Raphaël CASTANIER

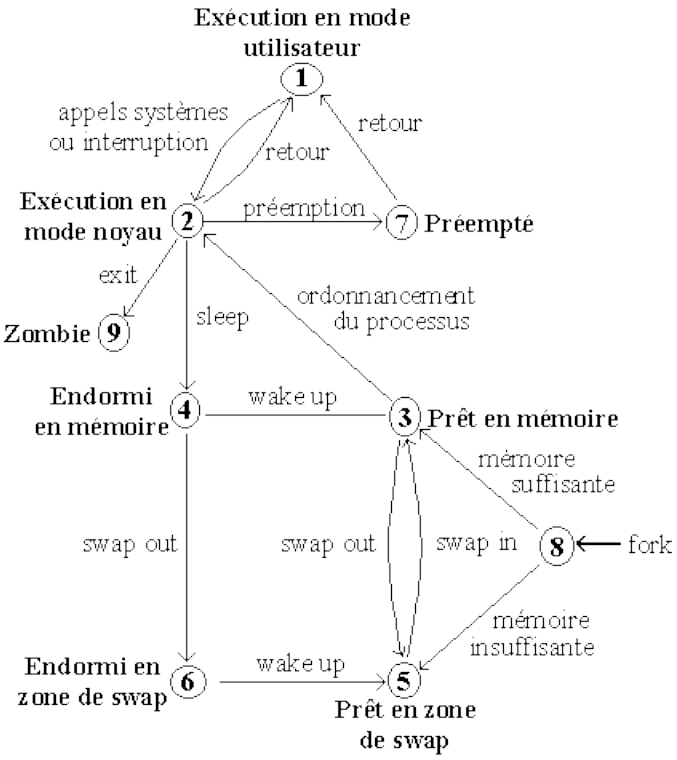
Joffrey RIUS BERNAL

Groupe TP2/ OS308

Compte Rendu TP3

Gestion des processus

```
  11 root       0 -20     0     0     0 S  0.0  0.0   0:00.00 khelper
  12 root      20   0     0     0     0 S  0.0  0.0   0:00.02 kdevtmpfs
  13 root       0 -20     0     0     0 S  0.0  0.0   0:00.00 netns
  14 root       0 -20     0     0     0 S  0.0  0.0   0:00.00 writeback
  15 root       0 -20     0     0     0 S  0.0  0.0   0:00.00 kintegrityd
  16 root       0 -20     0     0     0 S  0.0  0.0   0:00.00 bioset
admuser@srv01:~/htop-1.0.2$ ls /proc/
1     2    302  49   825        dma          latency_stats  slabinfo
10    20   348  5    856        driver       loadavg        softirqs
1061  200  349  506  9          execdomains  locks          stat
1062  201  350  576  914        fb           mdstat         swaps
11    202  4    617  937        filesystems  meminfo        sys
1160  21   409  68   acpi       fs           misc           sysrq-trigger
1185  22   42   69   asound     interrupts   modules        sysvipc
12    24   429  7    buddyinfo  iomem        mounts         timer_list
13    25   43   789  bus        ioports      mtrr           timer_stats
14    26   430  797  cgroups    irq          net            tty
15    27   432  8    cmdline    kallsyms     pagetypeinfo   uptime
16    28   45   808  consoles   kcore        partitions     version
17    29   46   810  cpuinfo    key-users    sched_debug    version_signature
18    297  462  814  crypto     kmsg         schedstat      vmallocinfo
184   3    466  820  devices    kpagecount   scsi           vmstat
```



Année Académique 2017/2018

Sommaire

TP3 - Gestion des processus

1) Contrôle des processus

1.1 Travaux en arrière-plan (backgrounding)

Manipulation 1

1. Allez dans le dossier TP1 et compilez le programme chrono.c
`$ gcc chrono.c -o chrono`
2. Dans deux terminaux séparés, lancez le programme chrono avec et sans le &.
a. Dans terminal 1 `$ ./chrono`
b. Dans terminal 2 `$ ./chrono &`

3. Pour chaque manipulation, lors de l'exécution du programme,
a. tapez la commande `ls` qui affiche le contenu du dossier courant. Quelle constatation peut-on faire ?
b. cliquez sur `Ctrl + c` qui permet de terminer le processus en cours. Quelle constatation peut-on faire ?

Dans terminal 1 (Sans &) :

La commande 'ls' ne s'affiche pas pendant que `chrono` tourne mais apparait après son exécution. Pour terminer le processus étant donné que le programme est en 1er plan la commande 'Ctrl +C' arrête le processus.

Dans terminal 2 (Avec &) :

La commande 'ls' fonctionne simultanément avec l'exécution de 'chrono car le `&` a lancé son exécution en tâche de fond ce qui libère la main au Shell pour lancer plusieurs commandes. Etant donné que l'esperluette (&) a mis le programme en arrière plan, 'chrono' s'exécute jusqu'à son nombre d'itération max.

Manipulation 2

4. Lancez le programme chrono dans un terminal (sans le &).
`$ ./chrono`
5. Testez l'utilisation de la commande `ls`
6. Mettez le processus en arrière-plan et testez à nouveau avec la commande `ls`

Méthode :

1) `ls` ne fonctionne pas.
2) `CTRL+Z` met bien en arrêt.
3) `bg` relance en arrière plan.
4) `ls` fonctionne

En utilisant cette méthode, nous reprenons les caractéristiques de la manipulation N°1.

1.1 Travaux en premier plan (Foregrounding)

Manipulation 3

7. Affichez les pages d'aide des commandes suivantes dans un seul terminal et en même temps : `ps` , `kill` , `top` et `exec` .
Exemple : `$ man ps`
8. Basculez d'une page à l'autre, tout en lisant la description de chacune de ces commandes.

Afin de lancer plusieurs 'man' nous avons saisi la commande ci-dessous :

`man ps & man kill & man top & man exec &`

Ensuite, pour lister les processus et pouvoir vérifier les numéros d'indice afin d'ouvrir la page voulue, nous avons saisi cette commande :

`jobs`

Une fois la liste affichée, nous pouvons sélectionner le processus à mettre en premier plan en écrivant :

`fg <numero_processus>`

Puis pour arrêter ce processus :

`CTRL+Z`

Ce fonctionnement nous permet donc de créer plusieurs processus puis de sélectionner celui que l'on veut mettre en 1er plan (Foregrounding).

1.2 Commande PS

Manipulation 4

9. Effectuez des recherches pour trouver la définition de la notion de processus *démon*. Donnez cette définition.
10. Ouvrez deux terminaux différents. Sur l'un d'eux essayez la séquence de commandes suivantes. Voir l'aide de la commande `ps` afin de comprendre le résultat de chaque option.
a. `$ ps -e` quelle est la différence avec un `ps` seul ?
b. `$ ps -f` que représentent les colonnes `UID`, `PPID` et `STIME` ?
c. Combiner les deux options précédentes et retrouver les ascendants du processus `ps`.
d. Quel est le processus père de tous les autres processus ?
e. `$ ps U root` que fait cette commande ? Il est aussi possible de mettre son nom d'utilisateur à la place de « root »

Processus démon : Processus qui n'a pas d'interaction avec l'utilisateur, il est fils du processus de `PID=1`. Un processus devient un démon lorsque son père est tué. Le meilleur moyen de créer un démon est donc de tuer le processus du père.

`ps` : affiche la liste des processus de l'utilisateur

`ps -e` : affiche la liste de tous les processus

`UID` : identifiant de l'utilisateur qui a lancé ce processus

`PPID` : identifiant du processus père

`STIME` : temps système du processus

Le processus père de tous les autres est `init` (1).

`ps U root` : affiche les processus de l'utilisateur `root`

1.4 Commande KILL

Arrêter un processus en connaissant son PID (Envoi du signal SIGTERM)

$ kill +N° PID

Arrêter un processus de façon non régulière (Envoi du signal SIGKILL)

$ kill -9 +N° PID

1.5 Commande TOP

Manipulation 5

11. Consultez la page d'aide de la commande `top`, et trouvez la bonne option pour afficher seulement les processus qui vous appartiennent.

`top -u user` : affiche seulement les processus de l'utilisateur `user`

2) Création de processus (fonction fork)

Exercice 1

1. Après compilation de exo1.c, qui se trouve en annexe, exécutez le programme exo1 plusieurs fois.

```
$ gcc exo1.c -o exo1
$ ./exo1
```

Pourquoi cinq lignes sont-elles affichées alors qu'il n'y a que trois printf dans le programme?

2. Ajoutez maintenant la ligne suivante après l'appel à fork() :

```
if (valeur == 0) sleep (4);
```

Que se passe-t-il ? Pourquoi ?

exo1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void)
{
  int valeur;
  printf("printf-0 Avant fork: ici processus numero %d\n",(int)getpid());
  valeur = fork();
  printf("printf-1 Valeur retournee par la fonction fork: %d\n", (int)valeur);
  printf("printf-2 Je suis le processus numero %d\n", (int)getpid());
  return 0;
}
```

1) Lors de l'exécution de `exo1`, le premier `printf` est exécuté, puis `fork()` est appelé.

Ensuite, le père **ET** le fils exécute les deux autres `printf`.

Au total, 5 `printf` sont affichés à l'écran.

2) Avec `if (valeur == 0) sleep (4);`, le fils attend 4 secondes avant d'exécuter les 2 `printf`du fils

Cela permet de voir l'évolution des 'printf' et de pouvoir mieux comprendre le fonctionnement.

Le 'if' sélectionne uniquement le fils car il teste la valeur de retour du 'fork()' égale à 0 donc la valeur du processus fils.

Exercice 2

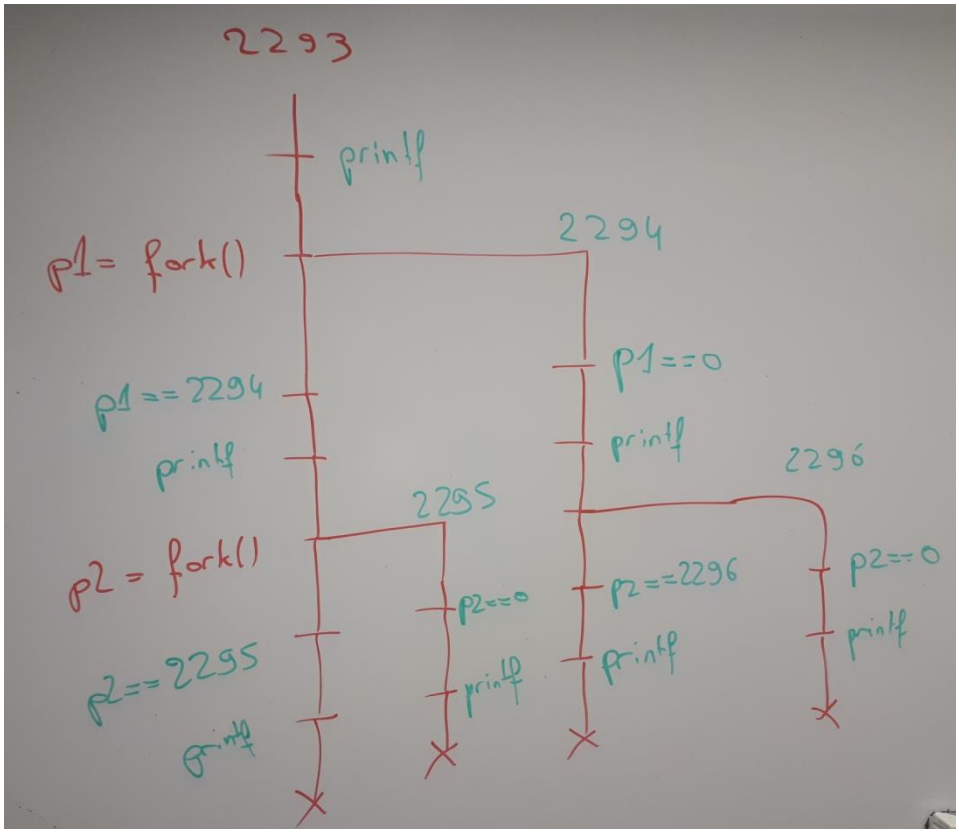
3. Compilez le programme exo2.c, donné en annexe. Après exécution et à l'aide du schéma suivant, relevez les numéros des processus et numérotez l'ordre d'exécution des instructions printf() de façon à retrouver l'ordre d'exécution des processus.

Une fois l'exo2.c compilé et exécuté, nous avons obtenu le résultat suivant, qui nous a permis de dessiner le schéma ci-dessous.

Retour de exo2

PID	PPID	INSTRUCTION	RETOUR-FORK
2293	2	print 1	
2293	2	print 2	2294
2294	2293	print 2	0
2293	2	print 3	2295
2295	2293	print 3	0
2294	2293	print 3	2296
2296	2294	print 3	0

Schéma fork()



Analyse :

On peut voir sur celui-ci qu'une fois que le fork() a été appelé une 1ère fois, si nous refaisons un fork(), le fils le fait également. Avec ce fonctionnement cela ne nous fait donc pas 1 processus père et deux fils mais 4 branches (voir ci-dessus).

Notes :

- Le schéma précédent montre la création de fils et les valeurs de retour des fork()
- Il ne rend pas compte du PPID, qui peut varier selon les exécutions car l'ordre de terminaison des processus n'est pas déterministe
- L'ordre des printf n'est pas non plus déterministe
- Les numéros des PID sont susceptibles de varier à chaque exécution