Jean-Paul-Rigault, Stéphane Lavirotte
Polytech'Nice – Sophia
SI3
2017-2018

TD n° 2
Bibliothèques Statiques et Dynamiques

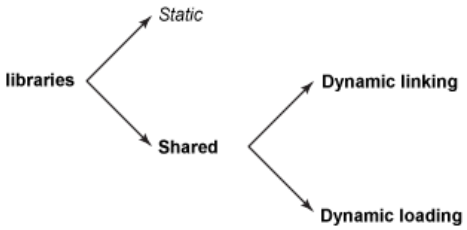
1 Objectif

1.1 Différents types de bibliothèques binaires

Cette séance de TD introduit les différents types de bibliothèques binaires[1], qui sont des collections de fichiers-objets (`.o`) regroupés en une seule entité. On laisse alors à l'éditeur de liens le soin de trouver, dans la bibliothèque, les modules contenant les fonctions dont il a besoin. C'est la résolution des références externes, évoquée en cours.

GNU/Linux (de même que la plupart des Unix modernes[2]) connait deux types de bibliothèques, statiques (extension `.a`) et partagées (`.so`, comme *shared object*), et trois manières d'éditer les liens :

- statiquement,
- dynamiquement avec chargement automatique,
- dynamiquement avec chargement explicite.



Ce TD explore ces trois possibilités dans les 3 sections suivantes.

1.2 Code fourni

Pour faire ce TD, nous n'allons pas avoir le temps de produire le code de l'application et des bibliothèques.

`http://trolen.polytech.unice.fr/cours/progsys/td02/td02_distrib.zip`

Nous utiliserons donc une implémentation de divers algorithmes de tri (tri à bulles, tri par insertion, …) et un programme générique qui utilisera ces différentes implémentations d'algorithmes de tri.

Si vous utilisez Visual Studio Code, une fois le dossier dézippé, ajoutez-le à la liste des dossiers de l'espace de travail : menu « Fichier / Ajouter un dossier à l'espace de travail… ».

Exercice n°1:

Etudiez rapidement le code source qui vous est fourni pour en comprendre la structure. Donnez un résumé de ce que vous avez compris à ce code source.

Le `Makefile`, tel qu'il est fourni, construit autant de programmes qu'il y a d'algorithmes de tri implémentés. Chaque programme peut prendre des options permettant de spécifier si on affiche plus de messages (le tableau des nombres à trier et une fois trié) et combien de nombres on souhaite générer aléatoirement. On a veillé pour faire ces tests à faire un tirage aléatoire qui soit toujours identique (utilisation d'une graine identique à chaque lancement).

Nous allons donc voir dans ce TD, les différents types d'utilisation des bibliothèques.

2 Bibliothèque statique, édition de liens statique

2.1 Introduction aux bibliothèques statiques

Lors de la compilation d'un programme C, chaque fichier-source (`.c`) est compilé séparément, produisant un fichier-objet (`.o`). Puis l'éditeur de liens prend tous les fichiers-objets et les regroupe en un seul fichier binaire exécutable. Cette phase d'édition de liens est cruciale. Elle permet d'opérationnaliser la communication entre les différents fichiers constituant le programme : certaines variables sont définies (allouées et initialisées) dans un fichier et utilisées

[1] On dit parfois « librairie », qui est un anglicisme, ou même librairie d'objets (object library). Objet est bien entendu à prendre, ici, dans le sens de fichier-objet (`.o`)
[2] De nombreux systèmes connaissent également ces distinctions. Sous MS Windows, une bibliothèque dynamique s'appelle une dll (Dynamically Loadable Library), la version statique étant un fichier-objet `.lib`

dans d'autres (variables externes) ; de même, certaines fonctions peuvent avoir leur corps défini dans un fichier alors qu'elles sont invoquées dans d'autres (fonctions externes). On parle de symboles externes, ou de références externes. L'établissement des correspondances entre définitions et utilisations de ces symboles constitue la résolution des références, le rôle principal de l'éditeur de liens. Sous Unix, l'éditeur de liens est une commande nommé `ld` (comme *loader*, nom assez mal choisi en fait), mais il est rare que vous ayez à l'utiliser directement. Ainsi la commande de compilation C (`cc` ou `gcc`) invoque-t-elle directement l'éditeur de liens si on lui donne des `.o` comme arguments :

```
gcc -o prog main.o prog1.o prog2.o prog3.o
```

Ceci construit l'exécutable `prog` en éditant les liens entre les quatre fichiers `.o` mentionnés.

Exercice n°2:

Lancez la commande `make` dans l'archive que vous avez récupérée. Examinez à l'aide de la commande `ldd` avec un des programmes générés pour savoir s'il utilise des bibliothèques ou non. Si oui, quelles sont les bibliothèques utilisées ?

Exercice n°3:

Comment rendre votre programme complètement indépendant des bibliothèques qu'il pourrait utiliser ? Modifiez le `Makefile` pour générer des programmes qui n'utiliserait pas de bibliothèque du tout. Ces programmes devront être nommés : `tri_xxx-staticExe.exe`.

2.2 Création de bibliothèques statiques (archives)

Cependant, la discipline de programmation modulaire fait en sorte que, pour des applications conséquentes, le nombre de fichiers `.o` à lier ensemble peut devenir considérable, et qu'il peut même devenir humainement impossible de déterminer ceux qui sont réellement utilisés.

La solution est donc de regrouper plusieurs .o dans un fichier unique, une bibliothèque[3]. La commande pour ce faire est nommée `ar` :

```
ar -r libprog.a prog1.o prog2.o prog3.o
```

Ceci crée la bibliothèque `libprog.a` (il est conventionnel de faire commencer le nom d'une bibliothèque par le préfixe `lib`) et lui donne les trois fichiers `.o` désignés comme contenu. L'option `-r` (replacement) permet de créer la bibliothèque avec le contenu indiqué, mais aussi de remplacer dans une bibliothèque existante certains des `.o` par de nouvelles versions.

Il est pratique de doter la bibliothèque d'un index des références qui y sont définies ou utilisées, afin de faciliter la tâche de l'éditeur de liens. Ceci est le rôle de l'utilitaire `ranlib`, dont l'utilisation est très simple :

```
ranlib libprog.a
```

La création d'une bibliothèque statique est donc en général constituée d'un appel à `ar`, suivi d'un appel à `ranlib`.

2.3 Édition de liens avec une bibliothèque statique

Une fois la bibliothèque ainsi créée, on peut l'utiliser dans une commande d'édition de liens. Par exemple :

```
gcc -o prog main.o libprog.a
```

si le fichier `libprog.a` est dans le répertoire courant ; ou bien

```
gcc -o prog main.o -lprog
```

si `libprog.a` est dans le chemin de recherche des bibliothèques, un ensemble de répertoires (typiquement `/lib`, `/usr/lib`, `/usr/local/lib`, etc.) prédéfini à la configuration du système (l'option `-lxxx` recherche une bibliothèque nommé `libxxx.a` dans le chemin en question) ; ou encore

[3] Unix appelle cette bibliothèque une archive, d'où l'extension `.a` et le nom de la commande associée, `ar`.

```
gcc -o prog main.o -L/home/user/lib -lprog
```

si on souhaite ajouter (option `-L`) le répertoire `/home/user/lib` en tête du chemin de recherche susmentionné. Ce type de bibliothèques est dit statique, comme l'édition de liens associée. On peut donc dire que le mécanisme est doublement statique : d'une part, les références sont résolues statiquement ; d'autre part, le code des fichiers `.o` sélectionnés dans la bibliothèque est copié directement dans l'exécutable.

2.4 Fichiers-objets et bibliothèques

Il y a une différence fondamentale entre l'édition de liens directement avec des `.o`

```
gcc -o prog main.o prog1.o prog2.o prog3.o
```

et l'édition de liens avec une bibliothèque

```
gcc -o prog main.o libprog.a
```

Dans le premier cas, tous les `.o` sont en quelque sorte concaténés et leur somme forme l'exécutable. Dans le second cas, l'éditeur de liens extrait de la bibliothèque uniquement les `.o` qui sont réellement utilisés par le programme. L'utilisation d'une bibliothèque peut donc contribuer à réduire la taille des exécutables. Une bibliothèque peut en effet avoir un contenu vaste, qui dépasse les besoins d'une application particulière.

2.5 Commandes de manipulation de bibliothèques statiques

La commande `ar` a de nombreuses options. En particulier, la commande `ar -t` permet de connaitre le contenu d'une bibliothèque (la liste des `.o` qui la composent).

La commande `nm` (*namelist*) appliquée à une bibliothèque donne la liste des symboles qui y sont définis ou utilisés. On se reportera avantageusement aux pages de manuel correspondantes (`man ar`, `man nm`, et pendant que vous y êtes, `man ranlib`).

Exercice n°4:

Votre première mission est de modifier le fichier `Makefile` fourni afin de créer quatre bibliothèques, une pour chaque tri, que je propose de nommer de la manière suivante : `libTri_xxx-staticLib.a`, où `xxx` désigne le tri (`xxx` est `bubble`, `insertion`, `merge` ou `quick`). Ces noms, de même que tous ceux qui suivent, sont ceux qui sont définis dans le fichier `Makefile` fourni ; encore une fois respectez-les.

Chaque bibliothèque contiendra le code correspondant au tri (`xxx.o`) et le code inutilisé (`unused.o`)

Utilisez ces quatre bibliothèques pour générer quatre exécutables nommés `tri_xxx-staticLib.exe`. Puis traitez les points suivants :

- Vérifiez que les résultats d'exécution de ces fichiers sont identiques à ceux des fichiers générés sans bibliothèque (`tri_xxx-staticExe.exe`).
- Comparez également leur taille à celle des exécutables précédemment générés (`basicExe` et `staticExe`). Vous utiliserez la commande `ls -l` pour cela. Ces tailles sont-elles fondamentalement différentes ? La commande Shell `size` vous permettra de déterminer dans quel(s) segment(s) se fait l'éventuel gain de taille. Dans le résultat de cette commande, *text* désigne la taille du segment de texte (les instructions), *data* celle du segment de données initialisées, et *bss* celle du segment de données non initialisées. Donc, la taille totale du segment de données est *data+bss* et correspond à l'ensemble des variables externes et statiques (à l'exclusion évidente de ce qui sera alloué lors de l'exécution par `malloc()`, `sbrk()`, etc.).
- Vérifiez que le code des fonctions de `unused.c` (les fonctions définies dans `unused.c` sont `foo` et `bar`) n'a pas été inclus dans les exécutables dans le cas de l'utilisation des bibliothèques statiques (contrairement à

ce qui se passe pour les exécutables générés sans bibliothèque). Pour cela, vous pouvez utiliser, par exemple, une commande Shell comme :

```
nm tri_xxx[-staticExe].exe
```

3 Bibliothèque dynamique, édition de liens statique

3.1 Motivation pour les bibliothèques dynamiques

Les bibliothèques statiques ont l'avantage de proposer une intégration sélective des fichiers-objets à lier à l'exécutable. Cependant elles souffrent de plusieurs inconvénients :

- La taille des fichiers binaires exécutables est importante, à cause de la copie du code.
- La taille de l'espace mémoire des processus est également importante. Aucun partage de mémoire n'est possible entre des processus exécutant des programmes différents, mais utilisant la même bibliothèque statique. Certes, il s'agit de mémoire « virtuelle », mais il faudra quand même bien la ranger quelque part !
- Si la bibliothèque change de version, il faut refaire l'édition de liens de tous les exécutables qui lui sont liés, et ceci même si l'interface de la bibliothèque (prototypes des fonctions, types des variables externes) n'a pas changé.

Pour pallier ces inconvénients, on a inventé il y a longtemps (pratiquement depuis la mémoire virtuelle) la notion de **bibliothèque partagée**, appelée aussi sous Unix (fichier-)objet partagé (d'où son extension `.so`, comme *shared object*). Le code d'une telle bibliothèque n'est pas recopié dans le binaire exécutable, mais partagé (sur disque comme en mémoire) entre tous les exécutables qui l'utilisent. Dès qu'un processus s'exécute qui nécessite la bibliothèque, cette dernière est chargée en mémoire (si elle n'y est pas déjà). D'où l'autre nom de bibliothèque (à chargement) dynamique (*Dynamic Loadable Library* de MS Windows, `.dll`). Le point clé est de savoir comment sont résolues les références dans le cas des bibliothèques partagées : elles peuvent l'être statiquement (à l'édition de liens, avant l'exécution), ou dynamiquement (lors de l'exécution). Nous traitons le premier cas dans cette section, et le second dans la section suivante.

3.2 Création d'une bibliothèque dynamique

Alors que pour créer une bibliothèque statique, il suffit de produire des fichiers-objets sans option particulière, ce n'est pas le cas ici. Pour créer une bibliothèque dynamique, il faut d'abord compiler chaque fichier-source avec l'option `-fPIC` ou `-fpic`, comme dans

```
gcc -c -fpic -std=c99 -Wall prog1.c
```

(Comme vous le savez déjà, l'option `-c` arrête le processus juste après la production du fichier `.o`, ici `prog1.o`.).

Puis la bibliothèque elle-même, disons `libprog.so`, est créée par une commande d'édition de liens spéciale :

```
gcc -shared -Wl,-soname,libprog.so -o libprog.so prog1.o prog2.o prog3.o
```

C'est tout ! Certes, la syntaxe des options apparait assez bizarre. Et oui, le nom de la bibliothèque figure deux fois ! Et non, il n'y a pas d'espaces dans l'argument de l'option `-W` ! Si vous voulez comprendre ces bizarreries, faites donc man `gcc`. Sinon, considérez cela comme une formule magique...

3.3 Édition de liens statique et chargement dynamique

Une fois la bibliothèque créée, on peut l'utiliser dans une commande d'édition de liens statique, de la même manière que s'il s'agissait d'une bibliothèque statique. C'est-à-dire

```
gcc -o prog main.o libprog.so
```

si le fichier `libprog.so` est dans le répertoire courant ; ou bien

```
gcc -o prog main.o -lprog
```

si `libprog.so` est dans le chemin de recherche des bibliothèques ; ou encore

```
gcc -o prog main.o -L/users/jpr/lib -lprog
```

si on souhaite ajouter (option `-L`) le répertoire /home/user /lib au chemin de recherche susmentionné. Les références sont alors résolues statiquement, et le fichier exécutable contient le résultat de cette résolution, mais pas le code correspondant. Celui-ci sera chargé lors de l'exécution.

3.4 Exécution d'un programme lié avec des bibliothèques dynamiques

Lorsqu'on exécute un programme lié à une bibliothèque statique, l'exécutable contient toute l'information. Ce n'est pas le cas avec une bibliothèque dynamique : il faut que l'on sache, lors du lancement du processus, où trouver le fichier `.so` pour pouvoir le charger.

La recherche de ce fichier se fait de plusieurs manières :

- le système connait un certain nombre de répertoires prédéfinis où chercher les `.so` (`/lib`, `/usr/lib`, `/usr/local/lib`, etc.) ;
- sous GNU/Linux, on peut configurer ce chemin de recherche globalement pour tous les utilisateurs du système (utilitaire `/etc/ldconfig` et fichier `/etc/ld.so.conf`) mais cela nécessite des droits d'administrateur (super-utilisateur) ;
- enfin, chaque utilisateur peut définir son propre chemin de recherche des bibliothèques à l'exécution grâce à la variable d'environnement LD_LIBRARY_PATH, qui est une suite de répertoires séparés par deux points. Par exemple

```
export LD_LIBRARY_PATH=.:~/lib:/perso/bizarre/lib
```

Les trois répertoires indiqués (dont le répertoire courant, désigné par le point `.`) seront examinés avant les répertoires système (ceux établis des deux premières manières).

Pour cet exercice, et même de manière générale, je vous suggère fortement de définir LD_LIBRARY_PATH de telle sorte que cette variable contienne au moins le répertoire courant (.). Par exemple vous pouvez placer la ligne suivante dans votre fichier `.bashrc` ou `.zshenv` (suivant l'interprète de commande que vous utilisez) :

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

afin de préserver la valeur que l'administrateur a établie par défaut.

3.5 Commandes de manipulation de bibliothèques dynamiques

La commande `nm`, déjà mentionnée, permet de connaitre les symboles définis et utilisés dans une bibliothèque dynamique (en fait `nm` fonctionne pour tout fichier-objet).

La commande `ldd` appliquée à un binaire exécutable liste les bibliothèques dynamiques dont cet exécutable dépend. Vous constaterez que, par défaut, tous les exécutables que vous avez produits jusqu'à présent (et d'ailleurs tous ceux à venir) dépendent d'un certain nombre de bibliothèques dynamiques prédéfinies, dont la bibliothèque standard C (`/lib/libc.so`) et le chargeur dynamique (`/lib/ld-linux.so`). Ce dernier a pour charge de gérer les bibliothèques dynamiques utilisées par l'exécutable.

Exercice n°5:

Faites exactement ce que vous avez fait dans l'exercice précédent, mais en utilisant des bibliothèques dynamiques au lieu de bibliothèques statiques. Vous nommerez ces bibliothèques dynamiques `libTri_xxx-dynamicLib.so`, où `xxx` est le nom de la stratégie et les exécutables `tri_xxx-dynamicLib.exe`.

Vous testerez bien entendu le bon fonctionnement des programmes créés en ajoutant une section de test dans le fichier `Makefile`.

TD n° 2
Bibliothèques Statiques et Dynamiques

Attention : Lors de l'exécution des programmes de test, vérifiez bien la valeur de votre variable d'environnement `LD_LIBRARY_PATH`.

4 Bibliothèque dynamique, édition de liens dynamique

4.1 Motivation pour le chargement dynamique explicite de bibliothèques

Dans l'utilisation des bibliothèques dynamiques de la section précédente, la résolution des références restait statique. Il y avait un lien très fort entre la bibliothèque et l'exécutable. On pourrait souhaiter que la résolution des références soit elle-même dynamique. Cela permettrait d'avoir plusieurs bibliothèques de fonctionnalités semblables, d'interfaces voisines voire identiques mais d'implémentations différentes. On aimerait alors être capable de choisir une de ces bibliothèques à l'exécution, voire de la changer dynamiquement.

Cet exemple n'est pas lancé au hasard, puisque nous somme précisément dans ce cas. Nous avons quatre bibliothèques (`libTri_xxx-dynamicLib.so`), implémentant la même interface, définie dans `sort.h`, mais de manières différentes. Le problème est que la résolution des références doit maintenant se faire à l'exécution. GNU/Linux et les Unix modernes en général, permettent ceci, mais le programmeur doit travailler ! C'est à lui de charger et de résoudre explicitement les références qui l'intéressent. Le système fournit pour cela une bibliothèque générique de chargement dynamique, dont le nom est `/lib/libdl.so`.

4.2 Édition de liens et chargement dynamique de bibliothèques

Pour utiliser cette facilité, on doit d'abord créer les bibliothèques dynamiques. La procédure est en tout point identique à celle décrite précédemment (section 3.2). Sous GNU/Linux (Unix), la même bibliothèque dynamique peut être chargée aussi bien automatiquement qu'explicitement, sans changement.

Ensuite il faut construire l'exécutable. Avec l'exemple utilisé précédemment, cela donnerait

```
gcc -o prog main.o -ldl
```

Notez que cette commande d'édition de liens ne comporte aucune indication des bibliothèques qui seront explicitement et dynamiquement chargées. L'exécutable produit n'en dépendra donc pas. La seule bibliothèque ici est celle de chargement générique, `libdl.so`. C'est donc au programmeur qu'il appartiendra de désigner, charger, et même résoudre les références. La bibliothèque `libdl.so` lui fournit pour cela quatre fonctions, dont le type est déclaré dans le fichier d'en-tête standard `<dlfcn.h>` (qu'il convient donc d'inclure) :

`void *dlopen(const char *filename, int flag);`
: Ouvre la bibliothèque dynamique de nom filename, et retourne un pointeur (`plib`) qui permettra de référencer cette bibliothèque dans `dlclose()` et `dlsym()`. Si la bibliothèque ne peut être ouverte, le pointeur retourné est NULL. Une bonne valeur pour flag est `RTLD_LAZY` (pour les curieux, voir `man dlopen` pour d'autres possibilités).
La bibliothèque est recherchée de la même manière qu'en 3.4, donc vous devez faire attention à `LD_LIBRARY_PATH` !

`void dlclose(void *plib);`
: Ferme la bibliothèque représentée par `plib` qui doit donc être le résultat d'un `dlopen()` précédent.

`void *dlsym(void *plib, const char *symbol_name);`
: Retourne un pointeur sur l'entité de nom `symbol_name` (ce peut être le nom d'une fonction ou d'une variable externe). Le type du pointeur retourné est `void *` (que pourrait-il être d'autre ?) ; il appartient donc au programmeur de connaitre le type exact de cette entité et d'effectuer le cast nécessaire.

`char *dlerror();`

Si la valeur de retour n'est pas le pointeur NULL, c'est un message indiquant la cause de l'erreur dans le dernier appel à l'une des quatre fonctions de `libdl.so`.

La page de man d'une quelconque de ces fonctions (par exemple, `man dlopen`) vous donnera tous les détails nécessaires, si vous le souhaitez, ainsi qu'un exemple d'utilisation (précieux, utilisez-le !).

Exercice n°6:

Pour éviter de générer autant de programmes qu'il n'y a d'algorithmes de tri, nous allons produire un seul et unique programme qui chargera dynamiquement et explicitement la bibliothèque de tri à utiliser (bibliothèque dynamique que nous avons construire lors de l'exercice précédent). Nous allons donc maintenant faire un programme nommé `tri.exe` qui prendra en paramètre le type d'algorithme de tri qu'il utilisera. Le programme chargera alors dynamiquement la bibliothèque implémentant cet algorithme de tri.

Pour réaliser cela, voici les étapes à suivre :

Copier le programme de test `main.c` dans `main_dynload.c`. Modifiez ce fichier pour que le nom de la bibliothèque puisse être passé en paramètre sur la ligne de commande. Le traitement de ce paramètre consistera à appeler la fonction :

```
void load_library(const char *library_name);
```

où `library_name` désigne le nom en question. Cette fonction `load_library` devra être définie dans un nouveau fichier-source, `load_library.c`. Elle aura pour rôle de :

- charger dynamiquement la bibliothèque correspondant au nom de la stratégie,
- résoudre les références, grâce à `dlsym()`, à la fonction d'interface de l'algorithme de tri

Pour vous aider, vous pourrez consulter la page de manuel de `dlopen` qui contient un exemple.

Vous définirez, toujours dans `load_library.c`, la fonction d'interface de `sort.h`, mais en leur donnant un corps qui est un simple relai d'appel de la « vraie » fonction (celle de la bibliothèque dynamique), à travers le pointeur que vous a renvoyé `dlsym()`.

Modifiez le fichier `Makefile` pour créer un exécutable unique, `tri.exe`, de cette manière :

```
gcc -rdynamic -o tri.exe main_dynload.o utils.o load_library.o -ldl
```

Notez l'option `-rdynamic`, qui permet à une bibliothèque chargée explicitement, de chercher ses références non résolues dans l'exécutable lui-même[4]. Examinez la taille de l'exécutable produit, et comparez-la à celle des exécutables obtenus précédemment. Le résultat peut vous paraitre paradoxal. Essayez de le justifier et de l'analyser. Déterminer, grâce à `ldd`, les bibliothèques dont dépend votre exécutable. Vous ne devez pas y trouver les bibliothèques spécifiques de tri (`libTri_xxx-dynamicLib.so`).

Enfin modifiez le fichier `Makefile` pour ajouter une section de test avec ce nouveau programme.

Exercice n°7:

Ajoutez un nouvel algorithme de tri. Vous reprendrez par exemple le code du tri vu lors du TD précédent (*Shell sort*) et vous en ferez une librairie dynamique que vous chargerez avec votre programme générique `tri.exe`. Ceci vous démontre qu'en respectant l'interface d'une fonction d'une bibliothèque, il est possible dynamiquement de fournir une autre implémentation. C'est typiquement sur ce modèle que sont aussi faits les plugins.

Exercice n°8:

[4] Lorsqu'une bibliothèque est chargée, les autres bibliothèques dont elle dépend (i.e., dans lesquelles elle a des références à résoudre) le sont aussi. En revanche, par défaut, la bibliothèque ne résout pas de références dans l'exécutable qui l'a chargée. L'option `-rdynamic` permet donc cette sorte de « retro » éditions de liens.

Au lieu de passer le type d'algorithme de tri utilisé en paramètre de votre exécutable, modifiez votre `main` pour charger successivement les différentes bibliothèques implémentant les algorithmes de tri. Vous serez alors amenés à utiliser le déchargement de bibliothèque.

5 Conclusion et conséquences du chargement dynamique de bibliothèque

Exercice n°9:

Quelles sont les bibliothèques dynamiques utilisées par les programmes `java` et `python` qui se trouvent dans `/usr/bin/` ? Que pouvez-vous dire sur les bibliothèques utilisées ? Que pouvez-vous en conclure ?

Que le chargement dynamique de bibliothèque soit implicite par l'exécutable (donc référencement créé lors de l'édition de liens) ou bien explicite par votre programme (c'est lui qui charge explicitement telle ou telle bibliothèque dynamique), il faut que vous soyez bien conscient des avantages mais aussi des inconvénients.

En particulier, si vous jouez avec la variable d'environnement `LD_LIBRARY_PATH` (en la mettant à la valeur du dossier courant par exemple) et que je viens mettre dans ce dossier courant une bibliothèque qui remplacera la bibliothèque C du système, en ajoutant un peu de code qui me permettra de capturer et d'envoyer sur Internet tout ce que vous tapez au clavier lors de l'exécution de ce programme ou de tout autre programme qui utilise la bibliothèque C… Et bien vous n'y verrez que du feu.

Et si vous autorisez le chargement dynamique explicite de bibliothèque par votre programme, vous devez faire confiance à la bibliothèque qui vous est fournie car celle-ci peut ajouter n'importe quelle implémentation qui sera appelée à l'appel d'une des fonctions de l'interface…

Vous pourrez aussi vous pencher sur l'utilisation de la variable d'environnement `LD_PRELOAD` (voir la page : https://www.0x0ff.info/2014/hook-lib-linux-ld_preload/)… Pour tous les hackers en puissance que vous êtes, cela doit ouvrir des perspectives quant à la sécurité des programmes si l'on comprend bien comment tout cela fonctionne !