TP1 : Programmation Système (partie 1) - 2 heures

Contents

1 embsys

L'ensemble des cours, exemples de cours, 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 l'archive **labs.tar.gz**.

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

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.

3 Norme NMEA 0183

3.1 National Marine Electronics Association

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 suivant 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 :

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39
$GPGSV,2,1,08,01,40,083,46,02,17,308,41,12,07,344,39,14,22,228,45*75
$GPRMC,123519,A,4807.038,N,01131.000,E,022.4,084.4,230394,003.1,W*6A
$GPGLL,4916.45,N,12311.12,W,225444,A,*1D
...
```

Aujourd'hui, nous allons travailler avec un simulateur GPS "de fortune" simulant l'envoie 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 periodiquement 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.

4 Compilation du simulateur GPS

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 librairies (après compilation)
- include : répertoire contenant les headers (après compilation)

- Makefile : fichier définissant les règles de compilation
- run.sh : fichier lançant le simulateur GPS

Lancer la compilation :

```
make
```

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

5 Exercice 1 : GDB et fichier core

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

```
$ sh run.sh
PTTY: /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 le vérifiez vous?*

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
```

Relancer 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 très nombreuses commandes. Nous allons ici en voir simplement quelques unes.

Pour lancer GDB et analyser un fichier core :

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

Ensuite, dans le prompt GDB, utiliser 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, analyser la source du problème du binaire **gps**. Quelle partie du code est fausse? Pourquoi?*

Question 4 : *À quoi sert la fonction **snprintf**?*

GDB peut être aussi lancé de manière interactive :

```
$ gdb <binary>
```

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

Question 5 : *Que se passe-t-il quand vous lancez GDB en mode intéractif sur le binaire **gps**? Pourquoi?*

Suite au problème repéré à la **Question 4**, aller dans le répertoire **labs/gps/bin** et lancez la commande suivante :

```
ldd ./gps
```

Question 6 : *À quoi sert la commande ldd? Quelle information supplémentaire cela vous apporte-t-il (par rapport à la **Question 4**)?*

Question 7 : *Comment résoudre ce problème en tant qu'utilisateur? (ne pas hésiter à regarder le fichier **labs/gps/run.sh**)*

Relancer ldd et 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 8 : *Dans quel contexte ce type d'outils peut être intéressant?*

NOTE : Un debugger est le BFF du développeur!

6 Exercice 2 : LD_PRELOAD et gestionnaire de signaux

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 d'abord 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.

Question 9 : *Implémentez la fonction à l'origine du problème repéré au sein du simulateur GPS mais cette fois-çi sans erreurs et dans un fichier séparé.*

Question 10 : *Écrivez un Makefile pour la compilation de cette fonction sous forme d'une librairie partagée (s'inspirer de **labs/gps/src/lib/ptmx/Makefile**).*

Question 11 : *Utilisez LD_PRELOAD pour hooker le binaire **labs/gps/bin/gps** avec votre propre librairie et ainsi empêcher le simulateur GPS de bugger.*

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**.

Question 12 : *Utiliser le **man** pour déterminer le prototype de **printf**. Comment est appellé ce type de fonction?*

Question 13 : *Hookez le simulateur pour que ce dernier ne puisse plus être interrompu par le signal SIGINT (Ctrl-C) en passant par la fonction **printf**. Pour cela, utilisez la fonction **sigaction** pour mettre en place un gestionnaire de signaux (s'inspirer de **gps/src/bin/gps/gps.c**).*

Question 14 : *Comment faire pour interrompre le processus étant donné que ce dernier ne répond plus au Ctrl-C?*

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...

7 Exercice 3 : select, fork et pipe

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

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

Question 15 : *Selon vous, à quoi correspond le champs indiqué par **PTTY**?*

Nous allons maintenant compiler un binaire permettant de lire les trames NMEA émises par le GPS. Pour cela :

```
cd 1_sysprog_part1/src/gps_reader
make
```

Un binaire **gps_reader** est alors généré.

Question 16 : *Lancez le reader sans paramètre pour avoir l'aide et déduire son utilisation. Observez les trames NMEA. Grâce à des recherches Internet (ou en fouinant dans le code du simulateur), déterminer dans quel trame et dans quel champs la date est définie.*

Question 17 : *Quelles fonctions sont utilisées pour ouvrir et lire le port virtuel du simulateur? Comment s'appelle ce type de programmation?*

Question 18 : *Modifiez le code du **reader** afin qu'il puisse écouter les trames venant 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.*

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.*

8 Conclusion : les points à connaître après le TP

En programmation de base :

- compilation d'un binaire et d'une librairie partagée grâce à un Makefile
- gestion de base des signaux
- lecture de port série (**select**, **FD_SET**, . . .)
- passage de paramètres en ligne de commande (**getopt**)
- LD_LIBRARY_PATH
- LD_PRELOAD

Les commandes :

- dmesg
- ldd
- ps
- kill
- ulimit
- man

Les outils :

- gdb
- minicom