CSEL

Rapports des TP

Git du projet : https://github.com/Mathistis/csel-workspace

Fribourg, May 13, 2022

Auteurs
Macherel Rémy
remy.macherel@master.hes-so.ch
Raemy Mathis
mathis.raemy@master.hes-so.ch

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Version: 1.0

Contents

Raemy Mathis, Macherel Rémy, CSEL

1. TP Systèmes de fichiers

1.1 Résumé du travail pratique

Ce travail consiste à réaliser une application qui contrôle la fréquence de clignotement d'une LED sur la carte NanoPi. L'application doit permettre grâce aux systèmes de fichiers et en passant par les boutons de la board de régler la fréquence de la LED. Un programme de base (silly_led_control.c) est fourni mais celui-ci consomme l'entièreté d'un coeur du processeur. Il nous est donc demandé de développer un programme qui utilise les systèmes de fichiers pour contrôler la LED ainsi que les boutons et qui fonctionnerait de manière plus optimisée.

1.2 Infos utiles à retenir

Pour obtenir le numéro de GPIO correspondant à une pin, il faut lire la configuration à l'aide de la commande :

```
1    mount -t debugfs none /sys/kernel/debug
2    cat /sys/kernel/debug/gpio
```

Afin d'utiliser des event produits par les gpio associés aux boutons, il est nécessaire d'utiliser *EPOLLERR* car en utilisant *EPOLLIN* cela ne fonctionne pas. Nous n'avons pas réellement trouvé la raison de ceci et suspectons un bug dans le kernel. Pour les boutons, au moment du open sur le fd, il faut faire un pread pour quittancer l'interruption alors que pour le timer le read suffit.

1.3 Feedback global

Nous avons dû retrouver dans un ancien TP les numéros de pin des boutons de la carte car en utilisant les commandes de la section précédente, si ceux-ci ne sont pas activés cela ne les affiche pas.

2. TP 5

2.1 Résumé du laboratoire

Le but de ce laboratoire est de mettre en pratique la théorie vue dans le cours en ce qui concerne le multiprocessing ainsi que les ordonnanceurs. Il se compose de deux exercices, un sur les processus et signaux de communication et l'autre sur l'utilisation des CGroups. Les réponses aux questions ainsi que les difficultés rencontrées ou information utiles à retenir seront détaillés dans plus loin dans ce document.

2.2 Réponses aux questions

2.2.1 Quel effet a la commande echo $$ > ... sur les cgroups ?

L'utilisation des caractères *$$* permet d'obtenir le PID du processus actuel et donc dans notre cas du shell duquel sera lancé le processus. Cette commande permet donc de définir les limites de ressources que pourra utiliser notre processus. Il est utile de préciser que tout processus enfant du shell (correspondant au PID $$) auront les mêmes limites.

2.2.2 Quel est le comportement du sous-système memory lorsque le quota de mémoire est épuisé ? Pourrait-on le modifier ? Si oui, comment ?

2.2.3 Est-il possible de surveiller/vérifier l'état actuel de la mémoire ? Si oui, comment ?

2.3 Synthèse des connaissances acquises

2.3.1 Non acquis

2.3.2 Acquis, mais à exercer

2.3.3 Parfaitement acquis

2.4 Feedback exercices

2.4.1 Exercice 2

Il a été observé que si l'on lance le programme et que l'on essaie de limiter le nombre de bytes possibles alors qu'il a déjà atteint le quota, l'écriture est bloquée et la limite ne peut être fixée. Cependant si l'on écrit la limite avant que le programme ne l'ait atteinte, elle sera bien fixée et le programme s'arrêtera lorsqu'il atteindra cette limite.
Commande pour limiter la quantité max de mémoire :

```
echo 20M > /sys/fs/cgroup/set/program/memory.limit_in_bytes
```

Nous avons également pu observer que si l'on crée deux cgroup de limitation de mémoire, un de 20M et l'autre de 30M. Si l'on ajoute notre PID dans celui de 20M puis dans celui de 30M, au moment de l'ajout dans celui de 30M il est automatiquement retiré des tasks du cgroup qui limitait à 20M.

3. TP 6

3.1 Résumé du laboratoire

3.2 Synthèse des connaissances acquises

3.2.1 Non acquis

3.2.2 Acquis, mais à exercer

3.2.3 Parfaitement acquis

3.3 Feedback