spécifique, changez (dans le `Makefile` à la ligne qui définit `IMAGE`) cette étiquette `latest` par `weekNN` où `NN` est le numéro de semaine désiré, p.ex. :

```
IMAGE=chappeli/pps21-feedback:week04
```

Organisation du travail

Libre à vous de vous organiser au mieux dans votre travail suivant vos objectifs et vos contraintes ; mais pensez à vous répartir correctement la tâche entre les deux membres du groupe. A ce sujet (charge de travail), si vous ne l'avez pas encore lue entièrement, nous vous conseillons la lecture de la fin de la page expliquant le barème du cours (ici en HTML et ici en PDF).

Rendu

Vous n'avez pas à rendre de suite le travail de cette première semaine de projet, celui-ci ne sera à rendre qu'à la fin de la semaine 8 (délai : le dimanche 25 avril 23:59) en même tant que le travail des semaines 5 à 7.

Ceci dit, nous vous conseillons de marquer par un commit lorsque vous pensez avoir terminé le travail correspondant à cette semaine (vous pouvez en faire d'autres avant, bien sûr !) :

1. ajoutez le nouveau fichier `imgst_list.c` au répertoire `done/` (de votre dépôt GitHub **de groupe** ; c.-à-d. correspondant au projet), ainsi qu'éventuellement vos propres tests :

   ```
   git add imgst_list.c
   ```

2. ajoutez aussi les fichiers modifiés (mais **PAS** les `.o`, ni les exécutables, svp !) : `tools.c`, `imgStore.h` et `Makefile` :

   ```
   git add -u
   ```

3. vérifiez bien que tout est ok :

   ```
   git status
   ```

   voire :

   ```
   git status -uno
   ```

   pour cacher les fichiers non voulus, mais attention à ne justement pas cacher un fichier voulu !... ;

4. puis faites le commit :

   ```
   git commit -m "version finale week04"
   ```

Nous vous conseillons en effet fortement de travailler régulièrement et faire systématiquement ces commits réguliers, au moins hebdomadaires, lorsque votre travail est opérationnel. Cela vous aidera à sauvegarder votre travail et à mesurer votre progression.

Et n'oubliez pas de faire le rendu (individuel, depuis votre dépôt *personnel*) de la semaine passée avant ce dimanche soir.