Projet programmation système W04

Introduction

Nous abordons cette semaine la seconde (et principale) partie de ce cours-projet : la construction d'un utilitaire en ligne de commande pour gérer des images dans une base de données de format spécifique, inspiré de celui utilisé par Facebook. Pour votre information, le système de Facebook s'appelle « Haystack » et est décrit dans le papier suivant : https://www.usenix.org/event/osdi10/tech/full_papers/Beaver.pdf). Vous ne devez *pas* lire ce papier dans le cadre du cours (c'est juste pour information) car, bien évidemment, nous allons implémenter une version simplifiée de ce système.

Les réseaux sociaux doivent gérer des centaines de millions d'images. Les systèmes de fichiers usuels (tel que celui utilisé sur votre disque dur par exemple) ont des problèmes d'efficacité avec de tels nombres de fichiers. De plus, ils ne sont pas conçus pour gérer le fait que l'on veut avoir chacune de ces images en plusieurs résolutions, par exemple toute petite (icône), moyenne pour un « *preview* » rapide et en taille normale (résolution d'origine).

Dans l'approche « Haystack », plusieurs images se trouvent dans un même fichier. De plus, différentes résolutions de la même image sont stockées automatiquement. Ce fichier unique contient à la fois les données (les images) et les métadonnées (les informations concernant chaque image). L'idée clé est que le serveur d'images a une copie de ces métadonnées en mémoire, afin de permettre un accès très rapide à une photo spécifique, et ce dans la bonne résolution.

Cette approche a un certain nombre d'avantages : tout d'abord, elle réduit le nombre de fichiers gérés par le système d'exploitation ; d'autre part, elle permet d'implémenter de manière élégante deux aspects importants de la gestion d'une base d'images :

1. la gestion automatique des différentes résolutions d'image, dans notre cas les trois résolutions supportées ;
2. la possibilité de ne pas dupliquer des images identiques soumises sous des noms différents (p.ex. par des utilisateurs différents chez Facebook) ; c'est une optimisation extrêmement utile dans tout réseau social.

Cette « déduplification » (« *deduplication* » en anglais) se fait à l'aide d'une « fonction hash » qui résume un contenu binaire (dans notre cas une image) en une signature beaucoup plus courte. Nous utiliserons ici la fonction « SHA-256 » qui résume tout contenu binaire en 256 bits, avec comme propriété cryptographique intéressante que la fonction est résistante aux collisions : pour une image donnée, il est pratiquement impossible de créer une autre image qui aurait la même signature. Dans le cadre de notre projet, nous considérons que deux images ayant la même signature SHA-256 sont nécessairement identiques. Même si cela

peut paraître surprenant, de nombreux systèmes informatique de production sont basés sur ce principe.

Durant le reste du semestre, vous allez construire un serveur d'images ad-hoc, dans une version inspirée et simplifiée de Haystack. Durant les premières semaines, il s'agira d'abord d'implémenter les fonctions de base du système, à savoir :

- lister les informations (métadonnées, liste des images) ;
- rajouter une nouvelle image ;
- effacer une image ;
- extraire une image donnée, dans une résolution spécifique au choix : « *original* », « *small* » ou « *thumbnail* ».

Dans cette première phase, les fonctions seront exposées via un utilitaire de ligne de commande. Durant les dernières semaines du semestre, vous construirez un véritable serveur web qui exposera la même fonctionnalité avec la plus grande performance possible via le protocole HTTP.

Format des données

Vous allez donc utiliser un format spécifique – appelons le « `pictDB`» (*Picture Database*) – pour représenter un « système de fichiers d'images » (ou « base de donnée d'images »). Un fichier de type `pictDB` (représentant donc un tel système) comprend trois parties distinctes :

- un `header`, de taille fixe, qui rassemble les éléments de configurations du système ; le contenu du `header` est créé lors de la création du`pictDB` ;
- le tableau des métadonnées ; il s'agit d'un tableau dont la taille est spécifiée par le champ `max_files` du `header` ; chaque entrée du tableau décrit toutes les métadonnées d'une seule image, et en particulier la position de l'image (sous ses différentes résolutions) dans le fichier ;
- les images elles-mêmes ; chaque image est stockée sur une partie contiguë du fichier, les unes après les autres.

Ces trois parties sont plus précisément constituées des information suivantes :

1. `struct pictdb_header` : le « *header* » qui contient les informations de configuration de la `pictDB`, à savoir :
    - `db_name` : un tableau de `MAX_DB_NAME` plus un caractères, contenant le nom de la base d'images ;

- `db_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 pict_metadata` : métadonnées d'une image :

   - `pict_id` : un tableau de `MAX_PIC_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 `pictDB.h`) ;
   - `offset`: un tableau de `NB_RES unsigned int` sur 64 bits ; les positions dans le fichier « base de donnée d'images » des images aux différentes résolutions possibles (dans le même ordre que pour `size` ; utilisez aussi les indices `RES_X` définis dans `pictDB.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 pictdb_file` :

- `fpdb`: un `FILE*` indiquant le fichier contenant tout (sur le disque) ;
- `header`: une `struct pictdb_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 pict_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 pict_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, `sizeof(struct pict_metadata)` doit donner 216).

Matériel fourni

L'objectif de cette semaine est d'implémenter la fonctionnalité la plus simple, qui liste le contenu des métadonnées.

Dans votre dépôt GitHub, vous trouverez des fichiers suivants :

- `pictDB.h` - qui a pour rôle de définir les structures de données nécessaires pour la solution `pictDB`, et en particulier les structures de donnés en C qui définissent le format du fichier ;

- `pictDBM.c` - le coeur du « *database manager* », l'utilitaire en ligne de commande pour gérer des `pictDB` ; il lit une commande et appelle les fonctions correspondantes pour manipuler la base de données ;
- `db_utils.c` - les « fonctions outils » pour les `pictDB`, telles que par exemple afficher les structures de données utilisées ;
- `error.c` - les messages d'erreurs ;
- `error.h` - le « fichier d'en-tête » pour l'utilisation des messages d'erreurs ;
- deux `pictDB` pour vos tests : `testDB01.pictdb_static` et `testDB02.pictdb_static`.

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 `pictDB` ;
2. prototyper et définir une fonction `print_header` permettant d'afficher le header d'une `pictDB` ;
3. compléter la définition de la fonction `print_metadata` permettant d'afficher les métadonnées d'une `pictDB` ;
4. définir la fonction `do_list` permettant d'afficher les informations d'une `pictDB` ;
5. créer un `Makefile` pour compiler le tout (création d'un exécutable nommé `pictDBM`).

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 `pictDB` est donné par la description ci-dessus. Les types

- `struct pictdb_header`
- `struct pict_metadata`
- `struct pictdb_file`

sont à définir en remplacement de « `TODO WEEK 04: DEFINE YOUR STRUCTS HERE.` » dans le fichier `pictDB.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 `pictDB.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 `db_utils.c` avec votre implémentation de la fonction qui imprime le contenu du header.

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

```
****************************************
**********DATABASE HEADER START**********
DB NAME:              EPFL PictDB binary
VERSION: 0
IMAGE COUNT: 0          MAX IMAGES: 10
THUMBNAIL: 64 x 64      SMALL: 256 x 256
***********DATABASE HEADER END***********
****************************************
```

Notes :

1. On utilisera le format `"%31s"` pour afficher `db_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 `db_utils.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`)) :

```
PICTURE 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
****************************************
```

4. Définir la fonction `do_list`

Pour commencer, définir le prototype de la fonction `do_list` à l'endroit indiqué dans le fichier `pictDB.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 `db_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 database >>
  ```

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 `pictDBM`.

Tests

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

```
./pictDBM list testDB01.pictdb_static
```

(modifier si nécessaire le chemin vers le dernier argument, nom de fichier) devrait afficher (fichier exact `testDB01-w04.txt` ici) :

```
****************************************
**********DATABASE HEADER START**********
DB NAME:              EPFL PictDB binary
VERSION: 0
IMAGE COUNT: 0          MAX IMAGES: 10
THUMBNAIL: 64 x 64      SMALL: 256 x 256
***********DATABASE HEADER END***********
****************************************
<< empty database >>
```

et la commande

```
./pictDBM list testDB02.pictdb_static
```

devrait afficher (fichier exact `testDB02-w04.txt` ici) :

```
****************************************
**********DATABASE HEADER START**********
DB NAME:              EPFL PictDB binary
VERSION: 2
IMAGE COUNT: 2          MAX IMAGES: 10
THUMBNAIL: 64 x 64      SMALL: 256 x 256
***********DATABASE HEADER END***********
****************************************
PICTURE 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
****************************************
PICTURE 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 :

```
./pictDBM list testDB02.pictdb_static > mon_res_02.txt
diff -w testDB02-w04.txt mon_res_02.txt
```

avec le fichier fourni ci-dessus. Plus de détails : `man diff`.

Rendu

Pour rendre la partie correspondant à cette semaine, ajoutez les nouveaux fichiers `db_list.c` et `Makefile` ainsi que vos révisions des fichiers `db_utils.c` et `pictDB.h` dans un **répertoire `done/pictDBM`** de votre GitHub, puis « pousser » le résultat vers GitHub (`commit` plus `push`).

Le délai pour faire le `push` de rendu est : dimanche 03 avril 23:59.

Projet programmation système W05

Introduction

L'objectif de la semaine est d'implémenter deux fonctions pour le gestionnaire d'images :

- la fonction `help` — élément standard et indispensable de toute utilitaire de commande ;
- la fonction `create` — pour créer un nouveau fichier (vide) au format `pictDB` (= une nouvelle base d'images).

Le but de cet exercice est d'apprendre à écrire des structures de données sur disque en utilisant les opérations de base d'entrées/sorties.

Comme les semaines précédentes, il s'agira d'écrire votre code en modifiant des éléments fournis.

Afin de simplifier l'exercice, un certain nombres de paramètres, notamment le nombre maximal d'images ainsi que les résolutions « *thumbnail* » et « *small* », sont ici déterminées par des constantes à l'intérieur du code fourni (cf la fonction `do_create_cmd` dans `pictDBM.c`), plutôt que des arguments passés sur la ligne de commande ; ce qui sera le cas plus tard dans le projet.

Matériel fourni

Dans votre dépôt GitHub (`provided/week05`), vous trouverez le nouveau fichier suivant :

- `db_create.c` : implémentation de la commande `create`.

Pour le reste, il vous faudra continuer à modifier sur les fichiers déjà utilisés la semaine passée : `pictDBM.c`, `pictDB.h` et bien sûr le `Makefile`.

Travail à faire

Pour l'exercice de cette semaine, vous utiliserez simplement la structure de type `pictdb_file` de la semaine précédente, définie pour `MAX_MAX_FILES=10` images au maximum. En conséquence, votre implémentation ne créera (pour le moment) que des base d'images de cette taille.

Votre travail de la semaine consiste en cinq modifications, résumées ici et détaillées plus bas si nécessaire :

1. prototyper la fonction `do_create` dans le fichier `pictDB.h` ; cette fonction doit prendre deux arguments : le nom du fichier et une structure de type `pictdb_file` ; elle retourne un code d'erreur (`int`) ou `0` s'il n'y a pas d'erreur ;
2. dans le fichier `db_create.c`, implémenter la fonction `do_create` ayant pour but de créer une nouvelle base d'images dans un fichier (binaire) sur disque ;
3. compléter la fonction `do_create_cmd` dans le fichier `pictDBM.c` afin d'appeler correctement `do_create` ;
4. définir la fonction `help` qui imprimera les instructions pour utiliser l'utilitaire `pictDBM` ;
5. mettre à jour le `Makefile` pour la compilation et création de l'utilitiaire `pictDBM`.

Définir `do_create`

Cette fonction doit terminer d'initialiser la structure `pictdb_file` reçue avant de l'écrire sur le disque, le « *header* » en premier, puis les « *metadatas* ». Elle doit utiliser les fonctions standard C d'entrée/sortie pour créer la nouvelle base d'images dans un fichier *binaire* sur disque.

Il est important de bien initialiser explicitement tous éléments *pertinents* avant l'écriture. Et il est évidemment essentiel d'écrire le tableau de `metadata` de la bonne taille dans le fichier.

Il est également important de bien traiter tous les cas d'erreurs possibles. En absence d'erreur, `do_create` doit retourner 0 ; en cas d'erreur, retourner le code de valeur correspondant tel que défini dans `error.h`.

Cette commande n'étant utilisée qu'une fois (pour créer une base) et toujours depuis l'utilitaire ligne de commande `pictDBM` (jamais depuis un serveur, par exemple), nous allons **exceptionnellement** y ajouter un effet de bord sous la forme d'un affichage indiquant le (*vrai*) nombre d'objets sauvés sur le disque. Dans le cas présent (un « *header* » puis dix « *metadatas* »), on aura alors l'affichage suivant :

```
11 item(s) written
```

`11` parce que le « *header* » puis chacun des dix « *metadatas* » ont été écrits avec succès par `fwrite`.

Compléter `do_create_cmd`

Nous vous fournissons une implémentation incomplète de `do_create_cmd`. Dans le cadre de votre solution, vous devez créer un `pictdb_file`, initialiser les champs `max_files` et `res_resized` de son « *header* » avec les valeurs fournies, puis appeler ensuite `do_create` (qui initialisera les autres champs).

`do_create_cmd` doit retourner le code d'erreur retourné par `do_create`.

Définir `help`

La commande `help` est destinée à être utilisée dans deux cas différents (déjà traités) :

1. lorsque les arguments passés à l'utilitaire ne sont pas valables ;
2. lorsque l'utilisateur demande explicitement la liste des possibilités en tapant `pictDBM help`.

L'output de la commande doit avoir *exactement* le format suivant :

```
pictDBM [COMMAND] [ARGUMENTS]
  help: displays this help.
  list <dbfilename>: list pictDB content.
  create <dbfilename>: create a new pictDB.
```

Pour résoudre cette partie (facile !), écrire la fonction à l'endroit prévu dans `pictDBM.c`.

Test

Testez vos deux nouvelles commandes (utilisez `help` pour savoir comment utiliser `create` ;-P).

Pour vérifier que le fichier binaire a été correctement écrit sur disque, utilisez la fonction `list` de la semaine passée.

Rendu

Pour rendre la partie correspondant à cette semaine, ajoutez les nouvelles versions des fichiers `pictDBM.c`, `pictDB.h` et `Makefile`, ainsi que le nouveau fichier `db_create.c` dans le répertoire `done/pictDBM` de votre GitHub, puis « pousser » le résultat vers GitHub (`commit` plus `push`).

Le délai pour faire le `push` de rendu est : dimanche 10 avril 23:59.

Projet programmation système W06

Introduction

Cette semaine, nous allons ajouter une nouvelle fonctionnalité à notre outil : la suppression d'images ; mais aussi mettre en oeuvre vos tous nouveaux savoirs sur les pointeurs pour écrire du code plus « pro ». Nous allons pour cela devoir faire ce qui se passe assez souvent dans de vrais projets : réviser le code déjà écrit (« *code refactoring* »). Nous allons en particulier pouvoir modulariser un peu plus notre interpréteur de commande (`pictDBM`) en ajoutant deux nouvelles fonctions : `do_open` and `do_close`.

Matériel fourni

Nous ne vous fournissions rien de nouveau cette semaine. Vous allez cette fois devoir créer par vous-même le fichier supplémentaire nécessaire. Bien évidemment, vous allez également ré-utiliser et réviser les fichiers de la semaine précédente.

Travail à faire

Votre travail de la semaine consiste en quatre modifications :

1. remplacer chaque utilisation de `struct` comme argument de fonction par un pointeur ou un « const pointeur » suivant les cas ; comme indiqué en cours, cela permet de faire des passages par référence lorsque c'est nécessaire, ou d'éviter des copies lors du passage par valeur ;
2. définir les fonctions `do_open` et `do_close` : `do_open` doit ouvrir le fichier de base de données d'images, lire son « *header* » ainsi que le tableau des « métadonnées » ; la fonction `do_close` ferme le fichier de base de donnée d'images ouvert ;
3. prototyper `do_delete` dans le fichier `pictDB.h` et l'implémenter dans un nouveau fichier `db_delete.c` ; la fonction `do_delete` doit « effacer » une image spécifiée (on va voir ci-dessous ce que cela signifie vraiment) ;
4. réviser le coeur de l'interpréteur de commande, `pictDBM.c`, pour lui apporter les modifications précédentes.

1. Passage par référence/pointeurs constants

La signature de la fonction `do_list` de la semaine 4 était :

`do_list (const struct pictdb_file db_file)`

Comme vous l'avez vu en classe, le contenu entier de la structure `db_file` est alors copiée lors de l'appel (passage par valeur). Lorsque la structure est grande, cela a un coût non négligeable. Plus grave, la pile (« *stack* ») est souvent d'une taille très limitée dans beaucoup d'environnements et l'utilisation abusive de la pile pour y copier des objets crée le risque d'un dépassement de pile (en anglais « *stack overflow* »).

Pour éviter cela, il suffit de passer comme argument l'adresse de la structure au lieu de la structure elle-même.

Modifiez toutes les fonctions qui ont des arguments de type « structure » pour éviter les passages par valeurs. Faites attention à définir des passages par référence lorsque cela est nécessaire et des passage par « const pointeur » sinon.

Pour rappel, la notation C « `X->Y` » est une version plus courte, et surtout plus fréquente, pour « `(*X).Y` ». Personne n'utilise cette seconde syntaxe, préférez donc la première.

2. `do_open` et `do_close`

L'objectif est ici de modulariser le code : séparer/expliciter les fonctions d'ouverture et de fermeture de la base d'images. Cela simplifiera l'organisation du code par la suite.

Vous devez écrire les prototypes (dans le fichier `pictDB.h`) et les définitions (dans le fichier `db_utils.c`) de `do_open` et `do_close`.

La fonction `do_open` prend comme arguments :

- le nom de fichier de la base d'image (`const char *`) ;
- le mode d'ouverture du fichier (`const char *`, par exemple `"rb"`, `"wb+"`) ;
- la structure `picdb_file` dans lesquelles stocker les données lues (n'oubliez pas la section 1. précédente ; nous ne le rappellerons plus).

La fonction doit (1) ouvrir le fichier, (2) lire le contenu du « *header* » et (3) lire le contenu des « métadonnées ». La fonction doit retourner la valeur zéro si tout s'est correctement passé, et sinon un code d'erreur approprié en cas de problèmes. Comme dans les semaines précédentes, vous devez traiter tous les cas d'erreurs possibles dans cette fonction et utiliser les définitions de `error.h`.

La fonction `do_close` prend un seul argument de type structure `pictdb_file` et doit fonction doit fermer le fichier (qu'elle contient). Elle ne retourne pas de valeur. La aussi, pensez à traiter le cas d'erreur possible : si le fichier (`FILE`) est `NULL`. Cela doit être un réflexe qui va de soit lorsque vous écrivez du code, et notamment lorsque vous utilisez un pointeur. Nous ne le rappelerons plus dans la suite.

3. Prototyper et définir `do_delete`

Le moment est venu d'implémenter la fonctionnalité qui permet d'effacer une image. L'idée est la suivante : on n'efface pas réellement le contenu de l'image car ce serait trop couteux (surtout en temps). En fait, la taille sur disque du fichier base d'images ne diminue jamais, même lorsqu'on demande d'« effacer » une image de la base.
Concrètement, on « efface » une image en

1. trouvant la réference de l'image avec le même nom dans les « métadonnées » ;
2. en invalidant la référence en écrivant la valeur zero dans `is_valid` ;
3. ajustant les informations du « *header* ».

Les changements doivent être faits d'abord sur les « métadonnées » (mémoire puis disque), puis sur le « *header* » en cas de succès.
Note : pour des raisons de compatibilité entre systèmes (« ___offsets_ »), il est préférable d'écrire toute la « `struct` » sur disque plutôt que les champs modifiés.

En utilisant cette approche, il est facile de rajouter une nouvelle image : il suffit simplement de trouver la première entrée dans les « métadonnées » qui n'est pas valide et l'utiliser. A ce moment, la position dans la « métadonnées » sera réutilisée pour la nouvelle image, mais les photos elles-mêmes seront rajoutées en fin du fichier base d'images.

La fonction `do_delete` prend les arguments suivants :

- un identifiant (chaîne de caractères, `const char *`) ;
- une structure `pictdb_file`.

Pour écrire les changements sur disque, il faut d'abord se positionner au bon endroit dans le fichier en utilisant `fseek` (voir le cours et `man fseek`) et `fwrite`.

Il faut évidemment traiter correctement le cas où la référence dans la base d'image n'existe pas (et qu'il n'y a pas d'invalidation).

N'oubliez pas de mettre à jour le « *header* » si l'operation est un succès. Il faut également augmenter de 1 le numéro de version (`db_version`), ajuster le nombre d'images valides stockées (`num_files`) et également écrire le « *header* » sur disque.

Comme toujours, il faut s'occuper des cas d'erreurs qui pourraient se produire dans votre fonction. `do_delete` retourne zero en cas de success et un code défini dans `error.h` en cas d'erreur (l'erreur `FILE_NOT_FOUND` est à utiliser lorsqu'aucune image valide correspondant à l'identifiant donné n'a pu être trouvée).

4. Adapter `pictDBM.c`

La première chose à faire est de penser à ajouter la description de la commande « *delete* » dans l'aide :

```
delete <dbfilename> <pictID>: delete picture pictID from pictDB.
```

Ensuite modifiez aux endroits pertinents l'ouverture, la lecture et la fermeture du fichier : remplacez-les par des appels à `do_open` et `do_close`.

Modifiez également tous les appels qui doivent l'être (pointeurs ; section 1. ci-dessus).

Dans la commande `do_create_cmd`, maintenant que `myfile` est passé par référence à `do_create`, nous allons pouvoir ajouter un affichage informatif lors de la création d'une base d'images : ajoutez un appel à `print_header` après l'appel à `do_create`, si ce dernier a réussi.

Complétez enfin le code de `do_delete_cmd`. Si la `pictID` reçue est vide ou de longueur (`strlen(pictID)`) supérieure à `MAX_PIC_ID`, `do_delete_cmd` doit retourner l'erreur `ERR_INVALID_PICID` (définie dans `error.h`).

N'oubliez pas, enfin, d'adapter votre `Makefile`.

STYLE !

Note importante : écrire du code propre, lisible par tous, est aussi important. Il semble que ce ne soit pas le cas de tout le monde jusqu'ici `;-)`. Nous vous imposons donc à partir de maintenant de strictement suivre le style définit par `astyle -A8`.
`astyle` est un programme qui a justement pour but de reformater les codes sources pour suivre un standard (`man astyle` pour plus de détails).
A partir de maintenant, tout code ne respectant le standard `astyle -A8` se vera affecter des points de pénalité.

Tests

Utiliser le fichier `testDB02.pictdb_static` des semaines précédentes (**faites en une copie !!**) pour voir son contenu, supprimer une, deux image(s). Vérifier à chaque fois en regardant le résultat avec `list`.

Faites également tous les tests de cas problèmatiques auxquels vous pouvez penser.

Rendu

Pour rendre la partie correspondant à cette semaine, ajoutez les nouvelles versions des fichiers `db_utils.c`, `pictDB.h`, `pictDBM.c`, `db_list.c`, `db_create.c` et `Makefile`, ainsi que le nouveau fichier `db_delete.c` dans le répertoire `done/pictDBM` de votre GitHub, puis « pousser » le résultat vers GitHub (`commit` plus `push`).

Le délai pour faire le `push` de rendu est : dimanche 17 avril 23:59.

Projet programmation système W07

Introduction

L'objectif de cette semaine est double (pensez à vous répartir le travail) :

1. incorporer les nouveaux concepts du langage C vus pendant le cours - notamment l'allocation dynamique de mémoire ;
2. préparer la mise en place des fonctions de manipulation d'images (`read` et `insert`) qui seront finalisées dans deux semaines.

Matériel fourni

Il n'y a pas de nouveau fichier C cette semaine. Comme la semaine passée, vous construisez votre projet en vous basant sur les versions des semaines précédentes et créerez par vous-même les nouveaux fichiers nécessaires.

La seule chose que nous vous fournissons cette semaine est `testDB02.pictdb_dynamic`, une base d'images existante au nouveau format développé cette semaine (allocation dynamique). Vous pourrez l'utilisez pour tester votre programme.

Travail à faire

L'objectif de la semaine consiste en les étapes suivantes, décrites plus en détail plus bas :

1. l'allocation dynamique de la partie « métadonnées » ;
2. la création d'images de tailles réduites (formats « *small* » et « *thumbnail* ») ;
3. adapter votre `Makefile`.

1. Allocation dynamique

Actuellement, le champ `metadata` de la structure de données `pictdb_file` consiste en un tableau de taille fixe, alloué statiquement donc.

Cette définition a l'avantage de la simplicité de programmation, puisque que toute la partie `metadata` *a priori* utilisable est déjà stockée dans la même structure (et contiguë en mémoire). Malheureusement, cette simplicité a un coût énorme des points de vues de l'occupation mémoire et de la flexibilité : le nombre maximum d'images stockées dans un fichier de base de données d'images occupe déjà toute la place et est par ailleurs lié à une constante de compilation (`MAX_MAX_FILES`). Idéalement, le nombre maximum d'images devrait être laissé à la discrétion de l'utilisateur du programme, et non à celle de celui/celle qui l'a écrit.

Cette approche a un troisième défaut — rédhibitoire celui-là : le programme ne fonctionne correctement que si la valeur `MAX_MAX_FILES` ne change pas entre la création du fichier et son utilisation. Tout changement de cette valeur, suivi d'une recompilation de l'outil de commandes, rendrait impossible le décodage des images des bases précédentes !

L'objectif est donc d'offrir flexibilité à l'utilisateur et robustesse à l'usage.

Pour ce faire, nous allons mettre en place l'utilisation **dynamique** de la partie `metadata`.

Les changements doivent avoir lieu a plusieurs endroits :

- tout d'abord, le champ `metadata` de la structure `pictdb_file` doit devenir un pointeur ;
- ce pointeur doit ensuite être initialisé en allouant la mémoire de manière dynamique à chaque endroit qui initialise une structure de type `pictdb_file`, à savoir dans `do_create()` et dans `do_open()` ;
- mettez enfin à jour toute partie du code devant l'être suite aux changements précédents.

Pour allouer la mémoire, utilisez la fonction `calloc`. Si nécessaire, revoyez le cours ou regardez la « *man page* » pour plus d'explications et n'oubliez pas que la fonction peut retourner une erreur. En cas d'erreur, votre fonction doit retourner `ERR_OUT_OF_MEMORY` (voir `error.h` et `error.c`).

Une fois ces changements faits, la constante `MAX_MAX_FILES` change de rôle : elle ne définit plus le nombre maximum d'images de la table, et ne détermine plus non plus la quantité de mémoire requise pour pouvoir exécuter le programme. Il s'agit maintenant juste d'une valeur maximale, spécifiée en fonction des besoins extrêmes.
Changez sa valeur à 100000. Est-ce que votre programme tourne toujours ? (il devrait...)

2. Création et gestion d'images dérivées

Une des fonctions principale de `pictDB` est de gérer de manière transparente et efficace les différentes résolutions d'une même image (pour rappel : dans ce projet, nous aurons la résolution d'origine et les résolutions « *small* » et « *thumbnail* »).

Dans une première étape cette semaine, vous devez implémenter une fonction `lazily_resize`. Son nom fonction suggère son emploi : en informatique, « *lazy* » correspond à une stratégie couramment utilisée qui consiste à différer le travail jusqu'au dernier moment, dans l'idée d'éliminer du travail inutile.
(**Note de la part des enseignants :** ne pas confondre « informatique » et « études en informatique » `;-)`).

Cette fonction a trois arguments :

- un entier correspondant à un code interne d'une des résolutions dérivées de l'image : `RES_THUMB` ou `RES_SMALL` (voir `pictDB.h`) ;
  (note : si c'est `RES_ORIG` qui est passé, la fonction ne fait simplement rien et ne retourne pas d'erreur (`0`)) ;
- une structure `pictdb_file` (celle avec laquelle on travaille) ;
- et un index, de type `size_t`, position/index de l'image à traiter.

Elle doit implémenter la logique suivante :

- vérifier la légitimité des arguments, et retourner si nécessaire une valeur d'erreur appropriée (voir `error.h` et `error.c`) ;
- si l'image demandée existe déjà dans la résolution correspondante, ne rien faire ;
- dans les autres cas, il faut d'abord créer une nouvelle variante de l'image spécifiée, dans la résolution spécifiée ;
- ensuite, copier le contenu de cette nouvelle image à la fin du fichier `pictDB` ;
- finalement, mettre à jour le contenu de la `metadata` en mémoire ainsi que sur disque.

Pour ce faire, vous utiliserez la bibliothèque `VIPS` introduite en semaine 2.
Nous vous conseillons par ailleurs d'avoir une approche modulaire (ceci devrait toujours être le cas !!)

Votre solution devra consister en :

- un nouveau fichier `image_content.c` qui implémente la fonction `lazily_resize()` ;
- un nouveau fichier `image_content.h` qui prototype `lazily_resize()` ;
- les changements nécessaires de votre `Makefile` (voir ci-dessous).

3. Bibliothèque VIPS et modification du Makefile

Un des objectifs explicite de cette semaine est d'apprendre à incorporer dans votre solution des bibliothèques complexes (parfois très complexes). En l'occurrence ici, afin de comprimer une image en utilisant la librairie VIPS

Pour vous aider, veuillez regarder comme exemple l'utilisation de VIPS en semaine 2 (`thumbify.c`), ainsi que toute documentation sur Internet concernant cette bibliothèque.

Les images n'étant pas ici stockées de façon individuelle dans des fichiers séparés, vous ne pouvez pas utiliser `vips_image_new_from_file`. Il faudra utiliser `fread`, puis `vips_object_local_array` et `vips_jpegload_buffer` pour charger l'image originale en mémoire.

De même, pour écrire l'image en nouvelle résolution sur le disque (en fin de fichier pictDB), vous ne pouvez pas utiliser `vips_image_write_to_file`, mais utilisez plutôt `vips_jpegsave_buffer` et `fwrite`.

Nous insistons sur le fait que c'est une partie, *non négligeable*, de **votre** travail de cette semaine que de comprendre et adapter le code fourni en semaine 2.

Utilisez également le `Makefile.vips` fourni cette semaine là (ainsi que le tutoriel) pour adapter votre `Makefile`.

Tests

L'allocation de mémoire dynamique peut se tester en créant des fichiers de différentes tailles (en utilisant la fonction `create` de la semaine 5) ainsi que le fichier `testDB02.pictdb_dynamic` fourni cette semaine.

Pour le moment, pour tester la fonction `lazily_resize`, il vous faut modifier de manière temporaire votre programme principal (ou travailler sur une copie) afin de tester sa fonctionnalité (qui sera intégrée plus tard).
Vous pouvez par exemple modifier la fonction `do_list_cmd` pour y insérer les appels à `lazily_resize` que vous voudrez tester, entre autant d'appels à `do_list` que vous jugerez nécessaires. Assurez-vous également que le fichier a crû du nombre attendu d'octets (vrai accroissement si les images n'existaient pas encore dans les résolutions demandées, et aucune modification si les résolutions avaient déjà été demandées une première fois).

Pensez également à ajouter `VIPS_INIT` et `vips_shutdown` au `main()` (voir `thumbify.c` de la semaine 2).

Outils

Avec l'arrivée des pointeurs, et surtout cette semaine de l'allocation dynamique, il peut être utile de connaître d'autres outils que le débogueur pour trouver des erreurs, en particulier de mémoire. Nous vous recommandons cependant de continuer à utiliser un débogueur : les nouveaux outils présentés ici sont *complémentaires* de celui-ci.

valgrind

Valgrind (et son GUI compagne Valkyrie) est un outil d'analyse dynamique de code qui permet en particulier de détecter des erreurs d'accès mémoire.

Son utilisation est très simple : sur la ligne de commande, faites précéder votre commande d'exécution du mot `valgrind` et lisez tout ce qu'il vous raconte. Par exemple :

```
valgrind ./pictDBM
```

Pour plus de détail, nous vous renvoyons au site de Valgrind.

scan-build

Valgrind fait de l'analyse dynamique, mais il peut aussi être utile de faire de l'analyse *statique* de code (i.e. au moment de la compilation au lieu de l'exécution). En particulier, le compilateur `clang` vient avec un analyseur statique assez puissant : `scan-build`.

Son utilisation est très simple : sur la ligne de commande, faites précéder votre commande de compilation du mot `scan-build` et lisez tout ce qu'il vous raconte. Par exemple :

```
scan-build make
```

Pour plus de détail, nous vous renvoyons :

- à cette page pour l'utilisation ;
- et à celle-ci pour l'installation si nécessaire (mais est souvent déjà installé avec `clang`).

Rendu

Vu que nous n'avons qu'un travail partiel (et qu'il y a la série notée du cours-compagnon qui se profile...), il y a cette semaine une pause dans les rendus : rien à rendre cette fois, mais comme sous-entendu plus haut : n'en profitez pas pour prendre du retard ! Cette semaine et la prochaine sont à notre avis assez chargées.

Projet programmation système W08

Introduction

Cette semaine consiste en deux objectifs distincts (pensez à vous répartir le travail) :

- traiter de façon plus portable les arguments de ligne de commande en utilisant la fonctionnalité des pointeurs sur fonctions ; nous en profiterons pour ajouter des options à la commande `create` ;
- incorporer les éléments qui permettront la dé-duplication des images sauvegardées (ne pas dupliquer des images identiques).

Note/Rappel : si vous travaillez en avance sur le rendu (et êtes assez à l'aise avec git) allez voir ce fil de discussion pour ne pas « polluer » vos rendus avec du travail en avance : http://moodle.epfl.ch/mod/forum/discuss.php?d=5403

Matériel fourni

Comme précédemment, vous construisez votre projet sur la base des semaines précédentes. En plus, nous vous fournissons deux fichiers, `pictDBM_tools.c` et `pictDBM_tools.h`, qui fournissent deux fonctions outils (`atouint16` et `atouint32`) permettant de convertir une chaîne de caractères contenant un nombre en sa valeur en `uint16` ou `uint32`.

Nous vous encourageons à utiliser ces fonctions `atouint16` et `atouint32` pour convertir les chaînes de caractères des arguments de ligne de commande. Ces deux fonctions gèrent les différents cas d'erreurs en cas de conversion d'un nombre non valide, ou d'un nombre trop grand pour le type spécifié (par exemple en essayant de convertir 1000000 en un nombre de 16 bits). Elles retournent 0 dans ces cas. Utilisez les pour implémenter correctement votre code.

Travail à faire

Analyse (« *parsing* ») des arguments de ligne de commande

Unification de la signature des fonctions de commande

Regardez la fonction `main` dans la version actuelle de `pictDBM.c` : elle contient à la fois le *parsing* des commandes (par exemple, « `list` », « `create` », « `delete` », etc.) ainsi que le *parsing* des arguments de ces fonctions (par exemple, `do_delete_cmd` prend 2 arguments, mais `do_create_cmd` n'en prend qu'un seul). Cette double logique rend le code difficile à maintenir et à faire évoluer (ajout de nouvelles commandes). Par exemple, si on décide de changer les arguments

d'une commande particulière, il faut à la fois changer l'implémentation de de cette commande, ainsi que changer l'appel de cette fonction dans `main`.

L'objectif de cette semaine est d'unifier la gestion des commandes reçues en utilisant les pointeurs de fonctions. On pourrait même imaginer (extension hors du projet, non demandée) transformer le programme en un interpréteur de commandes (tournant sans fin jusqu'à la commande `quit`).

Tout d'abord, vous devez changez la signature de toutes les fonctions `do_COMMAND_cmd` (et `help`).
Actuellement, elles ont chacune une signature propre. Pour simplifier la logique, nous devons uniformiser leur signature. La solution est toute simple : plutôt que de *parser* les arguments dans `main`, il suffit de déléguer le *parsing* à ces fonctions et de passer tous les arguments de ligne de commande à ces fonctions :

```
int do_COMMAND_cmd(int args, char *argv[])
```

Donnez aussi ce même prototype à la fonction `help`.

Maintenant, `main` peut simplement appeler chaque commande de manière identique.

Utilisation de tables pour simplifier la logique

La version originale de `main` utilise une approche « if-then-else » pour traiter des différentes commandes possibles. Cette approche rend plus difficile le rajout de nouvelles commandes (qui arriveront bientôt dans les semaines suivantes) puisqu'il faut à chaque fois rajouter un nouveau cas. Elle rend aussi le code rapidement illisible en cas de nombreuses commandes.

Cette deuxième modification a pour but d'éliminer ces tests répétitifs en les remplaçant par une simple boucle ! L'objectif de cette boucle est tout simplement de rechercher la commande parmi une liste de commandes possibles (pour le moment « *list* », « *create* », « *help* », et « *delete* ») et d'appeler la fonction correspondant à chaque commande.

Nous allons donc mettre les différentes fonctions `do_COMMAND_cmd` précédemment unifiées (et `help`) dans un tableau. On en profitera pour associer les noms des commandes avec leur fonctions respectives (p.ex. la chaîne `"list"` avec la fonction `do_list_cmd`).

Pour cela, dans `pictDBM.c` : 1. définissez un type `command`, pointeur sur fonction telles que celles unifiées ci-dessus ; 2. définissez un type `struct command_mapping` contenant une chaîne de caractères (constante) et une `command`.

Utiliser ensuite ces définitions pour créer un tableau nommé `commands` associant les commandes « *list* », « *create* », « *help* », et « *delete* » aux fonctions correspondantes.

Finalement, vous réécrivez le `main()` en utilisant ce tableau à l'intérieur d'une boucle. Lorsque la bonne commande est trouvée, il vous suffit d'appeler la fonction pointée dans l'entrée correspondante du tableau, en passant l'ensemble des arguments de ligne de commande.

Par exemple, si vous appelez le programme

```
./pictDBM list db_file
```

alors, votre code doit appeler `do_list_cmd` avec les paramètres suivants: `argc=2` et `argv = { "list", "db_file" }`.

Votre code doit correctement traiter le cas où la commande n'est pas définie : appeler `help()` et retourner `ERR_INVALID_COMMAND` dans ce cas-là.

Votre code peut tout à fait faire l'hypothèse que toutes les commandes du tableau `commands` sont distinctes. Réfléchissez-y et simplifiez en conséquence.

Gestion des paramètres

Maintenant que l'infrastructure de *parsing* est plus flexible, il est temps de déplacer le traitement des arguments dans les fonctions `do_COMMAND_cmd` correspondantes.

Ceci fait, nous allons ensuite reprendre la commande `create` afin d'y introduire une plus grande flexibilité. Dans cette étape, vous devez rajouter les arguments suivants à la fonction `create`:

- `-max_files <MAX_FILES>`: le nombre d'images maximal dans un seul `pictDB` ;
- `-thumb_res <X_RES> <Y_RES>`: résolution des images « thumbnail » ;
- `-small_res <X_RES> <Y_RES>`: résolution des images en format « petit ».

Dans le premier cas, l'argument est un nombre de 32 bits. Dans les autres cas, les résolutions sont des nombres entiers positifs de 16 bits.

Ces trois nouveaux arguments de la commande `create` sont optionnels. Si l'argument n'est pas spécifié dans la ligne de commande, vous devez continuer à utiliser comme valeur par défaut les valeurs actuellement utilisées par `do_create_cmd` (par exemple `max_files = 10`).

Ces trois nouveaux arguments auront également des valeurs maximales acceptables possibles telles qu'indiquées ci-dessous (dans `help`).

Changer `help`

Changez la commande `help` afin de refléter les nouveaux paramètres de cette façon :

```
> ./pictDBM help
pictDBM [COMMAND] [ARGUMENTS]
  help: displays this help.
  list <dbfilename>: list pictDB content.
  create <dbfilename> [options]: create a new pictDB.
      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
  delete <dbfilename> <pictID>: delete picture pictID from pictDB.
```

Parsing des arguments de la commande `create`

Si ce n'est pas encore fait, la dernière modification à apporter à `do_create_cmd` est de *parser* correctement tous ses arguments – les arguments obligatoires ainsi que les arguments optionnels.

Votre solution doit avoir la structure suivante : * commencez par récupérer l'argument obligatoire (`<dbfilename>`) * itérer ensuite sur `argv` ; * à chaque itération, déterminer tout d'abord si il s'agit d'un argument optionnel acceptable (`-max_files`, `-thumb_res` ou `small_res`) ; * si c'est le cas, vérifier s'il reste encore assez de paramètres pour les valeurs correspondantes (au moins un pour `-max_files` et au moins 2 pour les deux autres) ; si ce n'est pas le cas, retourner `ERR_NOT_ENOUGH_ARGUMENTS` ; * convertissez ensuite le ou les deux paramètres suivants dans le bon type ; vérifier que la valeur est correcte (non nulle ni trop grande) ; en cas d'erreur, retournez soit `ERR_MAX_FILES` (pour `-max_files`), soit `ERR_RESOLUTIONS` ; * si il ne s'agit pas d'un argument optionnel, retourner l'erreur `ERR_INVALID_ARGUMENT`.

A noter :

- les arguments optionnels pourraient être répétés, par exemple `-max_files 1000 -max_files 1291` ; dans ce cas, seule la dernière valeur fait foi ;
- l'argument obligatoire ne peut pas être répété.

Après avoir traité de tous les arguments et paramètres, le reste du code ne change pas, si ce n'est qu'évidemment les fichiers de type `pictDB` doivent être créés avec les valeurs reçues en paramètres !

Dé-duplication des images

Le second compensant de la semaine concerne la dé-duplication d'images, éviter qu'une même image (même contenu) soit présente plusieurs fois dans la base. Au niveau d'un réseau social, ce type d'optimisation permet de gagner beaucoup de place (et de temps).
Pour ce faire, vous devez écrire une fonction `do_name_and_content_dedup` à définir dans un nouveau fichier `dedup.c` (et à prototyper dans `dedup.h`).

Cette fonction retourne un code d'erreur (`int`) et prend deux arguments :

- un fichier `pictDB` précédemment ouvert ;
- un index (type `uint32_t`ici) qui spécifie la position d'une image donnée dans le tableau `metadata`.

Dans le fichier `dedup.c`, implémentez ensuite cette fonction comme suit.

Pour toutes les images valables dans le fichier `db_file` (autre que celle à la position `index`) :

- si le nom (`pict_id`) de l'image est identique à celui de l'image à la position `index`, retourner `ERR_DUPLICATE_ID` ; il s'agit ici de vérifier que la base d'image ne contient pas deux images distinctes avec le même indentifiant interne ;
- si la valeur SHA de l'image est identique à celle de l'image à la position `index`, on peut alors éviter la duplication de l'image à la position `index` (pour toutes ses résolutions).

Pour dé-dupliquer, il faut modifier l'entrée `index` de les métadonnées pour y référencer les attributs de l'autre copie de l'image (les trois offset et les deux/trois tailles ; notez que la taille d'origine est forcément la même).

Note : on ne modifie surtout pas le nom (`pict_id`) de l'image à la position `index` : ce ne sont que les contenus qui sont dé-dupliqués ; on aura deux images de nom différent mais pointant vers le même contenu.

Si l'image à la position `index` n'a pas de doublon de contenu, mettez son offset de `RES_ORIG` à 0. Si l'image à la position `index` n'a pas de doublon de nom (`pict_id`), retourner 0.

Suggestion : comme première étape, écrivez une fonction qui compare deux valeurs de SHA-hash pour déterminer si la valeur est identique.

Tests

Pour le traitement de ligne de commande, tester différentes combinaisons (valables et non-valables).

La dé-duplication ne peut pas être facilement testée cette semaine. Pour l'instant, assurez vous que votre code compile correctement. Vous pourrez tester cette fonctionnalité la semaine prochaine (`do_insert`).

Rendu

Pour rendre la partie correspondant à ces deux dernières semaines, ajoutez les nouvelles versions des fichiers `db_utils.c`, `pictDB.h`, `pictDBM.c`, `db_create.c` et `Makefile`, ainsi que les nouveaux fichiers `dedup.c`, `dedup.h`, `image_content.c` et `image_content.h` dans le répertoire `done/pictDBM` de votre GitHub, puis « pousser » le résultat vers GitHub (`commit` plus `push`).

Le délai pour faire le `push` de rendu est : dimanche 1er mai 23:59.

Projet programmation système W09

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 lundi matin. Pensez également à bien vous répartir le travail. 2. Le rendu de cette semaine est par ailleurs le rendu le plus important du cours (cf barème) et constitue 40% du total. Tout le code du « *command line manager* » que vous avez produit jusqu'ici sera à nouveau évalué. Veillez donc bien à mettre à jour tous vos fichiers (suite aux remarques des corrigés précédents) et à bien rendre le tout.

Pour ces raisons, vous disposez d'une semaine supplémentaire pour rendre votre projet (lundi 16 mai au lieu du dimanche 8 mai) ; cela *ne* veut *pas* dire de commencer plus tard, mais simplement d'étaler un peu plus le travail si nécessaire !

Matériel fourni

Nous vous fournissons trois images JPEG pour tester les fonctionnalités d'insertion et de lecture.

L'exercice de cette semaine se construit sur tout votre travail des semaines précédentes.

Travail à faire

Rajouter les déclarations de fonctions de la semaine

Tout d'abord, rajouter dans `pictDB.h` les déclarations manquantes en replacement 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 `pictdb_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 `pictdb_file` (dans laquelle on ajoutera l'image) ;
  cette fonction retourne un entier (code d'erreur).

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 `db_utils.c`.

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

`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 7 et 2) 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 : 0 ou `ERR_VIPS` 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 *`). En effet, en allant voir le code, cette fonction ne fait appel qu'à `vips_blob_new` où le second argument est correctement qualifié de `const void *`. Vous pouvez donc sans aucun risque (je pense) passer `image_buffer` à `vips_jpegload_buffer` (en le castant... `:-( >:-o`).

`do_insert`

La fonction `do_insert` rajoute une image dans la « pictDB ». Créez un nouveau fichier `db_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_DATABASE` 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 2 pour le calcul des SHA256) ;
- copier la chaîne de caractères `pict_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 dé-duplication 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 !).

`db_read`

La deuxième fonction principale de la semaine est `do_read`, à implémenter dans `db_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 pictDB, n'a pas pu être trouvé).

Intégration dans le programme principal

Dans `pictDBM.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_DATABASE` 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 :

```
original_prefix + resolution_suffix + '.jpg'
```

où :

- `original_prefix` est l'identifiant de l'image ;
- `resolution_suffix` correspond a « `_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 :

```
> ./pictDBM help
pictDBM [COMMAND] [ARGUMENTS]
  help: displays this help.
  list <dbfilename>: list pictDB content.
  create <dbfilename> [options]: create a new pictDB.
      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   <dbfilename> <pictID> [original|orig|thumbnail|thumb|small]:
    read an image from the pictDB and save it to a file.
    default resolution is "original".
insert <dbfilename> <pictID> <filename>: insert a new image in the pictDB.
delete <dbfilename> <pictID>: delete picture pictID from pictDB.
```

Test

En utilisant le matériel fourni, vous pouvez tester les commandes d'insertion et de lecture. Utiliser l'application de visualisation d'images de votre choix pour vérifier que les images ont été correctement transformées.

Nous vous recommandons par ailleurs, pour ce rendu final, de bien re-tester toutes les commandes sur tous les cas particuliers.

Rendu

Le rendu de cette semaine constitue le rendu le plus important du cours (cf barème, 40% du total). Tout le code du « *command line manager* » que vous avez produit jusqu'ici sera à nouveau évalué. Veillez donc bien à mettre à jour tous vos fichiers (suite aux remarques des corrigés précédents) et à **rendre le tout**.

Le délai pour faire le `push` de rendu est : ***lundi*** 16 mai 23:59.

Projet programmation système W10

Introduction

L'utilitaire de ligne de commande `pictDBM` est maintenant opérationnel (nous le compléterons encore en dernière semaine). Nous pouvons alors, cette semaine, commencer le serveur web, qui nous occupera deux semaines. L'objectif principal de cette semaine ci est de mettre en place une première version, de façon à offrir l'équivalent de la fonction `do_list` aux clients de ce serveur.

Lorsqu'il sera complet (semaine prochaine), le serveur web implémentera les mêmes fonctionnalités que l'utilitaire de commande à ce stade, excepté « *create* » qui restera spécifique à la ligne de commande. La différence essentielle est que le serveur tournera en continu : il interagira via une connexion réseau (un `socket`) avec un programme client (qui tournera à l'intérieur d'un navigateur). Nous vous fournissons le code du client, écrit en Javascript (comme l'essentiel des applications web actuelles).

Pour l'implémentation du serveur web, nous nous baserons sur la bibliothèque `libmongoose`. Il s'agit d'une bibliothèque très simple à utiliser qui implémente le protocole HTTP (par exemple, `apache` et `ngnx` sont des serveurs web plus robustes et avec plus de fonctionnalités, mais également beaucoup plus difficile à intégrer).

`libmongoose` s'occupe en particulier d'écouter sur un port choisi (8000 pour nous), d'accepter de nouvelles connections, et d'implémenter le protocole HTTP. Votre projet devra implémenter les commandes spécifiques liées a `pictDB` dans le cadre d'intégration sur le protocole HTTP.

Matériel fourni

Même si l'on passe à un nouvel exécutable, vous construirez également cette phase du projet sur la base des semaines précédentes. En plus, nous vous fournissons :

- `libmongoose` : nous vous fournissons le code source de la bibliothèque ainsi que le `Makefile` pour la compiler ;
- `index.html` : page HTML qui contient le programme client JavaScript qui tournera dans dans votre browser.

Vous aurez de plus à utiliser la bibliothèque `libjson` qui permet de parser et d'écrire des commandes en format « JSON ». Il s'agit du format standard communément utilisé par les applications Javascript, facile à lire (et beaucoup plus simple à traiter et interpréter que XML). `libjson` est une bibliothèque standard à installer dans votre système.

Travail à faire

Installer et utiliser `libjson`

- Pour les utilisateurs `Ubuntu/debian`, vous pouvez installer `libjson` avec la commande :

  ```
  sudo apt-get install libjson-c-dev
  ```

  Pour vérifier que vous avez la bonne version, tapez `apt-cache show libjson-c-dev` et vérifier que la `Homepage` est bien https://github.com/json-c/json-c (il peu y avoir différentes variantes de cette bibliothèque).

- Pour les utilisateurs OS X, il faut compiler et installer manuellement `libjson` :

```
git clone https://github.com/json-c/json-c.git
cd json-c
sh autogen.sh
./configure
make
make install
```

Pour vérifier l'installation, vous pouvez ensuite taper `make check`.

Pour utiliser la bibliothèque :

- l'interface est définie dans `<json-c/json.h>` — à étudier ; rajouter le `include` dans tous les fichiers `.c` qui ont besoin de cette fonctionnalité ;
- il est possible que vous deviez rajouter `-ljson-c` dans `LDLIBS` dans votre `Makefile`.

La documentation de l'API se trouve ici : http://json-c.github.io/json-c/json-c-0.11/doc/html/index.html

Les fonctions que vous aurez à utiliser sont : * `json_object_new_array()` * `json_object_new_string()` * `json_object_array_add()` * `json_object_new_object()` * `json_object_object_add()` * `json_object_to_json_string()` * `json_object_put()`

Modifier do_list()

Le premier objectif est d'intégrer le format JSON dans l'application `pictDBM`; cette étape est indépendante de l'intégration du serveur web et peut se faire en parallèle dans le cadre de votre binôme.

Nous allons changer le prototype de `do_list` afin de 1. prendre comme paramètre additionnel un `enum` pour choisir entre les différents formats ; et 2. retourner le contenu comme une chaîne de caractères (plutôt que d'imprimer directement à l'intérieur de la fonction).

Concrètement, il faut faire les modifications suivantes dans `pictDB.h` : * définir un `enum do_list_mode` comprenant les modes `STDOUT` et `JSON` ; * changer le prototype de `do_list` pour qu'il retourne une chaîne de caractères et qu'il prenne un paramètre supplémentaire de type `enum do_list_mode`.

Modifier ensuite l'implémentation de `do_list` (dans `db_list.c`) :

- si le mode est `STDOUT`, la fonction doit opérer comme avant et retourner `NULL` ;
- si le mode est `JSON`, la fonction doit utiliser la bibliothèque `libjson` (voir ci-dessus) pour construire un objet JSON avec la structure suivante :

  ```
  {
      "Pictures": [] # an array of the strings of the pict_id fields from the metadata
  }
  ```

  (c'est donc un «*objet*» JSON qui contient une «*array*» de «*string*» qui sont les `pict_id`) ; puis le convertir en chaîne de caractères (`json_object_to_json_string`) pour le retourner.
  Attention à la durée de vie/portée des objets manipulés !.. En particulier, les chaînes contenues dans un «*objet JSON*» lui appartiennent et disparaissent avec lui.
- si le mode n'est pas connu (ni `STDOUT` ni `JSON`) la chaîne retournée sera un *message* d'erreur, p.ex. `"unimplemented do_list mode"`.

Compiler et utiliser libmongoose

Nous vous fournissons `libmongoose` dans un répertoire qui contient un Makefile qui génère une bibliothèque `libmongoose.so`.

Dans le `Makefile` du projet lui-même, vous devez vous assurer : 1. que le répertoire `libmongoose` est dans la liste des répertoires de bibliothèques (`LDFLAGS` avec le préfixe `-L`) ; et 2. de rajouter la bibliothèque dans la liste des éléments de compilation en rajoutant `-lmongoose` dans `LDLIBS`.

Développer le serveur web

Tout est en place. Il s'agit maintenant d'écrire le serveur web qui combinera 1. la bibliothèque `libmongoose` qui se charge du protocole HTTP, 2. avec

les fonctionnalités développés depuis plusieurs semaines pour le traitement du format `pictDB`.

Vous aller écrire le code du serveur web dans le fichier `pictDB_server.c`.

Dans le `Makefile`, créer une deuxième cible exécutable `pictDB_server` en plus de `pictDBM`. c.-à-d. que `make pictDB_server` doit compiler et faire l'édition de liens d'un programme exécutable (`pictDB_server`) qui est lui-même un serveur web.

Pour la structure de `pictDB_server.c`, commencer par étudier quelques exemples fournis par libmongoose, en particulier :

- simplest_web_server
- coap_server
- restful_server
- json_rpc_server

(Il ne s'agit pas de tout comprendre en détail, mais de s'en inspirer)

La documentation se trouve ici : https://github.com/cesanta/mongoose/tree/master/docs ; par exemple : `mg_set_protocol_http_websocket`.

Configuration de départ du serveur web

Lorsque le serveur web démarre, il doit prendre un seul argument de commande : le nom du fichier qui a les données en format `pictDB` ; par exemple :

```
./pictDB_server testDB02.pictdb_dynamic
```

En particulier, veillez à ce que :

- au démarrage, il utilise la fonction `db_open` pour ouvrir le `pictDB` ; il doit ensuite imprimer (sur le terminal) l'entête en utilisant la fonction `print_header` (des semaines passées) ;
- à l'arrêt (« *shutdown* »), il doit utiliser la fonction `db_close`.

Pour la gestion de l'arrêt, la version la plus simple mais pas très propre est de s'inspirer de l'exemple « simplest_web_server » ; une version plus correcte mais *plus avancée* étant de gérer les signaux (= messages du système) `SIGINT` et `SIGTERM` comme fait dans l'exemple « coap_server ».

Interface et protocole

Cette semaine, nous nous limitons a la fonction de liste des images.

Le protocole de communication est imposé (et utilisé par le programme client `index.html` fourni) :

- le serveur doit écouter le port 8000 ;
- le serveur doit répondre par une réponse HTTP valide, en format JSON a l'URI `/pictDB/list` (l'équivalent de `/api/v1/sum` ou `/printcontent` dans l'exemple restful_server) ; écrivez pour cela une fonction `handle_list_call` ;
- pour toutes les autres commandes, il est acceptable qu'il serve simplement du contenu statique en appelant la fonction `mg_serve_http`, à savoir en appelant :

  mg_serve_http(nc, hm, s_http_server_opts); /* Serve static content */

(comme dans l'exemple restful_server).

Le message HTTP que la commande de liste fournit doit avoir le format suivant :

```
HTTP/1.1 200 OK\r\n
Content-Type: application/json\r\n
Content-Length: XXX\r\n\r\n
YYY
```

La valeur `XXX` correspondent à la longueur de la chaîne de caractères fournie (`YYY`) ; le contenu `YYY` correspond au contenu de `do_list` retourné en format `JSON`.

Faites attention aux éléments suivants :

- utiliser les routines de `libmongoose` pour écrire la réponse dans la connexion (voir par exemple l'exemple json_rpc_server) ;
- ne pas faire de fuite mémoire.

Tests

Pour tester la nouvelle version de `list`, vous pouvez simplement modifier son appel dans votre ancien `pictDB` et voir si vous obtenez la bonne séquence JSON, par exemple :

```
{ "Pictures": [ "pic2", "pic3" ] }
```

Pour tester votre serveur web, lancez simplement votre `pictDB_server` depuis un répertoire où se trouve le `index.html` fourni, puis ouvrez `http://localhost:8000/` dans un navigateur Web. Vous devriez obtenir quelque chose comme ça (dépend de la pictDB avec laquelle vous lancez votre serveur, ici la `testDB02.pictdb_dynamic` fournie en semaine 7) :

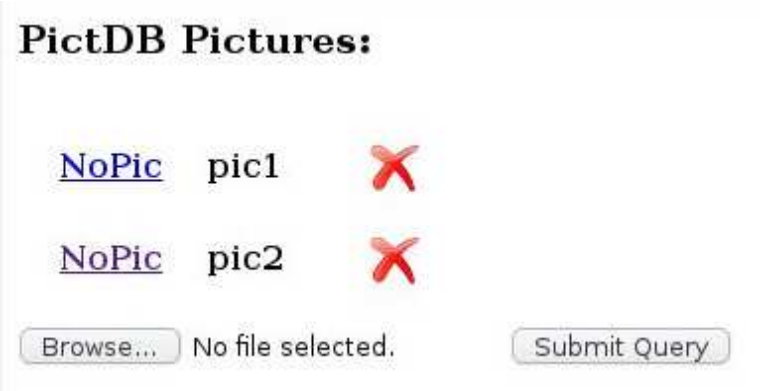

Figure 1: Résultat de la commande list du serveur Web

NOTE : pour que la bibliothèque `libmongoose` soit trouvée lors de l'exécution de `pictDB_server`, il faut indiquer au système le *répertoire* où la chercher en affectant la variable d'environnement :

- sous Linux : `LD_LIBRARY_PATH` ;
- sous Mac OS : `DYLD_FALLBACK_LIBRARY_PATH`.

Par exemple (sous Linux) :

- exemple 1 (en 2 commandes indépendantes) :

```
export LD_LIBRARY_PATH=../provided/libmongoose
./pictDB_server testDB02.pictdb_dynamic
```

- exemple 2 (1 seule ligne) :

```
LD_LIBRARY_PATH=../provided/libmongoose ./pictDB_server testDB02.pictdb_dynamic
```

Rendu

Vu que nous n'avons qu'un travail partiel, il y a cette semaine une pause dans les rendus : rien à rendre cette fois, mais comme la charge de cette semaine est relativement importante, faites attention à ne pas prendre de retard !

Projet programmation système W11

Introduction

L'objectif de cette semaine est de compléter le serveur web en implémentant les fonctions manquantes :

- le rajout d'une image (« *upload*») ;
- la lecture d'une image, dans la résolution demandée (« *download* ») ;
- l'effacement d'une image.

Matériel fourni

Aucun, vous vous basez sur votre travail de la semaine passée.

Travail à faire

Nouvelles fonctions

Fonctions utilitaires

Parsing

Le serveur web doit implémenter différents type de commandes, et chaque commande a ses propres arguments. Le format générique d'un URI (« *Uniform Resource Indicator* ») est :

```
http://servername:port/dir1/dir2/dir3/file?arg1=foo&arg2=bar
```

Nous devons parser les arguments de telles URIs pour nos commandes. Par exemple, l'URI pour charger une image sera :

```
http://localhost:8000/pictDB/read?res=orig&pict_id=chaton
```

La bibliothèque `mongoose` sépare déjà le « *servername* » et « *port* » de l'URI. Dans votre « *handler mongoose* », la structure de type `struct http_message` contient les champs `uri` et `query_string` qui correspondent respectivement à `/pictDB/read?res=orig&pict_id=chaton` et à `res=orig&pict_id=chaton`.

A noter : ces informations ne sont pas des chaînes de caractères standard de C mais sont de type `mg_str` ; leur longueur est par exemple spécifiée explicitement par le champs `len` (et non implicitement par un `\0'` final). Il y a une logique

à cela : cela permet une utilisation plus efficace de la mémoire, sans dupliquer l'information ;. par exemple, `uri` et `query_string` se chevauchent en fait dans la mémoire.

La bibliothèque `mongoose` n'interprète pas plus que cela les arguments de la `query_string` et c'est à vous de le faire.

Dans une première étape, nous vous recommandons d'implémenter une fonction qui va séparer la `query_string` en morceaux, avec la signature suivante :

```
void split (char* result[], char* tmp, const char* src, const char* delim, size_t len);
```

où:

- `result` est un tableau de pointeurs de chaînes de caractères de taille `MAX_QUERY_PARAM` qui sera le nombre maximum de paramètre que nous accepterons ; on peut par exemple définir :

  ```
  #define MAX_QUERY_PARAM 5
  ```

- `tmp` est une chaîne de caractères de taille `(MAX_PIC_ID + 1) * MAX_QUERY_PARAM` qui contiendra l'ensemble des parties de la « *query string* » l'une dernière l'autre (on va en fait y copier la « *query string* » puis remplacer les `'&'` ou `'='` par des `'\0'`) ;
- `src` est la « *query string* » à découper ;
- `delim` est une chaîne de caractères qui contient la liste de délimiteurs de token (voir la manpage de la fonction `strtok`) ; par exemple `"&="` signifie que les caractères `'&'` et `'='` séparent des morceaux que l'on désire ;
- `len` est la longueur de la « *query string* » à découper (nous rappelons que les `mg_str` ne sont *pas* terminés par le caractère `'\0'`, mais que la longueur est donnée par le champ `len`).

Pour implémenter cette méthode, utilisez la fonction `strtok` (lisez attentivement sa `man page`).

L'objectif de cette routine est de séparer les arguments, à savoir pour notre exemple ci-dessus :

```
result[0] = tmp
result[1] = tmp + 4
result[2] = tmp + 9
result[3] = tmp + 17
result[4] = NULL
```

avec `tmp` contenant `"res\0orig\0pict_id\0chaton"` ; ce qui fait que du point de vue C, on aura finalement :

```
result[0] = "res"
result[1] = "orig"
result[2] = "pict_id"
result[3] = "chaton"
result[4] = NULL
```

A noter que les chaînes de caractères de `result` sont bien toutes terminées par un `\0` (voir à nouveau `strtok`).

Erreurs HTTP

Nous vous recommandons également d'écrire une routine qui va retourner un code d'erreur HTTP — à utiliser si les arguments de la `query` ne correspondent pas à l'API spécifiée, ou si la requête ne peut être traitée avec succès.

Cette méthode appellera `mg_printf` pour retourner :

- le code d'erreur HTTP 500 (chaîne `"HTTP/1.1 500 "`) ;
- suivi du message d'erreur (tableau `ERROR_MESSAGES`).

La « *Content-Length* » d'un code d'erreur HTTP est zéro.

Le prototype de cette méthode doit être :

```
void mg_error(struct mg_connection* nc, int error);
```

Lecture

Implémenter une fonction `handle_read_call`, équivalent de `handle_list_call` mais pour le préfixe d'URI `/pictDB/read`.

Cette fonction doit appeler `split` avec les délimiteurs standard `"&="` puis ensuite en interpréter les résultats par paire (argument, valeur) pour les arguments suivants :

- `res` : la version texte de la résolution d'image demandée ; à convertir avec `resolution_atoi` (cf la commande `read` de votre interpréteur de commande des semaines précédentes) ;
- `pict_id` : la clé de recherche de l'image dans la base (son « nom » dans la base).

Ces deux paramètres sont obligatoires, mais l'ordre n'est pas important.
Exemple d'URI :

```
http://localhost:8000/pictDB/read?res=orig&pict_id=chaton
```

Appelez ensuite la fonction `do_read` avec les bons arguments.

En cas de succès, envoyer la réponse HTTP avec le format suivant :

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: <XXX>
```

suivi de l'image elle-même. La longueur de `Content-Length` doit être la taille de l'image (en octets). Utiliser la fonction `mg_send` pour envoyer le contenu de l'image elle-même.

En cas d'erreur, appeler la fonction `mg_error` que vous avez implémenté ci-dessus.

A noter : penser à bien désallouer la mémoire dynamique qui aurait été allouée par le programme...

Insertion

Il s'agit d'implémenter la fonction `handle_insert_call` pour traiter les commandes d'URI `/pictDB/insert`.

La logique d'insertion est différente de celle qui retourne la liste (`list`) ou une image (`read`). En effet, l'insertion utilise la commande `HTTP POST`, alors que les deux autres utilisent `HTTP GET`. En gros, un `GET` contient tous les arguments dans l'URI, alors qu'un `POST` a des arguments additionnels en plus de l'URI. En particulier, la commande `/pictDB/insert` utilise un `POST` pour le contenu même de l'image à insérer.

Pour ce faire, utiliser la fonction `mg_parse_multipart`. Elle vous permettra de récupérer a la fois le nom de l'image et l'image elle-même. Référez-vous à la documentation de `libmongoose` pour les détails.

Une fois que vous avez ces paramètres, appelez `do_insert`.

En cas d'erreur, faites attention de bien retourner un message d'erreur (en utilisant `mg_error`). En cas de succès, retourner le message suivant :

```
HTTP/1.1 302 Found
Location: http://localhost:PORT/index.html
```

où `PORT` est le port HTTP utilisé par le serveur (8000).

Effacer une image

Implémenter la fonction `handle_delete_call` en réponse aux requêtes HTTP d'URI `/pictDB/delete`.

Pour ce faire, vous devez d'abord récupérer les arguments suivant une logique similaire à `/pictDB/read`.

Cet URI n'attend qu'un seul argument : `pict_id`

Une fois l'argument récupéré (et vérifié), appeler la fonction `do_delete`. En cas de succès, retourner la même réponse HTTP qu'en cas d'insertion :

```
HTTP/1.1 302 Found
Location: http://localhost:PORT/index.html
```

où `PORT` est le port HTTP utilisé par le serveur (8000).

En cas d'erreur, utilisez l'usuel `mg_error`.

Finalisation

N'oubliez pas d'ajouter vos trois nouveaux URI au `db_event_handler`.

De plus, puisque la lecture des images utilise la bibliothèque VIPS (rappelez vous de `lazy_resize`), il ne faudra pas oublier de la démarrer (`VIPS_INIT`) au lancement du serveur, et de la fermer (`vips_shutdown()`) à son arrêt.

Makefile

Important — n'oubliez pas d'inclure dans votre dépôt GIT votre `Makefile`, qui doit pouvoir compiler avec succès votre projet `pictDB_server`. Vous devrez probablement modifier le `Makefile` afin d'y inclure la bibliothèque VIPS.

Tests

Pour tester votre serveur web, comme la semaine passée : lancez simplement votre `pictDB_server` depuis un répertoire où se trouve le `index.html` fourni la semaine passée, puis ouvrez `http://localhost:8000/` dans un navigateur Web. Vous devriez obtenir quelque chose comme ça (cela dépend de la pictDB avec laquelle vous lancez votre serveur) :

- Cliquez sur une croix rouge à droite pour faire un `delete`.
- Cliquez sur une image pour la voir en taille d'origine (`read`).
- Utilisez les boutons « *Browse* » puis « *Submit Query* » pour ajouter une image (`insert`).

Figure 1: Résultat de la commande list du serveur Web

Rendu

Rendez tout le code de votre serveur Web (et `Makefile`).

Le délai pour faire le `push` de rendu est : dimanche 29 mai 23:59.

Projet programmation système W13

Introduction

Dans cette semaine finale (la dernière vous étant laissée pour finaliser ou ajouter quelques extensions optionnelles à votre projet), vous allez rajouter une dernière commande à votre utilitaire de ligne de commandes (« *command line manager* ») `pictDBM` : la commande `gc`.

Cette commande fait un certain « nettoyage (*garbage collecting*) » de la base d'images (et non pas dans la mémoire gérée p.ex. comme en Java). Elle va collecter et supprimer divers « trous » dans un fichier au format `pictDB` sur le disque. En effet, puisque la commande `delete` n'efface pas directement les images et ne réduit pas la taille du fichier mais rend simplement leur contenu inaccessible, certaines parties du fichier sont en fait inutiles. `gc` va les supprimer, sans, bien sûr, rien modifier au contenu effectivement utilisé (le résultat de la fonction `list`, par exemple, doit rester inchangé).

La commande `gc` aura donc comme conséquences de :

- ne pas changer du tout le fichier s'il y n'y a pas de « trous » (et, en toute rigueur, ni de doublon de « petites » images ; mais c'est un détail ici) ;
- ne pas changer le contenu visible/utilisé de la base d'images ;
- ne pas changer la taille du tableau de `metadata` ;
- réduire au maximum la taille du fichier s'il existe des images non utilisées ;
- peut-être supprimer certains des doublons de « petites » images (« *small* » ou « *thumnail* ») éventuellement existantes (mais pour simplifier au niveau de ce projet, ceci ne sera qu'un effet secondaire de la méthode simple proposée et non pas un nouvel algorithme, ni un but en soi).

L'objectif principal de cette semaine est donc d'implémenter `gc` tout en :

- réutilisant des fonctions existantes autant que possible ;
- écrivant aussi peu de lignes de code que possible ;
- modularisant votre code autant que possible ;
- et profitant bien de cet exercice en vous amusant, puisque c'est le dernier `:-)`.

Un objectif secondaire est, si ce n'est pas déjà fait, de mettre à jour votre utilitaire « *command line manager* » `pictDBM` par rapport aux modifications des dernières semaines.

Matériel fourni

Aucun nouveau matériel n'est fourni cette semaine ; vous vous basez sur votre travail des semaines passées.

Travail à faire

Modifications préliminaires

Vous devez apporter deux modifications mineures à votre « *command line manager* » par rapport à la semaine 9 :

- reporter la modification faite en semaine 10 à la commande `do_list` (ajout d'un mode) ;
- si nécessaire (cela devrait normalement être le cas), modifier la commande `do_create` (et son utilisation) de sorte à ce que la fermeture de fichier ne se fasse pas dans cette commande ;
  nous allons en effet utiliser la commande `do_create` dans le code de cette semaine en *continuant* à utiliser la base créée ; il est donc nécessaire de revoir ce point (fermeture) : laisser le fichier ouvert un moment plutôt que de bêtement le fermer (dans `do_create`) puis le ré-ouvrir juste après pour manipulation par le « *garbadge collector* ».

Garbage collection

La commande `gc` est similaire à tous les autres commandes dans le « *command line manager* ». Elle prend deux arguments de ligne de commande : 1. le nom du fichier `pictdb` à traiter ; 2. un nom de fichier temporaire (qui *a priori* n'existe pas encore ; mais ce n'est pas la peine de tester ça).

Pourquoi ce fichier temporaire ?
Pour plusieurs raisons (simplification du sujet, mais aussi garantie d'intégrité en cas d'échec), la commande `gc` ne fait pas le « *garbage collecting* » sur place dans le fichier d'origine, mais en *copiant* le contenu. Cette copie nécessite un fichier transitoire temporaire dont il est préférable de donner le nom ici dès le départ : l'ingénieur système décidant de lancer un « *garbage collecting* » sur une (grosse) base d'images choisit ainsi où il souhaite que son fichier temporaire soit stocké (et accessoirement comment il va s'appeler, pour éventuellement le retrouver, en cas de crash par exemple).

Un exemple de commande `gc` serait donc :

```
./pictDBM gc my_favorite_dbfile.pictdb /tmp/gc_20160523082000.pictdb
```

Le cœur de cette commande devra être implémenté dans une fonction `do_gbcollect` à mettre dans un nouveau fichier `db_gbcollect.c`. La fonction `do_gbcollect` retourne un `int` correspondant à un code d'erreur ou 0 en cas de succès et prend 3 arguments :

1. un `pictdb_file` ;

2. le nom original du fichier`pictdb` ouvert (il aura déjà été ouvert et correspond au fichier pointé dans le `pictdb_file`) ;
3. le nom d'un nouveau fichier, qui sera utilisé temporairement pour la nouvelle version, « nettoyée ».

Durant l'essentiel de la phase de « nettoyage », il y aura donc deux fichiers avec les mêmes images : le fichier original (non modifié) et le fichier temporaire, en cours de création/modification, version « nettoyée » du fichier d'origine.

Une fois la nouvelle version totalement terminée, le fichier temporaire est alors renommé avec le nom permanent (nom d'origine). Pour cette rocade, nous vous demandons d'utiliser simplement les fonctions `rename` et `remove` de `stdio.h` ; une version complète et robuste étant beaucoup trop compliquée à ce niveau (pour ceux que cela intéresse, voir par exemple le code source de `mv` et de `copy`).

Note technique : À noter donc, qu'en raison de ces simplifications (fonctions `rename` et `remove`), le fichier temporaire doit nécessairement être sur le même système de fichiers (*filesystem*) que le fichier de départ. Il se peut donc que, sur certaines machines, la commande ci-dessus avec un fichier temporaire sur `/tmp` ne fonctionne pas correctement, typiquement lorsque `/tmp` est un autre *filesystem* que celui de votre fichier de départ. Dans ce cas, testez simplement votre programme avec des commandes du genre :

```
./pictDBM gc my_favorite_dbfile.pictdb tmp.pictdb
```

(où les deux fichiers sont donc sur le même *filesystem*).
[fin de note]

Pour implémenter la fonction `do_gbcollect`, réutilisez le plus possible les fonctions existantes. Dans le cas idéal, cela inclut `do_create`, `do_read`, `do_insert` et `lazily resize`. Vous pouvez faire toutes les modifications dont vous avez besoin (et uniquement celles qui sont nécessaires) avec et sans changer la sémantique des ces autres commandes.

La fonction `do_gbcollect` doit conserver toutes les images utiles, y compris les « petits formats » (« *small* » ou « *thumnail* ») déjà présents (sauf doublons, bien sûr ; et sans en rajouter).

En cas d'erreur sur `remove` ou `rename`, la fonction retournera simplement `ERR_IO`.

Enfin, la fonction `do_gbcollect` doit fermer tous les flots qu'elle a ouvert.

Intégration dans l'utilitaire

Comme d'habitude, il faut ajouter la fonction correspondant à `do_gbcollect` dans le « *command line manager* » :

1. rajouter une fonction `do_gc_cmd` qui, si la commande `gc` a été donnée sur la ligne de commande, va au moins appeler la fonction `do_gbcollect` ;
2. complétez l'aide pour ajouter l'information suivante :

```
gc <dbfilename> <tmp dbfilename>: performs garbage collecting on pictDB. Requires a tem
```

Makefile

Faire les modifications nécessaires sur votre `Makefile`.

Tests

Pour vérifier votre implémentation, vous pouvez :

- insérer plusieurs images (y compris des doublons. . .) ;
- faire quelque `read` pour ajouter des « petites » images ;
- faire `list` pour vérifier ; utiliser la commande `ls` pour déterminer la taille du fichier ;
- effacer quelques images ;
- encore faire `list` et vérifier la taille du fichier ; normalement, elle n'a pas changé ;
- faire `gc` ;
- vérifier que la sortie de `list` n'a pas changé et que la taille du fichier est réduite.

Rendu

Ce rendu fait partie du rendu final. Pour cela, vous devez rendre la version complète du webserver et la version complète du command line manager (avec la fonction `gbcollect`).

Ce rendu est le deuxième plus important du cours — 25% du total, cf le barème.

Le délai pour faire le `push` de rendu est : **dimanche 5 juin 23:59**. Aucune extension de délai d'aucune sorte ne saurait être accordée.

Projet programmation système W14

Description

Il n'y a plus rien de nouveau à faire cette semaine. Elle est consacrée à la finalisation et au rendu définitif de votre projet.

Si vous le *souhaitez*, vous pouvez ajouter des extensions, totalement *optionnelles*, telles que :

- faire du « *command line manager* » un vrai intepréteur de commandes : ajoutez une commande `interpretor` (argument de la ligne de commande) qui lance une boucle de lecture qui lit les commandes sur l'entrée standard (au lieu de les lire sur les arguments de la ligne de commande) et ajoutez une commande `quit` pour en sortir ;
- améliorer le webserver : ajouter de vraies pages d'erreur (en HTML) (en l'état les erreurs sont simplement enyovées en HTTP, mais ne correspondent à aucun affichage de contenu) ;
- améliorer le webserver : prendre en compte les caractères spéciaux passées dans les URL (« *URL encoding* »; cf http://moodle.epfl.ch/mod/forum/discuss.php?d=6994) ;
- améliorer le « *garbage collector* » : ajoutez-y une déduplication complète de toutes les résolutions (« *small* » et « *thumbnail* »).

Rendu

Ce dernier rendu est le deuxième plus important du cours — 25% du total, cf le barème et doit contenir **tout** le code du projet : la version complète du webserveur **et** la version complète du « *command line manager* » (avec la fonction `gbcollect`).

Le délai pour faire le `push` de rendu est : **dimanche 5 juin 23:59**. Aucune extension de délai d'aucune sorte ne saurait être accordée.