TP de Systèmes d'exploitation
François Cayre / francois.cayre@grenoble-inp.fr

# Greffons

Notions abordées : chargeur dynamique , greffons, API.

Fonctions utilisées : `dlopen(3)`, `dlsym(3)`, `dlclose(3)`.

## 1 Applications extensibles

L'utilisation effective d'une application est souvent favorisée en prévoyant qu'elle puisse être étendue par du code provenant de tiers. Par exemple, une application de traitement audio deviendra d'autant plus populaire que ses utilisateurs pourront s'échanger des filtres d'effet adaptés à leurs besoins probablement davantage variés que ce que les concepteurs de l'application auront pu prévoir au début. Une application de calcul scientifique peut souhaiter proposer plusieurs manières de réaliser un calcul (voir le TP suivant). Un navigateur *web* peut devoir être étendu avec, par exemple, un greffon bloqueur de publicité, ou un autre bloqueur de scripts (car JavaScript *est* le Mal).

La tradition dans le monde Unix est de s'échanger du code source et de vendre le service associé au logiciel. Les gens ayant un autre modèle économique vont préférer distribuer des versions binaires compilées de leur code. Dans les deux cas, on va vouloir être capable d'étendre une application à l'aide de greffons (*plugins*) : des bouts de code qui vont pouvoir être exécutés *dans* l'application hôte.

Cela nécessite que l'application puisse être *programmée* : elle doit *publier* la liste des fonctions qu'un greffon va pouvoir utiliser. Par exemple, une application de calcul scientifique devra publier, entre autres, la fonction permettant de créer une matrice en mémoire. On appelle API (*Application Programming Interface*) cet ensemble de fonctions permettant de programmer l'application. Tout greffon doit donc s'appuyer sur l'API pour pouvoir être exécuté par l'application hôte.

Le C n'est assurément pas le meilleur langage pour développer une application extensible par greffons, mais nous pourrons néanmoins donner les grandes lignes de la conception d'une telle application en utilisant le chargeur dynamique du système d'exploitation. La syntaxe du C nous limitera notamment pour implanter des mécanismes standard d'enregistrement des greffons dans l'application hôte.

Le chargeur dynamique permet d'appeler une fonction située dans une librairie, et dont on connaît le prototype. Ainsi, on pourrait l'utiliser pour exécuter la fonction `cos(3)` en allant la chercher explicitement *par son nom* dans la librairie mathématique `libm.so`. Voir l'exemple dans la page du manuel.

## 2 Donc il faut une application…

Nous prendrons le prétexte fallacieux d'une application de calcul scientifique (ressemblant de très loin à Matlab) pour illustrer ce TP. Disons déjà que vous n'aurez qu'à coder (1) le mécanisme de chargement d'un greffon, et (2) au moins

un greffon (multiplication de deux matrices), qui sera utilisé dans le TP suivant. Libre à vous d'en coder d'autres. Mais nous vous fournissons tout le reste, c'est à dire un embryon de syntaxe dans un interpréteur, et des greffons permettant de procéder à quelques manipulations de base.

Nous donnons maintenant les principaux éléments de la syntaxe rudimentaire implantée dans l'interpréteur. Nous vous recommandons d'essayer les exemples suivants pour vous familiariser avec. Notez que l'interpréteur affiche le temps de calcul (en ms) de la dernière commande. Dans le répertoire du code fourni, on démarre l'interpréteur avec :

```
$ ./appli/shell
```

Les expressions se terminent par un point-virgule.

On déclare une variable `toto` comme suit :

```
var toto = 42 ;
```

On affiche le contenu d'une variable en entrant son nom :

```
toto ;
```

Une fois `toto` déclarée, son type peut changer. Ici pour devenir une matrice de 4 lignes et 6 colonnes contenant des entiers pseudo-aléatoires entre 0 et 9 :

```
toto = rand(4,6,"int10") ;
```

On peut déclarer plusieurs variables à la suite :

```
var A = rand(10,40,"int10"), B = rand(40,13,"int10"), C = A*B ;
```

Nous vous fournissons le greffon permettant de tester l'égalité entre deux variables `A` et `B` :

```
A==B ;
```

On peut appeler une fonction quelconque `albert` avec :

```
albert(A,B) ;
```

En réalité, l'interpréteur transforme l'expression `A*B` en `mult(A,B)`, et l'expression `A==B` en `equals(A,B)`.

# 3 Travail à réaliser

Outre son intérêt intrinsèque manifeste, ce TP est la base du suivant. Il est donc *indispensable* de le terminer pour pouvoir continuer l'aventure.

## *3.1 Implantation du système de greffons*

Notre système de greffon ne peut pas être plus simple : chaque appel à une fonction `albert` devra déclencher (1) la recherche d'un fichier de greffon `op_albert.so` (une librairie) dans le répertoire `plugins`, et (2) l'exécution de la fonction `op_albert` de type `object(*)(shell,object)` située dans ce fichier.

Cette étape est à réaliser dans le fichier `appli/eval.c`.

## *3.2 Écriture d'un greffon*

L'exemple ci-dessus (le produit de deux matrices `C = A*B`) utilise le greffon que vous avez à terminer : `op_mult.so` peut calculer les autres produits (nombres et vecteurs), mais pas celui de deux matrices. Cette étape est à réaliser dans le fichier `plugins/op_mult.c`. Cette multiplication sera utilisée dans le TP suivant.

Si vous vous ennuyez, utilisez votre imagination pour écrire d'autres greffons ! Nous mettons à votre disposition des greffons pour lire et écrire des images au format PGM.

Bon travail !