

Environnement de développement sous Linux

Module 2I012-2018fev

Dominique Béréziat (Dominique.Bereziat@lip6.fr),
Valérie Ménissier-Morain



Troisième partie III

L'interprète de commandes `bash`

Plan

Introduction

Le shell, pour quoi faire ?

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Pour conclure

Plan

Introduction

Le shell, pour quoi faire ?

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

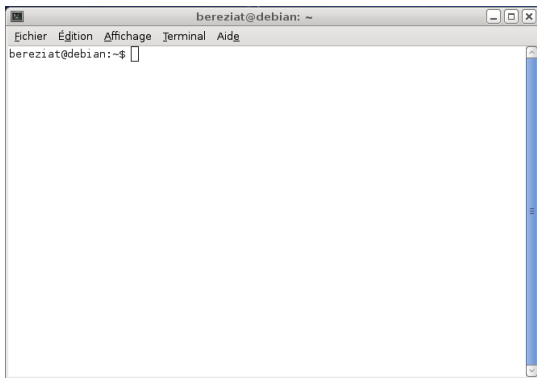
Pour conclure

Les missions du shell

- ▶ Le *shell* = l'interprète de commandes (= `bash` dans ce cours)
- ▶ Une commande = un programme = un processus qui s'exécute sur une machine
- ▶ Ses missions :
 - ▶ interface de l'utilisateur avec le système : il lance les commandes, manipule les fichiers, ... Possède des mécanismes sophistiqués d'édition des commandes (historique, complétion)
 - ▶ scripts : lancer de façon automatisée un ensemble de commandes notamment : le démarrage du système Linux
- ▶ Dans ce cours, nous apprendrons notamment à :
 - ▶ bien utiliser le shell
 - ▶ à écrire de beaux scripts

Le shell concrètement

- En mode interactif :
 - le processus "gnome" lance un processus "émulateur de terminal texte" qui lance un processus "bash" :



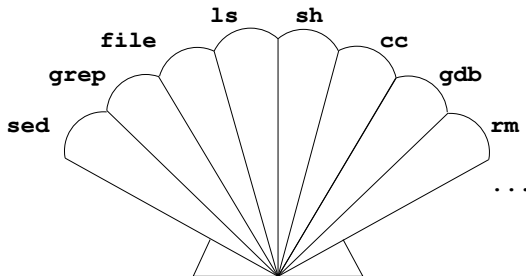
- en attente des ordres de l'utilisateur ...
- En mode non interactif : un programme exécutant des ordres (scripts)

Shell, pourquoi ce nom ?

- ▶ Le shell a été créé en même temps qu'Unix et le langage C (années 1975)
- ▶ Le système repose alors sur une trinité :
 - ▶ des commandes (nombreuses, simples et spécialisées)
 - ▶ le shell qui permet de lancer les commandes et de les combiner (en général à l'aide de scripts, mais pas nécessairement)
 - ▶ le compilateur (C) n'est là que pour palier à l'absence d'une commande pour une nouvelle tâche spécialisée
- ▶ Pourquoi réinventer la roue ?

Shell, pourquoi ce nom (suite) ?

- *seashell*, la coquille Saint-Jacques, chaque branche offre un service (une commande)



- Pour en savoir plus (un classique) : *The Unix Programming Environment* par B. Kernighan (inventeur du langage C) :
<http://cs.uwec.edu/~buipj/teaching/cs.388.s14/static/pdf/upe.pdf>

Histoire du shell

- ▶ `sh` : avec Unix v6 (1975) écrit par Thomson
- ▶ `bsh` : Unix v7 : écrit par Bourne (Bourne shell, 1977) apportant la plupart des caractéristiques modernes du shell (redirection, structures de contrôle, gestionnaire de tâche, ...)
- ▶ `csch` : version de `sh` avec syntaxe à la C (C-shell)
- ▶ `tcsh` : version améliorée de `csch` (complétion notamment), (Tenex C-shell)
- ▶ `bash` : réécriture moderne de `bsh` (Bourne Again Shell) il s'est imposé comme l'interpréteur par défaut
- ▶ `ksh` : Korn shell - un shell compatible avec `bash` et qui incorpore quelques éléments de `csch`. Supporte aussi le calcul flottant, la programmation objet, et peut être "compilé à la volée" ce qui le rend plus performant
- ▶ `zsh` : un `bash` avec des interactions utilisateurs plus puissantes

Les shells aujourd'hui

- ▶ `sh` est normalisé (POSIX)
- ▶ `sh` est souvent un alias de `bash`
- ▶ On peut trouver des implémentations POSIX de `sh` qui ne soient pas `bash`, par exemple `dash` *Debian Almquist Shell*
- ▶ Un shell évolué demande plus de ressource :

```
% wc -c < `which dash`  
109768  
% wc -c < `which bash`  
955024  
% bash --version | head -n 1  
GNU bash, version 4.2.24(1)-release (x86_64-pc-linux-gnu)
```

- ▶ `csh` et `tcsh` sont obsolètes, très peu utilisés (dernière mise à jour 2012 pour `tcsh`)
- ▶ `bash` est le standard
- ▶ `ksh` et `zsh` utilisés et maintenus.

Choix du shell

► Vous avez donc le choix !

```
% echo $SHELL      # nom de votre shell
% cat /etc/shells  # liste des shells dispo (Debian)
/bin/sh
/bin/dash
/bin/bash
/bin/rbash          # bash -r (mode restreint)
/usr/bin/fish        # friendly interactive shell
% chsh             # pour choisir son shell favori
```

► Le shell par défaut est indiqué dans le fichier /etc/passwd

```
% grep bereziat /etc/passwd
bereziat:x:1000:1000:Dominique Bereziat,,,:
/home/bereziat:/bin/bash
```

Plan

Introduction

Notions de base sur l'OS Linux

- L'OS Linux

- Le système de fichiers

- La mémoire et les processus

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Pour conclure

Plan

Introduction

Notions de base sur l'OS Linux

- L'OS Linux

- Le système de fichiers

- La mémoire et les processus

L'interprète de commandes

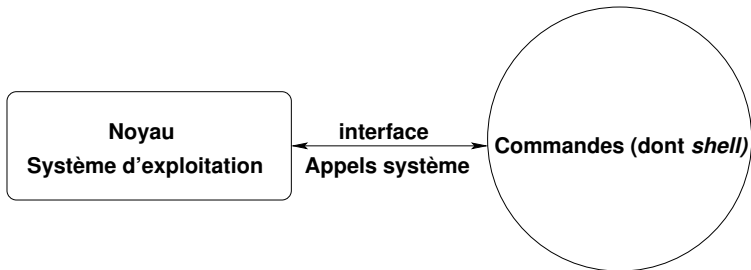
L'écriture de scripts

Scripts avancés

Pour conclure

Le système d'exploitation Linux

- Implémentation **libre** d'un *Unix-like*
- Le noyau linux est le premier processus lancé. Tous les autres processus n'existent que dans le processus noyau
- On n'accède aux fonctionnalités de la machine qu'au travers des services offerts par le noyau, **et jamais directement !**



- L'ensemble de ces services constitue le système d'exploitation

La documentation

- ▶ Linux est un système documenté : commandes, appels systèmes, ...
- ▶ La commande `man` (*manual*), on parle aussi de *manpages Unix*
- ▶ Les documentations sont classées en sections (voir TME et `man man`) :

Section	Contenu
1	Programmes exécutables ou commandes de l'interpréteur de commandes (shell)
2	Appels système (fonctions fournies par le noyau)
3	Appels de bibliothèque (fonctions fournies par les bibliothèques des programmes)
4	Fichiers spéciaux (situés généralement dans <code>/dev</code>)
5	Formats des fichiers et conventions. Par exemple <code>/etc/passwd</code>
6	Jeux
7	Divers (y compris les macropaquets et les conventions), par exemple <code>man(7)</code> , <code>groff(7)</code>
8	Commandes de gestion du système (généralement réservées au super-utilisateur)
9	Sous-programmes du noyau [hors standard]

Le système de fichiers (1)

Particularité de Linux

- ▶ Sous Linux, **tout est fichier** : fichier ordinaire, répertoire, processus, connexion réseau, terminal, périphérique
- ▶ On peut en principe interagir avec le système à l'aide des fonctions d'entrées/sorties (`open`, `read`, `write`, `close`)
- ▶ Le système de fichiers est **unifié** (racine `/`), même si plusieurs disques (locaux ou distants) sont montés : chemin d'accès unique
- ▶ Le système de fichiers unifié est organisé (`/bin`, `/usr`, `/usr/bin`, `/usr/local`, ...) selon une convention (des différences mineures existent d'une distribution à l'autre)
- ▶ Certains fichiers sont **permanents**, on les retrouve si on redémarre le système, et d'autres non.

Le système de fichiers (2)

Particularité de Linux

- ▶ Tout fichier possède un **chemin**
- ▶ Un chemin est une suite de noms séparés par des / et décrit la position du fichier :
 - ▶ par rapport à la racine : **chemin absolu**, il commence donc par /
 - ▶ par rapport au répertoire **courant** : **chemin relatif**
- ▶ Répertoire **courant** : c'est le répertoire où l'on travaille, il s'appelle .
- ▶ Chaque répertoire a un répertoire **parent** : . .
- ▶ Un fichier possède des droits : droit de lecture, d'écriture, . . .
le système de droit dépend de la nature du système de fichiers : nous parlons dans ce cours du système `ext4`, mais il en existe pleins d'autres !

Parcours dans l'arborescence de fichiers

- Commandes de l'interprète à connaître (TME) : `cd`, `pwd`, `ls`
- Exemples :
 - lorsqu'on ouvre un terminal texte, le répertoire courant est le répertoire de travail de compte de l'utilisateur :

```
$ pwd  
/home/bereziat
```

- et déplacement dans l'arborescence :

```
$ cd .  
$ pwd  
/home/bereziat  
$ cd ..  
$ pwd  
/home  
$ ls  
bereziat dominique  
$ ls -l  
total 8  
drwxr-xr-x 24 bereziat bereziat 4096 oct. 28 11:49 bereziat  
drwxr-xr-x 17 dominique dominique 4096 oct. 28 11:50 dominique  
$ cd bereziat  
$ cd /tmp
```

Sur la sortie de `ls -la`

```
% ls -la
drwxr-xr-x  3 bereziat bereziat 4096 oct.  28 14:50 .
drwxr-xr-x 25 bereziat bereziat 4096 oct.  28 14:49 ..
-rw-r--r--  1 bereziat bereziat   0 oct.  28 14:50 fic
drwxr-xr-x  2 bereziat bereziat 4096 oct.  28 14:50 rep
-----
droits      ^    ^          groupe      ^    horodatage    nom fichier
            |    |
            |    +- propriétaire    +-- taille (octets)
            +- nb de liens physiques
```

- ▶ L'horodatage : par défaut date de dernière modification.
- ▶ Droits (dix caractères) :
 - ▶ le premier caractère indique le type du fichier :
 - ▶ - pour les fichiers ordinaires,
 - ▶ d pour un répertoire (*directory*),
 - ▶ l pour un lien symbolique,
 - ▶ b, c, p, s fichiers spéciaux, non traité dans ce cours
 - ▶ 3 caractères pour le propriétaire, - signifie l'absence de droit :
 - ▶ r droit de lecture (*read*)
 - ▶ w droit d'écriture (*write*)
 - ▶ x droit d'exécution (*eXec*) ou de parcours du répertoire
 - ▶ puis 3 pour le groupe d'utilisateurs
 - ▶ puis 3 pour les autres utilisateurs (*others*)

Gestion des groupes Unix

- ▶ Groupe unix : un ensemble d'utilisateurs
- ▶ Permet d'autoriser des droits supplémentaires pour les utilisateurs du groupe à certains fichiers
- ▶ `groups` liste les groupes auquel l'utilisateur appartient :

```
% groups  
berezia cdrom floppy sudo audio dip video plugdev netdev scanner blu
```

- ▶ `chgrp` permet de changer l'appartenance d'un fichier à un groupe
- ▶ La notion de groupe n'est pas utile dans ce cours, nous vous renvoyons à l'UE système 2I010 pour en savoir plus

Gestion des droits : la commande `chmod`

- ▶ `chmod` : change les droits d'accès d'un fichier. Deux modes de paramétrage
- ▶ Mode octal :
 - ▶ 3 chiffres en base huit :
 - ▶ 1er chiffre : droits du propriétaire
 - ▶ 2nd chiffre : droits pour le groupe
 - ▶ 3eme chiffre : droits pour les autres
 - ▶ un chiffre en base 8 code les droits :
 - ▶ 0 : aucun droit
 - ▶ 1 : droit d'exécution (bit 1)
 - ▶ 2 : droit d'écriture (bit 2)
 - ▶ 4 : droit de lecture (bit 3)
 - ▶ en addition ces chiffres, on obtient la combinaison de droits souhaitée (ex : 2+4=6 soit lecture et écriture)
- ▶ Mode symbolique : un ou plusieurs mots de syntaxe :
[uoga] [-+=] [rwx]
 - ▶ + ajoute le(s) droit(s), - le(s) retire(s), = ajoute le(s) droit(s) et retire les autres
 - ▶ u pour *User*, g pour *Group*, o pour *Others*, a pour *All*
 - ▶ si rien devant -, + ou = synonyme de *All* mais suis la valeur d'`umask`
- ▶ Il existe d'autres droits (X, t, s) : inutile dans ce cours

Des exemples avec chmod

- Mettre le droit d'exécution à un script :

```
|| % chmod +x monscript
```

- Rendre privée toute une arborescence avec l'option récursive -R :

```
|| % chmod -R go-rwx mondossier
```

- Donner tous les droits à tout le monde à un fichier :

```
|| % chmod 777 fichier  
| # ou encore  
|| % chmod a=rwx fichier
```

- Droit d'écriture pour soi, et lecture pour tout le monde :

```
|| % chmod 644 fichier  
|| % chmod u+w,a+r fichier
```

- Droit de lecture pour soi et le groupe, pas pour les autres :

```
|| % chmod ug+r,o-w fichier
```

- A savoir : un dossier doit avoir les droits `r` et `x` pour être parcouru.

Gestion des droits des fichiers : la commande `umask`

- ▶ C'est la commande la plus importante à connaître : droits par défaut des fichiers créés !
 - ▶ retourne le masque des droits par défaut
 - ▶ change le masque des droits par défaut
- ▶ Masque de droits :
 - ▶ par défaut un fichier a les droits `666` et un répertoire les droits `777`
 - ▶ le fichier sera alors créé avec les droits `droit & ~mask` ou `droit` vaut `666` ou `777` et `mask` est la valeur retournée par `umask`
 - ▶ revient à soustraire (retirer, masquer) les droits
- ▶ Exemple : avec le masque `022`, les fichiers auront les droits `644` (`666` moins les droits `022`) et les dossier les droits `755`
- ▶ Usage de la commande `umask` :
 - ▶ en mode octal : `chmod 0000` le premier chiffre est utilisé pour les droits spéciaux, on le laisse à zéro, les 3 autres sont comme dans `chmod` (mais n'ont pas la même signification car c'est un masque de bit). Le premier chiffre peut être omis
 - ▶ en mode symbolique, comme `chmod` et ont la même signification

Des exemples avec umask

► Exemples courants :

```
% umask 0022
% umask -S
u=rwx,g=rx,o=gx
% touch t
% ls -l t
-rw-r--r-- 1 bereziat bereziat 0 oct. 28 15:27 t
% umask 0002
% umask -S
u=rwx,g=rwx,o=gx
% rm t; touch t
% ls -l t
-rw-rw-r-- 1 bereziat bereziat 0 oct. 28 15:28 t
```


Des exemples avec umask

► Mode parano (pour la PPTI) :

```
% umask 0066
% touch a
% ls -l a
-rw----- 1 bereziat bereziat    0 oct.  28 15:36 a
```

C'est un peu extrême : je recommande d'avoir un dossier privé et un dossier public et de laisser le masque à 022

► Modes absurdes :

```
% umask 0777
% touch t
% ls -l t
----- 1 bereziat bereziat 0 déc.   3 11:15 t
% rm t
rm : supprimer fichier vide (protégé en écriture) « t » ? y
% umask 0000
% touch t
% ls -l t
-rw-rw-rw- 1 bereziat bereziat 0 déc.   3 11:18 t
```

Gestion des fichiers

- ▶ Manipuler des fichiers/répertoires (il y a des options ! pensez au `man`) :
 - ▶ `cp` (*CoPy*) : copier des fichiers, des répertoires (`cp -r`)
 - ▶ `mv` (*MoVe*) : déplacer, renommer des fichiers ou des répertoires
 - ▶ `rmdir` (*ReMove DIRectory*) : supprime un répertoire (doit être vide)
 - ▶ `rm` (*ReMove*) : supprime un fichier (attention : irréversible)
- ▶ Examiner des fichiers :
 - ▶ `ls` (*LiSt catalog*)
 - ▶ `more`, `less` : pour des fichiers textes
 - ▶ `file` : des informations sur la nature du fichier (utilise le fichier `/etc/magic`, voir `man magic`)

Motifs du shell

- ▶ Séquences de caractères interprétées par `bash` pour désigner une liste de fichiers effectivement présents dans le répertoire courant
- ▶ Même notion que le 'masque' de fichiers sous DOS/Windows
- ▶ Combinables avec des caractères ordinaires ou entre eux
- ▶ Motifs existants :
 - ▶ `*` n'importe quel groupe de caractères
 - ▶ `?` n'importe quel caractère
 - ▶ `[liste]` un caractère parmi la liste indiquée, la liste peut être une plage du type `a-z`
- ▶ Exemples :
- ▶ Si un motif n'est pas trouvé, il est gardé tel quel !

```
% ls /etc/*
% ls /etc/*.conf
% ls /etc/p*
% ls /etc/*.??
% rm *~ *. [oa]
% ls */*/*.c
```

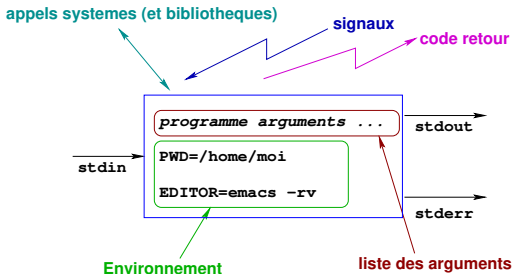
```
% ls
hello.c world.c
% echo *.c
hello.c world.c
% echo les fichiers *.tata n'existent pas
les fichiers *.tata n'existent pas
```

La mémoire, les processus et les flux

- ▶ La mémoire vive du système est le lieu de vie des **processus**
- ▶ Un **processus** est un programme chargé en mémoire et qui interagit avec le système d'exploitation et éventuellement l'utilisateur (via les périphériques)
- ▶ Certains **processus** sont permanents (le noyau linux, les démons) d'autres sont **temporaires** : un début et une fin d'exécution
- ▶ La mémoire est également le lieu de transit des **flux** : les flux sont les données échangées par les processus ou avec le noyau (entrées/sorties)
- ▶ Linux est organisé autour des processus, des flux et du système de fichiers

Les processus

► Les interactions du processus avec l'OS :



- les arguments
- le code retour
- l'environnement (variables d'environnement)
- le flux d'entrée (stdin)
- les flux de sortie (stderr, stdout)
- les signaux
- les appels systèmes

Les arguments

- ▶ Ils permettent, soit de paramétrer la commande, soit de fournir des données à traiter
- ▶ Ils sont fournis par l'utilisateur via l'interface du shell
- ▶ En C, le système `argc/argv`.
- ▶ Exemple, implémentation de la commande `echo` :

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char **argv) {
5      int nl = 1;
6      if( argc > 1 && !strcmp(argv[1], "-n")) nl=2;
7      for( int i=nl; i<argc; i++)
8          printf("%s_", argv[i]);
9      if( nl == 1) printf("\n");
10     return 0;
11 }
```

Les arguments

```
% gcc echo.c -o echo  
% ./echo Bonjour le monde  
Bonjour le monde  
% ./echo -n Bonjour le monde  
Bonjour le monde% echo Bonjour le monde  
Bonjour le monde  
% echo -n Bonjour le monde  
Bonjour le monde%
```

Le code de retour

- ▶ Par convention, toute commande retourne une valeur 0 signifiant au système : "pas d'erreur"
- ▶ Une valeur non nulle indique une erreur et possiblement un code d'erreur (si documenté)
- ▶ Illustration en C avec les deux commandes les plus simple à écrire : `true` et `false` et la variable spéciale `$?`

```
1 int main() { return 0; }
```

```
1 int main() { return 1; }
```

```
% which true  
/bin/true  
% ./true  
% echo $?  
0
```

```
% which false  
/bin/false  
% ./false  
% echo $?  
1
```


L'environnement

- Ensemble de variables héritées du processus père
- Exemple avec la commande `printenv`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char **argv, char **env) {
4     if( argc == 1)
5         for( ; *env; puts(*env++));
6     else
7         puts(getenv(argv[1]));
8     return 0;
9 }
```

```
% ./printenv | head -n 6
XDG_VTNR=7
SSH_AGENT_PID=1095
XDG_SESSION_ID=1
GPG_AGENT_INFO=/run/user/1000/keyring/gpg:0:1
TERM=xterm
SHELL=/bin/bash
% ./printenv USER
bereziate
```

- Voir aussi : `man 3 setenv`

Les flux d'entrée et de sortie standard (1)

- ▶ Le processus est vu comme un filtre : il traite les données reçues dans l'entrée standard (*stdin*) et écrit le résultat dans la sortie standard (*stdout*)
- ▶ Exemple avec la commande `cat` et les appels systèmes `read` et `write`. Ces fonctions utilisent un *file descriptor* (un entier) identifiant un fichier. Les valeurs 0, 1, et 2 sont réservées par le système. 0 est celui de l'entrée standard, 1 celui de la sortie standard.

```
1  #include <unistd.h>
2
3  #define BUFSIZE 512
4  char buf[BUFSIZE];
5
6  int main(int argc, char **argv) {
7      int fd;
8      while( read( 0, buf, BUFSIZE))
9          write( 1, buf, BUFSIZE);
10     return 0;
11 }
```

Les flux d'entrée et de sortie standard (2)

- ▶ d'où vient le flux d'entrée ? où va le flux de sortie ?
 - ▶ ces flux sont gérés par le shell (voir suite du cours)
 - ▶ au lancement de la commande, le shell ouvre pour le processus les trois descripteurs de fichiers 0, 1 et 2. Pas besoin donc de les ouvrir !
- ▶ Voir démo !

La sortie d'erreur

- ▶ C'est un second canal de sortie, en plus de la sortie standard.
- ▶ Son *file descriptor* est 2
- ▶ Il sert conventionnellement à afficher un diagnostic d'erreur
- ▶ Pourquoi ne faut-il pas utiliser la sortie standard pour afficher le diagnostic d'erreur ?

Les signaux

- ▶ Le système peut envoyer des signaux à un processus
- ▶ Signaux courants : suspension, terminaison, extinction, ...
- ▶ Le processus peut réagir explicitement à ces signaux (sinon il existe un comportement par défaut)
- ▶ Voir `kill`, `trap`
- ▶ Gestion des processus : voir `jobs`, `bg`, `fg`
- ▶ Vu un peu en TME, sinon, dans 2I010

Appels systèmes (1)

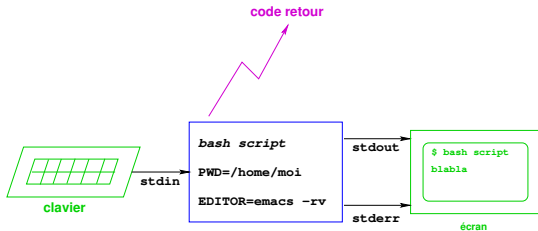
- ▶ Un processus peut tout faire (sans la mesure des permissions)
 - ▶ lire, écrire des fichiers, adresser certains périphériques
 - ▶ créer les sockets, des threads, des pipelines, ...
 - ▶ faire appel à d'autres bibliothèques
- ▶ Un exemple de fonction système simple : lecture d'un fichier et la commande `cat` version 2 :

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 #define BUFSIZE 512
5 char buf[BUFSIZE];
6
7 int main(int argc, char **argv) {
8     int i, fd;
9     for(i=1; i<argc; i++) {
10         fd = open( argv[i], O_RDONLY);
11         while( read( fd, buf, BUFSIZE))
12             write( 1, buf, BUFSIZE);
13         close(fd);
14     }
15     return 0;
16 }
```

Appels systèmes (2)

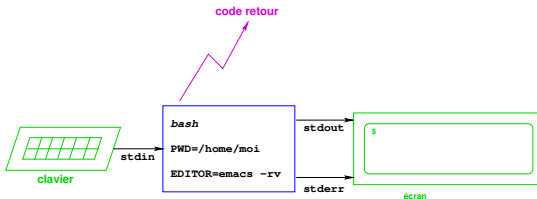
- ▶ Ici la commande `cat` ouvre elle-même les fichiers pour les lire et n'attend pas des données dans son entrée standard
- ▶ Autres appels systèmes : programmation système, voir 2I010

Le processus comme script shell



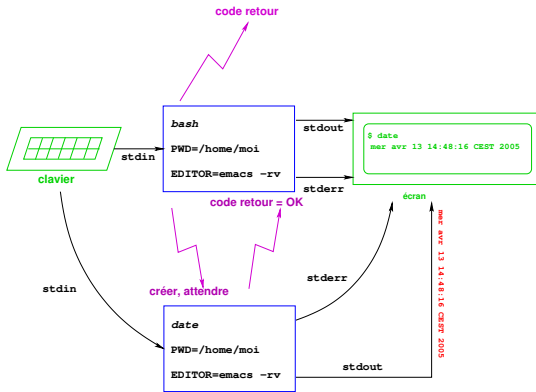
- L'entrée de `bash` est connectée au clavier (via l'émulateur de terminal `gnome-term`)
- La sortie standard et d'erreur est dirigée vers l'affichage textuel de l'émulateur de terminal

Le processus shell en mode interactif (1)



- Variante du cas précédent : `bash` est en attente de données en provenance de l'entrée standard, donc de l'émulateur de terminal, donc du clavier, donc de l'utilisateur

Le processus shell en mode interactif (2)



- ▶ Si `bash` lance une commande, il déconnecte son entrée standard pour la connecter à celle de la commande lancée
- ▶ Idem avec les sorties standard et d'erreur

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

Lancement des commandes

Redirections des flux

Composition des commandes

Interactions avec l'utilisateur

L'écriture de scripts

Scripts avancés

Pour conclure

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

Lancement des commandes

Redirections des flux

Composition des commandes

Interactions avec l'utilisateur

L'écriture de scripts

Scripts avancés

Pour conclure

Syntaxe des appels

- ▶ La ligne de commande est découpée en "mots", les espaces, les tabulations et les saut de lignes sont les séparateurs
- ▶ Le premier mot est la commande
- ▶ Les mots suivants sont les arguments et sont transmis à la commande, voir `argc`, `argv` en C
- ▶ Les arguments sont typiquement des noms de fichiers ou des options commençant conventionnellement par un `-`
- ▶ Exemples :
 - ▶ `ls -color=auto -l /var`
 - ▶ `man -k cat`
 - ▶ `cat -n macros.sty`
 - ▶ `ps -help`

Comment les mots deviennent des arguments ?

- Réécriture de la commande `echo` en C avec traçage des arguments :

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv) {
4      for( int i=1; i<argc; i++)
5          printf("argv[%d]=\"%s\"_", i, argv[i]);
6      printf("\n");
7      return 0;
8  }
```

- Observons le découpage opéré par `bash` et comment valeurs sont transmises à la commande :

```
% ./echoverb un homme pressé
argv[1]="un" argv[2]="homme" argv[3]="pressé"
```

- Les espaces entre les mots ne comptent pas

Où se trouve la commande ?

- Si le nom de la commande contient /, alors c'est un chemin (absolu ou relatif) indiquant où se trouve la commande :

```
/bin/ls, dir/toto, ./a.out
```

- Dans le cas contraire : la commande est recherchée comme *primitive* du shell (ex : `cd`, `pwd`, ...), puis dans les répertoires des commandes, indiquée par la variable d'environnement `PATH`

```
% printenv PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/game
```

- Une primitive est partie intégrante du shell, alors qu'une commande est un programme séparé du shell
- Si la commande n'est pas trouvée, le shell indique une erreur.

```
% toto
bash: toto : commande introuvable
```

- La *commande* `which` indique où se trouve la commande

```
% which ls
/bin/ls
```

Commande ou primitive ?

- La *primitive* type indique la nature d'une commande.

```
% type cd
cd est une primitive du shell
% type type
type est une primitive du shell
% type which
which est haché (/usr/bin/which)
% type file
file est /usr/bin/file
% type -a test
test est une primitive du shell
test est /usr/bin/test
```

- Dans ce cours, on parlera de **commandes** (même si c'est des primitives)

Commande ou primitive ou alias ?

- Un troisième de genre de commandes : les *alias*

```
% type -a rm
rm est /bin/rm
% alias rm='rm -i'
% type -a rm
rm est un alias vers « rm -i »
rm est /bin/rm
% cd; mkdir -p bin; touch bin/rm; chmod +x bin/rm
% PATH=$HOME/bin:$PATH
% type -a rm
rm est un alias vers « rm -i »
rm est /home/toto/bin/rm
rm est /bin/rm
% which -a rm
/home/toto/bin/rm
/bin/rm
```

Commandes essentielles

- ▶ Il est important d'abord de bien connaître certaines commandes simples mais essentielles
- ▶ Elles lisent des fichiers ou leur entrée standard et travaillent ligne par ligne
 - ▶ `cut` : extraction de mots
 - ▶ `sort` : tri des lignes d'un fichier
 - ▶ `uniq` : élimination des doublons de ligne
 - ▶ `head`, `tail` : premières lignes et dernières lignes
 - ▶ et toutes les autres vues en TD/TME et en cours
- ▶ Penser à :
 - ▶ lire leur documentation `man`, pour connaître leur paramétrage
 - ▶ faire les micro-sujets
- ▶ Pour les primitives du shell, utiliser `help` (ex : `help type`), `help` sans argument donne la liste des primitives documentées

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

Lancement des commandes

Redirections des flux

Composition des commandes

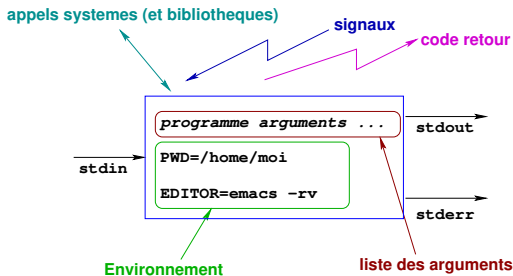
Interactions avec l'utilisateur

L'écriture de scripts

Scripts avancés

Pour conclure

Flux usuels



- l'entrée standard (*stdin*, numéro 0) peut être redirigée en lecture
- la sortie standard (*stdout*, numéro 1) peut être redirigée en écriture
- la sortie d'erreur (*stderr*, numéro 2) peut être redirigée en écriture
- il est possible d'avoir des flux de numéros supérieurs que le `bash` peut rediriger (vu en TD/TME)

L'entrée standard (<)

- ▶ Redirection de l'entrée standard à un fichier : `commande <fichier` permet de connecter l'entrée standard de la commande à un fichier
- ▶ `bash` ouvre le fichier en lecture, le lit et envoie les données dans l'entrée standard de `command`
- ▶ Exemple :

```
% wc < /etc/passwd
39    65 2214
```

- ▶ On peut aussi écrire `command 0< fichier` (car 0=stdin)

La sortie standard (>)

- Redirection de la sortie standard : `commande >fichier` permet de connecter la sortie standard de la commande à un fichier
- `bash` ouvre le fichier en écriture : il est créé s'il n'existe pas ou sinon son contenu est écrasé, lit les données dans le flux de sortie et les écrit dans le fichier
- Exemple

```
% mkdir t; cd t
% wc </etc/passwd >output
% ls -l
total 4
-rw-r--r-- 1 bereziat bereziat 15 nov.   8 22:11 output
% cat output
 39   65 2214
% true > output
% ls -l
total 0
-rw-r--r-- 1 bereziat bereziat 0 nov.   8 22:12 output
```

- On peut aussi écrire `command 1> fichier`

La sortie d'erreur (2>)

- Redirection de la sortie d'erreur : `command 2>fichier`
- `bash` ouvre le fichier en écriture, lit les données dans le flux d'erreur et les écrit dans le fichier
- Exemple :

```
% wc toto 2> output
% cat output
cat: toto: Aucun fichier ou dossier de ce type
% echo toto 2>output
toto
% cat output
%
```

Redirection en ajout (>>)

- Redirection en écriture

- pour la sortie standard : `commande >>output`
- pour la sortie d'erreur : `commande 2>>output`

- `bash` ouvre le fichier en mode ajout (`open (nom, O_APPEND)`), on écrit donc à la fin du fichier) et redirige les données dans ce fichier.

- Exemple :

```
% echo Hello > output
% echo World >> output
% cat output
Helllo
World
```


Duplication d'un descripteur de sortie (>&)

- ▶ Descripteur = numéro du flux
- ▶ Permet de rediriger un flux vers un autre (fonction système `dup()`)
- ▶ Syntaxe : `commande n>&m` où `n` et `m` sont deux descripteurs (en pratique 1 ou 2)
- ▶ `bash` fait les duplications dans l'ordre ...
- ▶ Deux usages (voir `man bash`)

- ▶ écrire dans la sortie d'erreur :

```
|| % echo message 1>&2
```

`stdout` est dupliqué dans `stderr`, donc ce qui est écrit dans `stdout` l'est dans `stderr`

- ▶ écrire les deux flux de sortie dans un même fichier

```
|| % command >log 2>&1
```

`stdout` est redirigé dans `log`, puis `stderr` est dupliqué dans `stdout` qui est `log`

Duplication d'un descripteur de sortie (suite)

- ▶ Que se passe-t-il si on écrit : commande `2>log 1>&2` ?
 - ▶ bash redirige `stderr` dans le fichier `log`, puis `stdout` est dupliqué dans `stderr` donc `stdout` est redirigé dans `log`
 - ▶ Cas équivalent à commande `>log 2>&1`
- ▶ Que se passe-t-il si on écrit : commande `2>&1 >log` ?
 - ▶ bash duplique `stdout` dans `stdout` qui est connecté au terminal (par défaut), dans le même d'argent ou dans le terminal puis bash redirige le `stderr` dans le fichier `log`.
 - ▶ Cas équivalent à commande `>log 2>log` car le `stderr` est par défaut dupliqué dans `stderr`.
 - ▶ La redirection `2>&1` est donc inutile.

Duplication d'un descripteur de sortie (suite)

- ▶ Que se passe-t-il si on écrit : commande `2>log 1>&2` ?
 - ▶ bash redirige `stderr` dans le fichier `log`, puis `stdout` est dupliqué dans `stderr` donc `stdout` est redirigé dans `log`
 - ▶ Cas équivalent à commande `>log 2>&1`
- ▶ Que se passe-t-il si on écrit : commande `2>&1 >log` ?

On a donc toujours `stderr` et `stdout` qui sont dupliqués dans le même fichier, mais dans un ordre différent. Dans le premier cas, `stderr` est dupliqué dans `log` puis `stdout` est dupliqué dans `stderr`. Dans le second cas, `stdout` est dupliqué dans `log` puis `stderr` est dupliqué dans `stdout`.

Ces commandes ne sont pas recommandées car elles peuvent être confuses. On préfère utiliser des redirections explicites pour chaque flux. Par exemple, pour rediriger `stderr` et `stdout` dans le fichier `log`, on utilise la commande `>log 2>&2`.

Duplication d'un descripteur de sortie (suite)

- ▶ Que se passe-t-il si on écrit : commande `2>log 1>&2` ?
 - ▶ bash redirige `stderr` dans le fichier `log`, puis `stdout` est dupliqué dans `stderr` donc `stdout` est redirigé dans `log`
 - ▶ Cas équivalent à commande `>log 2>&1`
- ▶ Que se passe-t-il si on écrit : commande `2>&1 >log` ?

bash duplique `stderr` dans `stdout` qui est connecté au terminal (par défaut), donc la sortie d'erreur va dans le terminal puis bash redige la sortie standard vers le fichier `log`.

Ces deux commandes sont équivalentes car le terme d'organe est par défaut `&1` (la sortie standard) et `&2` (la sortie d'erreur) est par défaut `&1`.

Duplication d'un descripteur de sortie (suite)

- ▶ Que se passe-t-il si on écrit : commande `2>log 1>&2` ?
 - ▶ `bash` redirige `stderr` dans le fichier `log`, puis `stdout` est dupliqué dans `stderr` donc `stdout` est redirigé dans `log`
 - ▶ Cas équivalent à commande `>log 2>&1`
- ▶ Que se passe-t-il si on écrit : commande `2>&1 >log` ?
 - ▶ `bash` duplique `stderr` dans `stdout` qui est connecté au terminal (par défaut), donc la sortie d'erreur va dans le terminal puis `bash` redige la sortie standard vers le fichier `log`
 - ▶ Cas équivalent à commande `>log` car la sortie d'erreur est par défaut dirigé vers la sortie standard
 - ▶ L'ordre des redirections est donc important !

Duplication d'un descripteur de sortie (suite)

- ▶ Que se passe-t-il si on écrit : commande `2>log 1>&2` ?
 - ▶ `bash` redirige `stderr` dans le fichier `log`, puis `stdout` est dupliqué dans `stderr` donc `stdout` est redirigé dans `log`
 - ▶ Cas équivalent à commande `>log 2>&1`
- ▶ Que se passe-t-il si on écrit : commande `2>&1 >log` ?
 - ▶ `bash` duplique `stderr` dans `stdout` qui est connecté au terminal (par défaut), donc la sortie d'erreur va dans le terminal puis `bash` redige la sortie standard vers le fichier `log`
 - ▶ Cas équivalent à commande `>log` car la sortie d'erreur est par défaut dirigé vers la sortie standard
 - ▶ L'ordre des redirections est donc important !

Duplication d'un descripteur de sortie (suite)

- ▶ Que se passe-t-il si on écrit : commande `2>log 1>&2` ?
 - ▶ `bash` redirige `stderr` dans le fichier `log`, puis `stdout` est dupliqué dans `stderr` donc `stdout` est redirigé dans `log`
 - ▶ Cas équivalent à commande `>log 2>&1`
- ▶ Que se passe-t-il si on écrit : commande `2>&1 >log` ?
 - ▶ `bash` duplique `stderr` dans `stdout` qui est connecté au terminal (par défaut), donc la sortie d'erreur va dans le terminal puis `bash` redige la sortie standard vers le fichier `log`
 - ▶ Cas équivalent à commande `>log` car la sortie d'erreur est par défaut dirigé vers la sortie standard
 - ▶ L'ordre des redirections est donc important !

Duplication d'un descripteur de sortie (suite)

- ▶ Que se passe-t-il si on écrit : commande `2>log 1>&2` ?
 - ▶ `bash` redirige `stderr` dans le fichier `log`, puis `stdout` est dupliqué dans `stderr` donc `stdout` est redirigé dans `log`
 - ▶ Cas équivalent à commande `>log 2>&1`
- ▶ Que se passe-t-il si on écrit : commande `2>&1 >log` ?
 - ▶ `bash` duplique `stderr` dans `stdout` qui est connecté au terminal (par défaut), donc la sortie d'erreur va dans le terminal puis `bash` redige la sortie standard vers le fichier `log`
 - ▶ Cas équivalent à commande `>log` car la sortie d'erreur est par défaut dirigé vers la sortie standard
 - ▶ L'ordre des redirections est donc important !

Duplication d'un descripteur de sortie (suite)

► Démonstration avec :

```
1 #include <stdio.h>
2 int main() {
3     fprintf(stdout, "sortie_standard\n");
4     fprintf(stderr, "sortie_d'erreur\n");
5     return 0;
6 }
```

```
% ./redir > t
sortie d'erreur
% cat t
sortie standard
% ./redir 2> t
sortie standard
% cat t
sortie d'erreur
% ./redir >t 2>&1
% cat t
sortie standard
sortie d'erreur
```

Duplication d'un descripteur de sortie (suite)

- Exercice : comment forcer `echo` à écrire dans la sortie d'erreur ?

Duplication d'un descripteur de sortie (suite)

- ▶ Exercice : comment forcer `echo` à écrire dans la sortie d'erreur ?
- ▶ Réponse : `echo "stderr" >&2`

Autres redirections de l'entrée standard

► La directive "Here Document"

```
|| commande <<EOF  
|| texte  
|| texte  
|| EOF
```

`bash` lit la portion de texte comprise entre les balises `EOF` (au choix de l'utilisateur) en conservant les espaces, les sauts de lignes, en évaluant les variables et transmet le tout à l'entrée standard de la commande.

► Transmission d'une donnée à une commande :

```
|| commande <<< "chaine_de_caracteres"
```

alternative efficace à

```
|| echo "chaine_de_caracteres" | commande
```

Commentaires

- ▶ Les redirections sont lues par `bash` puis retirés de la ligne de commandes : leur positions par rapport aux arguments de la commande n'ont pas d'importance
- ▶ Exemple : `cat -n /etc/passwd >toto -l` ou `cat >toto -n /etc/passwd` sont équivalents
- ▶ On peut indifféremment rediriger l'entrée après les sorties ou le contraire
- ▶ Exemple : `wc </etc/passwd >toto` ou `wc >toto </etc/passwd`
- ▶ En revanche l'ordre des redirections de sortie est important
- ▶ Avec des redirections, ce n'est pas la commande qui ouvre les fichiers mais `bash` ! La commande ne connaît pas le nom des fichiers :

```
% wc /etc/passwd
39    65 2214 /etc/passwd
% wc </etc/passwd
39    65 2214
```

- ▶ Maladresse courante : écrire `cat toto | commande` au lieu de `commande <toto` peu efficace car utilise deux processus au lieu d'un.

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

Lancement des commandes

Redirections des flux

Composition des commandes

Interactions avec l'utilisateur

L'écriture de scripts

Scripts avancés

Pour conclure

Tube ou *pipe* ou cascade (|)

- ▶ Permet de connecter la sortie standard d'une commande à l'entrée standard d'une autre commande
- ▶ Syntaxe : `commande1 | commande2`
on a déjà vu deux exemples précédemment !
- ▶ Les données transitent en mémoire, très efficace surtout sur multi-coeurs !
- ▶ **Code de retour** : `commande2` est exécutée quelque soit le code retour de `commande1`, et le code retour final est celui de `commande2`
- ▶ **Sous-shell** : chaque commande d'un tube est **exécutée** dans une instance indépendante de `bash` (détail en programmation système)
- ▶ Variante pour la sortie d'erreur (`bash4`) : `commande1 |& commande2`

Tube : illustration du code retour

```
# pas de commande toto en cours d'exécution
% ps -C toto
PID TTY                TIME CMD
# rien trouvé donc code de retour non nul
% echo $?
1
% ps -C toto | grep TTY
PID TTY                TIME CMD
% echo $?
0
% ps -C emacs
PID TTY                TIME CMD
25813 pts/2            00:02:02 emacs
# pas de toto dans la réponse de la commande ps -C emacs
% ps -C emacs | grep toto
% echo $?
1
```


Tube : exemples

► Affinage d'une recherche

```
# Les exécutables emacs disponibles
locate -r '/emacs' | grep '/bin/'
```

ou

```
# Liste des noms (colonne 3) des étudiants du groupe 1
# dans le fichier inscrits.csv
# Les colonnes sont séparées par des ;
grep "grp1" inscrits.csv | cut -d';' -f3
```

ou

► On peut composer autant de commandes que nécessaire :

```
# Nombre d'étudiants inscrits en 2I012 et 2I001,  $|A \cap B|$ 
cat 2I012.csv 2I001.csv | sort | uniq -d | wc -l
# Étudiants inscrits en 2I012 mais pas en 2I001,  $A - B = A - A \cap B$ 
cat 2I012.csv 2I001.csv | sort | uniq -d | \
cat - 2I012.csv | sort | uniq -u
```

Séquences

► Séquence inconditionnelle :

```
|| commande1; commande2
```

écriture en une ligne de :

```
|| commande1  
|| commande2
```

► Séquence conditionnelle ET :

```
|| commande1 && commande2
```

- lance `commande1`, attend qu'elle termine, et
- si `commande1` réussit (code retour nul), exécute `commande2`

► Séquence conditionnelle OU :

```
|| commande1 || commande2
```

- lance `commande1`, attend qu'elle termine, et
- si `commande1` échoue (code retour non nul), exécute `commande2`

► `$?` contient **toujours** le code retour de la **dernière** commande exécutée

Composition de commandes : exemples de séquences

- ▶ `echo` renvoie toujours un code de retour nul,
- ▶ `ls toto` renvoie un code de retour non nul si `toto` n'existe pas

```
% ls toto; echo "on_s'en_fout"
/bin/ls: ne peut accéder toto: Aucun fichier ou dossier de ce type
on s'en fout
% echo $?
0
% ls toto && echo OK
/bin/ls: ne peut accéder toto: Aucun fichier ou dossier de ce type
% echo $?
2
% ls toto || echo KO
/bin/ls: ne peut accéder toto: Aucun fichier ou dossier de ce type
KO
% echo $?
0
```

Composition de commandes : mise en arrière-plan

- **Syntaxe** : `commande &`
`bash` lance en arrière-plan (en parallèle, *background*) la commande qui rend la main **immédiatement**
- Flux de la commande en arrière plan :
 - le flux d'entrée est déconnecté du terminal
 - les flux de sortie restent connectés au terminal
- La commande lit effectivement dans son entrée standard, alors `bash` **suspend** la commande :

```
% cat &  
[1] 78129  
[1]+ Stoppé  
% xeyes &  
[2] 78130
```

- Voir `fg`, `bg`, `C-z`, *job manager* en TD (et `man bash`)

Composition de commandes : exemple

```
% locate nvidia | grep kernel | \  
    grep -o -E '[0-9]+((\.|-)[0-9]+)+' | sort -u  
195.36.24  
2.6.32-21  
2.6.32-22  
2.6.32-23  
2.6.32-24  
% date  
mercredi 1 septembre 2010, 08:11:05 (UTC+0200)  
% ( sleep 30 ; echo "L'eau_bout" ) &  
[2] 19124  
% L'eau bout  
  
[2]+ Done ( sleep 30; echo "L'eau_bout" )  
% date  
mercredi 1 septembre 2010, 08:11:38 (UTC+0200)
```

Substitution de commandes

- Lancer `commande1` et récupérer sa sortie comme arguments pour `commande2` :
`commande2 $(commande1)` ou encore `commande2 `commande1``
- La première syntaxe autorise des appels imbriqués `$(commande1 $(commande2 ...))`
- Exemple (presque) idiot :

```
% seq 4
1
2
3
4
% echo $(seq 4)
1 2 3 4
```

- Exemples classiques :

```
# Liste des fichiers d'extension .tex qui contiennent
# le mot commandes dans le répertoire courant
% grep -l commandes `find . -name \*.tex`
# compresser l'archive tar non compressée la plus ancienne
% gzip `ls -rt *.tar | head -n 1`
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

Lancement des commandes

Redirections des flux

Composition des commandes

Interactions avec l'utilisateur

L'écriture de scripts

Scripts avancés

Pour conclure

bash en mode interactif

- ▶ `bash` en mode interactif possède des fonctionnalités étendues :
 - ▶ modes d'édition
 - ▶ historique
 - ▶ complétion des commandes, des arguments, voire plus encore
 - ▶ configuration du shell
- ▶ Ces fonctionnalités permettent d'utiliser le shell plus efficacement, plus rapidement

Mode d'édition

- ▶ Par défaut, `bash` comprend les commandes d'édition d'`emacs`
- ▶ par exemple : `C-a`, `C-e`, `C-n`, `C-p`, `M-n`, `M-p`, `C-k`, `C-y`, ...
- ▶ Les utilisateur de `vi` ne sont pas en reste :
 - ▶ `set -o vi` : passe en mode d'édition `vi`
 - ▶ `set -o emacs` : passe en mode d'édition `emacs`

Historique (1)



- Chaque instruction passée au shell est conservée dans un historique :

```
% bash
% echo $HISTFILE
/home/bereziat/.bash_history
% exit
% tail -2 $HISTFILE
echo $HISTFILE
exit
```

Le fichier `.bash_history` (valeur par défaut de `$HISTFILE`) est écrit par `bash` lorsqu'il se termine

- On peut contrôler la taille (nombre de lignes) de l'historique avec la variable `HISTFILESIZE`
- On peut choisir ce qui est conservé ou ignoré dans l'historique à l'aide des variables `HISTCONTROL` et `HISTIGNORE` (voir `man bash`)

Historique (2)

- ▶ La commande `history n` affiche les `n` dernières lignes de l'historique, `history` affiche tout l'historique
- ▶ Rappeler une ligne de l'historique :
 - ▶ `C-p`, `C-n` : permet de se déplacer dans l'historique (*previous*, *next*) en mode d'édition `emacs` (mode par défaut), ces commandes sont normalement reliées aux touches  et  du clavier
 - ▶ `C-r` : recherche dans l'historique par une expression
 - ▶ `!!` : rappel de la dernière ligne d'historique
 - ▶ `!chaine` : rappel de la dernière commande commençant par `chaine`
 - ▶ `!n` : rappel de la `n`-ième ligne d'historique
 - ▶ `!-n` : rappel de la `n`-ième ligne d'historique en partant de la fin
 - ▶ `!*` : les arguments de la dernière commande
 - ▶ `!!:s/s1/s2` : rappel de la dernière commande en remplaçant `s1` par `s2` fonctionne avec tout ce qui commence par `!`

Complétion

- ▶ `bash` utilise la complétion pour accélérer la saisie de commandes :
 - ▶ l'appui sur la touche `TAB` après la saisie partielle d'une commande la complète en fonction des commandes trouvées dans `PATH`
 - ▶ si plusieurs noms sont possible, `bash` affiche la liste
- ▶ `bash` complète également les noms des fichiers passés en argument
- ▶ si le paquet `bash-extension` est installé (le cas sur Debian) `bash` peut compléter :
 - ▶ les noms des options des commandes
 - ▶ les cibles des `Makefile` (voir cours `Makefile`)
 - ▶ les noms des fichiers derrière une connexion `ssh` !

Configuration de bash

- ▶ On peut personnaliser `bash`, il faut distinguer :
 - ▶ les fichiers lus par le shell de connexion
 - ▶ les fichiers lus par les shells lancés après le shell de connexion
 - ▶ les autres shells héritent de l'environnement du shell de connexion (ils n'ont donc pas besoin de lire ces fichiers de configuration)
- ▶ Les noms et emplacement varient en fonction des systèmes et des distributions
- ▶ Sur Debian :
 - ▶ Le shell de connexion lit `/etc/profile` puis `~/.bash_login` (s'il s'agit de `bash`) puis `~/.profile` (s'ils existent)
 - ▶ Les processus `bash` lancés après connexion lisent `~/.bashrc` s'il existe
 - ▶ Les processus `bash` qui se terminent lisent `~/.bash_logout` s'il existe
- ▶ Que mettre dans son `~/.bashrc` ?
 - ▶ Un `PATH` spécifique (et tout autre variable d'environnement, `EDITOR`, `HISTFILESIZE`, ...)
 - ▶ Un joli `PROMPT` (<http://bashrcgenerator.com/>)
 - ▶ Des alias, par exemple : `alias ls='ls -F -color=auto'`
 - ▶ Des fonctions utiles glanées sur le Web

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

- Le fichier script, arguments positionnels et code retour

- Variables (ou paramètres)

- Variables d'environnement

- Lecture dans l'entrée standard

- Chaînes et expansion

- Structures de contrôle

Scripts avancés

Pour conclure

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

- Le fichier script, arguments positionnels et code retour

- Variables (ou paramètres)

- Variables d'environnement

- Lecture dans l'entrée standard

- Chaînes et expansion

- Structures de contrôle

Scripts avancés

Pour conclure

Motivation et définition

- ▶ Longue série de commandes avec un paramétrage particulier ? Comment s'en souvenir pour la réutiliser ?
- ▶ Il faut en faire un script !
- ▶ Un script est un fichier texte de la forme :

```
1  #! /bin/bash
2
3  commandes ...
```

- ▶ La première ligne s'appelle le *shebang* (*# = sharp*, *! = bang*) et permet au système d'identifier l'interpréteur du script (même chose avec Python par exemple). Tout se passe comme si on écrivait

```
|| % /bin/bash script
```

- ▶ Un script doit être exécutable (`chmod +x monscript`), **emacs** le fait pour vous !
- ▶ Un script est une **commande** (revoir les interactions d'un processus)

Premier script

- Soit la commande :

```
# Étudiants inscrits en 2I012 mais pas en 2I001,  $A - B = A - A \cap B$   
cat 2I012.csv 2I001.csv | sort | uniq -d | \  
    cat - 2I012.csv | sort | uniq -u
```

- Et créons le script `minus` :

```
1 #! /bin/bash  
2 # Description: minus A B  
3 # Les lignes de A qui ne sont pas dans B  
4 cat $1 $2 | sort | uniq -d | cat - $1 | sort | uniq -u
```

- Et la mise en œuvre :

```
% chmod -x minus  
% ./minus 2I012.csv 2I001.csv
```

- `$1` est le premier argument passé au script, `$2` le second, ...

Les arguments positionnels

- ▶ Rappel : `bash` découpe les lignes en mots, le premier mot est la commande et les suivants sont les arguments de la commande qui se retrouvent dans les arguments positionnels si la commande est un script.
- ▶ `$0` est le nom du script, `$1, ..., $9, ${10}, ...` les arguments passés à la commande
- ▶ piège : `$10` signifie `${1}0`
- ▶ `$0` permet de connaître son nom pour des appels récursifs
- ▶ `$#` nombre d'arguments
- ▶ `$@` (ou `$*`) est expansé en `$1 $2 $3 ...`
- ▶ `"$@"` est expansé en `"$1" "$2" "$3" ...` (permet de respecter les espaces dans les arguments, abordé plus loin)

Manipulation des arguments positionnels (`shift`)

- ▶ `set` : empiler de nouvelles valeurs dans les arguments positionnels
- ▶ `shift` : dépiler des arguments
- ▶ permet un changement dynamique des arguments

```
% echo $#: "$@"  
0:  
% set a b c d e  
% echo $#: "$@"  
5: a b c d e  
% shift  
% echo $#: "$@"  
4: b c d e  
% shift 3  
% echo $#: "$@"  
1: e  
% set `date` fin  
% echo $#: "$@"  
7: mercredi 9 novembre 2016, 21:21:40 (UTC+0100) fin
```

Manipulation des arguments positionnels (set)

- Les arguments passés à `set` sont positionnés dans les variables `$1, $2, ...`
- Exemple pour accéder à un mot d'une phrase :

```
% set cette phrase comprend cinq mots
% echo $# mots: le troisieme est \"$3\"
5 mots: le troisieme est "comprend"
% PHRASE="Ce_qui_se_concoit_bien_s'enonce_clairement."
% set $PHRASE
% echo "$#_mots:_le_septieme_est_\"$7\",_execution_de_\"$0\""
7 mots: le septieme est "clairement.", execution de "-bash"
```

set pour découper

- Pour découper en mots une ligne, `bash` utilise comme délimiteur un des caractères donnés dans la variable `IFS` (*Internal Field Separator*)
Par défaut, `IFS` liste les caractères blancs (espace, tabulation, retour à la ligne)
- On peut modifier `IFS` pour changer la manière de découper les mots :

```
% cd; pwd
/home/bereziat
% set `pwd`; echo '$#="'$#'" $1="'$1'" $3="'$3'"
$#="1" $1="/home/bereziat" $3=""
% bak=$IFS IFS="/"
% set `pwd`; echo '$#="'$#'" $1="'$1'" $3="'$3'"
$#="3" $1="" $3="bereziat"
% IFS=$bak
```

- En pratique, on préfère utiliser la commande `cut` :

```
% pwd | cut -d'/' -f3
bereziat
% a=`pwd`
% cut -d'/' -f2 <<< $a
home
```

Autres comportements de set

- **set sans argument** liste l'environnement (et davantage) de `bash` :

```
% set | head -2
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_alias
force_ignore:histappend:interactive_comments:progcomp:prompt
% VIDE=""
% set $VIDE | head -2
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_alias
force_ignore:histappend:interactive_comments:progcomp:prompt
% set | wc -l
2948
```

- **set avec des options** paramètre le comportement de `bash` (voir `man bash`, voir scripts avancés)

Code retour d'un script

- ▶ Comme toute commande un script doit retourner un code d'erreur
- ▶ Par défaut c'est 0 (pas d'erreur)
- ▶ Sinon, c'est le code de la dernière commande du script
- ▶ On peut utiliser le mot-clé **exit** pour retourner une valeur explicite
- ▶ Exemple :

```
% echo 'false' > t
% bash t; echo $?
1
% echo 'exit 0' >> t
% bash t; echo $?
0
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Le fichier script, arguments positionnels et code retour

Variables (ou paramètres)

Variables d'environnement

Lecture dans l'entrée standard

Chaînes et expansion

Structures de contrôle

Scripts avancés

Pour conclure

Définition d'une variable

- **Syntaxe** : `nom=valeur`
- **Attention** : **jamais d'espaces** autour du caractère d'affectation
- **nom** :
 - premier caractère est une lettre majuscule ou minuscule non accentué
 - caractères suivant : comme le premier augmenté des chiffres et de `_`
- **valeur** :
 - suite quelconque de caractères sans espace ni tabulation, ni `"` ou `'`
attention : certains caractères sont *expansés* (on le verra) par `bash`
 - pour insérer des espaces, il faut protéger la valeur avec des quotes ou des guillemets (double-quotes), ou par `\`
- Les valeurs n'ont pas de types : ce sont toujours des chaînes de caractères
- Exemples

```
% i=100
% file=/etc/password
% ll="bash_c'est_ouf"
% a=3 b=5
# défini 2 variables!
```

```
% echap='\'; echo $echap
'
% err = toto
bash: err : commande introuvable
% err =toto
bash: err : commande introuvable
```

Expansion des variables

- ▶ Terme `bash` pour dire évaluation ou dérérérencement, et déclenché par le caractère spécial `$`
- ▶ Syntaxe : `$nom` ou `${nom}` pour délimiter le nom dans les chaînes de caractères
- ▶ Si la variable n'existe pas, `bash` considère que sa valeur est une chaîne vide
- ▶ `unset` supprime la variable de la mémoire
- ▶ Exemples

```
% a=3 b=${a}cm
% echo "a=$a_b=$b"
a=3 b=3cm
% unset a
% echo a=$a
a=
```

- ▶ Les arguments positionnels `$1`, `$2`, ... sont des variables spéciales et réservées par `bash`. Idem pour `$?`, et il en existe pleins d'autres

Calcul sur les variables selon des motifs

- ▶ motifs : chaîne contenant des caractères ordinaires et des caractères spéciaux `*?[]`
- ▶ Suppression de motifs :
 - ▶ `${VAR#motif}` retire du plus petit préfixe dans `$VAR` correspondant au motif
 - ▶ `${VAR##motif}` retire du plus grand préfixe correspondant au motif
 - ▶ `${VAR%motif}` retire du plus petit suffixe correspondant au motif
 - ▶ `${VAR%%motif}` retire du plus grand suffixe correspondant au motif

▶ Exemples :

- ▶ la commande `basename` peut être remplacée par `${VAR##*/}`

```
% dir=/Users/bereziat/Enseignement/2I012/Cours/Regexp
% echo ${dir##*/}
Regexp
% echo ${dir##*/}
Users/bereziat/Enseignement/2I012/Cours/Regexp
```

- ▶ la commande `dirname` peut être remplacée par `${VAR%/*}`

```
% echo ${dir%/*}
/Users/bereziat/Enseignement/2I012/Cours
```

- ▶ supprimer l'extension d'un nom de fichier : `${VAR%.*}`

Calcul sur les variables selon des motifs (suite)

► Substitution de motifs :

- `${VAR/motif/remplacement}` remplace la première occurrence de motif **par** remplacement
- `${VAR//motif/remplacement}` remplace toutes les occurrences de motif
- comportement spécifique :
 - si motif commence par # alors il doit correspondre au début de \$VAR
 - si motif commence % alors il doit correspondre à la fin de \$VAR

► Transformation de la casse (bash4) :

- `${VAR^motif}` passe en majuscule le premier caractère correspondant à motif
- `${VAR^^motif}` passe en majuscule de toutes les caractères correspondants à motif
- `${VAR,motif}` passe en minuscule le premier caractère correspondant à motif
- `${VAR,,motif}` passe en minuscule de toutes les caractères correspondants à motif
- ici * et ? jouent le même rôle

Calcul sur les variables, des exemples

```
% i='La vie est belle, je dis La vie est belle'
% echo ${i:7:9} ${i:18}
est belle je dis La vie est belle
% echo ${i/est/serait}
La vie serait belle, je dis La vie est belle
% echo ${i//est/serait}
La vie serait belle, je dis La vie serait belle
% echo ${i/#est/serait}
La vie est belle, je dis La vie est belle
% echo ${i/%belle/moche}
La vie est belle, je dis La vie est moche
% echo ${i,?}
la vie est belle, je dis La vie est belle
% echo ${i^?}
LA VIE EST BELLE, JE DIS LA VIE EST BELLE
% echo ${i,,L}
la vie est belle, je dis la vie est belle
% set abba abracadabra; echo ${@/a/o}
obba obracadabra
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

- Le fichier script, arguments positionnels et code retour

- Variables (ou paramètres)

- Variables d'environnement**

- Lecture dans l'entrée standard

- Chaînes et expansion

- Structures de contrôle

Scripts avancés

Pour conclure

Définition

- ▶ Une variable d'environnement est une variable transmise aux processus fils via un tampon appelé environnement : **c'est une copie et pas un partage**
- ▶ En bash, on utilise `export` pour créer ces variables :

```
export nom=valeur  
# ou bien  
nom=valeur  
export nom
```

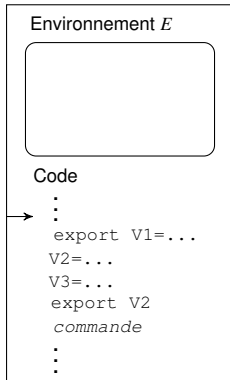
- ▶ **Affichage** : `printenv nom` ou `echo $nom`
`printenv` ne fonctionne pas avec des variables ordinaires
- ▶ `printenv` sans argument les affiche toute !
- ▶ Exemple de variables d'environnement usuelles :

```
% echo $HOME $USER $SHELL  
/home/bereziat bereziat /bin/bash  
% printenv PATH  
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```


Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement

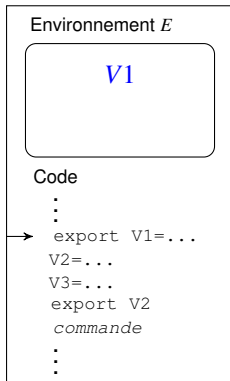
Processus père



Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement

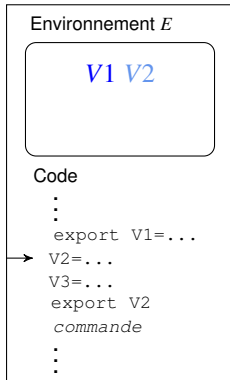
Processus père



Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement

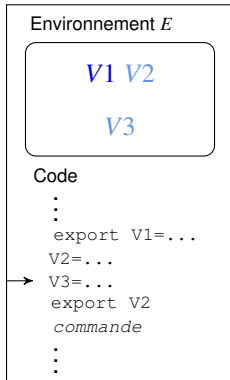
Processus père



Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement

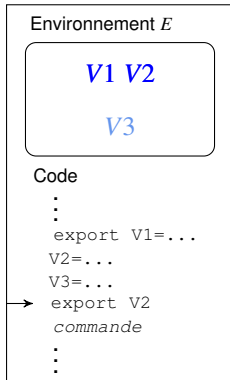
Processus père



Transmission

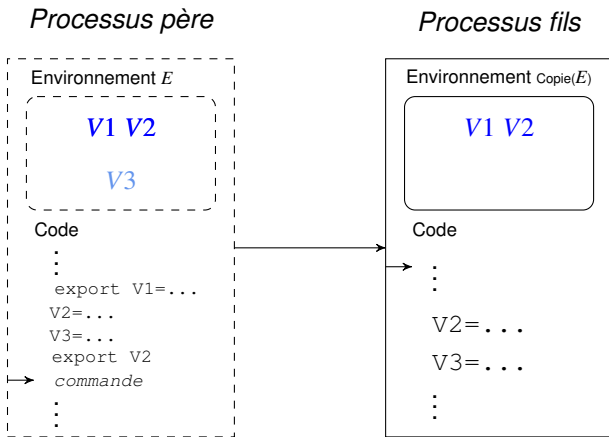
Exécution de commande dans un sous-shell avec copie de l'environnement

Processus père



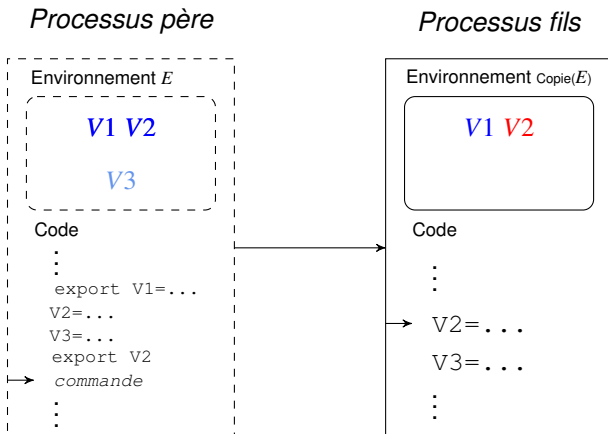
Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement



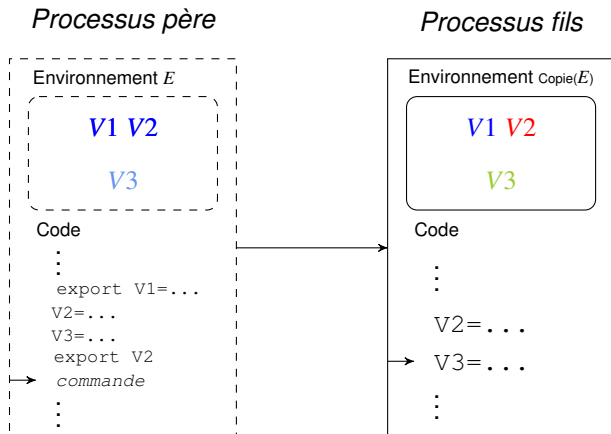
Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement



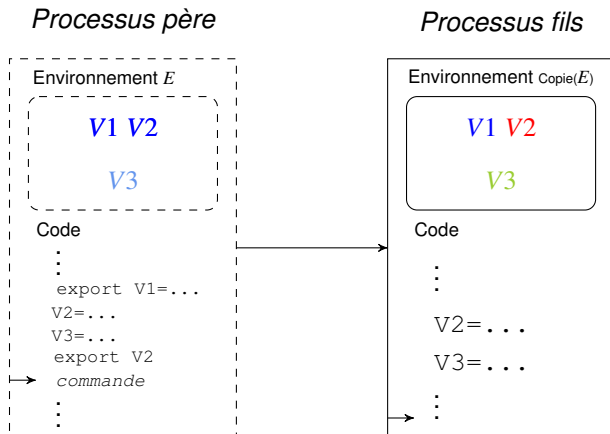
Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement



Transmission

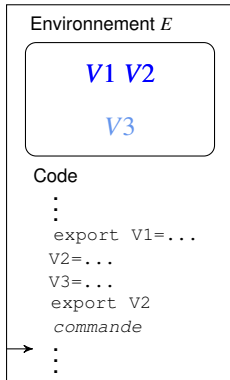
Exécution de commande dans un sous-shell avec copie de l'environnement



Transmission

Exécution de commande dans un sous-shell avec copie de l'environnement

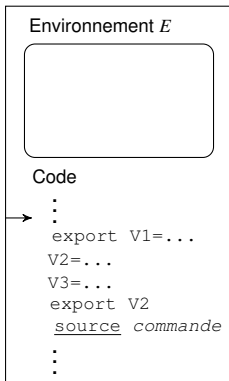
Processus père



Transmission (source ou .)

Exécution de commande avec partage de l'environnement

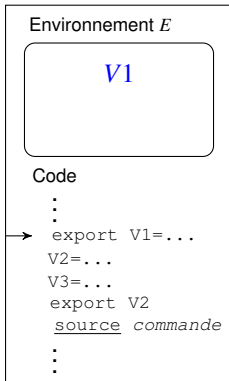
Processus



Transmission (source ou .)

Exécution de commande avec partage de l'environnement

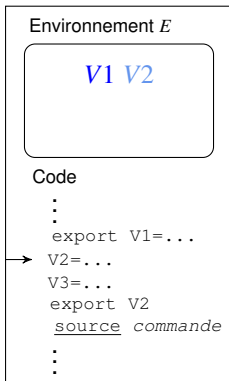
Processus



Transmission (source ou .)

Exécution de commande avec partage de l'environnement

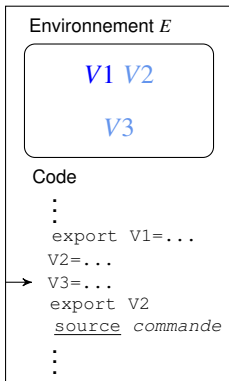
Processus



Transmission (source ou .)

Exécution de commande avec partage de l'environnement

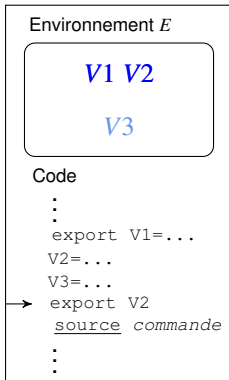
Processus



Transmission (source ou .)

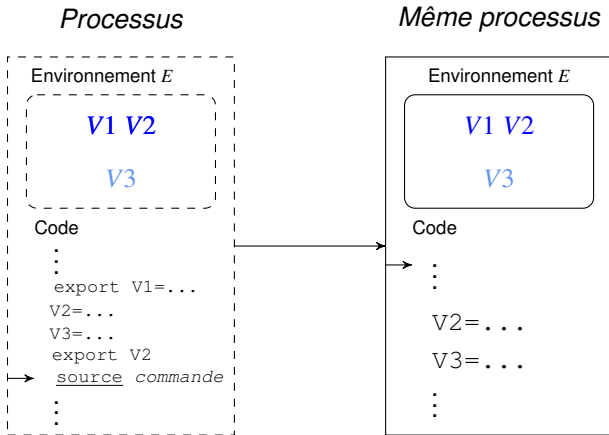
Exécution de commande avec partage de l'environnement

Processus



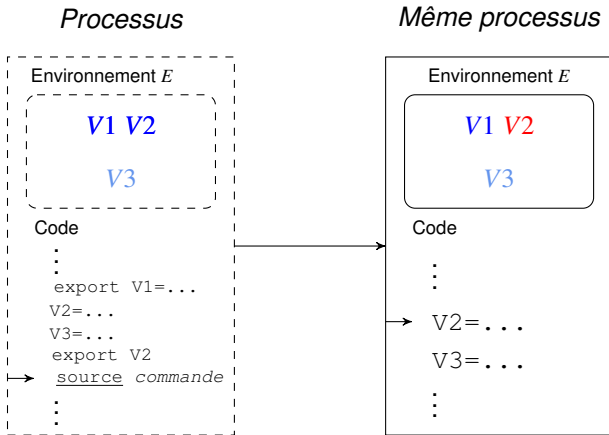
Transmission (source ou .)

Exécution de commande avec partage de l'environnement



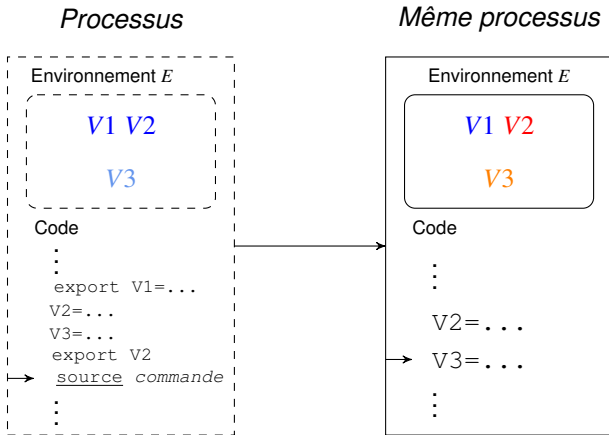
Transmission (source ou .)

Exécution de commande avec partage de l'environnement



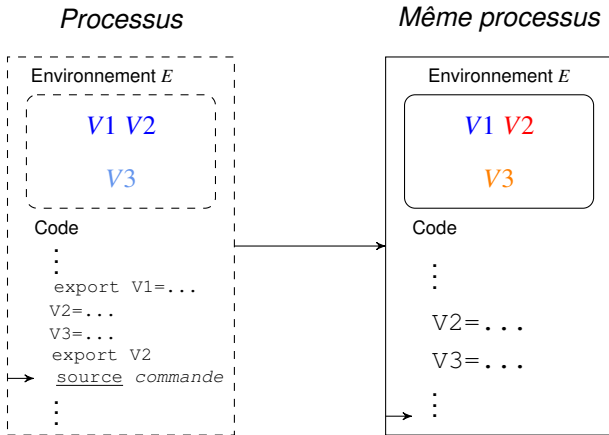
Transmission (source ou .)

Exécution de commande avec partage de l'environnement



Transmission (source ou .)

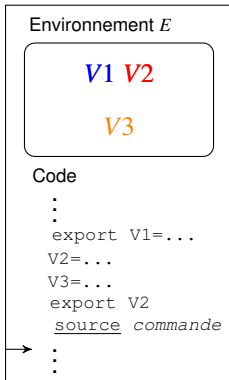
Exécution de commande avec partage de l'environnement



Transmission (source ou .)

Exécution de commande avec partage de l'environnement

Processus



Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

- Le fichier script, arguments positionnels et code retour

- Variables (ou paramètres)

- Variables d'environnement

- Lecture dans l'entrée standard**

- Chaînes et expansion

- Structures de contrôle

Scripts avancés

Pour conclure

D'où proviennent les données à traiter ?

1. de la liste des arguments. Par exemple `echo` envoie dans la sortie standard les paramètres donnés sur sa ligne d'arguments
2. de variables d'environnement. Par exemple `which` utilise la variable `PATH` pour localiser une commande
3. **de l'entrée standard bien-sûr !** Le cours de `bash` tourne autour de ça, car on aime faire des tubes (efficacité)
4. d'appels systèmes comme l'ouverture explicite et la lecture d'un fichier. Mais on peut se ramener au cas 3. grâce à la commande `cat` :

```
1  #!/bin/bash
2
3  # ouverture et traitement du fichier passé en argument
4  cat $1 | ...
```

La primitive read

- **Syntaxe** : `read VAR1 VAR2 ... VARn`
- La primitive `read` lit une ligne dans l'entrée standard, la découpe en mots selon `IFS`, place le premier mot dans `VAR1`, le second dans `VAR2`, et ainsi de suite ...
 - s'il y a plus de mots que de variables, les derniers mots sont placés dans la dernière variable
 - s'il y a moins de mots que de variables, les dernières variables ne sont pas initialisées
- `read` retourne code d'erreur non nul si il n'y a plus rien à lire dans l'entrée standard
- On peut rediriger l'entrée standard de `read`, par défaut l'entrée standard de `read` est celle du script ou du terminal (si interactif)

```
% echo 'deux mots' >toto
% read a b c <toto
% echo a=\"$a\" b=\"$b\" c=\"$c\"
a='deux' b='mots' c=''
```

read : options utiles

- Attention : les caractères d'échappements (voir plus loin) sont interprétés par `read`, par exemple un `\` placé en fin de ligne signifie qu'il faut ignorer le saut à la ligne et continuer sur la ligne suivante :

```
% cat >toto
cette premiere ligne continue \
sur la suivante
ceci est la suivante
% read a <toto
% echo $a
cette premiere ligne continue sur la suivante
```

- `read -r` permet de ne pas interpréter les échappements
- `read -a` stocke les mots dans un tableau (traité plus loin)
- `read -d` permet de choisir le délimiteur de **lignes** (pas de mots !)
- `read -n nb` limite la lecture à `nb` caractères
- `read -p invite` affiche `invite` avant la lecture
- `read -s` n'affiche pas ce qui est lu (secret)

read : des exemples

```
% read -n 5 a <<< 1234567890
% echo $a
12345
% echo 1234567890 | read a
% echo \'$a\'
'' # pourquoi ??
% read -p 'login? ' login
login? bereziat
% echo $login
bereziat
% read -p 'passwd? ' passwd
passwd?
% echo $passwd
dominique
% tail -1 /etc/passwd
bereziat:x:1000:1000:Dominique Bereziat,,,:/home/bereziat:/bin/bash
% tail -1 /etc/passwd >t
% IFS=':'
% read login passwd uid gid fullname home shell <t
% echo $home
/home/bereziat
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

- Le fichier script, arguments positionnels et code retour

- Variables (ou paramètres)

- Variables d'environnement

- Lecture dans l'entrée standard

- Chaînes et expansion**

- Structures de contrôle

Scripts avancés

Pour conclure

Le quoting

- ▶ Rappel : une ligne lue par `bash` est découpée en mots, puis chaque mots subit une série d'expansions.
- ▶ Nous connaissons plusieurs types d'expansion :
 - ▶ celle des variables
 - ▶ celle des motifs du shell
 - ▶ celle de substitution de commandes (``` ou `$()`)
- ▶ L'utilisation de *quote* permet de définir des chaînes contenant des espaces et qui ne seront pas découpées en mots
- ▶ Deux type de *quote* :
 - ▶ Les *simple quote*, elle protège la chaîne des expansions du shell : `' ... '`
 - ▶ Les *double quote*, elle autorise les expansions : `" ... "`
- ▶ Après découpage, les quotes **délimiteurs** sont enlevés

Protection des chaînes de caractères

► Considérons script1a :

```
1  #! /bin/bash
2
3  echo "$#_arguments"
4  echo 'args $0: "'$0'", $1: "'$1'", ${#1}: '${#1}\'
5      ', $2: "'$2'", ${12}: "'$2'""
6  echo '$@="'$@'""'
```

► Exemple avec appel explicite de bash :

```
% bash script1a a    b c
3 arguments
args $0: "script1a", $1: "a", ${#1}: 1, $2: "b", ${12}: ""
$@="a_b_c"
```

► Exemple avec l'exécutable :

```
% chmod +x script1a
% ./script1a a b c d e    f    g h i j k l m n o p
16 arguments
args $0: "./script1a", $1: "a", ${#1}: 1, $2: "b", ${12}: "l"
$@="a_b_c_d_e_f_g_h_i_j_k_l_m_n_o_p"
```

Protection des chaînes de caractères (2)

- Considérons `script1b` :

```
1  #! /bin/bash
2
3  echo "$#_arguments"
4  echo 'args $0: "$$0"', $1: "'$1'", ${#1}: '${#1}\
5      ', $2: "'$2'", ${12}: "'${12}''
6  echo '$@="'$@''
```

- Exemple avec des espaces dans \$1

```
% ./script1a "a__b" c
2 arguments
args $0: "./script1a", $1: "a_b", ${#1}: 5, $2: "c", ${12}: "
$@="a_b_c"
% ./script1b "a__b" c
2 arguments
args $0: "./script1b", $1: "a__b", $2: "c", ${12}: ""
$@="a__b_c"
```

- Les guillemets autour de \$1 (et \$@) permettent de protéger les espaces passés dans le premier argument, (les arguments).

Protection par échappement

- Autre façon de protéger les espaces dans les chaînes de caractère : les échapper avec \

```
% mkdir 'dossier avec espaces'
% ls dossier avec espaces
ls: impossible d'accéder à dossier: Aucun fichier ou dossier de ce type
ls: impossible d'accéder à avec: Aucun fichier ou dossier de ce type
ls: impossible d'accéder à espaces: Aucun fichier ou dossier de ce type
% ls dossier\ avec\ espaces -d
dossier avec espaces
```

- Le caractère d'échappement permet d'empêcher les expansions du shell :

```
% echo \'$a\'
'$a'
% printf \\ttab
\ttabulation,\t est une tabulation.
```

- pour le second exemple, \ échappe \, et printf peut lire \ttabulation, \t est une tabulation.

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

- Le fichier script, arguments positionnels et code retour

- Variables (ou paramètres)

- Variables d'environnement

- Lecture dans l'entrée standard

- Chaînes et expansion

- Structures de contrôle**

Scripts avancés

Pour conclure

Répétitions

► Boucle `while` :

```
while commande  
do instructions  
done
```

- les instructions sont répétées tant que `commande` retourne 0

► Boucle `until` :

```
until commande  
do instructions  
done
```

- les instructions sont répétées tant que `commande` retourne une valeur non nulle

► Remarques :

- toute commande retourne un code erreur. Un code C sans `return` ou `exit` retournera une valeur aléatoire !
- `instructions` peut contenir plusieurs instructions séparées par des `;` ou des sauts à la ligne

Exemple avec while

- Un usage classique (dans ce cours) avec `while` et `read` :

```
1  #!/bin/bash
2  co=0
3  while read line; do
4      co=$((expr $co + 1))
5  done < $1
6  echo Le fichier $1 contient $co lignes
```

- `read` retourne un code non nul lorsque la fin du fichier est atteint, ce qui permet de sortir de la boucle
- Cette version est particulièrement inefficace sur des long fichiers (ici le code source de ce cours)

```
% time ./nlines shell.org
Le fichier shell.org contient 2507 lignes

real          0m6.477s
user          0m2.106s
sys           0m2.785
```

- On verra comment faire mieux

Exemple avec while

- Erreur classique (constatée dans ce cours) :

```
1 #!/bin/bash
2 co=0
3 cat $1 | while read line; do
4     co=$(expr $co + 1)
5 done
6 echo Le fichier $1 contient $co lignes
```

```
% ./nlines shell.org
Le fichier shell.org contient 0 lignes
```

- N'oublions jamais que les commandes d'un tube sont exécutées dans des sous-shells !
- Boucle infinie, on en sort avec `break` ou `exit` :

```
while true; do
    ...
done
```

- Utilisation de `;` pour avoir une instruction plus compacte

Alternative

- Alternative `if then elif else`, (les parties entre crochets sont optionnelles) :

```
if commande
then instructions
[elif commande2
  then instructions]
[else instructions]
fi
```

- L'alternative est vraie si la commande ne retourne pas d'erreur (code retour nul)
- Tester si une commande échoue : préfixe !

```
if ! commande
then echo "command_retourne_valeur_non_nulle"
[elif ... ]
```

Alternative

Exemples

```
# Compilation
if gcc toto.c; then echo "Compilation_réussie"
else echo "Compilation_échoue"; fi

# Commenter une large partie d'un script
if false; then
...
fi

# Tester la présence d'une commande (utile pour le script)
if ! which convert; then
    echo "Erreur:_commande_`convert`_non_installée"
    exit 1
fi
```

Tests ([])

- ▶ Le mot-clé du shell `[]` permet de faire des tests (sur les variables et les fichiers)
- ▶ **À savoir** : il existe aussi des commandes `[`, `test` et des primitives `[` et `test`

```
% type -a test
test est une primitive du shell
test est /usr/bin/test
% type -a [
[ est une primitive du shell
[ est /usr/bin/[
% type -a [[
[[ est un mot-clé du shell
```

- ▶ Les commandes `test` et `[` sont identiques (voir `man test` et `man [`)
- ▶ Les primitives `test` et `[` sont identiques (voir `help test` et `help [`)
- ▶ La primitive `test` existe pour éviter le lancement coûteux de la commande `test`
- ▶ Les commandes `test` et `[` existent pour compatibilité POSIX
- ▶ `[[` est plus riche que `[`, mais n'est pas POSIX

Tests ([])

- Comparaison de chaînes de caractères, de nombres, et tests sur les fichiers

```
|| [[ condition ]]
```

- Les tests ont souvent leur place dans les boucles et les alternatives (mais pas systématiquement !)
- Tests sur les fichiers, retourne 0 si :

```
[[ -e FILE ]] # FILE existe
[[ -d FILE ]] # FILE existe et est un répertoire
[[ -f FILE ]] # FILE existe et est un fichier ordinaire
[[ -s FILE ]] # FILE existe et a une taille non nulle
[[ -N FILE ]] # FILE existe et a été modifié depuis sa
               # dernière lecture
[[ FILE1 -nt FILE2 ]] # FILE1 est plus récent que FILE2
[[ FILE1 -ot FILE2 ]] # FILE1 est plus vieux que FILE2
```

(liste non exhaustive, voir `man bash /CONDITIONS`)

Tests ([])

- Tests sur les chaînes de caractères, retourne 0 si :

```
[[ -n S ]]      # la longueur de S est non nulle
[[ -z S ]]      # la longueur de S est nulle
[[ S1 == S2 ]]  # les deux chaînes sont égales
[[ S1 = S2 ]]   # les deux chaînes sont égales
[[ S1 != S2 ]]  # les deux chaînes sont différentes
[[ S1 < S2 ]]   # S1 avant S2 (ordre lexicographique)
[[ S1 > S2 ]]   # S1 après S2 (ordre lexicographique)
[[ S1 =~ S2 ]]  # S1 correspond au motif S2 (regex)
```

- l'opérateur =~ est présenté dans le cours sur les expressions régulières
- grep est préféré dans ce cours à =~
- L'opérande à droite de = admet des motifs du shell :

```
% a=toto.tex
% [[ $a = *.tex ]] && echo OK
OK
% ls $a
ls: cannot access 'tt': No such file or directory
% [ $a = *.tex ] || echo KO
KO
% touch toto.tex; [ $a = *.tex ] && echo OK
OK
```

Tests ([])

- Test sur des entiers (mais ce sont toujours des chaînes de caractères)

```
[[ N1 -eq N2 ]] # N1 et N2 sont égaux
[[ N1 -ne N2 ]] # N1 et N2 sont différent
[[ N1 -ge N2 ]] # N1 est supérieur ou égale à N2
[[ N1 -gt N2 ]] # N1 est supérieur strictement à N2
[[ N1 -le N2 ]] # N1 est inférieur ou égale à N2
[[ N1 -lt N2 ]] # N1 est inférieur strictement à N2
```


Tests ([])

- Composition des conditions (opérateurs booléens) :
 - ! exp : vrai si exp est fausse
 - exp1 && exp2 : vrai si exp1 et exp2 sont vraies, exp2 n'est pas évaluée si exp1 est fausse
 - exp1 || exp2 : vrai si exp1 ou exp2 sont vraies, exp2 n'est pas évaluée si exp1 est vraie
 - (exp) vrai si exp est vrai (modification de priorité)
- **Important** il faut bien respecter les espaces entre chaque arguments :

```
% a=1
% [[ $a=1 ]]
[[1=1]] commande introuvable
% [[ $a = 1 ]] && echo a vaut 1
a vaut 1
```

- [] est un mot-clé du langage de bash (et pas une primitive) car il change la grammaire des symboles (,) , && , || , < et >
- [identique à [[exception faites des opérateurs ci-dessus et =~

Exemples avec tests

```
if [[ $# -lt 1 ]]
then
    echo "Usage_..._" 1>&2
    exit 1
fi
while [[ $# -gt 0 ]]
do
    if [[ $1 = -x ]] ; then
        echo option -x
    fi
    ...
    shift
done

[[ -d "$f" ]] && echo "Répertoire"
```

- Penser à protéger les opérandes dans le test pour le cas où leur valeur serait une chaîne vide

Boucle for

► Syntaxe :

```
for VAR [in VALUES]
do instructions
done
```

- VALUES est une chaîne de caractères qui sera expansée puis découpée en mots (IFS) par le shell
- VAR prendra successivement les valeurs de la liste de mots induite par VALUES

► for sans la partie in VALUES est équivalent à

```
for VAR in "$@"
do instructions
done
```

Exemple avec for

```
1 #!/bin/bash
2 # tri_rep: affiche séparément fichiers et répertoires
3 for i; do
4     if [[ -d "$i" ]]
5     then DIRS="$DIRS_$i"
6     elif [[ -f "$i" ]]
7     then FILES="$FILES_$i"
8     fi
9 done
10 cat <<EOF
11 Répertoires: $DIRS
12 Fichiers: $FILES
13 EOF
```

```
% ./tri_rep *
Répertoires: C figs sh
Fichiers: Makefile macros2017.sty shell.org shell.pdf
% for a in *; do file $a; done
C: directory
Makefile: ASCII make commands text
figs: directory
macros2017.sty: ASCII text
sh: directory
shell.org: UTF-8 Unicode text
shell.pdf: PDF document, version 1.5
```

Choix multiples

- Syntaxe du `case` :

```
case expression in  
cas1) instructions ;;  
cas2) instructions ;;  
...  
casn) instructions ;;  
*)      instructions ;;  
esac
```

- Les cas sont des *motifs* du shell
- Plusieurs cas peuvent être regroupés à l'aide du caractère `|` :
`cas1|cas2)`
- Dès qu'un cas est trouvé, on sort de la structure (à la différence de C)
- Le code retour est celui de la dernière commande ou 0 si un cas n'est pas rencontré ou bien qu'il n'y pas d'instruction
- `expression` et les motifs peuvent contenir des variables qui seront expansés

Exemple choix multiples

► Filtrage de fichiers :

```
1  #!/bin/bash
2  # usage: ./lecteur liste
3  # Lit chacun des fichiers de la liste
4  # avec un lecteur video ou audio selon son extension
5
6  for f; do
7      case $f in
8          *.mp3|*.flac) rhythmbox $f;;
9          *.avi) totem $f;;
10         *.mkv) totem $f;;
11         *) echo "'$f':_Format_non_reconnu"; exit 1;;
12     esac
13 done
```

► Alternative au test et if

```
read -p "confirmation?_" yesno
case $yesnot in
yes|y|o|oui) ...;;
esac
```

Exemple choix multiples (suite)

► Filtrage de nombre :

```
1  #!/bin/bash
2  # ./commentaire note
3  # Commente le niveau de la note, illustration du case
4
5  case $1 in
6  [0-5]) echo "Il_faut_vraiment_se_mettre_au_travail";;
7  [6-9]|10) echo "Il_faut_travailler_davantage";;
8  1[1-5]) echo "Pas_mal";;
9  1[6-9]|20) echo "Bien";;
10 *) echo "Trop_fort";;
11 esac
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

- Commandes composées

- Fonctions

- Expressions arithmétiques

- Expansion des accolades

- Évaluation provoquée

- Ordre des expansions

- Tableaux

Pour conclure

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

- Commandes composées

- Fonctions

- Expressions arithmétiques

- Expansion des accolades

- Évaluation provoquée

- Ordre des expansions

- Tableaux

Pour conclure

Commande groupée

- Une commande groupée est vue comme une unique instruction partageant l'environnement du shell
- Syntaxe :

```
{ command; command; }  
# ou  
{  
command  
command  
}
```

Attention : espace ou saut à la ligne obligatoire devant { !

- Ainsi, il est possible de transmettre des variables :

```
% echo "premiere  
seconde" >t  
% { read a; read b; } <t  
% echo a=$a b=b$b  
a=premiere b=seconde
```

Autres commandes composées

- Variante de la précédente, exécutée dans un sous-shell :

```
|| (command)  
|| (command; command)
```

- Exemple :

```
|| % (echo Bonjour; echo monde) | wc -l  
|| 2
```

- Attention : pas de transmission de variables !

```
|| % a=2  
|| % (a=10)  
|| % echo $a  
|| 2
```

Flux et commandes composées

- Les commandes du tube sont exécutées dans un sous-shell : tout se passe comme si on écrivait : `(commande) | (commande)`
- Dans le second cas le calcul et l'affichage du résultat est dans le même sous-shell
- Ici utiliser `()` ou `{ }` est équivalent à cause du tube

```
1 #!/bin/bash
2
3 MOTS=0
4 cat "$@" | while read mots
5 do
6     set ' ' $mots
7     MOTS=$(expr $MOTS + $# - 1 )
8 done
9 echo $MOTS
```

```
1 #!/bin/bash
2
3 cat "$@" | ( MOTS=0
4     while read mots ; do
5         set ' ' $mots
6         MOTS=$(expr $MOTS + $# - 1 )
7     done
8     echo $MOTS )
```

```
% echo '1 2 3
4 5 6' | ./compte_mot1
0
```

```
% echo '1 2 3
4 5 6' | ./compte_mot2
6
```

Flux et commandes composées

- Une structure de contrôle est une instruction comme une autre, on peut donc rediriger ses entrées/sorties

```
% for i in $(seq 5); do  
echo "L$i_suite"  
done > some.txt  
% cat some.txt  
L1 suite  
L2 suite  
L3 suite  
L4 suite  
L5 suite
```

```
% if read ; then  
read a b ; echo $a  
fi <some.txt  
L2  
% if (read;read); then  
read a b ; echo $a  
fi <some.txt  
L3
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Commandes composées

Fonctions

Expressions arithmétiques

Expansion des accolades

Évaluation provoquée

Ordre des expansions

Tableaux

Pour conclure

Définition d'une fonction

- ▶ Une fonction est une commande groupée que l'on nomme et à laquelle on peut passer des paramètres
- ▶ Syntaxe :

```
function nom commande_groupée  
# ou bien (POSIX, sh)  
nom () commande_groupée
```

- ▶ Une fois définie une fonction devient une primitive, elle partage l'environnement du shell. Elle peut donc être vue comme un sous-script
- ▶ Une fonction retourne comme code erreur celle de la dernière commande de son corps, ou bien celle spécifiée par la primitive `return`
- ▶ Des arguments peuvent être donnés à une fonction. Celle-ci les récupère alors dans les arguments positionnels `$1`, `$2`, ...

Des exemples

```
% ismp3() { file $1 | grep -sq 'MPEG.*layer III'; }
% ismp3 Blackwater-park.mp3 && echo Fichier MP3
Fichier MP3
% function mkpw {
    head /dev/urandom | uuencode -m - | sed -n 2p \
        | cut -c1-${1:-8}; }
% mkpw
RfnzjhnR
% mkpw 10
JtrvYwELqw
% moy() {
    echo '(' $(tr ' ' '+' <<<"$@" ")_/_$#)_ " | bc -l
}
% m=$(moy 1 2 3 4 5); echo $m
3.0000000000000000
% function noaccent { sed 'y/éèêëàùûî/eeeeauui/; }
% noaccent < code.c | lpr
```


Portées des variables

- La fonction partage l'environnement du script :
 - les variables du scripts sont modifiables dans la fonction
 - la fonction peut définir de nouvelles variables utilisables hors de la fonction

```
% function adda { a=$(expr $a + 1); }  
% a=4; echo $a  
4  
% adda; echo $a  
5
```

- On peut définir des variables locales à l'aide de la primitive `local` :

```
% function repete {  
  local s=''  
  local i  
  for i in $(seq $2); do  
    s="$s$1"  
  done  
  echo "$s"; }  
% repete '*' 5  
*****
```

- Les arguments positionnels d'une fonction sont locales à la fonction (et n'écrasent pas les arguments positionnels du script ou de la fonction appelante)

Fonctions récursives

- Une fonction peut être récursive, pas de limite sur le nombre d'appels :

```
% repeterrec () {  
    if [[ $2 -gt 0 ]]; then  
        echo "$1$(repeterrec_$1_$(expr_$2_-_1)) "  
    fi  
}  
% repeterrec z 4  
zzzz
```

Un exemple plus programmatique

```
#!/bin/bash
# usage: ./creation_texte texte
function reverse {
    local LINE="$1"
    local RLINE=''
    local i=0
    while [[ $i -le ${#LINE} ]]; do
        RLINE=${LINE:$i:1}$RLINE
        i=$((expr $i + 1 ))
    done
    echo $RLINE
}
LINE="$1" i=1
while [[ $i -le ${#LINE} ]]; do
    echo ${LINE:0:$i}
    i=$((expr $i + 1 ))
done
i=0 s=`reverse $LINE`
while [[ $i -lt ${#LINE} ]]; do
    echo ${s:$i}
    i=$((expr $i + 1 ))
done
```

```
% ./creation_texte abcd
a
ab
abc
abcd
dcba
cba
ba
a
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

- Commandes composées

- Fonctions

- Expressions arithmétiques**

- Expansion des accolades

- Évaluation provoquée

- Ordre des expansions

- Tableaux

Pour conclure

Expressions arithmétiques (1)

- ▶ Beaucoup d'exemples avec `expr` sont inefficaces pour un grand nombre d'itérations
- ▶ `bash` permet l'évaluation d'expression arithmétique entière (POSIX, utilisable dans `sh`)
- ▶ Syntaxe : `$((expression))`
- ▶ `expression` est une expression arithmétique avec :
 - ▶ des constantes (entières)
 - ▶ des variables
 - ▶ des opérateurs : les mêmes qu'en C : `+` `-` `*` `/` `%` `()` `++` `-`, etc (voir `bash man /ÉVALUATION ARITHMÉTIQUE`)
- ▶ Dans une expression arithmétique, les identificateurs sont pris comme des variables sans risque de confusion avec les constantes qui sont entières. L'usage de `$` est donc superflue

```
% a=23
% echo $((a*10+12))
242
```

Expressions arithmétiques (2)

- Tous les exemples précédents avec `expr` peuvent et devraient être remplacés par `$ (())`
- On se rappelle du script `nlines`

```
1  #!/bin/bash
2  co=0
3  while read line; do
4      co=$((expr $co + 1))
5  done < $1
6  echo Le fichier $1 contient $co lignes
```

```
% time ./nlines shell.org
Le fichier shell.org contient 2507 lignes

real      0m6.477s
user      0m2.106s
sys       0m2.785
```

Expressions arithmétiques (3)

► Le script `nlines` plus efficace !

```
1  #!/bin/bash
2  co=0
3  while read line; do
4      co=$((co+1))
5  done < $1
6  echo Le fichier $1 contient $co lignes
```

```
% time ./nlines_arith shell.org
Le fichier shell.org contient 2507 lignes

real      0m0.034s
user      0m0.027s
sys       0m0.007s
```

► On a économisé le lancement de 2507 processus `expr` !

Autres instructions arithmétiques

- ▶ La primitive `let` : tout ce qui suit `let` est considéré comme arithmétique :
 - ▶ `let co=co+1`
 - ▶ `let co++`
 - ▶ mise à l'exposant : `let co=co**2`
 - ▶ tous les opérateurs C logiques, bits, arithmétiques, pré et post incrémentations sont autorisés (`help let`)

- ▶ La boucle `for` arithmétique : `for ((EXPR; EXPR; EXPR))`

```
for (( a=0; a<100; a++)) ; do echo $a; done  
# aussi efficace que:  
for a in $(seq 100); do echo $a; done
```

- ▶ L'alternative `if` arithmétique : `if ((EXPR))`

```
if(( a<10)); then echo 'plus petit que 10';  
elif(( a<20)); then ...  
fi
```


Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Commandes composées

Fonctions

Expressions arithmétiques

Expansion des accolades

Évaluation provoquée

Ordre des expansions

Tableaux

Pour conclure

Expansion des accolades (1)

- ▶ `bash4` peut générer facilement des séquences alphanumériques
- ▶ **Syntaxe** : `{deb..fin}`
`deb` et `fin` peuvent être soit des nombres, soit un caractère

```
% echo {1..10}
1 2 3 4 5 6 7 8 9 10
% echo {a..f}
a b c d e f
% echo {A..F}
A B C D E F
% echo {J..d}
J K L M N O P Q R S T U V W X Y Z [ ] ^ _ ` a b c d
% echo {a..F}
a ` _ ^ ] [ Z Y X W V U T S R Q P O N M L K J I H G F
% echo {1..f}
{1..f}
```

- ▶ Les séquences générées sont bien indépendantes des fichiers du répertoire courant

Expansion des accolades (2)

- La commande `seq` est spécialisée sur les nombres (entiers ou flottants)
- On peut mettre plusieurs séquences : la règle est que la séquence la plus à gauche varie le moins souvent :

```
% echo {a..d}{1..4}  
a1 a2 a3 a4 b1 b2 b3 b4 c1 c2 c3 c4 d1 d2 d3 d4
```

- Bien pratique pour générer des noms de fichiers :

```
% echo file{1..5}.txt  
file1.txt file2.txt file3.txt file4.txt file5.txt
```

- Les séquences sont expansées avant les variables, donc bornes statiques ! ou utilisation de `seq` si possible, ou ... diapositive suivante

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Commandes composées

Fonctions

Expressions arithmétiques

Expansion des accolades

Évaluation provoquée

Ordre des expansions

Tableaux

Pour conclure

La primitive eval

- ▶ Attention, primitive importante et subtile !
- ▶ Syntaxe : `eval expression`
- ▶ Mécanisme : `eval` procède en deux temps :
 1. `expression` (qui peut être quelconque) est expansée
 2. le résultat de l'expansion est considérée comme une nouvelle expression à évaluer : elle sera donc expansée une seconde fois !
- ▶ Exemple : des variables qui contiennent une référence d'un autre variable

```
% set un deux trois
% a=3
% echo $$a
47406a # $$ est une variable du shell contenant le pid du shell !
% echo \$$a
$3 # on y presque ! on voudrait imprimer $3
% b=\$$a
$ echo $b
$3 # on n'est pas plus avancé !
$ eval echo \$$a $b
trois trois # ok !
```

La primitive `eval`, N-ième mot d'une phrase

► version 1

```
1  #!/bin/bash
2  position=$1
3  phrase=$2
4  set $phrase
5  eval mot='$'$position echo $mot
```

► version 2 : cas où la phrase est vide

```
1  #!/bin/bash
2  position=$1
3  phrase=$2
4  set ' ' $phrase
5  echo eval mot='$'$((position+1))
```

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

- Commandes composées

- Fonctions

- Expressions arithmétiques

- Expansion des accolades

- Évaluation provoquée

- Ordre des expansions**

- Tableaux

Pour conclure

Ordre des expansions

- Dans l'ordre
 1. Développement des accolades
 2. Développement du tilde : `~` est remplacé par la valeur de `$HOME` et `~nom` est remplacé par le répertoire de l'utilisateur `nom`
 3. Remplacement et calculs des variables : `$` et `${...}`
 4. Substitution de commandes : ``...`` ou `$(...)`
 5. Évaluation arithmétique : `$(...)`
 6. Découpage en mots (selon `$IFS`)
 7. Développement des noms de fichiers (motifs du shell)
- Pendant ces phases : suppression des quotes (ceux qui sont non protégés)

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

- Commandes composées

- Fonctions

- Expressions arithmétiques

- Expansion des accolades

- Évaluation provoquée

- Ordre des expansions

- Tableaux

Pour conclure

Tableaux (1)

- ▶ En `bash` deux types de tableaux
 - ▶ indexés par des entiers
 - ▶ indexés par des chaînes de caractères : tableaux associatifs (`bash4`)
- ▶ Pas de limitation de tailles
- ▶ Les tableaux ne sont pas contigus en mémoire, il peut y avoir des indices pour lesquels il n'y a pas valeur
- ▶ Les indices commencent à zéro
- ▶ `bash` utilise beaucoup de tableaux internes pour son fonctionnement :

```
% declare -a # liste les tableaux (tout type) en mémoire
declare -a BASH_ARGC='()'
declare -a BASH_ARGV='()'
declare -a BASH_LINENO='()'
declare -ar BASH_REMATCH='()'
...
```

Tableaux (2)

► Déclaration :

- tableau normal (facultatif) : `declare -a tab`
- tableau associatif (obligatoire) : `declare -A tab`

► Affectation d'un élément d'indice ou de clé `i` : `tab[i]=valeur`

```
% tab[3]="Bonjour_le_monde"  
% ass[la terre]="est_belle"  
bash: la terre : erreur de syntaxe dans l'expression (le  
symbole erroné est « terre »)  
% declare -A ass  
% ass[la terre]="est_belle"
```

La clé peut contenir des espaces. Inutile de protéger avec des quotes
les [] font office de délimiteurs

► Accès à l'élément d'indice ou de clé `i` : `${tab[i]}`

Tableaux (3)

- Initialisation : on peut déclarer et initialiser dans la foulée un tableau :
`tab=(elem_1 elem_2 ... elem_n)`
où `elem_i` est de la forme `[indice]=valeur` ou simplement `valeur`
- Pour les tableaux associatifs la première forme est obligatoire
- Pour les tableaux normaux, la première forme est optionnelle : par défaut les indices commencent à 0
- Pas de besoin de `declare` pour des initialisations

```
% premiers=(2 3 5 7 11 13)
% joursparmois=([janvier]=31 [fevrier]=28 [mars]=31
[avril]=30 [mai]=31 ...)
% numeromois=([1]=janvier fevrier mars avril ...)
% sparse=([10]="dix" [8]="huit")
```

Tableaux (4)

- Lecture d'une ligne et affectation dans un tableau avec `read` :

```
|| read -a tab
```

la ligne lue est découpée en mots (`$IFS`), chaque mot est affecté à un élément du tableau

- Parcours d'un tableau :

- `${tab[@]}` est expansé en liste des valeurs du tableau
- `"${tab[@]}"` est expansé en liste des valeurs du tableau, chaque valeur étant protégée par des `"`
- `${!tab[@]}` est expansé en liste des indices du tableau
- `${#tab[@]}` est le nombre d'éléments d'un tableau
- attention** : `${#tab[i]}` est la longueur de la valeur d'indice `i` du tableau

- Les tableaux sont donc naturellement parcouru à l'aide de boucle `for`

Tableaux : exemples

► N-ième mots d'une phrase

```
1  #!/bin/bash
2  # ./mot phrase n
3  read -a mots <<< $1
4  echo ${mots[${2-1}]}
```

```
% ./mot "Chacun_cherche_son_chat" 1
cherche
```

► Parcours par indice à la C :

```
for i in $(seq 0 ${#tab[@]}-1); do
    echo "tab[$i]=${tab[$i]}"
done
```

à comparer avec :

```
for i in ${!tab[@]}; do
    echo tab[$i]=${tab[$i]}
done
```

Tableaux : exemples

- Soit un fichier `europe.csv` chaque ligne ayant la structure :
nom pays;capital;nombre d'habitant;autres informations ...
- Le script suivant lit le fichier et affiche le nom du pays le plus peuplé ainsi que le nombre d'habitants et la capitale :

```
1  #! /bin/bash
2  IFS=";"
3  declare -A capitale population
4  while read pays cap hab reste; do
5      capitale[$pays]=$cap
6      population[$pays]=$hab
7  done <europe.csv
8  maxp=0
9  for pays in ${!capitale[@]}; do
10     if [[ ${population[$pays]} -gt $maxp ]]; then
11         maxp=${population[$pays]}
12         imaxp=$pays
13         icap=${capitale[$pays]}
14     fi
15 end
16 echo "Pays_le_plus_peuplé:_$imaxp_population_$maxp_capitale_$icap"
```

Tableaux et boucles : attention aux espaces !

► Soit le code suivant :

```
1 #!/bin/bash
2 tab=("bonjour_le_monde" "hello_world")
3 for a in ${tab[@]}; do
4     echo $a
5 done
```

```
% ./monde
bonjour
le
monde
hello
world
```

► Correction :

```
1 #!/bin/bash
2 tab=("bonjour_le_monde" "hello_world")
3 for a in "${tab[@]}"; do
4     echo $a
5 done
```

```
% ./mondeok
bonjour le monde
hello wold
```


Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Pour conclure

- Mise au point

- Efficacité des scripts

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Pour conclure

Mise au point

Efficacité des scripts

Mise au point (1)

- Premier conseil : utiliser `emacs` en mode majeur `sh-mode`, et utiliser les modèles dans le menu `Sh`
- Séquence typique :

```
% emacs -nw nouveauscript  
A-x sh-mode  
C-c :  
...  
C-x C-s
```

`emacs` crée le fichier, positionne le flag d'exécution (`chmod +x`) ainsi que le `shebang`

- Les modèles d'`emacs` limitent les erreurs de syntaxe
- La couleur et l'indentation peuvent détecter des erreurs de syntaxe
- Le mode mineur `flycheck-mode` permet une vérification à la volée de la syntaxe
- Le jour de l'examen, ayez votre `.emacs.d` et votre `.emacs` prêts sur une clef USB

Mise au point (2)

- Modification du comportement de `bash` via la primitive `set`
- `set -u` provoque une erreur si une variable n'est pas définie :

```
% set -u
% unset toto
% echo $toto
-bash: toto: variable sans liaison
```

- `set -v` provoque l'affichage de la ligne avant les phases d'expansion

```
% a=1
% set -v
% echo $a
echo $a
1
% set +v
```

- `set -n` qui n'exécute pas les commandes : `set -nv` permet de voir le déroulement du script sans exécutions des commandes

Mise au point (3)

- `set -x` provoque l'affichage de la ligne après les phases d'expansion :

```
% set -x
% echo $USER
+ echo bereziat
bereziat
% set +x
```

- Les options de `set` peuvent être passées directement à `bash` (pour éviter de modifier le fichier) :

```
% echo 'cmd=pwd'
x=$cmd
$x
' > monscript
% bash -x monscript
+ cmd=pwd
+ x=pwd
+ pwd
/home/bereziat
```

- Les options `-x` et `-v` sont très verbeuses, on peut donc utiliser `set -x` et `set +x` pour les activer *localement*

Plan

Introduction

Notions de base sur l'OS Linux

L'interprète de commandes

L'écriture de scripts

Scripts avancés

Pour conclure

Mise au point

Efficacité des scripts

Le bon langage pour un problème donné

- Exemple avec le problème de compter les lignes d'un fichier (les sources de ce cours, 2641 lignes)

commande	temps (ms)
wc -l	3
sed -n \$=	3
nlines.c	3
perl '{ \$n++ } END { print \$n }'	5
nlines_arith	31
nlines.py	34
nlines	6741

```
1 #include <stdio.h>
2 #define BUFSIZE 1024
3 int main(int argc, char **argv) {
4     char buf[BUFSIZE];
5     int n;
6     for( n=0; fgets( buf, BUFSIZE, stdin); n++);
7     printf("%d\n",n);
8     return 0;
9 }
```

```
1 #!/bin/usr/env python3
2 import sys
3 f = open(sys.argv[1], 'r')
4 lignes = f.readlines()
5
6 print (len(lignes))
```

Scripts efficaces

- ▶ `bash` est spécialisé dans le lancement des processus, mais pour autant, lancer un processus est coûteux.
- ▶ Comment écrouler une machine `Linux` ?

```
#!/bin/bash  
# rater une récursion par exemple  
$0 &
```

- ▶ Moralité : lancer une dizaine de processus, pas de problème, plusieurs milliers bof !
- ▶ La stratégie :
 - ▶ résoudre le problème avec une suite de commandes. En général, c'est la solution la plus rapide (en temps d'exécution) et aussi à écrire : mais il faut la voir !
 - ▶ solution plus 'programmation classique' avec boucle `while` typiquement : penser à limiter le plus possible le nombre de commandes dans le corps de la boucle. Normalement `bash` est suffisamment complet pour ne pas avoir à utiliser une commande extérieure
 - ▶ évidemment, il peut y avoir des solutions mixtes : une série de commandes et de boucles `while`

Exemple d'un cas réel (partiel octobre 2012)

- ▶ Fichier CVS de déplacements de véhicules dans la ville de Chicago
- ▶ Une ligne du fichier : un déplacement d'un véhicule. Format :

`date,marque,couleur,immat,etat,adresses1,adresse2,motif,id`

- ▶ Exemple :

```
10/16/2012,Buick,Green,K741253,Illinois,1744,N,  
ST LOUIS,AVE,3535,W,BLOOMINGDALE,,Water Management,  
12-01756344
```

- ▶ Adresse sur 4 champs : numéro, direction, voie, suffixe
- ▶ Fichier trié par ordre chronologique (plus récent en tête)

La question !

- ▶ Quelle est la date à laquelle il y a eu le plus de déplacement ?
- ▶ Exemple :

```
|| % date_max_deplacements Relocated_Vehicule16.cvs  
|| 09/27/2012
```

- ▶ Digression : en examen sur machine, on recommande mettre le répertoire courant dans le `PATH`, soit d'écrire dans son shell au début de l'épreuve : `PATH=. : $PATH`
- ▶ L'idée pour répondre à la question : compter les déplacements par jour PUIS extraire le maximum
- ▶ On supposera que ce max est unique

Approche programmatique (1)

1. Boucler sur les lignes du fichier à traiter

```
#!/bin/bash  
while read line; do  
  
done <$1
```

Approche programmatique (2)

1. Boucler sur les lignes du fichier à traiter
2. Extraire la date

```
#!/bin/bash
IFS=', '
while read date reste; do

done <$1
```

Approche programmatique (3)

1. Boucler sur les lignes du fichier à traiter
2. Extraire la date
3. Compter les répétitions de cette date

```
#!/bin/bash
IFS=', '
REP=1
while read date reste; do
    if [[ "$date" == "$prev" ]]; then
        REP=$((REP+1))
    else
        prev=$date; REP=1
    fi
done <$1
```

Approche programmatique (4)

1. Boucler sur les lignes du fichier à traiter
2. Extraire la date
3. Compter les répétitions de cette date
4. Calculer le max

```
#!/bin/bash
IFS=', ' REP=1
MAX=1
while read date reste; do
    if [[ "$date" == "$prev" ]]; then
        REP=$((REP+1))
    else
        prev=$date; REP=1
    fi
    if [[ "$REP" -gt "$MAX" ]]; then
        MAX=$REP DMAX=$REP
    fi
done <$1
echo $DMAX
```

Approche programmatique (5)

- ▶ Cette solution repose sur le tri par date du fichier
- ▶ Si ce n'était pas le cas, il suffirait de trier avec la commande `sort` :

```
sort -t',' -nrk1 $1 | (  
    while read date reste; do  
        ...  
    done  
    echo $DMAX  
)
```

- ▶ Attention au sous-shell !

Approche programmatique (6)

- Autre solution : utilisation d'un tableau associatif (fréquence de chaque date), puis recherche du max. Pas besoin de trier :

```
1 declare -A deps
2 IFS=', '
3 while read date reste; do
4     ${deps[$date]} = "a${deps[$date]}"
5 done < $1
6 max=0 dmax=
7 for d in ${!deps[@]}; do
8     val=${deps[$d]}
9     if [[ ${#val} !gt $max ]]; then
10         max=${#val}
11         dmax=$d
12     fi
13 done
14 echo $dmax
```


Approche par compositions (1)

- Chaque tâche est réalisée par la bonne commande !
- Extraire la date :

```
$ cut -d',' -f1 Relocated_Vehicles16.csv  
10/16/2012  
10/16/2012  
...
```

- et compter les répétitions :

```
$ cut -d',' -f1 Relocated_Vehicles16.csv | uniq -c  
53 10/16/2012  
73 10/15/2012  
21 10/14/2012
```

Approche par compositions (2)

- Puis tri numérique sur la première colonne :

```
$ cut -d',' -f1 Relocated_Vehicles16.csv \  
  | uniq -c | sort -nr \  
  92 09/07/2012 \  
  79 10/12/2012 \  
  75 08/31/2012
```

- et extraction de la première ligne :

```
$ cut -d',' -f1 Relocated_Vehicles16.csv \  
  | uniq -c | sort -nr | head -1 \  
  92 09/27/2012
```

- Et enfin, récupérer la date :

```
% set $(cut -d',' -f1 Relocated_Vehicule16.csv \  
  | uniq -c | sort -nr | head -1 ) \  
% echo $2 \  
09/27/2012
```

Ici la commande `cut` n'est pas pratique à cause des espaces multiples

Cas d'un maxima multiple

- Approche programmatique : distinguer le cas où le max est atteint (DMAX peut contenir une liste de date) du cas où il n'est pas atteint (et on réinitialise DMAX) :

```
#!/bin/bash
IFS=', ' REP=1
MAX=1
while read date reste; do
    if [[ "$date" == "$prev" ]]; then
        REP=$((REP+1))
    else
        prev=$date; REP=1
    fi
    if [[ "$REP" -gt "$MAX" ]]; then
        MAX=$REP DMAX=$REP
    fi
    [[ "$REP" == "$MAX" ]] && DMAX="$DMAX_$date"
done <$1
echo $DMAX
```

Cas d'un maxima multiple

- Approche composition : plus subtile ! on peut procéder en deux temps :
 1. calcul de la valeur du max
 2. rechercher ces valeurs (dans la sortie d'`uniq`)
- On doit donc utiliser un fichier temporaire :

```
$ tempfile=$(mktemp)
$ cut -d',' -f1 Relocated_Vehicule16.cvs \
  | uniq -c | sort -nr > $tempfile
% set $(head -1 $tempfile)
% sed -nr "s/^_+$1_+//p" $tempfile
% rm $tempfile
```