Université Côte d'Azur
Faculté des Sciences - Département d'Informatique

Licence d'Informatique L2
Introduction aux Systèmes et Réseaux

TD n ° 3 : Processus en Unix - Recouvrement

1 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 `os.execv` pour exécuter un programme en lui passant une liste d'arguments, et `os.execve` en lui passant en outre un dictionnaire 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, ...)

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

La primitive `execv` ne comporte pas l'argument `envp`. Le paramètre `filename` contient le nom (absolu ou relatif) du fichier exécutable, `argv` contient la liste des arguments et `envp` contient le dictionnaire 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. La primitive `os.execvpe` se sert du `PATH` pour localiser l'exécutable.

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

1.1 Question 1

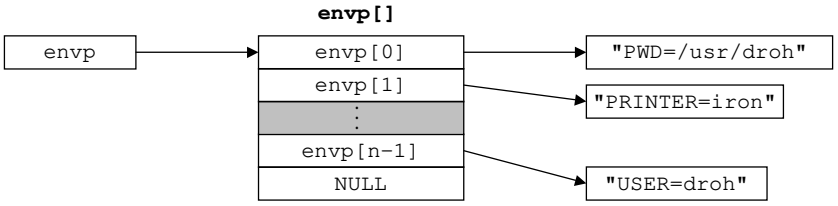
Que fait le programme ci-dessus ? Noter que les primitives `exec` provoquent le "recouvrement" de la mémoire virtuelle du processus appelant par le nouveau fichier exécutable. Il n'y a donc pas normalement de retour (sauf en cas d'erreur, par exemple fichier inconnu, auquel cas la primitive lève l'exception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

1.2 Question 2

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 dictionnaire appelée `os.environ` contient, par convention, les variables d'environnement dans la mémoire d'un processus (voir la figure ci-dessus).



2 Recouvrement - à vous de jouer

Écrire un programme `verifier.py` dont l'usage sera `verifier.py com arg1 .. argn`, et qui lance la commande `com arg1 .. argn`, signale une éventuelle erreur lors du lancement, attend la fin de l'exécution et précise par un message le résultat (succés ou échec).

3 Recouvrement séquentiel

Écrivez un programme Python qui, par le biais de créations de processus et de recouvrements, exécute la suite de commandes `who ; pwd ; ls -l` (rappel : le point-virgule signifie qu'une commande est exécutée lorsque la commande précédente est terminée).

4 La commande `myif`

Écrire un programme Python `myif.py` qui admet la ligne de commande suivante :
`myif.py` *command1 args* ...`--then` *command2 args* ...[`--else` *command3 args* ...] `--fi`

Le programme exécute la première commande puis, selon qu'elle ait réussi ou échoué, exécute la deuxième ou la troisième commande.

5 La commande `mywhile`

On se propose d'écrire un programme Python `mywhile.py` qui admet la ligne de commande suivante :
`mywhile.py commande1 [arg ...] --do commande2 [arg ...] --done`

1. Écrire la fonction `indice(liste, élément)` qui renvoie l'indice de la première occurence de `élément` dans `liste`, ou `-1` sinon.

   Note : la méthode `index(e)` des listes, retourne la première occurrence de `e` dans la liste. Elle lève une `ValueError` si l'élément est absent.

2. Écrire le programme principal, qui recherche l'indice des arguments `--do` et `--done`, puis affiche l'usage du programme et le termine si la syntaxe n'est pas respectée.

   Ensuite, le programme exécute `commande1` et attend sa fin. Si elle a échoué, le programme se termine, sinon il exécute `commande2` et attend sa fin, puis recommence à exécuter `commande1`, etc.

   Lorsque le programme se termine, il renvoie le *statut* de la dernière exécution de `commande2`, à défaut `1`. Pour chaque exécution de commande, le programme se duplique et le fils se recouvre.