Partie No 1

Définition

Historique

Boucle d'interprétation, commandes simples: forme générale

Erreurs classiques et aide en ligne

Systèmes de gestion de fichiers (SGF)

Fichiers, inodes, dossiers, arborescence

droit d'accès

commandes sur les fichiers/dossiers

Définition

SHELL: programme en mode texte assurant l'interface entre l'utilisateur et le système unix.

S'utilise :

- En interactif depuis une fenêtre terminal (xterm, connexion distante texte, ...) : interpréteur de commande
- Pour réaliser des scripts (fichiers de commandes) : langage de programmation

Shell: où que je clique ?

On ne clique pas : ça s'utilise avec une souris à 105 touches et sans boule : un clavier :-)

L'accès est moins immédiat que celui d'une interface graphique

Plus de liberté/possibilités qu'avec une interface graphique

Langage de programmation: possibilité d'exprimer des requètes complexes

Utilisation interactive ou pour écrire des fichiers de commandes (scripts)

Historique

Les deux shells des origines sont à l'origine de deux familles de shells aux syntaxes incompatibles :

- Le shell le plus ancien : sh ou Bourne shell écrit dans les années 70 par Steve Bourne. Tout système système unix a un shell /bin/sh qui est un bourne shell (ou un shell compatible);
- Le csh: écrit à la même époque par Bill Joy incompatible avec le bourne shell mais offrant quelques fonctionnalités supplémentaires (historique des commandes, aliases, contrôle de tâches, ...

Historique (2)

Ksh: korn shell (David Korn, 1983) sur la base du bourne sh. Le ksh 88 (ou +) est livré avec tous les unix commerciaux. Base de la norme IEEE Posix 1003.2;

Tcsh: un shell évolué de la famille csh utilisé dans les années 90 comme shell interactif;

Bash: Bourne Again sh, le shell de la FSF. Compatible posix 1003.2. Le shell de base des distribution linux.

Zsh: un shell riche en fonctionnalités. Probablement le meilleur choix actuel en interactif.

Historique (3)

POSIX:

SUS: Single Unix Specification: spécification suivie par les unix commerciaux (et de nombreux non commerciaux) modernes. Proche de la norme POSIX.

se limiter à SUSv3/POSIX garantit une compatibilité maximale avec les unix utilisés de nos jours

SUS: http://www.unix.org/what_is_unix/single_unix_specification.html

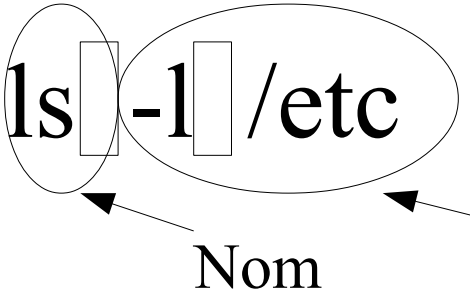
De nos jour, il est conseillé d'utiliser un shell compatible posix/sus: ksh, bash et zsh.

Boucle d'interprétation

Le shell est un programme qui réalise la boucle suivante :

- Boucle :
  - Lire la ligne de commande
  - Décoder la ligne de commande
  - Exécution de la ligne de commande en créant un processus dans le cas de commandes externes
  - Attendre la fin de l'exécution du processus
  - Retourner en début de boucle

commandes simples: forme générale



arguments:

paramètres optionnels permettant de modifier le comportement de la commande

liste des entités auxquelles doit s'appliquer la commande (nom de fichier, processus, utilisateur, ...)

Exemples:

mozilla

mozilla -P toto www.univ-evry.fr

ls -lrt /etc

find ~ -name *.avi -exec rm -f {} \;

quelques commandes simples

who: liste des utilisateurs ayant une session en cours sur l'ordinateur

w: idem mais indique aussi ce qu'ils font

date: date courante

echo: affiche ses arguments séparés par une espace

Exemples

Le shell va servir à lancer des commandes internes ou externes

Exemple de session :

```
#un commentaire commence par #
# lister les fichiers présents
# dans le dossier /etc
ls -l /etc
# liste des utilisateurs connectés
# sur l'ordinateur
who
```

Erreurs

5 causes classiques d'erreurs

- syntaxe ou chemin incorrect (commande inconnue, ...)
- paramètres incorrects (fichier inconnu, ..)
- droits d'accès : permission refusée (accès à un fichier, ...)
- options invalides (syntaxe des options de la commande)
- erreur de conception : le comportement n'est pas celui attendu

A l'aide

le manuel

option « --help » de certaines commandes

la documentation de votre système d'exploitation ou du programme posant problème

- souvent /usr/share/doc, /usr/local/share/doc

recherche sur le WeB: quelqu'un d'autre a forcément déjà eu ce problème

Le manuel

dans une version ultérieure de ce document

les sections du manuel: ràf

exemples d'utilisation: ràf + exemple dans 2 sections

Systèmes de gestion de fichiers (SGF)

SGF: mode d'organisation et de stockage des données sur disque;

Exemples: FAT32, NTFS, ext2fs, ext3fs, reiserfs, UFS, ...

Les SGF ont des propriétés et fournissent des services variés

Exemple:

- les SGF Unix (ext2fs, UFS, ...) : droits sur les fichiers.
- FAT32: pas de droits d'accès aux fichiers

SGF (suite)

Les SGF unix fournissent un sous-ensemble commun de fonctionnalités: celui dont nous parlerons.

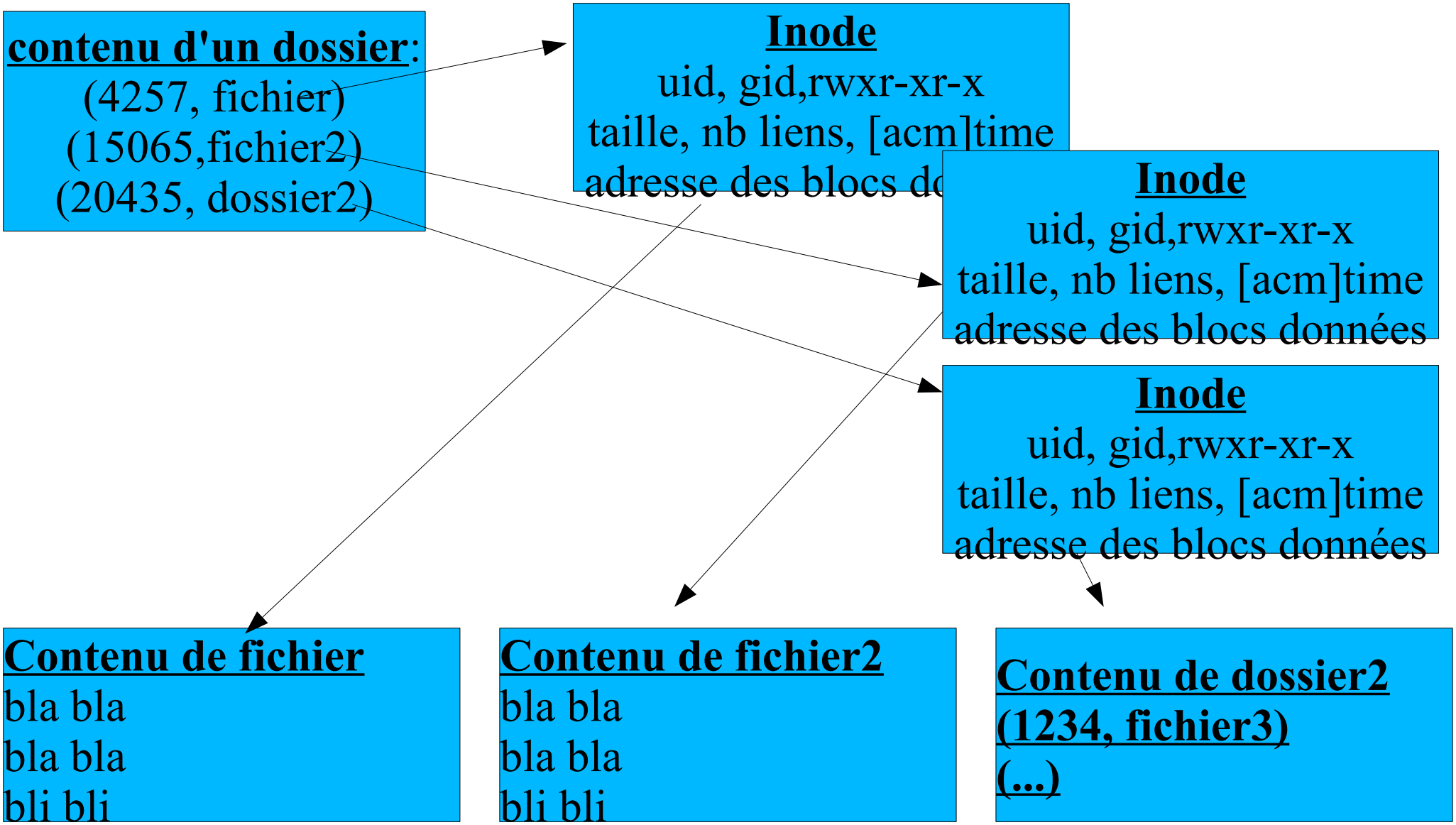
Chaque SGF peut fournir plus que ce sous-ensemble

Fichier unix: fichier disque mais aussi ressource du système (pilote de périphérique, terminal, mémoire, ...)

/dev/hda1 : partition 1 du disque 1 (Linux)

/dev/kmem: mémoire virtuelle du noyau

Fichiers


contenu d'un dossier:
(4257, fichier)
(15065,fichier2)
(20435, dossier2)
Inode
uid, gid,rwxr-xr-x
taille, nb liens, [acm]time
adresse des blocs do
Inode
uid, gid,rwxr-xr-x
taille, nb liens, [acm]time
adresse des blocs données
Inode
uid, gid,rwxr-xr-x
taille, nb liens, [acm]time
adresse des blocs données
Contenu de fichier
bla bla
bla bla
bli bli
Contenu de fichier2
bla bla
bla bla
bli bli
Contenu de dossier2
(1234, fichier3)
(...)

SGF : inode

Inode: attributs + localisation des blocs contenant les données du fichier

Inode:

- Id. du disque logique où est le fichier,
  numéro du fichier sur ce disque
- Type de fichier et droits d'accès
- Nombre de liens physiques
- Propriétaire, groupe propriétaire
- Taille
- Dates :
  - De dernier accès (y compris en lecture): atime
  - Date de dernière modification des données: mtime
  - Date de dernier modification de l'inode: ctime

Dossier/répertoire

Deux grandes classes de fichiers :

- Fichier ayant un contenu sur disque : fichiers réguliers, dossiers, liens symboliques, tubes
- Ressources : Fichiers spéciaux (pilotes de périphériques, ràf, ...)

Dossiers: listes de couples (nom, No inode)

Un couple est appelé « lien physique » (hardlink)

Du point de vue de l'utilisateur, un dossier contient des fichiers (fichiers réguliers, dossiers, ...).

Inodes/Nom: conséquences

Créer/détruire un fichier: ajouter/retirer un couple dans le dossier

opération nécessitant un droit au niveau du dossier pas du fichier

Le système travaille avec des No d'inode, l'utilisateur avec les noms de fichiers :

- Ce sont les dossiers qui permettent de faire le lien entre les deux
- On trouve le coupe (nom, inode) du dossier où est le fichier
- Pour trouver ce dossier, on applique le même principe (pour Unix, un dossier est aussi un fichier)
- voir plus loin Arborescences/algo de recherche

Fichiers: résumé:

ce que l'utilisateur perçoit comme un fichier identifié par un nom peut se décomposer en trois notions sous unix :

- un inode: informations (taille, dates, uid, gid, droits) et localisation des données sur disque
- le contenu du fichier: les données qui y sont stockées
- un lien physique: associe un nom à un inode. Un même inode peut avoir plusieurs lien.

Droits d'accès aux fichiers

3 types d'accès: lecture (R), écriture (W) et exécution (X)

Objet/Droit	R (lecture)	W (écriture)	X (exécuter)
fichier régulier	lire le contenu	modifier le fichier	exécuter le fichier
dossier	lister le contenu du dossier	modifier le contenu du dossier (y compris destruction de fichier)	utiliser le dossier dans un chemin ou s'y positionnner

3 classes d'utilisateurs: le propriétaire du fichier, le Groupe du propriétaire du fichier, les Autres utilisateurs.

type fichier	Propriétaire			Groupe du proprio			Autres utilisateurs		
-	R	W	X	R	-	X	R	-	X

informations dans l'inode, affichage avec « ls », changement avec chmod, chgrp et chown

Droits d'accès (2): suid, sgid, sticky bit

3 autres « droits » spéciaux:

- bit SUID: le programme s'exécute avec les droits de son propriétaire (au lieu de ceux de l'utilisateur qui le lance)
- bit SGID: le programme s'exécute avec les droits du groupe propriétaires du fichier
- sticky bit :
  - sur un fichier exécutable : (obsolète) maintient le fichier en mémoire après l'exécution pour diminuer le temps de la prochaine invocation
  - sur un dossier: seul le propriétaire du fichier a le droit de le supprimer. Exemple: /tmp/

Commandes de base: chmod

chmod [-R] mode fichier ...

-R: fichier est un dossier, chmod agit récursivement sur fichier et sur son contenu

mode:

- forme numérique: 644
  - pour u: 400 (r), 200 (w) et 100 (x)
  - pour g: 40 (r), 20 (w) et 10 (x)
  - pour o: 4 (r), 2 (w) et 1 (x)
- forme symbolique: [ugo][+-=][rwxXstguo]

chmod: exemples

chmod ràf

commande de base: umask

définit les droits d'accès par défaut d'un fichier

les droits sont le complément du paramètre d'umask: on laisse tout sauf les droits précisés

Exemple:

umask 002 : mode par défaut: RWXRWXR-X (tout sauf 002)

umask 026: mode par défaut: RWXR-X--X (tout sauf 026)

umask a=rx,gu+w: mode par défaut: RWXRWXR-X

umask -S : affiche le l'état courant sous forme symbolique : u=rwx,g=rwx,o=rw dans notre exemple.

Commandes de base: chown, chgrp

chown -R [-H | -L | -P] proprio[:groupe] fichier

chgrp -R [-H | -L | -P] groupe fichier ...

chown/chgrp: exemples

Commandes de base: ls

Commandes de base: cat

Commande de base: stat

Exemples

Stat fichier (noter ctime, mtime et atime)

Cat fichier

Stat fichier (atime a changé)

Chmod fichier

Stat fichier (ctime a changé)

Modif fichier

Stat fichier (mtime a changé)

Arborescence

Sous unix, on a une arborescence unique (donc pas de C:\, D:\, ...comme sous windows)

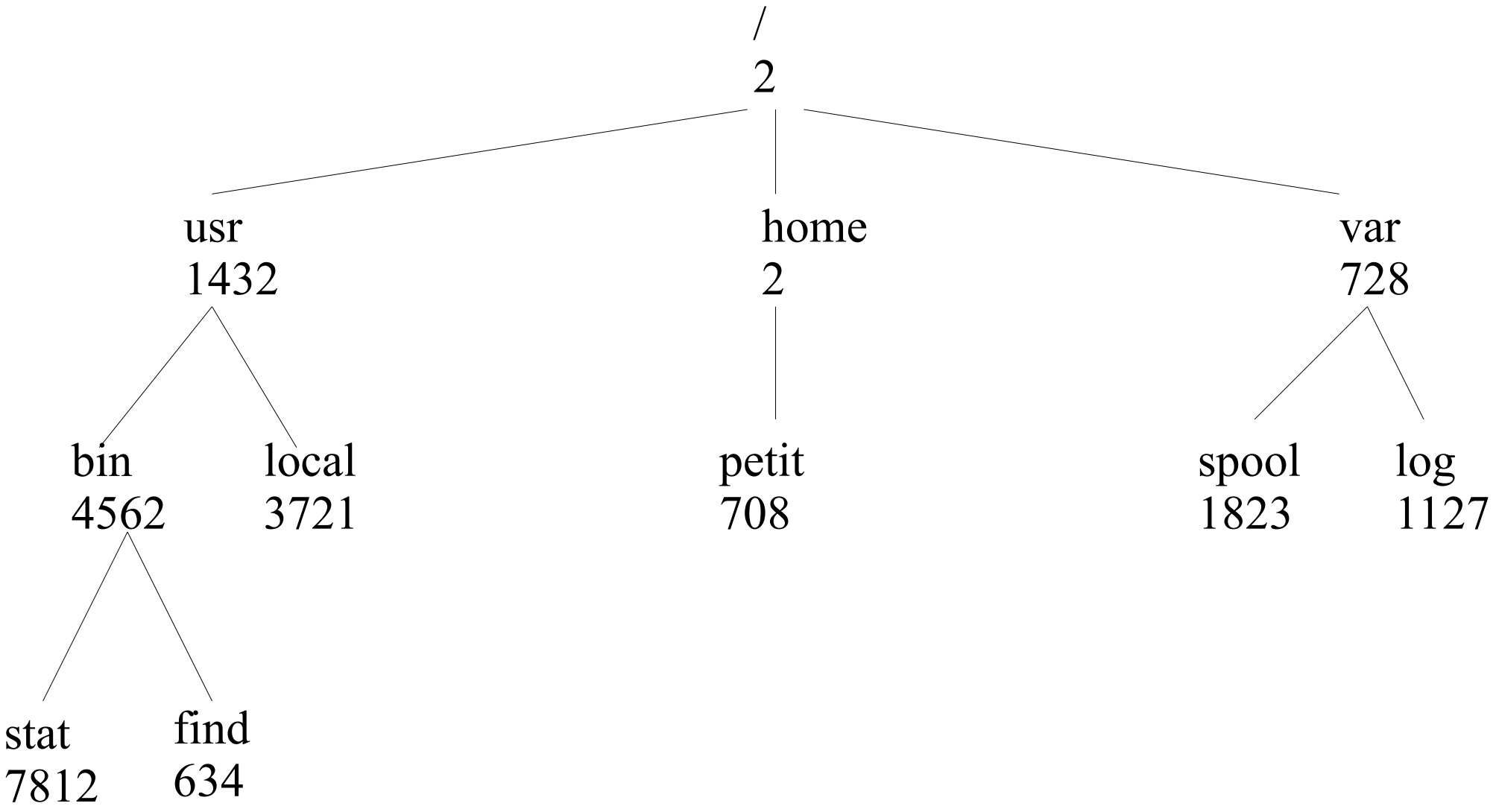
Le disque système contient la racine absolue /

toute l'arborescence est sous cette racine absolue

Les systèmes de fichiers des autres partitions s'intègrent dans l'arborescence en prenant la place d'un dossier existant

la racine d'un système de fichier a 2 comme numéro d'inode

arborescence



algo de recherche

/usr/bin/stat

algo de localication:

examiner le contenu du dossier d'inode 2 pour trouver le No d'inode du dossier usr : 1432 par exemple.

examiner le contenu de dossier d'inode 1432 pour trouver le No d'inode du dossier bin. 4562 par exemple

examiner le contenu de dossier d'inode 4562 pour trouver le No d'inode du fichier stat. 7812 par exemple

exécuter le fichier d'inode 7812

Chemin

/usr/bin/stat: chemin absolu du fichier stat

chemin absolu: chemin depuis la racine absolue

notion de dossier courant

chemin relatif: chemin depuis le dossier courant

Commandes de base:

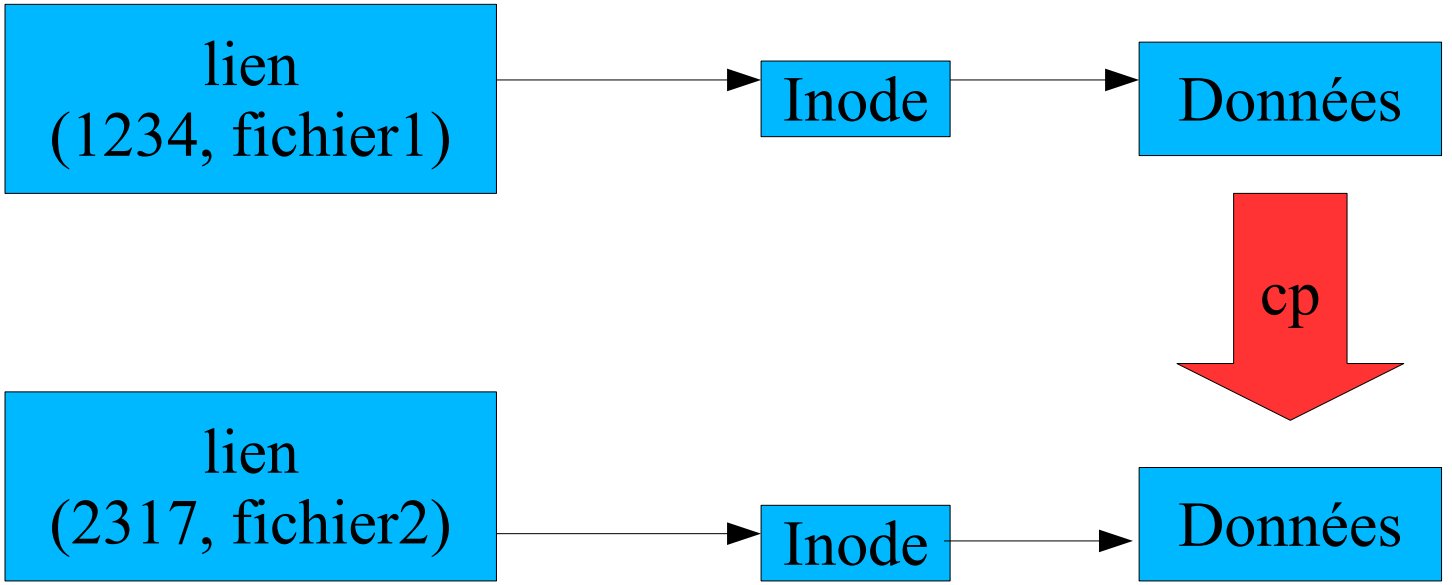
pwd : indique le dossier courant

cd : changer de dossier courant

mkdir: pour créer un dossier

rmdir: détruit les dossiers vides

commande de base: cp



cp fichier1 fichier2

Commandes de base: cp

- copie des données d'un fichier (source) dans un autre (cible) :
  - la cible n'existe pas : création d'un nouvel inode et recopie des données du fichier
  - la cible existe: inode destination inchangée, recopie des données du fichier dans la cible

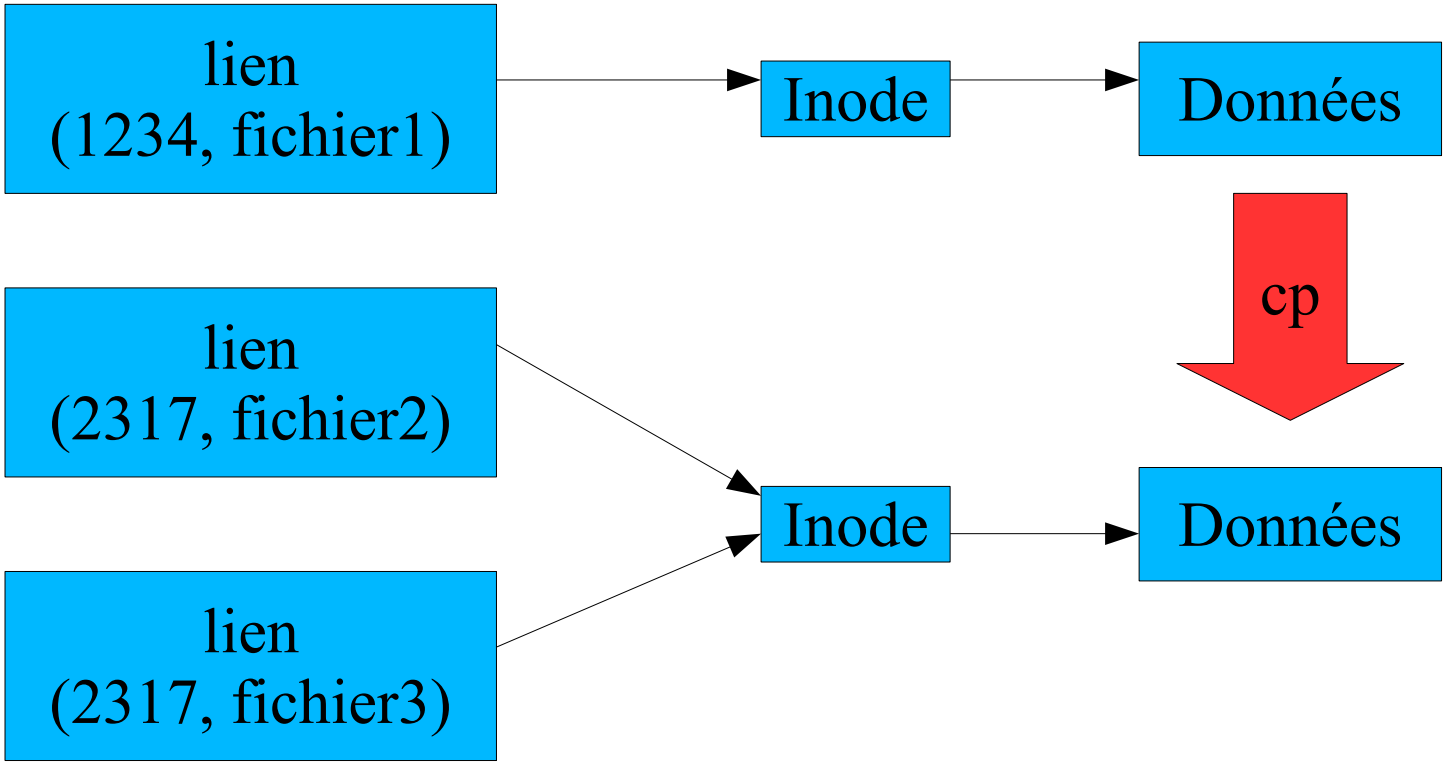
cp (2)

gnu cp: en standard sous Linux, installable facilement ailleurs

fournit des options non standard mais pratiques

ràf

cp et liens physiques



cp fichier1 fichier2

Commandes de base: rm

suppression d'un lien d'un fichier ou plusieurs fichiers: `rm [-fiRr] fichier1 ...`

options:

- -i: demande de confirmation pour tout fichier à supprimer (aff sur stderr et lecture sur stdin)
- -f: supprime les messages d'erreur lorsqu'un fichier n'existe pas et supprime la demande d'acquittement si l'utilisateur de rm n'a pas les droits d'écriture sur le fichier à supprimer
- -R ou -r: supprime récursivement le contenu d'un dossier avant d'appliquer rmdir au dossier.

rm :exemples

Commandes de base: mv

mv [-fi] source destination

renomme un lien. Source et fichier sont des fichiers réguliers

mv [-fi] source1 ... destination

renomme les sources en les déplaçant **dans** le dossier destination

la commande fonctionne aussi si sources et destinations sont dans des systèmes de fichiers différents.

la seconde forme est utilisée si la destination est un dossier existant

mv: exemples

mv [-fi] source destination

renomme un lien. Source et fichier sont des fichiers réguliers

mv [-fi] source1 ... destination

renomme les sources en les déplaçant **dans** le dossier destination

la commande fonctionne aussi si sources et destinations sont dans des systèmes de fichiers différents.

la seconde forme est utilisée si la destination est un dossier existant

Commandes de base: ln

ln fichier nouveau_lien_physique

ln -s fichier lien_symbolique

options:

- -s: crée un lien symbolique
- -f: force la création même si la destination existe déjà
- --: fin des options (pour permettre le traitement d'un fichier dont le nom commence par « « - »

Exemples

Bilan

A la fin de cette première séance, vous devez :

- connaître les notions de systeme de gestion de fichiers, fichiers, inode, dossier, chemin
- connaître la forme générale d'une commande
- savoir utiliser le manuel unix
- savoir vous déplacer dans une arborescence,
- créer/déplacer/copier des fichiers, des dossiers

Partie No 2

processus

quelques filtres classiques

processus

un programme: un fichier sur disque

un processus: un programme en cours d'exécution

- le code exécutable du programme
- les données de l'instance en train de s'exécuter

programme réentrant:

- deux instances du même programme partagent le même code exécutable
- elles ont par contre chacune leurs données

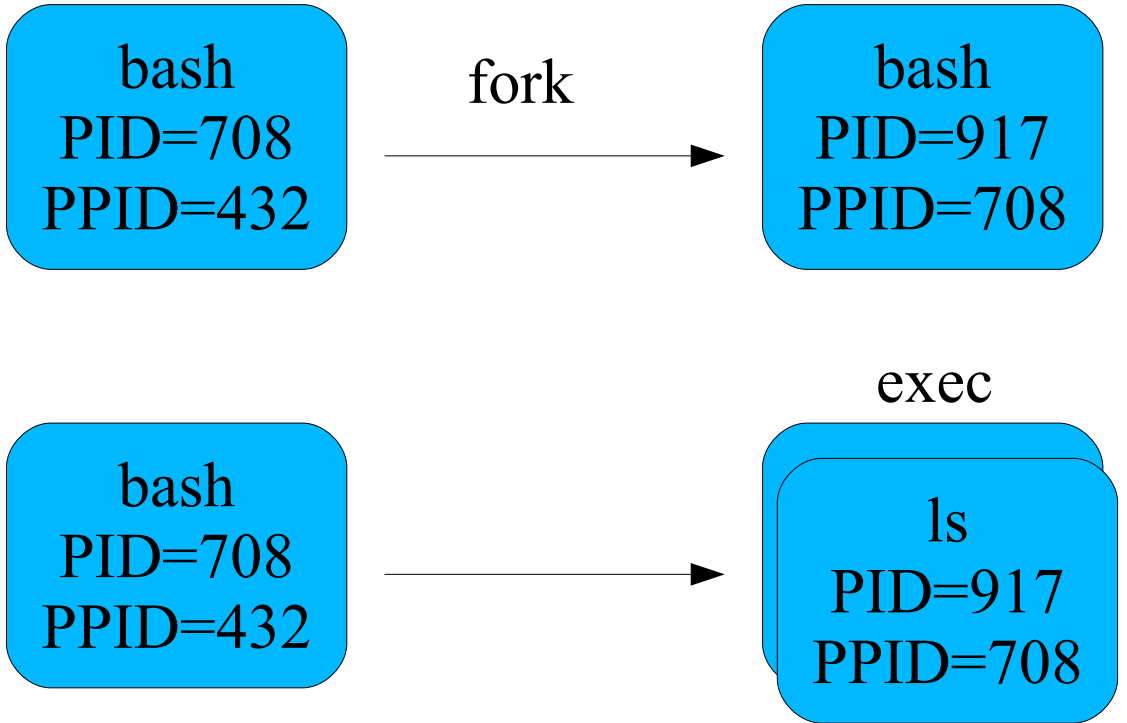
processus système (daemon)/utilisateur

hiérarchie de processus, recouvrement

un processus (processus fils) est toujours créé par un autre processus (processus père):

fork: création d'une copie du processus père

exec: recouvrement par le processus fils



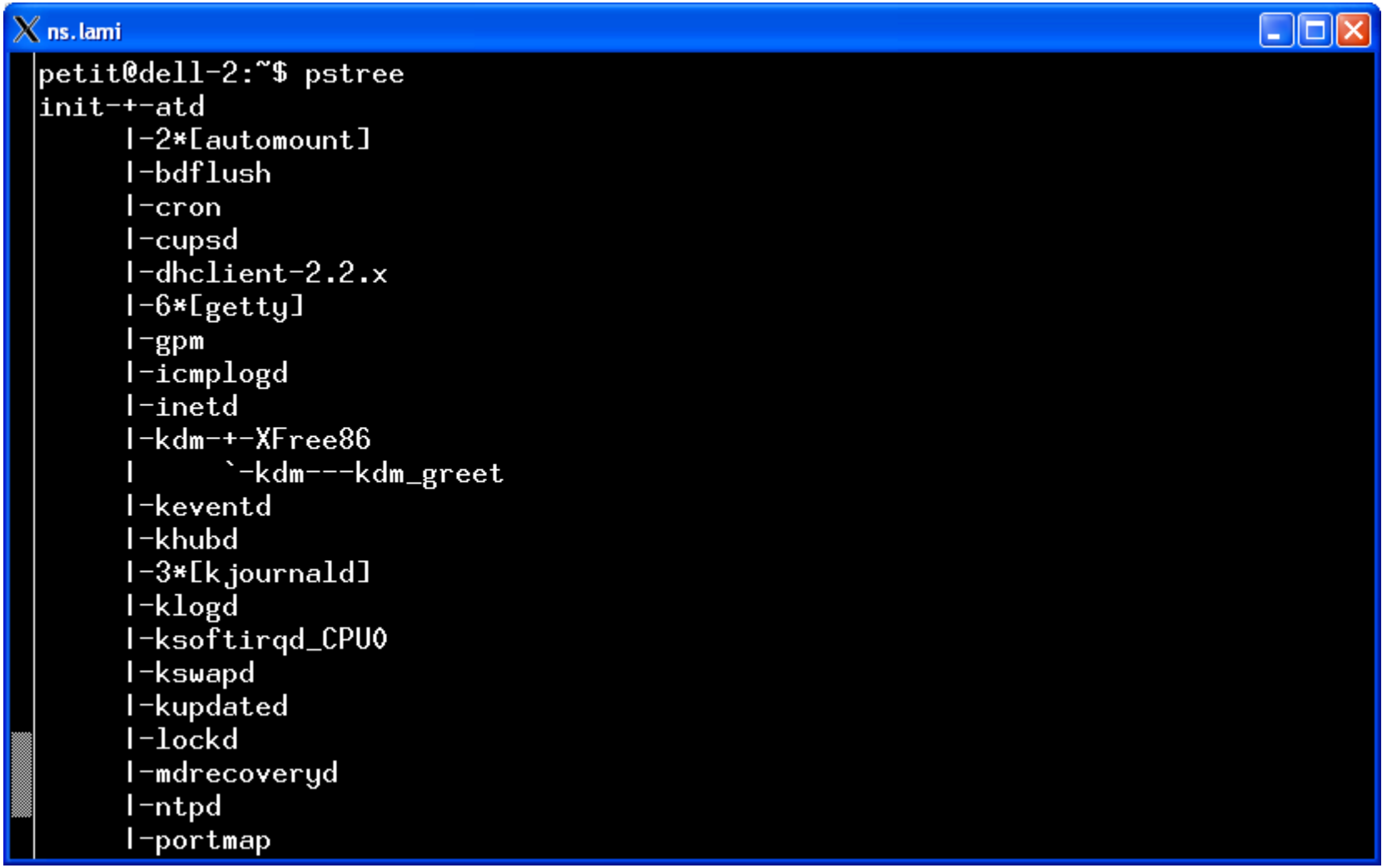
Hiérarchie de processus

tout processus a un processus parent sauf le processus initial

processus initial : init (pid 1)

arrêter la machine: demander à init d'arrêter tous ses processus fils

pstree


ns.lami
petit@dell-2:~$ pstree
init-+-atd
|-2*[automount]
|-bdflush
|-cron
|-cupsd
|-dhclient-2.2.x
|-6*[getty]
|-gpm
|-icmplogd
|-inetd
|-kdm-+-XFree86
|`-kdm---kdm_greet
|-keventd
|-khubd
|-3*[kjournald]
|-klogd
|-ksoftirqd_CPU0
|-kswapd
|-kupdated
|-lockd
|-mdrecoveryd
|-ntpd
|-portmap

commandes internes/externes

commande externe: fichier exécutable:

- création d'un nouveau processus chargé d'exécuter la commande

commande interne:

- exécutée par le shell (pas de création de nouveau processus)

type

type indique si une commande est interne

options non standard:

-a: indique toutes les implémentations

-p : indique le chemin de la commande (rien si interne)

Exemples: testez type sur les commandes suivantes :

cd

ls

pwd

file

caractéristiques des processus

statiques

- PID
- PPID
- propriétaire
- terminal d'attache pour les entreés-sorties

dynamique

- priorité
- nice
- consommation cpu/mémoire
- dossier de travail

commande ps

2 syntaxes:

syntaxe Systeme V: option précédées de -

syntaxe BSD: options NON précédées de -

quelques options SysV:

-e ou -A: tous les processus

-a: tous les processus associés à un terminal

-H: représentation hiérarchique (forêt)

-f: format complet;-l: format long (encore plus détaillé)

-o: pour modifier le format de sortie (cf manuel)

-g, -p, -t, -u: n'affiche que les processus des groupe (-g), processus (-p), terminaux (-t) ou utilisateurs (-u) listés.

commande ps: exemple

Etat d'un processus

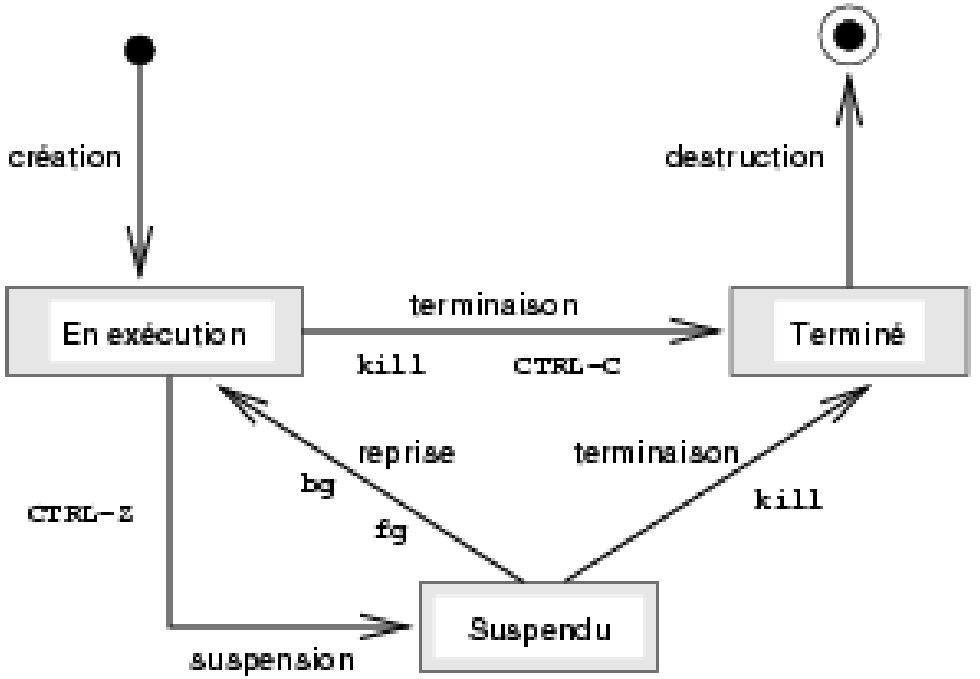
R: exécution

Z: zombi: il est mort mais son père ne le sait pas

D: le processus ne peut être interrompu

S: suspendu

T: terminé



gestion de processus

&

bg

fg

jobs

Ctrl-C

Ctrl-Z

Signaux

permettent au système de communiquer avec les processus

signaux utiles

- STOP: suspendre
- CONT: reprendre
- HUP (1): souvent: relecture configuration
- KILL(9): tuer sans possibilité de traitement
- INT(2): équivalent à Ctrl-C: interruption gérable. permet au processus de gérer son interruption

kill -signal PID

trap

dans une version future de ce document

avant plan/arrière plan/détachement

dans une version future de ce document (ràf)

priorité des processus

l'exécution des divers processus est gérée par un ordonnanceur (scheduler)

une priorité est définie dynamiquement

but: que chaque processus puisse avancer son exécution tout en respectant des priorités

nice: permet d'influer sur la priorité des processus

- de 0 à 19 pour un utilisateur
- de -20 à 19 pour root
- plus le chiffre est élevé, moins le processus est prioritaire

code de retour

valeur à laquelle le processus père peut accèder

0: terminaison normale

autre valeur: situation anormale

commande1 && commande2:la commande2 est exécutée si la commande 1 réussit

commande1 || commande2: la commande2 est exécutée si la commande 1 échoue

exemple: commande test

exemple: construction if/then/else/fi

commande test

réalise des tests simple, le code de retour indique que le test est positif ou négatif

test -d /var/tmp : teste si /var/tmp est un dossier

test -x /bin/ls: teste si /bin/ls est un exécutable

test 1 = 2: teste l'égalité de deux chaînes

forme alternative :

```
test -d /var/tmp
[ -d /var/tmp ]
```

structure de contrôle if

syntaxe:

```
if commande1
then
    commande2
[elif commande3
then commande4]
...
[else
    commande5]
fi
```

si commande1 retourne 0, on exécute commande2
sinon, si commande3 retourne 0, on exécute commande4 ... sinon commande5

Entrées-sorties

Entrées-sorties

- entrée standard: 0
- sortie standard: 1
- sortie d'erreur standard: 2

Héritage

les descripteurs d'un processus enfant sont initialement les mêmes que ceux du processus père.

si on ne les modifie pas, en sortie, les affichages s'entrelacent.

il est possible de changer la valeur de entrée standard et des sorties standard et en erreur

pour les rediriger depuis/vers un fichier : <, >, >>, 2>, 2>>

pour les rediger depuis/vers un processus : |

redirection des sorties

>: le contenu du fichier est remplacé par la sortie de la commande

>>: la sortie s'ajoute à la fin du fichier

exemples:

ls /etc > /tmp/foo.txt

cat ls

ls /usr/bin >> /tmp/foo.txt

du -sk /var/* > /tmp/bar.txt

date > /tmp/bar.txt (noter que le No d'inode est inchangé)

redirection de la sortie d'erreur standard

commande 2> fichier

commande 2>> fichier

pratique pour isoler messages d'erreur et sortie

/dev/null: le trou noir: pour éliminer les messages d'erreur

du -sk /var/* 2> /dev/null

redirection simultanées

on peut rediriger plusieurs descripteurs sur une même ligne de commande

ls > /tmp/f1 2> f2

les redirections sont traîtées de gauche à droite

du -sk /var/* > /tmp/resultat 2> /tmp/erreur

en cas de redirection simultanée avec cette syntase: impérativement vers des fichiers différents

redirection en entrée

commande < fichier

exemples:

mail petit < texte

wc -l < /etc/passwd

redirections avancées

&1: valeur du descripteur de fichier 1

&2: valeur du descripteur de fichier 2

1>&2: l'entrée standard est redirigée vers le même fichier que la sortie standard

sert pour des redirection simultanées vers un même fichier

Exemples:

ls > /tmp/test 2>&1 #OK

ls 2>&1 >/tmp/test #pas OK: stderr est toujours lié au terminal

<<

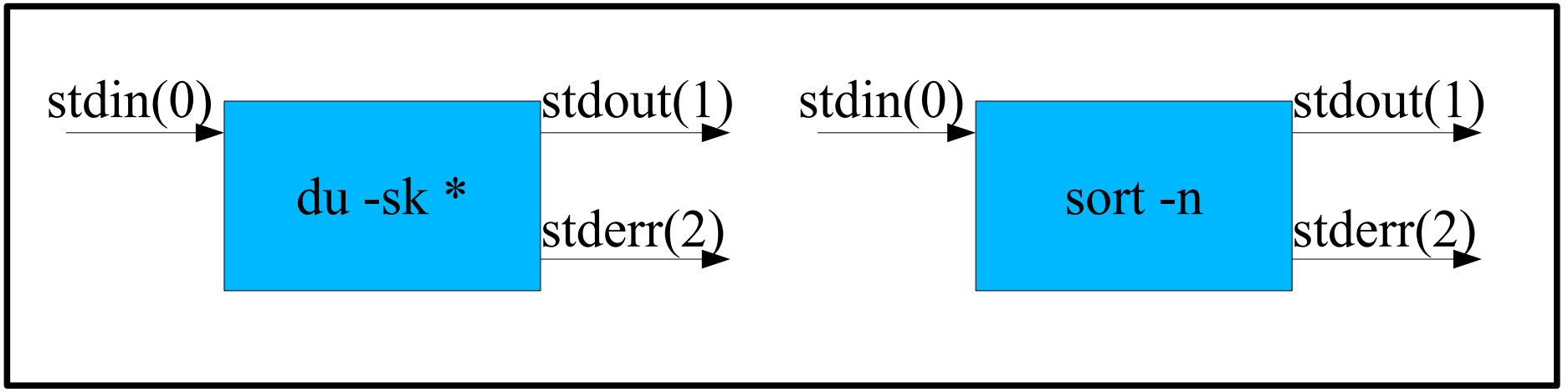
dans une version ultérieure de ce document

fermeture d'un descripteur

dans une version ultérieure de ce document

enchaînement de commandes

du -sk * |sort -n



L'ensemble forme une nouvelle commande

Filtres

commande lisant leurs données sur l'entrée standard et envoyant leur sortie sur la sortie standard

pratique pour les enchaînements de commandes

philosophie unix: des commandes simples que l'on combine entre elles

sort

selon SUSv3, sort a trois fonctions sur son entrée standard ou des fichiers textes constituées de lignes contenant un ou plusieurs champs :

- trier les données (par défaut)
- fusionner des fichiers triées en une sortie globale triée (option -m)
- vérifier que les données sont triées (option -c)

Sort: options courantes:

-t car_sep: permet de préciser le caractère qui sépare les champs du fichier

-k : précise les champs sur lesquels portent le tri

-o: indique un fichier de sortie (par défaut: sortie standard)

-d: supprime les doublons

-c : vérifie si un fichier est trié. Le résultat est indiqué uniquement par le code de retour: 0 si trié, 1 sinon.

-m : fusionne des fichiers supposés déjà triés

uniq

supprime les doublons d'une liste triée

exemple: cat /tmp/test.txt |sort|uniq

voir manuel pour les autres options

tail/head

queue/tête d'un fichier

wc

compte le nombre de lignes, de mots et de caractères

wc -l : nombre de lignes

wc -w : nombre de mots

wc -c : nombres de caractères

ls | wc -l : donne le nombre de fichier du dossier courant

grep, egrep & Co

grep chaine: sélectionne les lignes qui contiennent la chaine

cut

sélectionner certaines colonnes

tr

more/less

find

commandes qui ne lisent pas leur entrée standard

ls, who, find

chmod, cp, mv, rm, ln, mkdir

date

kill

file, type

echo

Bilan

A la fin de cette deuxième partie, vous devez :

- savoir ce qu'il un PID, PPID
- comprendre et savoir utiliser sur des exemples de base les redirections et les enchaînements de commandes
- comprendre ce qu'est un filtre
- connaître quelques filtres de base

Partie No 3

premiers scripts shell

variables

Scripts shell

fichier texte contenant des commandes

nouvelle commande

lancement d'un script shell

méthode nécessitant une accès en lecture

sh nomScript

sh < nomScript (peu utilisée)

Méthodes nécessitant un accès en exécution et en lecture

nomScript ou chemin/nomScript ou ./nomScript

la première ligne du script qui précise le choix de l'interpréteur est de la forme:

#! /bin/bash

Exemple:

considérons le script /tmp/foo.sh:

```
#! /bin/bash
pwd
cd /etc
pwd
echo coucou
```

Utiliser les trois méthodes proposées pour le lancer

exécuter la commande pwd après chaque lancement

ces 3 méthodes exécutent le script dans un processus shell enfant. Comment le vérifier ?

Exécution par le shell courant

on utilise la commande interne « . »

. foo.sh

conséquences :

- le script est exécuté par le shell courant
- il peut modifier les données de ce shell courant (variables, dossier de travail, ...)

Exemple: exécuter « . foo.sh »

Erreurs classiques

le dossier courant n'est pas dans le PATH. Indiquer le chemin absolu ou relatif. Par ex.: ./foo.sh

```
cd /tmp;foo.sh
```

```
commande non trouvée
```

le fichier n'est pas exécutable. Utilisez chmod pour lui ajouter le droit x:

```
cd /tmp;./foo.sh
```

```
permission non accordé
```

Code de retour

exit: termine le script en retournant un code de retour

Rappel:

0 : exécution normale

autre valeur: situation anormale

sans utilisation de exit: le code de retour est celui de la dernière commande du shell

Variables

variables d'environnement

- exportée ou non exportées

variables utilisateurs

variables réservées du shell

Variables d'environnement

variables au shell et/ou au commandes lancées à partir de lui

set : pour obtenir la liste des variables d'environnement

obtenir la valeur d'une variable: $nomVariable ou ${nomVariable}

changement de valeur: nomVariable=valeur

variables exportées:

variables communiquées aux commandes lancées
env: pour obtenir la liste des variables exportées
export nomVariable

Exemples de variables d'environnement

Nom	rôle
PATH	liste des dossiers où le shell cherche les exécutables
HOME	dossier personnel de l'utilisateur
PWD	dossier courant
TERM	type de terminal
SHELL	nom du shell en cours d'exécution
USER	
LANG	lié aux locale et à la gestion des langues
DISPLAY	« display » X Window (clavier/écran/souris)

Quelques variables d'environnement définies sous bash

variables utilisateur

définie par l'utilisateur pour ses besoins

nom des variables

- premier caractère [a-zA-Z]
- caractères suivants [a-zA-Z0-9_]

définition: nom=mot

valeur: $nom ou ${nom}

variables utilisateur (2)

variables indéfinie:

- une variable jamais initialisée est vide
- unset permet d'annuler la définition d'une variable qui devient alors indéfinie

```
$test=hello
$echo $test
hello
$unset hello
$echo $hello
```

variables utilisateurs (3)

espace dans les valeurs: " ou \

```
test1="m1 m2 m3"
test1=m1\ m2\ m3
```

Attention à bien isoler le nom d'une variable:

```
f= toto;g=titi;h=$titi_$toto # $titi_ est vide
f= toto;g=titi;h=${titi}_$toto
```

Substitution de commandes

remplacer une commande par l'affichage de son résultat d'exécution

ancienne syntaxe à éviter : `commande`

syntaxe moderne: $(commande)

Exemples:

```
$echo je suis $(whoami)
je suis petit
$nom=$(whoami)
$echo je suis $nom
```

caractères de protection

apostrophe : protège tous les caractères spéciaux sauf elle

echo '$HOME' affiche '$HOME'

echo $HOME affiche la valeur de la variable HOME

\: protège le caractère qui suit

guillemets : protège tout sauf lui-même, $, $() et \

variables réservées du shell

paramètres positionnels

- $#: nombre d'arguments
- $0: nom du script
- $1: valeur du premier argument
- $2: valeur du deuxième argument
- ...
- $9: valeur du neuvième argument
- ${10}: avec des shells modernes
- $* et $@: liste de tous les arguments

variables réservées du shell (2)

shift [n]: décale de n positions la liste des arguments vers la gauche

- par défaut, n vaut 1
- shift: remplace $1 par $2, $2 par $3, ...

$?: code de retour de la dernière commande exécutée

$$: pid du shell

$!: pid du dernier processus lancé en arrière plan

Bilan partie No 3

A la fin de cette partie No 3, vous devez :

- être à l'aise dans l'utilisation des filtres
- savoir définir une variable et utiliser sa valeur
- savoir écrire de petits scripts shell
- faire la différence entre variables exportées et non exportées

Partie No 4

fonctions du shell

substitutions de noms de fichiers

structures de contrôles

for/in/do/done

expressions rationnelles, commande grep

fonctions du shell

permet d'isoler un traitement particulier constitué de plusieurs commandes

une fonction est identifiée par son nom

elle doit être définie avant d'être utilisées. Une fois définie, une fonction est considérée comme une commande interne.

une fonction s'utilise comme une commande: « nom liste_de_paramètres »

la définition d'une fonction n'entraîne l'exécution d'aucune commande. C'est chaque appel à cette fonction qui déclenche l'exécution des fonctions

définition d'une fonction

syntaxe 1 (conseillée car la plus compatible):

```
mafonction () {
    commande1
    commande2
    ...
}
```

syntaxe 2 (shells récents):

```
function maFonction {
    commande1
    commande2
    ...
}
```

Exemple

considérons le script suivant:

```
#! /bin/sh
echo "1ere commande"
maPremiereFonction(){
    echo "coucou depuis maPremiereFonction"
}
echo "2e commande"
maPremiereFonction
echo "après l'appel de maPremièreFonction"
```

son exécution provoque l'affichage suivant:

```
1ere commande
2e commande
coucou depuis maPremièreFonction
après l'appel de maPremièreFonction
```

Paramètres des fonctions

comme avec toute commande, il est possible de fournir des paramètres à une fonction.

- utilisation :
  - les paramètres sont ajoutés après le nom de la fonction, séparés par des espaces ou tout autre caractère séparateur
  - exemple: test premier second
- définition de la fonction
  - les paramètres sont des variables locales à la fonction nommées: $1, $2, ..., $ç, ${10}, ...
  - les variables spéciales $#, $* et $@ sont aussi disponibles
  - $0 contient par contre le nom du script (et pas le nom de la fonction)

paramètres: exemple

script: test.sh

```
#! /bin/bash
f1 () {
    echo "1er argument de f1: $1"
    echo "deuxieme argument de f1: $2"
    echo "nombre d'arguments de f1: $#"
    echo "tous les arguments de f1: $*"
    echo "nom du script: $0"
}
echo début du script
echo "1er argument du script: $1"
echo "deuxieme argument du script: $2"
echo "nombre d'arguments du script: $#"
echo "tous les arguments du script: $*"
echo "nom du script: $0"
f1 quand est-ce "qu'on" mange
```

paramètres: exemple

execution du script: « ./test.sh salut les tepos »

```
début du script
1er argument du script: salut
deuxieme argument du script: les
nombre d'arguments du script: 3
tous les arguments du script: salut les tepos
nom du script: test
f1 quand est-ce "qu'on" mange#! /bin/bash
1er argument de f1: quand
deuxieme argument de f1: est-ce
nombre d'arguments de f1: 4
tous les arguments de f1: quand est-ce qu'on mange
nom du script: ./test.sh
```

code de retour d'une fonction

comme toute commande, une fonction a un code de retour

commande « return n»: termine la fonction et retourne le code d'erreur « n ».

Erreur à ne pas faire: *return* ne retourne pas le résultat de la fonction mais son code d'erreur

Exemple : recherche dans un annuaire

on considère le fichier suivant:

```
$ cat annuaire
petit 0169478047
acces2400 0136642424
joe 0404056789
sophie 0164570101
Malik 0237463201
```

script test.sh

```
#! /bin/bash
estDansAnnnuaire (){
   if grep $1 annuaire
   then return 0
```

Exemple revu (1)

script test.sh

```
#! /bin/bash
estDansAnnuaire (){
   return grep $1 annuaire
   }
if estDansAnnuaire petit
   then petit est présent dans annuaire
   else pas de petit en vue
fi
```

on utilise directement le code de retour de la commande grep.

Exemple revu (2)

script test.sh

```
#! /bin/bash
estDansAnnuaire (){
    if [ $# -ne 1 ]
    then
      echo "estDansAnnuaire: utilisation sans
     argument"
       return 1
    fi
    return grep $1 annuaire
    }
if estDansAnnuaire petit
    then petit est présent dans annuaire
    else pas de petit en vue
fi
```

programmation défensive: vérification du nombre de paramètres.

Variables

les variables utilisées dans un script sont globales.

une exception: les variables locales définies par *typeset* (ne sont pas au programme de l'UEL).

leur portée est dynamique: la variables est utilisable dès que le flot d'exécution l'a rencontrée.

variables locales

ràf: dans une version ultérieure de ce document et de cet enseignement.

fonction: retourner une valeur

une fonction à 3 modes de communication avec le monde extérieur:

- via le code de retour : ne pas faire sauf pour une valeur binaire vrai/faux « à la grep »
- via la sortie standard: méthode conseillée car utilisable via redirection. On se contente d'afficher le résultat sur la sortie standard
- via une variable : la variable porte en général le nom de la fonction.

Exemples

```
#! /bin/bash
double(){
   echo $1$1
}
triple (){
   triple=$1$1$1
}
d=$(double pa)
triple pa
echo le double de pa est $d
echo le triple de pa est $triple
```

exécution:

```
le double de pa est papa
le triple de pa est papapa
```

substitutions de noms de fichiers

les noms de fichiers paramètres de commandes du shell peuvent être cités exhaustivement ou décrits à l'aide d'expression générique (« caractères joker »)

caractères:

*: une suite de caractères

?: un unique caractère

[abcd]: l'un des caractère entre crochets

[a-d]: un caractère situé entre a et d

[^*liste*]: cité en première position, ^ est un caractère de négation: tous les caractères sauf ceux de *liste*.

Exemples

*.sh: les fichiers dont le nom finit par .sh

p*: les fichiers dont le nom commence par la lettre p

[A-Z]*.sh : les fichiers dont le nom commence par une majuscule et finissant par .sh

???: les fichiers dont le nom fait 3 lettres

[^a-z]* : les fichiers dont le nom ne commence pas par une miniscule non accentuée

structures de contrôle: for

syntaxe:

for *var* in *listeValeur*

do

commande1

...

done

sémantique:

la variable *var* prend chaque valeur de la liste de valeur

les commandes situées entre do et done sont exécutées pour chaque valeur de la variable *var*

la valeur de la variable i est utilisable dans zone située

Exemple 1: liste de valeurs citée

```
$for i in 1 2 3 4
 do
      echo $i
done
1
2
3
4
```

Exemple 2: liste des valeurs contenues dans une variable:

```
$ varTest="a b c d"
$for i in $varTest
do
   echo $i
done
a
b
c
d
```

Exemple 3: liste des valeurs via substitution de commande

```
$for i in $(ls)
do
  echo $i
done
fichier1
fichier2
... (résultat de la commande ls, un fichier par ligne)
```

Exemple 4: liste des valeurs via génération de noms de fichiers

```
$for i in f*
do
  echo $i
done
fichier1
fichier2
... (les noms des fichiers commençant par f dans le
   dossier courant, un fichier par ligne)
```

Expressions rationnelles

utilisées par des nombreux outils pour décrire des chaînes de caractères

Il existe deux familles d'expressions rationnelles :

- expression régulières de base utilisées par
  - vi
  - grep
  - expr
  - sed
- expressions régulière étendues utilisées par :
  - grep -E et egrep
  - awk

expressions rationnelles: éléments communs

les caractères suivants sont communs aux expressions rationnelles de base et étendues:

- ^: début de ligne
- $: fin de ligne
- .: un caractère quelconque
- [liste]: un caractère de la liste (idem [] du shell, ^: négation)
- *: 0 à n fois le caractère/expression précédente
- \<, \>: chaîne vide de début et de fin de mot
- \c: protège le caractère c de tout interprétation. ex.: \$: caractère $

Exemples:

petit: la chaîne petit

^petit: chaîne démarrant par petit

petit$: chaîne finissant par petit

DSCN[0-9]*\.JPG: DSCN suivi de 0 à N chiffres suivi de .JPG
(noter la protection du caractère .)

^$: chaîne vide (utile pour détecter les lignes vides)

^[^:]*:[^:]*:[^:]*:1000 une chaîne contenant des champs séparés par le caractère : et dont le 4e champ vaut 1000. Appliqué sur un fichier /etc/passwd, sélectionne les lignes dont le groupe est 1000.

Bilan

A la fin de cette quatrième partie, vous devez :

- être capable de définir une fonction ayant des paramètres
- savoir utiliser la structure de contrôle for/in/do/done
- savoir utiliser grep et egrep (grep -E) avec des expressions rationnelles simples

partie No 5

Cours :

- caractères de protection
- regroupements de commandes
- expressions arithmétiques
- analyse et interprétation de la ligne de commande
- comparaison de $* et de $@
- mise au point des scripts shells
- structures de contrôles
  - while/do/done
  - case/in/esac
  - application au décodage des options d'un script

caractères de protection

\: protège le caractère suivant (sauf si c'est un saut de ligne)

\ en fin de ligne:

'chaine': protège tous les caractères de la chaîne

"chaine": protège tous les caractères sauf $, ` et \ dans le cas où \ est devant $, ", \ et en fin de ligne.

caractères à protéger :

| & ; < > ( ) $ ` \ " ' <espace> <tab>

<saut de ligne> * ? [# ~ = %

caractères de protection: exemples

```
$ echo 'un test $#\c\$'
un test $#\c\$
$ echo "un test $#\c\$"
un test 0\c$
$ echo un test \$
un test $
$ echo un test \$ \\
un test $ \
```

regroupement de commandes

() et {}

pour rediriger les sorties des commandes vers une même fichier

pour exécuter des commandes dans un même contexte

(): exécution dans un shell enfant

```
cd /;(cd /tmp;ls);pwd
(pwd;w)> /tmp/test1.txt
```

{}: exécution par le shell courant

il faut un espace avant et après chaque accolade et un ; après la dernier commande

```
cd /; { cd /tmp;ls; } ;pwd
{ pwd; w } > /tmp/test2.txt
```

expressions arithmétiques

mécanisme historique: via la commande expr

mécanisme moderne (SUS V3, POSIX): (( )) et $(( ))

expr:

l'expression arithmétique doit être passée sous la forme d'arguments séparés par des espaces (représentés ici par _)

1_+_2 : 3

1+2: chaîne 1+2

les caractères qui peuvent être interprétés par le shell doivent être protégés:

* (et pas * seul)

\> (et pas > seul)

...

expr: exemples

```
$ expr 2 + 5
7
$ expr 2 * 3
syntax error
$ expr 2 \* 3
6
$ x=1
$ y=$( expr $x + 3 )
$ echo $y
4
```

expr: quelques opérateurs

opérateur	sens
n1 + n2	
n1 - n2	
n1 * n2	
n1 / n2	
n1 % n2	modulo
\>, \<, \>=, \<=, =, !=	opérateurs de comparaison
chaine : expression_rationnelle	comparaison
\(expression \)	regroupement
s1 \& s2	vrai si s1 et s2 sont non nulles (valeur 0 et chaîne nulle)
s1 | s2	vrai si s1 ou s2 est non nulle

commande (( ))

syntaxe: ((expression)) ou let expression

avantages par rapport à expr:

- plus grand choix d'opérateurs (ceux du C)
- les caractères spéciaux du shell peuvent ne pas être protégés
- les noms de variables peuvent ne pas être préfixé par $
- les affectations se font directement dans la commande

(( )): quelques opérateurs

opérateur	sens
n1 + n2	
n1 - n2	
n1 * n2	
n1 / n2	
n1 % n2	modulo
>, <, >=, <=, ==, !=	opérateurs de comparaison
(expression)	regroupement
s1 && s2	ET logique
! e1	négation de e1
s1 || s2	OU logique
~, >>, <<, &, |, ^	travaillant sur les bits
+=, -=, *=, ...	n+=e équivaut à n=n+e

(( )) : exemples

```
$ x=1
$ ((y=x+1))
$ echo $y
2
$ z=7
$ z*=y
$ echo $z
14
$ if (( (x==1) && (y>10) ))
  then
```

substitution d'expressions arithmétiques

syntaxe: $((expression arithmétique))

Exemple:

```
$ echo 1 + 2 vaut $(( 1+2))
1 + 2 vaut 3
$ x=1
$ echo x vaut maintenant $(( x=x+1))
x vaut maintenant 2
$  echo essai ((1+2))
bash: syntax error near unexpected
  token `(('
$ echo essai $( ((1+2)) )
essai
```

Interprétation de la ligne de commande

séparation en mots

expansion des accolades

expansion du ~

expansion des paramètres, des variables

expansion des commandes

évaluation des expressions arithmétiques

découpage des mots

développement des noms de fichiers

suppression des " \ ' non protégés ne résultant pas d'un développement

découpage des mots

découpage de tout ce qui n'est pas entre guillemets

les caractères séparateurs sont dans la variable IFS (par défaut: espace, tab, retour chariot)

les arguments explicitement nuls (" ou "") sont conservés. Ceux résultant du développement d'un paramètre sans valeur sont éliminés

Développement

des accolades :

{mot1, mot2, mot3, ...}

file{2,34,IMP} donne file2 file34 fileIMP

du tilde:

~nomLogin

~petit

paramètres/variables:

${parametre}ou $parametre

$!var: on remplace !var par la valeur de var avant de faire l'expansion (indirection)

Développement des commandes

$(commande)

ou `commande` (ancienne forme déconseillée)

la sortie standard de la commande remplace l'expression

ls -l $(grep -il toto *.c)

développement des expressions arithmétiques

$((expression))

l'expression est traitée comme si elle se trouvait entre guillemets (sans traitement spécifique pour le guillemet)

l'expression subit le développement des paramètres/variables et des commandes

x=1; echo nouvelle valeur $((x=x+3))

développement des noms de fichiers

*: n'importe quelle séquence

?: un caractère quelconque

[abcdef]: un caractère parmi tous les caractères entre crochet

[a-f]: tous les caractères compris entre a et f

[- : le caractère -

[^ ou [!: tout sauf les caractères qui suivent ^ ou !

$* et $@

contiennent la liste des arguments d'un script ou d'une fonction

$* ou $@équivalentes si non entourées de guillements "

"$@" : les espaces internes aux arguments sont protégés

"$*": les espaces internes aux arguments ne sont pas protégés

"$@" sert traditionnellement quand un script peut avoir des noms de fichiers contenant des espaces comme paramètres.

$*: exemple

```
cat exemple.sh
#! /bin/bash

for i in $*
do
  echo $i
done

$ ./exemple.sh a "Program Files" test
a
Program
Files
test
```

"$*": exemple

```
cat exemple.sh
#! /bin/bash

for i in "$*"
do
  echo $i
done

$ ./exemple.sh a "Program Files" test
a Program Files test
```

"$@": exemple

```
cat exemple.sh
#! /bin/bash

for i in "$@"
do
  echo $i
done

$ ./exemple.sh a "Program Files" test
a
Program Files
test
```

mise au point des scripts

- La mise au point est un gros point faible des shells
  - pas de débugger
  - portée dynamique des variables
  - pas de compilation: les erreurs de syntaxe sont détectées quand le flot d'exécution les atteint
- les outils utilisés pour la mise au point sont minimaux et archaïques
  - des options du shell permettant un affichage verbeux lors de l'exécution
  - afficher la valeur de variables à l'aide de commandes echo ou printf
  - mettre des points d'arrêt en lançant des sous-shell

option du shell: -e

set -e: interrompt le shell à la première commande dont le code de retour est non nul

Exemple: que se passe-t-il si le dossier n'existe pas ?

```
#! /bin/bash
cd /var/spool/aSupprimer
rm -rf *
```

c'est une bonne pratique d'intégrer un 'set -e' à tous ses scripts.

option du shell: -u

set -u: l'utilisation de la valeur d'une variable non définie est considérée comme une erreur.

L'utilisation de ce paramètre supprime de nombreuses erreurs dues à des fautes de frappe

Exemple:

```
$ echo $aaa
-bash: a: unbound variable
$ aaa=12
$ echo $aaa
12
```

option du shell: -x

set -x: affiche les commandes telles qu'elles sont exécutées (après analyse de la ligne de commande et substitutions)

Exemple:

```
$cat ex2.sh
#! /bin/bash
set -x
x=1
echo $x $(( 1+3)) t{a,b} *
$ ls
F3  ex2.sh  f1  f2
petit@ns:/tmp/Test$ ./ex2.sh
+ x=1
+ echo 1 4 ta tb F3 ex2.sh f1 f2
```

option du shell: -v

set -v : affichage des commandes avant substitution

on reprend l'exemple précédente en remplaçant « set -x » par « set -v »:

```
$ ./ex2.sh
x=1
echo $x $(( 1+3)) t{a,b} *
1 4 ta tb F3 ex2.sh f1 f2
```

option du shell: -n

set -n: lecture des commandes sans les exécuter (pour détecter des erreur de syntaxe)

Exemple:

```
$ cat ./ex2.sh
#! /bin/bash
set -n
x=1
echo coucou
echo $x $(( 1+3) t{a,b} *
$ ./ex2.sh
./ex2.sh: line 6: unexpected EOF while looking
  for matching `)'
./ex2.sh: line 7: syntax error: unexpected end
  of file
```

structures de contrôle: case/in/esac

syntaxe:

```
case mot in
   modele1) liste de commandes;;
   modele2|modele3|modele4) liste de
     commandes;;
   ...
esac
```

les modèles sont des chaînes incluant éventuellement des caractères spéciaux * ? []^ (utilisés dans la substitution des noms de fichiers)

structures de contrôle: case/in/esac

principe

- la valeur du mot est comparée à chaque modèle
- la liste de commandes du premier modèle correspondant est exécutée jusqu'au « ;; »
- l'exécution se poursuit ensuite après le « esac ».

case/in/esac: exemples

```
$ cat codePostal.sh
#! /bin/bash

case "$1" in
  75[0-9][0-9][0-9])
    echo "code postal parisien"
    ;;
  7[78][0-9][0-9][0-9]|9[1-5][0-9][0-9][0-9])
    echo "code postal ile de France"
    ;;
  [0-9][0-9][0-9][0-9][0-9])
    echo "code postal de la france metropolitaine"
    ;;
  *)
   echo "code postal non reconnu"
   ;;
esac
```

structures de contrôles: while/do/done

syntaxe

```
while commande1
do
   liste de commandes
done
```

« commande1 » est exécutée,

si le code de retour vaut 0, la liste de commande est exécutée et on retourne exécuter « commande1 »

si le code de retour de « commande 1 » est différent de 0, la commande qui suit le « done » est exécutée.

while/do/done: exemples

```
$ cat facto.sh
#! /bin/bash
n=5
x=1
while (( n > 0))
do
   ((x*=n))
   ((n-=1))
done
echo "12!=$x"

$ ./facto.sh
12!=120
```

while/do/done: exemples

```
$ cat test.sh
#! /bin/bash
n=1
while (( $# > 0 ))
do
  echo "parametre No $n: $1"
  ((n=n+1))
  shift
done

$ ./test.sh a g df "ze rt"
parametre No 1: a
parametre No 2: g
parametre No 3: df
parametre No 4: ze rt
```

commande READ

syntaxe: read var1 var2 ...

lit son entrée standard et affecte le premier mot lu à var1, le deuxième à var2, ... le texte restant est affecté à la dernire variable

Exemple:

```
echo hello world et salut les tepos
  read a b c
```

a contient hello

b contient world

c contient « et salut les tepos »

commande READ: utilisation

2 utilisations typiques:

pour mettre dans une variable la saisie d'un utilisateur

en liaison avec while :

```
while read a b
do
   traitement de a et b
done
```

la boucle s'arrête en fin de fichier (caractère Ctrl-D si sur une saisie utilisateur)

READ : exemple

on suppose que le fichier nombres.txt contient deux entiers par ligne

```
$ cat test-read.sh
#! /bin/bash
while read a b
do
  echo $a + $b = $(( a+b ))
done
$ ./test-read.sh < nombres.txt
1 + 3 = 4
6 + 4 = 10
2 + -5 = -3
```

Bilan

A la fin de cette cinquième partie, vous devez savoir utiliser:

- les caractères de protection: ' " \
- les expressions arithmétiques entières avec (( ))
- les options et techniques de mise au point
- les structures de contrôle case et while
- la commande read

Partie No 6

Cours :

- substitution de variables
- entrées/sorties
- signaux
- analyse de paramètres de la ligne de commande
- commandes de base: find, xargs, tee

option du shell: -u

set -u: l'utilisation de la valeur d'une variable non définie est considérée comme une erreur.

L'utilisation de ce paramètre supprime de nombreuses erreurs dues à des fautes de frappe

Exemple:

```
$ echo $aaa
-bash: a: unbound variable
$ aaa=12
$ echo $aaa
12
```

de la bonne utilisation de la commande test et des valeurs des variables

"$var" vs $var

les bonnes pratiques concernant l'utilisation de la commande test :

- [-n "$a"] && ["$a" = "$b"] au lieu de [-n "$a" -a "$a" = "$b"]
- toujours mettre les variables entre apostrophes ": "$var" et pas $var
- utiliser les operateurs arithmetiques pour les nombres et literaux pour les chaines
- utiliser ["x$A" = "aexpression"] pour comparer la valeur de variables (pour éviter des problèmes si $A est vide)
- Les operateurs -e, -nt, -ot, -N, -L, -h, >, <, ne sont pas portables. -e, -h et -L sont POSIX toutefois. -h et -L sont raisonablement portables

substitution de variables

attribuer une valeur par défaut aux variables:

	valeur si la variable		
expression	est non nulle	est nulle	effet de bord
${variable:-valeur}	$variable	valeur	
${variable:=valeur}	$variable	valeur	la variable prend la valeur valeur
${variable:+valeur}	valeur	$variable	

afficher un message d'erreur si la variable est vide:

${variable:?message}: si la variable est vide, le shell affiche le message et provoque la terminaison du script shell. Si la variable n'est pas vide, l'expression vaut $variable

substitution de variables (2)

longueur de la valeur d'une variable: ${#variable}

suppression de fragments :

à gauche			
	#	plus petit	${variable#modele}
	##	plus grand	${variable##modele}
à droite			
	%	plus petit	${variable%modele}
	%%	plus grand	${variable%%modele}

le modèle doit être conforme à ceux utilisés pour la substitution de noms de fichiers du shell utilisé

avec bash, il faut positionner l'option extglob par « shopt -s extglob » pour pouvoir utiliser les expressions complexes dans les modèles.

substitution de variables : exemples

```
f=/usr/local/Photos/
  reveDeVacancesderevealamontagne.jpg
$ echo ${f#/}
usr/local/Photos/
  reveDeVacancesderevealamontagne.jpg
$# suppression du premier / du nom
$ echo ${f#*/}
usr/local/Photos/
  reveDeVacancesderevealamontagne.jpg
$ # suppression du chemin
$ g=${f##*/}
$ echo $g
reveDeVacancesderevealamontagne.jpg
```

substitution de variables : exemples (2)

```
$# la variable f n'a pas été modifiée
$ echo $f
/usr/local/Photos/
  reveDeVacancesderevealamontagne.jpg
$# pour supprimer l'extension d'un fichier
$ echo ${g%.*}
reveDeVacancesderevealamontagne
$#obtenir le chemin
$ echo ${f%/*}
/usr/local/Photos
$ echo ${g%%ve*}
re
$ echo ${g%ve*}
reveDeVacancesdere
```

break et continue

break: permet de forcer la sortie d'une boucle for, while ou until. Il peut prendre un argument strictement supérieur à 1 afin de spécifier le nombre de boucles imbriquées dont il faut sortir. Si la valeur spécifiée est supérieure au nombre de boucles imbriquées, on sort de toutes les boucles imbriquées.

continue: Il permet de passer `a la prochaine it´eration d'une boucle for, while ou until. Le mot clef continue peut aussi prendre un argument strictement supérieur à 1 afin de spécifier le nombre d'itérations à "sauter". Si la valeur spécifiée est supérieure au nombre d'itérations possibles, la dernière itération est toutefois exécutée.

initialisation des paramètres positionnels avec set

set: suivie d'arguments permet d'affecter ces arguments aux paramètres positionnels

les caractères présents dans la variable IFS séparent les arguments

Exemples:

```
$ set a b c
$ echo $1
a
$set $(date)
$echo $2
dèc
```

tableaux

non traités dans ce cours

gestion avancée des entrées/sorties

rappel:

- &1, &2 désignent respectivement la valeur courante de l'entrée standard et de la sortie en erreur
- exemple d'utilisation: affichage sur la sortie en erreur:

```
echo mon message >2&
```

redirection de l'entrée standard

- « exec 0<fichier » :l'entrée standard de toutes les commandes suivantes du script liront leurs données depuis fichier

redirections avancées: suite

redirection de la sortie standard

- « exec 1>fichier » :la sortie standard de toutes les commandes suivantes du script ira dans le fichier fichier

redirection de la sortie en erreur

- « exec 2>fichier » :la sortie en erreur de toutes les commandes suivantes du script ira dans le fichier fichier

redirection des deux sorties d'un seul coup:

- « exec 1>fichier 2>&1 »

gestion de fichiers

manipulation de fichiers associés aux descripteurs 3 à 9

ouverture de fichier

- en lecture : exec descripteur<fichier
- en écriture: exec descripteur>fichier

fermeture d'un fichier:

- fichier ouvert en lecture : exec descripteur<&-
- fichier ouvert en écriture : exec descripteur>&-

lecture : commande <&descripteur

écriture: commande >&descripteur

traitement d'un fichier

commande read

le traitement d'un fichier associe while et read dans une boucle de la forme

```
while read var1 var2 var3 <descripteur
do
  commandes
done
```

traitement de fichers structurés

rappel: IFS :

- variable contenant les caractères séparant les mots entre eux
- par défaut: espace, tabulation et saut de ligne

si les lignes du fichier f contiennent des champs séparés par un caractère c, il suffit d'affecter c comme valeur à IFS pour que read en tienne compte

traitement de fichers structurés: exemple

```
$ cat traitementFichierStructure.sh
#! /bin/bash

IFS=":"
exec 3</etc/passwd
while read a b c d e f g h i j k l m <&3
do
  echo "$a a l'uid $c"
done
exec 3<&-
 $ ./traitementFichierStructure.sh |head
root a l'uid 0
daemon a l'uid 1
bin a l'uid 2
sys a l'uid 3
```

gestion de fichiers: exemples

```
$ cat /tmp/nombres.txt
1 3
6 4
2 -5
$ exec 3</tmp/nombres.txt
$ while read a b<&3
> do
>   echo $((a+b))
> done
4
10
-3
$ exec 3<&-
```

commande eval

permet de réaliser une double évaluation

- première passe: les caractères spéciaux sont traités
- seconde passe: la ligne résultante est évaluée

exemple:

```
a=123
$ b=a
$ eval echo \$$b
$ a=ls
$ eval $a
[liste des fichiers suite à
 l'exécution de ls]
```

Signaux: principaux signaux

nom du signal	No	cas usuel	comportement par défaut d'un processus le recevant	comporte-ment modi-fiable ?
HUP	1	perte de connexion	Mort	Oui
INT	2	Ctrl-C sur le clavier	Mort	Oui
KILL	9	processus qui ne répond pas à kill -TERM	Mort	Non
TERM	15	sus en lui laissant la chance de faire le ménage	Mort	Oui

gestion des signaux: TRAP

TRAP: permet d'ignorer un signal ou de modifier le comportement associé à un signal

utile pour garantir qu'un script fera le ménage avant de mourir élégamment

Syntaxe:

trap 'liste de commandes' signal1 [signal2 [signal3 ...]]

Exemples:

ignorer un signal: trap '' signal

gérer un signal: trap 'liste de commandes » signal

remettre le traitement par défaut: trap - signal

TRAP: exemple

```
#! /bin/bash
fichierTemp=/tmp/$0.$$
#en cas de signal, on fait le ménage:
trap 'rm -f "$fichierTemp";exit 1'\
  SIGHUP SIGINT SIGTERM
#reste du script qui peut
#potentiellement modifier fichierTemp
commande1
commande2
```

analyse des options de ligne de commande

2 méthodes:

- utiliser la commande interne getopts qui permet de gérer des options d'une lettre
- à la main : boucle while/case/shift (avantage: on peut utiliser des noms longs pour les options)

getopt

une option est composée d'une lettre précédée d'un + ou d'un -

une option peut être suivie d'un argument

exemple: getopt "abc:d" var

- les options possibles sont -a, -b, -c, -d
- le « : » après le « c » indique que l'option -c attend un paramètre
- var est le nom de la variable dans laquelle sera mis le nom de l'option lue
- l'argument éventuel sera mis dans la variable réservée OPTARG
- getopt s'utilise dans une boucle while

getopt

Exemple:

```
while getopt "ac:" option
do
  case "$option" in
    a) echo option a
       ;;
    c) echo option a avec l'argument
  "$OPTARG"
       ;;
    \?) echo option invalide
       ;;
  esac
done
```

getopt: gestion des erreurs

lorsque getopt rencontre une option invalide:

- la variable var contient le caractère ?
- un message est affiché à l'écran

si : est en première position dans la liste des options:

- aucun message ne sera affiché à l'écran
- OPTARG contiendra la valeur de l'option fautive
- exemple: getopt `":abc:d" var`

paramètre manquant:

- var contient : et OPTARG contient le caractère de

gestion artisanale des arguments

```
while [ $# -gt 0 ]; do
  case "$1" in
    -Prefix) shift
             [ $# -gt 0 ] || { affErr; exit 1; }
             Prefix="$1"
             shift
             ;;
    -remoteComputer) shift
            [ $# -gt 0 ] || { affErr; exit 1; }
            remoteComputer="$1"
            shift
            ;;
    *) repASauver="$1"
       shift
       ;;
  esac
done
```

gestion d'un processus en arrière plan: wait

WAIT p: attend la terminaison du processus dont le pid est p

WAIT: attend la terminaison de tous les processus lancés en arrière plan à partir du shell courant

Exemple: pour profiter d'une machine bipro:

```
gzip -9 fichier1&
gzip -9 fichier2&
wait
mv fichier1 fichier2 /archive/
```

commande find

find permet de chercher récursivement les fichiers vérifiant une ou plusieurs conditions

outre les expression simples, l'expression que doit vérifier un fichier peut être de la forme (par priorité décroissante) :

(expression)
! expression
expression1 -a expression2 : ET logique
expression1 expression2: ET logique
expression1 -o expression2: OU logique

commande find

expressions élémentaires à argument numérique:

+n: toutes les valeurs supérieures ou égales à n

-n: toutes les valeurs inférieures ou égales à n

n: n exactement

par la suite, partout où on verra un argument numérique n, on pourra utiliser +n, n ou -n

exemples:

-size 1024k: les fichiers de taille égale à 1024 Ko

-size -1024k: les fichiers de taille inférieure égale à 1024 Ko

-size +1024k: les fichiers de taille supérieur ou égale à 1024 Ko

commande find: quelques expressions élémentaires

quelques expressions élémentaires:

-name motifProtégé: les fichiers vérifiant le motif

-size n: les fichiers de taille n

-mtime n, -ctime n, -atime n

-perm p avec p ayant la forme numérique ou symbolique des arguments de chmod

-type c avec c=b,c,d (dossier),l (lien symbolique),p,f (fichier ordinaire),s

-user u

-group g

-link n : nombre de liens physique sur le fichier

-print: provoque l'affichage des noms des fichiers vérifiant l'expression (par défaut sur le Gnu find)

...

commande find: -exec

-exec commande;

pour chaque fichier trouvé, la commande est exécutée.

si {}apparaît parmi les arguments de la commande, il est remplacé par le nom du fichier trouvé

-exec commande arguments {} +

syntaxe POSIX/SUSv3 (standard mais pas disponible sur toutes les plateformes)

les noms des fichiers trouvés sont accumulés dans une liste l

la commande est exécutée une seule fois, à la fin de la recherche et {}est remplacé par la liste des arguments

commande find : exemples

fichiers ordinaire nommés core de plus de 1024Ko

```
find -size +1024k -type f -name core -print
```

fichier ordinaires de l'utilisateur petit ou fichiers ordinaires de taille supérieure à 1024 Ko et de nom core

```
find -type f  \( -user petit -o \( -size +1024k -name core \) \) -print
```

fichier dont le nom commence par C

```
find -name c\* -print
```

commande find: -exec et les espaces (&Co)

rm est une commande qui ne lit pas sur son entrée standard. 3 méthodes pour effacer un ensemble de fichiers sélectionnés par find :

on lance un rm par fichier (lourd)

```
find -name \*.bak -exec rm -f {} \;
```

syntaxe POSIX: un seul rm global est lancé :

```
find -name \*.bak -exec rm -f {} \+
```

une solution avec les options spécifiques de Gnu find:

```
find -name \*.bak -print0 |xargs -0 rm -f
```

ne marche pas avec les noms de fichiers contenant des caractères à problème (espace, saut de ligne, ', etc.):

```
find -name \*.bak -print |xargs rm -f
```

commande xargs

xargs [options] commande

xargs rassemble ce qu'elle reçoit sur son entrée standard dans une liste l et exécute « commande l »

xargs s'utilise avec des commandes qui n'acceptent pas de données sur leur entrée standard

Exemple:

```
grep -l perso * | xargs chmod 700
```

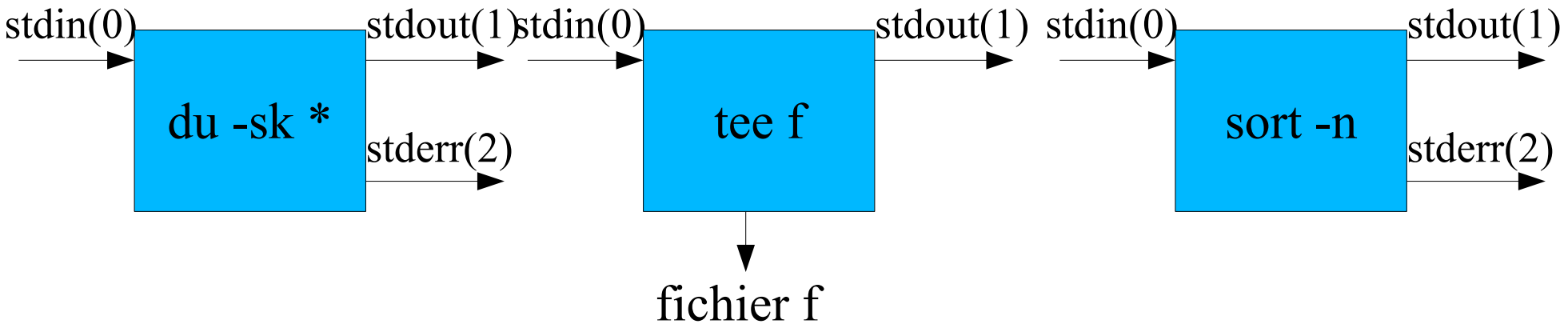
xargs : options utiles

-p: prompt mode. une confirmation est demandée à l'utilisateur pour chaque invocation de la commande

si le nombre ou la taille des arguments transmis via l'entrée standard est important, il est possible d'indiquer à xargs d'exécuter plusieurs fois la commande avec une liste limitée:

- -n nombre: la commande est invoquée plusieurs fois et chaque invocation a au plus nombre arguments
- -s taille: la commande est invoquée plusieurs fois et chaque invocation fait au plus s octets

commande tee



la commande tee envoie simultanément son entrée standard vers un fichier et vers sa sortie standard.

options:

-a: ajoute au fichier

-i : ignore le signal