info502 : Système d'exploitation
TP 2 : shell, scripts, introduction au C

Pierre Hyvernat, Sylvie Ramasso, Brice Videau
Pierre.Hyvernat@univ-savoie.fr
Sylvie.Ramasso@univ-savoie.fr
Brice.Videau@univ-savoie.fr

Pour ce TP, comme pour le suivant, votre travail devra être envoyé par email à votre intervenant. Les consignes suivantes sont non-négociables :
- chaque fichier doit impérativement commencer par une entête comportant votre filière et les noms de tous les membres du groupes ;
- pour les parties "programmation", les réponses aux questions posées devront être dans le même fichier que votre programme. Utilisez des commentaires ;
- si vous avez une partie "compte-rendu" qui ne rentre dans aucun fichier source, vous pouvez joindre un fichier au format texte (pas de fichiers Word) ;
- faites des efforts pour le formatage de vos fichiers : indentez, allez à la ligne etc.

Finalement, si vous contactez un intervenant par email, mettez toujours un sujet commençant par "`[info502]`". Sans ça, vous courrez le risque que votre message ne soit pas lu...

Exercice 1 : Redirections et tubes

Toutes les commandes font des entrées/sortie, en général via le clavier et l'écran. Il est possible de rediriger les sorties dans un fichier, ou au contraire de rediriger le contenu d'un fichier vers un commande. Pour ça, il faut utiliser "`>`" et "`<`". Par exemple, pour obtenir la liste de tous les fichiers du répertoire courant dans le fichier `liste`, il suffit de faire "`ls -AR > ./liste`".

Question 1. Essayer la commande précédente. Que se passe-t'il ? Vérifiez que `liste` contient bien la liste des fichiers. (Vous pouvez utiliser la commande `less` pour visualiser le contenu d'un fichier texte.)

Question 2. Essayer maintenant la commande "`ls -Aj > liste`". Que constatez-vous ? Que contient le fichier `liste`, et comment l'expliquez-vous ?

Question 3. Essayer de découvrir ce qu'il se passe si on remplace "`>`" par "`>>`" ou par "`2>`".

Question 4. Dans le répertoire `/tmp/`, créez deux fichiers vides `toto1` et `toto2` avec la commande "`touch toto1 toto2`". Tapez ensuite la commande "`rm -i toto1 toto2`". Que se passe t'il ?

Essayez de supprimer l'intervention de l'utilisateur en utilisant les redirections et "`<`". Testez.

On veut parfois utiliser la sortie d'une commande comme entrée d'une autre commande. Par exemple, la commande "`wc -l`" permet de compter le nombre de ligne données sur l'entrée standard. Pour compter le nombre de fichiers dans le répertoire `/usr/bin/`, on peut donc faire

```
ls -l /usr/bin/ > /tmp/toto ; wc -l < toto
```

(le "`;`" permet de séquentialiser les commandes)

Remarque : comme `wc` permet de donner un nom de fichier en argument, on peut également faire "`ls -l /usr/bin/ > /tmp/toto ; wc -l toto`"

Une meilleure solution est d'utiliser un "tube" (ou "pipe" en anglais) : on connecte ainsi directement la sortie d'une commande sur l'entrée d'une autre commande :

```
ls -l /usr/bin/ | wc -l
```

Question 5. Testez les deux méthodes et comparez les résultats.

Un autre avantage des tubes est que ceux-ci sont des FIFOs ("first in, first out"). Autrement dit, la seconde commande peut commencer à s'exécuter avant que la première ne soit terminée.

Question 6. La commande `less` permet d'afficher du texte en stoppant à chaque page. Comparez les commandes

- `ls -R /usr/lib/`
- `ls -R /usr/lib/ > /tmp/toto ; less < /tmp/toto`
- `ls -R /usr/lib/ | less`

Question 7. Que fait la commande `yes` ? Quelle peut être son utilité ?

Exercice 2 : Signaux

Lorsqu'un programme s'exécute, l'utilisateur peut envoyer des signaux au programme : par exemple, "Control-c" envoie le signal `SIGINT` (signal d'interruption) et "Control-z" envoie le signal `SIGTSTP`. Le programme peut prendre (ou non) ces signaux en compte.

Pour envoyer un signal, il suffit d'utiliser la commande "`kill -`*n pid*" où *n* est le numéro du signal à envoyer ("`kill -l`" donne la liste de tous les signaux existant) et *pid* le numéro du processus (vous pouvez le récupérer avec "`ps -e`".)

Question 1. Essayer d'envoyer les signaux `SIGINT` et `SIGTSTP` à un processus.

Pour gérer les signaux dans un script, il faut utiliser la commande `trap`. Créez un fichier `test-trap.sh` contenant

```
trap "echo Script terminé" EXIT
trap "echo Vous avez appuyé sur Ctrl-C" SIGINT
trap "echo Vous avez fait: kill $$" SIGTERM
trap "echo Je continue" SIGCONT
trap "echo SIGUSR1 ou SIGUSR2 ???" SIGUSR1 SIGUSR2
echo "Processus $$: J'attend un signal..."
#
# Compte à rebours
#
i=100
while [ $i -gt 0 ]; do
  echo -n "$i "
  sleep 1
  let $[ i -= 1 ]
done
echo "0"
```

Question 2. Testez avec les commandes suivantes :

- exécutez le script
- Control-z puis `fg`
- Control-c
- dans un autre terminal "`kill -9` *n*" (où *n* est le numéro du processus)
- dans un autre terminal "`kill -s SIGUSR1` *n*"
- dans un autre terminal "`kill -s SIG`(autre signal) *n*"

Question 3. Que se passe-t'il ? Que fait le `trap` ?

À quoi est-ce que la commande peut servir ? Donnez des exemples possibles d'utilisation.

Exercice 3 : Une vache qui dit l'heure...

Remarque : pour cet exercice, seule la dernière version de votre script est à rendre. (Mais avec tous les commentaires pertinents...)

"`cowsay`" est un programme qui permet d'afficher une vache avec une bulle de texte :

```
> cowsay "Salut, mon nom est Bob et je suis une vache..."
 ---------------------------------------
/ Salut, mon nom est Bob et je suis une \
\ vache...                              /
 ---------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Nous allons créer une version plus simple de ce programme inoubliable : la vache nous donnera seulement l'heure qu'il est :

```
> cowtime
 -----------------
( Il est 12h36... )
 -----------------
        o   ^__^
         o  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Détails et compléments :

- n'oubliez pas de mettre "`#!/bin/bash`" sur la *première* ligne de votre script : c'est ça qui permet de l'exécuter directement.
- Pour obtenir l'heure vous pouvez utiliser la commande "`date +%Dh%M`". (Pour configurer le format, faites un "`date --help`".)
- Pour affecter la valeur d'une commande à une variable, il faut faire "`nom_var=‘commande‘`".
- La commande `echo` permet d'afficher du texte et des variables, et la commande `cat` permet d'afficher le contenu d'un fichier.

Question 1. Programmez la commande `cowtime`.

Question 2. Nous allons maintenant rajouter des options à la commande `cowtime` : la première option sera l'option "`-e`" qui permettra de changer les yeux de la vache, par exemple :

```
> cowtime -e xx
 -----------------
( Il est 10h32... )
 -----------------
        o   ^__^
         o  (xx)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Rajoutez cette option, en mettant une valeur par défaut quand elle n'est pas présente.

Détails et compléments :

- les arguments d'un script sont contenus dans les variables `$1`, `$2`, etc. (La variable `$0` contient le nom du script.)
- On peut tester plein de trucs avec la commande `test`.

Question 3. Rajouter une deuxième option "`-t`" permettant de mettre une langue à la vache :

```
> cowtime -t U
 _________________
( Il est 10h32... )
 -----------------
        o   ^__^
         o  (oo)\_______
            (__)\       )\/\
             U  ||----w |
                ||     ||
```

Bien entendu, on doit pouvoir spécifier une langue et des yeux, dans n'importe quel ordre. On veut également que si une option est présente plusieurs fois, seule la dernière valeur est prise en compte : par exemple, "`cowtime -e oo -e OO -t u -e oO -t U`" doit avoir le même résultat que "`cowtime -e oO -t U`".

Rajoutez ensuite une option "`-h`" qui affiche une petite aide.

Détails et compléments :

- on peut faire des boucles `while` avec la syntaxe

```
while ...
do
  ...
done
```

- on oublie un argument en utilisant la fonction `shift` : elle décrémente tous les arguments... (`$2` devient `$1` etc.)
- pour gérer de nombreux argument, on peut aussi utiliser la commande `getopts`. Voir le manuel pour un exemple...

Question 4. Créez une option qui permet de faire dire la date (plutôt que l'heure) à votre vache. On doit avoir accès à cette option soit par une option, soit en appelant le script `cowdate` qui n'est rien d'autre qu'un lien symbolique vers `cowtime`. (Pour obtenir un tels lien symbolique : "`ln -s cowtime cowdate`")

Rajoutez une option qui permet à l'utilisateur de donner le format passé à `date` pour afficher ce qu'il veut.

Question 5. **(Bonus)** Rajouter la possibilité de préciser un fichier où se trouvera un autre type de vache en ASCII-art. (Toutes les options devront encore marcher avec ces nouvelles vaches.)

Exercice 4 : Un peu de C

Le C est un langage de programmation séquentiel (à la Pascal) qui permet de manipuler facilement la mémoire physique. C'est un langage de choix pour les applications nécessitant un accès direct aux données ; mais ce qu'on gagne en vitesse, on risque de le perdre en simplicité...

Un programme C typique ressemble à :

```
/* bibliothèques utilisées */
#include <stdlib.h>
#include <stdio.h>
#include "mes_declarations.h"

/* mes fonctions */
```

```
int cube (int n) {
  int m = n*n*n ;
  if (m<0) {return (-m);}
  else     {return (m);} }

/* la fonction principale */
int main (void) {
  int n = 0 ;
  printf("Entrez un nombre entier : ");
  scanf("%i",&n);
  printf("La valeur absolue du cube de %i est %i.\n",n,cube(n));
  return(0); }
```

avec le fichier "`mes_declarations.h`" contenant

```
int cube (int) ;
```

Les compléments concernant
- les structures de programmation (boucles, conditionnelles),
- les pointeurs et leur liens avec les tableaux,
- certains idiomes du C
- la compilation d'un programme

vous seront donnés à l'oral par votre intervenant de TP.

Question 1. Écrivez le programme "Hello world!" en C, compilez le et éxécutez le.

Question 2. Écrivez un programme C qui :
- demande un nombre de notes,
- demande chaque note ainsi que son coefficient,
- calcul la moyenne et l'affiche en décernant les mentions appropriées.

(Vous pouvez commencer par faire un programme similaire pour lequel le nombre de notes est une constante.)

Question 3. Allez récupérer le programme `pere-fils.c` dans la rubrique "Enseignement" sur le site `http://www.lama.univ-savoie.fr/~hyvernat`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (void) {
  pid_t pid ;
  if ((pid = fork()) < 0) { fprintf (stderr, "échec du fork()\n"); }
  if (pid != 0) {
    /* processus père */
    fprintf (stdout, "père : mon PID est %u.\n", getpid()) ;
    sleep(100) ;
    fprintf (stdout, "père : je me termine...\n");
    exit (0) ; }
  else {
    /* processus fils */
    fprintf (stdout, "fils : mon PID est %u.\n", getpid());
    sleep(1) ;
    fprintf (stdout, "fils : je me termine...\n");
    exit(0) ; }
  return(0); }
```

A l'aide des commandes spécifiques (`ps`, `pstree`) et le résultat de l'exécution de ce programme, essayez d'expliquer ce qu'il se passe ?

Question 4. Rajoutez le code suivant au niveau du processus fils. Que se passe t'il ?

```
pid2 = fork();
if (pid2 != 0) {
  /* processus fils */
  fprintf (stdout, "fils : mon PID est %u.\n", getpid());
  sleep(1) ;
  fprintf (stdout, "fils : je me termine...\n");
  exit (0) ; }
else {
  fprintf (stdout, "fils du fils : mon PID est %u.\n", getpid());
  sleep(50) ;
  fprintf (stdout, "fils du fils : je me termine...\n");
  exit (0) ; }
```

Question 5. À quoi peut servir un `fork` ? Quels sont les problèmes qui peuvent se poser ?