Université de Nice Sophia Antipolis
UFR Sciences - Dépt. Informatique

Licence d'Informatique L2
Introduction aux Systèmes et Réseaux

(feuille d'exercice adaptée du Cours de S. Krakowiak, U. Grenoble)

TD n° 1 : Introduction aux processus en Unix

L'objectif de ce TD[1] est d'approfondir les notions relatives aux processus et de les appliquer au cadre spécifique d'Unix. Ces notions seront appliquées dans le TP n°1. Ce texte ne donne pas tous les détails sur les primitives décrites. En cas de besoin, pour les TP, utiliser `man` pour la version C des primitives et l'aide de Python (modules `os,time,sys`).

1 Création de processus

La primitive `fork()` crée un processus "fils" du processus appelant (le "père"), avec le même programme que ce dernier. La valeur renvoyée par `fork()` est :

– au père : le numéro (PID) du processus fils.
– au fils : 0.

En cas d'échec (table des processus pleine), aucun processus n'est créé, et l'exception `OSError` est levée.

1.1 Question 1

Qu'affiche l'exécution du programme suivant :

```
import os,sys
x = 1

pid = os.fork()
if (pid == 0):
    x=x+1
    print "child : x=%d" %  (x)
    sys.exit(0)
# parent
x = x -1
print "parent: x=%d" % (x)
sys.exit(0)
```

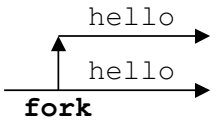
[1] Plusieurs des figures et exemples sont empruntés à R. E. Bryant, D. O'Hallaron. *Computer Systems : a Programmer's Perspective*, Prentice Hall, 2003.

1.2 Question 2

On considère les deux programmes suivants et le schéma de leur exécution (l'axe du temps est orienté vers la droite).

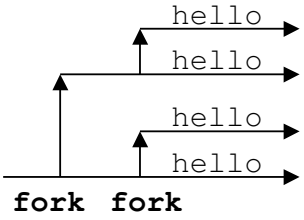
```
import sys,os

os.fork()
print "hello!"
sys.exit(0)
```




```
import sys,os

os.fork();
os.fork();
print "hello!"
sys.exit(0)
```



Illustrer l'exécution du programme obtenu en ajoutant un troisième `Fork()`.

1.3 Question 3

Combien de lignes "hello !" imprime chacun des deux programmes suivants ?

```
import sys,os

for i in range(2):
    os.fork()
print "hello!"
sys.exit(0)
```

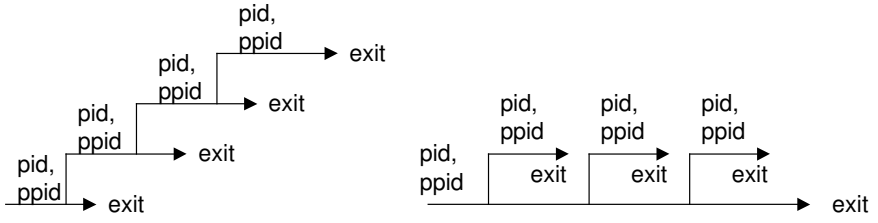
```
import sys,os

def doit():
    os.fork()
    os.fork()
    print "hello"
    return

if __name__ == "__main__":
    doit()
    print "hello"
    sys.exit(0)
```

1.4 Question 4

On considère les deux structures de filiation (chaîne et arbre) représentées ci-après.



Écrire un programme qui réalise une chaîne de `n` processus, où `n` est passé en paramètre de l'exécution de la commande (par exemple, n = 4 sur la figure ci-dessus). Faire imprimer le numéro de chaque processus et celui de son père. Même question avec la structure en arbre.

> En python, on recupere les arguments du programme dans la *liste* `sys.argv`. Attention le premier element de la liste (`sys.argv[0]`) contient le nom du programme. Le premier paramètre du programme se trouve donc dans `sys.argv[1]`. Pour savoir combien d'arguments ont été donnés, il suffit d'utiliser la fonction `len()` :
> `if len(sys.argv) == 1 : ...`

2 Relations entre processus père et fils

Un processus termine son exécution en appelant la primitive `sys.exit(statut)`. La valeur de `statut` est utilisée pour renvoyer un code de retour qui donne des informations sur la terminaison du processus. Habituellement, on utilise la valeur 0 pour une terminaison normale, et une valeur différente de 0 pour signaler une condition anormale (cette valeur indiquant la nature de l'anomalie selon une convention fixée).

Lorsqu'un processus se termine, il ne restitue pas toutes ses ressources et reste dans un état appelé "zombi", tant que son père n'a pas pris connaissance de son statut, par une primitive `os.wait` ou `os.waitpid`. Ces primitives sont utilisées par un processus père pour attendre la fin d'un ou plusieurs de ses fils[2].

Nous utilisons la primitive `os.waitpid`, définie comme suit :

```
import os
(pid,statut) = os.waitpid(pid, options)
```

Le paramètre `pid` permet de sélectionner le processus dont on attend la fin[3].

- si `pid` > 0, attendre la fin du processus fils de numéro `pid`
- si `pid` $= -1$, attendre la fin d'un processus fils quelconque

La valeur renvoyée est un tuple contenant le numéro du processus fils (zombi) effectivement pris en compte et l'information sur l'état du processus terminé. Pour l'interpréter, il est préférable d'utiliser des fonctions prédéfinies, par exemple :

- `os.WIFEXITED(statut)` vrai si le fils s'est terminé normalement (non interrompu par un signal), faux sinon.
- `os.WEXITSTATUS(statut)` donne le statut de sortie (code de retour) du fils (uniquement si la primitive précédente a renvoyé vrai)

Le programme ci-après lance un processus qui attend la fin de ses fils et imprime leur code de retour.

```
import sys,os,errno
N_wait = 2
```

[2] si un processus se termine sans avoir attendu la fin de ses fils, ceux-ci sont rattachés au processus de numéro `pid` = 1, qui finira par les éliminer.

[3] il y a d'autres possibilités liées aux groupes de processus, non examinées ici.

```
for i in range(N_wait):
    pid = os.fork()
    if (pid == 0):  # child
        sys.exit(100+i)

try:
    # parent waits for all of its children to terminate
    (pid,statut) = os.waitpid(-1,0)
    while pid > 0:
        if os.WIFEXITED(statut):
            print "child %d terminated normally with exit status=%d" %\
                (pid, os.WEXITSTATUS(statut))
        else:
            print "child %d terminated abnormally" % (pid)
        (pid,statut) = os.waitpid(-1,0)

except OSError, (err,strerror):
    if err != errno.ECHILD:
        print "waitpid error(%d): %s" % (err,strerror)

sys.exit(0)
```

2.1 Question 5

Modifier ce programme pour qu'il imprime les processus dans l'ordre dans lequel ils ont été créés.

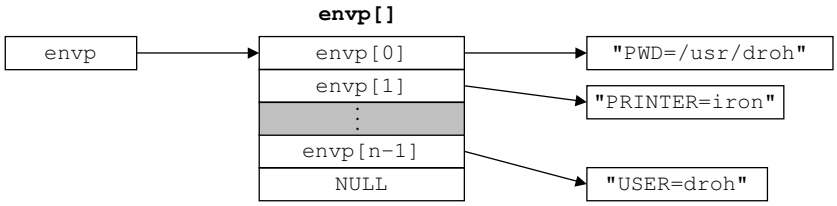
2.2 Question 6

On considère le programme suivant :

```
import sys,os

print "Hello"
pid = os.fork()
print "ici:%d", !pid
if pid != 0:
    (pid_wait,statut) = os.waitpid(-1, 0)
    if pid_wait > 0:
            if os.WIFEXITED(statut) != 0:
                print "la:%d" % (os.WEXITSTATUS(status))
    print "Bye"
    sys.exit(2)
```

Combien de lignes ce programme imprime-t-il ? Discuter les ordres possibles dans lesquels ces lignes sont imprimées.



3 Exécution d'un programme

La famille de primitives `exec` permet de créer un processus pour exécuter un programme déterminé (qui a auparavant été placé dans un fichier, sous forme binaire exécutable).

On utilise en particulier `os.execv` pour exécuter un programme en lui passant un tableau d'arguments, et `os.execve` en lui passant en outre un tableau de variables d'environnement (variables prédéfinies utilisées pour donner des informations telles que le nom de l'utilisateur, le *shell* préféré, les périphériques utilisés, les chemins de recherche des fichiers, etc.).

```
import os
os.execve(filename, argv, envp)
```

La primitive `execv` ne comporte pas l'argument `envp`. Le paramètre `filename` contient vers le nom (absolu ou relatif) du fichier exécutable, `argv` contient la liste des arguments et `envp` contient la liste des variables d'environnement. Par convention, le paramètre `argv[0]` contient le nom du fichier exécutable, les arguments suivants étant les paramètres successifs de la commande.

```
import os
MAX = 5

argv = ["ls", "-lt", "/"]
os.execv("/bin/ls", argv)
```

3.1 Question 7

Que fait le programme ci-dessus ? Noter que les primitives `exec` provoquent le "recouvrement" de la mémoire virtuelle du processus appelant par le nouveau fichier exécutable. Il n'y a donc pas normalement de retour (sauf en cas d'erreur, par exemple fichier inconnu, auquel cas la primitive lève l'exception `OSError`).

3.2 Question 8

Écrire un programme `execcmd` qui exécute une commande Unix qu'on lui passe en paramètre. Exemple d'exécution :

```
execcmd /bin/ls -Ft /
```

Pour écrire ce programme, il faut savoir qu'une variable prédéfinie de type liste appelée `os.environ` contient, par convention, les variables d'environnement dans la mémoire d'un processus (voir la figure ci-dessus).