UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Master M1 Informatique, CSE - TP Processus sous Linux/UNIX Année 2017-2018

Résumé

Cette fiche ne sera traitée que cette semaine. Vous devez fournir un compte-rendu de votre travail sur la plate-forme Moodle. Les objectifs de cette séance de TD/TP sont les suivants :
— Manipulation des processus et tubes sous Linux
— Primitives `fork` et `pipe`

Les exercices portant la mention "Travail personnel" figurant en fin de feuille de TD sont optionnels : le temps imparti à une séance de TD ne permettra sans doute pas de les traiter durant celle-ci et les évaluations de l'UE ne porteront pas sur les notions qu'ils permettent d'aborder. Pour les curieux qui s'y attaquent, si vous ne parvenez pas à les traiter seul, n'hésitez pas à demander de l'aide à vos enseignants.

1 Création de processus

L'appel système `fork()` crée un nouveau processus par clonage du processus appelant : le nouveau processus créé est appelé *fils*, tandis que le processus appelant est appelé *père*. En vertu du clonage effectué, à l'issue du `fork()`, les deux processus exécutent le même programme mais se distinguent par la valeur de retour du `fork()` :
— dans le père : le numéro (pid) du processus fils est retourné
— dans le fils : la valeur 0
Bien entendu, les processus se distinguent également au niveau du système par des numéros (PID) différents. En cas d'échec (table des processus pleine), aucun processus n'est créé par `fork()`, et la valeur -1 est renvoyée au processus appelant. Le comportement de l'appel système `fork()` est illustré par la figure 1

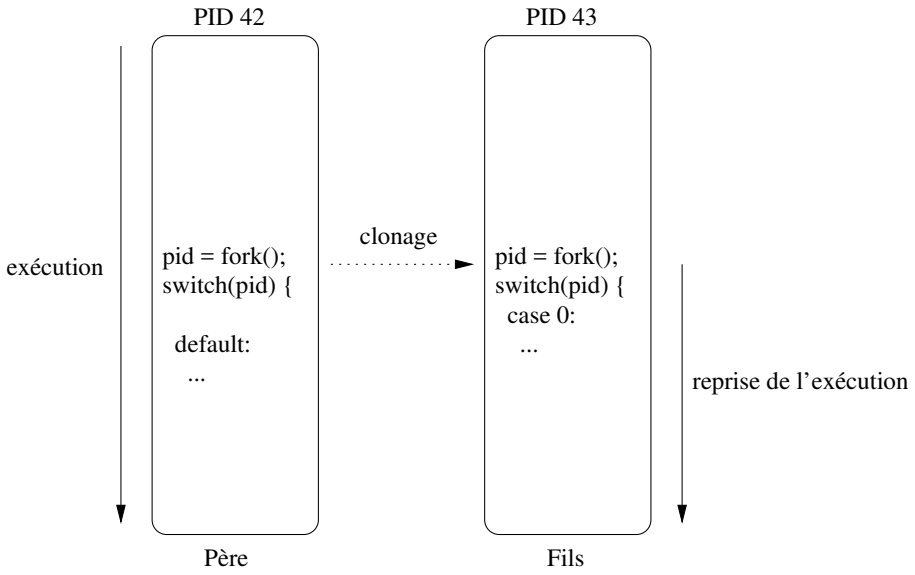

FIGURE 1 – Exemple de création d'un processus.

A titre d'exemple, le processus exécutant le programme suivant crée un nouveau processus, puis chaque processus affiche les informations le concernant :

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    pid_t pid;

    pid = fork();
    switch (pid) {
      case -1:
        fprintf(stderr, "Erreur de fork\n");
        exit(1);
      case 0:
        printf("Je suis le fils, mon pid est %d, celui de mon pere %d\n",
               getpid(), getppid());
        break;
      default:
        printf("Je suis le pere, mon pid est %d, celui de mon pere %d\n",
               getpid(), getppid());
    }
    return 0;
}
```

Question .1. *Combien de lignes affche le programme suivant ?*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;

    for (i=0; i<3; i++)
        fork();
    printf("Je suis le processus %d, mon pere est %d\n", getpid(), getppid());
    return 0;
}
```

Question .2. *Ecrire un programme qui crée N processus en chaîne, N étant donné en ligne de commande (le* 1*er processus en crée un* 2*ième qui en crée un* 3*ième et ainsi de suite).*

Question .3. *On souhaite écrire un programme où sont lancés n processus qui vont chacun générer une nombre aléatoire et l'afficher à l'écran. Voici un exemple d'exécution avec* 6 *processus.*

```
$ ring1 6
processus pid 25387 node 2 val = 1430826605
processus pid 25388 node 3 val = 48523501
processus pid 25389 node 4 val = 822619539
processus pid 25390 node 5 val = 1591287596
processus pid 25385 node 0 val = 2047288621
processus pid 25386 node 1 val = 1731323093
$
```

6 *processus sont donc lancés. Le nombre de processus est passé en paramètre du programme. Les processus sont identifiés par leur* pid *(retourné par le* `fork`*), un numéro de nœud (numéro d'ordre dans la création) et un nombre aléatoire généré par la fonction* `random()`*. Pour avoir plus d'informations sur cette fonction, vous pouvez les obtenir avec le manuel (*`man random`*). Prenez garde à ce que les séquences de nombres aléatoires générées par chaque processus soient différentes (voir la fonction* `srandom()`*).*

2 Tubes

Un tube (en anglais *pipe*) est un mécanisme système particulier qui sert de canal de communication entre deux processus. Un tube possède un flot d'entrée et un flot de sortie, et les données sont lues sur le flot de sortie dans l'ordre selon lequel elles sont écrites dans le flot d'entrée. Un tube peut

être créé au niveau du langage de commande par l'opérateur `|`, ou au niveau des appels système par la primitive `pipe()`. Le programme suivant est un exemple d'utilisation des tubes :

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    pid_t pid;
    int tube[2];
    long entier;

    if (pipe(tube) == -1) {
        fprintf(stderr, "Erreur de création du tube\n");
        exit(1);
    }
    pid = fork();
    switch (pid) {
      case -1:
        fprintf(stderr, "Erreur de fork\n");
        exit(2);
      case 0:
        close(tube[1]);
        read(tube[0], &entier, sizeof(entier));
        printf("Je suis le fils, j'ai lu l'entier %ld dans le tube\n", entier);
        close(tube[0]);
        break;
      default:
        close(tube[0]);
        srandom(pid);
        entier = random();
        printf("Je suis le père, j'envoie l'entier %ld à mon fils\n", entier);
        write(tube[1], &entier, sizeof(entier));
        close(tube[1]);
    }
    return 0;
}
```

Le comportement de ce programme est illustré par la figure 2. Nous pouvons remarquer plusieurs choses :

— le tube est un objet indépendant des processus, créé et géré par le système d'exploitation. Les processus ont connaissance du tube via des points d'accès en lecture et en écriture (appelés descripteurs de fichier), dont la valeur est initialisée par l'appel système `pipe()` lors de la création du tube. Plus précisément, un descripteur de fichier est une structure de donnée maintenue par le noyau dans son espace mémoire et permettant de conserver des informations sur l'état de l'accès à un objet correspondant à une séquence d'octets (fichier, tube, ...). Du point de vue de l'utilisateur il est identifié par une valeur entière attribuée par le système.

— le tube doit être créé avant le `fork()`, qui copie les descripteurs de fichiers ouverts lors du clonage. Dans le cas contraire, les processus n'ont pas moyen de partager les points d'accès au tube.

— après le `fork()`, les deux processus connaissent les deux points d'accès au tube. Ils peuvent donc tous les deux lire et écrire dans le tube. Néanmoins, le tube est un canal de communication monodirectionnel : si les deux processus écrivent dans le tube, leur données vont se mélanger et être inexploitables.

— chaque processus doit fermer les parties du tube qu'il n'utilise pas. Cela permet de libérer les ressources système associées au point d'accès fermé et cela permet également de propager la fin de fichier dans le tube lorsque le processus qui écrit aura terminé et fermé l'entrée du tube (la fin de fichier est propagée dans un tube lorsque toutes ses entrées sont fermées).

— les accès au tube se font via des primitives de lecture et d'écriture dites de bas niveau : `read()` et `write`, qui permettent respectivement de lire/d'écrire une suite d'octets depuis/vers un point d'accès donné par un descripteur de fichier. Les autres fonctions d'entrées/sorties (`fprintf`, `fscanf`, ...) que vous connaissez sont écrites en utilisant ces fonctions d'entrées sortie de bas niveau et y ajoutent une mise en mémoire tampon (pour limiter le nombre d'appels système

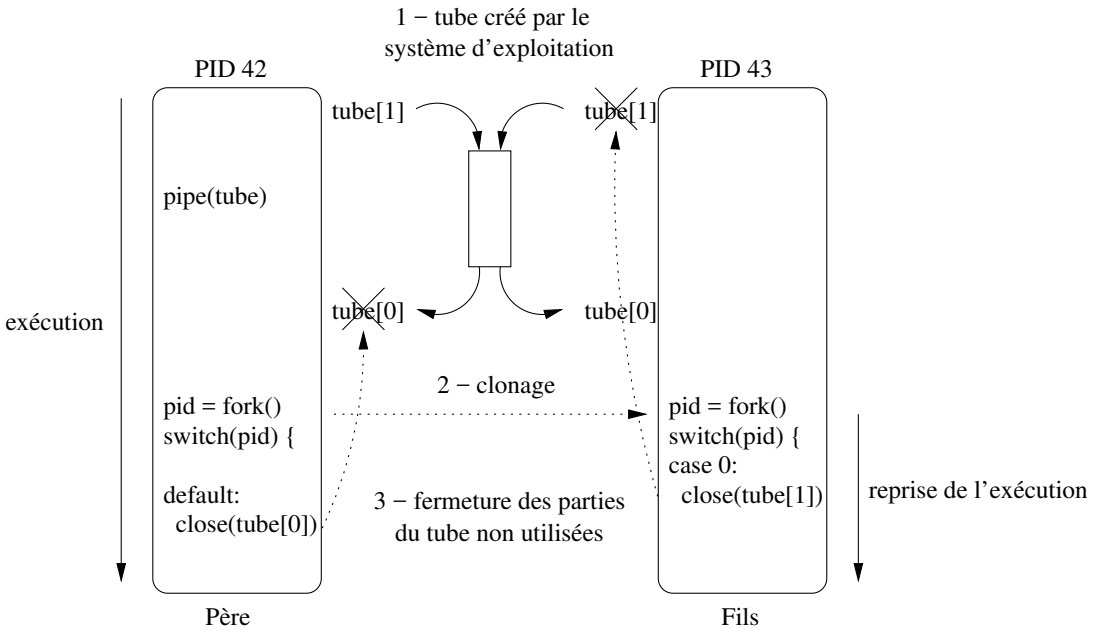

FIGURE 2 – Exemple d'utilisation d'un tube.

réalisés) et une structuration des données selon les types de base. Ces entrées/sorties, dites de haut niveau, utilisent une structure de type `FILE` nommée flux qui référence (entre autres) le descripteur de bas niveau associé. Les plus curieux pourront consulter le manuel de la fonction `fdopen` qui permet d'utiliser les entrées/sorties de plus haut niveau sur un canal de communication ouvert avec des primitives de bas niveau.

Question .4. *Ecrivez un programme qui crée un nouveau processus, puis effectue un* ping-pong *entre les deux processus : le père devra envoyer un message au fils par l'intermédiaire d'un tube et le fils devra renvoyer ce message au père par l'intermédiaire d'un autre tube après l'avoir reçu.*

Question .5. *Ecrivez un programme qui crée un nouveau processus, et dans lequel le père envoie, par l'intermédiaire d'un tube, le contenu d'un fichier (dont le nom sera transmis sur la ligne de commande) au fils, qui sera chargé d'afficher ce contenu.*

Question .6. *Reprenez le programme permettant de créer une chaîne de processus sur N générations et ajoutez y des tubes : les N processus devront chacun être connectés à leur père par l'intermédiaire d'un tube. Le processus initial devra envoyer une valeur générée aléatoirement à son fil, qui devra la retransmettre à son fils, et ainsi de suite. Tous les processus devront afficher la valeur générée.*

Question .7. *Reprenez la question précédente en la modifiant de la manière suivante :*

- *les processus devront être connectés en anneau (le processus N, la dernière génération, devra pouvoir envoyer une valeur au processus initial par l'intermédiaire d'un tube).*
- *le processus initial devra générer une valeur aléatoire et l'envoyer à son fils.*
- *chaque autre processus, devra générer une valeur aléatoire distincte et propager au processus suivant dans l'anneau le maximum entre la valeur qu'il a reçu de son père et la valeur qu'il a généré.*
- *à la réception de la valeur envoyée par le processus N, le processus initial devra déterminer le maximum de toutes les valeurs générées.*

```
$ ring2 6
processus pid 26603 node 1 val = 982695543
processus pid 26604 node 2 val = 679092432
processus pid 26605 node 3 val = 1441867048
processus pid 26606 node 4 val = 1135529882
processus pid 26607 node 5 val = 1906462017
processus pid 26602 node 0 val = 208930720
the winner is 1906462017 pid 26607 node 5
$
```

Question .8. *Pour cet exercice, on souhaite maintenant informer l'ensemble des processus du nom du gagnant. Sur l'exercice précédent, seuls les processus* 0 *et* 5 *connaissaient l'identité du processus gagnant. Pour cet exercice, le processus* 0 *doit faire passer sur l'anneau l'identité du gagnant. Le gagnant saura ainsi qu'il était le gagnant !*

```
$ ring3 6
processus pid 27739 node 1 val = 161183994
processus pid 27740 node 2 val = 925511728
processus pid 27741 node 3 val = 1709276223
processus pid 27742 node 4 val = 1410188147
processus pid 27743 node 5 val = 1099100972
processus pid 27738 node 0 val = 1532950038
Node 0 the winner is 1709276223 pid 27741 node 3
Node 1 the winner is 1709276223 pid 27741 node 3
Node 2 the winner is 1709276223 pid 27741 node 3
Node 3 the winner is 1709276223 pid 27741 node 3
Node 4 the winner is 1709276223 pid 27741 node 3
Node 5 the winner is 1709276223 pid 27741 node 3
$
```

3 Travail personnel : Vous n'aimez pas les tubes, vous allez peut-être aimer les segments de mémoire partagée !

Vous avez peut-être eu quelques difficultés à mettre en place l'anneau qui allait interconnecter vos processus. Nous vous proposons maintenant de mettre en place cette communication entre les processus par un segment de mémoire partagée.

Un segment de mémoire partagée est une zone mémoire qui va pouvoir être partagée entre plusieurs processus. Dès qu'un processus écrit dans cette zone mémoire, la modification va être automatiquement visible par les autres processus de l'application.

Pour créer un segment, il faut d'abord demander au système d'allouer un espace mémoire partagé (opération `shmget`). Le système nous fournit alors l'identifiant de cet espace mais celui-ci n'est pas accessible depuis l'espace mémoire du processus. Il faut donc attacher le segment à l'espace mémoire du processus (opération `shmat`). Ce mécanisme en deux temps permet, au besoin, de créer et partager un segment après le démarrage des processus : il suffit qu'un processus crée le segment et diffuse l'identifiant correspondant pour que tous les processus puissent l'attacher à leur espace mémoire.

Pour refaire les exercices de ce TD, nous pourrons nous limiter à une utilisation plus simple : le premier processus crée et attache le segment à son espace mémoire avant de créer les autres processus. Le segment est alors visible par tout le monde. Une fois l'utilisation du segment terminée, il ne faut pas oublier de le détacher (opération `shmdt`) et surtout de le libérer afin que le système puisse réccupérer la mémoire associée (opération `shmctl` avec la commande `IPC_RMID` passée en paramètre).

Voici un extrait de code illustrant la création et la destruction d'un segment de mémoire partagée :

```
void handle_error(char *function_name, int return_value) {
  char message[128] = "Error in function ";
  if (return_value == -1) {
    perror(strcat(message, function_name));
    exit(1);
  }
}

void *create_shared_memory(size_t size, int *shmid) {
  void *ptr;
  *shmid = shmget(IPC_PRIVATE, size, IPC_CREAT | 0666);
  handle_error("shmget",*shmid);
  ptr = shmat(*shmid, NULL, 0);
  handle_error("shmat", (int) ptr);
  return ptr;
}

void delete_shared_memory(void *ptr, int shmid) {
  int cr;
  cr = shmdt(ptr);
  handle_error("shmdt", cr);
  cr = shmctl(shmid, IPC_RMID, NULL);
  handle_error("shmctl", cr);
}
```

Le segment va donc être partagé par l'ensemble des processus. Chaque processus va écrire ses informations dans sa zone. Il va ensuite lire ce que les autres processus auront écrits. Si les autres processus n'ont pas encore

écrit leurs valeurs, le processus va devoir attendre que les autres processus aient écrit avant de pouvoir décider qui a remporté cette élection. Chaque processus prend la décision à partir des données écrites par les autres processus.

4 Travail personnel : lister les processus du système

L'objectif de cet exercice est de lister les processus du système un peu de la même manière que le font les commandes `ps`,`top` ou bien `htop`. Pour réaliser cet exercice, nous nous appuierons sur le contenu du répertoire `proc`.

```
$ ls /proc
1 1630 2024 2415 2538 2717 2926 2977 3032 332 5015 860 diskstats kcore self
11980 17666 2027 2449 2553 2771 2928 298 3040 3337 5017 861 dma kmsg slabinfo
1291 17667 203 2459 2569 2772 2943 299 3048 334 6 9 driver loadavg stat
1293 19144 204 2499 2578 2773 2947 3 305 337 690 966 execdomains locks swaps
1295 19190 205 2512 2590 2774 2948 3006 3094 3449 7 acpi fb meminfo sys
1297 19330 206 2513 2604 2775 3096 3454 749 asound filesystems misc sysvipc
1299 19352 207 2520 2631 2776 2964 301 3106 3783 762 buddyinfo fs modules tty
13 19360 2120 2527 2640 2867 2967 3012 3107 3803 775 bus interrupts mounts uptime
3 14 19559 2131 2528 2670 2911 2968 3017 3140 4 8 cmdline iomem mtrr version
15 1986 2132 2533 2705 2914 2969 3022 3239 457 857 cpuinfo ioports [net vmstat
158 1996 2155 2535 2707 2915 297 4992 858 crypto irq partitions zoneinfo
1602 2 2163 2537 2713 2917 2974 3027 3313 5 859 devices kallsyms scsi
$
```

Les noms en gras (ou en couleur) sont des noms de répertoires. Les noms de répertoires qui sont des nombres corres- pondent aux PID des processus. Pour chaque processus, il y a donc un répertoire qui a comme nom la chaîne de caractères correspondant au PID du processus. Pour avoir la liste des processus du système, il suffit donc de lister le contenu de ce répertoire et prendre les noms de répertoires correpondant à des nombres. Il faut pour cela utiliser les fonctions `opendir`, `readdir`, `closedir`.

Nous allons maintenant consulter le contenu d'un répertoire correspondant à un processus.

```
$ll /proc/1867
dr-xr-xr-x 2 marangov ima-all 0 sept. 14 11:26 attr
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 autogroup
-r-------- 1 marangov ima-all 0 sept. 14 11:26 auxv
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 cgroup
--w------- 1 marangov ima-all 0 sept. 14 11:26 clear_refs
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 cmdline
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 comm
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 coredump_filter
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 cpuset
lrwxrwxrwx 1 marangov ima-all 0 sept. 14 11:26 cwd -> /home/m/marangov
-r-------- 1 marangov ima-all 0 sept. 14 11:26 environ
lrwxrwxrwx 1 marangov ima-all 0 sept. 14 11:26 exe -> /bin/bash
dr-x------ 2 marangov ima-all 0 sept. 14 11:13 fd
dr-x------ 2 marangov ima-all 0 sept. 14 11:26 fdinfo
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 gid_map
-r-------- 1 marangov ima-all 0 sept. 14 11:26 io
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 limits
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 loginuid
dr-x------ 2 marangov ima-all 0 sept. 14 11:26 map_files
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 maps
-rw------- 1 marangov ima-all 0 sept. 14 11:26 mem
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 mountinfo
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 mounts
-r-------- 1 marangov ima-all 0 sept. 14 11:26 mountstats
dr-xr-xr-x 7 marangov ima-all 0 sept. 14 11:26 net
dr-x--x--x 2 marangov ima-all 0 sept. 14 11:26 ns
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 numa_maps
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 oom_adj
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 oom_score
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 oom_score_adj
-r-------- 1 marangov ima-all 0 sept. 14 11:26 pagemap
-r-------- 1 marangov ima-all 0 sept. 14 11:26 personality
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 projid_map
lrwxrwxrwx 1 marangov ima-all 0 sept. 14 11:26 root -> /
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 sched
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 sessionid
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 setgroups
```

```
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 smaps
-r-------- 1 marangov ima-all 0 sept. 14 11:26 stack
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:13 stat
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 statm
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:13 status
-r-------- 1 marangov ima-all 0 sept. 14 11:26 syscall
dr-xr-xr-x 3 marangov ima-all 0 sept. 14 11:26 task
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 timers
-rw-r--r-- 1 marangov ima-all 0 sept. 14 11:26 uid_map
-r--r--r-- 1 marangov ima-all 0 sept. 14 11:26 wchan
$
```

On retrouve donc dans ce répertoire les principales informations que conserve le noyau du système pour chaque processus. On peut donc y voir par exemple :

— le lien `exe` pour le nom du programme qu'exécute le processus,
— le nom du répertoire `cwd` courant où se trouve le processus,
— le fichier `environ` avec les variables d'environnement,
— le fichier `maps` qui donne une description de l'espace de mémoire virtuelle,
— le répertoire `fd` où on retrouve l'ensemble des descripteurs de fichiers qui ont été ouverts,
— et bien d'autres choses qu'on vous laisse découvrir...

A partir de ces informations, vous pouvez essayer de lister l'ensemble des processus du système avec pour chaque processus le nom du programme qu'il exécute. Vous pourrez ensuite compléter avec les informations trouvées dans la hiérarchie pour vous rapprocher de ce que donne la commande `ps`.