CSEL

Rapports des TP

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

Fribourg, June 3, 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 :

```
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 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 - Ordonnanceur

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 Doumentation 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 implemente un notifier 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 attente 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 optenir 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 controller 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:

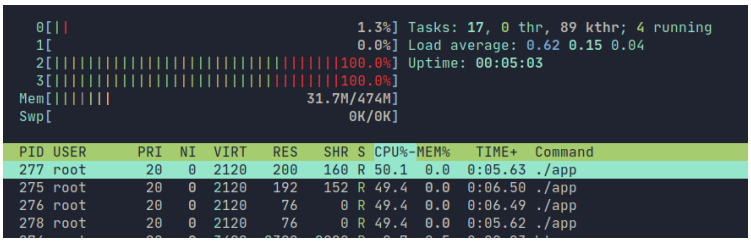

Figure 2.1: Two different shells executing app

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 cœur 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%.

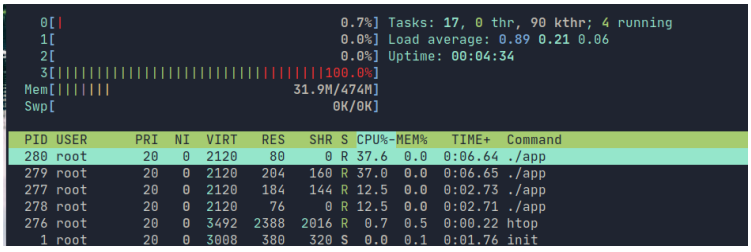

Figure 2.2: 75% and 25% of the CPU on core 4

2.3 Synthèse des connaissances acquises

2.3.1 Non acquis

Nous n'avons pas réellement décelé des connaissances que nous pourrions qualifier de non-acquises. En effet, après avoir suivi le cours et réalisé les TP, cela nous a permis de comprendre la matière ciblée par ce travail pratique.

2.3.2 Acquis, mais à exercer

Le domaine d'utilisation des cgroups afin de limiter les ressources disponibles pour des processus est très vaste et ce travail pratique nous a permis d'en découvrir quelques facettes mais nous pensons que ce sont des aspects que nous pourrions mieux aprofondir avec plus de pratique.

2.3.3 Parfaitement acquis

Comme cité précedemment dans la section des connaissances à exercer, nous ne pourrions pas dire que nous avons parfaitement acquis les connaissances relatives aux cgroups, mais nous les avons cependant bien comprises et nous sommes en mesure de pouvoir affirmer que nous saurions les réutiliser.

2.4 Infos utiles à retenir

La fonction c *strcmp* 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 ou 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 grace 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 :

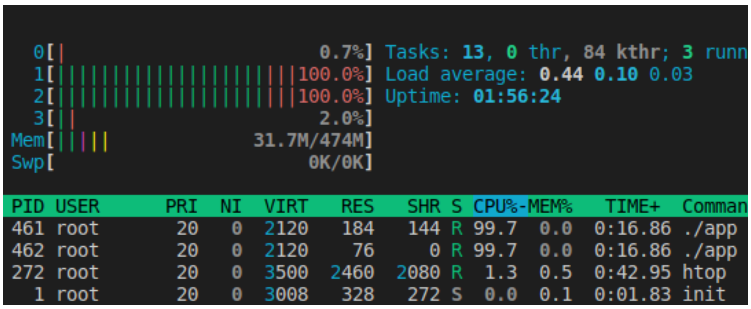

Figure 2.3: Multi Core usage

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 - Optimisation

3.1 Résumé du laboratoire

Ce laboratoire permet de prendre en main l'outil *perf* et de tester quelques unes de ses fonctionnalités. Nous avons préalablement installé cet outil dans le buildroot avec les *binutils* associées puis recompilé celui-ci.

3.2 Réponses aux questions

3.2.1 Exercice 1 - Ce programme contient une erreur triviale qui empêche une utilisation optimale du cache. De quelle erreur s'agit-il ?

Premièrement nous avons remarqué que le programme parcourt le tableau par colonne au lieu de le faire par ligne et comme le cache charge lors d'un accès mémoire les cases adjacentes. Et comme ici on parcourt par colonne cela donne des grands sauts de mémoire et fait que l'on utilise quasiment pas le cache. Nous avons donc modifié le code afin qu'il parcourt le tableau par ligne et comme on peut le voir sur la figure suivante cela a déjà réduit le temps d'exécution d'un facteur de 17.96. (voir screenshots dans la section 3.3.1)

3.2.2 Exercice 1 - Relevez les valeurs du compteur L1-dcache-load-misses pour les deux versions de l'application. Quel facteur constatez-vous entre les deux valeurs ?

Après la modification du code nous avons relancé les deux versions avec la commande:

```
perf stat -e L1-dcache-load-misses ./ex1
```

Qui permet de mesurer le nombre de cache misses qui sont parvenus lors de l'exécution du programme. Les résultats sont les suivants:

```
# perf stat -e L1-dcache-load-misses ./ex1

 Performance counter stats for './ex1':

         406862875      L1-dcache-load-misses

      42.774015312 seconds time elapsed

      42.032729000 seconds user
       0.292383000 seconds sys
```

(a) Sans modification de code

```
# perf stat -e L1-dcache-load-misses ./ex1

 Performance counter stats for './ex1':

           1342006      L1-dcache-load-misses

       2.496445251 seconds time elapsed

       2.232255000 seconds user
       0.221470000 seconds sys
```

(b) Avec modification de code

Figure 3.1: Cache misses avec et sans correction du code

On observe donc un facteur de 303.175 et on remarque que effectivement comme cité à la section 3.2.1, le cache n'est quasiment pas utilisé dans la version initiale du code et la modification permet d'en tirer avantage et de largement diminuer le temps d'exécution.

3.2.3 Mesure de l'impact de perf sur la performance du programme

```
# time perf stat ./ex1

 Performance counter stats for './ex1':

         41749.71 msec task-clock                #    1.000 CPUs utilized
               21      context-switches          #    0.503 /sec
                0      cpu-migrations            #    0.000 /sec
            48868      page-faults               #    1.170 K/sec
      34067454493      cycles                    #    0.816 GHz
       1674211435      instructions              #    0.05  insn per cycle
        270058755      branches                  #    6.469 M/sec
          1066650      branch-misses             #    0.39% of all branches

      41.768301145 seconds time elapsed

      40.996004000 seconds user
       0.335764000 seconds sys


real    0m 41.97s
user    0m 41.00s
sys     0m 0.38s
```

Figure 3.2: Impact de la fonction perf sur la performance du programme

```
# time ./ex1
real    0m 39.15s
user    0m 38.40s
sys     0m 0.30s
```

Figure 3.3: Exécution sans perf

On peut observer ici que la fonction *perf* dans le cas de ce programme rallonge son temps d'exécution de environ 2.8s. On peut donc en déduire qu'il est très utile d'utiliser cette fonction pour mesurer les performances du programme dans une phase de développement et de debug puis de l'enlever dans une phase de déploiement finale.

3.2.4 Explication des éléments

- instructions :
- cache-misses :
- branch-misses :
- L1-dcache-load-misses :
- cpu-migrations :
- context-switches :

3.3 Synthèse des exercices

3.3.1 Exercice 1

Nous avons tout d'abord lancé la commande

```
perf stat ./ex1
```

Qui a donné le résultat suivant :

```
# perf stat ./ex1

 Performance counter stats for './ex1':

         39429.15 msec task-clock                #    1.000 CPUs utilized
               22      context-switches          #    0.558 /sec
                0      cpu-migrations            #    0.000 /sec
            48868      page-faults               #    1.239 K/sec
      32173864412      cycles                    #    0.816 GHz
       1672460593      instructions              #    0.05  insn per cycle
        269903186      branches                  #    6.845 M/sec
          1051958      branch-misses             #    0.39% of all branches

     39.441660560 seconds time elapsed

     38.729619000 seconds user
      0.288050000 seconds sys
```

Figure 3.4: Perf stat without options

On peut observer un nombre de cycles très élevé ainsi qu'un temps d'exécution élevé également. Après la modification du code, nous avons relancé la commande et d'ores et déjà pu observer une nette réduction du temps comme on peut le voir sur la figure suivante :

```
# perf stat ./ex1

 Performance counter stats for './ex1':

          2464.49 msec task-clock                #    0.992 CPUs utilized
               16      context-switches          #    6.492 /sec
                0      cpu-migrations            #    0.000 /sec
            48868      page-faults               #   19.829 K/sec
       2010903905      cycles                    #    0.816 GHz
       1381489331      instructions              #    0.69  insn per cycle
        265238313      branches                  #  107.624 M/sec
           675820      branch-misses             #    0.25% of all branches

      2.483508418 seconds time elapsed

      2.155626000 seconds user
      0.285237000 seconds sys
```

Figure 3.5: Perf stat with code modifications

3.3.2 Exercice 2

3.3.2.1 Analyse du code source

Le programme créé un tableau de tille SIZE et remplis chaque cases avec des entiers aléatoires entre 0 et 511. Après, il fait la somme de toutes les valeurs plus grandes que 255 dans le tableau et cela répété 10000 fois.

3.3.2.2 Mesure du temps d'exécution

```
# perf stat ./ex2
sum=125454290000

 Performance counter stats for './ex2':

         26173.74 msec task-clock                #    0.999 CPUs utilized
               17      context-switches          #    0.650 /sec
                0      cpu-migrations            #    0.000 /sec
               75      page-faults               #    2.865 /sec
      21357647586      cycles                    #    0.816 GHz
      14769643994      instructions              #    0.69  insn per cycle
        988408995      branches                  #   37.763 M/sec
        327882508      branch-misses             #   33.17% of all branches

      26.192219179 seconds time elapsed

      26.113381000 seconds user
       0.007896000 seconds sys
```

Figure 3.6: Mesure du temps d'exécution du code

3.3.2.3 Mesure du temps d'exécution de la version modifiée du code

```
# perf stat ./ex2
sum=125454290000

 Performance counter stats for './ex2':

         23428.47 msec task-clock                #    0.999 CPUs utilized
               17      context-switches          #    0.726 /sec
                0      cpu-migrations            #    0.000 /sec
              108      page-faults               #    4.610 /sec
      19117527969      cycles                    #    0.816 GHz
      14819195509      instructions              #    0.78  insn per cycle
        996698354      branches                  #   42.542 M/sec
           840355      branch-misses             #    0.08% of all branches

     23.443337928 seconds time elapsed

     23.373975000 seconds user
      0.007920000 seconds sys
```

Figure 3.7: Mesure du temps d'exécution de la version modifiée du code

Lors de la réalisation de la boucle (sans optimisation) il est impossible pour le processeur de prédire si il faudra faire le branchement ou non car les données à l'intérieur sont aléatoires. Grâce au tri du tableau (version optimisée) il saura que au début il faut jamais faire le branchement et après il faudra toujours le faire. La prédiction de branchement sera donc très efficace dans ce cas.

3.3.3 Exercice 3

Avec l'analyse du perf report nous remarquons qu'effectivement *std::operator==<char>* (qui est une comparaison entre deux caractères) est la fonction la plus chronophage en termes de temps de cycle. Etant donné que le programme recherche des adresses IP uniques cela a du sens car en effet il est ainsi obligé de comparé les adresses lues aux autres adresses afin de savoir si celle-ci est unique.

3.4 Synthèse des connaissances acquises

3.4.1 Non acquis

3.4.2 Acquis, mais à exercer

3.4.3 Parfaitement acquis

3.5 Feedback