Programmation Unix 1 – cours n°4

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2016

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°4

1. Processus Unix et sous-shell
2. Variables d'environnement
3. Document en ligne
4. Commande reliée à un tube
5. Évaluation arithmétique

1 - Processus Unix et sous-shell

Un processus =

- programme en cours d'exécution
- contexte d'exécution
- données du programme

Unix = multi-tâche :

- temps partagé (petits intervalles de temps)
- préemptif

L'ordonnanceur (uk: scheduler) :

- module du noyau
- chargé de l'avancement de l'exécution des processus
- calcul dynamique des priorités

Filiation

Arbre des processus :

- Chaque processus a un parent, et 0, 1 ou + fils
- Le processus racine est `init`
- `init` adopte automatiquement les orphelins (processus dont le père est terminé)

Pour afficher la table des processus : `ps` ou `pstree`

Selon système : `ps -aux` , `ps aux` , `ps -edalf`

Propriétés d'un processus

- Identifiant unique : le PID (Process ID)
- Identifiant de son parent : le PPID
- Propriétaire et groupe
- Terminal d'attachement
- Priorité, les temps d'exécution
- Variables d'environnement
- Redirections
- État
- Données
- Programme
- etc

Création de processus

Une seule façon de créer un processus : par duplication (clonage) avec la fonction C `fork()`.

Un processus qui se duplique devient père, le nouveau processus est son fils.

Le fils hérite d'une copie des données du père, et du programme en cours d'exécution, puis continue.

Il n'y a pas de partage des données : chaque processus travaille sur les siennes, n'a pas d'accès aux autres.

Sous-shell

Un sous-shell = shell fils du shell courant

Exécuter des `commandes` dans un sous-shell : `(commandes)`

Un sous-shell permet d'isoler des opérations :

```
$ pwd
/home/thiel/unix
$ (cd .. ; pwd)
/home/thiel
$ pwd
/home/thiel/unix
```

La durée de vie du sous-shell n'excède pas `")"` :

```
$ (cd ..) ; (pwd)
/home/thiel/unix
```

PID du sous-shell

Le PID du script ne change pas dans tout le script :

```
$ echo $$; (echo $$)
1234
1234
```

Voir le PID du (sous-)shell : variable automatique `BASHPID`

```
$ echo $BASHPID; (echo $BASHPID)
1234
1569
```

ou encore :

```
$ declare -p BASHPID ; (declare -p BASHPID)
declare -ir BASHPID="1234"
declare -ir BASHPID="1877"
                                # -ir = integer read-only
```

Héritage, non partage

Le sous-shell hérite des variables du père :

```
$ a="ga" ; (declare -p a)
declare -- a="ga"
```

mais il n'y a pas de partage inverse :

```
$ (a="bu") ; declare -p a
declare -- a="ga"
```

Substitution de commandes

La substitution de commande `$(...)` est faite dans un sous-shell :

```
$ echo $BASHPID ; p=$(echo $BASHPID) ; echo $p
1234
5618
```

Syntaxe bien choisie :

- `(...)` : sous-shell
- `$` : substituer par

Recouvrement = exécution d'un programme

L'exécution d'un programme / commande / script se fait en 3 temps :

- création d'un subshell (fonction `fork` du C) ;
- *recouvrement* du subshell (fonction `exec` du C) ;
- le père attend la fin du fils recouvert (fonction `wait` du C).

→ Les builtin de bash sont plus efficaces que les commandes externes.

Effets du recouvrement

- Toutes les données sont écrasées ;
- on conserve le PID, le PPID et les redirections ;
- on reçoit les variables d'environnement (vues ensuite).

Exemple :

```
$ a=ga
$ bash -c "declare -p a"
bash: ligne 0 : declare: a : non trouvé
```

→ Une variable simple n'est pas transmise au processus recouvert.

2 - Variables d'environnement

Les variables d'environnement sont transmises aux processus fils *et* aux processus recouverts.

```
$ printenv
HOME=/home/thiel
USER=thiel
PATH=/bin:/usr/bin:/home/thiel/bin
PWD=/home/thiel/unix
DISPLAY=:0
SHELL=/bin/bash
...
```

Initialisation :

```
/etc/profile
$HOME/.profile
```

Export de variables

Les variables d'environnement se manipulent comme les variables :

```
echo "$PWD" ; PWD="/etc"
```

Créer une variable d'environnement : `export nom_var[=valeur]`

```
$ a=ga
$ bash -c "declare -p a"
bash: ligne 0 : declare: a : non trouvé
$ export a
$ bash -c "declare -p a"
declare -x a="ga"                    # -x : exportée
```

Comme pour les variables simples, pas de partage inverse.

```
$ bash -c "a=bu"
$ echo "$a"
ga
```

Export de fonctions

On peut aussi exporter des fonctions :

```
export -f ma_fonction
```

Exemple :

```
$ yolo () { echo "Carpe diem" ;}
$ (yolo)                          # dans un sous-shell
Carpe diem
$ bash -c "yolo"                  # dans proc recouvert
bash: yolo : commande introuvable
$ export -f yolo
$ bash -c "yolo"
Carpe diem
```

3 - Document en ligne

uk : here document

- Permet d'embarquer un fichier dans un autre :
  - sortie : génération de scripts, pages web, etc
  - entrée : automatisation de commandes interactives
- Redirection spéciale :

```
<< motfinal
...
motfinal
```

Le shell redirige l'entrée standard avec les lignes qui suivent << jusqu'à la ligne contenant uniquement `motfinal`
(rien avant, rien après).

Substitutions

Les substitutions sont effectuées dans le document en ligne.

```
$ cat << STOP
Mon répertoire est $HOME
La taille est $(du -sk $HOME 2> /dev/null | cut -f 1)
STOP
```

```
Mon répertoire est /home/thiel
La taille est 30761728
```

Ne pas indenter

Génération de script

```
$ cat >| monscript.sh << FINSCRIPT
#! /bin/bash
# Script généré le $(date)
echo "Il y a \$# paramètres"
exit 0
FINSCRIPT

$ cat monscript.sh
#! /bin/bash
# Script généré le dimanche 12 octobre 2014
echo "Il y a $# paramètres"
exit 0

$ chmod +x monscript.sh

$ ./monscript.sh ga bu
Il y a 2 paramètres
```

Génération de page web

```
titre="Ma page web"
cat >| mapage.html << FINPAGE
<html>
  <head>
    <title>$titre</title>
  </head>
  <body>
    <h1>$titre</h1>
    <p>Le répertoire $(pwd) contient :</p>
    <ul>
$(
  for f in * ; do
    echo "      <li><a href=\"$f\">$f</a></li>"
  done
)
    </ul>
  </body>
</html>
FINPAGE
```

Automatisation de saisie

Soit le script `naissance.sh` :

```
#! /bin/bash
echo -n "Jour ? " ; read jour
echo -n "mois ? " ; read mois
echo -n "An   ? " ; read an
echo "Date naissance : $jour/$mois/$an"
exit 0
```

On peut l'appeler ainsi :

```
./naissance.sh << FIN
25
12
1901
FIN
```

Triple chevron

Permet la redirection d'une chaîne sur l'entrée standard.
uk : here string

```
<<< "ligne de texte"
```

est équivalent à

```
<< FINTEXTE
ligne de texte
FINTEXTE
```

Exemples :

```
cat <<< "Sucre"                    # echo "Sucre"
sort -r > tmp <<< "$(ls)"          # ls | sort -r > tmp
```

Permet le "retournement de tube" (vu ensuite)

4 - Commande reliée à un tube

Chaque commande reliée à un tube
est lancée dans un sous-shell

Variables modifiées autour d'un tube $\rightarrow$ perdues pour le shell

```
a=1 ; a=2 | a=3 ; echo "$a"          # 1
```

- Lecture d'un résultat :

```
grep toto fichier | wc -l | read n   # n indéfini
n=$(grep toto fichier | wc -l)       # solution
```

- Lecture de plusieurs valeurs :

```
echo 1 2 3 | read a b c              # mauvais.
read a b c <<< $(echo 1 2 3)         # retournement
                                     # de tube
```

Traitement ligne à ligne

Pour traiter ligne à ligne la sortie d'une commande :

```
commande | while read ligne
do
  traitement $ligne
done
```

Le `while .. do .. done` est effectué dans un sous-shell
$\rightarrow$ les variables dans `do .. done` ne sortent pas.

Solution (si besoin de ces variables) : retournement de tube

```
while read ligne
do
  traitement $ligne
done <<< "$(commande)"                # "" importantes
```

Exemple : compter les lignes

```
n=0
ls -l | while read ligne
do
  n=$(expr $n + 1)
done
echo "$n"                    # 0
```

Solution :

```
n=0
while read ligne
do
  n=$(expr $n + 1)
done <<< "$(ls -l)"
echo "$n"                    # 231  ok
```

5 - Évaluation arithmétique

Expressions arithmétiques en bash :

- exclusivement avec des entiers
- opérateurs et syntaxe du C
- pas de découpage sur les blancs
- on peut omettre les $ si non-ambigu

Deux formes : `(( ))` et `$(( ))`

`man bash` : section Arithmetic evaluation

Substitution arithmétique

```
$((expression))
```

est substitué par la valeur de l'expression.

```
$ echo $((20+30))
50
$ x=20 ; y=30 ; echo $(($x+$y))
50
```

On peut omettre les $:

```
$ echo $((x+y))
50
```

... sauf en cas d'ambiguité :

```
$ set 5 ; echo $((x+$1))
25
```

Calculs

Priorités respectées :

```
$ echo $((10-3*(4+5)))
-17
```

Division entière :

```
$ echo $((17/7))
2
$ echo $((17.0/7))
bash: 17.0/7 : erreur de syntaxe : opérateur arith-
métique non valable (le symbole erroné est ".0/7")
```

Pour les nombres réels, utiliser la commande `bc` :

```
$ echo "scale=8; 17/7" | bc
2.42857142
$ bc <<< "scale=4; sqrt(2)"
1.4142
```

Expressions du C

Opérateurs logiques $\rightarrow$ 0 ou 1

```
$ echo $((10==10))
1
```

Opérateur ternaire :

```
$ echo $((3>=5 ? 3:5))
5
```

Expression avec virgules : évaluée de gauche à droite, valeur à droite

```
$ echo $((x=y=4, y++, x+y))
9
```

Exemple avec fonction

Le nombre de Frobenius de a et b, premiers entre eux, est le plus grand entier qui ne peut être obtenu par combinaison linéaire positive de a et b.

```
$ frobenius() # a b
  {
     local a="$1" b="$2"
     local f=$((a*b-a-b))
     echo $f
  }
$ frobenius 3 4
5
$ k=$(frobenius 5 7) ; echo $k
23
$ echo $((6+$(frobenius 3 7)))
17
```

Ajouts par rapport au C

Calcul de puissances :

```
$ echo $((2**5))
32
```

Bases : 2 à 64

```
$ echo $((0100))          # base 8   C : idem
64
$ echo $((0x100))         # base 16  C : idem
256
$ echo $((2#100))         # base 2   C GNU : 0b100
4
$ k=5; echo $(($k#100))   # $ obligatoire pour
25                        # développer $k avant
```

Réussite d'une expression

`((expression))` réussit si l'expression est vraie, c-à-d $\neq 0$

```
$ ((1)) ; echo $?          # vrai
0                          # succès
$ ((0)) ; echo $?          # faux
1                          # échec
```

Usage :

```
if ((expression)); then .. ; fi

while ((expression)); do .. ; done

fonction() { .. ; ((expression)) ;}
```

Exemples

```
((x = 3, y=x+2))
((x++)) ; y=$((y+1))

if (($#<2)); then
  echo "2 arguments attendus" > /dev/stderr ; exit 1
fi

est_positif() # x
{ local x="$1"
  ((x>=0))
}
if est_positif "$y"; then .. ; fi

i=10
while est_positif "$i"; do echo "$i"; ((i--)); done
```

La boucle `for` du C

```
for ((expr1; expr2; expr3)); do .. ; done
```

Comme en C : équivalente à

```
((expr1))
while ((expr2)); do
  ..
  ((expr3))
done
```

Exemple :

```
$ for ((i=0,j=5;i<j;i++,j--)); do echo "$i $j"; done
0 5
1 4
2 3
```

Expression vide

Une expression vide est fausse :

```
$ echo $(())
0
$ (()); echo $?
1
```

Sauf dans un `for` où elle s'évalue à 1

→ `for ((;;)); do .. ; done` = boucle infinie

Donc l'équivalence entre le `for` et le `while` est fausse dans ce cas.