

Systèmes centralisés – Shell

1 IMA

28 février 2017

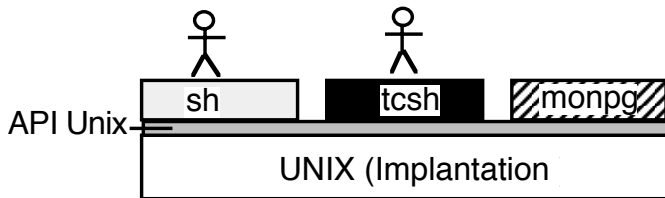
Contenu de cette partie

- rôle d'un interpréteur de commandes
- utilisation interactive
- scripts
- commandes avancées
- expressions régulières

Plan

- ❶ Rôle d'un interpréteur de commandes (shell)
- ❷ Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- ❸ Programmation dans le large
 - Composition de commandes
 - Scripts
- ❹ Commandes avancées

Combien d'interfaces sur cette figure ?



Rôle d'un interpréteur de commandes (shell)

Shell = interface utilisateur pour les services système

- services système
 - gestion des ressources
 - gestion des processus
 - utilitaires
- interface interactive
 - langage de commande
 - interpréteur de commandes
 - facilités de saisie
 - supervision des tâches lancées
- programmation dans le large : automatisation des utilisations répétitives/régulières des services systèmes
 - composition de commandes
 - idée : construction de commandes complexes, adaptées à un usage particulier par combinaison de commandes élémentaires
 - scripts

Comparatif

Shell	Commande	interactivité	programmation
C shell	csh	+	--
Toronto C shell	tcsh	++	--
Bourne shell	sh	-	+
Zero shell	zsh	++	+
Korn shell	ksh	+	++
Bourne again shell	bash	++	+

Choix du shell

- fixé dans le fichier `/etc/passwd`
- commentaire en première ligne du script de la forme
`#!/chemin_d_acces/shell_choisi options`

Exemples :

- `#!/bin/csh`
- `#!/bin/sh -x`
- lancement comme commande
`sh mon_programme`

Note : le **bash** est le shell utilisé pour la suite de cette présentation (et pour les TP)

Plan

- 1 Rôle d'un interpréteur de commandes (shell)
- 2 **Interpréteur de commandes interactif : principe et utilisation**
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- 3 Programmation dans le large
 - Composition de commandes
 - Scripts
- 4 Commandes avancées

Plan

- 1 Rôle d'un interpréteur de commandes (shell)
- 2 Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- 3 Programmation dans le large
 - Composition de commandes
 - Scripts
- 4 Commandes avancées

Shell = interface utilisateur pour les services système

→ le shell

- permet d'appeler les différentes fonctions du système depuis une interface utilisateur **interactive**
- propose donc un **langage de commande** : un ensemble de commandes que l'utilisateur pourra saisir (texte/graphique) et que le shell traduira en appels aux fonctions système.
- fonctionne comme un **interpréteur de commandes** :

répéter

```
lgn := lire_une_ligne_de_commande();
```

```
appel_système := interpréter(lgn);
```

```
exécuter(appel_système);
```

sans fin

Ce code présente un inconvénient. Lequel ?

Comment permettre un langage de commande extensible ?

Principe de base des shells Unix

- commande = code exécutable + paramètres
- le shell lance un nouveau processus pour exécuter le code contenu dans le fichier indiqué par la ligne de commande

répéter

```
lgn := lire_une_ligne_de_commande();  
programme_et_paramètres := interpréter(lgn);  
id_fils := lancer_processus(programme_et_paramètres);  
attendre_terminaison(id_fils);
```

sans fin

Note : pour des raisons d'efficacité ou de mise en œuvre, certaines commandes restent intégrées au shell (**commandes internes**)



Exemple

```
sh-3.2$ type cd
cd is a shell builtin
sh-3.2$ type ls
ls is hashed (/bin/ls)
sh-3.2$ ls
Makefile hanoi.c main.c stack.c stack.h
sh-3.2$ set -x
sh-3.2$ ls *.c
+ ls hanoi.c main.c stack.c
hanoi.c main.c stack.c
sh-3.2$
```

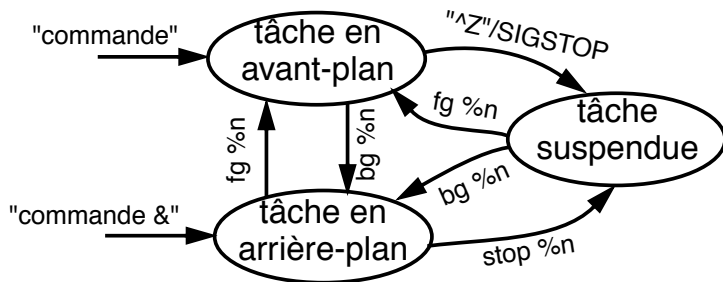
- *cd* = commande interne, *ls* = programme externe
- "*ls *.c*" est traduit en "*ls hanoi.c main.c stack.c*"
- *ls* est complété (variable *\$CLASSPATH*) en */bin/ls*
- un processus est créé, pour exécuter le contenu du fichier "*/bin/ls*", avec les paramètres "*hanoi.c*", "*main.c*", "*stack.c*"



Plan

- ❶ Rôle d'un interpréteur de commandes (shell)
- ❷ Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- ❸ Programmation dans le large
 - Composition de commandes
 - Scripts
- ❹ Commandes avancées

Supervision des tâches lancées



- jobs fournit la liste des tâches

Aide à la saisie

Historique des commandes saisies (navigation, édition)

Abréviations

- chemins relatifs, chemins prédéfinis (., .., ~)
- complétion
- motifs pour les noms de fichiers (?, * ...)

Personnalisation de l'environnement

- alias

alias rm 'rm -i'

- Fichiers (scripts) de configuration.

Exemples (bash) :

- en début de session : /etc/profile, puis ~/.bash_profile, puis ~/.bash_login, puis ~/.profile
- en fin de session : ~/.bash_logout
- ~/.bashrc au lancement d'un shell en cours de session

- Variables d'environnement

- utilisées par le shell,
- et accessibles aux processus lancés depuis le shell

Variables shell

Le shell permet de définir des variables

- non typées, pas de déclaration
- **\$**V désigne la valeur de la variable d'identifiant V
- Syntaxe de l'affectation : id_variable= valeur (sans espaces)

```
sh-3.2$ x=bonjour  
sh-3.2$ echo x
```

Variables shell

Le shell permet de définir des variables

- non typées, pas de déclaration
- $\$V$ désigne la valeur de la variable d'identifiant V
- Syntaxe de l'affectation : `id_variable=` valeur (sans espaces)

```
sh-3.2$ x=bonjour
```

```
sh-3.2$ echo x
```

```
x
```

```
sh-3.2$ echo $x
```

Variables shell

Le shell permet de définir des variables

- non typées, pas de déclaration
- **\$V** désigne la valeur de la variable d'identifiant V
- Syntaxe de l'affectation : `id_variable=`**valeur** (sans espaces)

```
sh-3.2$ x=bonjour
sh-3.2$ echo x
x
sh-3.2$ echo $x
bonjour
sh-3.2$ echo $xy
```

Variables shell

Le shell permet de définir des variables

- non typées, pas de déclaration
- **\$V** désigne la valeur de la variable d'identifiant V
- Syntaxe de l'affectation : `id_variable=` **valeur** (sans espaces)

```
sh-3.2$ x=bonjour
sh-3.2$ echo x
x
sh-3.2$ echo $x
bonjour
sh-3.2$ echo $xy

sh-3.2$ echo ${x}y
```

Variables shell

Le shell permet de définir des variables

- non typées, pas de déclaration
- $\$V$ désigne la valeur de la variable d'identifiant V
- Syntaxe de l'affectation : $\text{id_variable}=\text{valeur}$ (sans espaces)

```
sh-3.2$ x=bonjour
```

```
sh-3.2$ echo x
```

```
x
```

```
sh-3.2$ echo $x
```

```
bonjour
```

```
sh-3.2$ echo $xy
```

```
sh-3.2$ echo ${x}y
```

```
bonjoury
```

```
sh-3.2$
```

Dans le contexte des scripts (cf infra), un certain nombre de variables sont prédéfinies : paramètres d'appel, code retour, identifiant du processus actif...

« Héritage » des variables

Les variables définies dans un processus peuvent être passées à ses fils

- passage par copie
- les variables transmises doivent avoir été déclarées comme exportées (commande **export** v1 v2... du shell)

```
sh-3.2$ x=1
sh-3.2$ y=2
sh-3.2$ export x
sh-3.2$ echo $x,$y
1,2
sh-3.2$ sh
sh-3.2$ echo $x,$y
```

« Héritage » des variables

Les variables définies dans un processus peuvent être passées à ses fils

- passage par copie
- les variables transmises doivent avoir été déclarées comme exportées (commande **export** v1 v2... du shell)

```
sh-3.2$ x=1
sh-3.2$ y=2
sh-3.2$ export x
sh-3.2$ echo $x,$y
1,2
sh-3.2$ sh
sh-3.2$ echo $x,$y
1,
sh-3.2$ x=3
```

```
sh-3.2$ y=4
sh-3.2$ echo $x,$y
```

« Héritage » des variables

Les variables définies dans un processus peuvent être passées à ses fils

- passage par copie
- les variables transmises doivent avoir été déclarées comme exportées (commande **export** v1 v2... du shell)

```
sh-3.2$ x=1
sh-3.2$ y=2
sh-3.2$ export x
sh-3.2$ echo $x,$y
1,2
sh-3.2$ sh
sh-3.2$ echo $x,$y
1,
sh-3.2$ x=3
```

```
sh-3.2$ y=4
sh-3.2$ echo $x,$y
3,4
sh-3.2$ export y
sh-3.2$ exit
exit
sh-3.2$ echo $x,$y
```


« Héritage » des variables

Les variables définies dans un processus peuvent être passées à ses fils

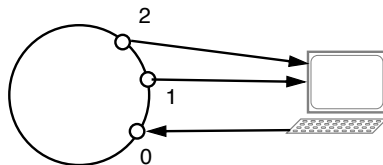
- passage par copie
- les variables transmises doivent avoir été déclarées comme exportées (commande **export** v1 v2... du shell)

sh-3.2\$ x=1	sh-3.2\$ y=4
sh-3.2\$ y=2	sh-3.2\$ echo \$x,\$y
sh-3.2\$ export x	3,4
sh-3.2\$ echo \$x,\$y	sh-3.2\$ export y
1,2	sh-3.2\$ exit
sh-3.2\$ sh	exit
sh-3.2\$ echo \$x,\$y	sh-3.2\$ echo \$x,\$y
1,	1,2
sh-3.2\$ x=3	sh-3.2\$

Utilisé pour transmettre les **variables d'environnement** :

PS1 (invite), HOME (répertoire privé), TERM (type du terminal utilisé),
PATH (liste des chemins où chercher les commandes) ...

E/S : modèle (utilisateur) de l'exécution d'un calcul



- les **processus** communiquent avec leur environnement au moyen de **flots de données**
- flot** = **file d'octets**, non structurée, non limitée
- ressources** (périphériques, fichiers...) = sources/puits pour les flots de données
→ vues et manipulées comme des **files** (fichiers séquentiels)
- un processus lancé via le shell dispose de 3 flots prédéfinis, associés à des identifiants (**descripteurs**) fixés :
entrée standard (0), sortie standard (1), sortie erreur standard (2)

Gestion des flots d'E/S à partir du shell : redirections

Syntaxe générale

cde **n>&m** redirige le flot n du processus exécutant cde vers le flot m

- La valeur par défaut de n est 1
- &m peut être remplacé par un chemin d'accès (fichier)

Cas particuliers

- C < F associe (redirige) le fichier F à l'**entrée standard** de la commande C
- C > F redirige la **sortie standard** de C vers le fichier F.
Si F existait, il est écrasé.
- C » F redirige la **sortie standard** de C vers le fichier F.
Les données produites par C sont ajoutées en fin de fichier.

Plan

- ❶ Rôle d'un interpréteur de commandes (shell)
- ❷ Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- ❸ Programmation dans le large
 - Composition de commandes
 - Scripts
- ❹ Commandes avancées

Structure d'une ligne de commande

- Une ligne de commande est une suite de mots, séparés par des espaces
- Le premier mot désigne la commande
chemin d'accès, ou identifiant prédéfini (commande interne)
- Les mots suivants sont les paramètres
- Les premiers paramètres sont souvent des options de la commande
- Les options sont souvent précédées d'un "-"

Exemple : `rm -ri file.o proc.o lulu` est décomposée en

- `rm`, qui est le nom (relatif) du fichier à exécuter (commande)
- `-ri`, qui sont les options de la commande
- `file.o`, `proc.o`, et `lulu` qui sont les paramètres



Définition de motifs pour les noms de fichiers

Le shell interprète certains caractères comme des motifs pour les noms de fichiers (chemins d'accès) :

- `*` correspond à une suite quelconque de caractères
- `?` correspond à un caractère quelconque
- `[liste_de_caractères]` correspond à un caractère quelconque de la liste

Dans ce contexte, $\alpha\text{-}\beta$ désigne l'ensemble des caractères compris entre α et β dans le jeu ASCII

- `\` permet de neutraliser (déspécialiser) le métacaractère qui suit

Après interprétation, le motif est remplacé par la liste (triée par ordre alphabétique) des chemins appariés au motif

Motifs : exercices (1/2)

Lister **tous** les fichiers du répertoire courant (fichiers cachés compris)

Motifs : exercices (1/2)

Lister **tous** les fichiers du répertoire courant (fichiers cachés compris)

```
sh-3.2$ ls -a
```

.	corexo1.tex	exoprocl1a.tex	logo-n7.jpg	td.htoc
..	corexo2.tex	gliens.tex	mybook.hva	td.image.tex
.xwkr	demofcnt1.c	html	proc.tex	td.log
Makefile	demoselect.c	intro.tex	td.aux	td.pdf
com??.tex	ent2_nonbloc.c	logo-n7.eps	td.haux	td.tex

```
sh-3.2$ ls
```


Motifs : exercices (1/2)

Lister **tous** les fichiers du répertoire courant (fichiers cachés compris)

```
sh-3.2$ ls -a
```

```
.                corexo1.tex      exoprocl1a.tex  logo-n7.jpg     td.htoc
..               corexo2.tex      gliens.tex      mybook.hva      td.image.tex
.xwkr           demofcntl.c    html            proc.tex        td.log
Makefile        demoselect.c    intro.tex       td.aux          td.pdf
com??.tex       ent2_nonbloc.c  logo-n7.eps     td.haux         td.tex
```

```
sh-3.2$ ls
```

```
Makefile        demoselect.c    intro.tex       td.aux          td.pdf
com??.tex       ent2_nonbloc.c  logo-n7.eps     td.haux         td.tex
corexo1.tex     exoprocl1a.tex  logo-n7.jpg     td.htoc
corexo2.tex     gliens.tex      mybook.hva      td.image.tex
demofcntl.c     html            proc.tex        td.log
```

```
sh-3.2$ ls *
```



Motifs : exercices (1/2)

Lister **tous** les fichiers du répertoire courant (fichiers cachés compris)

```
sh-3.2$ ls -a
```

```
.                corexo1.tex      exoprocl1a.tex  logo-n7.jpg     td.htoc
..               corexo2.tex      gliens.tex      mybook.hva      td.image.tex
.xwkr           demofcntl.c      html            proc.tex         td.log
Makefile        demoselect.c      intro.tex       td.aux           td.pdf
com??.tex       ent2_nonbloc.c    logo-n7.eps     td.haux          td.tex
```

```
sh-3.2$ ls
```

```
Makefile        demoselect.c      intro.tex       td.aux           td.pdf
com??.tex       ent2_nonbloc.c    logo-n7.eps     td.haux          td.tex
corexo1.tex     exoprocl1a.tex    logo-n7.jpg     td.htoc
corexo2.tex     gliens.tex        mybook.hva      td.image.tex
demofcntl.c     html              proc.tex        td.log
```

```
sh-3.2$ ls *
```

```
Makefile        demoselect.c      intro.tex       td.aux           td.pdf
com??.tex       ent2_nonbloc.c    logo-n7.eps     td.haux          td.tex
corexo1.tex     exoprocl1a.tex    logo-n7.jpg     td.htoc
corexo2.tex     gliens.tex        mybook.hva      td.image.tex
demofcntl.c     html              proc.tex        td.log
```

```
sh-3.2$ ls [^.]*
```

Motifs : exercices (2/2)

```
sh-3.2$ ls
```

Makefile	demoselect.c	intro.tex	td.aux	td.pdf
com??.tex	ent2_nonbloc.c	logo-n7.eps	td.haux	td.tex
corexo1.tex	exoprocl.a.tex	logo-n7.jpg	td.htoc	
corexo2.tex	gliens.tex	mybook.hva	td.image.tex	
demofcntl.c	html	proc.tex	td.log	

Lister les fichiers cachés du répertoire courant

```
sh-3.2$
```

Motifs : exercices (2/2)

```
sh-3.2$ ls
```

Makefile	demoselect.c	intro.tex	td.aux	td.pdf
com??.tex	ent2_nonbloc.c	logo-n7.eps	td.haux	td.tex
corexo1.tex	exoprocl1a.tex	logo-n7.jpg	td.htoc	
corexo2.tex	gliens.tex	mybook.hva	td.image.tex	
demofcntl.c	html	proc.tex	td.log	

Lister les fichiers cachés du répertoire courant

```
sh-3.2$ ls -d .*
```

```
..
```

```
sh-3.2$ ls *\?
```

```
..
```

Motifs : exercices (2/2)

```
sh-3.2$ ls
Makefile      demoselect.c  intro.tex     td.aux        td.pdf
com??.tex     ent2_nonbloc.c logo-n7.eps   td.haux       td.tex
corexo1.tex   exoprocla.tex logo-n7.jpg   td.htoc
corexo2.tex   gliens.tex    mybook.hva   td.image.tex
demofcntl.c   html          proc.tex     td.log
```

Lister les fichiers cachés du répertoire courant

```
sh-3.2$ ls -d .*
.
.
sh-3.2$ ls *\?
ls: *?: No such file or directory
sh-3.2$
```

Motifs : exercices (2/2)

```
sh-3.2$ ls
Makefile      demoselect.c  intro.tex     td.aux        td.pdf
com??.tex     ent2_nonbloc.c logo-n7.eps   td.haux       td.tex
corexo1.tex   exoprocl1a.tex logo-n7.jpg   td.htoc
corexo2.tex   gliens.tex    mybook.hva   td.image.tex
demofcntl.c   html          proc.tex     td.log
```

Lister les fichiers cachés du répertoire courant

```
sh-3.2$ ls -d .*
.
..
.xwkr
sh-3.2$ ls *\?
ls: *?: No such file or directory
sh-3.2$ ls *\?*
com??.tex
sh-3.2$ ls ??[^.]*
```

Interprétation des lignes de commandes saisies

Motifs : exercices (2/2)

```
sh-3.2$ ls
Makefile      demoselect.c  intro.tex     td.aux        td.pdf
com???.tex    ent2_nonbloc.c logo-n7.eps   td.haux       td.tex
corexo1.tex   exoprocl1a.tex logo-n7.jpg   td.htoc
corexo2.tex   gliens.tex    mybook.hva    td.image.tex
demofcntl.c   html          proc.tex      td.log
```

Lister les fichiers cachés du répertoire courant

```
sh-3.2$ ls -d .*
.
..
.xwkr
sh-3.2$ ls *\?
ls: *?: No such file or directory
sh-3.2$ ls *\?*
com???.tex
sh-3.2$ ls??[^.]*
Makefile      corexo2.tex   ent2_nonbloc.c  html          logo-n7.jpg
com???.tex    demofcntl.c  exoprocl1a.tex  intro.tex     mybook.hva
corexo1.tex   demoselect.c gliens.tex      logo-n7.eps   proc.tex
```

Motifs : exercices (2/2)

```
sh-3.2$ ls
Makefile      demoselect.c  intro.tex     td.aux        td.pdf
com???.tex    ent2_nonbloc.c logo-n7.eps   td.haux       td.tex
corexo1.tex   exoprocl1a.tex logo-n7.jpg   td.htoc
corexo2.tex   gliens.tex    mybook.hva   td.image.tex
demofcntl.c   html          proc.tex     td.log
```

Lister les fichiers cachés du répertoire courant

```
sh-3.2$ ls -d .*
.
..
.xwkr
```

```
sh-3.2$ ls *\?
ls: *: No such file or directory
```

```
sh-3.2$ ls *\?*
com???.tex
```

```
sh-3.2$ ls ??[^.]*
```

```
Makefile      corexo2.tex    ent2_nonbloc.c  html          logo-n7.jpg
com???.tex    demofcntl.c   exoprocl1a.tex  intro.tex     mybook.hva
corexo1.tex    demoselect.c  gliens.tex      logo-n7.eps   proc.tex
```

Lister les fichiers dont le suffixe comporte un 'e'

Interprétation des lignes de commandes saisies

Motifs : exercices (2/2)

```
sh-3.2$ ls
Makefile      demoselect.c  intro.tex     td.aux        td.pdf
com??.tex     ent2_nonbloc.c logo-n7.eps   td.haux       td.tex
corexo1.tex   exoprocl1a.tex logo-n7.jpg   td.htoc
corexo2.tex   gliens.tex    mybook.hva   td.image.tex
demofcntl.c   html         proc.tex     td.log
```

Lister les fichiers cachés du répertoire courant

```
sh-3.2$ ls -d .*
.
..
.xwkr
```

```
sh-3.2$ ls *\?
ls: *?: No such file or directory
```

```
sh-3.2$ ls *\?*
com??.tex
```

```
sh-3.2$ ls ??[^.]*
```

```
Makefile      corexo2.tex   ent2_nonbloc.c  html          logo-n7.jpg
com??.tex     demofcntl.c  exoprocl1a.tex  intro.tex     mybook.hva
corexo1.tex   demoselect.c gliens.tex      logo-n7.eps   proc.tex
```

Lister les fichiers dont le suffixe comporte un 'e'

```
sh-3.2$ ls *.*e*
```

Étapes de l'interprétation)

Le shell interprète chaque ligne de commande en plusieurs étapes :

- 1 Identifier les opérateurs de composition de commandes : `() ; & | && ||`
→ décomposition de la ligne en **commandes élémentaires**
- 2 Exécuter les commandes entre `` `` (**antiquotes**), (sauf à l'intérieur de `' '` (quotes)), puis substitution par le résultat (sortie) de l'exécution
- 3 Remplacer les noms de **variables** précédés du caractère `$` par leur valeur (sauf à l'intérieur des `"` (simples quotes)).
- 4 Découper la ligne en **arguments** (mots séparés par des espaces)
 - arguments explicitement nuls (chaînes de la forme `" "` ou `" "`) conservés
 - arguments implicitement nuls (variables vides/non définies) éliminés
- 5 Mise en place des **redirections**
- 6 Evaluer les **métacaractères** / motifs de chemins
Les chaînes comprises entre `" "` et `" "` (doubles et simples quotes) ne sont pas interprétées
- 7 Elimination des `" "` et `" "` (doubles et simples quotes) extérieures

Contrôle de l'interprétation

- \ permet de neutraliser un caractère (métacaractère, motif..) → * sera interprété comme le caractère *, non comme un motif.
- une chaîne entre quotes ('chaîne') n'est jamais interprétée
- dans une chaîne entre double quotes ("chaîne") , seules les variables sont substituées
- une chaîne entre antiquotes (`chaîne`) est considérée comme une commande. L'interprétation consiste à exécuter cette commande puis remplacer le texte de la commande (chaîne) par ce qu'elle produit sur sa sortie standard.

Contrôle de l'interprétation : exemple

```
sh-3.2$ ls
com??.tex    corexo2.tex  html          logo n7.eps  mybook.hva   td.aux
corexo1.tex  demo fcntl.c intro.tex     logo-n7.jpg  proc.tex     td.pdf
sh-3.2$ x=`echo *\ *.c`
sh-3.2$ ls $x
```

Contrôle de l'interprétation : exemple

```
sh-3.2$ ls
com??.tex      corexo2.tex    html          logo n7.eps   mybook.hva    td.aux
corexo1.tex    demo fcntl.c  intro.tex     logo-n7.jpg   proc.tex      td.pdf
sh-3.2$ x=`echo *\ *.c`
sh-3.2$ ls $x
ls: demo: No such file or directory
ls: fcntl.c: No such file or directory
sh-3.2$ ls "$x"
```

Contrôle de l'interprétation : exemple

```
sh-3.2$ ls
com??.tex      corexo2.tex    html          logo n7.eps   mybook.hva    td.aux
corexo1.tex    demo fcntl.c  intro.tex     logo-n7.jpg   proc.tex      td.pdf
sh-3.2$ x=`echo *\ *.c`
sh-3.2$ ls $x
ls: demo: No such file or directory
ls: fcntl.c: No such file or directory
sh-3.2$ ls "$x"
demo fcntl.c
sh-3.2$ ls `echo *\ *.c`
```

Contrôle de l'interprétation : exemple

```
sh-3.2$ ls
com??.tex    corexo2.tex  html         logo n7.eps  mybook.hva  td.aux
corexo1.tex  demo fcntl.c  intro.tex    logo-n7.jpg  proc.tex    td.pdf
sh-3.2$ x=`echo *\ *.c`
sh-3.2$ ls $x
ls: demo: No such file or directory
ls: fcntl.c: No such file or directory
sh-3.2$ ls "$x"
demo fcntl.c
sh-3.2$ ls `echo *\ *.c`
ls: demo: No such file or directory
ls: fcntl.c: No such file or directory
sh-3.2$ ls `echo *.tex`
```

Contrôle de l'interprétation : exemple

```
sh-3.2$ ls
com??.tex      corexo2.tex    html           logo n7.eps    mybook.hva     td.aux
corexo1.tex    demo fcntl.c  intro.tex      logo-n7.jpg    proc.tex       td.pdf
sh-3.2$ x=`echo *\ *.c`
sh-3.2$ ls $x
ls: demo: No such file or directory
ls: fcntl.c: No such file or directory
sh-3.2$ ls "$x"
demo fcntl.c
sh-3.2$ ls `echo *\ *.c`
ls: demo: No such file or directory
ls: fcntl.c: No such file or directory
sh-3.2$ ls `echo *.tex`
com??.tex      corexo1.tex    corexo2.tex    intro.tex      proc.tex
```


Plan

- ① Rôle d'un interpréteur de commandes (shell)
- ② Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- ③ Programmation dans le large
 - Composition de commandes
 - Scripts
- ④ Commandes avancées

Plan

- ❶ Rôle d'un interpréteur de commandes (shell)
- ❷ Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- ❸ Programmation dans le large
 - Composition de commandes
 - Scripts
- ❹ Commandes avancées

Opérateurs de composition

; (séquence) `C1;C2;C3` spécifie que les `C1`, `C2` et `C3` devront s'exécuter l'une après l' autre

|| (séquence ou) `C1||C2||C3` exécute en séquence `C1`, `C2`, `C3` jusqu'à ce qu'une commande ait un retour normal

&& (séquence et) `C1&&C2&&C3..` exécute en séquence `C1`, `C2`, `C3` jusqu'à ce qu'une commande ait un retour anormal

(...) (exécution par un sous-processus shell)

& (lancement en arrière-plan)

| (couplage par tubes) `C1|C2` lance en parallèle `C1` et `C2` et crée un tube (pipe), reliant la sortie standard de `C1` à l'entrée standard de `C2` (cf infra)

Architecture en pipeline

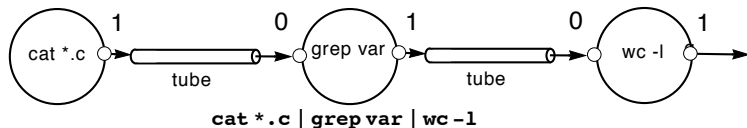
Le shell permet de construire un schéma de traitement parallèle particulier, appelé pipeline :

- un traitement comportant plusieurs étapes successives E1, E2, E3... doit être appliqué à chacun des éléments d'un ensemble
- le principe est de lancer autant de processus P1, P2, P3... que d'étapes, chaque processus réalisant l'une des étapes
- le traitement est alors organisé ainsi
 - P1 commence en traitant successivement chacun des éléments de l'ensemble. Dès qu'un élément a été traité, le résultat est transmis à P2, pour qu'il réalise la deuxième étape
 - P2 fait de même vis-à-vis de P3, au fur et à mesure qu'il reçoit les résultats de P1, etc...
 - Une fois le pipeline amorcé, tous les processus tournent en parallèle

Tubes et filtres

Le shell permet de réaliser cette architecture avec

- une catégorie particulière de commandes, les **filtres**, qui sont écrites pour lire leurs données d'entrée, par défaut, sur leur entrée standard, et écrire leurs résultats sur leur sortie standard
- les **tubes** qui permettent de rediriger la sortie standard d'une commande vers l'entrée standard d'une autre commande



Filtres

Quelques filtres réalisant les opérations d'un SGBD relationnel

`grep` **sélection** : filtre les lignes contenant un motif donné

`cut` **projection** : extraction de colonnes d'un fichier

`sort` **tri** des lignes d'un fichier sur un champ

`join` **jointure** : jointure de fichiers préalablement triés (par `sort`)
sur le champ de jointure

`cat` **union** : concaténation de fichiers ;

`sort` (ou `uniq`) permet d'éliminer les doublons

`comm` **intersection** et **différence** des lignes de deux fichiers

`paste` **juxtaposition** des lignes de fichiers (cat horizontal)

`head` extraction des premières lignes d'un fichier

`tail` extraction des dernières lignes d'un fichier

Autres filtres

`wc` statistiques sur le contenu d'un fichier

`sed` édition orientée ligne d'un fichier

`tr` conversion/suppression des caractères d'un fichier

`diff` comparaison du contenu de deux fichiers

...

Plan

- ❶ Rôle d'un interpréteur de commandes (shell)
- ❷ Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- ❸ Programmation dans le large
 - Composition de commandes
 - Scripts
- ❹ Commandes avancées

Principe, rôle des scripts

Idée

Automatiser des enchaînements complexes ou répétitifs de commandes shell

→

Définir un langage de programmation (variables, structures de contrôle) permettant de spécifier ces enchaînements

- Éléments de ce langage (structures de contrôle, etc)
= **commandes internes** de l'interpréteur de commandes
- Ces programmes sont appelés des **scripts**
- Le shell lit, interprète et exécute les fichiers de script ligne par ligne, de la même manière qu'il traite les lignes provenant de son entrée standard

Définition et utilisation des scripts

Exemple : le fichier `listerRepPriv` contient les lignes

```
# commentaire :  
# ce script liste les fichiers du répertoire privé  
cd  
pwd  
ls
```

Pour exécuter le contenu de `listerRepPriv` :

```
sh listerRepPriv
```

ou bien :

```
chmod +x listerRepPriv
```

```
# listerRepPriv devient exécutable
```

```
listerRepPriv
```

```
# listerRepPriv est appelé comme une commande
```

Résultat d'une commande/d'un script

Lorsqu'un traitement s'achève, un code retour entier permet de fournir une indication sur le résultat de ce traitement.

- Par convention, ce code retour est nul cas de terminaison normale, non nul sinon.
- En C, ce code retour est la valeur de retour de la fonction `main`; il vaut souvent -1 en cas d'erreur (+ détails → variable `errno`)
- En shell, le code retour (entre 0 et 255) est accessible via la variable prédéfinie `$?` :

```
sh-3.2$ ls
corexo1.tex      exoprocl1a.tex  logo-n7.jpg      td.htoc
sh-3.2$ rm corexo1.txt
rm: corexo1.txt: No such file or directory
sh-3.2$ echo $?
1
sh-3.2$ rm corexo1.txt
sh-3.2$ echo $?
0
```

Dans le cas d'un script, la commande **exit** permet de terminer le script en fixant une valeur pour le code retour



Scripts

Paramètres d'un script

- A l'instar des commandes, un script peut accepter des paramètres d'appel : liste des mots (séparés par des espaces) suivant le chemin d'accès au script.
- Au sein du script, les paramètres d'un script/d'une commande sont désignés par leur position (de 1 à 9) ; la commande est désignée par le chiffre 0.
 - par exemple, \$4 désigne la valeur du 4ème paramètre
 - **shift** (commande de décalage à gauche des paramètres), permet d'accéder aux paramètres au delà du 9ème
 - la variable prédéfinie ***** désigne l'**ensemble des paramètres** d'appel
 - la variable prédéfinie **#** désigne le **nombre de paramètres** d'appel

Autres variables prédéfinies du (Bourne) shell

\$ numéro de processus courant

? code retour de la dernière commande exécutée.



Opérations sur les paramètres

set *arg1 arg2...* définit *arg1 arg2 ...* comme ensemble des paramètres

shift décale \$2 \$3 vers la gauche, en écrasant \$1.

S'il y a plus de 9 paramètres, le 10ème devient accessible via \$9.

Exemple : fichier essai

```
#!/bin/sh
echo appel : "$0" "$*"
set par1 par2
echo set : "$0" "$*"
shift
echo shift : "$0" "$*
```

```
sh-3.2$ ./essai a b c d e
appel : ./essai a b c d e
set : ./essai par1 par2
shift : ./essai par2
```

Structures de contrôle (1/6) : commande **test**

Syntaxe

test expression ou [expression]

- Code de retour nul = test vrai
- Les composants de **expression** doivent être séparés par des **espaces**

Expressions

- Tests élémentaires

-r/w/x <fichier> <fichier> peut être lu/écrit/exécuté

-f <chemin> <chemin> est un fichier

-d <chemin> <chemin> est un répertoire

-z <chaîne> <chaîne> est vide

-n <chaîne> <chaîne> n'est pas vide

- Comparaisons

- Chaînes de caractères : =, !=

- Nombres : -eq, -ne, -gt...

- Composition

! (non), -a (et), -o (ou), \ (, \)

Scripts

Structures de contrôle (2/6) : **for***Syntaxe*

```
for variable in liste_de_valeurs
do
    liste de commandes
done
```

Exemples

```
for i in 1 2 3
do
    echo bonjour
    echo $i
done
```

```
for i in $*
do
    cd $rep; pwd; ls
done
```

Structures de contrôle (2/6) : **for**

Syntaxe

```
for variable in liste_de_valeurs
do
    liste de commandes
done
```

Exemples

```
for i in 1 2 3
do
    echo bonjour
    echo $i
done
```

```
for i in $*
do
    cd $rep; pwd; ls
done
```

Structures de contrôle (3/6) : **while**

Syntaxe

```
while liste1_de_commandes  
do  
    liste2_de_commandes  
done Exécute liste2 tant que le résultat de liste1 est normal
```

Exemple

```
while test -n $1  
do  
    echo $1  
    shift  
done
```


Structures de contrôle (4/6) : if

Syntaxe

```
if liste1_de_commandes
  then liste2_de_commandes
  else liste3_de_commandes
fi
```

Si le résultat de `liste1` est normal, `liste2` exécutée, sinon `liste3`

Exemple

```
if rm $*
  then echo détruit
  else echo non trouve
fi
```

Remarques

- La partie `else` est facultative
- En général `liste1` est réduite à une commande `test`

Structures de contrôle (5/6) : case

Syntaxe

```
case variable in
  modèle1) liste1_de_commandes ;;
  modèle2) liste2_de_commandes ;;
  ...
esac
```

Exemple

```
case $2 in
  term) pr $1|more ;;
  impr) pr $1|lpr ;;
  fich) pr $1 > $3 ;;
  *) echo "erreur de parametre" ;;
esac
```

Remarques

- * \approx else/otherwise
- appel : `lister proj.1 impr`

Scripts

E/S : lecture de caractères sur l'entrée standard (6/6)

Syntaxe`read` x

Attend la saisie d'une ligne sur l'entrée standard et l'affecte à x

Plan

- ① Rôle d'un interpréteur de commandes (shell)
- ② Interpréteur de commandes interactif : principe et utilisation
 - Rôle et principe du shell
 - Facilités d'utilisation
 - Interprétation des lignes de commandes saisies
- ③ Programmation dans le large
 - Composition de commandes
 - Scripts
- ④ Commandes avancées

Préambule

Syntaxe

man sujet

- affiche la **documentation** en ligne relative au sujet (commande, notion...) fourni en paramètre
- options utiles : -f (whatis), -k (apropos), -t (formatage)

Exemples

man man

man ls

man hier

man intro

find (rappel)

Syntaxe

find répertoire expression

- **Parcourt récursivement** le répertoire fourni en paramètre, **et évalue** expression sur chacun des fichiers rencontrés.
- L'évaluation de l'expression, trouvée vraie sur un fichier F , peut amener l'exécution de commandes portant sur F

Exemple

```
sh-3.2$ find . -name core -print -exec rm {} \;  
/core  
/LANGAGES/C/TP1/core  
/LANGAGES/C/TP2/core
```

- recherche les fichiers core à partir du répertoire courant (.),
- affiche le chemin d'accès aux fichiers trouvés (-print), et
- exécute la commande rm (-exec) sur chacun de ces fichiers.

Ici, 3 fichiers core ont été trouvés et supprimés..



Expressions régulières

Les expressions régulières

Langage permettant de définir des motifs à distinguer dans un texte.

Ce langage est commun à plusieurs commandes utiles : `grep`, `sed`, `expr`...

Motifs élémentaires

- `.` désigne un caractère quelconque
- `^` et `$`, contraignent respectivement le motif à être situé en début et en fin de ligne.
- Définition d'ensembles :
 - ◊ une liste de caractères entre crochets (`[` et `]`) désigne un élément quelconque de la liste
 - ◊ `-` permet de définir des intervalles de caractères au sein d'un ensemble :
`[f-j]` désigne tous les caractères compris entre `f` et `j`, soit : `{f,g,h,i,j}`
 - ◊ `^` immédiatement après le `[` signifie qu'il faut considérer le complémentaire de l'ensemble
`[^zf-j5]` désigne tous les caractères *sauf* `{f,g,h,i,j,z,5}`

Facteur de répétition

- doit suivre immédiatement le motif dont il précise la répétition.
- `*` signifie que le motif qui précède est répété un nombre de fois quelconque (0 compris).
- `\{n,m\}`, (`n` et `m` entiers) signifie que le motif qui précède est répété entre `n` et `m` fois
- `\{n,\}`, (`n` entier) signifie que le motif qui précède est répété au moins `n` fois
- `\{n\}`, (`n` entier) signifie que le motif qui précède est répété exactement `n` fois

Exemple

`^[^:]*:a[4-8fk-m1][^:;]*[0-9]$`



expr (1/2)

Syntaxe

expr expression

évalue des expressions (entiers/chaîne) et affiche (stdout) le résultat.

Expressions

- expression1 **opérateur** expression2
 - si **opérateur** est =, !=, >, >=, < ou <=
renvoie le résultat de la comparaison entre entiers si les deux expressions sont des entiers, sinon renvoie le résultat de la comparaison lexicale (vrai : 1 ou faux : 0)
 - si **opérateur** est +, -, *, / ou %
opération arithmétique sur les expressions (entiers)
- chaîne : expression_régulière
recherche la plus longue sous-chaîne (sc) commençant en début de chaîne et correspondant à expression_régulière
 - si expression_régulière contient un sous-motif entre parenthèses (\(et \)), affiche la **sous-chaîne de sc appariée** au sous-motif
 - sinon, expr affiche la **longueur** de sc

expr (2/2)

Composition

- `expression1 | expression2` renvoie le résultat de `expression1` si celui-ci n'est ni vide, ni zéro, sinon renvoie le résultat de `expression2`
- `expression1 & expression2` renvoie le résultat de `expression1` si aucun résultat n'est vide / nul, sinon renvoie zéro
- les composants de l'expression peuvent être parenthésés par `\(` et `\)`
- **tous** les composants doivent être séparés par des **espaces**.

Exercices avec la commande `expr`

- calculer la longueur d'une chaîne donnée
- extraire la sous chaîne de taille `T`, commençant à la position `P`
- dans une chaîne donnée, indiquer la position de la première occurrence d'un caractère d'une liste donnée

expr (2/2)

Composition

- `expression1 | expression2` renvoie le résultat de `expression1` si celui-ci n'est ni vide, ni zéro, sinon renvoie le résultat de `expression2`
- `expression1 & expression2` renvoie le résultat de `expression1` si aucun résultat n'est vide / nul, sinon renvoie zéro
- les composants de l'expression peuvent être parenthésés par `\(` et `\)`
- **tous** les composants doivent être séparés par des **espaces**.

Exercices avec la commande `expr`

- calculer la longueur d'une chaîne donnée
`expr $chaîne : '.*'`
- extraire la sous chaîne de taille T, commençant à la position P
- dans une chaîne donnée, indiquer la position de la première occurrence d'un caractère d'une liste donnée



expr (2/2)

Composition

- `expression1 | expression2` renvoie le résultat de `expression1` si celui-ci n'est ni vide, ni zéro, sinon renvoie le résultat de `expression2`
- `expression1 & expression2` renvoie le résultat de `expression1` si aucun résultat n'est vide / nul, sinon renvoie zéro
- les composants de l'expression peuvent être parenthésés par `\(` et `\)`
- **tous** les composants doivent être séparés par des **espaces**.

Exercices avec la commande `expr`

- calculer la longueur d'une chaîne donnée
`expr $chaîne : '.*'`
- extraire la sous chaîne de taille T, commençant à la position P
`expr $chaîne : ".$P\}\(.\{$T\}\)"`
- dans une chaîne donnée, indiquer la position de la première occurrence d'un caractère d'une liste donnée

expr (2/2)

Composition

- `expression1 | expression2` renvoie le résultat de `expression1` si celui-ci n'est ni vide, ni zéro, sinon renvoie le résultat de `expression2`
- `expression1 & expression2` renvoie le résultat de `expression1` si aucun résultat n'est vide / nul, sinon renvoie zéro
- les composants de l'expression peuvent être parenthésés par `\(` et `\)`
- **tous** les composants doivent être séparés par des **espaces**.

Exercices avec la commande `expr`

- calculer la longueur d'une chaîne donnée
`expr $chaîne : '.*'`
- extraire la sous chaîne de taille T, commençant à la position P
`expr $chaîne : ".$P\}\(.\{$T\}\)"`
- dans une chaîne donnée, indiquer la position de la première occurrence d'un caractère d'une liste donnée
`expr `expr $chaîne : '[^liste_car]*' ` + 1`

xargs

Syntaxe

`xargs commande`

exécute `commande`, en fournissant comme paramètres à `commande2` les données arrivant sur l'entrée standard

Schéma d'usage

`commande1 | xargs commande2`

`commande1` produit sur sa sortie standard les paramètres utilisés par `commande2`

Exemple

Afficher le nombre de lignes de chacun des fichiers de suffixe `.tex` contenus dans le répertoire courant

xargs

Syntaxe

`xargs commande`

exécute `commande`, en fournissant comme paramètres à `commande2` les données arrivant sur l'entrée standard

Schéma d'usage

`commande1 | xargs commande2`

`commande1` produit sur sa sortie standard les paramètres utilisés par `commande2`

Exemple

Afficher le nombre de lignes de chacun des fichiers de suffixe `.tex` contenus dans le répertoire courant

```
sh-3.2$ ls -l *.tex | xargs wc -l
```