

TP1 : Programmation Système (partie 1) - 2 heures

Contents

| | | |
|----------|---|----------|
| 1 | Généralités | 1 |
| 1.1 | embsys | 1 |
| 1.2 | Compte rendu | 1 |
| 2 | Avant toutes choses, rappel sur les pointeurs | 1 |
| 3 | Simulateur GPS | 2 |
| 3.1 | National Marine Electronics Association (Norme NMEA 0183) | 2 |
| 3.2 | Communication série | 3 |
| 3.3 | Compilation du simulateur | 3 |
| 4 | Exercice 1 : GDB et fichier core | 3 |
| 4.1 | Les questions | 3 |
| 4.2 | Ce qu'il faut retenir | 4 |
| 5 | Exercice 2 : LD_PRELOAD et sigaction | 5 |
| 5.1 | Les questions | 5 |
| 5.2 | Ce qu'il faut retenir | 5 |
| 6 | Exercice 3 : select, fork et pipe | 6 |
| 6.1 | Les questions | 6 |
| 6.2 | Ce qu'il faut retenir | 6 |

1 Généralités

1.1 embsys

L'ensemble des cours, exemples, PDF et TP sont disponibles sur le dépôt github <https://github.com/pblottiere/embsys>.

Si vous voulez cloner entièrement le dépôt :

```
$ git clone https://github.com/pblottiere/embsys
```

Si vous voulez cloner le dépôt mais avoir simplement les labs dans votre répertoire de travail :

```
$ git clone -n https://github.com/pblottiere/embsys --depth 1
$ cd embsys
$ git checkout HEAD labs
```

Si vous n'avez pas **git**, téléchargez le ZIP sur la page d'accueil de **embsys**.

Le TP d'aujourd'hui se trouve ici : https://github.com/pblottiere/embsys/1_sysprog_part1.

1.2 Compte rendu

Un compte rendu est demandé pour le TP sous forme d'archive (tarball, zip, ...) contenant :

- les réponses aux questions dans un format libre (.txt ou autre)
- le code source associé

Le compte rendu :

- doit être rendu au plus tard 1 semaine après le TP
- ne sera pas noté mais permettra d'adapter les TP suivants et me rendra plus indulgent (si le contenu est correct) si vous avez un "pépin" lors de l'examen

Le travail peut être effectué en binôme.

2 Avant toutes choses, rappel sur les pointeurs

Un morceau de code source rappelant le principe de pointeur en C :

```
#include <stdio.h>

void print_str(char * str)
{
    printf("%s\n", str);
}

void print_valeur(int valeur)
{
    printf("%x\n", valeur);
}

int main()
{
    int valeur = 10;
    int * pointeur_valeur = &valeur;
```

```

int * pointeur2_valeur;
pointeur2_valeur = &valeur;

printf("%x\n", valeur);
printf("%x\n", *pointeur_valeur);
printf("%x\n", pointeur_valeur);
printf("%x\n", &valeur);

char str[10] = "SKYWALKER";
print_str(str);
}

```

Lors de l'exécution :

```

a
a
bf8b89e4
bf8b89e4
SKYWALKER

```

3 Simulateur GPS

3.1 National Marine Electronics Association (Norme NMEA 0183)

La norme NMEA 0183 est une spécification pour la communication entre équipements marins, dont les équipements GPS. Elle est définie et contrôlée par la National Marine Electronics Association (NMEA), association américaine de fabricants d'appareils électroniques maritimes, basée à Severna Park au Maryland (États-Unis d'Amérique). La norme 0183 utilise une simple communication série pour transmettre une phrase à un ou plusieurs écoutants. Une trame NMEA utilise tous les caractères ASCII. (Wikipedia)

Ces trames NMEA 0183 sont de tailles variables et codées en ASCII (caractères 8 bits) contrairement aux trames NMEA 2000. Elles commencent toutes par le caractère \$ (excepté les trames venant de l'AIS commençant par !). Les deux caractères suivants indiquent le type de matériel. Ainsi, on a par exemple pour les trois premiers octets :

- \$GP : trame GPS
- \$HC : trame compas
- \$RA : trame radar
- ...

Par exemple :

```

$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39
$GPGGA,123519,4807.038,N,01131.324,E,1,08,0.9,545.4,M,46.9,M, , *42
$GPGLL,4916.45,N,12311.12,W,225444,A
$GPVTG,054.7,T,034.4,M,005.5,N,010.2,K
...

```

Aujourd'hui, nous allons travailler avec un simulateur GPS "de fortune" simulant l'envoi de deux types de trames par liaison série virtuelle :

- trame GLL : Geographic position, latitude / longitude
- trame VTG : Track made good and ground speed

Ces trames sont envoyées périodiquement par le binaire **gps** que nous verrons plus tard.

NOTE : Grâce aux équipements marins, une carte en ligne indique en temps réel la position des navires : <http://www.marinetraffic.com/-fr/>.

3.2 Communication série

Dans la vraie vie, une communication série (ou USB) transmet ces trames NMEA et un ordinateur peut alors les récupérer.

Lors de la connexion d'un port USB-Série, une entrée est créée par le kernel dans le répertoire **/dev/** (généralement par l'utilitaire **udev**). En examinant les sorties du kernel, on peut trouver le port exact :

```
$ dmesg
...
[15976.212024] usb 2-2: new full-speed USB device number 2 using uhci_hcd
[15976.375039] usb 2-2: New USB device found, idVendor=067b, idProduct=2303
[15976.375044] usb 2-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[15976.375047] usb 2-2: Product: USB-Serial Controller
[15976.375049] usb 2-2: Manufacturer: Prolific Technology Inc.
[15978.090200] usbcore: registered new interface driver usbserial
[15978.090219] usbcore: registered new interface driver usbserial_generic
[15978.090234] usbserial: USB Serial support registered for generic
[15978.174057] usbcore: registered new interface driver pl2303
[15978.174078] usbserial: USB Serial support registered for pl2303
[15978.174101] pl2303 2-2:1.0: pl2303 converter detected
[15978.186178] usb 2-2: pl2303 converter now attached to ttyUSB0
```

Ici, on s'attend à avoir un port **/dev/ttyUSB0**.

Dans le cadre du TP, le simulateur va lui aussi créer une entrée dans le **/dev/**. Cependant, cette entrée étant virtuelle et créée par programmation, elle ne sera pas visible à travers les messages kernel.

3.3 Compilation du simulateur

Le simulateur GPS se trouve dans le répertoire **labs/gps/** qui contient lui même :

- **src** : répertoire contenant les sources
- **bin** : répertoire contenant les binaires (après compilation)
- **lib** : répertoire contenant les bibliothèques (après compilation)
- **include** : répertoire contenant les headers (après compilation)
- **Makefile** : fichier définissant les règles de compilation
- **run.sh** : fichier lançant le simulateur GPS

Lancez la compilation :

```
make
```

Deux bibliothèques sont compilées **lib/libnmea.so** et **lib/libptmx.so** ainsi que le binaire **bin/gps**.

4 Exercice 1 : GDB et fichier core

4.1 Les questions

Une fois le simulateur GPS compilé, le lancer grâce au script **labs/gps/run.sh** :

```
$ sh run.sh
PTY: /dev/pts/X
```

Question 1 : *Que se passe-t-il au bout de quelques secondes? Qu'en déduisez vous?*

Question 2 : *Quel signal a reçu le processus pour se terminer ainsi? Comment vérifiez vous le numéro du signal reçu?*

Lors d'une terminaison anormale, un fichier **core** peut être généré. Par défaut, la génération d'un fichier core est généralement désactivée :

```
$ ulimit -c
0
```

Ici la commande renvoie grâce au paramètre **-c** la taille du fichier core à générer. La taille étant 0, aucun fichier n'est créé. Pour y remédier :

```
$ ulimit -c unlimited
$ ulimit -c
unlimited
```

Relancez le simulateur GPS. Suite au crash, un fichier core doit être généré dans le répertoire courant.

Nous allons ici utiliser GDB pour analyser le dump mémoire afin de trouver l'origine de l'erreur. GDB est un outils très complet fournissant de nombreuses commandes.

Pour lancer GDB et analyser un fichier core :

```
$ gdb <binary> <core>
```

Ensuite, dans le prompt GDB, utilisez la commande **bt** (pour **backtrace**) afin de savoir comment votre programme en est arrivé là (image de la pile).

Question 3 : *Grâce à GDB et au fichier **core** généré, analysez la source du problème du binaire **gps**. Quelle partie du code est fausse? Pourquoi?*

GDB peut être aussi lancé de manière interactive :

```
$ gdb <binary>
```

Une fois dans le prompt, il faut lancer la commande **r** (comme **run**).

Question 4 : *Que se passe-t-il quand vous lancez GDB en mode interactif sur le binaire **gps**? Pourquoi?*

Suite au problème repéré, allez dans le répertoire **labs/gps/bin** et lancez la commande suivante :

```
ldd ./gps
```

Question 5 : *À quoi sert la commande **ldd**? Quelle information supplémentaire cela vous apporte-t-il?*

Question 6 : *Comment résoudre ce problème en tant qu'utilisateur? N'hésitez à regarder le fichier ***labs/gps/run.sh****

Relancez **ldd** puis **GDB** pour vérifier que votre solution a porté ses fruits.

Il existe aussi une version de GDB pour déboguer à distance. Il y a alors un GDBServer tournant sur la cible où le programme à déboguer est exécuté. Ensuite, un client GDB tourne sur la machine servant à déboguer et communique avec le serveur grâce au réseau.

Question 7 : *Dans quel contexte ce type d'outils peut être intéressant?*

4.2 Ce qu'il faut retenir

- l'utilité de **ulimit** et comment déclencher la génération d'un fichier core
- à quoi sert **GDB** et comment l'utiliser
- l'utilité de **ldd**
- pourquoi, quand et comment utiliser la variable d'environnement **LD_LIBRARY_PATH**

Ces outils vous permettront de savoir comment réagir face à un logiciel buggé (GDB est le BFF du développeur)!

5 Exercice 2 : LD_PRELOAD et sigaction

5.1 Les questions

Maintenant que le problème est identifié, nous allons le résoudre. Cependant, nous partons du principe que le code source du simulateur **NE DOIT PAS ÊTRE MODIFIÉ**. Pour corriger le problème, nous allons utiliser la variable d'environnement **LD_PRELOAD**. Cette variable permet de *hooker* (comprendre *usurper*) certaines fonctions d'une application.

Utilisation :

```
LD_PRELOAD=<libhook.so> <binary>
```

En faisant ainsi, le binaire cherchera en priorité les fonctions dont il a besoin dans **libhook.so**! Pour que cela fonctionne, il faut que les fonctions définies dans libhook aient exactement le même prototype.

Pour les questions suivantes, allez dans le répertoire de travail **labs/1_sysprog_part1/src/ld_preload**. Vous devrez travailler sur trois fichiers:

- hook.c
- Makefile
- run.sh

Question 8 : Implémentez dans le fichier hook.c la fonction à l'origine du problème repéré au sein du simulateur GPS mais cette fois-ci sans erreur.

Question 9 : Éditez le Makefile pour compiler hook.c sous la forme d'une librairie partagée nommée libhook.so (s'inspirer de labs/gps/src/lib/ptmx/Makefile). Testez la compilation.

Question 10 : Éditez le fichier run.sh pour utiliser LD_PRELOAD au moment de lancer le simulateur et ainsi hooker le binaire avec la librairie libhook.so. Exécutez run.sh : le simulateur ne doit plus partir en segfault.

Nous avons ici hooké une fonction définie dans une librairie "utilisateur". On peut réaliser la même opération sur les librairies systèmes. Par exemple, le simulateur GPS utilise la fonction **printf** dès son lancement.

Question 11 : Utilisez le **man** pour déterminer le prototype de la fonction **printf** (expliquez comment vous utilisez **man** dans ce cas et pourquoi). Comment est appelé ce type de fonction?

Question 12 : Hookez le simulateur pour que ce dernier ne puisse plus être interrompu par le signal SIGINT (Ctrl-C) en réimplémentant la fonction **printf** dans libhook.so. Pour cela, utilisez la fonction **sigaction** pour mettre en place un gestionnaire de signaux (s'inspirer de gps/src/bin/gps/gps.c).

Question 13 : Comment faire pour interrompre le processus étant donné que ce dernier ne répond plus au Ctrl-C? Citez deux méthodes.

Pour les parties suivantes, enlevez le hook du **printf** pour assurer un fonctionnement valide du simulateur.

NOTE : LD_PRELOAD peut servir dans bien des situations...

5.2 Ce qu'il faut retenir

- comment compiler des fichiers C pour obtenir une librairie partagée
- le fonctionnement global d'un Makefile
- comment utiliser le **man**
- la mise en place d'un gestionnaire de signaux grâce à **sigaction**
- l'utilité et le fonctionnement de **LD_PRELOAD**

6 Exercice 3 : select, fork et pipe

6.1 Les questions

Lors du lancement du simulateur GPS, nous obtenons le message suivant :

```
$ sh run.sh
PTY: /dev/pts/3
```

Question 14 : Selon vous, à quoi correspond le champs indiqué par **PTY**?

Pour la suite, placez vous dans le répertoire **labs/1_sysprog/src/gps_reader** contenant :

- **reader.c** : le code du reader que nous allons modifier
- **Makefile** : les règles de compilation
- **util.c / util.h** : des fonctions utiles pour l'exercice

Pour compiler, lancez :

```
make
```

Un binaire **gps_reader** est alors généré.

Lancez le reader sans paramètre pour avoir l'aide et en déduire son utilisation. Puis exécutez le avec les paramètres nécessaires et observez les trames NMEA.

Question 15 : En regardant le code de **reader.c**, y a-t-il quelque chose qui vous chagrine?

Question 16 : Grâce à des recherches Internet (ou en fouinant dans le code du simulateur), déterminez dans quelle trame et dans quel champs la date est définie.

Question 17 : Quelles fonctions sont utilisées dans **reader.c** pour ouvrir/écouter/lire/fermer le port virtuel du simulateur? Comment s'appelle ce type de programmation?

Question 18 : Modifiez le code de **reader.c** afin qu'il puisse écouter les trames provenant de deux simulateurs GPS différents (ports paramétrables au lancement). Vérifiez le bon fonctionnement en lançant deux instances du simulateur GPS.

Question 19 : Utilisez l'appel système **fork** et faites en sorte que le père s'occupe de la lecture des ports (le fils ne fait rien pour l'instant).

Question 20 : Créez un pipe et faites en sorte que le père envoie la date de la trame GLL au fils grâce au pipe. Vous pouvez utiliser les fonctions du fichier **util.h** pour vous aider.

Question 21 : Le fils doit écouter la sortie du pipe (toujours **select**) et utiliser **syslog** pour afficher l'heure dans la console ainsi que le PID du père.

Question 22 : La taille des messages échangés entre le père et le fils grâce au pipe étant toujours la même, quel autre mécanisme IPC aurait-on pu utiliser?

6.2 Ce qu'il faut retenir

- comment utiliser **getopt** pour passer des paramètres en ligne de commande à votre application
 - comment utiliser **select** pour monitorer plusieurs file descriptors
 - comment utiliser **fork**
 - comment utiliser un **pipe** pour établir une IPC
 - l'utilisation globale de **syslog**
-