Projet programmation concurrence et parrallelle

Axel Viala <axel.viala@darnuria.eu>

Version du: Mercredi 22 janvier 2020

Objectifs : Ce travail pratique a pour but de vous initier à la programmation concurrente et de mettre en pratique les concepts de théories des systèmes d'exploitation que nous avons vu ensemble.
Vue d'ensemble : On souhaite écrire un programme qui compte avec plusieurs threads le nombre de pixels noirs et en bonus fait des manipulations d'images dans une image au format PixMap.

Rendus : J'attends dans votre rendu votre code documenté.

Consignes : L'image devra être lue en format binaire et gérer un encodage des pixels sur 24 bits, 8 bits pour chaque canal bleu, jaune, rouge. Je vous recommande de bien lire la page Wikipédia sur le format. PortablePixMap (PPM)

Vous pourrez compiler votre projet à l'aide de la commande suivante : `gcc -std=c11 -pthread -Werror -Wall -Wextra main.c -o superpixmanip`

Exigences : Attention j'exige que votre code compile sans erreur mémoire, ni warning de compilation avec `-Wall -Wextra`.
Je demande aussi que le code soit documenté, vous pouvez suivre le standard de documentation JSdoc ce format est reconnu par de nombreux éditeurs de texte.
Norme C : Vous pouvez déclarer et directement assigner où vous voulez, cependant une bonne organisation du code est attendue. Si vous avez besoin de tableau agrandissable, codez ce dont vous avec besoin ou utilisez les Garray de la glib .
Indices : Tentez d'utiliser des structures `struct` et des fonctions pour les manipuler. C'est la clef pour écrire des abstractions en C. ;)
Faites des fonctions simples bien documentées avec un nom explicite, pensez à écrire des structures quand ça deviens complexe. Écrivez des test de vos fonctions, et utilisez la fonction `assert` pour avoir des vérifications à l'execution.
Évitez les pointeurs de pointeurs bien souvent on se plante.

1 Questions

1. Écrire une structure `pixel_t` qui représente un pixel dans votre image, un pixel est représenté par 3 canaux, un pour le rouge, un pour le vert un pour le bleu, l'intensité du canal va de 0 à 255. Indice : Un canal d'un pixel va de 0 à 255 quelle nombre de bit avez vous besoin et donc quel `uintX_t` allez vous utiliser pour représenter un canal ?
2. Écrire une fonction pour construire un pixel :

   ```
   pixel_t pixel_new(uint8_t red, uint8_t green, uint8_t blue);
   ```

   .
3. Écrire des fonctions pour obtenir chacunes des couleurs d'un `pixel_t` :
   — `uint8_t pixel_blue(const pixel_t * p);`
   — `uint8_t pixel_red(const pixel_t * p);`
   — `uint8_t pixel_green(const pixel_t * p);`

4. Écrire une fonction

   ```
   bool pixel_equals(const pixel_t *self, const pixel_t * other);
   ```

   pour vérifier que deux `pixel_t` sont égaux entre eux. `bool` proviends de `stdbool.h`.

5. Écrire une fonction pour inverser la valeur d'un pixel (faire un negatif), réutilisez

   ```
   pixel_t pixel_invert(const pixel_t * p);
   ```

   , Il faut inverser bit à bit les bits du pixel. Indice : L'opérateur ~ est votre ami ;

6. Écrire une `struct ppm_image_t` qui représente votre image ; vous aurez besoin de quoi sauvegarder la hauteur `height`, largeur `width` et un tableau de pixels avec sa taille `length` Indice : Le type standard `size_t` peut être utile pour representer des tailles.

7. Écrire une fonction pour construire votre image : `ppm_image_t ppm_new(const char *pathname);` depuis un chemin. Si vous préférez ous pouvez écrire la fonction :

   ```
   ppm_image_t * ppm_malloc(const char *pathname);
   ```

   qui alloue la struct dans la heap.

8. Écrire des fonctions dont vous prefixez leur nom par `ppm_` suivi du nom de votre fonction, pour accéder à la hauteur, largeur et taille du tableau de pixel depuis la structure `ppm_image_t`, Indice : vous pourrez marquer `const` le pointeur vers votre structure car nous ne modifions pas la struct.

9. Écrire une fonction pour accéder à un `pixel` depuis la structure `ppm_image_t` :

   ```
   pixel_t ppm_pixel(const pgm_image_t *img, const size_t x, const size_t y);
   ```

   Indice le calcul pour accèder à un element avec x et y dans un tableau monodimentionnel est le suivant $index = x + width * y$ (à vous de transformer ça en C ;)).

10. Écrire une fonction de signature `void ppm_negatif(ppm_image_t *img);`pour obtenir le négatif de votre image, réutilisez

    ```
    pixel_t pixel_invert(const pixel_t * p);
    ```

    .

11. Réussir à lire en *binaire* un fichier au format PPM avec des pixels sur 24 bits et les charger dans la structure `ppm_image_t`.

12. Afficher dans le terminal la valeur d'un pixel lu et chargé en mémoire depuis un fichier ppm.

13. Écrire le code néccessaire pour compter les pixel noirs à un thread puis à 2 threads. Attention réflechissez bien à comment vous découpez le travail. ;)

2 Bonus

Les bonus sont principalement pour le plaisir.

1. Écrire une mini ligne de commande avec la bibliothèque Argp pour le logiciel. Tutoriel d'usage : https://makework.blog/
2. Écrire un encodeur et décodeur Run Length Encoding pour améliorer la taille en terme de mémoire de nos images.
3. Refaire le comptage des pixels noirs en mono-thread avec l'encodage RLE, quel est le gain ? Mesurer les performances avec et sans.
4. Gérer les formats 8 et 16 bits du format Netpbm.
5. Écrire une fonction de signature `void ppm_rotation(ppm_image_t *img);`pour obtenir une image ayant subie une rotation. Bonus, bonus : Écrire cette fonction sans allouer de second tableau. ;)

6. (optimisation) Écrire une fonction pour faire le negatif non plus byte à byte mais par paquets de 32 bits. identifier les problèmes de cette approche, et le gain subtentiel. Attention vous devrez écrire du code spécial pour le début et la fin du traitement. ;)
7. Si vous voulez allez plus loin regardez les exercices : « Filtrage d'images » du travail pratique du cours 2I001 de licence 2 de Sorbonne Université dont s'inspire librement ce sujet de TP. Ils sont disponnibles :ici. Sorbonne Universités possèdes les droits sur le document de TP référé par le lien précédent.

3 Rappels

En C une structure sert à représenter en C un type de données. En mémoire les différents champs de votre structure seront contingus (côtes à côtes dans la mémoire).
Pour accèder à un champ depuis un pointeur sur votre structure par exemple `string_t * s` il faudra écrire `s->champ`.

Si il n'y a pas de pointeur c'est `s.champ`.
Imaginons que nous souhaitions faire un type pour abstraire des chaînes de caractères avec leur taille.

```c
#include <string.h>
#include <stdint.h>
#include <stdlib.h>

// typedef is used to make a type alias. With it you can avoid
// typing `struct _string_t`, you just have to type `string_t`.
typedef struct _string_t {
    uint8_t * ptr;    // Pointer to the string memory zone.
    size_t    length; // length of the string
} string_t;


/** Construct a string_t with coping the array behind ptr in order to avoid
 *  pointers aliasing bug. */
string_t string_new_copy(const char * ptr) {
    const size_t length = strlen(ptr);
    uint8_t * copied = malloc(sizeof(uint8_t) * length);
    memcpy(copied, ptr, length);

    string_t s;
    s.length = length;
    s.ptr = copied;
// Here we return by copy (may be optimized) our struct
// here it's ok because the struct is small (64 + 64 bits).
    return s;
}
```

Si nous voulions faire une fonction pour manipuler votre `string_t`, il suffit de prendre par pointeur votre structure comme ici :

```c
/** Returns the length of a `string_t`. */
size_t string_length(const string_t * string) { return string->length; }

/** Return a constant pointer usable with syscall and libC function.
 *  becareful with usage of the pointer! :) */
```

```c
const char * string_os_str(const string_t * string) { return (const char *)string->ptr; }

/** Return a unsafe mutable pointer usable with syscall and libC function.
 *  becareful many times with usage of the pointer! :) */
const char * string_os_str_unsafe(string_t * string) { return (char *) string->ptr; }
```