Système d'exploitation

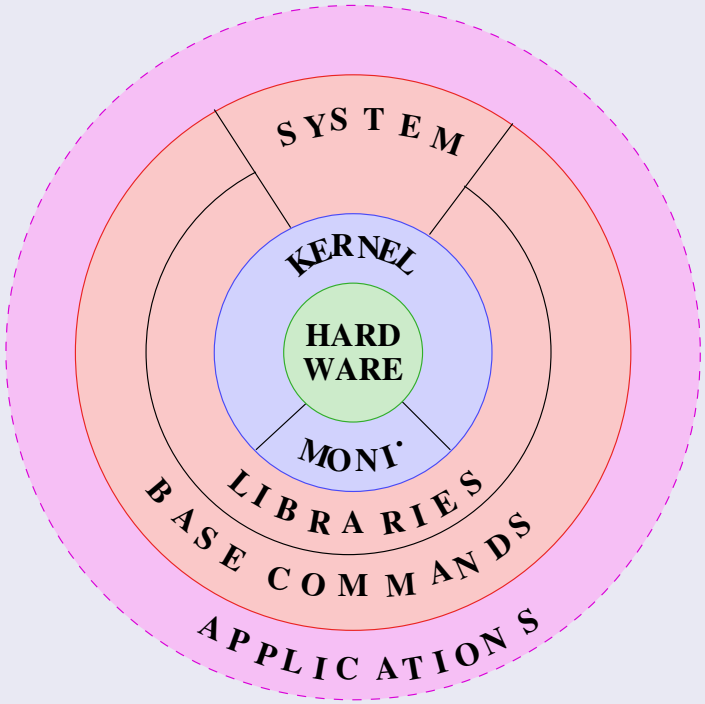
Partie 1: Shell

Le 30 août 2022, SVN-ID 425

30 août 2022

Table des matières

Organisation



matériel CPU, RAM, contrôleurs et périphériques.

moniteur Petit programme en ROM, qui tourne au démarrage de la machine.

noyau Gère et donne accès au matériel

système Couche de standardisation

applications

Table des matières

UID & GID (User & Group IDentifier)

Le système de protection est basé sur les identifiants d'utilisateurs UID et de groupes GID. Ce sont des entiers, des tables permettent de les convertir en un nom humainement compréhensible.

utilisateur toute personne travaillant sur une machine a ouvert une session

$\Longrightarrow$ 1 UID, 1 GID principal et 0 ou plusieurs GID auxiliaires.

- 1 UID identifie un utilisateur.
- 2 utilisateurs peuvent appartenir à un même groupe.

programme un programme est lancé par un utilisateur

$\Longrightarrow$ 1 UID, 1 GID principal et 0 ou plusieurs GID auxiliaires.

fichier il appartient à 1 seul utilisateur et à un seul groupe.

création de fichier il appartient à l'UID et au GID principal de l'utilisateur (programme) qui l'a créé.

Droits d'un fichiers

droits d'accès				propriétaire et groupe	
masque octal	format usuel				
	prop.	group	autre	UID	GID
777	rwx	rwx	rwx	101	100
600	rw-	---	---	110	200
700	---	rw-	---	110	200
644	rw-	r--	r--	0	0
755	rwx	r-x	r-x	0	0

Pour un fichier non répertoire

- r : accès en lecture
- w : accès en écriture
- x : il est possible d'essayer de le lancer

Pour un fichier répertoire

- r : les noms des fichiers du répertoire peuvent être lus
- w : un fichier du répertoire peut être créé ou détruit
- x : les fichiers du répertoire peuvent être accédés

Droits d'un fichiers

droits d'accès				propriétaire et groupe	
masque octal	format usuel				
	prop.	group	autre	UID	GID
777	rwx	rwx	rwx	101	100
600	rw-	---	---	110	200
700	---	rw-	---	110	200
644	rw-	r--	r--	0	0
755	rwx	r-x	r-x	0	0

Pour un fichier non répertoire

- r : accès en lecture
- w : accès en écriture
- x : il est possible d'essayer de le lancer

Pour un fichier répertoire

- r : les noms des fichiers du répertoire peuvent être lus
- w : un fichier du répertoire peut être créé ou détruit
- x : les fichiers du répertoire peuvent être accédés

Droits d'un fichiers

droits d'accès				propriétaire et groupe	
masque octal	format usuel				
	prop.	group	autre	UID	GID
777	rwx	rwx	rwx	101	100
600	rw-	---	---	110	200
700	---	rw-	---	110	200
644	rw-	r--	r--	0	0
755	rwx	r-x	r-x	0	0

Pour un fichier non répertoire

- r : accès en lecture
- w : accès en écriture
- x : il est possible d'essayer de le lancer

Pour un fichier répertoire

- r : les noms des fichiers du répertoire peuvent être lus
- w : un fichier du répertoire peut être créé ou détruit
- x : les fichiers du répertoire peuvent être accédés

Changement des droits

chmod masque-octal f1 f2 ...

- chmod 755 tutu
- chmod 640 titi

chmod [ugoa] [+-=] [rwx] f1 f2 ...

- chmod a-x tutu
- chmod go+rx titi toto

Exercice

Le répertoire D contient le fichier F.

mon		répertoire D			fichier F			accès	
uid	gid	uid	gid	masque	uid	gid	masque	r	w
*	*	*	*	755	*	*	666	*	*
10	20	10	21	6**	11	20	6**		
10	20	10	21	5**	11	20	2**		*
10	20	10	21	1**	11	20	6**	*	*
10	20	11	20	*6*	11	20	*6*		
10	20	11	20	*5*	11	20	*2*		*
10	20	11	20	*1*	11	20	*4*	*	
10	20	11	21	**5	10	20	6**	*	*
10	20	11	21	**0	10	20	6**		
10	20	11	21	**6	10	20	6**		

Exercice

Le répertoire D contient le fichier F.

mon		répertoire D			fichier F			accès	
uid	gid	uid	gid	masque	uid	gid	masque	r	w
*	*	*	*	755	*	*	666	*	*
10	20	10	21	6**	11	20	6**		
10	20	10	21	5**	11	20	2**		*
10	20	10	21	1**	11	20	6**	*	*
10	20	11	20	*6*	11	20	*6*		
10	20	11	20	*5*	11	20	*2*		*
10	20	11	20	*1*	11	20	*4*	*	
10	20	11	21	**5	10	20	6**	*	*
10	20	11	21	**0	10	20	6**		
10	20	11	21	**6	10	20	6**		

Exercice

Le répertoire D contient le fichier F.

mon		répertoire D			fichier F			accès	
uid	gid	uid	gid	masque	uid	gid	masque	r	w
*	*	*	*	755	*	*	666	*	*
10	20	10	21	6**	11	20	6**		
10	20	10	21	5**	11	20	2**		*
10	20	10	21	1**	11	20	6**	*	*
10	20	11	20	*6*	11	20	*6*		
10	20	11	20	*5*	11	20	*2*		*
10	20	11	20	*1*	11	20	*4*	*	
10	20	11	21	**5	10	20	6**	*	*
10	20	11	21	**0	10	20	6**		
10	20	11	21	**6	10	20	6**		

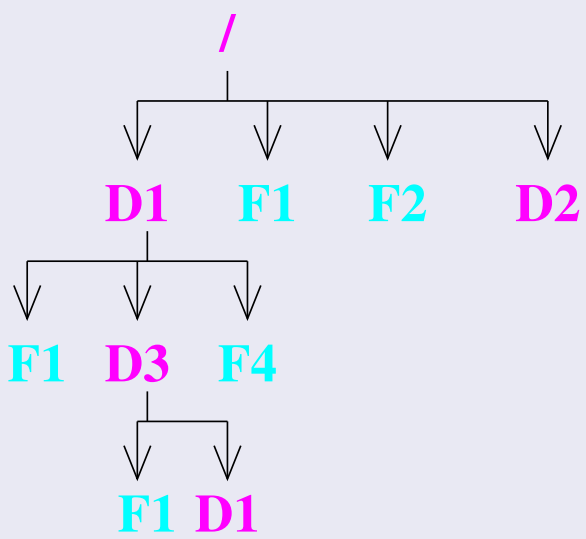
Exercice

Le répertoire D contient le fichier F.

mon		répertoire D			fichier F			accès	
uid	gid	uid	gid	masque	uid	gid	masque	r	w
*	*	*	*	755	*	*	666	*	*
10	20	10	21	6**	11	20	6**		
10	20	10	21	5**	11	20	2**		*
10	20	10	21	1**	11	20	6**	*	*
10	20	11	20	*6*	11	20	*6*		
10	20	11	20	*5*	11	20	*2*		*
10	20	11	20	*1*	11	20	*4*	*	
10	20	11	21	**5	10	20	6**	*	*
10	20	11	21	**0	10	20	6**		
10	20	11	21	**6	10	20	6**		

Table des matières

Définition

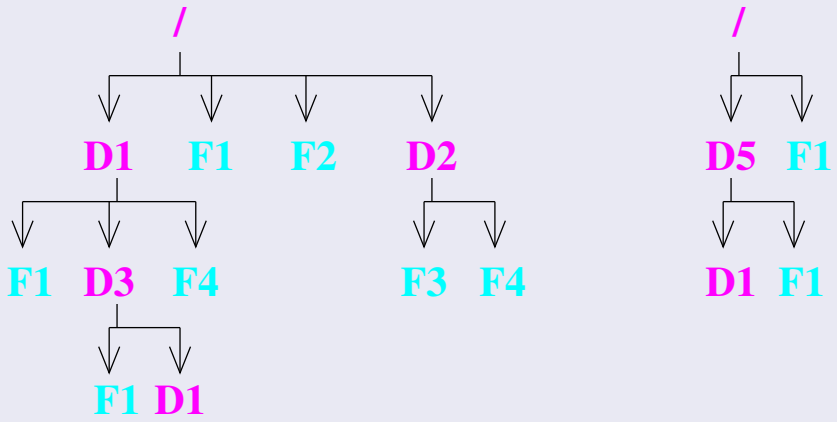


Arborescence de fichiers :

- les noeuds : fichiers répertoires
- les feuilles : fichiers ou répertoires vides
- la racine le haut de l'arbre

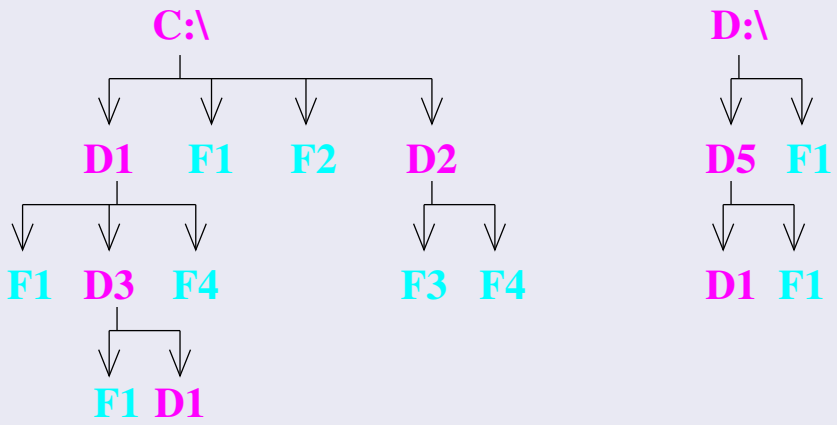
Support physique : disques durs, RAM

Plusieurs systèmes de fichiers (Windows)



Chaque disque et/ou partition a un système de fichiers
$\Longrightarrow$ nombreux systèmes de fichiers.

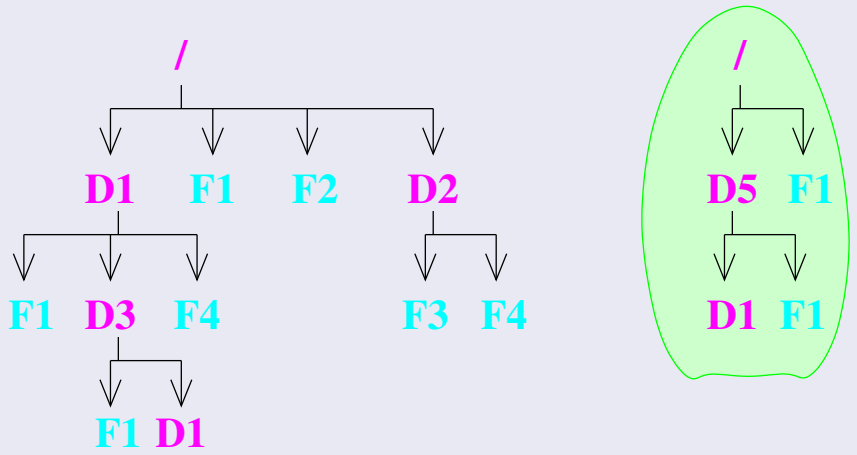
Plusieurs systèmes de fichiers (Windows)



Sous Windows, les systèmes de fichiers sont visibles

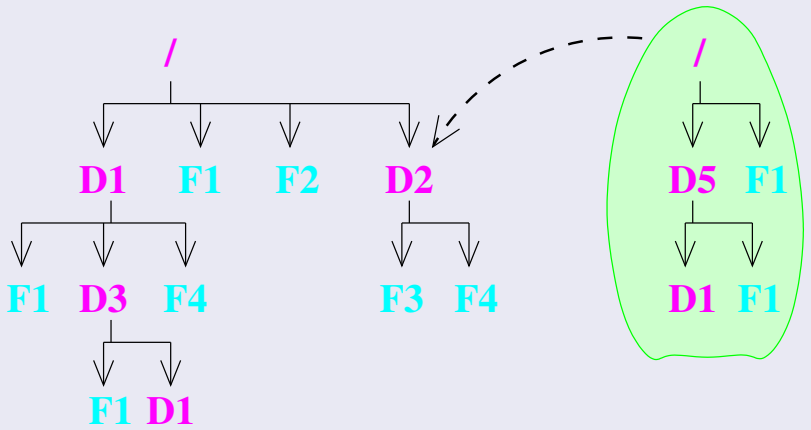
$\Longrightarrow$ identifiés par une lettre suivie de ' :'.

Plusieurs systèmes de fichiers (Unix)



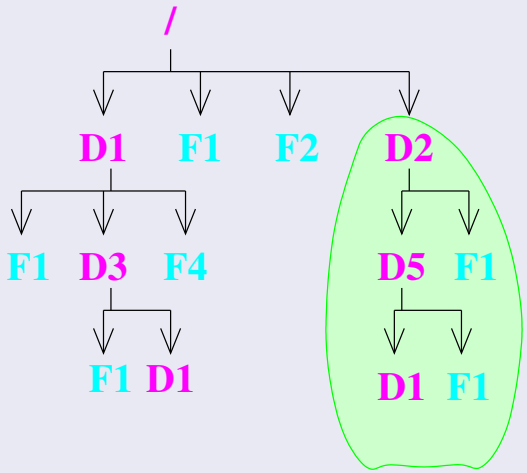
Chaque disque et/ou partition a un système de fichiers
$\Longrightarrow$ nombreux systèmes de fichiers.

Plusieurs systèmes de fichiers (Unix)



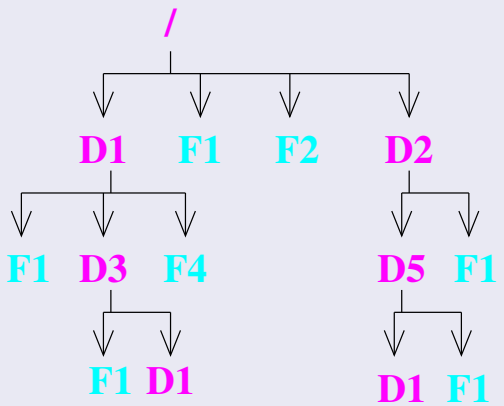
Un système de fichiers principal sur le quel sont montés les systèmes de fichiers auxiliaires $\Longrightarrow$ un seul système de fichiers.

Plusieurs systèmes de fichiers (Unix)



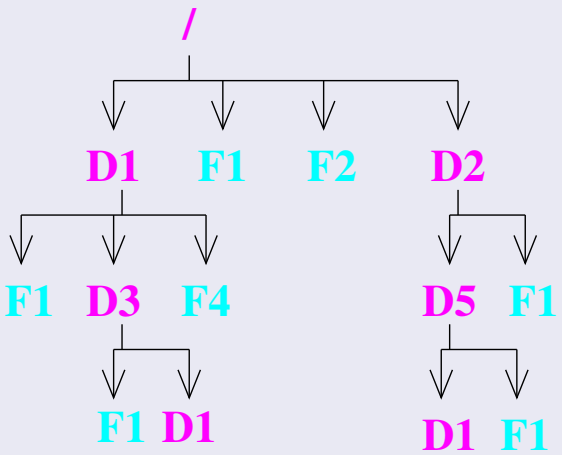
Un système de fichiers principal sur le quel sont montés les systèmes de fichiers auxiliaires $\Longrightarrow$ un seul système de fichiers.

Plusieurs systèmes de fichiers (Unix)



Un système de fichiers principal sur le quel sont montés les systèmes de fichiers auxiliaires $\Longrightarrow$ un seul système de fichiers.

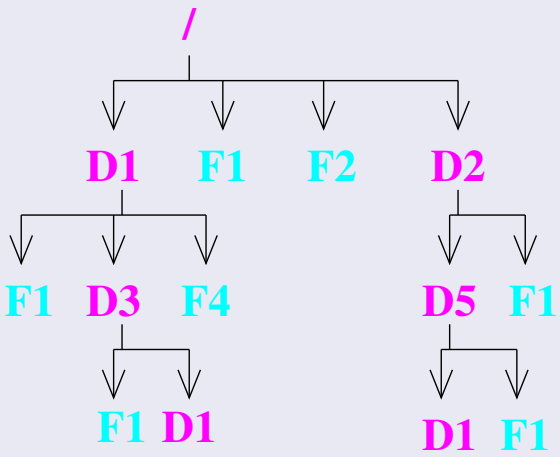
Fichiers . & ..

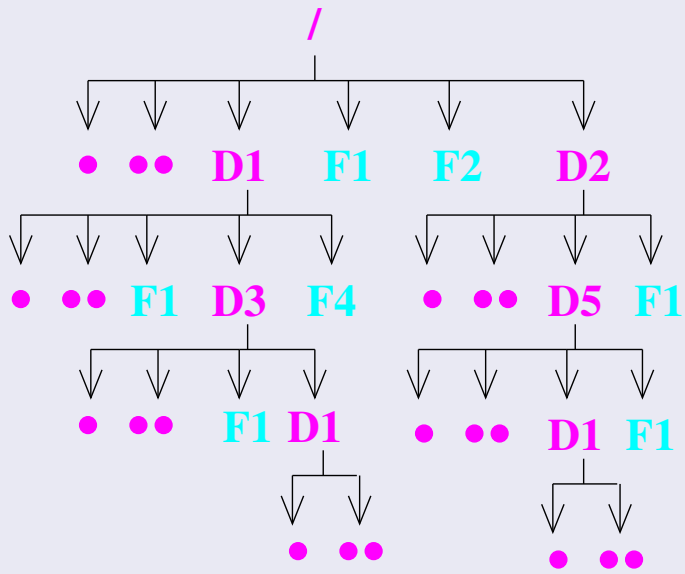


Tout répertoire contient au moins 2 fichiers :

- . alias du répertoire courant.
- .. alias du répertoire supérieur.
- .. alias du répertoire courant pour la racine.

Fichiers . & ..





Tout répertoire contient au moins 2 fichiers :

- • alias du répertoire courant.
- •• alias du répertoire supérieur.
- •• alias du répertoire courant pour la racine.

Types de fichiers

répertoire

opérations disponibles : création et suppression de fichiers.

non répertoire

opérations disponibles : lecture, écriture, positionnement du pointeur (optionnel).

régulier ils sont soit binaire, soit texte, ils ont une fin et le positionnement est disponible.

spécial ils sont associés à des périphériques et/ou à des drivers (terminal, disque dur, flux vidéo audio, ...).

lien leurs contenus sont la référence d'un autre fichier.

- Lire/écrire un fichier lien revient à lire/écrire le fichier référencé.
- Un lien peut référencer un autre lien.
- Un lien mort : le référencé n'existe pas.

Types de fichiers

Quelques fichiers spéciaux :

/dev/sda

le premier disque dur, il a une fin et le positionnement du pointeur est disponible.

FIFO

Fichier ou tout ce qui est écrit ne peut être lu qu'une seule fois. Il est créé avec la commande mkfifo.

/dev/ttyS0, /dev/pts/0

liaison série physique (RS232) ou émulée (terminal), pas de fin, pas de positionnement, configurable.

Types de fichiers

Quelques fichiers spéciaux :

/dev/null

fichier poubelle, capacité infinie.

/dev/zero

fichier sans fin contenant que des octets nuls (0x00).

/dev/random /dev/urandom

fichier sans fin de nombres aléatoires.

Définition

Suite de noms (nom=chaine de caractères sans '/') séparés par au moins un '/' et précédés et terminés par $\emptyset$ ou plusieurs '/' :

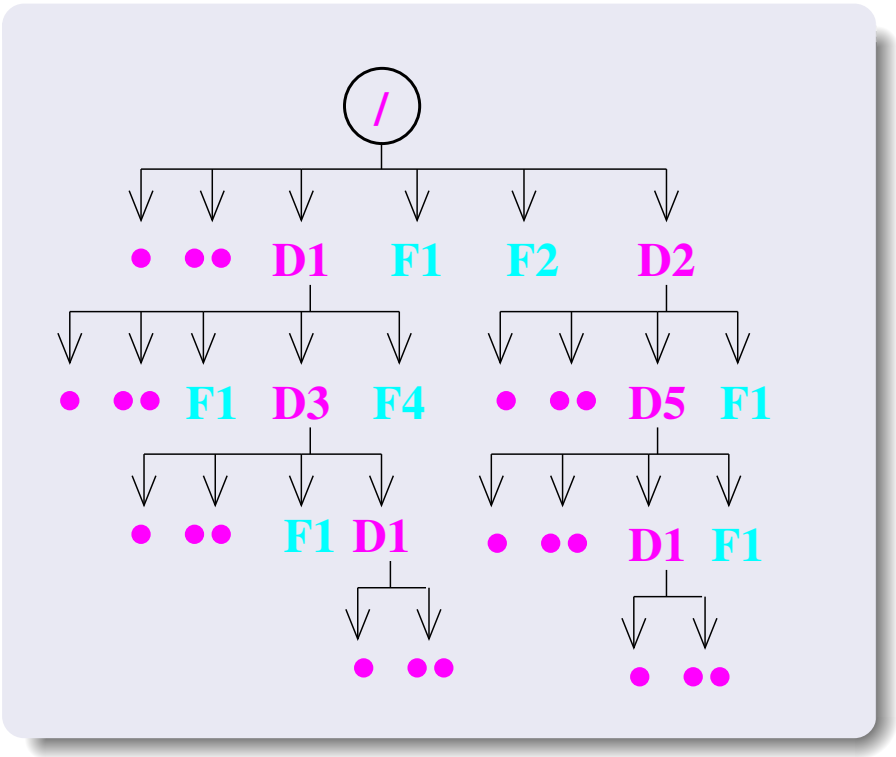
$[/]nom_0/nom_1/.../nom_n[/]$

Un chemin est valide si

- les sous chemins "$[/].../nom_i$" pour $i < n$ doivent être des répertoires de "$[/].../nom_{i-1}$.
- nom_n doit être un fichier ou un répertoire du répertoire $[/]nom_0/nom_1/.../nom_{n-1}$.
- si le chemin se termine par '/', nom_n doit être un répertoire.

Chemins absolus

Un chemin absolu commence par un '/', on part de la racine du système de fichier.
Parmi les chemins ci-dessous indiquez ceux qui sont identiques et ceux qui n'en sont pas.

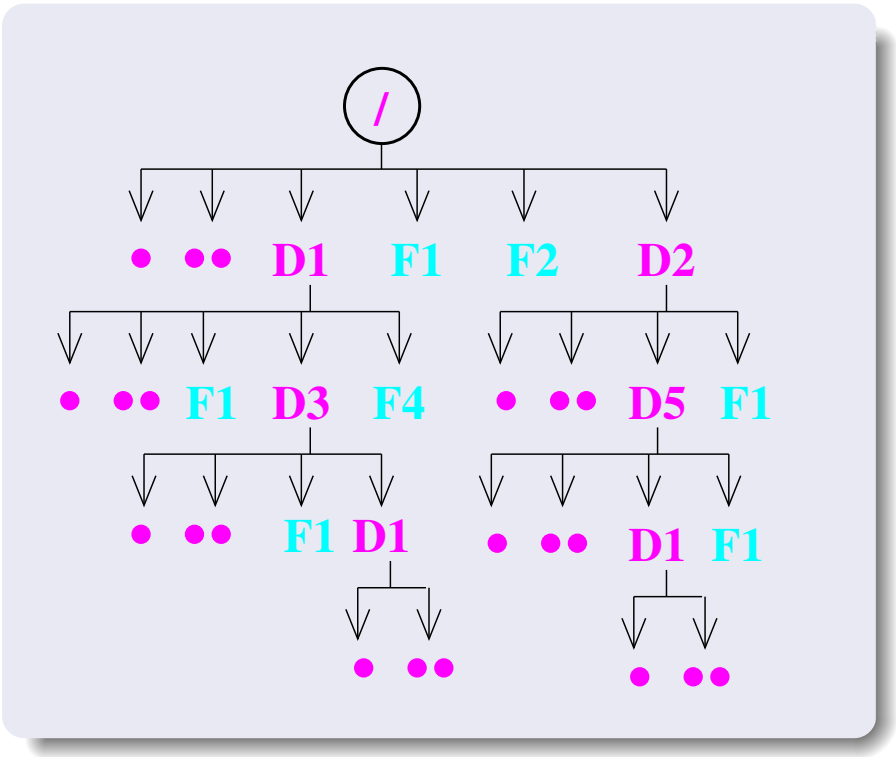


/D1/D3/D1
/D1/D3/D1/
/D1/D3/D1/.
/D1/D3/D1.
///D1//D3////D1

/D2/D5/D1/../../F1
/D2/D5/D1/../../F1/
/../D2/D5//.//D1/../../F1
/D2/D5/D1/../XX/../../F1

Chemins absolus

Un chemin absolu commence par un '/', on part de la racine du système de fichier.
Parmi les chemins ci-dessous indiquez ceux qui sont identiques et ceux qui n'en sont pas.

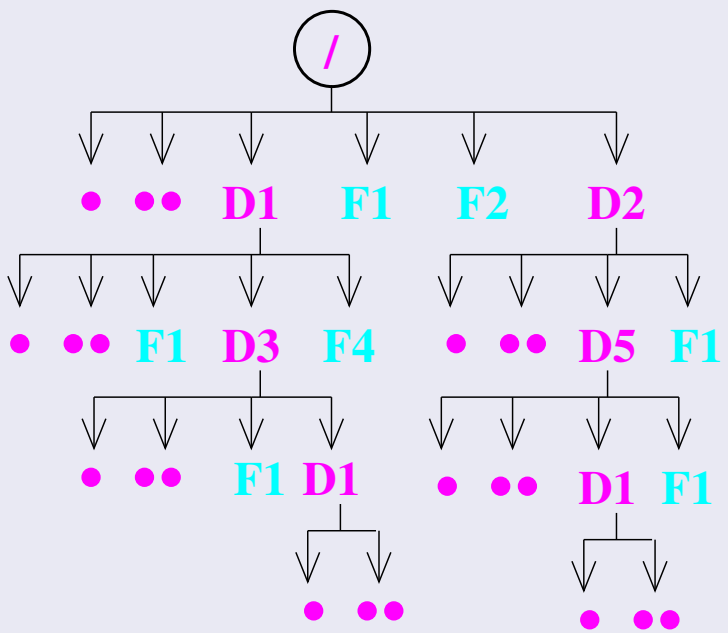


/D1/D3/D1
/D1/D3/D1/
/D1/D3/D1/.
~~/D1/D3/D1.~~
///D1//D3///D1

/D2/D5/D1/../../F1
/D2/D5/D1/../../F1/
/../D2/D5//.//D1/../../F1
/D2/D5/D1/../XX/../../F1

Chemins absolus

Un chemin absolu commence par un '/', on part de la racine du système de fichier.
Parmi les chemins ci-dessous indiquez ceux qui sont identiques et ceux qui n'en sont pas.

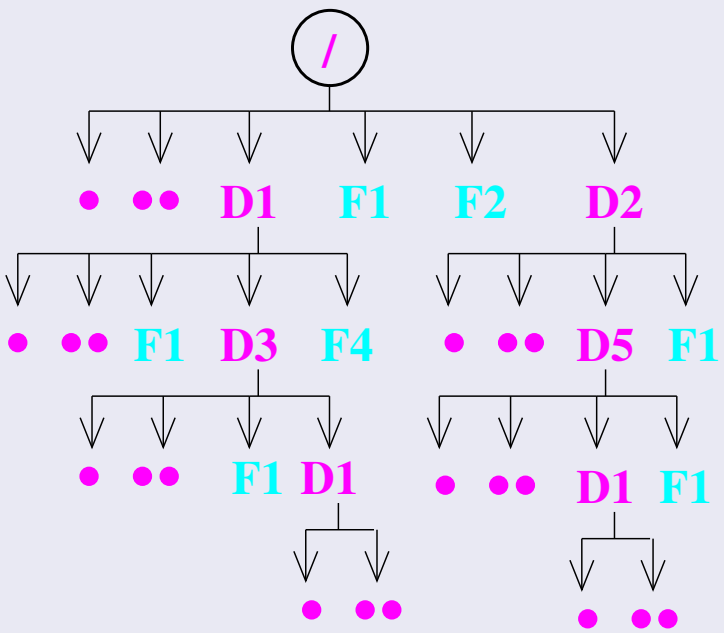


/D1/D3/D1
/D1/D3/D1/
/D1/D3/D1/.
///D1//D3////D1

/D2/D5/D1/../../F1
/D2/D5/D1/../../F1/
/../D2/D5//.//D1/../../F1
/D2/D5/D1/../XX/../../F1

Chemins absolus

Un chemin absolu commence par un '/', on part de la racine du système de fichier.
Parmi les chemins ci-dessous indiquez ceux qui sont identiques et ceux qui n'en sont pas.



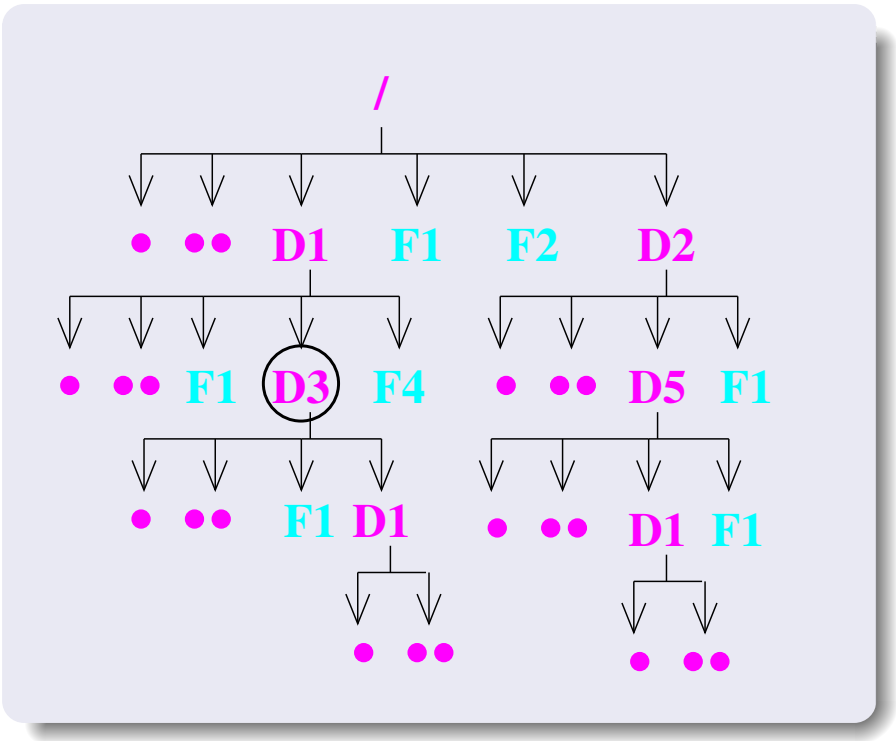
/D1/D3/D1
/D1/D3/D1/
/D1/D3/D1/.
///D1//D3////D1

/D2/D5/D1/../../F1
~~/D1/D2/D5/D1/../../F1/~~
/../D2/D5//.//D1/../../F1
~~/D1/D2/D5/D1/../XX/../../F1~~

Chemins relatifs

Tout programme qui tourne a un répertoire de travail associé qui s'appelle CWD ou WD (Current Working Directory).
Un chemin relatif ne commence pas par un '/', il part du répertoire CWD.

Le chemin absolu du chemin relatif CHE est : CWD/CHE



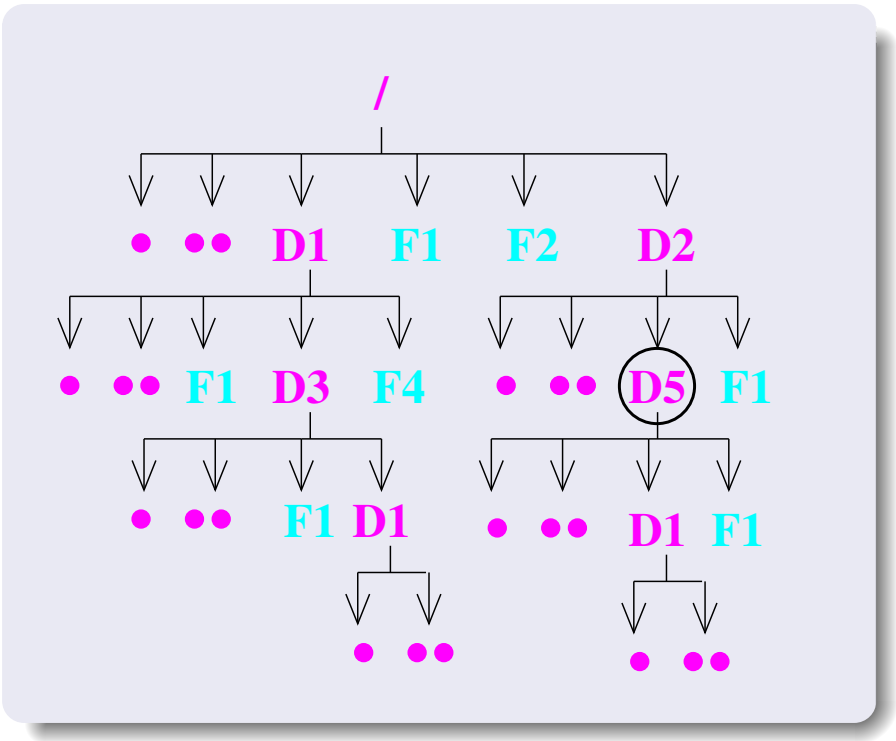
Donnez les chemins relatifs de :

- /D2/F1 : ../../D2/F1
- /F1 : ../../F1
- /D1/D3/F1 via D1 : ../D3/F1 ou ../../D1/D3/F1
- /D1/D3/F1 le plus court ne commençant pas par 'F' : ./F1

Chemins relatifs

Tout programme qui tourne a un répertoire de travail associé qui s'appelle CWD ou WD (Current Working Directory).
Un chemin relatif ne commence pas par un '/', il part du répertoire CWD.

Le chemin absolu du chemin relatif CHE est : CWD/CHE



Donnez les chemins relatifs de :

- /D2/F1 : ../F1
- /F1 : ../../F1
- /D1/D3/F1 : ../../D1/D3/F1
- /D2/D5/F1 le plus court ne commençant pas par 'F' : ./F1

Quizz

1. Il existe un chemin absolu pour chaque fichier.
2. Il existe un chemin absolu unique pour chaque fichier.
3. Soit un CWD et un fichier, il existe toujours un chemin relatif partant de ce CWD et désignant ce fichier.
4. Les chemins "F" et "./F" donnent toujours le même fichier.
5. Les chemins "../F" et "F" donnent toujours des fichiers différents.
6. Les chemins "D1/F" et "D1/D2/../F" donnent toujours le même fichier.
7. Les chemins "/F" et "F" donnent toujours des fichiers différents.

Table des matières

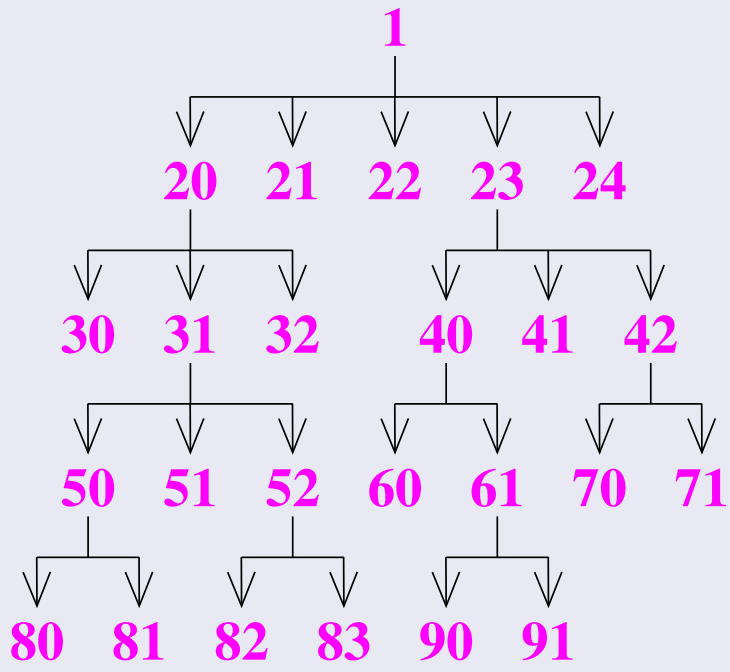
Définition

- Entité d'exécution $\Longrightarrow$ un programme qui tourne.
- Identifié par un numéro PID.
- Ils sont organisés en arbre.
- Ils sont groupés en session.
- Ils sont groupés en groupe terminal avec un leader.

Comme tout programme, il produit des sorties en fonction d'entrées.

Définition

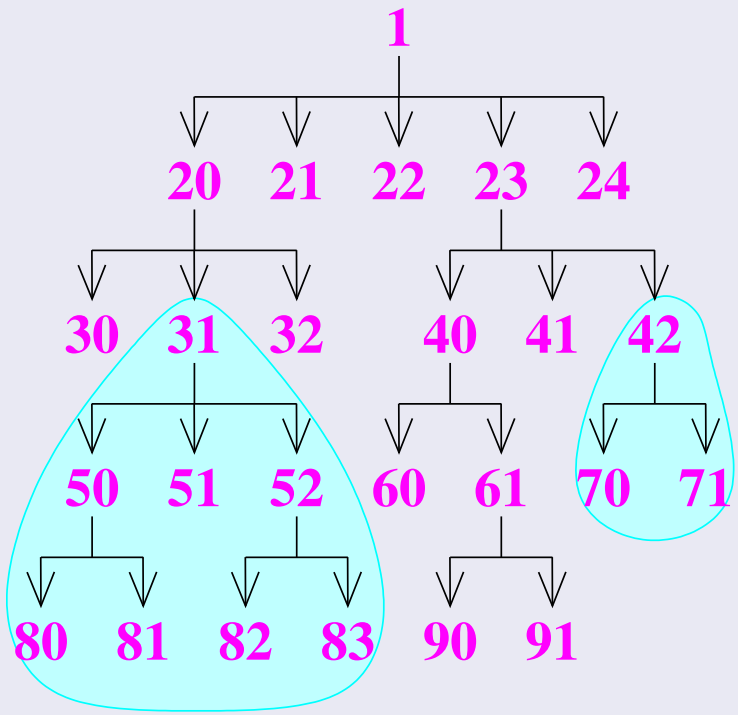
- Entité d'exécution $\Longrightarrow$ un programme qui tourne.
- Identifié par un numéro PID.
- Ils sont organisés en arbre.
- Ils sont groupés en session.
- Ils sont groupés en groupe terminal avec un leader.



Comme tout programme, il produit des sorties en fonction d'entrées.

Définition

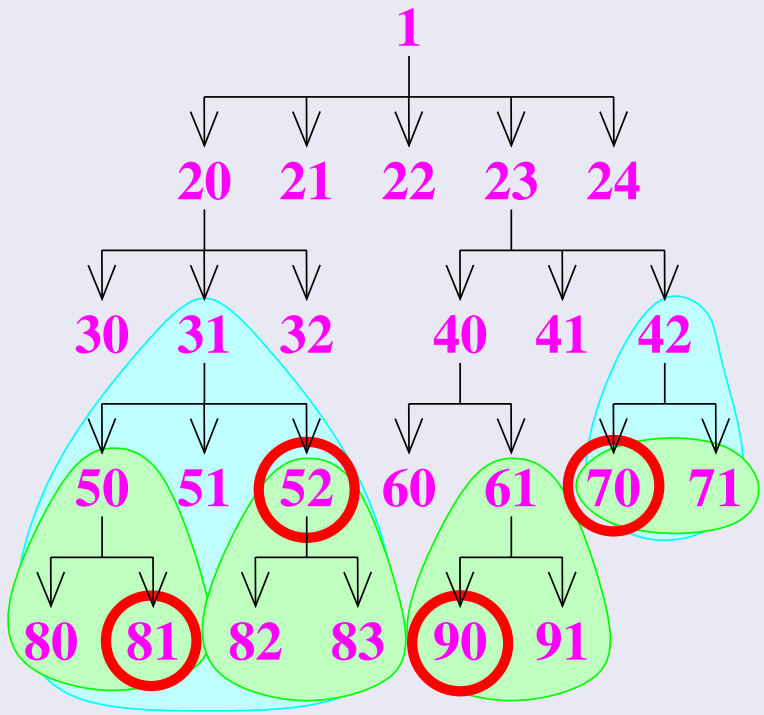
- Entité d'exécution $\Longrightarrow$ un programme qui tourne.
- Identifié par un numéro PID.
- Ils sont organisés en arbre.
- Ils sont groupés en session.
- Ils sont groupés en groupe terminal avec un leader.



Comme tout programme, il produit des sorties en fonction d'entrées.

Définition

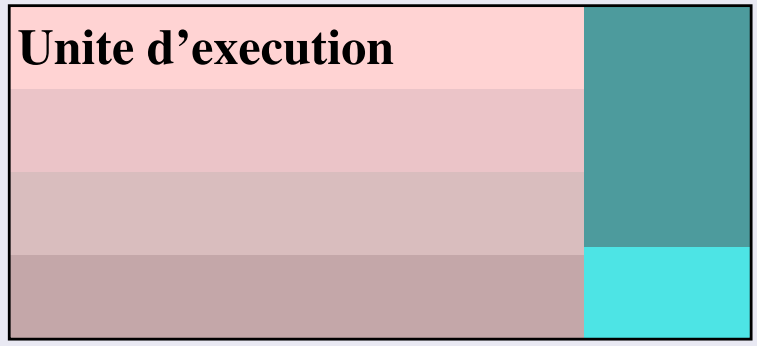
- Entité d'exécution $\Longrightarrow$ un programme qui tourne.
- Identifié par un numéro PID.
- Ils sont organisés en arbre.
- Ils sont groupés en session.
- Ils sont groupés en groupe terminal avec un leader.



Comme tout programme, il produit des sorties en fonction d'entrées.

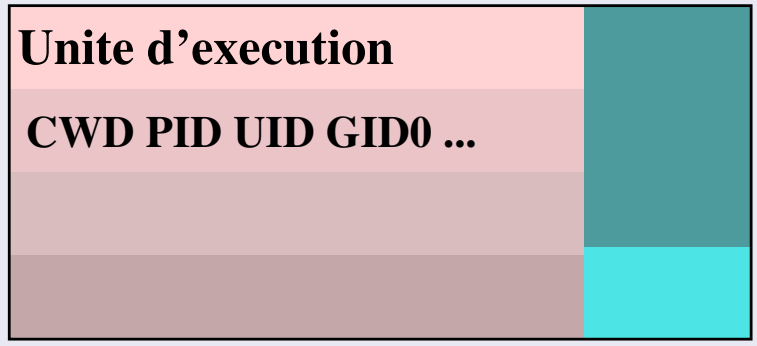
Entrées/sorties

- Unité d'exécution : code, données, pile.
- Identifiant de processus.
- Identifiant d'utilisateur et identifiants de groupe.
- Identifiant de session et de groupe terminal.
- Arguments : tableau de chaines de caractères.
- variables d'environnement : tableau de chaines de caractères
  nom-var=valeur-var
- flux d'entrée et de sortie.
- statut : valeur entre 0 et 255.



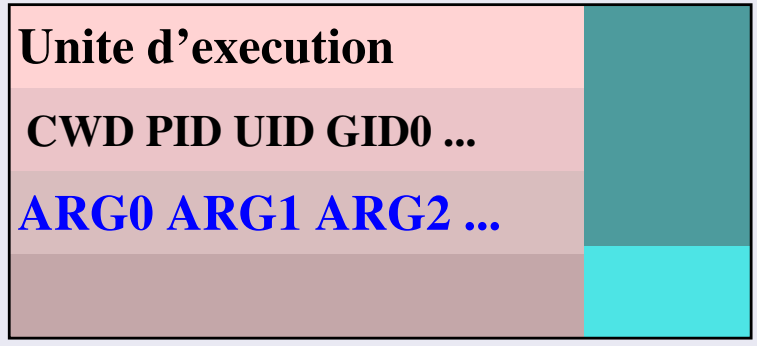
Entrées/sorties

- Unité d'exécution : code, données, pile.
- Identifiant de processus.
- Identifiant d'utilisateur et identifiants de groupe.
- Identifiant de session et de groupe terminal.
- Arguments : tableau de chaines de caractères.
- variables d'environnement : tableau de chaines de caractères
  nom-var=valeur-var
- flux d'entrée et de sortie.
- statut : valeur entre 0 et 255.



Entrées/sorties

- Unité d'exécution : code, données, pile.
- Identifiant de processus.
- Identifiant d'utilisateur et identifiants de groupe.
- Identifiant de session et de groupe terminal.
- Arguments : tableau de chaines de caractères.
- variables d'environnement : tableau de chaines de caractères
  nom-var=valeur-var
- flux d'entrée et de sortie.
- statut : valeur entre 0 et 255.



Entrées/sorties

- Unité d'exécution : code, données, pile.
- Identifiant de processus.
- Identifiant d'utilisateur et identifiants de groupe.
- Identifiant de session et de groupe terminal.
- Arguments : tableau de chaines de caractères.
- variables d'environnement : tableau de chaines de caractères
  nom-var=valeur-var
- flux d'entrée et de sortie.
- statut : valeur entre 0 et 255.

Entrées/sorties

- Unité d'exécution : code, données, pile.
- Identifiant de processus.
- Identifiant d'utilisateur et identifiants de groupe.
- Identifiant de session et de groupe terminal.
- Arguments : tableau de chaines de caractères.
- variables d'environnement : tableau de chaines de caractères
  nom-var=valeur-var
- flux d'entrée et de sortie.
- statut : valeur entre 0 et 255.

Entrées/sorties

- Unité d'exécution : code, données, pile.
- Identifiant de processus.
- Identifiant d'utilisateur et identifiants de groupe.
- Identifiant de session et de groupe terminal.
- Arguments : tableau de chaines de caractères.
- variables d'environnement : tableau de chaines de caractères
  nom-var=valeur-var
- flux d'entrée et de sortie.
- statut : valeur entre 0 et 255.

Convention : Arguments

Les arguments passés à un programme sont un tableau de chaine de caractères. La seule convention est :

Le premier argument est le nom d'invocation du programme

Le premier argument est donc souvent inutilisé mais est utile pour :

1. Écrire des messages d'erreur avec le nom du programme.
2. Retrouver le répertoire d'installation du programme.
3. Écrire un seul programme qui en fonction du nom d'invocation fait des choses différentes (ex : busybox)
   factorisation de code $\Longrightarrow$ gain de place

Convention : Variables d'environnement

Le format (convention) des variables d'environnement passées à un programme est :

nom-var=valeur-var Quelques variables d'environnement conventionnelles :

HOME Sa valeur est le chemin absolu du répertoire de travail de l'utilisateur UID.

TERM Sa valeur est le type de terminal (linux, xterm, vt100, ...).

LANG Sa valeur donne le langage de l'utilisateur et le charset utilisé.

PATH Sa valeur est une suite de chemins séparés par le caractère ' :' (ex : PATH=/bin :/usr/bin :/usr/local/bin).
Si on lance un programme par un chemin sans '/' (ex : gnu) alors le système lancera le premier fichier gnu exécutable trouvé dans la suite de répertoires du PATH (ex : soit les chemins /bin/gnu, /usr/bin/gnu, /usr/local/bin/gnu).

Convention : Flux d'entrée/sortie

Un programme qui démarre dispose de 3 flux d'entrée/sortie :

Flux 0

flux standard d'entrée, ouvert en lecture, abréviation stdin (libc) :

$\Longrightarrow$ dédié à l'acquisition de données d'entrée.

Flux 1

flux standard de sortie, ouvert en écriture, abréviation stdout (libc) :

$\Longrightarrow$ dédié à l'écriture de données de sortie.

Flux 2

flux standard d'erreur, ouvert en écriture, abréviation stderr (libc) :

$\Longrightarrow$ dédié à l'écriture de message d'erreur.

Statut

La valeur renvoyée par un programme ($0 \leq \textit{valeur} \leq 255$) indique si le programme s'est déroulé sans problème.
Elle peut être récupérée par l'entité qui a lancé le programme.

$$\text{statut} = 0 \Longrightarrow \text{ok},$$
$$\text{statut} \neq 0 \Longrightarrow \text{erreur}.$$

Table des matières

Fonction

Le Shell est un programme dont les fonctions sont :

Shell interactif C'est l'interface utilisateur standard d'Unix.

- Dès qu'un utilisateur ouvre un terminal, il discute avec un Shell.
- Il permet de lancer des commandes simple ou avec des d'options.
- Il permet de chainer des commandes de façon très souple.
- Il permet de taper très rapidement grâce à ses mécanismes d'expansion, la complétion et le rappel de commandes.

Shell script C'est un langage de programmation complet (variables, alternatives, boucles) dédié à l'écriture de programme système. Ces programmes lancent et/ou chainent d'autres programmes de manière automatique.

Historique

1977 sh (Bourne shell)

1978 csh (C shell)

1981 tcsh (C shell)

1983 ksh (Kom shell)

1988 bash (Bourne again shell)

1990 ash (réécriture du Bourne shell)

1990 zsh

A part csh et tcsh qui ont divergé, les autres sont compatibles avec le Bourne shell $\Longrightarrow$ scripts écrits il y a 40 ans fonctionnent encore.
bash et zsh sont très confortables et très semblables.

Syntaxe

En avant plan (fg)

```
sh> chmod 644 gnu/bee <C-R>
sh>
```

En arrière plan (bg)

```
sh> sleep 60 & <C-R>
sh>
```

avec

sh> l'invite de commande (prompt) écrite par le Shell.

sleep 60 &

chmod 644 gnu/bee la commande tapée par l'utilisateur. C'est une suite de chaines de caractères séparées par des espaces (séquence d'au moins un blanc ou tabulation).

- **chmod** et **sleep** la première chaine de caractères est le programme à exécuter (chemin relatif ou absolu ou basename avec le PATH).
- **644** et **60** 1^{er} argument, sa signification dépend de la commande.
- **gnu/bee** 2^{nd} argument, sa signification dépend de la commande.

& en fin de commande indique l'arrière plan

<C-R> La touche entrée tapée par l'utilisateur. Elle indique au Shell que la commande est complète et qu'elle doit être exécutée.

Job contrôle

<CTL-C>	demande au processus en avant plan de se terminer.
<CTL-Z>	demande au processus en arrière plan de suspendre son exécution.
fg	• fait passer en avant plan le dernier processus lancer en arrière plan. • si ce processus était suspendu, il le relance.
bg	relance le processus suspendu en arrière plan

Raccourcis

Sur l'invite de commande du Shell on peut :

- Avec $\uparrow$ et $\downarrow$ se déplacer dans l'historique des commandes déjà exécutées.
- Avec $\leftarrow$ et $\rightarrow$ se déplacer dans la commande rappelée et la modifier.
- Enfin la touche tabulation déclenche la complétion :
  - Sur le premier mot de la commande la complétion est faite sur le PATH
  - Sur les autres mots de la commande la complétion est faite sur le système de fichiers.

  Si il y a conflit pour compléter, une deuxième tabulation affiche les possibilités.

Liste

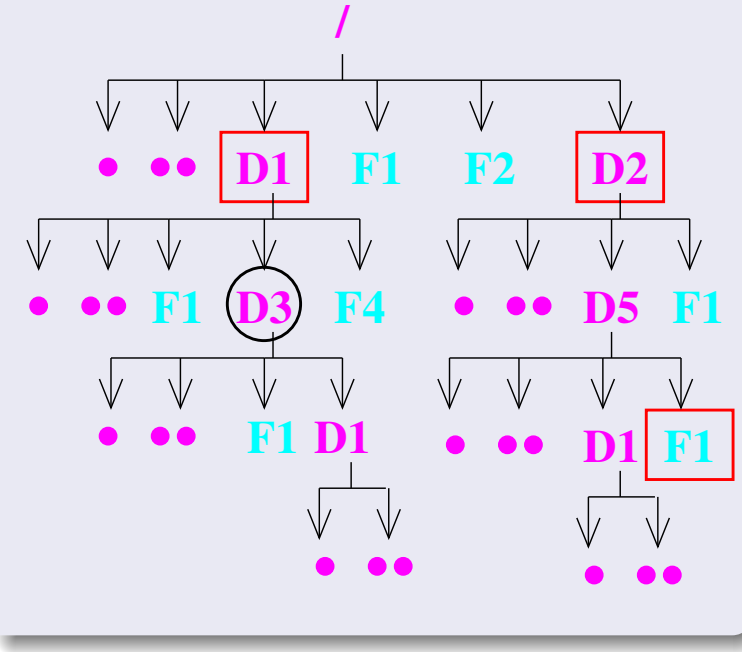
Les commandes de bases en mode interactif :

cd	:	change le CWD.
pwd	:	affiche le CWD.
mkdir	:	création de répertoires
rmdir	:	suppression de répertoires
ls	:	affiche les informations relatives aux fichiers et répertoires
cp	:	copie de fichiers et répertoires
mv	:	déplacement/renommage de fichiers et répertoires
rm	:	suppression de fichiers et répertoires
which	:	affiche le path absolu d'une commande
less	:	un pageur
chmod	:	modification des permissions de fichiers et répertoires
wc	:	compte le nombre de lettres, mots et lignes d'un fichier
find	:	recherche de fichier dans une arborescence
grep	:	recherche un motif dans un fichier
sort	:	tri de fichier
stat	:	affiche les meta-données (taille, type, . . .) d'un fichier
tar	:	archiveur

Les commandes de bases utiles pour les scripts :

echo	:	affiche un message sur le flux standard de sortie
read	:	lit un message sur le flux standard d'entrée
cat	:	concaténation de fichiers
test	:	compare des nombres, des chaines ; obtention de propriétés de fichiers
dirname	:	obtention des répertoires des chemins (/aa/bb/cc/dd)
basename	:	obtention des noms de base des chemins (/aa/bb/cc/dd)

Quelques exemples



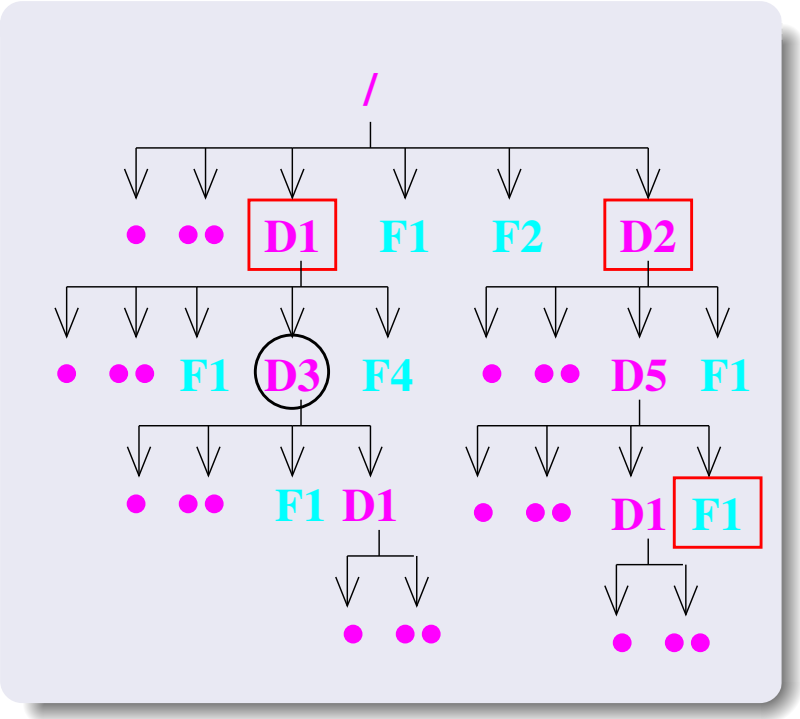
1. Faites que le CWD soit le répertoire D2 encadré.

```
sh> cd /D2
sh> pwd
/D2
sh> ls
D5 F1
sh>
```

ou

```
sh> cd ../../D2
sh>
```

Quelques exemples

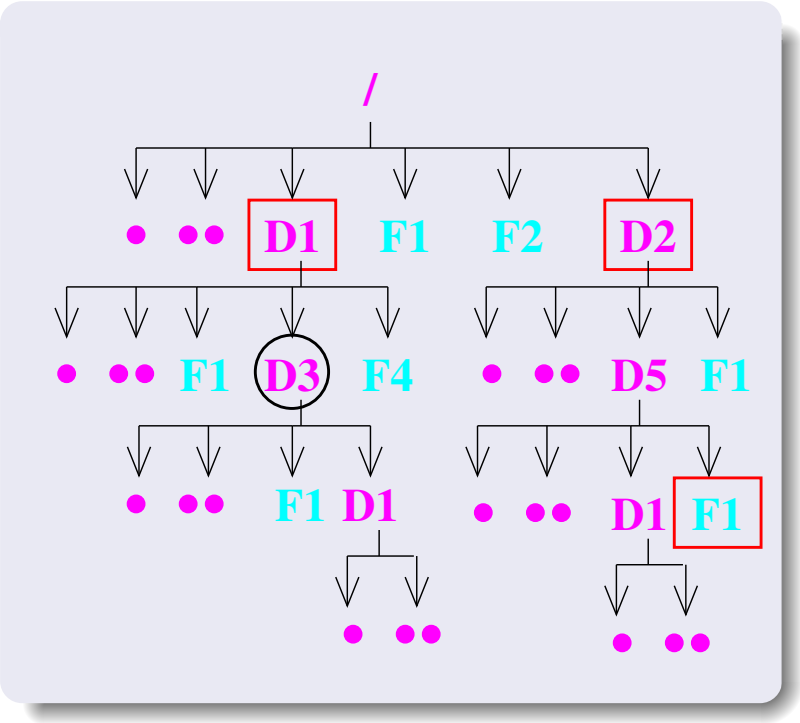


2. Affichez le contenu des répertoires D1 et D2 encadrés.
   - mauvaise méthode

     sh> cd ..
     sh> ls
     F1 D3 F4
     sh> cd /D2
     sh> ls
     D5 F1
     sh> cd ../D1/D3
     sh>
   - bonne méthode

     sh> ls .. /D2
     .. : :
     F1 D3 F4
     /D2 :
     D5 F1 :
     sh>

Quelques exemples

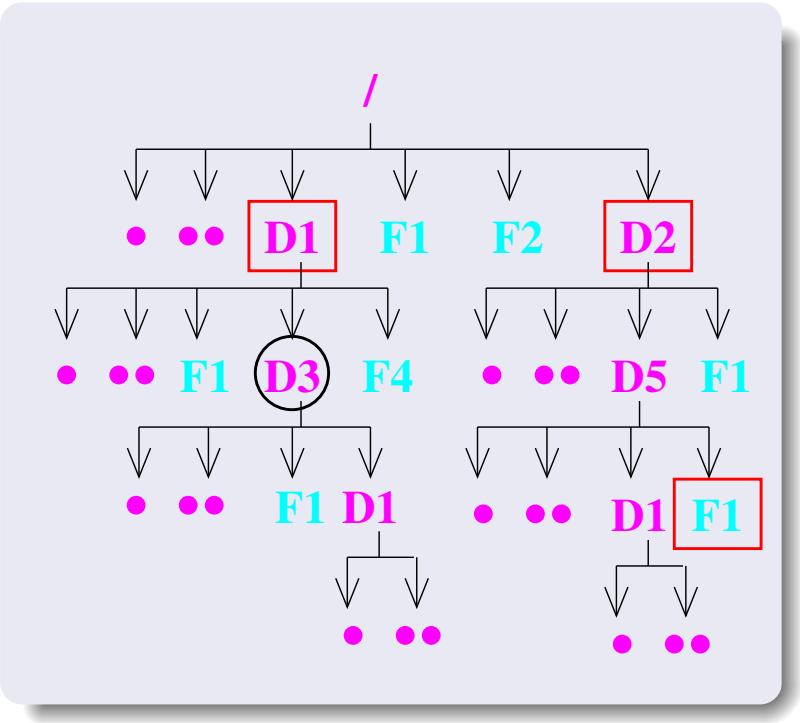


3. Copier le fichier F1 encadré dans D1/gnu.
   - mauvaise méthode

     sh> cd /D2
     sh> ls
     D5 F1
     sh> cd /D5
     sh> ls
     D1 F1
     sh> cp F1 /D1/D3/D1/gnu
     sh> cd /D1/D3/
     sh>
   - bonne méthode

     sh> cp /D2/D5/F1 D1/gnu
     sh>

Quelques exemples



4. Détruisez l'arborescence donnée par le répertoire D2 encadré.

```
sh> rm /D2
sh>
```

5. Pourquoi la séquence ci-dessous ne détruit pas le fichier "-f" ?

```
sh> ls .
gnu -f bee
sh> rm -f
sh> ls .
gnu -f bee
sh>
```

Car -f est une option de **rm**. Comment le détruire ?

```
sh> rm ./-f
sh>
```

Table des matières

Séquence simple

La fin d'une commande est indiquée par un <C-R>ou le caractère ' ;'

```
sh> gcc 1.c
sh> ./a.out 2 + 3
5
sh>
```

```
sh> gcc 1.c ;  ./a.out 2 + 3
5
sh>
```

Si la compilation échoue, la commande **./a.out** est lancée

Le statut renvoyé par la séquence est celui de la dernière commande

Séquence conditionnelle

Les opérateurs && (et) et || (ou) permettent de chainer 2 commandes en fonction du statut de la première. L'opérateur ! (non) permet d'inverser la condition logique.

```
sh> ./a.out -1 gnu || ./a.out -2 gnu || ./a.out -3 gnu
```

1. La seconde commande **./a.out** n'est lancée que si la première échoue.
2. La troisième commande **./a.out** n'est lancée que si les 2 premières échouent.
3. Le statut de la séquence est faux ($\neq 0$) si les 3 commandes échouent.

```
sh> gcc 1.c && ./a.out 2 + 3
```

1. La commande **./a.out** n'est lancée que si la compilation a réussi.
2. Le statut de la séquence est vrai ($= 0$) si les 2 commandes réussissent.

Groupe d'instructions

Une séquence d'instructions peut être parenthésée soit avec des accolades (lancée par le Shell courant), soit avec des parenthèses (lancée par un autre processus Shell).

```
sh> { true  && false ;  } || ( false && true  ) || { false && false ;  }
```

```
sh> ( cd gnu/bee ;  cp f1 f2 ) # CWD inchangé
```

Table des matières

Variables locales

Une variable est identifiée par un nom, l'expression régulière "[a-zA-Z_][a-zA-Z0-9_]*" définit les noms valides.

Affectation d'une variable

```
sh> CMD=cd
sh> DIR=bee
sh> SDIR=gnu
sh>
```

Pas d'espace autour du '='.

Valeur d'une variable

```
sh> $CMD $DIR/$SDIR
```

ou

```
sh> ${CMD} ${DIR}/${SDIR}
```

1. La valeur d'une variable non définie $\Longrightarrow$ chaine vide.
2. La forme parenthésée permet la concaténation de variables :
   $DIRgnu $\Longrightarrow$ **DIRgnu** non définie
   ${DIR}gnu $\Longrightarrow$ beegnu
3. La forme parenthésée permet de nombreuses extensions :
   ${DIR :-undef} $\Longrightarrow$ undef si **DIR** non définie.

Variables d'environnement

Au démarrage le Shell crée une variable locale en la marquant pour chaque variable d'environnement définie.
On peut marquer une variable avec la commande Shell **export** :

```
sh> NVE=gnu
sh> export NVE
sh>
```

```
sh> export NVE
sh> NVE=gnu
sh>
```

```
sh> export NVE=gnu
sh>
```

Toutes les variables marquées sont transmises aux processus créés.

Variables d'environnement

La commande "**export -n NVE**" démarque la variable **NVE**.

Les variables marquées peuvent être visualisées par la commande Shell **export** sans argument ou la commande Unix **env**.

On ajoute une ou plusieurs variables d'environnement à une commande par :

```
sh> VE1=gnu VE2=bee cmd gnat
sh>
```

Table des matières

Principales expansions

Le Shell a beaucoup de commandes implicites qu'il étend pour les exécuter. Les principales sont :

$vname La valeur de la variable **vname**.

${vname...} Une valeur déduite de la valeur de la variable **vname**.

$$ Le PID du Shell.

$? Le statut de la dernière commande exécutée.

!n La $n^{ième}$ commande de l'historique.

!str La dernière commande entrée commancant par str.

$[expr **]**

$((expr **))** Le résultat de l'expression arithmétique expr.

‘ cmd gnu bee ... ‘

$(cmd gnu bee ...)

Le flux stdout de la commande **cmd gnu bee ...**.

Quelques exemples

```
sh> !gcc
gcc -I . -L /usr/local/lib bee.c
sh>
```

```
sh> i=10
sh> i=$[2 * $i + 1]
sh> echo $i $[$i + 1]
21 22
sh>
```

```
sh> cmd $(cat gnu/bee) $[2<<10]
sh>
```

```
sh> cmd $(ls)
sh>
```

Principe

Soit une commande :

```
str0 $[str1 $(str2 str3) ] str4 $str5 ${str6 :-str7 str8} str9
```

La plupart des str_i sont considérées comme des expressions régulières sur le système de fichiers moins ceux commençant par '.'.

1. Si un ou plusieurs fichiers sont compatibles avec cette expression régulière, alors str_i est remplacée par ces fichiers séparés par un espace.
2. Si aucun fichier n'est compatible avec cette expression régulière, alors str_i est conservée telle quelle.

```
sh> ls
sh> echo *
*
sh>
```

```
sh> ls
bee gnu
sh> echo *
bee gnu
sh>
```

Format des expressions régulières

~str en début de chaine (str sans /) donne le HOME de l'utilisateur str.

~/ en début de chaine donne le HOME.

~ en début de chaine et la chaine n'a qu'un caractère donne le HOME.

* 0 ou plusieurs caractères sauf le '/'.

? 1 caractère sauf le '/'.

[$m_0 m_1 m_2 ...$] 1 caractère appartenant à au moins un motif m_i. m_i est
soit un caractère (ex : x),
soit une séquence de 3 caractères (ex : E-K $\Longrightarrow$ {E F G H I J K}).

{expr0,expr1,...} correspond au choix à une des expressions.

autre caractère le caractère.

Quelques exemples

```
sh> ls /[a-zA-Z]*/[a-zA-Z]*/*.[024]*
sh> gcc src/*.c
sh> wc -l prj/*.[ch]
sh> wc -l prj/*{.c,.h}
```

```
sh> ls
echo main.c
sh> *
main.c
sh>
```

Échappement

\c Le caractère c devient un caractère standard.

```
sh> echo \$FILES
$FILES
sh>
```

```
sh> ls gnu\ bee
ls : gnu bee : No such file or directory
sh>
```

"str" Dans la chaine de caractères str :
les espaces sont échappés,
les expansions de fichiers sont échappées,
les expansions de commandes sont toujours actives.

```
sh> ls "gnu bee"
ls : gnu bee : No such
file or directory
sh>
```

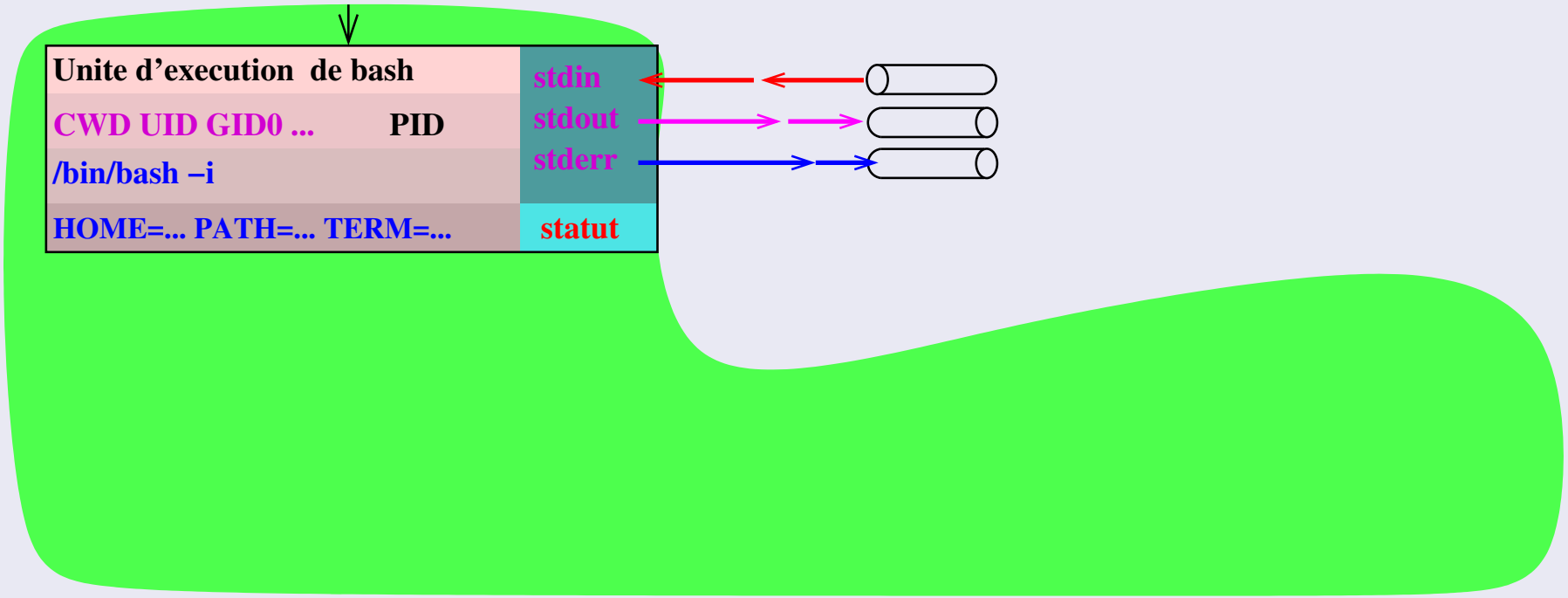
```
sh> FILES="/*"
sh> echo "$FILES ${FILES} $[1+1]"
/* /* 2
sh>
```

'str' Dans la chaine de caractères str tout est échappé.

Table des matières

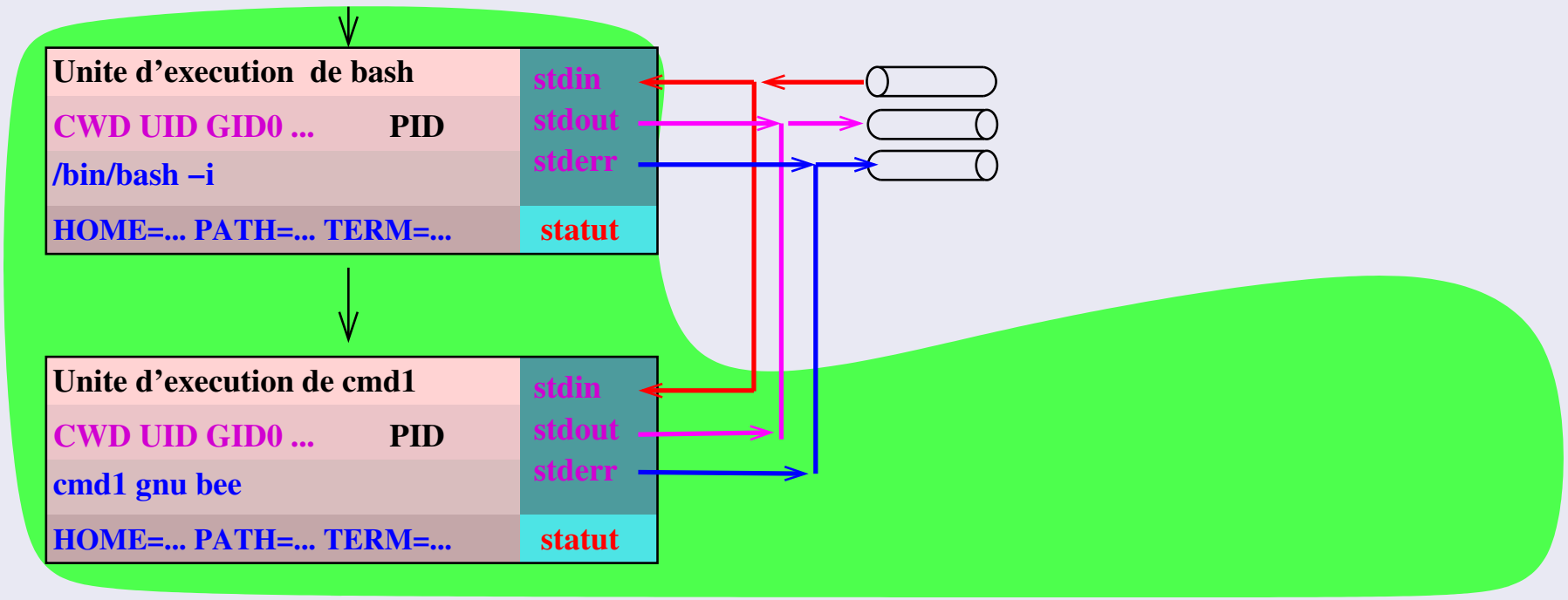
Principe

sh>


Unite d'execution de bash
CWD UID GID0 ... PID
/bin/bash –i
HOME=... PATH=... TERM=...
stdin
stdout
stderr
statut

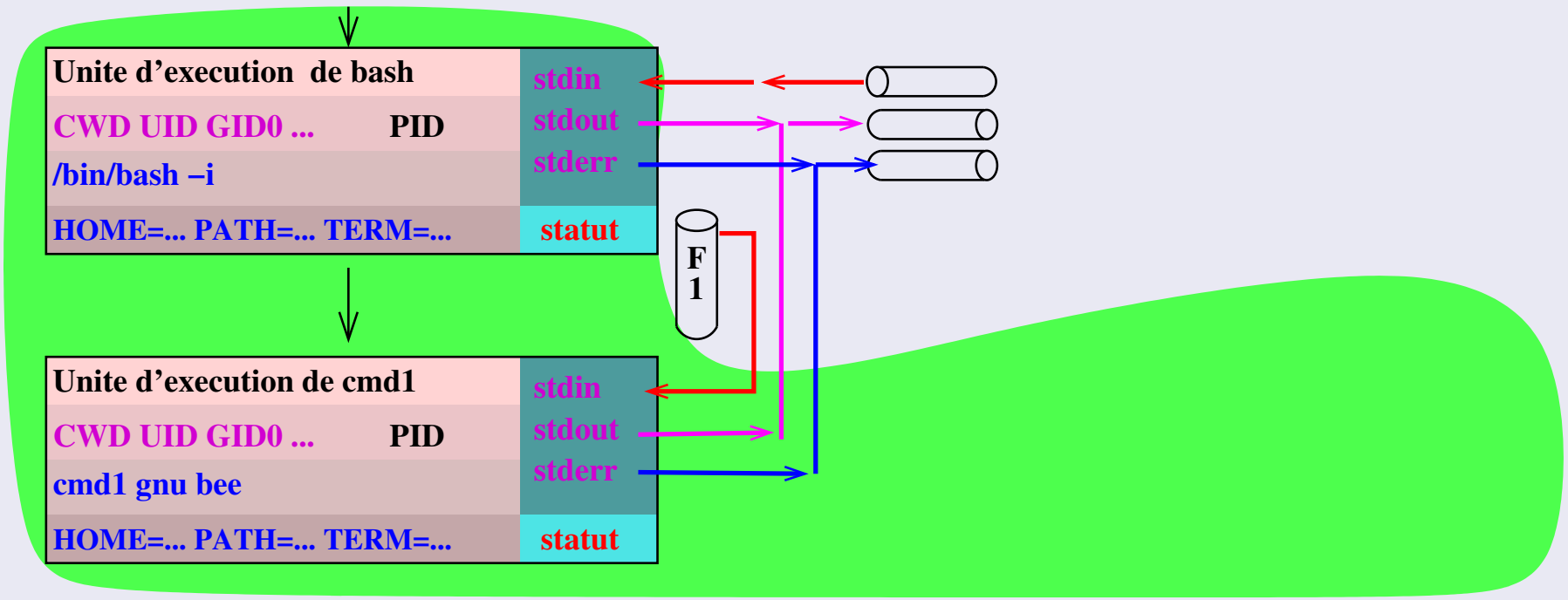
Principe

sh> cmd1 gnu bee



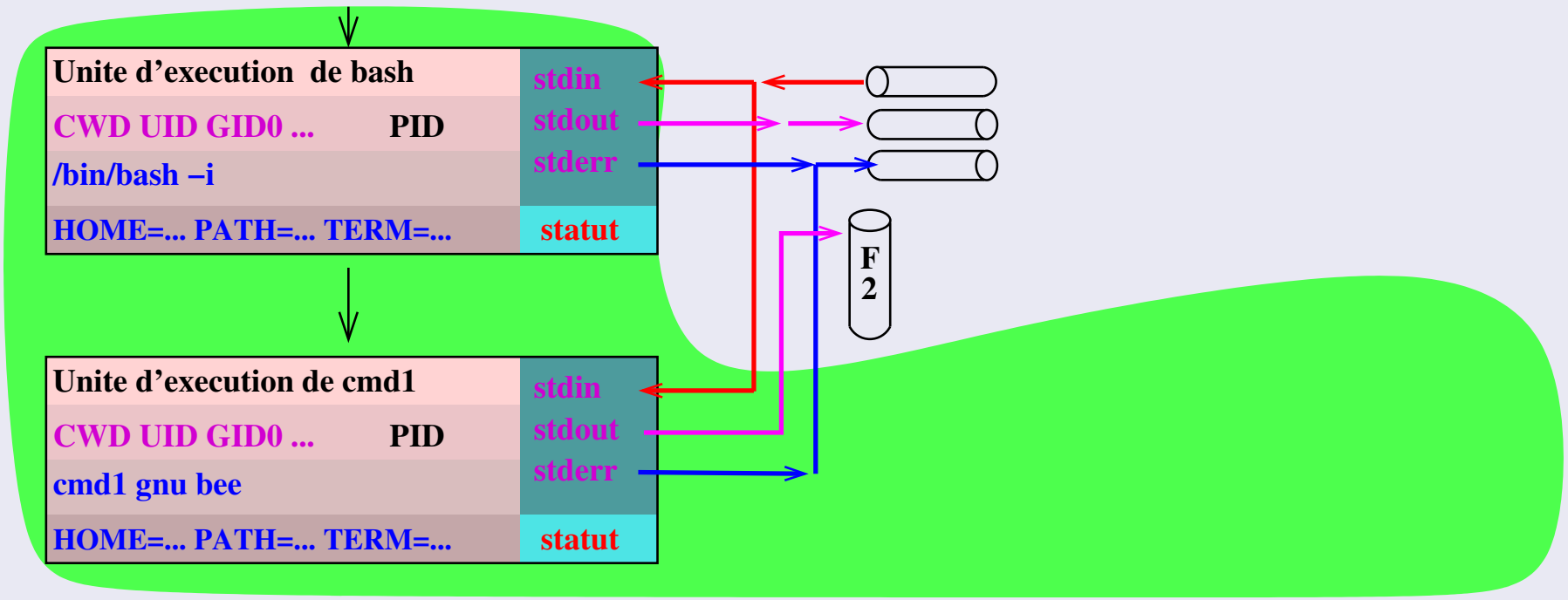
Principe

sh> cmd1 gnu bee < f1



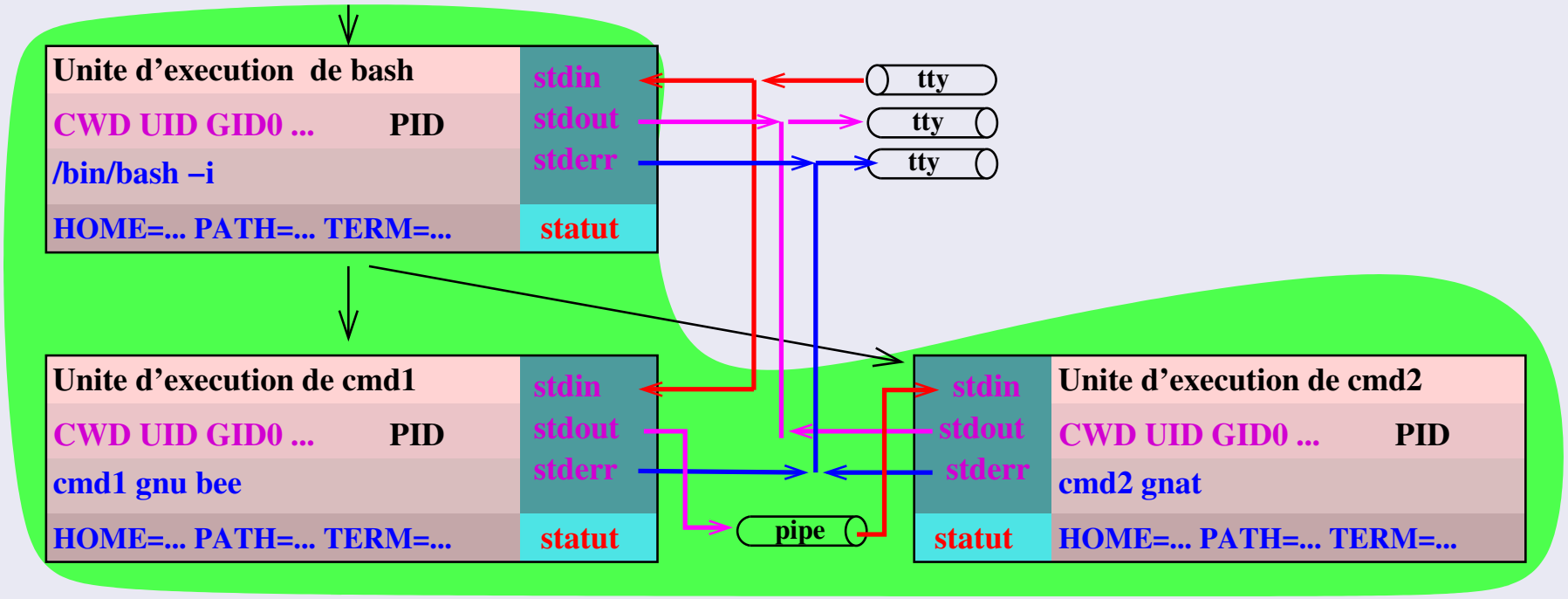
Principe

sh> cmd1 gnu bee > f2



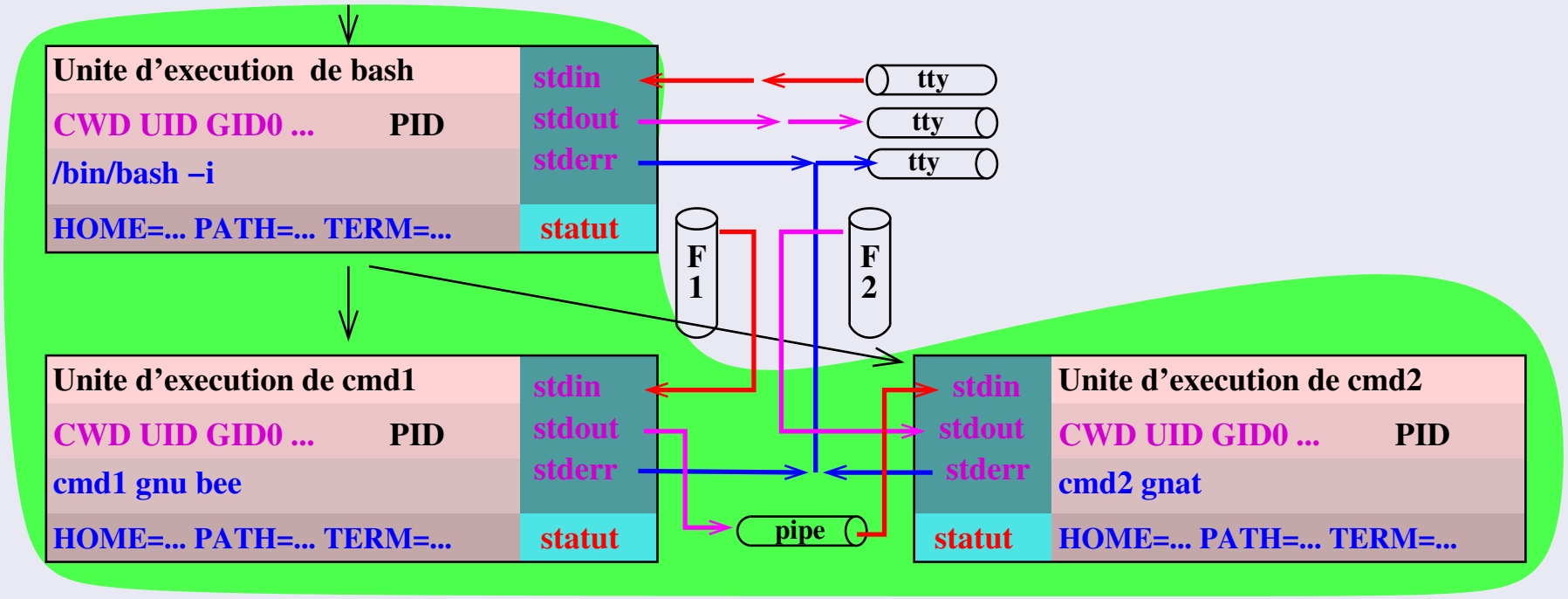
Principe

sh> cmd1 gnu bee | cmd2 gnat



Principe

sh> cmd1 gnu bee < f1 | cmd2 gnat > f2



Syntaxe

cmd ... < path redirige le fichier path vers le flux stdin.
Le fichier path doit exister et être accessible en lecture.

cmd ... > path redirige le flux stdout vers le fichier path.
Si le fichier path existe, il est écrasé.
Si le fichier path n'existe pas, il est créé.
Il doit être accessible en écriture.

cmd ... 2> path similaire à "> path" mais pour le le flux stderr.

cmd ... ¦ **cmd** ... redirige le flux stdout de cmd1 sur le flux stdin de cmd2.

Syntaxe

cmd ... >> path redirige le flux stdout vers le fichier path.
Si le fichier path existe, les écritures se feront à la fin du fichier.
Si le fichier path n'existe pas, il est créé.
Il doit être accessible en écriture.

cmd ... 2>> path similaire à ">> path" mais pour le le flux stderr.

cmd ... 1>&2 redirige le flux stdout sur stderr.

cmd ... 2>&1 redirige le flux stderr sur stdout.

```
cmd ... <<EOF
hello
word
EOF
```

initialise le flux stdin à hello word.

Les redirections peuvent être écrites n'importe où sur la ligne de commande.

Table des matières

Exécution

Fichier hello.sh :

```
1 #!/bin/bash
2
3 echo -n "␣␣" Hello
4 echo World
```

Pour lancer le script :

Méthode 1 `sh> bash hello.sh`

Méthode 2 rendre le script exécutable (chmod) puis `sh> ./hello.sh`

! doivent être les 2 premiers caractères du script.

Le système lance la commande suivant "# !" en ajoutant l'argument "./hello.sh" à la fin.

Méthode 3 `sh> bash < hello.sh`

Exécution

Fichier hello.sh :

```
1 #!/bin/bash
2
3 echo -n "  " Hello
4 echo World
```

Pour lancer le script :

Méthode 1 **sh> bash hello.sh**

Méthode 2 rendre le script exécutable (chmod) puis **sh> ./hello.sh**

#! doivent être les 2 premiers caractères du script.

Le système lance la commande suivant "#!" en ajoutant l'argument "./hello.sh" à la fin.

Méthode 3 **sh> bash** < hello.sh

Dans tous les cas /bin/bash lit le script et il exécute les commandes. Il s'arrête à la fin du script.

Exécution

Fichier hello.sh :

```
1 #!/bin/bash
2
3 echo -n "␣␣" Hello
4 echo World
```

Pour lancer le script :

Méthode 1 **sh> bash hello.sh**

Méthode 2 rendre le script exécutable (chmod) puis **sh> ./hello.sh**

! doivent être les 2 premiers caractères du script.

Le système lance la commande suivant "# !" en ajoutant l'argument "./hello.sh" à la fin.

Méthode 3 **sh> bash** < hello.sh

Les méthodes 1 et 2 sont équivalentes.
Dans la méthode 3 le flux stdin du Shell est le script lui même.

```
1 echo -n "␣␣" Hello
2 cat
3 echo World
```

$\Longrightarrow$

Exécution

Fichier hello.sh :

```
1 #!/bin/bash
2
3 echo -n "␣␣" Hello
4 echo World
```

Pour lancer le script :

Méthode 1 **sh> bash hello.sh**

Méthode 2 rendre le script exécutable (chmod) puis **sh> ./hello.sh**

! doivent être les 2 premiers caractères du script.

Le système lance la commande suivant "# !" en ajoutant l'argument "./hello.sh" à la fin.

Méthode 3 **sh> bash** < hello.sh

Les méthodes 1 et 2 sont équivalentes.
Dans la méthode 3 le flux stdin du Shell est le script lui même.

```
1 echo -n "␣␣" Hello
2 cat
3 echo World
```

$\Longrightarrow$ Hello echo World

"Sourcer" un script

Dans un Shell qui tourne, on peut à tout moment "sourcer" un script par la commande "." ou "source"

`sh> source hello.sh` ou `sh> . hello.sh`

Dans ce cas, le Shell interrompt sa lecture du flux stdin, il lit le script à la place. A la fin du script, il se remet à lire le flux stdin.

Dans ce cas, le script peut modifier les données du Shell (ex : PATH).

Les scripts Shell de démarrage (.bashrc, .profile, ...) sont sourcés. Ceci permet de configurer son Shell.

Table des matières

Alternative : Syntaxe

```
1 if cmd-cond
2 then
3     ...
4 else
5     ...
6 fi
```

```
1 if cmd-cond ; then
2     ...
3 else
4     ...
5 fi
```

- La clause then est obligatoire.
- La clause else est facultative.
- Le ; est quasi obligatoire sinon then est compris comme un argument de cmd-cond.
- cmd-cond est soit une commande simple ou un groupe de commandes.
- Si la commande cmd-cond renvoie le statut 0 (ok) la clause then est exécutée.
- Si la commande cmd-cond renvoie un statut $\neq 0$ (pas ok) la clause

Alternative : Exemple 1

Tester si le fichier file contient main.

```
1 if grep -q main file ; then
2   echo main dans file
3 else
4   echo pas de main dans file
5 fi
```

Si file n'existe pas grep écrit un message d'erreur d'où :

```
1 if grep -q main file 2> /dev/null ; then
2   echo main dans file
3 else
4   echo pas de main dans file
5   echo ou file non accessible
6 fi
```

Alternative : Exemple 2

Écrire oui si on est en 2016 ?

```
1 if date | grep -q 2016 ; then
2   echo oui
3 fi
```

Enfin pour les petites demi alternatives, on peut aussi utiliser les opérateurs logiques && et ||.

```
1 date | grep -q 2016 && echo oui
```

Cas : Cas d'une constante

```
1 case mot in
2   motif )
3     ...
4     ;;
5   motif | motif )
6     ...
7     ;;
8   ...
9 esac
```

Format des motifs :

- ~str : en début de chaine (str sans /) donne le HOME de l'utilisateur str.
- ~/ : en début de chaine donne le HOME.
- ~ : en début de chaine et la chaine n'a qu'un caractère donne le HOME.
- * : 0 ou plusieurs caractères.
- ? : 1 caractère.
- [$m_0 m_1 m_2 ...$] : 1 caractère défini par les m_i. m_i est soit un caractère (ex : x), soit une séquence de caractères (ex : E-K $\Longrightarrow$ {E F G H I J K}).

- La première clause dont un motif décrit le mot est exécutée puis le contrôle reprend après le cas.
- Si aucun motif ne décrit le mot, aucune clause n'est exécutée et le contrôle reprend après le cas.

Cas : Choix conditionnel d'expressions booléennes

```
1 if cmd-cond1 ; then
2     ...
3 elif cmd-cond2 ; then
4     ...
5 elif cmd-cond3 ; then
6     ...
7 else
8     ...
9 fi
```

elif permet d'emboiter des si sans multiplier les fin de si.

- Choisit la première clause dont la condition cmd-cond$_i$ est vraie.
- Si aucune cmd-cond$_i$ n'est vraie exécute la clause else.
- La clause else est facultative.

Cas : Exemple

Tester si la variable n est un nombre de 1 à 3 chiffres.

```
case $n in
   [0-9])          good=1 ;;
   [0-9][0-9])     good=1 ;;
   [0-9][0-9][0-9])
                   good=1 ;;
   *)              good=0 ;;
esac
```

```
case $n in
   [0-9] | [0-9][0-9] |
   [0-9][0-9][0-9] )
          good=1 ;;
   * )    good=0 ;;
esac
```

Déterminer si le mot donné par la variable m commence, se termine ou contient la chaine de caractères ia.

```
debut=0; fin=0; mil=0;
case $m in
   ia* )          debut=1 ;;
   *ia )          fin=1 ;;
   *ia*)          mil=1 ;;
esac
```

While : Syntaxe

```
1 while cmd-cond
2 do
3       ...
4       ...
5 done
```

```
1 while cmd-cond ; do
2       ...
3       ...
4 done
```

- Le ; est quasi obligatoire sinon “do” est compris comme un argument de cmd-cond.
- Si la commande cmd-cond renvoie le statut 0 (ok), le corps de la boucle est exécuté puis le contrôle reprend en début de boucle.
- Si la commande cmd-cond renvoie un statut $\neq 0$ (pas ok), le contrôle reprend après la boucle.
- Dans le corps de boucle les commandes suivantes sont disponibles :
  continue branche en début de boucle,
  break donne le contrôle après la boucle.

While : Exemple

La commande ping -c 1 host permet de tester si la machine host est accessible via le réseau.

Écrire un script qui toutes les 30 secondes émet 5 beeps si la machine 192.168.1.10 n'est pas accessible.

```
1 while true ; do
2     sleep 30
3     ping -c 1 192.168.1.10 > /dev/null 2>&1 || \
4       echo -e -n "\b\b\b\b\b"
5 done
```

For Syntaxe

```
1 for v in str-list
2 do
3       ...
4       ...
5 done
```

```
1 for v in str-list ; do
2       ...
3       ...
4 done
```

- Le ; est quasi obligatoire sinon “do” est compris comme un élément de str-list.
- str-list est un suite de chaines de caractères str_i séparées par un ou plusieurs espaces.
- La variable v prend dans l'ordre de str-list toutes les valeurs str_i et le corps de boucle est exécuté pour chaque valeur.
- Dans le corps de boucle les commandes suivantes sont disponibles :
  continue branche en début de boucle,
  break donne le contrôle après la boucle.
- La chaine de caractères {n..m} est expansée en tous les nombres de n à m compris.
- La chaine de caractères {n..m..i} est expansée en tous les nombres de n

For Exemple

Écrire Hello World lettre par lettre.

```
1 for l in H e l l o "␣" W o r l d ; do
2   echo -n $l
3 done
4 echo
```

Écrire 10 lignes contenant chacune 5 fois silence.

```
1 for i in {1..10} ; do
2   for j in {19..15} ; do
3     echo -n silence "␣"
4   done
5   echo
6 done
```

For Exemple

Ecrire sur une ligne les noms de base des fichiers du répertoire donné par la variable dir, suffixés entre parenthèses de leur nombre de lignes.

```
1 for bn in $(cd $dir ; ls) ; do
2   echo -n "$bn(" $(wc -l < $dir/$bn 2> /dev/null) "
3 done
4 echo
```

Table des matières

Commande test

Pour pouvoir écrire des programmes, il faut pouvoir faire des tests :

- le fichier str existe-t-il ?
- le fichier str est il un répertoire ?
- les chaines str_1 et str_2 sont-elles égales ?
- n_1 est-il inférieur à n_2 ?
- ...

la commande **test** répond à ce besoin.

Les arguments de la commande définissent une expression booléenne, elle renvoie un statut de 0 si l'expression est vraie et de 1 sinon.

sh> **test** **arg_0** **arg_1** **arg_2** **...**

ou

sh> **[** **arg_0** **arg_1** **arg_2** **...** **]**

test : Arguments

str

-n str str est non vide

-z str str est vide

str_1 = str_2 str_1 et $str2_2$ sont égales (!= pour différentes).

str_1 \< str_2 str_1 est inférieure (ordre lexicographique) à $str2_2$ (\> pour supérieure) (bash builtin).

n_1 OP n_2 l'entier n_1 est OP à l'entier n_2 avec OP dans { -eq, -ne, -lt, -le, -gt, or -ge } (égal, diffèrent, inférieur, inférieur ou égal, supérieur, supérieur ou égal).

test : Arguments

-e file le fichier file existe et n'est pas un lien mort.

-f file le fichier file** existe et est régulier.

-d file le fichier file** existe et est un répertoire.

-h file le fichier file existe et est lien symbolique.

-r file le fichier file** existe et peut être lu (file peut être un répertoire).

-w file le fichier file** existe et peut être écrit (file peut être un répertoire).

-x file le fichier file** existe et peut être exécuté (file peut être un répertoire).

! expr opérateur booléen non.

expr -o expr opérateur booléen ou.

expr -a expr opérateur booléen et.

\(et \) permet de parenthèser une expression booléenne.

Note ** : ou dans le cas d'un lien, le fichier pointé final.

test : Exemples

Déterminez si les chaines de caractères définies par les variables S1 et S2 sont égales :

```
1 if test $S1 = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [ $S1 = $S2 ] ; then
2   echo S1 = S2
3 fi
```

Sauf si S1 est la chaine vide ou S1 n'est pas définie, l'expansion sera :

```
1 if test = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [ = $S2 ] ; then
2   echo S1 = S2
3 fi
```

Dans ce cas, la commande test écrit un message d'erreur "mauvais formatage de l'expression" et renvoie un statut différent de ∅. D'où la correction :

```
1 if test "$S1" = "$S2" ; then
2   echo S1 = S2
3 fi
```

```
1 if [ "$S1" = "$S2" ] ; th
2   echo S1 = S2
3 fi
```

test : Exemples

Déterminez si les chaines de caractères définies par les variables S1 et S2 sont égales :

```
1 if test $S1 = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [ $S1 = $S2 ] ; then
2   echo S1 = S2
3 fi
```

Sauf si S1 est la chaine vide ou S1 n'est pas définie, l'expansion sera :

```
1 if test = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [ = $S2 ] ; then
2   echo S1 = S2
3 fi
```

Dans ce cas, la commande test écrit un message d'erreur "mauvais formatage de l'expression" et renvoie un statut différent de $\emptyset$. D'où la correction :

```
1 if test "$S1" = "$S2" ; then
2   echo S1 = S2
3 fi
```

```
1 if [ "$S1" = "$S2" ] ; th
2   echo S1 = S2
3 fi
```

test : Exemples

Déterminez si les chaines de caractères définies par les variables S1 et S2 sont égales :

```
1 if test $S1 = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [ $S1 = $S2 ] ; then
2   echo S1 = S2
3 fi
```

Sauf si S1 est la chaine vide ou S1 n'est pas définie, l'expansion sera :

```
1 if test = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [ = $S2 ] ; then
2   echo S1 = S2
3 fi
```

Dans ce cas, la commande test écrit un message d'erreur "mauvais formatage de l'expression" et renvoie un statut différent de $\emptyset$. D'où la correction :

```
1 if test "$S1" = "$S2" ; then
2   echo S1 = S2
3 fi
```

```
1 if [ "$S1" = "$S2" ] ; th
2   echo S1 = S2
3 fi
```

test : Exemples

Déterminez si l'entier défini par la variable n est compris entre 10 et 23 compris :

```
1 if test 10 -le $n -a $n -le 23 ; then
2   echo oui
3 fi
```

ou

```
1 if [ 10 -le $n -a $n -le 23 ] ; then
2   echo oui
3 fi
```

test : Exemples

Écrire $n lignes contenant le mot silence :

```
1 i=0
2 while test $i -lt $n ; do
3   echo silence
4   i=$[i+1]  # ou $[i++]
5 done
```

test : Exemples

Écrire ok si le fichier $f est régulier et accessible en lecture, écriture et non accessible en exécution :

```
1 if test -f $f -a -r $f -a -w $f -a ! -x $f ; then
2   echo ok
3 fi
```

ou

```
1   if test -f $f && [ -r $f -a -w $f -a ! -x $f ] ;
2     echo ok
3   fi
```

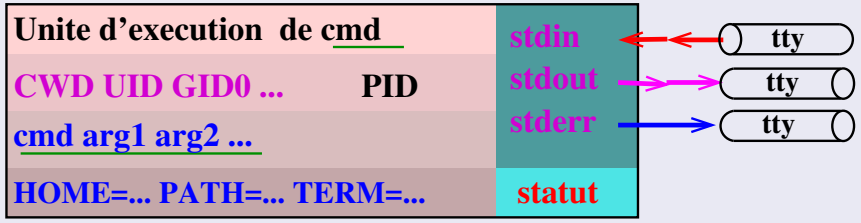
Autres commandes

Ces commandes sont des builtin commandes (commandes internes, elles n'ont pas d'exécutables associés).

exec cmd arg_1 arg_2 ...

Remplace l'unité d'exécution du processus Shell, par celle de l'exécutable **cmd** et les arguments par arg_1 arg_2 ....

avant l'exec



après l'exec

$\Longrightarrow$ pas de création de processus.

$\Longrightarrow$ l'instruction qui suit l'exec ....

Autres commandes

exit n

Termine le Shell avec le statut n. Si n est omis, le statut est $\emptyset$.

read v_1 v_2 v_3

Lit une ligne dans le flux stdin et affecte le contenu la ligne dans les variables v_i. La ligne est considérée comme n chaines de caractères séparées par un ou plusieurs espaces. Soit s_1 s_2 s_3 le contenu d'une ligne

inst. \ var.	v_1	v_2	v_3	v_4
read				
read v_1	s_1 s_2 s_3			
read v_1 v_2	s_1	s_2 s_3		
read v_1 v_2 v_3	s_1	s_2	s_3	
read v_1 v_2 v_3 v_4	s_1	s_2	s_3	" "

Arguments : Expansions dédiées

Quand on lance un script Shell, comment récupère-t-on ses arguments

```
sh> ./script petit lapin
sh>
```

```
1 #!/bin/bash
2
3 taille=... # arg1 (petit)
4 animal=... # arg2 (lapin)
```

Dans un script, on a les expansions suivantes :

- `$#` le nombre d'arguments.
- `$0` le nom de la commande.
- `$1` le premier argument.
- `$2` le second argument.
- `$i` le $i^{ième}$ argument.
- `$*` la liste des arguments séparés par un espace.
- `"$@"` la liste des arguments quotés séparés par un espace.

Arguments : Expansions dédiées

Quand on lance un script Shell, comment récupère-t-on ses arguments

```
sh> ./script petit lapin
sh>
```

```
1 #!/bin/bash
2
3 taille=... # arg1 (petit)
4 animal=... # arg2 (lapin)
```

Dans un script, on a les expansions suivantes :

- `$#` le nombre d'arguments.
- `$0` le nom de la commande.
- `$1` le premier argument.
- `$2` le second argument.
- `$i` le $i^{ième}$ argument.
- `$*` la liste des arguments séparés par un espace.
- `"$@"` la liste des arguments quotés séparés par un espace.

Arguments : Exemples

Écrire le script "**punition n word**" qui écrit sur le flux stdout n lignes contenant le mot word :

```
1 #!/bin/bash
2 #
3 # usage: punition n word
4
5 i=0
6 while test $i -lt $1 ; do
7    echo $2
8    i=$[i+1]
9 done
```

Arguments : Exemples

Écrire le script "**somme** n_1 n_2 **...**" qui écrit sur le flux stdout la ligne "n_1 + n_2 ... + $_n$".

```
1  #!/bin/bash
2  #
3  # usage: somme n1 n2 n3 ...
4
5  first=1
6  for n in $* ; do
7     test $first = 0 && echo -n +
8     echo -n $n
9     first=0;
10 done
```

Arguments : Commandes dédiées I

shift Équivalent à "shift 1".

shift n Décale les arguments de n positions vers la gauche.

La table ci-dessous donne les valeurs des $1, $2, $3, $4, $5 en supposant que leurs valeurs initiales sont v_i et que les valeurs des variables 6,7, ... sont la chaine de caractères vide.

inst. \ var.	$1	$2	$3	$4	$5
echo $*	v_1	v_2	v_3	v_4	v_5
shift 1	v_2	v_3	v_4	v_5	
shift 2	v_3	v_4	v_5		
shift 3	v_4	v_5			
shift 4	v_5				
shift 5					
shift 6					
...					

Arguments : Commandes dédiées II

set -- str_1 str_2 ... str_n

Affecte str_i à $\$_i$ pour i inférieur ou égal à n.

Affecte la chaine vide à $\$_i$ pour i supérieur n.

Arguments : Exemples

Écrire le script "**punition n word**" qui écrit sur le flux stdout n lignes contenant le mot word. Si n est omis, sa valeur par defaut est 5.

```
1  #!/bin/bash
2  #
3  # usage: punition [n] word
4
5  test $# = 1 && set -- 5 "$1"
6
7  i=0
8  while test $i -lt $1 ; do
9     echo $2
10    i=$[i+1]
11 done
```

Arguments : Exemples

Écrire le script "**punition n word$_1$ word$_2$ ...**" qui écrit sur le flux stdout n lignes contenant les mots word$_i$:

```
1  #!/bin/bash
2  #
3  # usage: punition n word1 word2 ...
4
5  n=$1
6  shift
7  i=0
8  while test $i -lt $n ; do
9     echo $*
10    i=$[i+1]
11 done
```

Table des matières

Création de Builtin commande

```
1  #!/bin/bash
2  ...
3  # usage: plus n1 n2 n3 ...
4  function plus( )
5  {
6     s=0
7     while test $# -gt 0 ; do
8        s=$[$s+$1]
9        shift
10    done
11    echo $s
12 }
13 ...
14 plus 10 12 100
15 ...
16 s=$(plus 1 2 3)
17 ...
```

Définit la commande plus.

Les tokens "function" et "()" sont facultatif mais il en faut au moins un des 2.

Une fois définie la commande plus s'utilise comme n'importe quelle commande.

Si une commande plus existait accessible par le PATH, elle est masquée.

Si une builtin commande plus existait déjà, elle est écrasée.

Création de Builtin commande

```
1  #!/bin/bash
2  ...
3  # usage: plus n1 n2 n3 ...
4  function plus( )
5  {
6     s=0
7     while test $# -gt 0 ; do
8        s=$[$s+$1]
9        shift
10    done
11    echo $s
12 }
13 ...
14 plus 10 12 100
15 ...
16 s=$(plus 1 2 3)
17 ...
```

variable

- non déclarée locale $\Longrightarrow$ globale
- déclarée locale $\Longrightarrow$ locale à partir de la déclaration

"local v" ou "local v=6" pour déclarer une variable v locale.

statut de retour le statut de la dernière commande exécutée.

return identique à "return 0"

return n quitte la commande avec un statut de n.

Création de Builtin commande

```
1  #!/bin/bash
2  ...
3  # usage: plus n1 n2 n3 ...
4  function plus( )
5  {
6     s=0
7     while test $# -gt 0 ; do
8        s=$[$s+$1]
9        shift
10    done
11    echo $s
12 }
13 ...
14 plus 10 12 100
15 ...
16 s=$(plus 1 2 3)
17 ...
```

La commande exit peut être appelée dans une builtin commande, elle termine le script.

La commande return ne peut pas être appelée en dehors d'une builtin commande.

Table des matières

Introduction

Pour illustrer ce chapitre, nous allons prendre l'exemple :

Réalisez le script "ecp f1 f2" qui copie le fichier régulier f1 dans f2. Le chemin de f2 est éventuellement créé.

Entête

```
1 #!/bin/bash
2 #
3 # usage: ecp f1 f2
4 # fonction: copie f1 dans f2
5 #    en creant le chemin f2
```

- ligne 1 automatique
- écrivez l'usage et la fonction (fixe les idées et aide la reprise)

```
6   # analyse des args
7   if test $# != 2 ; then
8     echo "$0: ..." 1>&2
9     exit 1
10  fi
11  src="$1"
12  des="$2"
```

- vérifiez le nombre d'arguments
- message d'erreur clair + sur stderr + exit avec statut d'erreur
- donnez des noms significatifs aux arguments

Test des arguments

```
13      # test de src
14      if ! test -f "$src"
15              -a -r "$src" then
16        echo "$0: ..." 1>&2
17        exit 1
18      fi
```

- vérifie si src est lisible et n'est pas un répertoire.
- message d'erreur clair + sur stderr + exit avec statut d'erreur

Test des arguments

```
19      # test de dest
20      if test -d "$des" ; then
21        des="$des/$(basename␣"$src")"
22      fi
23      if test -h "$des" &&
24           ! rm "$des" 2> /dev/null the
25        echo "$0:␣..." 1>&2 ; exit 1
26      fi
27      if test -d "$des" ; then
28        echo "$0:␣..." 1>&2 ; exit 1
29      fi
```

- calcule le vrai nom du fichier destination
- vérifie que ce n'est pas un répertoire
- le supprime si c'est un lien

```
30      # des est regulier ou n'existe pas
31      if ! mkdir -p "$(dirname "$des")"
32              2> /dev/null ; then
33        echo "$0: ..." 1>&2 ; exit 1
34      fi
35      if ! cp "$src" "$des" \
36            2>/dev/null then
37        echo "$0: ..." 1>&2 ; exit 1
38      fi
39      exit 0
```

- création des répertoires du fichier destinatio besoin (mkdir -p)
- laisse cp faire le
- le exit 0 final n'e pas nécessaire.

Debug

Pour déboguer, le bash propose l'option -x qui affiche toutes les commandes expansées.

Pour l'activer, il faut soit lancer le script cmd par

bash -x **cmd** **arg$_1$** **arg$_2$** **. . .**

ou ajouter -x à la premiere ligne du script :

#!bash -x

ou insérer un "set -x" dans le script, le mode debug sera actif après cette commande.

Le mode debug peut être désactivé par l'exécution de la commande "set +x".

Pour déboguer, le bash propose aussi l'option -v qui affiche toutes les commandes mais sans les expanser. Elle s'active de la même manière et les 2 options sont cumulables (set -xv).

Table des matières

Césure de lignes

Soit la ligne : **if test** –f f && rm f 2>/dev/null ; **then**

```
1 | if test -f f && rm f
2 |   2>/dev/null ; then
```

```
1 | if test -f f && rm f \
2 |   2>/dev/null ; then
```

```
1 | if test -f f && rm
2 |   f 2>/dev/null ; then
```

```
1 | if test -f f && rm \
2 |   f 2>/dev/null ; then
```

```
1 | if test -f f &&
2 |   rm f 2>/dev/null ; then
```

OK

$\Longrightarrow$ En cas de doute, un \ ne peut pas faire de mal.

Espace dans les Expansions

```
1 x=8
2 if test $x ; then
```

VRAI : x est non vide

```
1 x="a␣b"
2 if test $x ; then
```

FAUX : x est vide + message d'erreur de test

```
1 x="1␣=␣0"
2 if test $x ; then
```

FAUX : x est vide

$\Longrightarrow$ double quoter $x ("$x")

Espace dans les Expansions

```
1 p="bee/gnat"
2 if test -d $(dirname $p)
```

OK : test -d bee

```
1 p="bee/gnat␣gnu"
2 if test -d $(dirname $p)
```

ERREUR : dirname bee/gnat gnu
$\Longrightarrow$ message d'erreur

```
1 p="bee/gnat␣gnu"
2 if test -d $(dirname "$p")
```

OK : test -d bee

Espace dans les Expansions

```
1 p="bee␣old/gnat␣gnu"
2 if test -d $(dirname "$p")
```

ERREUR : test -d bee old $\Longrightarrow$ message d'erreur

```
1 p="bee␣old/gnat␣gnu"
2 if test -d "$(dirname␣"$p")"
```

OK : test -d "bee old"

$\Longrightarrow$ double quoter $(cmd) ("$(cmd)")

Espace dans les Expansions

```
1 set -- "a" "b" "c"
2 for s in $* ; do
```

OK : 3 fois la boucle

```
1 set -- "a" "b" "c␣d"
2 for s in $* ; do
```

ERREUR : 4 fois la boucle

```
1 set -- "a" "b" "c␣d"
2 for s in "$*" ; do
```

ERREUR : 1 fois la boucle

```
1 set -- "a" "b" "c␣d"
2 for s in "$@" ; do
```

OK : 3 fois la boucle

$\Longrightarrow$ ne pas utiler $* mais ("$@")

Variable dans un sous processus

```
1 ( cd .. ; test "$x" || \
2       x=$(echo *) )
```

ARGH ! : x est inchangée

```
1 test "$x" || \
2       x=$(cd .. ; echo *)
```

OK : si x était vide, x est initialisée aux fichiers de ..

```
1 cd .. ; test "$x" || \
2       x=$(echo *) ; cd -
```

OK : mais plus lourd

Variable dans un sous processus

```
1 | read x y < f
```

OK : lit la $1^{ière}$ ligne de f dans x et y

```
1 | cat f | read x y
```

ARGH ! read est fait dans un sous processus

Les principaux éléments de syntaxe qui créent des sous processus sont : ( ), $( ) et ¦

Utilisation de read

```
1 read a b
2 read x y
```

OK : lit 2 lignes du flux stdin

```
1 while read a b ; do
2     ...
3 done
```

OK : lit tout le flux stdin ligne par ligne

Utilisation de read

```
1 read a b < f1
2 read x y < f1
```

ARGH ! : lit 2 fois la même ligne (sauf si f1 est un flux tty, audio)

```
1 while read a b < f1 ; do
2     ...
3 done
```

ARGH ! : lit à l'infini la $1^{ière}$ ligne du flux stdin.

```
1 while read a b ; do
2     ...
3 done < f1
```

OK : lit tout le fichier f1 ligne par ligne

Utilisation de read

```
1 x=0 ; a=0
2 sed /^#/d f1 / \
3 while read a b ; do
4    x=$a
5    ...
6 done
```

OK : lit tout le fichier f1 ligne par ligne en sautant les lignes commençant par #. Que valent ces variables après le done ?

x

a

Utilisation de read

```
1 x=0 ; a=0
2 sed /^#/d f1 | \
3 while read a b ; do
4    x=$a
5    ...
6 done
```

OK : lit tout le fichier f1 ligne par ligne en sautant les lignes commençant par #. Que valent ces variables après le done ?

x 0.

a 0.