

MA-CSEL Conception système Embarqué Linux

Kirill GOUNDIAEV & Tanguy DIETRICH

June 2, 2023



Contents

1	Système de fichier	3
2	Multiprocessing et Ordonnanceur	6
2.1	Exercice 1 Processus et signaux	6
2.2	Exercice 2 CGroups Limitation mémoire	7
2.3	Exercice 3 CGroups Controle du CPU	9
3	Outils d'analyse de performance pour Linux	12

1 Système de fichier

L'objectif de cet exercice est de reprendre un code faisant clignoter une LED utilisant 100% d'un coeur du processeur et de le modifier afin qu'il utilise le système de fichier virtuel pour contrôler la LED. Au démarrage, la LED clignote à une fréquence de 2Hz. Les boutons nous permettent d'augmenter, diminuer ou réinitialiser la fréquence de clignotement. Les changements de fréquence de la LED sont reportés avec *syslog*.

Pour commencer, nous ouvrons les descripteurs de fichier des boutons d'une manière similaire à celle utilisée pour la LED avec quelques modifications. La fonction ci-dessous permet de configurer les GPIOs des boutons en entrée, avec un détecteur de flanc montant. Elle retourne le descripteur de fichier de la valeur du bouton.

```
1 int open_button(const char *gpio_path, const char *gpio_num){
2     // unexport pin out of sysfs (reinitialization)
3     int f = open(GPIO_UNEXPORT, O_WRONLY);
4     write(f, gpio_num, strlen(gpio_num));
5     close(f);
6
7     // export pin to sysfs
8     f = open(GPIO_EXPORT, O_WRONLY);
9     write(f, gpio_num, strlen(gpio_num));
10    close(f);
11
12    // config pin
13    char *path = malloc(strlen(gpio_path) + strlen("/direction") + 1);
14    strcpy(path, gpio_path);
15    strcat(path, "/direction");
16    f = open(path, O_WRONLY);
17    write(f, "in", 2);
18    close(f);
19    free(path);
20
21    // config edge
22    path = malloc(strlen(gpio_path) + strlen("/edge") + 1);
23    strcpy(path, gpio_path);
24    strcat(path, "/edge");
25    f = open(path, O_WRONLY);
26    write(f, "rising", 6);
27    close(f);
28    free(path);
29
30    // open gpio value attribute
31    path = malloc(strlen(gpio_path) + strlen("/value") + 1);
32    strcpy(path, gpio_path);
33    strcat(path, "/value");
34    f = open(path, O_RDWR);
35    free(path);
36    return f;
37 }
```

Listing 1: Fonction d'ouverture et configuration de bouton

Ces descripteurs de fichier sont placés dans un tableau de structure que nous avons créé, qui stocke l'entité avec son descripteur de fichier. Ce qui nous permet d'accéder plus facilement à tous nos descripteurs. Cette structure comporte également une variable nous permettant de savoir si l'évènement arrive pour la première fois pour éviter le 'faux' premier évènement des boutons.

Nous avons ensuite initialisé le timer¹. Ce timer nous permet de générer un évènement à interval régulier. Nous avons utilisé la fonction *timerfd_settime()* pour configurer le timer. Cette fonction prend en paramètre une structure qui contient le temps avant le premier évènement et le temps entre chaque évènement.

```
1 int open_timer(){
2     int fd = timerfd_create(CLOCK_MONOTONIC, 0);
3     if (fd == -1) {
4         printf("error timerfd_create: %s\n", strerror(errno));
5         return 1;
6     }
7     // default 2Hz => 500ms
8     struct itimerspec new_value;
9     // interval for periodic timer
10    new_value.it_interval.tv_sec = 0;
```

¹https://man7.org/linux/man-pages/man2/timerfd_create.2.html

```

11 new_value.it_interval.tv_nsec = DEFAULT_PERIOD;
12
13 // initial expiration (timer before first expiration)
14 new_value.it_value.tv_sec = 1;
15 new_value.it_value.tv_nsec = 0;
16 if (timerfd_settime(fd, 0, &new_value, NULL) == -1) {
17     printf("error timerfd_settime: %s\n", strerror(errno));
18     return 1;
19 }
20 syslog(LOG_INFO, "frequency: %.5fHz", S_IN_NSEC / ((double)DEFAULT_PERIOD));
21 return fd;
22 }

```

Listing 2: Initialisation du timer

Une fois que nos périphériques sont correctement initialisés, nous créons un *epoll* qui nous permet de multiplexer la surveillance de nos descripteurs de fichier. Nous y ajoutons nos descripteurs de fichier avec l'évènement qui leur sont associés. Dans notre cas, les boutons sont associés à l'évènement *EPOLLET* pour détecter le flanc montant et le timer à l'évènement *EPOLLIN* qui permet de recevoir un évènement à chaque fois que le timer a expiré. La fonction *epoll_ctl* nous permet d'ajouter le contexte à surveiller dans l'*epoll*. Et enfin, la fonction *epoll_wait* nous permet d'attendre l'arrivée d'un évènement sur tous nos descripteurs de fichier virtuel. Voici notre boucle principale qui permet de surveiller les évènements sur nos descripteurs de fichier.

```

1 while(1){
2     // wait for event
3     struct epoll_event event_arrived[5];
4     int nr = epoll_wait(epfd, event_arrived, 5, -1);
5     if (nr == -1){
6         printf("error epoll_wait: %s\n", strerror(errno));
7         return 1;
8     }
9     for (int i = 0; i < nr; i++){
10         my_context *ctx = event_arrived[i].data.ptr;
11         switch (ctx->ev)
12         {
13             case EV_BTN_1: // increase frequency
14             case EV_BTN_2: // resert frequency
15             case EV_BTN_3: // decrease frequency
16                 if (ctx->first_done == 0){
17                     ctx->first_done = 1;
18                     break;
19                 }
20                 button_action(ctx->ev, 0);
21                 break;
22             case EV_TIMER:
23                 read(ctx->fd, &time, sizeof(time));
24                 change_led();
25                 break;
26             default:
27                 printf("error: unknow event\n");
28                 break;
29         }
30     }
31 }

```

Listing 3: Boucle principale

Nous commençons par une attente passive sur l'arrivée d'un évènement sur nos descripteurs de fichier. Une fois que nous avons reçu un évènement, nous regardons quel est le contexte associé à ce dernier. Si c'est un bouton, nous appelons la fonction *button_action* qui permet de modifier la fréquence du timer, si ce n'est pas le premier évènement. Si c'est le timer, nous appelons la fonction *change_led* qui permet de changer l'état de la LED.

La fonction *change_led* écrit successivement 1 et 0 dans le descripteur de fichier de la LED afin de la faire clignoter. La fonction *button_action* quant à elle identifie quel bouton a été pressé et modifie la fréquence du timer en fonction du bouton. Cette fonction prend un paramètre supplémentaire qui nous permet de choisir si nous voulons finir le cycle actuel du timer ou le faire passer directement sur la nouvelle fréquence. Nous gérons également l'underflow et l'overflow de la fréquence.

```

1 void change_led() {
2     static int cpt = 0;
3     cpt = (cpt + 1) % 2;

```

```

4  pwrite(ctx[FD_LED].fd, cpt ? "1" : "0", sizeof("0"), 0);
5  }
6
7  void button_action(enum my_event ev, int wait_for_first_event){
8      static uint64_t period = 0;
9      struct itimerspec current_value;
10     struct itimerspec new_value;
11     uint64_t last_period;
12
13     switch (ev){
14     case EV_BTN_1: // increase frequency
15         period /= 2;
16         if(period == 0){
17             period = 1;
18         }
19         break;
20     case EV_BTN_2: // reset frequency
21         period = DEFAULT_PERIOD;
22         break;
23     case EV_BTN_3: // decrease frequency
24         last_period = period;
25         period *= 2;
26         if(period < last_period){
27             period = last_period;
28         }
29         break;
30     default:
31         printf("error: unused event\n");
32         break;
33     }
34     // interval for new periodic timer
35     new_value.it_interval.tv_sec = period / S_IN_NSEC;
36     new_value.it_interval.tv_nsec = period % S_IN_NSEC;
37
38     // if complete current period
39     if(wait_for_first_event){
40         // time until next expiration
41         if(timerfd_gettime(ctx[EV_TIMER].fd, &current_value) == -1){
42             printf("error timerfd_gettime: %s\n", strerror(errno));
43             return;
44         }
45         new_value.it_value.tv_sec = current_value.it_value.tv_sec;
46         new_value.it_value.tv_nsec = current_value.it_value.tv_nsec+1;
47     }else{// direct reset timer
48         new_value.it_value.tv_sec = 0;
49         new_value.it_value.tv_nsec = 1;// +1 to force to start timer
50     }
51     if (timerfd_settime(ctx[EV_TIMER].fd, 0, &new_value, NULL) == -1) {
52         printf("error timerfd_settime: %s\n", strerror(errno));
53         return;
54     }
55     syslog(LOG_INFO, "frequency: %.5fHz", S_IN_NSEC / (double)period);
56 }

```

Listing 4: Routine LED et bouton

Notre code fonctionne correctement selon les spécifications. Nous avons également ajouté un `syslog`² qui nous permet de voir les changements de fréquence de la LED.

Nous avons pu bien visualiser avec *htop* que notre programme n'utilise plus 100% du CPU si la fréquence n'est pas trop élevée. Quand la période diminue à quelques nanosecondes, les événements arrivent trop rapidement et nous pouvons observer la croissance de l'utilisation du CPU.

Cet exercice était particulièrement intéressant, car il nous a permis d'utiliser le système de fichier virtuel de Linux pour contrôler nos périphériques. Nous avons également pu voir l'importance de la gestion des événements pour la programmation système. La gestion des événements en utilisant *epoll* avec un multiplexage apporte une nouvelle esthétique au code que nous avons bien apprécié.

²https://www.gnu.org/software/libc/manual/html_node/Syslog-Example.html

2 Multiprocessing et Ordonnanceur

2.1 Exercice 1 Processus et signaux

Le but de cet exercice est de mettre en place une communication entre deux processus en utilisant un service Linux tel que socketpair. Le processus enfant envoie des messages au processus parent qui les affiche. Lorsque le processus enfant envoie le message "exit" le programme se termine. Il est aussi demandé de fixer l'affinité des processus afin que le thread enfant effectue ces tâches sur le CPU 1 et le thread parent sur le CPU 0. Il est également demandé d'ignorer les signaux SIGHUP, SIGINT, SIGQUIT, SIGABRT et SIGTERM.

Afin d'ignorer les signaux, nous utilisons la structure sigaction qui permet de modifier le comportement d'un signal. Chaque handler de signal est assigné à SIG_IGN, ce qui permet d'ignorer le signal. Voici un extrait du code :

```
1 struct sigaction sa;
2 sa.sa_handler = SIG_IGN;
3 sigemptyset(&sa.sa_mask);
4 sa.sa_flags = 0;
5 sigaction(SIGHUP, &sa, NULL);
6 sigaction(SIGINT, &sa, NULL);
7 sigaction(SIGQUIT, &sa, NULL);
8 sigaction(SIGABRT, &sa, NULL);
9 sigaction(SIGTERM, &sa, NULL);
```

La création du socketpair se fait simplement avec ce bout de code :

```
1 int fd[2];
2 if(socketpair(PF_LOCAL, SOCK_STREAM, 0, fd) < 0) {
3     perror("socketpair");
4     exit(1);
5 }
```

Le paramètre PF_LOCAL permet de créer un socket local et SOCK_STREAM permet de créer un socket de type TCP. La valeur 0 permet de spécifier le protocole par défaut (ici TCP).

Afin de créer les deux processus, nous utilisons la fonction fork() qui permet de créer un processus enfant identique au processus parent. Voici un extrait du programme après la fonction fork() :

```
1 pid = fork();
2 if (pid == 0) { // child
3     close(fd[PARENTSOCKET]);
4     // set thread affinity
5     if(setAffinity(1) == -1) { perror("sched_setaffinity");}
6     child(fd[CHILDSOCKET]);
7     close(fd[CHILDSOCKET]);
8     exit(0);
9 }
10 else { // parent
11     close(fd[CHILDSOCKET]);
12     // set thread affinity
13     if(setAffinity(0) == -1) { perror("sched_setaffinity");}
14     parent(fd[PARENTSOCKET]);
15     close(fd[PARENTSOCKET]);
16 }
17 // must wait for child to exit
18 // waitpid(pid, NULL, 0);
19 wait(NULL);
```

La valeur de pid retournée par la fonction fork() permet de savoir si le processus est le parent ou l'enfant. Ensuite il faut assigner l'affinité des processus, pour cela nous avons utilisé la fonction sched_setaffinity() qui est appelée dans la fonction setAffinity() que nous avons créée. Pour terminer, il suffit de lancer la fonction correspondante au processus (enfant, ou parent). Et ne pas oublier d'effectuer une attente pour attendre que le processus enfant se termine, afin d'éviter les zombies. La fonction setAffinity que nous avons créée utilise sched_setaffinity, voici le code :

```
1 int setAffinity(int core) {
2     // set thread affinity
3     cpu_set_t cpuset;
4     CPU_ZERO(&cpuset);
5     // set this process to run on core 0
6     CPU_SET(core, &cpuset);
```

```

7 // here 0 mean use the calling process
8 if(sched_setaffinity(0, sizeof(cpuset), &cpuset) == -1) {
9
10     return -1;
11 }
12 return 0;
13 }

```

La communication entre les deux process est assez simple étant donnée, que nous avons simplement 2 descripteurs de fichier, du point de vue de notre programme. C'est comme si nous allions écrire ou lire dans un fichier. Il ne reste plus qu'à lire le descripteur de fichier dans le parent, et d'écrire avec l'enfant.

```

1 void child(int socket) {
2     // get the child socket
3     int cpt = 0;
4     int messageLength;
5     bool exitProcess = false;
6     while (!exitProcess) {
7         messageLength = strlen(messages[cpt])+1;
8         write(socket, messages[cpt], messageLength);
9         if (strcmp(messages[cpt], EXIT_MESSAGE) == 0)
10             {
11                 exitProcess = true;
12             }
13         // juste to give the time to the parent to
14         // read the message
15         // because the parent could read two message
16         // at once and not see the exit message
17         usleep(1000000);
18         cpt = (cpt + 1) % NUM_MESSAGE;
19     }
20     printf("Child exit\r\n");
21 }

```

Listing 5: Processus Enfant

```

1 void parent(int socket) {
2     // get the parent socket
3     char buffer[512];
4     bool exitProcess = false;
5     while (!exitProcess) {
6         if (read(socket, buffer, sizeof(buffer))
7             <= 0) {
8             perror("read");
9             exit(1);
10        }
11        printf("Parent received: %s\r\n", buffer);
12        if (strcmp(buffer, EXIT_MESSAGE) == 0) {
13            exitProcess = true;
14        }
15    }
16    printf("Parent exit\r\n");
17 }

```

Listing 6: Processus Parent

Il faut tout de même faire attention à plusieurs choses, premièrement le parent peut recevoir deux messages en même temps avec la fonction `read()`, il faut donc faire attention à bien lire le message en entier. Il est possible de recevoir "message""exit", avec un zéro terminal entre les deux. Ce qui ferait que le programme ne s'arrêterait pas. Ensuite, la taille du buffer peut ne pas être suffisante pour lire le message en entier, ce qui peut créer un buffer overflow.

Lors de ce TP, nous avons appris à utiliser les socketpairs afin de communiquer entre deux processus. Nous avons aussi appris à ignorer des signaux, mais il aurait aussi été possible de les rediriger vers une fonction dans le but d'effectuer une action. Grâce à ce TP, nous avons appris à gérer un processus (création, affinité, communication, etc.). En utilisant les fonctions `fork()`, `sched_setaffinity()`, `socketpair()`, etc...

Ce labo a été plus compliqué que nous le précédent, car nous avons eu des difficultés pour la réception du message "exit". La solution retenue est simple, mais pourrait poser des problèmes. Il serait mieux de regarder le nombre de bytes reçu, et de parser le message reçu dans différents strings, en utilisant les zéros terminaux.

2.2 Exercice 2 CGroups Limitation mémoire

Dans cet exercice, l'objectif est de montrer que les CGroups permettent de limiter l'utilisation de la mémoire d'un processus. Pour faire cela, nous avons créé un programme qui alloue de la mémoire par block de 2MB, et qui remplit cette mémoire avec des 0. voici ce programme :

```

1 #define NUM_BLOCKS 50
2 #define MEGABYTE 1024 * 1024
3 #define BLOCK_SIZE (2 * MEGABYTE)
4
5 int main(void)
6 {
7     int i;
8     char *ptr[NUM_BLOCKS];
9     printf("Allocating memory...\n");
10    for (i = 0; i < NUM_BLOCKS; i++)
11    {

```

```

12     getchar();
13     printf("Allocating block %d\n", i);
14     ptr[i] = malloc(BLOCK_SIZE * sizeof(char));
15     if (ptr[i] == NULL){exit(EXIT_FAILURE);}
16     memset(ptr[i], 0, BLOCK_SIZE);
17 }
18 for (i = 0; i < NUM_BLOCKS; i++){free(ptr[i]);}
19 return 0;
20 }

```

Listing 7: Allocation de memoire

Afin de voir ce qu'il se passait, nous avons ajouté un print à chaque allocation pour voir à quel moment le programme serait stoppé. Nous avons aussi ajouté un `getchar()` afin de pouvoir faire allouer un nouveau bloc de mémoire.

Avant de lancer ce code, il est nécessaire de créer un groupe de contrôle, et de limiter la mémoire de ce groupe. Afin de simplifier l'utilisation de ce programme, nous avons écrit un script qui permet de faire cela.

```

1 #!/bin/sh
2 mount -t tmpfs none /sys/fs/cgroup
3 mkdir /sys/fs/cgroup/memory
4 mount -t cgroup -o memory memory /sys/fs/cgroup/memory
5 mkdir /sys/fs/cgroup/memory/mem
6 echo $$ > /sys/fs/cgroup/memory/mem/tasks
7 echo 20M > /sys/fs/cgroup/memory/mem/memory.limit_in_bytes
8 ./ex2

```

Ce script monte un système de fichier `tmpfs`, puis monte un groupe de contrôle dans ce système de fichier. Ensuite, il crée un groupe de contrôle, et limite la mémoire de ce groupe à 20MB à tous les processus qui y sont liés. Enfin, il lance le programme qui alloue de la mémoire.

```

1: tanguy@tanguy-Ubuntu: ~
tanguy@tanguy-Ubuntu:~$ ssh root@192.168.0.14
# ./runEx
runEx2.sh runEx3High.sh runEx3Low.sh
# ./runEx2.sh

mkdir: cannot create directory '/sys/fs/cgroup/memory/mem': File exists
Allocating memory...
Allocating block 0
Allocating block 1
Allocating block 2
Allocating block 3
Allocating block 4
Allocating block 5
Allocating block 6
Allocating block 7
Allocating block 8
Allocating block 9
Killed
#

```

Figure 1: exécution de l'exercice 2

Comme on peut le voir sur la figure 1, le programme se fait tuer par le kernel lorsque l'il essaie d'allouer le 10e bloc de mémoire. Ce qui correspond bien à la limite de 20MB que nous avons fixé.

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

`$$` en bash est le pid du processus courant. Donc cette commande permet d'écrire le pid du processus courant dans le fichier `/sys/fs/cgroup/memory/mem/tasks`. Ce qui permet de lier le processus courant au groupe de contrôle `mem`.

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

Lorsque le quota de mémoire est épuisé, le kernel tue le processus qui a essayé d'allouer de la mémoire. Selon cette documentation³. Il est possible de légèrement modifier le comportement en utilisant la commande :

³https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-memory


```
1      echo 1 > /sys/fs/cgroup/memory/mem/memory.oom_control
```

Cette commande désactive le "OOM Killer", si un programme atteint ça limite de mémoire, il sera mis en pause jusqu'à ce qu'il y ait de la mémoire de disponible.

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

Afin de surveiller la mémoire, il y a plusieurs options simples, comme la commande `top`, `htop`. La fonction `free` permet aussi de voir l'état de la mémoire. Elle donne des informations sur la mémoire globale et non sur un processus en particulier. Il est aussi possible de voir l'état de la mémoire d'un processus en utilisant la commande :

```
1      cat /proc/<PID>/stat | awk '{print $23}'
```

Nous ne connaissons pas du tout les CGroups, ce qui a rendu cet exercice très intéressant. Nous avons réussi à faire effectuer l'exercice, mais à un certain moment la limitation de mémoire ne fonctionnait pas. Il faudrait que nous regardions plus en détail comment fonctionne les CGroups, et comment les utiliser, la structure et le nombre de fichiers présents dans le système de fichier `/sys/fs/cgroup/memory/` est assez impressionnant. Toutefois, il est bon de savoir qu'il est possible de limiter la mémoire d'un processus, et de voir comment cela fonctionne.

2.3 Exercice 3 CGroups Controle du CPU

Dans cet exercice, l'objectif est de limiter l'utilisation du CPU par un processus. Pour cela, nous avons créé un programme qui lance 2 processus faisant des calculs afin de charger les CPU. Nous avons utiliser la fonction `fork()` vue dans l'exercice 1 (2.1).

Ce programme nous permet de simplement lancer 2 processus qui vont charger les CPU.

```
1  int main(void)
2  {
3      pid_t pid;
4      pid = fork();
5      if (pid == 0) { // child
6          compute();
7          exit(0);
8      }
9      else { // parent
10         compute();
11     }
12     // must wait for child to exit
13     // waitpid(pid, NULL, 0);
14     wait(NULL);
15     return 0;
16 }
17
```

Listing 8: Processus Enfant

```
1  void compute() {
2      int a = 0;
3      while (1) {
4          // do some random computation
5          a = (a + 1) % 1000000;
6      }
7  }
8
```

Listing 9: Processus Parent

Il reste à configurer les CGroups pour limiter l'utilisation du CPU par les processus. Pour faire cela, les lignes de commande nous ont été fournies.

```
1  #!/bin/sh
2  mkdir /sys/fs/cgroup/cpuset
3  mount -t cgroup -o cpu,cpuset cpuset /sys/fs/cgroup/cpuset
4  mkdir /sys/fs/cgroup/cpuset/high
5  mkdir /sys/fs/cgroup/cpuset/low
6  echo 3 > /sys/fs/cgroup/cpuset/high/cpuset.cpus
7  echo 0 > /sys/fs/cgroup/cpuset/high/cpuset.mems
8  echo 2 > /sys/fs/cgroup/cpuset/low/cpuset.cpus
9  echo 0 > /sys/fs/cgroup/cpuset/low/cpuset.mems
10 echo $$ > /sys/fs/cgroup/cpuset/high/tasks
11 ./ex3
```

Pour simplifier l'exercice, nous avons fait deux scripts bash, un qui lance le processus dans le groupe high, et un autre qui le lance dans le groupe low. La seule différence se trouve à la ligne 10, où nous changeons "high" en "low".

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

La documentation trouvable ici : ⁴ nous indique que `cpuset.cpus` et `cpuset.mems` sont mandatoires pour chaque cgroup. Le fichier `cpuset.cpus` permet de spécifier sur quels CPU le processus peut s'exécuter. Il est possible de spécifier un intervalle de cpu, ou une liste. Le fichier `cpuset.mems` permet de spécifier quels nœuds mémoires le processus peut utiliser.

Il est nécessaire de spécifier sur quelle CPU les processus vont s'exécuter.

2 Ouvrez deux shells distincts et placez une dans le cgroup high et l'autre dans le cgroup low, par exemple :

```
1 $ ssh root@192.168.0.14
2 $ echo $$ > /sys/fs/cgroup/cpuset/low/tasks
```

Lancez ensuite votre application dans chacun des shells. Quel devrait être le bon comportement ? Pouvez-vous le vérifier ?

Le programme appartenant au cgroup high devrait tourner sur le CPU 3, alors que le programme appartenant au groupe low devrait tourner sur le CPU 2. De plus comme un seul CPU est associé à 2 processus, le temps CPU devrait être partagé entre les deux processus. La charge sera donc toujours de 100% sur un CPU, mais partagés à 50% entre les deux processus. Afin de vérifier cela, nous pouvons utiliser la commande `htop`, qui nous permet de voir l'utilisation des CPU. Voici l'exécution du script `runEx3Low.sh` :

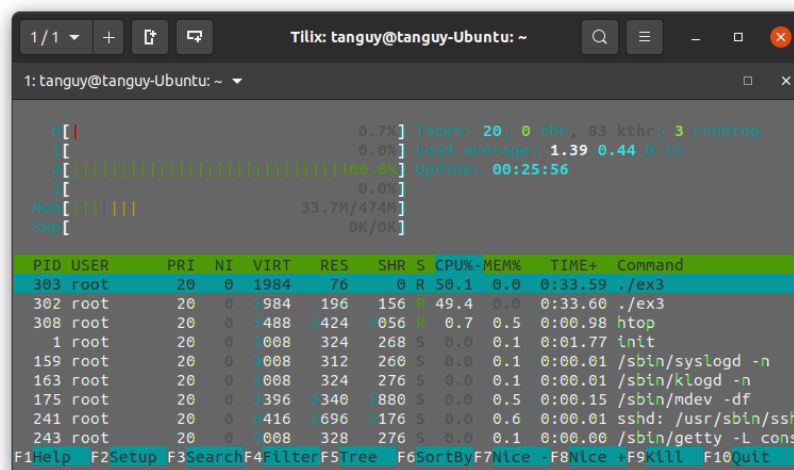


Figure 2: Exécution du script `runEx3Low.sh`

Sans arrêter le script `runEx3Low.sh`, nous pouvons lancer le script `runEx3High.sh` :

⁴https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpuset

```

0[          0.0%] Tasks: 23, 0 thr, 83 kthr, 4 running
1[          0.7%] Load average: 1.87 0.69 0.25
2[          100.0%] Uptime: 00:26:44
3[          100.0%]
Mem[          34.3M/474M]
Swap[          0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR  S  CPU%-MEM%   TIME+  Command
  ---
302 root        20   0 1984   196   156  R  50.8- 0.0  0:57.87 ./ex3
303 root        20   0 1984    76    0  R  49.5- 0.0  0:57.85 ./ex3
314 root        20   0 1984   196   156  R  49.5- 0.0  0:04.03 ./ex3
315 root        20   0 1984    76    0  R  49.5- 0.0  0:04.03 ./ex3
308 root        20   0 1488  1424  1056  R   1.3- 0.5  0:01.40 htop
  1 root        20   0  808   324   268  S   0.0- 0.1  0:01.77 init
159 root        20   0  808   312   260  S   0.0- 0.1  0:00.01 /sbin/syslogd -n
163 root        20   0  808   324   276  S   0.0- 0.1  0:00.01 /sbin/klogd -n
175 root        20   0 1396  1340  1880  S   0.0- 0.5  0:00.15 /sbin/mdev -df
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit

```

Figure 3: Exécution du script runEx3High.sh

En modifiant les lignes 6 et 8, nous pouvons faire en sorte que les processus se répartissent sur les 4 CPU, et non pas seulement 2.

```

1 echo 2-3 > /sys/fs/cgroup/cpuset/high/cpuset.cpus
2 echo 0 > /sys/fs/cgroup/cpuset/high/cpuset.mems
3 echo 0-1 > /sys/fs/cgroup/cpuset/low/cpuset.cpus
4 echo 0 > /sys/fs/cgroup/cpuset/low/cpuset.mems

```

```

0[          100.0%] Tasks: 23, 0 thr, 83 kthr, 4 running
1[          100.0%] Load average: 0.62 0.47 0.34
2[          100.0%] Uptime: 00:33:56
3[          100.0%]
Mem[          34.2M/474M]
Swap[          0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR  S  CPU%-MEM%   TIME+  Command
  ---
323 root        20   0 1984   188   144  R  100.- 0.0  0:13.99 ./ex3
331 root        20   0 1984    76    0  R  100.- 0.0  0:10.86 ./ex3
324 root        20   0 1984    80    0  R  99.3- 0.0  0:13.99 ./ex3
330 root        20   0 1984   196   156  R  98.7- 0.0  0:10.75 ./ex3
317 root        20   0 1500  1404  1020  R   0.7- 0.5  0:00.21 htop
  1 root        20   0  808   324   268  S   0.0- 0.1  0:01.77 init
159 root        20   0  808   312   260  S   0.0- 0.1  0:00.01 /sbin/syslogd -n
163 root        20   0  808   324   276  S   0.0- 0.1  0:00.01 /sbin/klogd -n
175 root        20   0 1396  1340  1880  S   0.0- 0.5  0:00.15 /sbin/mdev -df
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit

```

Figure 4: Exécution des deux scripts avec 4 CPU

3 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 ?

Ici nous voulons faire tourner tous nos programmes sur le CPU 4, donc un peut écrire :

```

1 echo 3 > /sys/fs/cgroup/cpuset/high/cpuset.cpus
2 echo 0 > /sys/fs/cgroup/cpuset/high/cpuset.mems
3 echo 3 > /sys/fs/cgroup/cpuset/low/cpuset.cpus
4 echo 0 > /sys/fs/cgroup/cpuset/low/cpuset.mems
5 # set the cpu.shares
6 echo 768 > /sys/fs/cgroup/cpuset/high/cpu.shares

```

```

7 echo 256 > /sys/fs/cgroup/cpuset/low/cpu.shares
8 # change high to low for the other script
9 echo $$ > /sys/fs/cgroup/cpuset/high/tasks
10 ./ex3

```

Pour ce test, nous avons réutilisé le programme qui génère deux processus, nous aurons donc 2 processus qui se partageront 75% du temps CPU, et deux autres pour les 25% restant.

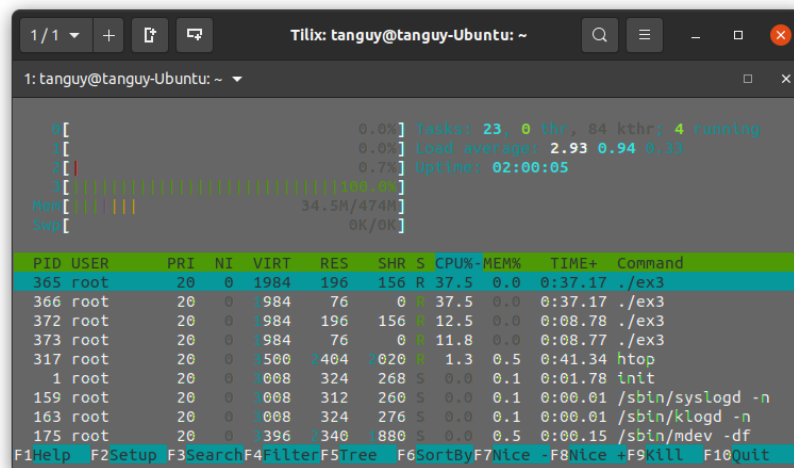


Figure 5: Exécution des deux scripts avec 4 CPU

Nous pouvons observer sur la figure 5 que les deux processus ont 37.5% du temps CPU, et les deux autres 12.5%, ce qui fait bien 75% et 25%.

Ce TP était très proche du précédent, mais nous avons appris à gérer les ressources CPU allouées à un processus et comment les partager entre plusieurs processus.

3 Outils d'analyse de performance pour Linux

Dans cette partie, nous allons nous familiariser avec l'outil perf de Linux. Cet outil permet de mesurer les performances d'un programme. Il est possible de mesurer le temps d'exécution d'un programme, le nombre de cycles d'horloge, le nombre d'instructions, le nombre de caches miss, etc.

Tout d'abord, nous commençons par ajouter binutils dans buildroot :

```

1 $ cd /buildroot
2 $ make menuconfig
3 $ Target packages -> Development tools -> binutils
4 $ Target packages -> Development tools -> binutils binaries
5 $ make -j4

```

Puis nous installons le nouveau buildroot en prenant garde de ne pas effacer la configuration dans /etc/fstab, en utilisant les commandes fournies dans le cours. Après avoir vérifié que perf soit bien installée, nous avons lancé la commande perf sur l'exercice 1 :

```
1 $ perf stat ./ex1
```

Et nous avons obtenu le résultat suivant :

```

1:tanguy@tanguy-Ubuntu: ~
# perf stat ./ex1

Performance counter stats for './ex1':

      42051.28 msec task-clock           #    1.000 CPUs utilized
         21      context-switches       #    0.499 /sec
          0      cpu-migrations          #    0.000 /sec
      48868      page-faults            #    1.162 K/sec
 34313523725      cycles                 #    0.816 GHz
 1675693617      instructions           #    0.05 insn per cycle
 270346594      branches                #    6.429 M/sec
 1091462      branch-misses             #    0.40% of all branches

      42.068733062 seconds time elapsed

      41.257007000 seconds user
       0.339370000 seconds sys

#

```

Figure 6: Exécution des deux scripts avec 4 CPU

- 1 Ce programme contient une erreur triviale qui empêche une utilisation optimale du cache. De quelle erreur s'agit-il ? L'erreur est que le tableau est parcouru en entier entre chaque incrément, si tout le tableau rentrait dans la cache, cela pourrait ne pas poser de problème. Mais ici, le tableau est trop grand pour rentrer dans la cache, et donc à chaque incrément, le cache est vidé, et le tableau est rechargé dans la cache, ce qui prend du temps. Une modification simple pour rendre ce code plus efficace serait de déplacer la boucle d'incrément à l'intérieur des deux boucles.

```

1 for (k = 0; k < 10; k++)
2 {
3     for (i = 0; i < SIZE; i++)
4     {
5         for (j = 0; j < SIZE; j++)
6         {
7             array[j][i]++;
8         }
9     }
10 }
11

```

Listing 10: Code fournis

```

1 for (i = 0; i < SIZE; i++)
2 {
3     for (j = 0; j < SIZE; j++)
4     {
5         for (k = 0; k < 10; k++)
6         {
7             // memory access already in cache
8             array[j][i]++;
9         }
10    }
11 }
12

```

Listing 11: Code modifié

Afin d'avoir un point de comparaison, nous avons lancé le programme de base avec la commande perf :

```
1 $ perf stat -e cache-misses ./ex1
```

Et nous avons obtenu le résultat suivant :

```

1:tanguy@tanguy-Ubuntu: ~
# perf stat -e cache-misses ./ex1

Performance counter stats for './ex1':

 407143880      cache-misses

      41.736347020 seconds time elapsed

      40.954292000 seconds user
       0.323720000 seconds sys

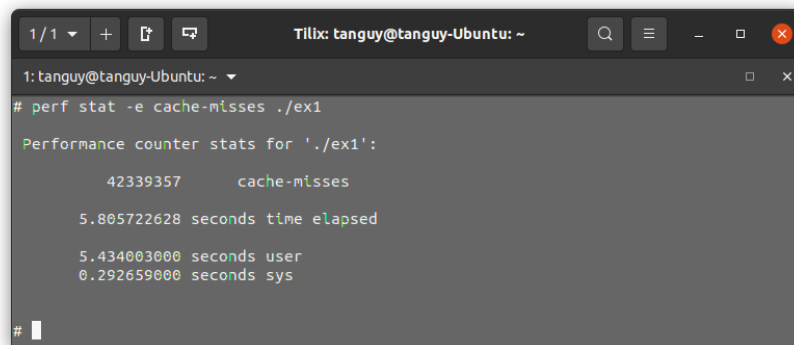
#

```

Figure 7: Exécution des deux scripts avec 4 CPU

2 Corrigez l'erreur, recompilez et mesurez à nouveau le temps d'exécution (soit avec perf stat, soit avec la commande time). Quelle amélioration constatez-vous ?

Puis nous avons relancé le programme avec la correction :



```

1/1 + [?] [?] Tilix: tanguy@tanguy-Ubuntu: ~
1:tanguy@tanguy-Ubuntu: ~
# perf stat -e cache-misses ./ex1
Performance counter stats for './ex1':

      42339357      cache-misses

      5.805722628 seconds time elapsed

      5.434003000 seconds user
      0.292659000 seconds sys

#

```

Figure 8: Exécution des deux scripts avec 4 CPU

Le temps passe de 41.7s à 5.8s et le nombre de caches miss de 407143880 à 42339357. Il y a 9.6x moins de cache miss, et le temps d'exécution est 7.2x plus rapide. Ceci en modifiant une simple boucle.

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

	Sans correction	Avec correction
L1-dcache-load-misses	406895610	42289308
cache-misses	407143880	42339357

Les résultats sont similaires aux deux résultats précédents. Il y a 9.6x moins de cache miss et le temps d'exécution est 7.2x plus rapide.

Décrivez brièvement ce que sont les évènements suivants :

- a instructions** : le nombre d'instructions exécuté
- b cache-misses** : le nombre de caches miss
- c branch-misses** : le nombre de branche miss
- d L1-dcache-load-misses** : le nombre de caches miss de niveau 1
- e cpu-migrations** : le nombre de migrations de processus
- f context-switches** : le nombre de changements de contexte

4 Lors de la présentation de l'outil perf, on a vu que celui-ci permettait de profiler une application avec très peu d'impacts sur les performances. En utilisant la commande time, mesurez le temps d'exécution de notre application ex1 avec et sans la commande perf stat

L'exécution de la commande perf stat a pris 5.477 secondes et l'exécution de la commande time a pris 5.07 secondes. Mais il faudrait effectuer plus de lancement afin d'obtenir une moyenne plus précise. Nous pouvons observer que le programme prend 0.4s de plus avec perf stat, ce qui est très peu, et donc nous pouvons dire que perf stat a très peu d'impact sur les performances.

Exercice 2 :

1 Décrivez en quelques mots ce que fait ce programme.

Le programme commence par remplir un tableau de 65536 entiers avec des valeurs aléatoires entre 0 et 512 (en réalité les valeurs ne sont pas aléatoires, car la seed est toujours la même, la somme sera toujours la même). Ensuite, deux boucles for effectuent 10000 fois la somme des valeurs plus grandes ou égales à 256.

2 Mesurez le temps d'exécution

Voici le résultat de l'exécution du programme :

```

1 # time ./ex2
2 sum=125454290000
3 real    0m 26.19s
4 user    0m 26.11s
5 sys     0m 0.00s

```

Le programme prend 26,19 secondes pour s'exécuter. Dans le prochain test, nous allons essayer de trier le tableau en ajoutant ces lignes de code :

```

1 static int compare (const void* a, const void* b)
2 {
3     return *(short*)a - *(short*)b;
4 }
5 qsort(array, SIZE, sizeof(short), compare);

```

Listing 12: Code modifié

Le temps est à présent de 23.45 secondes, nous avons gagné 2.74 secondes.

3 À l'aide de l'outil perf et de sa sous-commande stat, en utilisant différents compteurs déterminez pourquoi le programme modifié s'exécute plus rapidement.

D'après nos mesures, le gain de vitesse vient des prédictions de branchement. Voici les résultats de perf stat : (les autres résultats d'événement sont similaires).

```

32788425s dans les deux exécutions9. branch-misses # 33.17% of all branches
842261 branch-misses # 0.08% of all branches

```

Le fait que le tableau soit trié permet d'avoir de meilleure prédiction de branchement, donc le pipeline est moins souvent cassé.

Exercice 3 :

1 Compilez l'application et profilez l'application avec perf record :

Nous exécutons la commande suivante :

```

1 perf record --call-graph dwarf -e cpu-clock -F 75 \
2 ./read-apache-logs access_log_NASA_Jul95_samples

```

Un fichier report.data est généré, que nous analysons avec la commande suivante :

```
1 perf report --no-children --demangle
```

2 Avec les instructions précédentes, déterminez quelle fonction de notre application fait (indirectement) appel à std::operator== <char>.

Le problème vient de cette fonction :

```

1 bool HostCounter::isNewHost(std::string hostname)
2 {
3     return std::find(myHosts.begin(), myHosts.end(), hostname) == myHosts.end();
4 }

```

Listing 13: Extrait de hostcounter.cpp

3 Maintenant que vous savez quelle fonction utilise le plus de ressources CPU, trouvez une optimisation du code permettant de réduire drastiquement le temps d'exécution (vous devriez arriver à quelques dixièmes de secondes pour le fichier sample).

Nous modifions le code afin d'utiliser la structure set, au lieu de vector et testons le temps d'exécution avec la commande time. Le temps est à présent de 1.49s, contre 2 minutes et 18 secondes avant la modification. En exécutant à nouveau la commande perf record, nous pouvons voir que les fonctions memcmp et cfree, sont les plus utilisées.

4 Décrivez comment devrait-on procéder pour mesurer la latence et la gigue d'interruption, ceci aussi bien au niveau du noyau (kernel space) que de l'application (user space)

Dans le cas d'un système embarqué avec une routine d'interruption assignée à une pin, nous pouvons

envoyer un signal carré sur la pin d'interruption. Puis inverser l'état d'une pin de sortie, dans la routine d'interruption. Ensuite à l'aide d'un oscilloscope, nous pouvons mesurer la latence et la gigue d'interruption. En effectuant des moyennes et en calculant l'écart type.

Ce labo était particulièrement intéressant, nous n'avions jamais pris le temps d'observer les prédictions de branche, les caches miss, etc. Avant cela nous utilisions uniquement `time`, et les fonctions de mesure de temps fournis en C afin de tester les performances de nos programmes. À l'avenir nous utiliserons `perf stat` et `perf record` afin de tester les performances de nos programmes.