
CSEL

Rapports des TP

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

Fribourg, May 20, 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

Contents

1	TP Systèmes de fichiers	1
1.1	Résumé du travail pratique	1
1.2	Infos utiles à retenir	1
1.3	Feedback global	1
2	TP 5	2
2.1	Résumé du laboratoire	2
2.2	Réponses aux questions	2
2.2.1	Quel effet a la commande <code>echo \$\$ > ...</code> sur les cgroups ?	2
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
2.2.3	Est-il possible de surveiller/vérifier l'état actuel de la mémoire ? Si oui, comment ?	2
2.2.4	Les 4 dernières lignes sont obligatoires pour que les prochaines commandes fonctionnent correctement. Pouvez-vous en donner la raison ?	3
2.2.5	Comportement des shells dans deux cgroups différents	3
2.2.6	Sachant que l'attribut <code>cpu.shares</code> permet de répartir le temps CPU entre différents cgroups, comment devrait-on procéder pour lancer deux tâches distinctes sur le cœur 4 de notre processeur et attribuer 75% du temps CPU à la première tâche et 25% à la deuxième ?	4
2.3	Synthèse des connaissances acquises	5
2.3.1	Non acquis	5
2.3.2	Acquis, mais à exercer	5
2.3.3	Parfaitement acquis	5
2.4	Infos utiles à retenir	5
2.5	Feedback exercices	5
2.5.1	Exercice 1	5
2.5.2	Exercice 2	7
2.5.3	Exercice 3	7
3	TP 6	8
3.1	Résumé du laboratoire	8
3.2	Synthèse des connaissances acquises	8
3.2.1	Non acquis	8
3.2.2	Acquis, mais à exercer	8
3.2.3	Parfaitement acquis	8
3.3	Feedback	8

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 :

```
mount -t debugfs none /sys/kernel/debug  
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 quittance 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 ?

See [Documentation on cgroups about memory - 2.5 Reclaim](#).

Selon la documentation, chaque cgroup maintient une LRU (Least Recently used) pour sa mémoire. Lorsque la limite est atteinte, il essaie d'abord de récupérer une mémoire potentiellement disponible mais dans le cas contraire il invoque la routine OOM (Out Of Memory, voir doc 10. OOM Control). La routine OOM implémente un notifieur utilisant l'api de notifications du cgroup. Il permet d'enregistrer des notifications de dépassement de mémoire. Le OOM-Killer se charge quant à lui de stopper les tâches ayant dépassé leur limite en attendant qu'elles en aient à nouveau.

Lorsque la limite est atteinte le sous-système ne permet plus d'allocation et retourne un pointeur vide à chaque essai d'allocation.

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

Il est possible de surveiller l'état de la mémoire à l'aide de commandes telles que `top` ou `htop`. Ces commandes peuvent être utilisées sans arguments si l'on souhaite surveiller l'état global de la mémoire et des ressources. Cependant, si l'on veut surveiller un processus en particulier, on peut utiliser celles-ci de la manière suivante:

```
| top -p <PID>
```

Il existe plusieurs outils de ce type pour surveiller l'état de la mémoire et des ressources. Une liste de certains d'entre eux est disponible [ici](#). Il est également possible de surveiller ceci en utilisant les fichiers de `/proc/` qui est ce que l'on appelle un pseudo file system. ([doc proc](#)). On peut obtenir l'espace d'adressage avec la commande :

```
| cat /proc/<PID>/maps # or smaps
```

On peut également obtenir les stats et le status avec les commandes :

```
| cat /proc/<PID>/status # status  
| cat /proc/<PID>/statm # stat
```

Ces méthodes ne font cependant pas appel aux cgroups. Si l'on souhaite surveiller nos ressources en utilisant les cgroups à l'aide du *CPU Accounting Controller*. Il peut être créé de la manière suivante:

```
| mount -t cgroup -ocpuacct none /sys/fs/cgroup  
| cd /sys/fs/cgroup  
| mkdir g1 #group name can be changed  
| echo $$ > g1/tasks #add current process to group
```

On peut utiliser `/sys/fs/cgroup/g1/cpuacct.usage` pour obtenir les temps CPU (en nanosecondes) obtenus par ce groupe. `/sys/fs/cgroup/g1/cpuacct.stat` liste quelques statistiques comme par exemple:

- user : le temps passé par les tâches du cgroup en user mode.
- system : Le temps passé par les tâches du cgroup en mode kernel.

Plus d'informations sur ce type de cgroup sont disponibles [ici](#).

Il existe aussi des memory controllers comme ceux utilisés pour la limitation de mémoire. Il existe des moyens de contrôler l'usage de mémoire avec le cgroup `memory.usage_in_bytes`. Des possibilités comme `memory.failcnt` ou `stat` permettent d'obtenir des informations comme le nombre de fois que la limite de mémoire a été atteinte ou d'afficher diverses statistiques. ([cgroup memory](#)).

2.2.4 Les 4 dernières lignes sont obligatoires pour que les prochaines commandes fonctionnent correctement. Pouvez-vous en donner la raison ?

Les lignes 1 et 3 servent à définir le cpu sur lequel chacun des cgroups pourra exécuter ses tâches. Les deux autres écrivant 0 dans `cpuset.mems` servent à définir le noeud mémoire que va utiliser le cgroup. Ceci est souvent utilisé sur les systèmes à architecture NUMA (Non Uniform Memory Access) signifiant que le système peut contenir plusieurs types de mémoires différents symbolisés par des noeuds. Dans le cas du Nanopi nous n'avons qu'une seule mémoire et donc la documentation préconise d'inscrire 0 dans `cpuset.mems` pour les systèmes n'ayant pas de NUMA. (Info trouvée [ici](#)).

2.2.5 Comportement des shells dans deux cgroups différents

On observe que chaque application utilise le 100% du coeur qui lui est affecté. Les deux processus à l'intérieur de chaque application doivent par conséquent se partager le coeur pour leur exécution. On peut observer ceci sur la figure suivante:

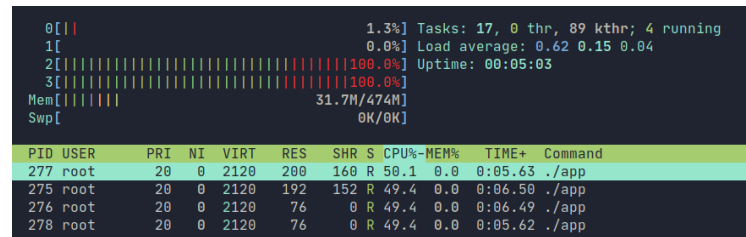


Figure 2.1: Two different shells executing app

On observe ici que la barre d'utilisation des coeurs 2 et 3 est à 100% mais avec deux couleurs ce qui représente les deux processus de chaque application se partageant le coeur.

2.2.6 Sachant que l'attribut `cpu.shares` permet de répartir le temps CPU entre différents cgroups, comment devrait-on procéder pour lancer deux tâches distinctes sur le coeur 4 de notre processeur et attribuer 75% du temps CPU à la première tâche et 25% à la deuxième ?

Nous avons procédé de la manière suivante :

Nous avons gardé les deux cgroup de la question précédente (Low et High) mais cette fois affecté le coeur 4 aux deux tâches. Nous avons ensuite défini à l'aide de l'attribut `cpu.shares` que le premier groupe (High) peut occuper 75% du temps du coeur et le deuxième (Low) uniquement 25%. Pour ce faire nous avons trouvé dans la documentation que la valeur 1024 dans `cpu.shares` correspondait au 100% du coeur nous avons donc mis 768 dans cet attribut pour le High cgroup et 256 pour le Low. On peut voir sur la figure suivante que le coeur 4 est occupé à 100% par 4 processus (deux pour chaque tâche), et que pour la tâche appartenant au groupe 1, les processus utilisent environ 37.5% chacun alors que pour la tâche appartenant au groupe 2, les processus utilisent environ 12.5% de du coeur. Ainsi on peut voir que la tâche 1 (appartenant au groupe High) utilise bien 75% du coeur et la tâche 2 (appartenant au groupe Low) utilise bien 25%.

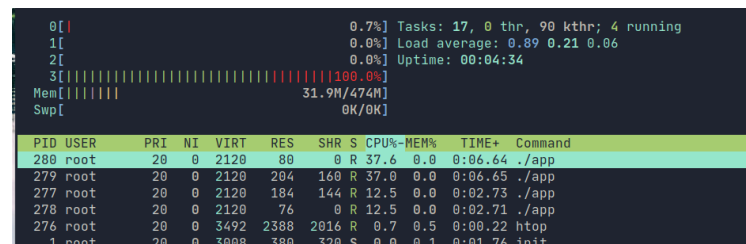


Figure 2.2: 75% and 25% of the CPU on core 4

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 Infos utiles à retenir

La fonction `c_strncmp` retourne la valeur 0 quand les deux chaînes de caractères sont identiques. Nous nous sommes faits avoir par ceci et avons perdu passablement de temps à chercher une erreur alors qu'elle se trouvait simplement à cet endroit.

Lors de la capture des signaux, si l'on capture un signal alors que le processus était en *sleep*, à la fin du traitement de la capture le signal sera réveillé car en effet cela reprend l'exécution du processus là où il en était mais dans le cas d'un *sleep* ne vérifie pas que celui-ci soit terminé. Si l'on voulait s'assurer que le temps du *sleep* soit bien respecté il faudrait utiliser le fait que la fonction *sleep* lorsqu'elle est interrompue retourne le nombre de secondes restantes à son exécution. On pourrait donc comparer à la suite de l'instruction *sleep* si le temps restant est de 0 ou non et dans le cas contraire replonger le processus en sommeil pour la durée de temps restante. De plus, dans le cas d'un besoin de précision il est conseillé d'utiliser *clock_nanosleep* au lieu de *sleep*.

2.5 Feedback exercices

2.5.1 Exercice 1

Nous avons réalisé cet exercice en utilisant un *socketpair* afin de communiquer entre deux processus. Le processus parent envoie via le socket des messages au processus enfant qui les affiche dans la console. Le processus enfant "répond" en affichant simplement dans la console car nous n'avons pas eu le temps de développer le fait qu'il renvoie un message dans le *socketpair*. Une fois le message correspondant à l'ordre de quitter reçu, le processus enfant se termine. Dans le main, le processus parent attend grâce à la fonction *wait* la fin du processus enfant.

L'étape suivante consistait à capturer et ignorer tous les signaux (sauf le 9 SIGSTOP et le 19 SIGKILL) et juste afficher un message à la réception d'un signal. Nous avons suivi la procédure du cours ainsi que quelques recherches et donc parcouru une boucle de 30 (correspondant aux 30 signaux p. 8 du cours) afin de tous leur affecter une *sigaction* permettant de les ignorer.

La dernière partie consistait à dispatcher les deux processus sur deux coeurs différents du nanopi. Pour ce faire nous avons utilisé les fonctions suivantes :

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(1, &set); // set to the core 1
int ret = sched_setaffinity(0, sizeof(set), &set);
```

Et ceci pour les deux processus. Notre application étant très rapidement exécutée et afin de vérifier le bon fonctionnement de ceci, nous avons ajouté dans le code une boucle *while* d'un grand nombre d'itérations effectuant une multiplication et un print afin de visualiser à l'aide de *htop* l'utilisation des coeurs comme le montre la figure suivante :

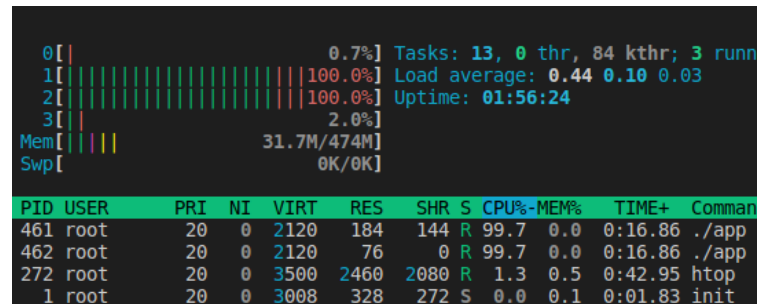


Figure 2.3: Multi Core usage

2.5.2 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.

Nous avons créé un script pour limiter la mémoire et celui-ci se lance de la manière suivante :

```
| ./set_memory_limit <PID> <MEMORY_LIMIT>
```

2.5.3 Exercice 3

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