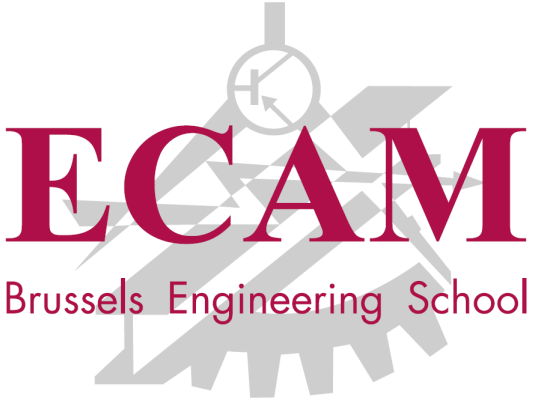
Laboratoire de système d'exploitation

Création d'un shell

24/12/2017

ALBERT Emile 14022@ecam.be
HAGOPIAN Armen 14040@ecam.be
SELLESLAGH Tom 14164@ecam.be



1 Introduction

Les objectifs de ce laboratoire étaient de penser et d'implémenter un shell. Notre groupe étant composé de trois électroniciens, tout trois participant au projet Eurobot, nous avons décidé de profiter de ce laboratoire pour créer une interface système permettant de monitorer les tests sur un robot commandé par une raspberry pi.

Le but de ce shell est donc de pouvoir observer lors de l'exécution d'un test de commande moteur sur la raspberry, un maximum d'informations intéressantes sur l'état du contrôleur. L'interface système devient alors une aide au debugging des programmes s'exécutant sur la raspberry.
Le code complet de notre solution est disponible sur github : `https://github.com/EmileAlbert/2I4030_Shell`

2 Implémentation

2.1 L'environnement du shell

La structure du shell telle que nous l'avons utilisée dans ce projet a été reprise du code créé par Stephen Brennan sur son compte Github : `https://github.com/brenns10/lsh`

La structure du shell peut être vue en plusieurs sections :

- La déclaration des fonctions de l'interface, des pointeurs vers ces fonctions et des noms d'appel
- L'implémentation des fonctionnalités
- Le cœur du shell

2.1.1 Déclaration des fonctions de l'interface, des pointeurs et des noms d'appel

Dans cette partie de configuration, il s'agit de déclarer les variables associées aux fonctions utilisées dans le programme ainsi que leur pointeur.

De plus, on définit les noms d'appel des fonctions. Ces noms correspondent aux chaînes de caractères utilisées par l'utilisateur pour invoquer telle ou telle fonction.

2.1.2 Implémentation des fonctions

Cette partie regroupe l'ensemble des fonctions appelées lors d'une commande utilisateur. Les fonctions propres à notre projet sont donc codées dans cette section ainsi que les commandes propres au shell (`help, clear, exit`)

2.1.3 Coeur du shell

Enfin, la dernière partie de la structure est composée des 6 fonctions requises au bon fonctionnement du shell. Ces 6 fonctions sont :

- `rpi_launch` - Lancement du shell
  Fork du programme principal et gestion des enfants créés
- `rpi_execute` - Exécution des fonctions
  Cette section lance les fonctions correspondantes à l'entrée de l'utilisateur
- `rpi_read_line` - Lecture des commandes user
  Fonction récupérant la chaîne de caractères entrée par l'utilisateur
- `rpi_split_line` - Décryptage des commandes user
  Fonction servant à décrypter et à conditionner les entrées de l'utilisateur pour que ses commandes soient comprises
- `rpi_loop` - Super loop d'exécution
  C'est la partie principale du programme qui appelle les fonctions nécessaires au shell en boucle
- `main` - Fonction principal du programme
  Script qui lance la boucle et donc le shell

2.2 Les commandes implémentées

2.2.1 `rpi_raspInfo` invoqué avec `rasp`

La commande `rasp` permet de récupérer des informations sur le hardware de la raspberry. Ces données peuvent être utiles tant pour le *debugging* que pour l'optimisation des performances d'un programme.

La fonction prend obligatoirement en paramètre l'information désirée. Les différentes informations disponible dans la fonction sont les suivantes :

- `rasp -t [PARAM]` permet de récupérer la température du processeur

  La commande prend obligatoirement en paramètre l'unité dans laquelle on veut récupérer la température :

  - `rasp -t cel` - Température en degré Celsius
  - `rasp -t far` - Température en degré Farenheit
  - `rasp -t kel` - Température en degré Kelvin

- `rasp -cpu` permet de récupérer le pourcentage d'utilisation du CPU au temps t et de façon statique
- `rasp -ram` permet de connaitre le pourcentage d'utilisation de la mémoire RAM au temps t et de façon statique

2.2.2 `rpi_pigpiod_status` invoqué avec `pigpio`

La commande `pigpio` permet de lister les pins gpio de la raspberry et également d'afficher l'état de chacune d'elles. La commande est uniquement compatible avec des raspberry possédant 26 pins gpio étant donné que cette commande s'attend à en trouver 26.

2.2.3 `rpi_help` invoqué avec `help`

La commande `help` peut être vue comme un récapitulatif global de toutes les fonctions. Elle affiche le contexte d'utilisation et les résultats attendus de la plupart des commandes. On notera que la fonction **help** se mettra automatiquement à jour lorsque l'on rajoute une nouvelle commande qui possède l'option **-h**

2.3 Appels systèmes utilisés

2.3.1 `fork`

Cet appel système est utilisé dans le script du shell pour créer un processus enfant gérant la commande et l'exécution de la fonction demandée.

2.3.2 `open`

Cet appel est utilisé dans la fonction `rasp` et plus précisément pour récupérer la valeur de la température inscrite plus tôt dans un ficher texte.

2.3.3 `_exit`

2.4 Gestion des erreurs

Les erreurs générées par une mauvaise entrée de l'utilisateur sont gérées dans le code de base de notre *shell*. Un mauvaise entrée n'aura pas pour conséquence l'arrêt du programme mais provoquera simplement un message d'alerte. Le code est donc robuste et ne risque pas de planter durant l'utilisation.

2.5 Script de compilation

Pour créer notre Fichier ELF, un fichier Makefile a été créé. Celui-ci utilise principalement le programme `gcc` pour compiler le fichier `main.c`.

3 Discussion

Un interpréteur de commandes peut exécuter les commandes lui-même ou déléguer cette exécution à d'autres exécutables. Quels sont les avantages et inconvénients de ces deux possibilités ?

Délégation à un autre exécutable
Avantages :

- Le shell délégant l'exécution des tâches à ses processus enfant ne crash pas si le processeur crash à cause de la commande ou retourne une erreur
- Possibilité de faire plusieurs tâches simultanément en plaçant les enfants en arrière plan
- Les processus sont indépendants entre eux

Inconvénients :

- Le processus parent peut attendre indéfiniment un fils zombie
- Plus grand espace requis

Les avantages de la méthode *fork* sont les désavantages de la méthode où l'interpréteur de commandes exécute lui même les commandes et inversement. Dans notre interpréteur de commandes, nous avons utilisé la méthode *fork*.

Pour interagir avec le hardware et le système d'exploitation, on peut utiliser un appel système, une fonction de la librairie standard C ou passer par un programme système. Quels sont les avantages et inconvénients de ces trois possibilités ?

Appel système
Avantages :

- Les appels systèmes sont des commandes prioritaires
- Les appels systèmes sont sécurisant, il est impossible pour un appel système *user* de pénétrer dans la partie de l'OS ou dans les documents d'autres utilisateurs

Inconvénients :

- Un appel système doit passer du mode *user* au mode *kernel* ce qui engendre un temps d'exécution plus long

Fonction de la librairie standart C
Avantages :

- Portable car peu volumineux grâce à l'implémentation de fonctions uniquement élémentaires (léger en comparaison de java par exemple)
- Facilite la création de programmes dû à l'abstraction proposée
- Empêche de causer des erreurs dans l'OS suite à d'éventuelles erreurs de la part de l'utilisateur

Inconvénients :

- Les fonctions fournies par les libraires restent dans le *userspace* et ne peuvent donc pas interagir avec le *hardware*
- Le temps d'exécution peut aussi être allongé dans le cas où la fonction utilise des appels systèmes (nécessaire pour exécuter certaines tâches)

Programme système
Avantage :

- Plus rapides si appels systèmes non requis

Inconvénient :

- Autorisation peu étendue

3.0.1 Utilisation

Notre implémentation du shell repose essentiellement sur l'utilisation des fonctions proposées par le librairie standard C. Cette méthode nous permet de ne pas exécuter des tâches avec des permissions élevée si cela n'est pas nécessaire. De plus, notre code étant conçu pour s'exécuter en parallèle d'un programme, limiter les appels systèmes permet donc de limiter le temps d'exécution.