Projet de programmation C.

Julien Allali : allali@univ-mlv.fr

Gestionnaire de fichiers sous ncurse avec plugins.

1 Présentation.

Le but de ce projet est d'écrire un gestionnaire de fichiers à l'aide de la librairie cdk, surcouche de la librairie ncurses. Les fonctionnalités du gestionnaire seront écrites sous forme de plugin qui seront chargés dynamiquement au lancement du programme.

2 Téléchargement du projet et documentation.

Vous devez tout d'abord télécharger le fichier *projet_programmation_c.tgz* à l'adresse www-igm.univ-mlv.fr/ allali/ dans la section "Projet de programmation c".

Installation de l'archive: installer l'archive en tapant:

```
tar -zxvf projet_programmation_c.tgz
```

ceci doit créer un répertoire projet_programmation_c dans lequel vous aurez:

- un répertoire cdk4.8.2 : la librairie cdk avec des exemples.
- un répertoire dlsample : un exemple de chargement dynamique de librairie.
- un répertoire mc_like : là où vous devez mettre votre projet.

Compilation:

Dans le répertoire cdk4.8.2 entrez la commande "make", cela va compiler la librairie cdk.
Dans le répertoire "lib/" doit se trouver la librairie cdk.
Dans le répertoire "examples/" vous trouverez un ensembles d'exemple.
Dans le répertoire "man/", les man de la librairie.
Dans le répertoire dlsample entrez la commande "make" puis "make plugins", vous avez alors un exemple de chargement dynamique de librairie.

Utilisation de cdk4.8.2

Dans le répertoire "cdk4.8.2/examples/" vous trouverez un exemple d'interface pour ce projet : le fichier "menu_ex.c".
Pour compiler un fichier utilisant la cdk4.8.2:
Le fichier source utilisant cdk doit contenir *#include<cdk.h>*.
Pour le compiler :

```
gcc -Wall source.c -I<REP_CDK_H> -L<REP_CDK_L> -lcdk -lncurses -lm
```

Où `REP_CDK_H` est le répertoire où se trouve le fichier *cdk.h* et `REP_CDK_L` est le répertoire où se trouve le fichier *libcdk.a*.
Par exemple: Dans le répertoire "examples/":

```
gcc -Wall menu_ex.c -I../include/ -L../lib/ -lcdk -lncurses -lm
```

Librairie et chargement dynamique: Dans le répertoire dlsample vous trouverez un exemple de chargement dynamique.
Le programme "chargeur" prend un répertoire en argument, puis il va scanner celui-ci dans le but de trouver tous les fichiers dont le nom est "*plug ....so*". Puis "chargeur" ouvre ces fichiers en tant que librairie (dlopen()) et tente de charger des fonctions de la librarie (dlsym()).
Une librarie dynamique est un objet contenant des symboles (fonctions, variables ...) et le code binaire associé à ces symboles.
Exemple:
vous avez le fichier toto.c suivant:

```
#include<stdio.h>
void print_toto(void)
{
   printf(``toto\n'');
}
```

Pour faire une librairie contenant la fonction print_toto on fait:

```
gcc -shared toto.c -o libtoto.so
```

Alors dans un autre programme test.c on peut faire:

```
#include<stdio.h>
#include<dlfcn.h>

typedef void (* prototype_void_void)(void);
int main(void)
{
  void *librarie;
  prototype_void_void la_fonction;
  librarie=dlopen(``libtoto.so'',RTLD_LAZY);
  if (librarie!=NULL)
    {
      la_fonction = dlsym(librarie, ``print_toto'');
      if (la_fonction != NULL)
        {
           (*la_fonction)();
        }
      else
        {
          printf(``le symbole print_toto n'existe''
                 ``pas dans la librairie toto\n'');
```

```
      }
    }
  else
    {
      printf(``Erreur lors du chargement de la lib.\n'');
    }
  return 1;
}
```

que l'on compile avec : `gcc test.c -ldl`
Si vous executez le binaire "a.out" depuis un autre repertoire (ex : rep/a.out) vous aurez l'erreur "Erreur lors du chargement de la lib.", c'est normal. Lorsque l'on charge une librairie le système va interpréter le paramétre tout d'abord comme un chemin relatif ou absolu puis s'il ne trouve pas la librarie il va la chercher dans des répertoires par defaut ainsi que dans les répertoires spécifiés dans la variable d'environnement LD_LIBRARY_PATH. Pour spécifier votre propre répertoire de librairie, utilisez la variable LD_LIBRARY_PATH en faisant: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:rep` où rep est le répertoire où se trouve le fichier libtoto.so.
Ainsi durant l'exécution de a.out, en faisant dlsym, on va charger en mémoire le code binaire de print_toto et récupérer un pointeur sur le début de cette fonction. Puis en faisant (*la_fonction)() on appelle la fonction à l'adresse précédemment chargée à savoir print_toto.
L'une des utilisations de ce mécanisme est la mise à jour simplifiée. En effet si vous modifiez le code de toto.c pour afficher tata et que vous recompilez la librairie libtoto.so, alors sans recompiler test.c, en exécutant a.out, vous appellerez la nouvelle fonction print_toto.
Rq: Pour compiler une librairie vous pouvez aussi donner une liste d'objets (.o), les symboles de ceux-ci seront regroupés dans la librairie.

les man.

Pour pouvoir utiliser les man de cdk4.8.2 vous devez:

- soit utiliser la commande : `man -M<rep> ...`
- soit utiliser la variable d'environnement `MANPATH`: `export MANPATH=<rep>:/usr/man`

Le répertoire rep correspond au répertoire : ~ /.../projet_programmation_c/cdk4.8.2/man/

3 Le projet.

Le but de ce projet est d'écrire un gestionnaire de fichiers dont les fonctionnalités seront des plugins chargés dynamiquement.

3.1 Les plugins.

Les plugins seront des libraries chargées dynamiquement par l'interface. Nous désignerons par plugin une librairie. Chaque plugin devra comporter les fonctions:

```
int    get_plugin_class(void);
char  *get_plugin_name(void);
void  *get_plugin_private_data(void);
void   display_plugin_information(CDKSCREEN *cdkscreen,void *private_data);
char **get_plugin_menu_entry(void);
```

La fonction get_plugin_class renvoie la classe à laquelle appartient le plugin, ceci sera expliqué plus bas.
La fonction get_plugin_name renvoie le nom interne du plugin.
La fonction get_plugin_private_data renvoie un pointeur sur une structure de données propre au plugin. Ceci permet au plugin de véhiculer des informations, la structure étant systèmatiquement transmise par l'interface au plugin.
La fonction display_plugin_information permet au plugin d'afficher des informations sur celui-ci. (via la fonction popupLabel par exemple (man misc)).
La fonction get_plugin_menu_entry renvoie un tableau d'entrées de menu terminé par NULL. Par exemple, le plugin PasteCutCopy renverra la table {"Paste","Copy","Cut",NULL}.
A chaque chaîne sera associée une entrée dans le menu de la classe.
La classe sert à déterminer pour quels types d'actions est destiné le plugin. A chaque classe sera associée une fonction dont le prototype sera spécifié plus bas.
L'identifiant de classe est défini dans le fichier "general_plugin_header.h"

3.1.1 La classe VISUAL.

Un plugin de cette classe aura pour objectif la mise en forme des listes de fichiers. Celui-ci devra retourner la valeur VISUAL_PLUGIN_CLASS pour la fonction get_plugin_class. VISUAL_PLUGIN_CLASS est définie dans le fichier "general_plugin_header.h". De plus, il devra avoir la fonction:

```
char **plugin_fonction(void *private_plugin_data, // données privées.
                        char **file_liste,// liste des noms de fichiers.
                        char **file_items,// liste des items de la ScrollListe.
                        char *path,// Répertoire où se trouve la liste.
                       );
```

Cette fonction renvoie la liste des items modifiés.
Cette classe permet entre autre de coloriser les fichiers selon leurs types, afficher la taille des fichiers à coté de leurs nom etc. . . .
Rq: Seuls les éléments file_items doivent être modifiés.

3.1.2 La classe ACTION.

Cette classe définie les plugins faisant une action spécifique sur un fichier comme par exemple l'afficher. Le plugin devra retourner l'entier ACTION_PLUGIN_CLASS pour la fonction get_plugin_class. ACTION_PLUGIN_CLASS est définie dans le fichier "general_plugin_header.h". De plus, il devra avoir la fonction:

```
int    plugin_fonction(void *private_plugin_data,
                        char *menu_entry,// entrée dans le menu.
                        char *file, // fichier.
                        char *path  // Repertoire dans lequel se trouve le fichier
                       );
```

Le champ menu_entry est l'entrée dans les menus de l'interface par laquelle le plugin a été appelé. La valeur de retour est REFRESH ou DONTREFRESH selon que l'action effectuée par le plugin nécessite le ré-affichage de la liste (valeurs définies dans "general_plugin_header.h").

3.1.3 La classe EDITING.

Dans cette classe se trouvent les plugins manupilant les fichiers des deux listes, par exemple : déplacement d'une liste à l'autre, couper-copier-coller, comparaison entre les fichiers selectionnés dans les deux listes etc...Le plugin devra retourner l'entier EDITING_PLUGIN_CLASS pour la fonction get_plugin_class. EDITING_PLUGIN_CLASS est définie dans le fichier "general_plugin_header.h".
De plus, il devra avoir la fonction:

```
int    plugin_fonction(void *private_plugin_data,
                        char *menu_entry,// entrée dans le menu.
                        char **file_list1, // fichiers de la liste 1.
                        char **file_list2, // fichiers de la liste 2.
                        int selected_file_list1, // fichiers selectionné
                                                 // dans la liste 1.
                        int selected_file_list2, // fichiers selectionné
                                                 // dans la liste 2.
                        char *path_list1, // Répertoire de la liste 1.
                        char *path_list2  // Répertoire de la liste 2.
                       );
```

La valeur de retour est REFRESH ou DONTREFRESH selon que l'action effectuée par le plugin nécessite le ré-affichage de la liste.

3.2 L'interface.

L'interface est le gestionnaire, c'est elle qui va charger les plugins, créer les menus et les entrées pour chaque plugin.
Cette interface doit être semblable à celle de "projet_programmation_c/cdk4.8.2/examples/menu_ex.c", elle comporte :

- 2 listes donnant le contenu des 2 répertoires courants de travaux.
- Une barre de menu(s) contenant une entrée pour chaque classe de plugin.
- Dans chaque menu: une (ou plusieurs) entrée(s) pour chaque plugin.

- Un menu “Plugin Info” comportant une entrée pour chaque plugin et permettant d'afficher les informations relatives au plugin (appel de la fonction display_plugin_information).
- Un menu visuel (ou une fenêtre comme dans selection_ex.c) permettant d'activer ou désactiver un pluggin visuel.
- Une zone de texte informative en bas donnant par exemple les actions claviers possibles en cours (ie. différentes si dans une liste ou dans un menu).

4 Instructions, conseils et rapport.

Il vous est demandé d'implémenter l'interface ainsi qu'au moins 2 plugins visuels, 1 plugin d'édition et 1 plugin d'action.
Votre projet devra être compilé à l'aide d'un makefile (aidez vous de celui de cdk4.8.2 et de dlsample). Regardez bien les exemples (dlsample et menu_ex.c) avant de commencer. Ce projet pose plus de problèmes de conception que des problèmes de programmation.
Ne pas hésiter à envoyer un mail en cas de problème sur le groupe de discussion: http://www.egroups.com/group/projet_prog_c
Ce projet devra être fait par groupe de deux ou par groupe de quatre auquel cas l'interface doit être réellement compléte (avec options) et vous devrez avoir au moins 10 plugins.
De plus, les plugins que vous développerez devront être compatibles avec au moins un autre groupe (à mettre dans le rapport). Le rapport devra être écrit en Latex, être clair et <u>concis</u> (le sujet du projet est écrit en latex : le prendre pour exemple, voir aussi la section Stage Unix IR1 de la page www-igm.univ-mlv.fr/ allali/).
Vous êtes libre de retoucher la librarie cdk (pour des problèmes de binding key entre autre).
Vous rendrez une archive projet_programmation_c.tgz qui contiendra entre autre le répertoire mc_like dans lequel se trouvera votre projet avec un fichier README contenant des informations sur votre projet (ex: ligne de commandes etc...) et un fichier INSTALL donnant la marche à suivre pour l'installer.