Christophe Blaess

Annexes

Ecriture de scripts shell

Ce document a été réalisé sur système Linux avec les logiciels libres :

- **Formation+** pour préparer le cours complet,
- **LaTeX** pour produire les transparents et le support de cours imprimé,
- **Xfig** pour les graphiques vectoriels,
- **The Gimp** pour les images et les photos.

Support de cours : version 17

Ecriture de scripts shell

Annexes

Christophe Blaess

– **Commandes avancées pour script shell . . . 3**
  – Affichage formatté . . . 3
  – Arguments en ligne de commande . . . 4
  – Création de menus . . . 5
  – Configuration du terminal . . . 6
  – Couleur de texte et position de curseur . . . 7
  – Interface semi-graphique . . . 8
  – Calculs réels . . . 9
  – Emission de signaux . . . 10
  – Réception de signaux . . . 11
– **F.A.Q. . . . 12**
  – F.A.Q. . . . 12
– **Bibliographie . . . 15**
  – Bibliographie . . . 15

Commandes avancées pour script shell

Affichage formatté

`printf` *format arguments...*
met en forme les arguments en fonction des directives de conversion.

Conversion :
`%`[*attribut*][*largeur*][.*precision*]*indicateur*

	Attributs
`-`	aligner à gauche
`+`	afficher le signe
espace	espace pour +
0	remplir avec des zéros

	Indicateurs
`diu`	entier décimal
`o`	entier octal
`xX`	entier hexadécimal
`feEgG`	réel format décimal et scientifique
`c`	caractère
`s`	chaîne
`%`	caractère `%`

```
$ annee=2004
$ printf "%d  %f\n" 2004 2004
2004  2004,000000
$ printf "%07d   %X \n" $annee $annee
0002004   7D4
$ pi=3,14159264
$ printf "%f    %e    %g\n" $pi $pi $pi
3,141593    3,141593e+00    3,14159
$ printf "%.2f     %.4f \n" $pi $pi
3,14     3,1416
$
```

Arguments en ligne de commande

`getopts` *options variable*
permet de lire les options se trouvant sur la ligne de commandes ayant invoqué le script.

La chaine d'*option* contient toutes les lettres représentant des options valides pour le script. Si une lettre y est suivie d'un deux-points, elle devra prendre un argument.

A chaque invocation, `getopts` remplit la *variable* avec la lettre d'option suivante. Si celle-ci prend un argument, il est placé dans la variable spéciale `OPTARG`.

Une fois toutes les options analysées, `getopts` renvoie *Faux* et place dans la variable `OPTIND` le numéro du premier paramètre positionnel qui ne soit pas une option.

Si la liste des options commence par un deux-points, `getopts` n'affiche pas de message en cas d'erreur mais remplit la *variable* avec '`?`' ou '`:`'.

```
#! /bin/sh

while getopts ":ab:h" opt; do
        case $opt in
                a) echo "Option -a rencontrée" ;;
                b) echo "Option -b avec argument $OPTARG" ;;
                h) echo "usage: ${0} [-a] [-b arg_b] fichiers..." ;;
                :) echo "L'option -$OPTARG necessite un argument"; exit 1 ;;
                ?) echo "L'option -$OPTARG n'existe pas"; exit 1 ;;
        esac
done

shift $(($OPTIND - 1))
while [ -n "$1" ]; do
        echo "Argument : $1"
        shift
done
```

```
$ ./script.sh -a -b 12 fichier_1 fichier_2
Option -a rencontrée
Option -b avec argument 12
Argument : fichier_1
Argument : fichier_2
$ ./script.sh -ah
Option -a rencontrée
usage: ./afac [-a] [-b arg_b] fichiers...
$ ./script.sh -z
L'option -z n'existe pas
$ ./script.sh -b
L'option -b necessite un argument
$
```

Création de menus

La commande `select` propose une liste d'options et exécute un bloc de commandes en ayant rempli une variable avec le choix de l'utilisateur.

```
#! /bin/sh

PS3="Votre choix ? "
select choix in "option A" "option B" "option C" "quitter" ; do
  case $choix in
    "option A" )  echo "Option choisie : A" ;;
    "option B" )  echo "Option choisie : B" ;;
    "option C" )  echo "Option choisie : C" ;;
    quitter    )  break ;;
    *          )  echo "réponse $REPLY interdite" ;;
  esac
done
```

```
$ ./script.sh
1) option A
2) option B
3) option C
4) quitter
Votre choix ? 2
Option choisie : B
Votre choix ? 8
réponse 8 interdite
Votre choix ? 1
Option choisie : A
Votre choix ? 4
$
```

Remarque : une construction comme `select fic in *` permet d'afficher la liste des fichiers d'un répertoire et de proposer à l'utilisateur d'en choisir un.

Configuration du terminal

`stty` *options...*
configure le terminal connecté sur son entrée standard.

Quelques options courantes :
- `stty -a` : affiche la configuration courante
- `stty sane` : ramène le terminal dans son état par défaut
- `stty -echo` : supprime l'écho des caractères saisis (mot de passe par exemple)
- `stty echo` : restitue l'écho des caractères
- `stty -icanon min 0 time 1` : permet de lire les touches à la volée (sans presser *Entrée*)

`stty` est normalisé par *SUSv3*.

```
#! /bin/sh
table[0]='|'
table[1]='/'
table[2]='-'
table[3]='\'
i=0
touche=""
echo "pressez 'q' pour quitter"
echo
stty -echo -icanon time 1 min 0
while [ "$REPLY" != "q" ] ; do
        read
        if [ -n "$REPLY" ]; then touche=$REPLY; fi
        echo -ne "\r${table[$i]} Dernière touche : ${touche}"
        i=$(( (i+1) % 4))
done
echo
stty sane
```

En début de ligne, une barre « tourne » sur elle-même, indiquant que le programme se déroule et capture les caractères au vol.

```
$ ./script.sh
pressez 'q' pour quitter

/ Dernière touche : a
```

Couleur de texte et position de curseur

`tput` *action...*
utilise la bibliothèque *Curses* pour accéder, depuis le shell aux commandes graphiques de la base de données Terminfo.

`tput init`	initialise l'écran graphique
`tput clear`	efface l'écran
`n=$(tput cols)`	donne le nombre de colonnes
`n=$(tput lines)`	donne le nombre de lignes
`n=$(tput colors)`	donne le nombre de couleurs
`tput bel`	émission d'un bip
`tput cup $y $x`	positionne le curseur
`tput cud1`	curseur vers le bas
`tput cuu1`	curseur vers le haut
`tput cub1`	curseur vers la gauche
`tput cuf1`	curseur vers la droite
`tput civis`	curseur invisible
`tput cvvis`	curseur visible
`tput setaf $n`	couleur du texte
`tput setaf $n`	couleur du fond
`tput blink`	clignotement
`tput bold`	surbrilance
`tput smul`	active le soulignement
`tput rmul`	arrête le soulignement
`tput sgr0`	revenir aux attributs par défaut

Couleurs habituelles :

		avec attribut `bold`
0	noir	gris foncé
1	rouge foncé	rouge vif
2	vert foncé	vert clair
3	marron	jaune
4	bleu foncé	bleu clair
5	magenta foncé	magenta clair
6	cyan foncé	cyan clair
7	gris clair	blanc

Interface semi-graphique

`dialog` *options...*
offre des éléments d'interface semi-graphiques.

`dialog` affiche des boîtes de saisie ou de dialogue, des menus, des listes à cocher, etc. et renvoie le résultat de la saisie sur sa sortie d'erreur, ainsi qu'un code de retour.

```
if dialog --yesno "Êtes-vous sûrs ?" 0 0 ; then
   ...
```

```
reponse=$(dialog --menu "Votre choix" 0 0 0 \
            "1" "Premier"    \
            "2" "Deuxième"   \
            2>&1)
```

Calculs réels

`bc` [***options...***]
permet de faire des calculs avec des nombres réels

Les calculs se font avec une précision arbitraire, contrairement aux bibliothèques mathématiques en virgule flottante.

L'option `-l` définit les fonctions mathématiques et fixe le nombre de décimales à 20.

Opérateurs : `++`, `--`, `+`, `-`, `^`, `*`, `/`, `%`, `= +=`, `-=`, `*=`, `/=`, `<`, `<=`, `>`, `>=`, `==`, `!=`

Structures de contrôle :
`if` (*expression*) { *action* }
`while` (*expression*) { *action* }
`for` (*action ;expression ;action*) { *action* }

Fonctions mathématiques :

`a`*(expr)*	arc tangente
`c`*(expr)*	cosinus
`e`*(expr)*	exponentielle
`l`*(expr)*	logarithme néperien
`s`*(expr)*	sinus
`j`*(n,expr)*	fonction de Bessel ordre n
`sqrt`*(expr)*	racine carrée

Variables spéciales :

`ibase`	base des nombres en entrée
`obase`	base à utiliser en sortie
`scale`	nombre de décimales en sortie

Invocation depuis un script :

```
$ pi=$(echo "scale=30; 4 * a(1)" | bc -l)
$ echo $pi
3.141592653589793238462643383276
$
```

```
...
# $nouveau et $ancien sont des variables du shell
# contenant des valeurs numériques
        taux=$(bc <<- FIN
                x=$nouveau
                y=$ancien
                taux = x / y * 100
                print taux
        FIN
        )
# la variable $taux contient maintenant le rapport
# entre les deux variables ramené à un pourcentage entier.
```

Emission de signaux

`kill` [`-l`] [`-s` **signal**] [**-signal**] **pid...**
envoie un signal aux processus dont les PID sont fournis

Par défaut `kill` envoie le signal 15 (`SIGTERM`)

`kill -l` donne la liste des signaux disponibles

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$
```

Exemples d'utilisation :

```
$ kill -HUP 1
$ kill -TERM 1234
$ kill -SIGQUIT 5678
$ kill -9 9012
$
```

Tuer tous les processus d'un utilisateur :

```
$ ps aux | grep [u]tilisateur | awk '{print $2}' | xargs kill -9
$
```

Réception de signaux

`trap` ***action signal***

indique le comportement à adopter lorsque le shell (le script) recevra la signal

L'*action* peut être :
- `""` pour ignorer le signal
- `"-"` pour reprendre le comportement par défaut du shell
- *commande* pour enregistrer une commande évaluée et exécutée à l'arrivée du signal

Lorsqu'une commande personnelle est exécuté à l'arrivée d'un signal, on dit que ce dernier est capturé.

Les signaux `SIGKILL` (9) et `SIGSTOP` (19) ne peuvent être ni ignorés ni capturés.

Un pseudo signal `EXIT` (0) est déclenché par le shell lorsque le script se termine.

`trap` sans argument fournit la configuration actuelle dans un format utilisable pour la restaurer ultérieurement :

```
...
sauvegarde=$(trap)
... modification temporaire ...
eval "$sauvegarde"
...
```

Exemple d'installation d'une routine gestionnaire de signal :

```
#! /bin/sh

function gestionnaire
{
        # Le gestionnaire recevra le nom du signal en argument
        if [ $# -eq 1 ]
        then
                echo "Signal $1 reçu"
        fi
}

for sig in INT HUP QUIT TERM
do
        trap "gestionnaire $i"   $i
done

echo "Mon PID est $$"
while true
do
        sleep 1
done
```

On peut lui envoyer des signaux depuis une autre console (avec `kill`) ou presser *Ctrl-C* pour voir le déclenchement des signaux.

F.A.Q.

On trouvera ici la réponse à des questions posées par des stagiaires au cours de différentes sessions de la formation *Programmation Shell et Langages de Scripts*.
Certains problèmes sont récurrents, et les réponses pourront éventuellement servir dans d'autres situations.

Comment convertir en majuscules une chaîne de caractères contenue dans une variable ?

```
chaine=$(echo "$chaine"  |  tr '[:lower:]' '[:upper:]')
```

On peut aussi invoquer Awk, comme dans la réponse suivante.

Peut-on passer en minuscules tous les noms de fichiers d'un répertoire ?

```
for fic in * ; do
  mv $fic $(echo $fic | awk '{print tolower($0)}')
done
```

Comment arrondir une valeur réelle à l'entier le plus proche ?

```
entier=$(echo "($reel + 0.5)/1"  | bc)
```

La division par 1 force `bc` à recalculer les décimales.

Quel est le répertoire dans lequel se trouve le script en cours d'exécution ?

```
# ! /bin/sh
REP=$(eval echo $(echo $(dirname $0)/ | sed 's/^\.\.?\//$PWD\/\0/'))
```

Comment lire de manière efficace la réponse à une question ?

```
oui_ou_non ()
{ # Cette routine renvoie Vrai ou Faux suivant la réponse à
  # la question fournie en argument.
  while true ; do
    echo -n "$@" " >" >&2
    if ! read < /dev/tty; then return 1 ; fi
    case $REPLY in
      [oOOyY]* ) return 0 ;;
      [nN]* )    return 1 ;;
    esac
  done
}
```

Peut-on charger toutes les lignes d'un fichier dans une variable tableau ?

```
i=1; while read ligne[i] ; do i=$((i+1)) ; done < fichier
```

Le nombre de lignes lues est `$((i-1))` en sortie de boucle.

Comment être sûr qu'une variable saisie par l'utilisateur contient une valeur numérique ?

```
if (echo "$saisie" | grep '^[0-9.]\+$' > /dev/null) ; then
    echo "Valeur numérique"
fi
```

Comment mettre en couleur une portion de message affiché par un script ?

En utilisant la commande `tput` :

```
debut_msg=$(tput init; tput setab 1; tput setaf 3; tput bold)
fin_mg=$(tput init)
...
echo "Voici un ${debut_msg}message important${fin_msg} mis en évidence"
```

Comment mettre en gras le *prompt* d'un utilisateur ?

En encadrant la chaîne de *prompt* par les commandes adaptées pour `tput` :

```
PS1=$(tput init; tput bold)${PS1}$(tput rmso)
```

Comment disposer de l'option `-n` de `echo` de manière portable ?

En faisant appel à `printf`(1), on peut définir une version personnelle portable de `echo` :

```
function mon_echo
{
    if [ "$1" = "-n" ]; then
        shift
        printf "%s" "$*"
    else
        printf "%s\n" "$*"
    fi
}
```

Comment remplacer dans un fichier chaque occurrence d'une ligne donnée par un texte de plusieurs lignes ?

Il est possible de regrouper les commandes nécessaires pour l'éditeur `ed` dans un fichier (nommé par exemple `remplacement.ed`) :

```
g/expression rationnelle décrivant la ligne à remplacer/c\
première ligne du texte de remplacement\
seconde ligne du texte\
troisième ligne du texte de remplacement\
.
w
q
```

On utilisera ce fichier en invoquant `ed` ainsi :

```
$ ed fichier_a_traiter < remplacement.ed
$
```

Notons que d'autres solutions sont possibles, notamment en utilisant `sed` par exemple.

Bibliographie

- Christophe Blaess - *Scripts sous Linux* - Editions Eyrolles, janvier 2004.
- Christophe Blaess - *Programmation système en C sous Linux* - Editions Eyrolles, mai 2000.

Unix

- Steve Bourne - *Le système Unix* - InterÉditions, 1985. Traduction française par Michel Dupuy (titre original The Unix System).
- AElen Frish - *Les bases de l'administration système* - O'Reilly & Associates, 1995. Traduction française par Céline Valot (titre original Essential System Administration).
- Jean-Marie Rifflet - *La programmation sous Unix* - Édiscience International, 1995.

Linux

- Matt Welsh et Lar Kaufman - *Le système Linux* - O'Reilly & Associates, 1995. Traduction française par René Cougnenc (titre original Running Linux).
- Éric Dumas - *Le guide du ROOTard pour Linux* - ftp ://ftp.lip6.fr/pub/linux/french/docs/GRL/Guide_du_Rootard/.
- Mike Loukides et Andy Oram - *Programmer avec les outils Gnu* - O'Reilly & Associates, 1997. Traduction française par Manuel et Nat Makarévitch (titre original Programming With Gnu Software).

Shell

- Dave Taylor - *100 scripts shell Unix* - Editions Eyrolles, 2004.
- Cameron Newham et Bill Rosenblatt - *Le shell Bash* - O'Reilly & Associates, 1995. Traduction française de René Cougnenc (titre original Learning the Bash Shell).
- Bill Rosenblatt - *Learning the Korn Shell* - O'Reilly & Associates, 1994.
- Paul Dubois - *Using csh & tcsh* - O'Reilly & Associates 1995.
- Giles Orr - *Bash Prompt Howto v0.60* - Linux Documentation Project 1999.

Sed / Awk

- Dale Dougherty et Arnold Robbins - *Sed & Awk* - O'Reilly & Associates, 1990