

MA-CSEL Conception systeme Embarqué Linux

Kirill GOUNDIAEV & Tanguy DIETRICH

May 29, 2023



Contents

1	Objectif	3
2	Programmation Systeme	3
2.1	Systeme de fichier	3
2.2	Multiprocessing et Ordonnanceur	3
2.2.1	Exercice 1 Processus et signaux	3
2.2.2	Exercice 2 CGroups Limitation memoire	4
2.2.3	Exercice 3 CGroups Controle du CPU	6
3	Conclusion	7

1 Objectif

This is my link: ¹.

2 Programmation Systeme

2.1 Systeme de fichier

2.2 Multiprocessing et Ordonnanceur

2.2.1 Exercice 1 Processus et signaux

Le but de cet exercice etait de mettre en place une communication entre deux processus en utilisant un service linux tel que socketpair. Le processus enfant devait envoyer des message au processus parent qui devait les afficher. Lorsque le process enfant envoie le message "exit" le programme doit se terminer. Il etait aussi de demander de fixer l'affinité des processus afin que le thread enfant effectue ces taches sur le CPU 1 et le thread parent sur le CPU 0. Ainsi que d'ignorer les signaux SIGHUP, SIGINT, SIGQUIT, SIGABRT et SIGTERM.

Afin d'ignorer les signaux, nous avons utiliser la structure sigaction qui permet de modifier le comportement d'un signal. Chaque handler de signal a été assigné a 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 parametre PF_LOCAL permet de créer un socket local, SOCK_STREAM permet de créer un socket de type TCP. La valeur 0 permet de spécifier le protocole par défaut (ici TCP).

Afin de crée les deux process, nous avons utiliser la fonction fork() qui permet de crée un processus enfant identique au processus parent. Voici un extrait du programme apres 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 retourner 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 utiliser la fonction sched_setaffinity()

¹<http://www.latex-tutorial.com>

qui est appelé dans la fonction `setAffinity()` que nous avons créer. Pour terminer il ne reste qu'à lancer la fonction correspondante au processus (enfant, ou parent). Et ne pas oublier d'effectuer un `wait` pour attendre que le processus enfant se termine. Pour éviter les zombies. La fonction `setAffinity` que nous avons créer 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 descripteur de fichier, du point de vue de notre programme, c'est comme si nous allions écrire/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 1: 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 2: 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. Car on pourrait 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 pourrait ne pas être suffisante pour lire le message en entier, ce qui pourrait créer un buffer overflow.

Lors de ce TP, nous avons appris à utiliser les `socketpair` 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 pensions, car nous avons eu de la difficulté pour la réception du message "exit". La solution que nous avons choisie est simple mais pourrait poser des problèmes. Il serait mieux de regarder le nombre de bytes reçus, et de parser le message reçu dans différents strings, en utilisant les zéros terminaux.

2.2.2 Exercice 2 CGroups Limitation mémoire

Dans cet exercice, l'objectif était 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 blocs de 2MB,

et qui remplit cette memoire 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 3: Allocation de memoire

Afin de voir ce qu'il se passait, nous avons ajouter un print a chaque allocation pour voir a quel moment le programme serait stoppé. Nous avons aussi ajouter un getchar() afin de pouvoir faire allouer un nouveau block de memoire.

avant de lancer ce code, il est necessaire de créer un groupe de controle, et de limiter la memoire 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 systeme de fichier tmpfs, puis monte un groupe de controle sur ce systeme de fichier. Ensuite il cree un groupe de controle, et limite la memoire de ce groupe a 20MB a tout les processus qui y sont lié. Enfin il lance le programme qui alloue de la memoire.

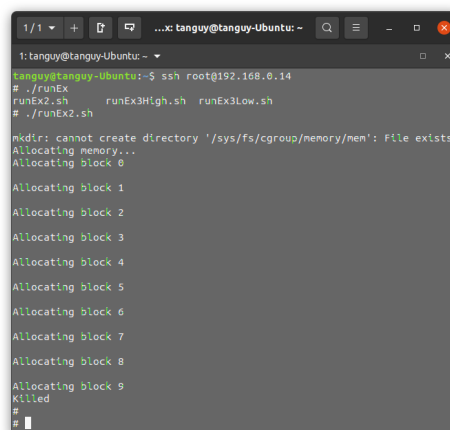


Figure 1: execution de l'exercice 2

Comme on peut le voir sur la figure 1, le programme se fait tuer par le kernel lorsque l'on essaie d'allouer le 10eme block de memoire. Ce qui correspond bien a 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 controle 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 memoire est epuisé, le kernel tue le processus qui a essayé d'allouer de la memoire. Selon cette documentation ². Il est possible de legerement modifier le comportement en utilisant le commande :

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

Cette commande desactive le "OOM Killer", si un programme atteint ça limite de memoire, il sera mis en pause j'usqu'a ce qu'il y ai de la memoire de disponible.

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

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

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

Nous ne conaission pas du tout les CGroups, ce qui a rendu cet exercice tres interessant. Nous avons réussi a faire effectuer l'exercice, mais a un certain moment la limitation de memoire ne fonctionnait pas. Il faudra que l'on regarde plus en detail comment fonctionne les CGroups, et comment les utiliser, la structure et le nombre de fichier present dans le systeme de fichier `/sys/fs/cgroup/memory/` est assez impressionnant. Toutefois, il est bon de savoir qu'il est possible de limiter la memoire d'un processus, et de voir comment cela fonctionne.

2.2.3 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 crée un programme qui lance 2 processus faisant des calcul afin de charger les CPU. Nous avons utiliser la fonction `fork()` vue dans l'exercice 1 (2.2.1).

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 4: 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 5: Processus Parent

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

```
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
```

²https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-memory

```
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 script 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 : ³ 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 nœud mémoire 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 ?

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 ?

3 Conclusion

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