TP 2– fork(), execv() et traitement d'image multi-process

Concepts abordés

- Manipulation de l'appel système fork()
- Concept de serveur multi-utilisateur
- Lecture de fichiers

Fonctions utiles

- execv() / execvp() / execl() / execlp() (au choix)
- fork() , wait()
- getpid()
- malloc(), free()
- system()
- mmap()
- kill (), sigaction()

N'hésitez pas à utiliser la commande man pour obtenir la documentation de chaque fonction.

Contraintes et remise des TPs

- **Le code de chaque exercice doit être dans un repertoire séparé (exercice1, exercice2, etc.).**
- **Chaque répertoire d'exercice doit contenir un fichier** `Makefile` **permettant de compiler les réponses à chaque question.**
- Quand une question ne demande pas uniquement du code mais aussi du texte, des explications, des commentaires, etc. vous pouvez soit les rendre dans un fichier texte (.txt) dans le dossier de l'exercice, soit si ce n'est pas trop gros les mettre en commentaire dans votre code. **Si une question demande des explications ou une réponse texte et que vous vous contentez de rendre du code, vous n'aurez pas tous les points.**
- Une note de propreté et style du code est attribuée à votre travail et prise en compte lors de l'évaluation. Elle inclue : la lisibilité du code (taille des fonctions, noms de variables explicites, etc.), sa propreté (libération de mémoire, contrôle des erreurs, réemploi de variables, etc.), sa qualité algorithmique, la présence et la pertinence des commentaires, les gardes `#ifndef` dans les headers, l'indentation et mise en page. L'ensemble de ces consignes est détaillé dans le document "propreté et lisibilité du code" disponible sur la page du module.
- Le code source doit être **mis en ligne sur Moodle** sous forme d'une archive tar.gz [1] **au plus tard 15 minutes après la fin du TP.** Votre archive doit être propre : pas d'exécu-

1. Pour faire une archive tar.gz : `tar cvzf mon_archive.tar.gz mon_repertoire`

tables, pas de .o, pas de fichiers temporaires. En cas de problème de remise, contacter vos encadrants.
— Les supports de cours, les exemples et les fichiers utiles aux TPs sont **sur la page du cours sur Moodle**.

Barème de notation

Exercice 1 : coefficient 2. Exercice 2 : coefficient 3. Style, propreté et lisibilité du code : coefficient 1.

Exercice 1 – préliminaires (maximum : 2 heures)

1. Écrivez un programme affichant son PID puis utilisant des appels à `fork()` dans une boucle **for** pour générer cinq processus enfant, puis se terminer. Chaque enfant doit lui-même afficher son PID, puis se terminer. Exemple (vos obtiendrez bien sûr d'autres PIDs ; le $ représente l'invite de commande du shell) :

```
% ./question1
Je suis le parent, PID 15825
Enfant : PID 15826
Enfant : PID 15827
Enfant : PID 15828
Enfant : PID 15829
Enfant : PID 15830
$
```

2. En utilisant la fonction `system()`, écrivez un programme permettant d'ouvrir avec le programme `emacs` un fichier dont le nom a été passé en argument d'entrée de l'exécutable et qui rend ensuite la main à l'utilisateur. Exemple :

```
$ ./question2 question2.c
(le fichier question2.c s'ouvre dans emacs)
$
```

3. Écrivez un programme C équivalent à la commande shell `who & ps & ls -l` *sans utiliser* `system()` (les commandes séparées par « & » s'exécutent en parallèle).
   Exemple (la sortie que vous obtiendrez sera différente) :

```
$ ./question3
mandor   tty7         2008-10-05 13:23 (:0)
  PID TTY          TIME CMD
total 4
 9612 pts/4    00:00:00 zsh
drwxr-xr-x 2 mandor mandor 4096 2008-10-03 10:43 sujet
 9667 pts/4    00:00:00 evince
 9699 pts/4    00:00:02 emacs
[1] - Done                    who
$  9852 pts/4    00:00:00 sh
 9870 pts/4    00:00:00 ps
```

4. Écrivez un programme C équivalent à la commande shell `who; ps; ls -l` *sans utiliser* `system()` (les commandes séparées par « ; » s'exécutent successivement). Exemple :

```
$ ./question4
8-10-05 13:23 (:0)
  PID TTY          TIME CMD
 9612 pts/4    00:00:00 zsh
 9667 pts/4    00:00:00 evince
 9699 pts/4    00:00:03 emacs
 9852 pts/4    00:00:00 sh
 9894 pts/4    00:00:00 ps
total 4
drwxr-xr-x 2 mandor mandor 4096 2008-10-03 10:43 sujet
$
```

Exercice 2 – Traitement d'images multi-processus

Cet exercice a pour but le développement d'un programme multi-processus afin de mettre en oeuvre une fonction simple de traitement d'images. Afin de ne pas alourdir le code et pour des raisons évidentes de temps, nous nous limitons au traitement d'images en niveaux de gris. Le format retenu est le format PGM (Portable Grey Map). Ce n'est pas le plus connu mais il a l'avantage d'être facile à manipuler. Vous pouvez très facilement obtenir des fichiers *pgm* à partir d'autres formats en utilisant l'utilitaire en ligne de commande `convert` ou des outils plus sophistiqués de traitement d'images :
`convert fichier.jpg fichier.pgm`?
Pour visualiser une image au format *pgm*, vous pouvez utiliser le programme `eog` (`eog fichier.pgm &` en ligne de commande).

Si on ouvre une image au format *pgm* avec son éditeur de texte préféré, on constate que ce format de fichier contient un en-tête de type ASCII composé de trois lignes contenant respectivement :

1. un code de format **P5** correspondant au format *pgm* ;
2. deux entiers séparés par un espace et correspondant à la largeur et la hauteur de l'image ;
3. un entier représentant le nombre de niveaux de gris utilisé pour l'image (si ce nombre est, par exemple, 255, cela signifie que les niveaux de gris sont codés dans l'intervalle $[0, 255]$).

A la suite de cet en-tête, on trouve le buffer binaire correspondant aux pixels de l'image rangés par lignes.

Code fourni

Les fichiers suivants vous sont fournis :

— `image.h` : il contient la définition du type structuré `image_t`? ainsi que les prototypes des fonctions utilisés pour manipuler les images : création, allocation, copie, destruction, chargement, sauvegarde ;

— `image.c` : il contient l'implémentation des fonctions déclarées dans le `.h`;
— `transf_image.h` : **A compléter**, il contient le prototype des fonctions de transformation d'images;
— `transf_image.c` : **A compléter**, il contient l'implémentation des fonctions déclarées dans le `.h`;
— `bruitage.c` : **A compléter**, il contient un main permettant de charger et de sauvegarder des images;
— `Makefile` : un fichier permettant de compiler les différentes questions séparément en utilisant `make`;
— `images` : ce répertoire contient un ensemble d'images au format *pgm* qui pourront être utilisées pour vos tests.

Les images étant rangées en mémoire sous la forme d'un vecteur colonne, la macro `VAL(img,i,j)` définie dans le fichier image.h permet d'accéder simplement au pixel (i,j) de l'image pointée par `img`.

Bruitage simple

1. **Bruitage d'une image**
   Cette question a pour but d'écrire une fonction permettant de bruiter une image.
   Ajoutez dans `transf_image.c` la fonction `bruiter_image()` dont le prototype vous est donné et qui bruite une image existante . Le principe du bruitage est simple : pour chaque pixel de l'image, une valeur aléatoire est tirée dans l'intervalle $[0, 100]$. Si cette valeur est inférieure au pourcentage fournit comme argument d'entrée de la fonction, le pixel prend une valeur aléatoire dans l'intervalle de niveau de gris de l'image.
2. Ajoutez l'instruction nécessaire au fichier `bruitage.c` afin de pouvoir tester la fonction de bruitage.

Bruitage d'une zone de l'image

3. Dans un premier temps, créez une fonction `bruit_image_zone()` prenant comme paramètres d'entrée un pointeur sur l'image destination, un entier non signé représentant le pourcentage de bruit, quatre entiers non signés représentant respectivement un numéro de colonne de départ, un numéro de ligne de départ, le nombre de colonnes de pixels à bruiter à partir de la colonne de départ et le nombre de lignes de pixels à bruiter à partir de la ligne de départ. Cette fonction ne bruite que les pixels de la zone dont les paramètres sont passés à la fonction.
   Créez ensuite un programme `bruitage_zone.c` avec un main permettant de tester la nouvelle fonction de bruitage en supposant qu'on commence le bruitage au pixel de 50ème colonne et de la 80ème ligne et qu'on bruite sur 100 colonnes et 100 lignes. (Vous pouvez recopier et modifier le code de `bruitage.c`)

Bruitage multi-processus

Cette série de questions a pour but d'obtenir une version multi-processus du programme de bruitage : on va réaliser le bruitage de l'image en découpant celle-ci en $n \times m$ zones rectangulaires et en faisant réaliser le bruitage de chaque zone par un processus différent.

Pour que ces processus puissent partager les données de l'image, nous allons créer une projection mémoire partagée (voir cours 5) où nous stockerons une copie de l'image.

Rermarque : dans un cas réel, il serait à la fois plus simple (pas besoin de mémoire partagée) et plus efficace d'utiliser des *threads* plutôt que des processus distincts pour ce type de traitement.

Pour cela, nous allons procéder de la facon suivante :

— Créer une projection mémoire (`mmap`) anonyme partagée (voir exemple `mmap-ipc.c` du cours 5), d'une taille suffisante pour stocker l'ensemble des données de l'image ;
— Copier l'ensemble des données de l'image vers cette zone ;
— Pour chaque zone de l'image, créer un processus enfant avec `fork()` et lui faire bruiter la zone correspondante de l'image stockée dans la projection mémoire anonyme (avec la fonction écrite dans la question 3) puis se terminer.
— Le parent attend la terminaison de tous les enfants, puis sauvegarde l'image stockée dans la projection mémoire anonyme sur le disque.

4. Créez une fonction `dupliquer_vers_mmap_anon` dans le fichier `image.c`, qui a la même signature que `dupliquer_image` et un comportement similaire sauf qu'il stocke l'image et ses données[2] dans une projection `mmap` anonyme partagée et non dans des espaces alloués par `malloc`.
5. Créez un programme `bruitage_multiprocess.c` (vous pouvez partir de `bruitage.c`) utilisant les fonctions `dupliquer_vers_mmap_anon`, `bruit_image_zone`, `fork` et `wait` pour réaliser le bruitage multiprocessus de l'image divisée en 4 zones (avec donc 4 processus).
6. Que pouvez-vous dire des performances respectives du programme monoprocessus et multi-processus ? Répondez en commentaire dans le code de `bruitage_multiprocess.c`. (Si vous peinez à observer une différence, utilisez une grande image, par exemple `12vaches.pgm`)

7. **BONUS**
   Dans le programme `bruitage_multiprocess.c`, écrivez et installez un gestionnaire de signaux pour le signal `SIGUSR1` qui affiche un message "J'ai recu un signal du processus <pid du process émetteur>", et faites envoyer ce même signal par les processus enfants à leur parent juste avant qu'ils commencent le bruitage.

2. Attention, il y a deux choses à copier : la structure `image_t` et le buffer contenant les données de l'image. On souhaite que ces deux choses soient stockées dans l'espace réservé par `mmap`, de facon contigue. Prenez le en compte pour demander une projection mémoire de taille suffisante.