Cours "Informatique Embarquée"

François Armand

M2 (IMPAIRS, LP, MIC, EIDD...)

Exercice N° 6 a, 8 Décembre 2017

A rendre avant le 31 Décembre 2017 minuit

armand@informatique.univ-paris-diderot.fr

Gestion mémoire et MMU

1 Consignes:

La remise se fera en suivant les consignes habituelles.

2 Introduction

Le but de ce TP est de :

- Chercher à construire un programme de toute petite taille
- Observer et comprendre les mécanismes de gestion MMU Linux

3 Un petit programme

3.1 Ré-écrivons la commande true :

Écrire un programme en C dont la seule fonction est de se terminer avec un code retour 0. En fait c'est la commande Unix « `true` » ! Compilez pour produire un binaire exécutable statique. Si vous êtes sur une machine 64 bits, utilisez de préférence l'option `-m32` pour générer du code 32 bits.

Utilisez les commandes `file, size,ls` et `readelf` pour obtenir :

- le type et la taille du fichier
- la taille des sections de code et de données
- la taille (mémoire et disque) des segments définis dans le fichier ELF (code et données)

Comparez et commentez.

3.2 Réduisez la taille du fichier :

- Utilisez la commande `strip` ou l'option `-s` de `gcc`
- Comparez et commentez.

3.3 Start versus main

- Avec la commande `nm`, cherchez les adresses des symboles `main` et `_start`
- Avec la commande readelf, trouvez l'adresse de la première instruction à exécuter
- Expliquez brièvement ce que fait la bibliothèque C avant d'invoquer la fonction main.

3.4 Plus de main !

- Modifiez votre magnifique commande `true`, pour qu'elle démarre par `_start` au lieu de main, et qu'elle se termine par `_exit`, plutôt que par `return` ou `exit`.
- Générez votre commande (binaire statique, toujours). Documentez vous sur l'option `-nostartfiles` de gcc . Et éventuellement sur l'option `-nostdlib.`
- Vérifiez les nouvelles tailles obtenues.

3.5 Plus de librairie ?

- On va fournir tout le code dont on a besoin, à savoir le code de l'appel système `_exit`.

```
void myexit()
{
__asm__
    (
     "  movl     $1,%eax   \n"
     "  movl     $42,%ebx  \n"
     "  int      $0x80     \n"
    );
}
```

- Recompilez
- Quelles sont les tailles obtenues ?
- Voir les liens suivants :
  - https://www.codeproject.com/articles/15971/using-inline-assembly-in-c-c
  - https://filippo.io/linux-syscall-table/

3.6 Encore plus petit ?

- Essayez de trouver si le binaire exécutable ELF ne contiendrait pas des sections dont on pourrait se passer, et utilisez la commande `strip -R` … pour les éliminez…
- Vérifiez que votre programme fonctionne toujours.
- Quelles tailles avez-vous obtenues ?
- Pourriez-vous faire encore plus petit ?

3.7 Références

- http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html
- http://timelessname.com/elfbin/
- http://www.muppetlabs.com/~breadbox/software/elfkickers.html

4 Observations du comportement de la gestion mémoire

4.1 Frère Jacques, dormez-vous ?

- Écrivez un programme dont la seule fonction est de dormir un nombre de secondes assez long pour vous laisser le temps de l'observer. Disons 3600 secondes.
- Générez un binaire exécutable statique (ex : `jacques`)
- Exécutez ce programme en background (ex : `./jacques &`)

- Pendant que ce programme « tourne »
  - trouvez son pid
  - Dans le fichier /proc/[pid]/maps retrouvez la région de code et la région de données de votre programme. On peut aussi utiliser la commande `pmap -x .`
  - Dans le fichier /proc/[pid]/smaps retrouvez la région de code et la région de données de votre programme.
    - Regardez les valeurs Size, RSS et PSS des régions de code et de données.
    - Voir la page manuel http://man7.org/linux/man-pages/man5/proc.5.html
    - Vous expliquerez brièvement ce que représentent ces trois champs.

4.2 Deux frères Jacques

- Exécutez une deuxième instance de votre programme
- Retrouvez les valeurs Size , RSS et PSS de la première instance.
  - Quels changements observez-vous ?
  - Expliquez.

4.3 Copy On Write

Dans le répertoire, ~armand/Donwloads vous trouverez une archive tpMemoire.tgz qui contient :

- Un `Makefile`
- Un fichier `fatiny.c`
- Un shell script `getpg.sh`

Vous étudierez le code de `fatiny.c`… En bref, c'est un programme qui fait fork. L'exécution est décrite par des traces et n'avance que lorsque l'utilisateur tape la touche « Entrée » du clavier.

`getpg.sh` prend en paramètre un pid, un « numéro de région » (cf /proc/[pid]/maps) et un nombre de pages ou « all ». Exemple :

```
# ./getpg.sh 2290 1 all
Process 2290 : Page 1 of region 1 is present (PFN:
a60000000000d2ea) and has count 0000000000000001
#./getpg.sh 2290 2 all
Process 2290 : Page 1 of region 2 is not present
```

On sait que la région N°1 (le code du binaire exécutable) du processus 2290 est présente en mémoire physique et qu'elle n'est référencée qu'une seule fois.

Par contre la région N°2 (les données du binaire exécutable) n'a pas encore de page en mémoire.

Vous exécuterez `fatiny` en pas à pas, et à chaque étape, vous utiliserez le shell script `getpg.sh` pour observer ce qui se passe. Vous consignerez vos observations et expliquerez ce qu'il se passe.