Bash

Julien Baste

IUT de Lille

Séance 04

2025/2026

Introduction

Aujourd'hui on va revenir sur l'utilisation de `bash`.

- `bash` est une commande.
  - On peut la lancer depuis une session `bash`.
- Comme toute commande elle a une page dans le manuel : `man bash`
  - Le manuel de `bash` est long
  - Il contient tout ce qu'il y a à savoir sur l'utilisation de `bash`

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :

4. Retour en 1

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. **Transformations successives des mots de la ligne** :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
4. Retour en 1

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. **Transformations successives des mots de la ligne** :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   2. **Exécution de la ligne transformée** :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

Transformations

Les transformations sont effectuées grâce à

- des **caractères spéciaux** (*meta-characters*)

  espace, tabulation,

  `| & ; ( ), < >`

- des **mots réservés**, notamment :
  - ceux commençant par les caractères : `$ ~ `
  - ceux contenant les caractères : `* ? [ ] ^ -`
  - les mots réduits aux caractères : `{ }`

Tous ces caractères ont donc **un sens particulier** pour le shell

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'un *backslash* \
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'un *backslash* \
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

- **Protection complète** : entourer la zone à protéger par des guillemets simples (*quote*) '
  ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'un *backslash* `\`
  - ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

- **Protection complète** : entourer la zone à protéger par des guillemets simples (*quote*) `'`
  - ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.

- **Protection simple** : entourer la zone à protéger par des guillemets doubles (*double-quote*) `"`
  - ➠ seul `\`, `$` et `` ` `` sont interprétés comme des caractères spéciaux.

Variables

Le shell est un langage de programmation ⇒ Il y a des variables.

- Les noms de variables sont composés de
  - lettres
  - chiffres
  - le caractère de soulignement _.
- La valeur d'une variable est forcément une suite de caractères
  - équivalent à un type string en ijava
- une variable ne peut être modifiée que par une affectation
  - Il n'y a pas de réévaluation des variables

Développement des variables

Une variable peut être appelée en utilisant :

$\$\{\langle nom \rangle\}$ ou $\$\langle nom \rangle$

- le shell remplace la variable par la chaine de caractère qui lui a été affectée
- les accolades `{` et `}` sont optionnelles
- si la variable n'existe pas, le shell remplace par une chaîne vide

Quelques variables particulières

variables	description
`$HOME`	Le chemin absolu du répertoire principal de l'utilisateur.
`$PWD`	Le répertoire de travail en cours.
`$PATH`	Liste des répertoires dans lesquels le shell recherche les commandes à exécuter. Les répertoires sont séparés par des deux-points `:`.
`$?`	Le code de retour de la dernière commande exécutée. → vaut 0 en cas de réussite et autre chose en cas d'échec
`$$`	Le PID du processus exécutant le shell en cours.
`$PPID`	Le PID du processus père du processus `$$`.
`$!`	Le PID du dernier processus exécuté en tâche de fond.
`$PS1`	Le message d'invite (*prompt*) principal du shell.
`$PS2`	L'invite secondaire du shell.

Affectation des variables

- Variables classiques :

  Variables définies dans

  - le contexte d'exécution du processus dans lequel elles sont déclarées.

  Utilisation : ⟨nom⟩=⟨valeur⟩

- Variables d'environnement :

  Variables définies dans

  - le contexte d'exécution du processus dans lequel elles sont déclarées et
  - tous les contextes d'exécution des processus fils.

  Utilisation : ⟨nom⟩=⟨valeur⟩

  puis : `export` ⟨nom⟩

Substitution des commandes

Les passages `$(`⟨cmd⟩`)` ou `` `⟨cmd⟩` `` d'une ligne de commande :

- lancent un processus fils exécutant ⟨cmd⟩
- sont remplacés par le contenu de la sortie standard de ce processus

```
echo 2 + 3 = $(expr 2 + 3)
                ⇓
echo 2 + 3 = 5
```

Résultat de l'exécution : `2 + 3 = 5`

Développement des chemins de fichiers

Un **joker** est un caractère utilisé *à la place* d'un (ou de plusieurs) autre(s).

Un **modèle** (*glob pattern*) est un mot qui contient un ou plusieurs jokers.

Lorsque le shell trouve un modèle sur la ligne de commande :

1. il cherche la liste des fichiers dont le **chemin correspond** au modèle
2. il remplace le modèle par les **chemins des fichiers** correspondant en les séparant par **un** espace.
3. si aucun fichier ne correspond le modèle est laissé tel quel

Jokers (*Méta-caractères*)

- `*` remplace n'importe quelle **suite de caractères** (y compris vide)
- `?` remplace n'importe quel **caractère**
- `[` ⟨liste⟩ `]` remplace n'importe quel caractère de ⟨liste⟩
  - on spécifie la liste des caractères que l'on veut représenter
  - `^` placé en début de liste signifie que l'on veut remplacer n'importe quel caractère **non présent** dans la liste
  - `-` utilisé dans la liste définit un intervalle plutôt qu'un ensemble de valeurs

Exemples de modèles de chemins

`f*`	Tous les fichiers dont le nom commence par `f`
`f?`	Tous les fichiers dont le nom fait 2 caractères et commence par `f`
`*.java`	Tous les fichiers dont le nom se termine par `.java`
`tit[oi]`	Les fichiers `titi` et `tito`
`[a-z]*[0-9]`	Tous les fichiers dont le nom commence par une minuscule et se termine par un chiffre
`[^a-z]*`	Tous les fichiers dont le nom **ne commence pas** par une minuscule
`???*`	Tous les fichiers dont le nom est composé d'au moins 3 caractères.
`../*`	Tous les fichiers du répertoire parent.
`/tmp/*.log`	Tous les fichiers se terminant par `.log` du répertoire `/tmp`.
`/tmp/.[^.]*`	*presque* tous les fichiers cachés du répertoire `/tmp`.

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables

2. Substitution de commandes

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables
2. Substitution de commandes
3. Développement des chemins de fichier

Découpage de la ligne en commandes

- Les mots sont séparés par des **blancs** non protégés

  Un blanc est un caractère espace ou une tabulation

- Une commande est un mot quelconque :
  - situé en **première position** de la ligne
  - ou situé **juste après un séparateur** de commandes
  - éventuellement **suivies** par des paramètres (d'autres mots)
  - ne contenant pas le caractère =
- Les commandes peuvent être séparées par :
  - des **points-virgules** ;

    Attendre la fin d'une commande avant de passer à la suivante
  - des **esperluètes** &

    Ne pas attendre la fin d'une commande pour passer à la suivante
  - des **tubes** |

    Démarrer les commandes en parallèle en les connectant

Opérateurs

Autres séparateurs possibles : opérateurs logiques séquentiels paresseux

- **ET** `cmd1 && cmd2`

  `cmd2` est exécutée si et seulement si `cmd1` a réussi
- **OU** `cmd1 || cmd2`

  `cmd2` est exécutée si et seulement si `cmd1` a échoué

➠ **Le shell essaie de faire réussir la ligne**

- évaluation de gauche à droite
- dès qu'on sait que la séquence ne peut pas réussir (ou qu'elle est déjà réussie) on arrête son évaluation

Définition de réussite (échec) dans le manuel

Recherche et exécution de la commande

- Si la commande est interne elle est exécutée directement
- Sinon
  - Recherche répertoire et fichier
    - si la commande contient au moins un caractère `/`

      extraction du répertoire et du nom de fichier
    - sinon pour tous les répertoires définis dans la variable `$PATH`

      recherche d'un fichier correspondant à la commande
  - Exécution ou erreur
    - si un fichier a été trouvé **et qu'il est exécutable**

      exécution du code qu'il contient dans un nouveau processus
    - sinon envoi d'un message d'erreur

Commandes internes

Les commandes internes (*builtins commands*) sont traitées directement :

- pas de nouveaux processus pour les exécuter
- leur code est intégré au shell
- peuvent modifier le contexte d'exécution du shell courant

commandes	description
`cd`	change le répertoire courant
`echo`	envoie ses arguments sur la sortie standard
`pwd`	envoie le nom du répertoire courant sur la sortie standard
`.` ou `source`	lit et exécute les commandes d'un fichier
`exec`	remplace le code par une autre commande
`exit`	termine le processus courant
`read`	affecte une variable en lisant l'entrée standard
`!`	inverse la réussite de la commande suivante

Documentées dans la page du manuel de `bash` et par la commande `help`

Commandes externes

Les commandes externes necessitent une communication avec le système :

- code stocké dans un **fichier régulier exécutable**
- rangée dans un répertoire de la hiérarchie du système

  la convention est d'utiliser des dossiers nommés `bin`
- recherchée dans une liste de dossiers

  répertoires séparés par des `:` dans la variable `$PATH`
- exécutée dans un nouveau processus par le shell
- elles **ne peuvent pas** modifier le contexte d'exécution du shell

Quelques exemples :

`/bin/ls`, `/bin/cp`, `/bin/mv`, `/bin/mkdir`, `/usr/bin/vi`,
`/usr/local/bin/regarder_ecran`

Algorithme classique d'un shell

1. Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard
2. Lire une ligne de commandes sur l'entrée standard
3. Interpréter cette ligne :
   1. **Transformations successives des mots de la ligne** :
      - Développement des variables
      - Substitution de commandes
      - Développement des noms de fichiers
   2. **Exécution de la ligne transformée** :
      - Découpage de la ligne en commandes
      - Préparation des processus (redirections, etc.)
      - Recherche de commande(s)
      - Exécution ou envoi d'un message sur la sortie d'erreur
4. Retour en 1

C’est tout pour aujourd’hui