Systèmes d'exploitation

Shell et Perl

Michaël HAUSPIE
Cours original de B. Beaufils

Université Lille 1

Année 2015/2016

Licence Professionnelle

da2i

Plan

Shell pour l'administrateur

1. Unix
2. Shell avancé
3. Quelques commandes shells
4. Expressions régulières

Perl et programmation système

5. Le langage Perl
6. Les processus
7. Communications entre processus (tubes/signaux)
8. Accès au système de fichiers

Plan

Shell pour l'administrateur

1. Unix
2. Shell avancé
3. Quelques commandes shells
4. Expressions régulières

Perl et programmation système

5. Le langage Perl
6. Les processus
7. Communications entre processus (tubes/signaux)
8. Accès au système de fichiers

Shell

Histoire d'Unix

- Créé chez AT&T au début des années 1970
- par Dennis Ritchie et Ken Thompson
- Parent de tous les systèmes actuels
  - de Windows à Mac OS X
  - de Android à iOS
- Beaucoup d'implémentations différentes
  - BSD vs System V
  - AIX, HP/UX, FreeBSD, Linux
- Définition d'une philosophie plus que d'une norme
  - norme POSIX

Histoire d'Unix

Ken Thompson (assis) et **Dennis Ritchie** sur un PDP-11

Tiré de « The art of Unix programming »
http://catb.org/esr/writings/taoup/html

Histoire d'Unix

- Créé chez AT&T au début des années 1970
- par Dennis Ritchie et Ken Thompson
- Parent de tous les systèmes actuels
  - de Windows à Mac OS X
  - de Android à iOS
- Beaucoup d'implémentations différentes
  - BSD vs System V
  - AIX, HP/UX, FreeBSD, Linux
- Définition d'une philosophie plus que d'une norme
  - norme POSIX

Histoire d'Unix

The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

Unix is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 a
computers. It offers a number of features seldom found even in larger operating systems, including

i
A hierarchical file system incorporating demountable volumes,

ii
Compatible file, device, and inter-process I/O,

iii
The ability to initiate asynchronous processes,

iv
System command language selectable on a per-user basis,

v
Over 100 subsystems including a dozen languages,

vi
High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

Communications of the ACM, 17, No. 7 (July 1974), pp. 365-375
http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html

Histoire d'Unix

- Créé chez AT&T au début des années 1970
- par Dennis Ritchie et Ken Thompson
- Parent de tous les systèmes actuels
  - de Windows à Mac OS X
  - de Android à iOS
- Beaucoup d'implémentations différentes
  - BSD vs System V
  - AIX, HP/UX, FreeBSD, Linux
- Définition d'une philosophie plus que d'une norme
  - norme POSIX

Philosophie d'UNIX

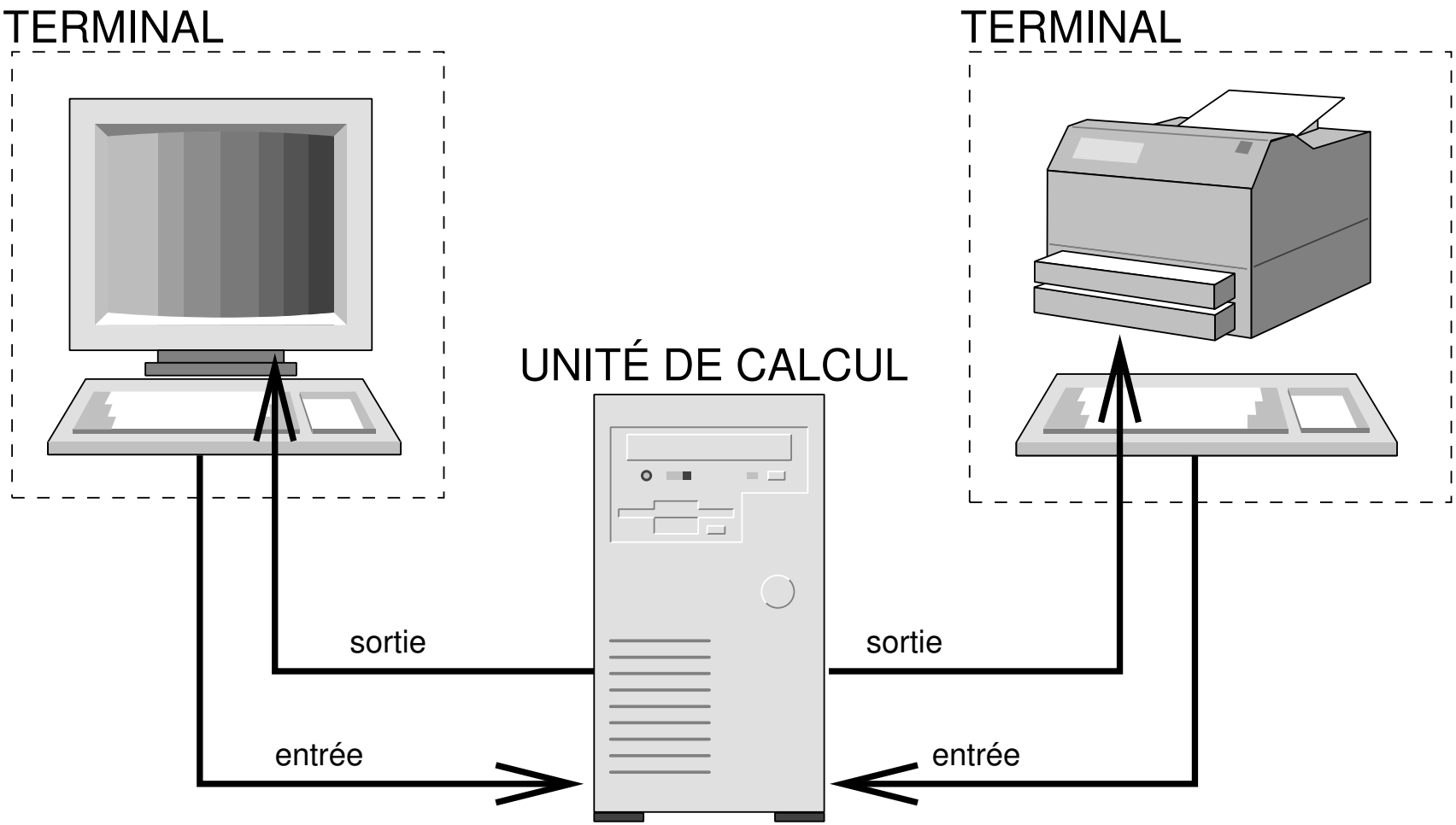
- Code source est (souvent) disponible et facile à lire
- Interface utilisateur simple (pas forcément très conviviale mais simple)
- **Petit nombre de primitives mais aux combinaisons très nombreuses**
- Toutes les interfaces avec les périphériques sont unifiées (via les fichiers)
- Le système est « *indépendant* » de l'architecture matérielle

Caractéristiques d'UNIX

UNIX est un système d'exploitation :

- multi-utilisateurs
- multi-tâches
- qui possède un système de gestion des fichiers à arborescence unique, même avec plusieurs périphériques de stockage
- dont les entrées/sorties et la communication inter-processus sont compatibles avec la notion de fichier (interface de manipulation unique)

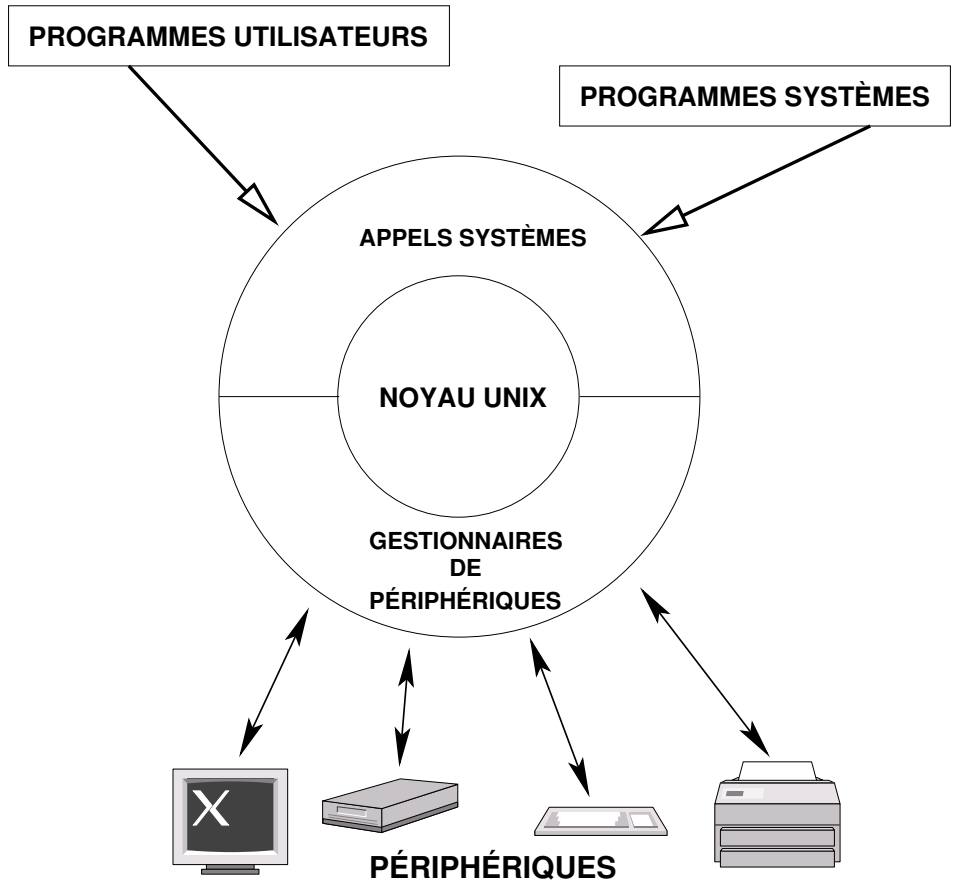
Terminologie



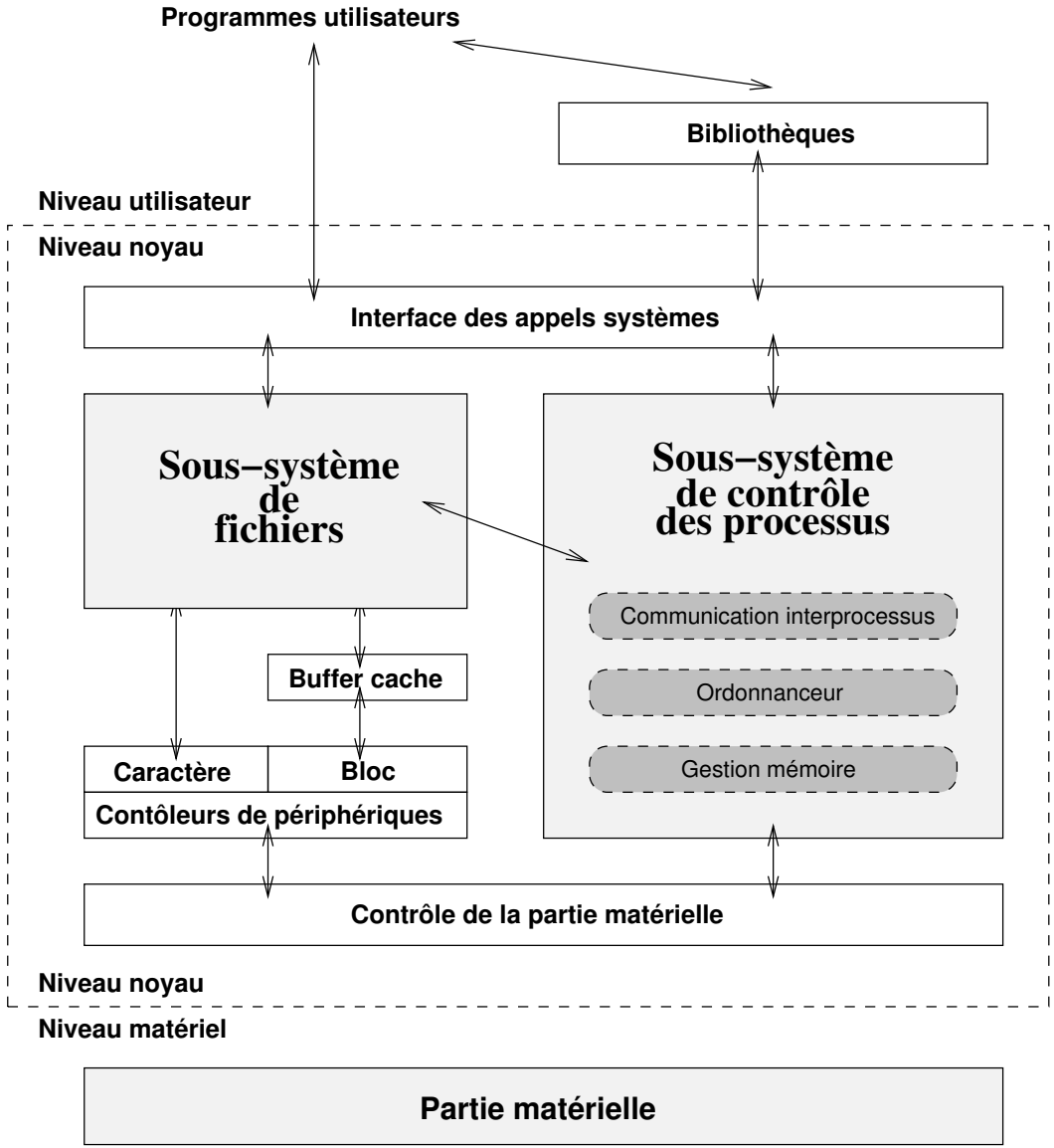
Une architecture en couche

Le fonctionnement d'UNIX est basé sur une architecture logicielle en couche :

- Programmes utilisateurs
- Programmes systèmes
- Noyau du système
- Matériel (« *hardware* »)



Le noyau d'Unix



Syntaxe générale des commandes UNIX

Les différents langages de commandes (« *shells* ») utilisent tous la même syntaxe générale pour la description d'une commande :

```
commande [options...] [arguments...]
```

Une commande peut être suivie de « *paramètres* » :

- des « *options* » ➥ COMMENT
  - pour préciser son fonctionnement
  - mot commençant généralement par le caractère -
- des « *arguments* » ➥ QUOI
  - pour spécifier des éléments que la commande doit prendre en compte
  - généralement pour identifier des fichiers

Beaucoup de libertés :

- chaque commande décide de sa syntaxe .... « *nature et ordre des paramètres* »
- une ligne peut comporter plusieurs commandes ..... « *séparation par ;* »

Documentation

man

```
man is the system's manual pager. Each page argument given to man is
normally the name of a program, utility or function.  The manual page
associated with each of these arguments is then found and displayed.
```

Documentation

man

```
man is the system's manual pager. Each page argument given to man is
normally the name of a program, utility or function.  The manual page
associated with each of these arguments is then found and displayed.
```

NAME	le nom et une description rapide de la commande
SYNOPSIS	toutes les possibilités de saisies liées à cette commande (syntaxe)
DESCRIPTION	une explication des conséquences de la commande
FILES	les fichiers modifiés par la commande ou nécessaires au moment de la saisie
OPTIONS	la liste des différentes options de cette commande
SEE ALSO	les références croisées vers d'autres commandes proches
DIAGNOSTICS	des explications sur les messages d'erreur
RETURN VALUES	ce que renvoie la commande
BUGS	des problèmes connus de cette commande
EXAMPLES	des exemples d'appel à cette commande
TIPS	des astuces pour utiliser cette commande.

Documentation

man

```
man is the system's manual pager. Each page argument given to man is
normally the name of a program, utility or function.  The manual page
associated with each of these arguments is then found and displayed.
```

```
1   Executable programs or shell commands
2   System calls (functions provided by the kernel)
3   Library calls (functions within program libraries)
4   Special files (usually found in /dev)
5   File formats and conventions eg /etc/passwd
6   Games
7   Miscellaneous (including macro packages and conventions)
8   System administration commands (usually only for root)
9   Kernel routines [Non standard]
```

`man(1)`, `info(1)`

Philosophie

Règle générale

Sous Unix **TOUT EST FICHIER** . . . ou presque

- En interne (vue du noyau)
  - les fichiers ont tous la même structure
- En externe (vue de l'utilisateur)
  - différents « *types* » de fichiers :
    - catalogues (ou répertoires)
    - liens symboliques
    - spéciaux
    - tubes
    - sockets
    - ordinaires (ou réguliers)
  - représentation hiérarchique du stockage des fichiers

Catalogues

Définition

Un catalogue (« *directory* »), ou répertoire, est un fichier qui contient une liste de fichiers.

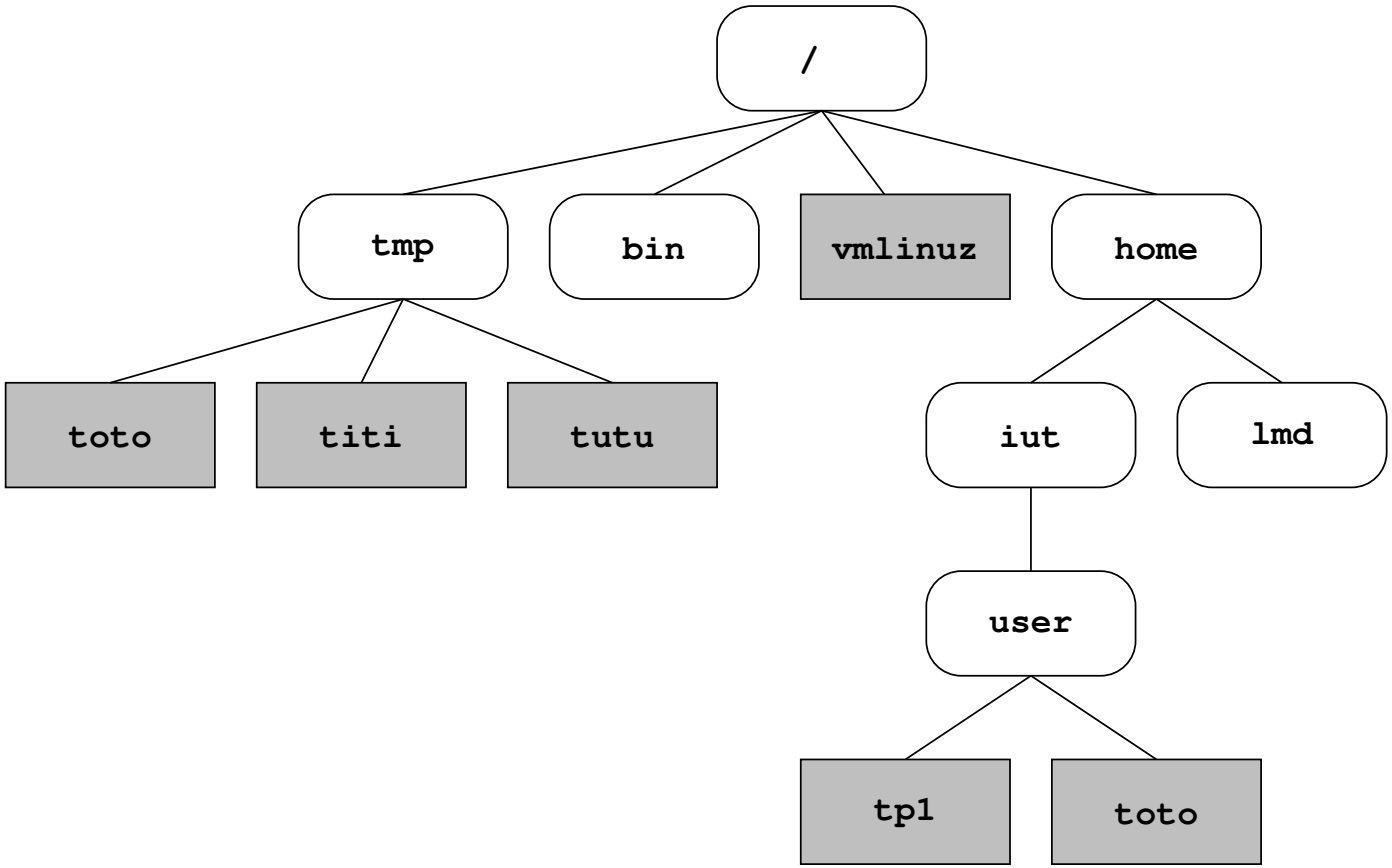
Un répertoire « *contient* » d'autres fichiers et peut donc contenir un ou des répertoires.

➠ Notion de hiérarchie (ou d'arbre)

Toutes les versions d'Unix ont une hiérarchie unique, dont le sommet est nommé **/** (« *slash* »).

Ce répertoire de base est la racine (« *root* ») de l'arbre hiérarchique.

Ce répertoire a toujours comme inode la valeur 2




```
/
|
|-- tmp
|   |
|   |-- toto
|   |-- titi
|   `-- tutu
|-- bin
|-- vmlinuz
`-- home
    |
    |-- iut
    |   |
    |   `-- user
    |       |-- tp1
    |       `-- toto
    `-- lmd
```

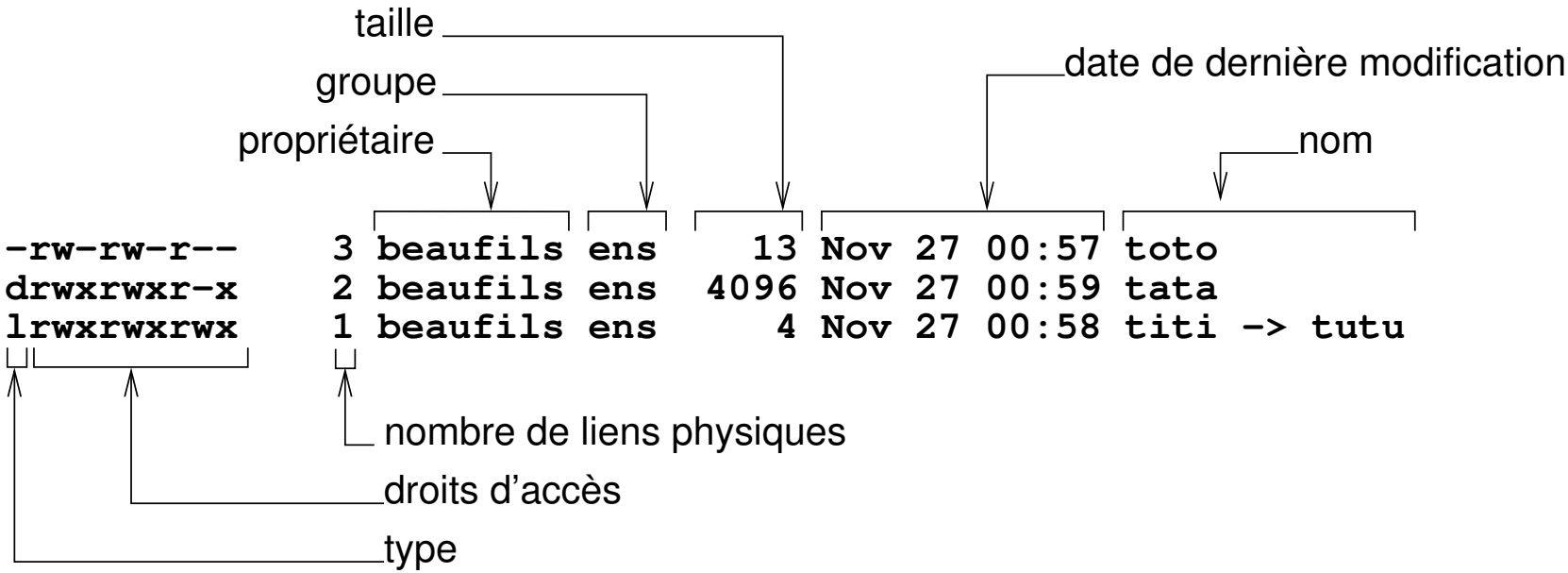
Quelques commandes utilisables à propos des répertoires :

`pwd`	permet d'obtenir le nom absolu du répertoire de travail courant
`cd`	permet de changer le répertoire de travail courant
`ls`	permet d'obtenir la liste des fichiers contenus dans un répertoire. Il existe de très nombreuses options parmi lesquelles : `-a` permet de voir les fichiers cachés `-i` permet de voir les inodes associées `-l` permet d'avoir les informations pour chaque fichier sur une ligne
`mkdir`	permet de créer un répertoire
`rmdir`	permet de supprimer un répertoire vide
`rm`	permet de supprimer une (ou des) entrée(s) du répertoire

Sans argument `ls` liste le contenu du répertoire de travail courant

`pwd(1)`, `ls(1)`, `mkdir(1)`, `rmdir(1)`, `rm(1)`

ls -l



Type	Caractères
fichier régulier	`-`
répertoire	`d`
lien (symbolique)	`l`
tube	`p`
socket	`s`
spécial	`c` ou `b`

Hiérarchie standard Unix

`/bin`	Commandes utilisateurs essentielles
`/dev`	Fichiers de périphériques
`/etc`	Fichiers de configuration spécifique à la machine
`/home`	Répertoires des utilisateurs
`/lib`	Librairies partagées
`/sbin`	Commandes d'administration essentielles
`/tmp`	Fichiers temporaires
`/usr`	Seconde hiérarchie
`/var`	Données variables
`/usr/bin`	La plupart des commandes utilisateurs
`/usr/include`	Fichier d'entêtes pour les programmes C
`/usr/lib`	Librairies
`/usr/local`	Hiérarchie locale
`/usr/sbin`	Commandes d'administrations non-vitales
`/usr/share`	Données indépendantes de l'architecture
`/usr/src`	Code source

Plus de détails sur l'effort de standardisation : `http://www.pathname.com/fhs/`

`hier(7)`

Tous les répertoires contiennent obligatoirement dans leur liste deux fichiers :

- `.` qui est un synonyme pour le répertoire lui-même
- `..` qui est un synonyme pour le répertoire qui le contient (son père)

Les fichiers dont le nom commence par un point `.` sont appelés « *fichiers cachés* » (par exemple par défaut la commande `ls` ne les montre pas).

Chemins

Pour identifier un fichier dans la hiérarchie on a besoin :

1. du chemin jusqu'au répertoire dans lequel il est stocké
2. de son nom de base

Définition

chemin	=	point de départ	+	liste des répertoires à traverser pour arriver au répertoire destination

La liste des répertoires est composée de répertoires séparés les uns des autres par le caractère **/**

- si le point de départ est la racine (et chemin le plus court possible)
  ➠ **chemin absolu**
- sinon
  ➠ **chemin relatif à un autre répertoire**

Manipulation du système de fichiers

Les commandes de manipulation des fichiers ont souvent la syntaxe suivante :

`commande` ⟨*source*⟩ ⟨*destination*⟩

- ⟨*source*⟩ et ⟨*destination*⟩ désigne chacun un chemin

`cp` permet de copier un fichier

`mv` permet de déplacer un fichier (donc aussi de le renommer)

`ln` permet de surnommer un fichier ou de créer un raccourci vers un fichier

`cp(1)`, `mv(1)`, `ln(1)`

Fichiers réguliers

Définition

Un fichier régulier est un fichier qui n'est ni un catalogue, ni un lien, ni un fichier spécial, ni un tube, ni une socket.

Quelques commandes utilisables sur des fichiers réguliers :

`stat`	permet d'afficher les caractéristiques de base de fichier(s)
`od`	permet d'afficher les octets d'un fichier sous différents formats
`cat`	permet d'afficher le contenu de fichier(s)
`touch`	permet de modifier les caractéristiques de dates de fichier(s). Cette commande permet également de créer un (ou des) fichier(s) vide(s)
`file`	permet de déterminer la convention de structure que respecte le contenu du fichier (donc son « *type applicatif* »)

`stat(1)`, `od(1)`, `cat(1)`, `touch(1)`, `file(1)`

Liens symboliques

Définition

Un **lien symbolique** (« *soft link* ») est un fichier (de type lien) qui contient le chemin et le nom d'un autre fichier.

Les accès à un lien ne sont rien d'autre que des redirections vers un autre fichier : les commandes qui manipulent un fichier lien manipule en fait le fichier dont le chemin est stocké dans le lien.

➠ Un lien est donc un raccourci (ou un alias) vers un autre fichier

Le contenu du fichier doit être un chemin

- soit absolu
- soit relatif. Dans ce cas le chemin doit être **valide depuis le répertoire dans lequel se trouve le fichier**.

Utilisateurs/Groupes

Unix est un système multi-utilisateurs. Les utilisateurs y sont rassemblés par groupe. Chaque utilisateur est donc identifié par le système par :

1. son « *login* » ..... au niveau noyau c'est un numéro unique : l'`uid`
2. son « *groupe* » ..... au niveau noyau c'est un numéro unique : le `gid`

Le système gère la correspondance entre identifiant symbolique et numérique via des fichiers textes :

- login et `uid` via le fichier `/etc/passwd`
- groupe et `gid` via le fichier `/etc/group`

Un utilisateur peut appartenir à plusieurs groupes, mais possède un groupe principal (spécifié dans le fichier `/etc/passwd`) dans lequel il est enregistré lors de chaque connexion.

Droits d'accès

Chaque fichier :

- appartient à un utilisateur (son « *propriétaire* ») et à un groupe.
- posssède des droits d'utilisation applicables :
  1. à son propriétaire
  2. aux utilisateurs appartenant à son groupe
  3. aux utilisateurs n'appartenant pas à son groupe

Pour chacune de ces trois catégories, il existe trois types de droits :

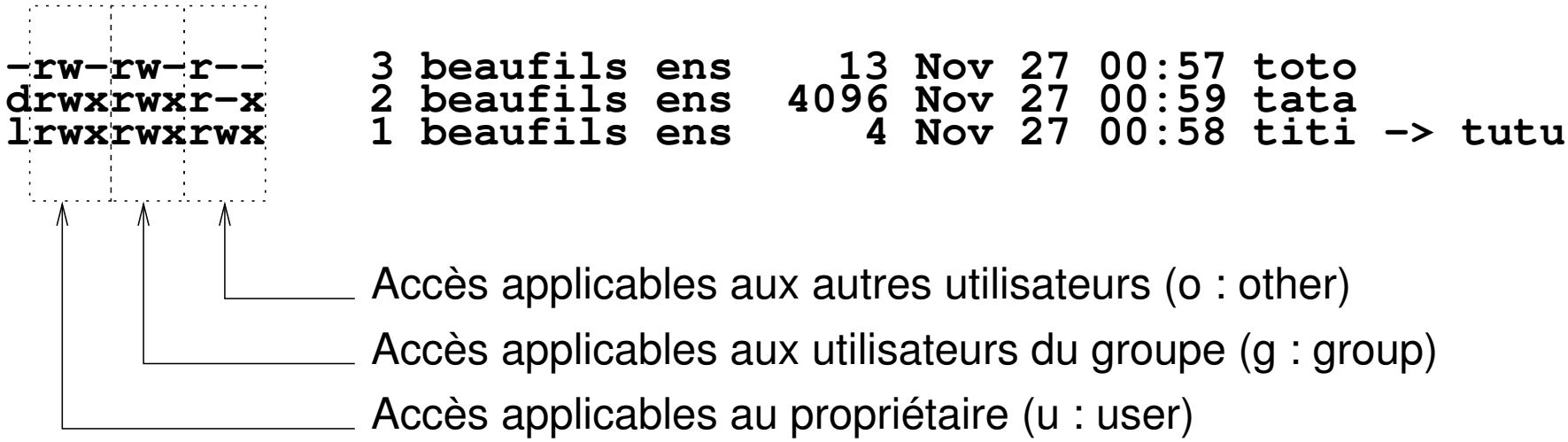
1. **lecture** : autorise la lecture du contenu du fichier
2. **écriture** : autorise la modification du contenu du fichier
3. **exécution/franchissement** :
   - autorise l'exécution d'un fichier régulier,
   - permet de traverser un répertoire

Remarque

Pour manipuler le système de fichier (copie, déplacement, etc.) un utilisateur doit avoir les droits correspondants sur les fichiers qu'il veut manipuler

L'option `-l` de la commande `ls` permet de voir les droits d'accès d'un fichier. Pour chacun des trois cas d'applicabilité les droits sont affichés par une chaîne de caractère avec la représentation suivante :

- `r` : l'accès en lecture est autorisé
- `w` : l'accès en écriture est autorisé
- `x` : l'accès en exécution/franchissement est autorisé
- `-` : à la place de `r`, `w` ou `x` signifie que l'accès correspondant n'est pas attribué.



chmod

Définition

Le mode d'utilisation d'un fichier est l'ensemble de ses droits d'accès.

La commande **chmod** permet au propriétaire d'un fichier de modifier son mode d'utilisation.

La syntaxe de **chmod** est la suivante :

chmod ⟨*mode*⟩ ⟨*fichiers*⟩

Le mode peut être précisé de deux manières :

- via la spécification des modifications à effectuer sur le mode courant :
  ➟ forme **symbolique**.
- via la spécification complète du nouveau mode :
  ➟ forme **numérique octale** (base 8)

chmod(1)

umask

Lorsqu'un programme crée un fichier, il spécifie les droits d'accès qu'il demande pour ce fichier.

Certains des droits demandés seront accordés d'autres seront refusés en fonction d'un « *masque de protection* » `AND NOT`.

La commande **`umask`** permet :

- de connaître la valeur du masque si elle est utilisée sans argument
- de modifier la valeur du masque si elle est utilisée avec un argument

Dans tous les cas elle utilise des masques sous forme numérique octale.

`bash(1)`

Structuration

Définition

Vue du noyau un fichier est une suite non-structurée d'octets (« *byte stream* »)

Pas de structuration directe au niveau du noyau mais possible au niveau des applications.

Exemple : les fichiers textes

- fichiers constitués d'une séquence de lignes.
- ligne constituée de caractères terminée par le caractère de passage à la ligne.
- caractère représenté par un octet suivant le code ASCII.
- caractère de passage à la ligne est le caractère de code 10 `\n`.

Cette structuration n'est qu'une convention utilisée par des programmes.

Inodes (1)

- Les caractéristiques d'un fichier sont stockées dans une structure de données ..... inode
- Le système gère une table de toutes les inodes disponibles ..... système de fichiers
- Une inode est repérée par son numéro dans cette table ..... son numéro d'inode
- Il y a un système de fichiers par zone de stockage matérielle (partition d'un disque dur par exemple)

Définition

Un fichier est repéré de manière unique par :

1. le système de fichier auquel il est attaché
2. son numéro d'inode

Inodes (2)

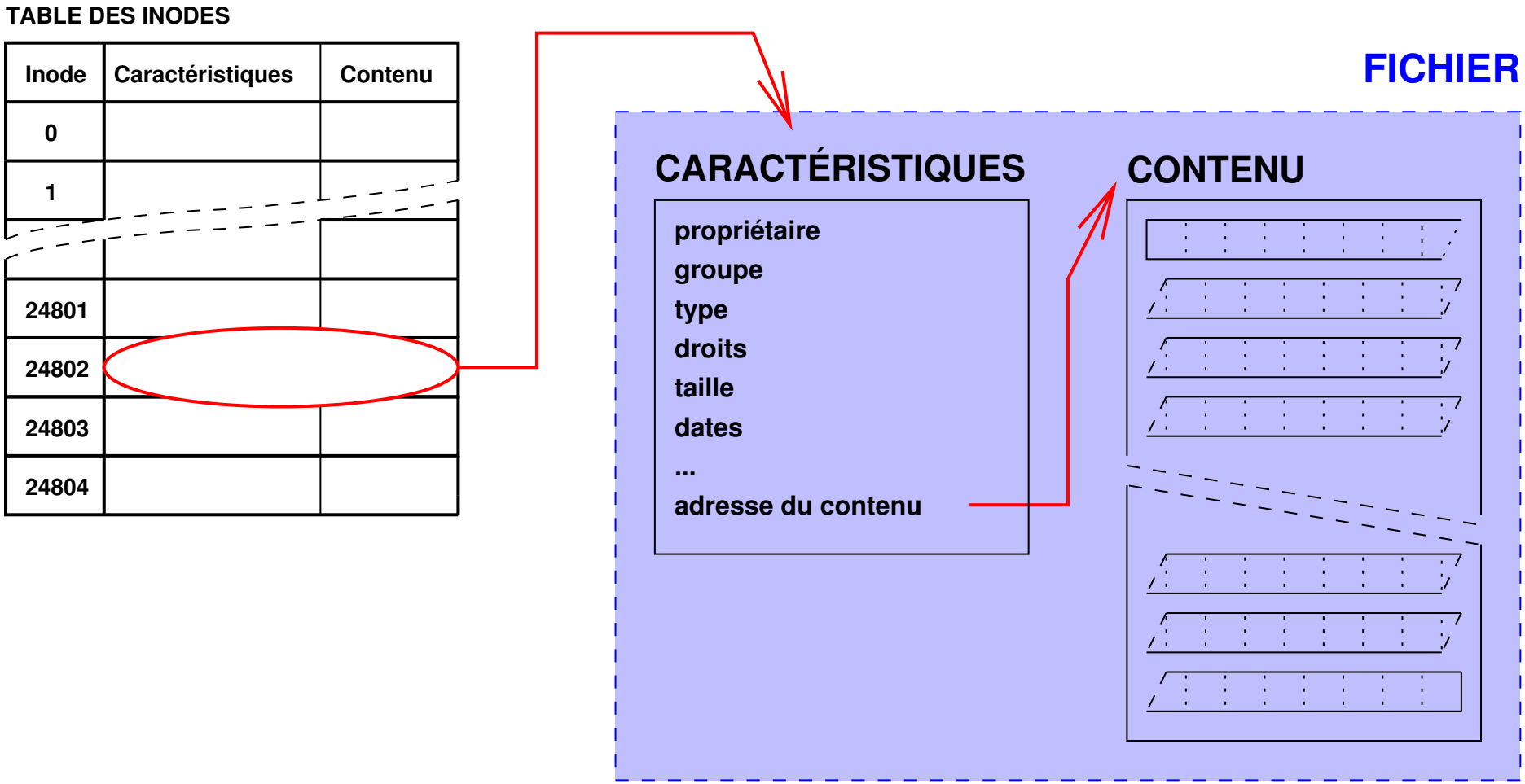
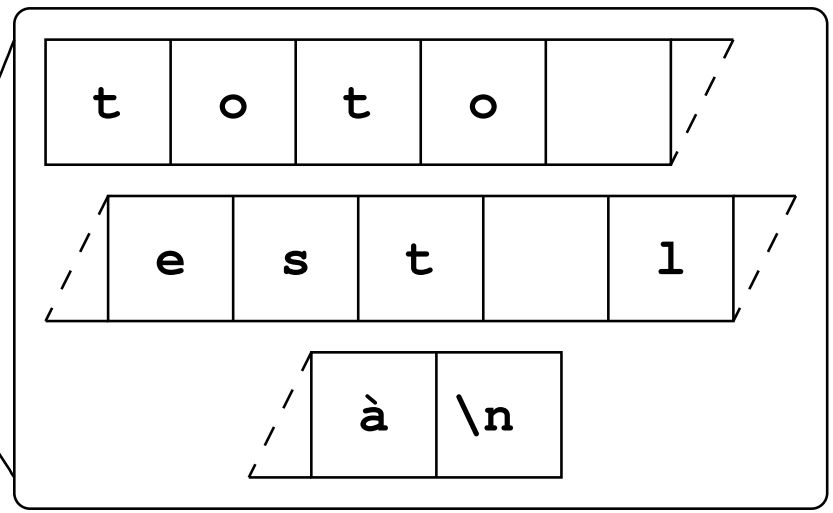


TABLE DES INODES
Inode
Caractéristiques
Contenu
0
1
24801
24802
24803
24804
FICHIER
CARACTÉRISTIQUES
propriétaire
groupe
type
droits
taille
dates
...
adresse du contenu
CONTENU

Fichiers réguliers (vue noyau)

TABLE DES INODES

Inode	Caractéristiques	Contenu
0		
1		
24801	... Répertoire ...	...
24802	... Régulier ...	
24803	... Lien ...	tutu



Fichiers catalogues (vue noyau)

Catalogues	=	Fichier comme un autre
	=	Caractéristiques + Contenu
Contenu	=	Liste de fichiers
	=	Ensemble de couples : (nom, inode)

TABLE DES INODES

Inode	Caractéristiques	Contenu
0		
1		
24801	... Répertoire ...	
24802	... Régulier ...	toto est là
24803	... Lien ...	tutu

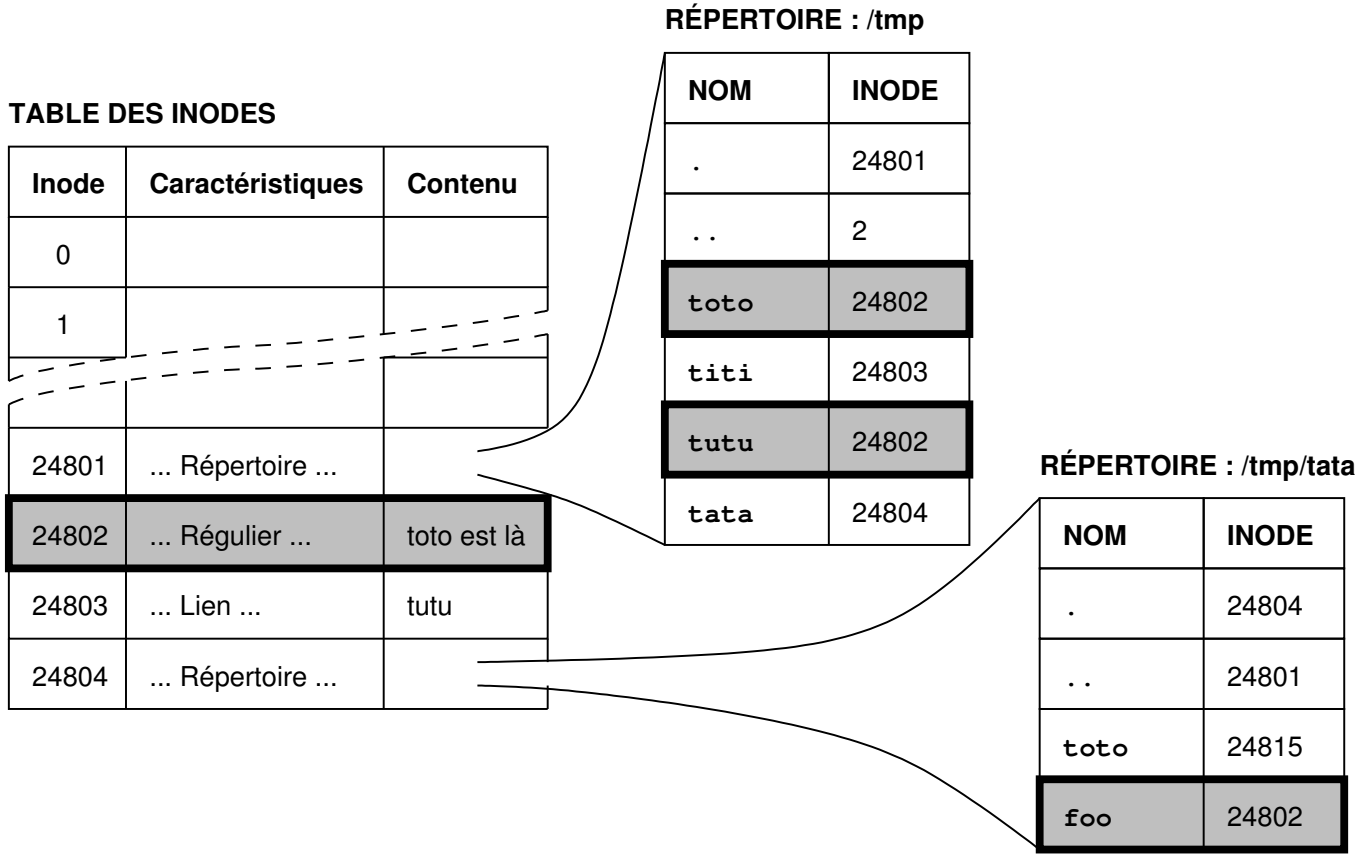
RÉPERTOIRE : /tmp

NOM	INODE
.	24801
..	2
`toto`	24802
`titi`	24803
`tutu`	24802
`tata`	24804

➥ « *entrée* » d'un répertoire (un des couples de la liste)

Noms des fichiers (vue noyau)

Le nom de fichier n'est rien d'autre qu'une entrée dans un répertoire. Un nom est donc un **lien physique** (« *hard link* ») vers un fichier.



Plusieurs entrées de répertoires peuvent utiliser la même inode.

➠ **Un même fichier peut avoir plusieurs noms.**

Abus de langage

Par abus de langage on a donc 2 notions différentes :

- les **liens physiques** ou **hard**, plusieurs entrées de répertoires utilisant la même inode ..... (fichiers de type régulier)
- les **liens symboliques** ou **soft**, plusieurs inodes différentes dont le contenu désigne un même fichier régulier ..... (fichiers de type lien)

La commande `ln` permet de créer des liens :

- sans option elle permet de créer des liens physiques
- avec l'option `-s` elle permet de créer des liens symboliques

Définitions

Un **programme** est une suite d'instructions que le système doit faire accomplir au processeur pour résoudre un problème particulier. Ces instructions sont rangées dans un fichier.

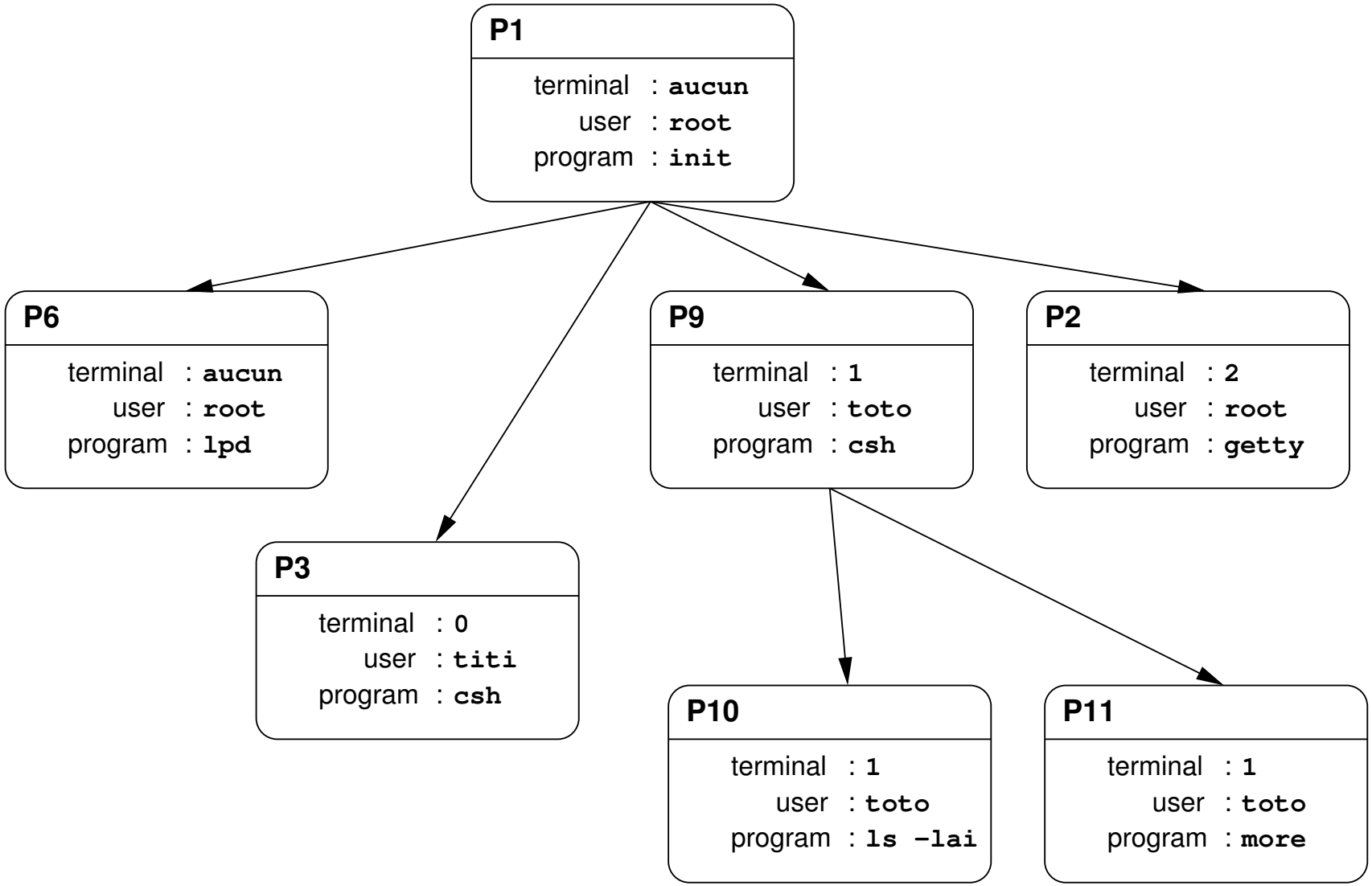
Définition

Un **processus** correspond au déroulement (« *l'exécution* ») d'un programme par le système dans un environnement particulier.

$$\begin{array}{ccc} \text{PROGRAMME} & \neq & \text{PROCESSUS} \\ \Updownarrow & & \Updownarrow \\ \text{Recette de cuisine} & \neq & \text{Préparation d'un plat} \end{array}$$

Généralités

- Chaque processus peut lui même démarrer d'autres processus ; dans ce cas le créateur est appelé le père et les processus qu'il a créé sont appelés ses fils.
  Notion d'arborescence des processus
- Au démarrage du système il n'existe qu'un seul processus qui est donc l'ancêtre de tous les autres (il exécute le programme `init`).
  Son rôle est de créer 2 type de processus :
  - **interactifs** associés à un terminal particulier
  - **non-interactifs** (« *daemons* ») rattachés à aucun terminal
- Les processus des utilisateurs sont démarrés par un processus interactif qui exécute un programme particulier : un interpréteur de commandes (« *shell* »).
- **Le shell démarre un processus pour chacun des ordres (commandes) de l'utilisateur associé.**


P1
terminal : aucun
user : root
program : init
P6
terminal : aucun
user : root
program : lpd
P3
terminal : 0
user : titi
program : csh
P9
terminal : 1
user : toto
program : csh
P2
terminal : 2
user : root
program : getty
P10
terminal : 1
user : toto
program : ls -lai
P11
terminal : 1
user : toto
program : more

Contexte d'exécution

- Le noyau maintient une table pour gérer l'ensemble des processus.
- Chaque processus est identifié par un index dans cette table
  son numéro d'identification ou **PID**
- Chaque entrée de la table correspond aux informations sur ce processus :
  - le numéro d'identification du processus père ..... **PPID**
  - l'identifiant de l'utilisateur qui exécute le processus ..... **UID**
  - l'identifiant du groupe de l'utilisateur qui exécute le processus ..... **GID**
  - le répertoire courant ..... **cwd**
  - la liste des fichiers utilisés par le processus
  - le masque de création des fichiers ..... **umask**
  - la taille maximale des fichiers que ce processus peut créer ..... **ulimit**
  - le terminal de contrôle associé
  - la zone mémoire associée ..... code, données, pile et tas

`ps(1)`, `kill(1)`, `proc(5)`

Représentation interne

Un processus est une zone mémoire de taille fixe qui permet de stocker :

- les informations sur le processus lui même
- le **code** : les instructions à exécuter (dans le langage du processeur)
- la **zone de données** : les variables manipulées par le code
- la **pile d'exécution** : les paramètres d'appels des fonctions

Un processus est donc représenté comme un programme qui s'exécute et qui possède son propre compteur ordinal (l'adresse en mémoire de la prochaine instruction à exécuter).

Les informations nécessaires au fonctionnement d'un processus (exécution, arrêt, reprise, etc.) constitue le **contexte d'exécution** de celui-ci.

Modes de fonctionnement

Un processus peut

- attendre la fin de son fils pour continuer ..... **avant plan** (« *foreground* »)
  mode synchrone : les processus s'exécutent en « *séquence* »
- ne pas attendre la fin de son fils pour continuer ..... **tâche de fond** (« *background* »)
  mode asynchrone : les processus s'exécutent en « *parallèle* »

Pour chaque commande exécutée le shell crée un nouveau processus.

- Par défaut en mode synchrone
- Les commandes peuvent être séparées par :
  - des **points-virgules** **;**
    Attendre la fin d'une commande avant de passer à la suivante
  - des **esperluètes** **&**
    Ne pas attendre la fin d'une commande pour passer à la suivante
  - des **tubes** **|**
    Démarrer les commandes en parallèle en les connectant

- Pour lancer une commande en avant-plan il suffit de taper cette commande :

```
$ commande
... résultat de la commande
$
```

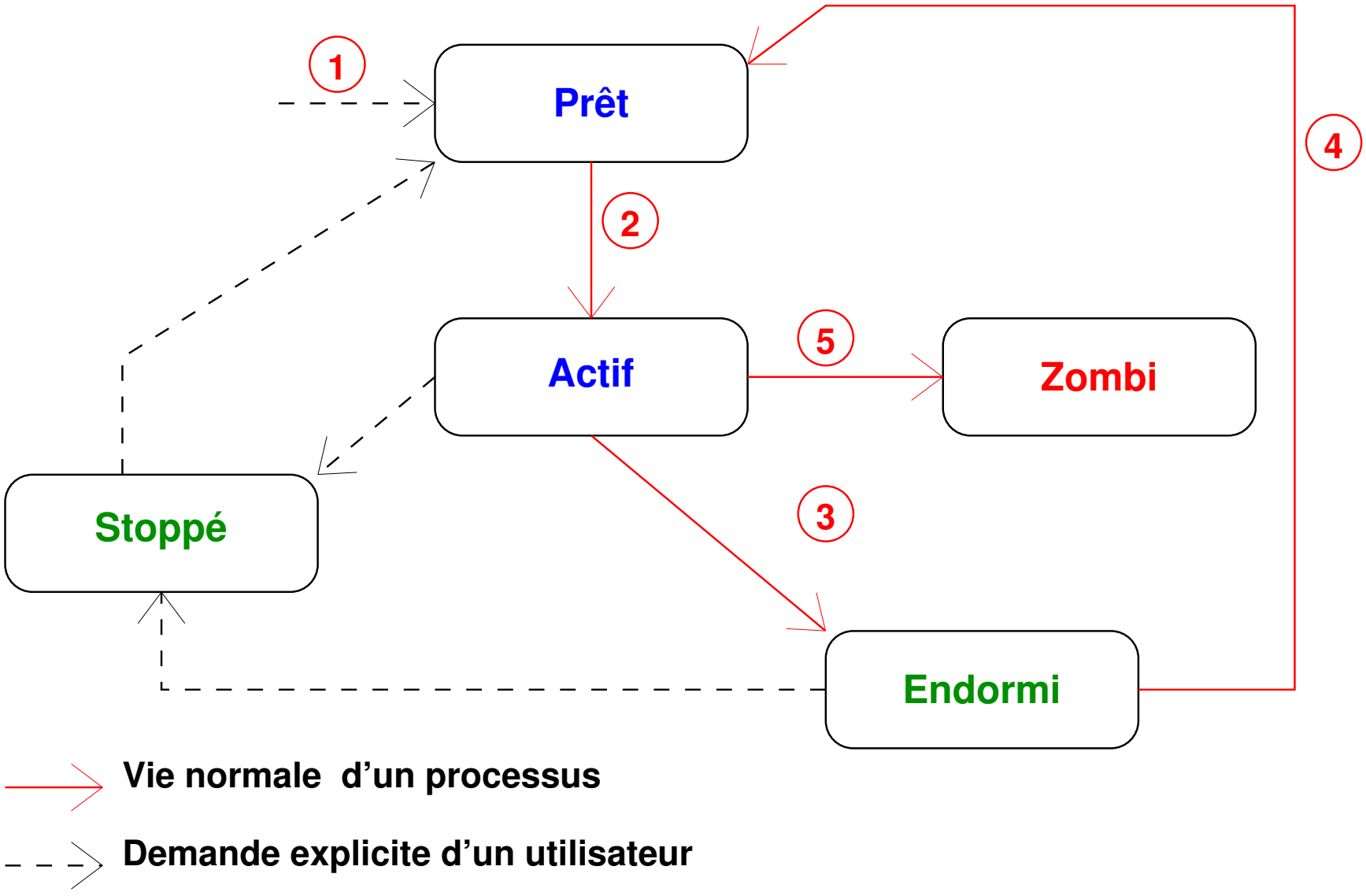
➥ Ce mode est le mode par défaut dans les shells.

- Pour lancer une commande en tâche de fond, il faut faire suivre cette commande par le caractère esperluète « `&` » :

```
$ commande &
[1] 31343
$
... résultat de la commande
```

Le Bourne shell (`sh`) affiche un numéro de tâche (« *job* ») entre crochets puis le PID du processus créé avant de rendre la main à l'utilisateur.

Changements d'états



Entrées/Sorties

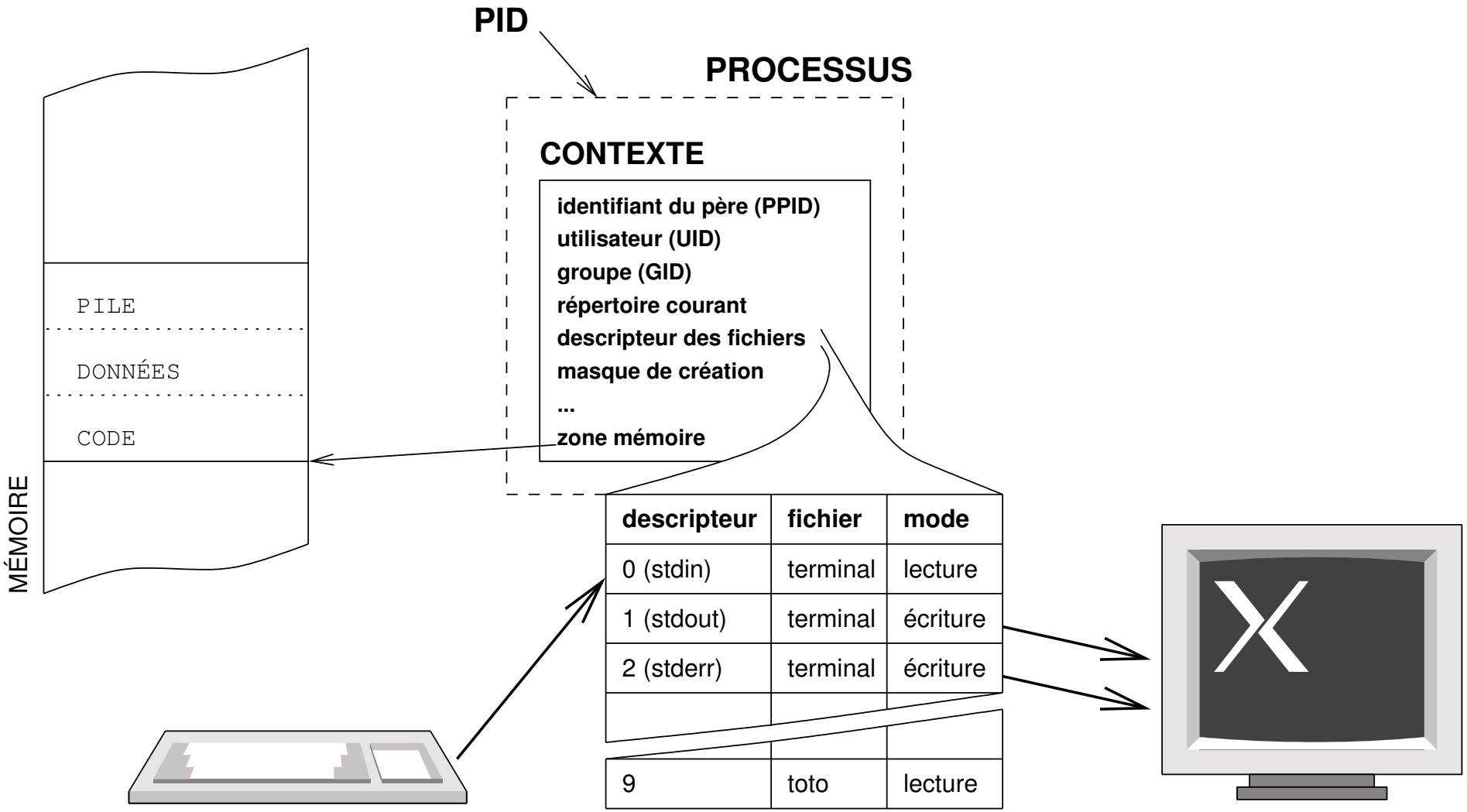
Tous les processus gère une table stockant le nom des différents fichiers qu'ils utilisent. Chaque index de cette table est appelé un « *descripteur de fichiers* ».

Par convention les trois premiers descripteurs correspondent à :

1. **l'entrée standard** :
   si le programme exécuté par le processus a besoin de demander des informations à l'utilisateur il les lira dans ce fichier (par défaut c'est le terminal en mode lecture).

2. **la sortie standard** :
   si le programme a besoin de donner des informations à l'utilisateur il les écrira dans ce fichier (par défaut c'est le terminal en mode écriture).

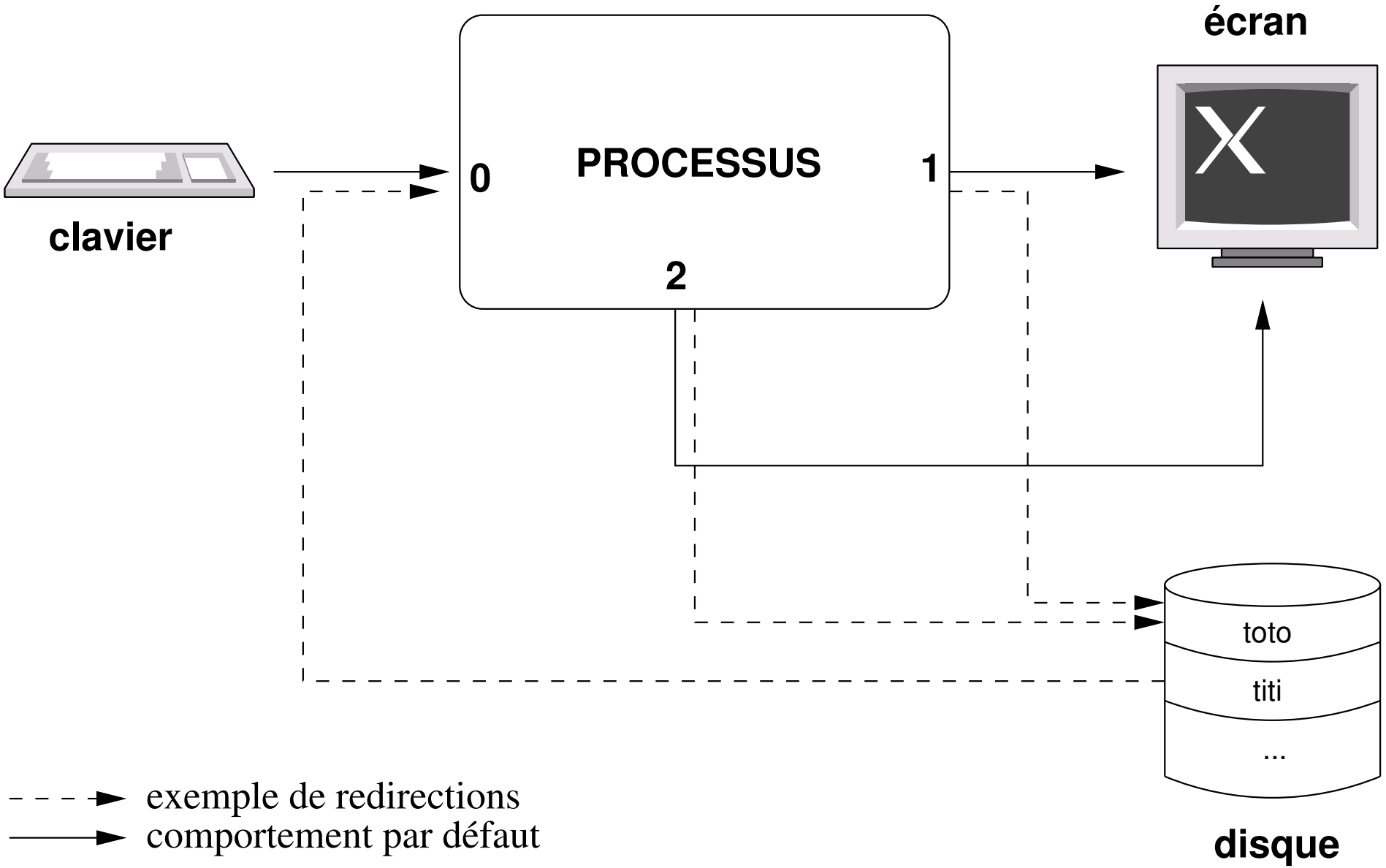
3. **la sortie d'erreur** :
   si le programme a besoin d'envoyer un message d'erreur à l'utilisateur il l'écrira dans ce fichier (par défaut c'est le terminal en mode écriture).


PID
PROCESSUS
CONTEXTE
identifiant du père (PPID)
utilisateur (UID)
groupe (GID)
répertoire courant
descripteur des fichiers
masque de création
...
zone mémoire
PILE
DONNÉES
CODE
MÉMOIRE
descripteur
fichier
mode
0 (stdin)
terminal
lecture
1 (stdout)
terminal
écriture
2 (stderr)
terminal
écriture
9
toto
lecture

Redirections

En shell, il est possible de modifier les fichiers identifiés par les descripteurs :

- Redirection de la sortie standard avec le caractère plus grand **>** :
  - **`commande > fichier`**
    Si le fichier n'existe pas, il est crée par le shell et s'il existe déjà le shell détruit son contenu pour le remplacer par la sortie de la commande
  - **`commande >> fichier`**
    Si le fichier n'existe pas, il est crée par le shell et s'il existe déjà la sortie de la commande est ajoutée à la fin du fichier.
- Redirection de l'entrée standard avec le caractère plus petit petit **<** :
  - **`commande < fichier`**
    La commande lit ses données dans le fichier.


écran
clavier
0
PROCESSUS
1
2
toto
titi
...
exemple de redirections
comportement par défaut
disque

Syntaxe générale des redirections

⟨*n*⟩`<`⟨*fichier*⟩	redirige le descripteur numéro *n* en lecture vers ⟨*fichier*⟩.
⟨*n*⟩`>`⟨*fichier*⟩	redirige le descripteur numéro *n* en écriture vers ⟨*fichier*⟩.
⟨*n*⟩`<<`⟨*marque*⟩	redirige le descripteur numéro *n* en lecture vers les lignes suivantes jusqu'à ce que la ⟨*marque*⟩ soit lue.
⟨*n*⟩`>>`⟨*fichier*⟩	redirige le descripteur numéro *n* à la fin de ⟨*fichier*⟩ sans détruire les données préalablement contenues dans ce fichier.
⟨*n*⟩`<&`⟨*m*⟩	duplique le descripteur numéro *n* sur le descripteur numéro *m* en lecture, ainsi *n* et *m* seront dirigés vers le même fichier.
⟨*n*⟩`>&`⟨*m*⟩	duplique le descripteur numéro *n* sur le descripteur numéro *m* en écriture.

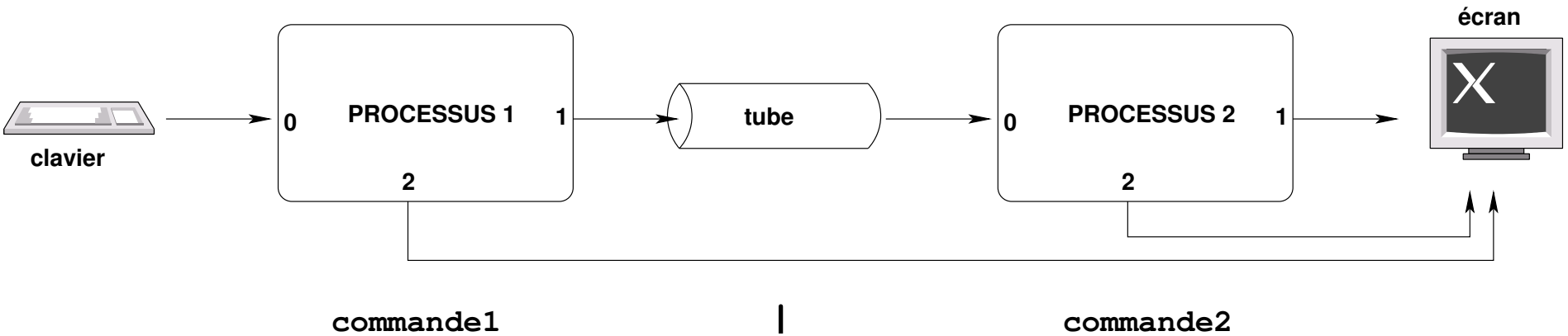
➥ Il est possible de mettre autant de redirections que voulues sur une ligne de commandes.

Communication inter-processus

Il est possible d'avoir plusieurs processus fonctionnant en « *parallèle* » qui communiquent entre eux par le biais de **tubes** (« *pipes* »). Le système assure alors la synchronisation de l'ensemble des processus ainsi lancés.

Le principe est assez simple :

> *La sortie standard d'un processus est redirigée vers l'entrée d'un tube dont la sortie est dirigée vers l'entrée standard d'un autre processus.*



Le lancement concurrent de processus communiquant deux par deux par l'intermédiaires des tubes sera réalisé par une commande de la forme :

```
commande1 | commande2 | ... | commandeN
```

- Ce mécanisme est une des forces d'UNIX :
  *un ensemble de petits programmes **fiables** qui communiquent entre eux via le système d'exploitation.*

Il existe de nombreuses commandes UNIX qui profitent de ce genre de communication, notamment les « *filtres* » :

Définition

des programmes qui lisent des données sur l'entrée standard, les modifient et envoient le résultat sur la sortie standard

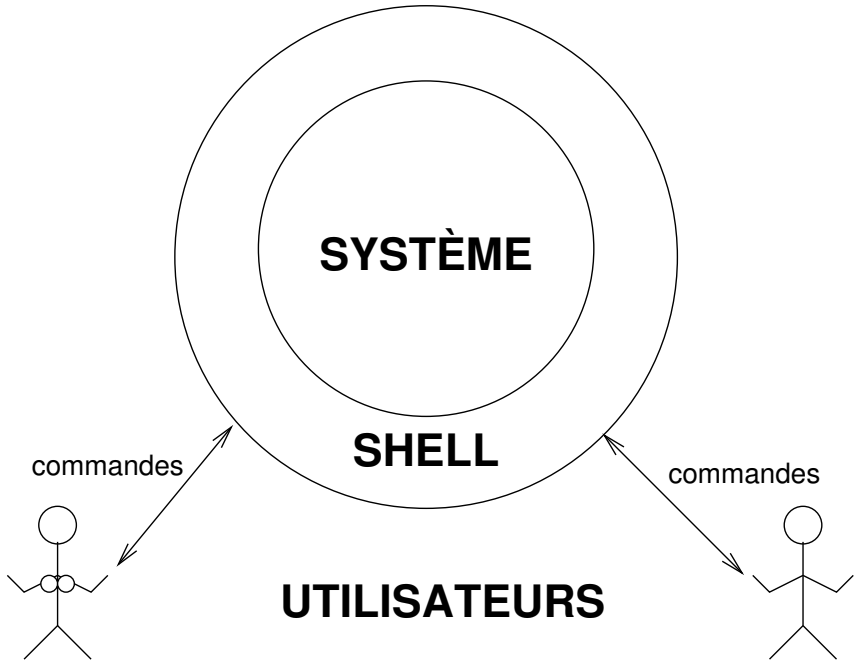
Quelques filtres

`cat`	retourne les lignes lues sans modification.
`cut`	ne retourne que certaines parties de chaque lignes lues.
`grep`	retourne uniquement les lignes lues qui correspondent à un modèle particulier ou qui contiennent un mot précis.
`head`	retourne les premières lignes lues.
`more`	retourne les lignes lues par bloc (dont la taille dépend du nombre de lignes affichables par le terminal) en demandant une confirmation à l'utilisateur entre chaque bloc.
`sort`	trie les lignes lues.
`tail`	retourne les dernières lignes lues.
`tee`	envoie les données lues sur la sortie standard **ET** dans un fichier passé en paramètre.
`tr`	remplace des caractères lus par d'autres.
`uniq`	supprime les lignes identiques.
`wc`	retourne le nombre de caractères, mots et lignes lus.
`sed`	édite le texte lu (requêtes `ed` comme avec la directive `:` de `vi`).

➜ Chacune de ces commandes possèdent de nombreuses options décrites dans le manuel.

Langages de commandes

Un langage de commande (« *shell* ») est un programme capable d'interpréter des commandes qui seront exécutées par le système d'exploitation.



- Interface entre le système d'exploitation et l'utilisateur.
- Permet d'écrire des programmes comportant plusieurs commandes.

Il existe un grand nombre de shells différents séparés, essentiellement par la syntaxe, en 2 grandes familles :

1. ceux dérivant du **Bourne-shell** (`/bin/sh`)
   Historiquement le premier shell (écrit par Steve Bourne).
   Plutôt orienté programmation qu'interaction.
   - **Bourne Again SHell** (`/bin/bash`)
     une implémentation du Bourne shell faite par le projet GNU.
   - **Korn SHell** (`/bin/ksh`)
     écrit par David Korn.

2. ceux dérivant du **C-shell** (`/bin/csh`)
   Syntaxe très proche de celle du langage C.
   Plutôt orienté interaction que programmation.
   - le **Tenex C-SHell** (`/bin/tcsh`) implémentation libre du C-shell (beaucoup de fonctionnalités dédiées interaction).

➠ nous allons étudier le Bourne Shell via `bash`

Fichier de commandes

Un programme shell (« *script* ») :

- est une commande constituée d'appel à d'autres commandes shells
- est écrit dans un simple fichier texte :
  1. la **première ligne** du fichier définie le « *langage* » à utiliser :
     #! ⟨*emplacement_du_shell*⟩
  2. le fichier doit être exécutable.
  3. le shell doit pouvoir trouver le fichier.
  4. le caractère **#** permet d'insérer des commentaires dans le fichier.

Un script est une commande comme une autre sur laquelle on peut faire redirections, tubes, etc.

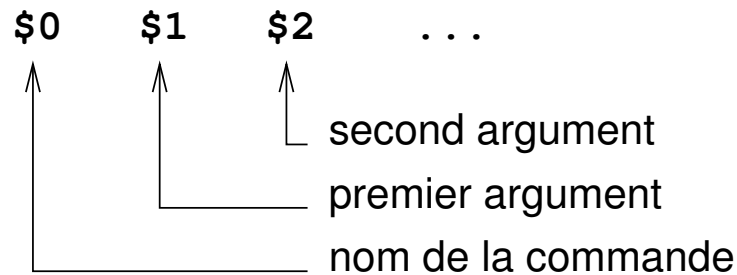
Paramètres de script (1)

Comme toute commande un script peut être appelé avec des paramètres :

- pour modifier son comportement
- pour spécifier les données qu'il doit manipuler

Chacun des paramètres passés sur la ligne de commandes :

- est repéré par sa position « *paramètres positionnels* »
- est utilisable dans le script via des « *variables* »



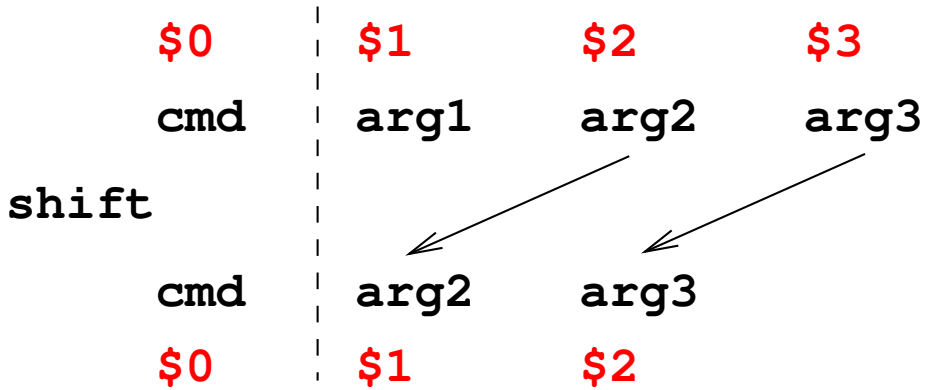
Paramètres de script (2)

Lors de l'exécution le shell remplace **automatiquement** certains « *mots* » :

`$0`	le nom du script tel qu'il a été appelé
`$1`	le premier mot apparaissant après le nom du script
`$2`	le second mot apparaissant après le nom du script
⋮	
`$n`	le n^{e} mot apparaissant après le nom du script
`$#`	le nombre de paramètres passés au script
`$*`	**une chaîne** contenant tous les paramètres passés au script (à partir de `$1`) séparés les uns des autres par **un** espace
`"$@"`	autant de chaîne que de paramètres passés au script

shift

La commande **shift** permet de décaler les paramètres positionnels vers la gauche.



Par défaut **shift** décale un argument à gauche.

Avec un nombre entier en paramètre **shift** décale plusieurs arguments.

Algorithme classique d'un shell

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

`bash(1)`, `dash(1)`, `csh(1)`, `ksh(1)`

Transformations successives de la ligne

La « *ligne* » lue subie plusieurs développements (« *expansion* ») avant exécution :

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

Protections

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

- **Protection d'un caractère** : faire précéder le caractère à protéger d'une barre de fraction inversée (« *backslash* ») \
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

- **Protection complète** : entourer la zone à protéger par des guillemets simples (« *quote* ») '
  ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.
- **Protection simple** : entourer la zone à protéger par des guillemets doubles (« *double-quote* ») "
  ➠ ormis \, $ et ‘ aucun caractère spécial n'est plus interprété.

Variables

En shell, comme dans tous langages de programmation il existe une notion de variables avec quelques spécificités :

- Les noms de variables sont des identificateurs ne comprenant que des lettres, des chiffres ou le caractère de soulignement _.

- Il n'existe pas de types de variables
  ➠ toutes les valeurs sont considérées comme des suites de caractères

- Il n'y a pas de réévaluation des variables
  ➠ une variable ne peut être modifiée que par une affectation

Affectation des variables

- **Variables classiques** :
  elles ne sont définies que dans le contexte d'exécution du processus dans lequel elles sont déclarées.

  ⟨*nom*⟩=⟨*valeur*⟩

- **Variables d'environnement** :
  elles sont définies dans le contexte d'exécution du processus dans lequel elles sont déclarées et dans tous les contextes d'exécution des processus que celui-ci peut créer (processus fils)

  ⟨*nom*⟩=⟨*valeur*⟩
  `export` ⟨*nom*⟩

`env(1)`

Développement des variables

`${`⟨*nom*⟩`}`

- le shell substitue la variable par la dernière valeur qui lui a été affecté
- les accolades `{` et `}` sont optionnelles, elles sont cependant souvent utilisées pour délimiter le nom de la variable
- si la variable n'existe pas le shell substitue par une chaîne vide

Quelques variables particulières

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

Substitution des commandes

Il est possible de remplacer un bout de la ligne de commande par le résultat de l'exécution d'une commande :

1. la zone représentant la commande à exécuter doit
   - être entourée de guillemet inverse (« *back-quote* ») `‘`
   - ou commencer par `$(` et se terminer par `)`
2. un processus fils (sous-shell) exécutant la commande située dans la zone entourée est créé
3. la sortie standard de ce processus est capturée et remplace la zone sur la ligne de commande.

```
echo 2 + 3 = ‘expr 2 + 3‘
```

⇓

```
echo 2 + 3 = 5
```

⇓

```
2 + 3 = 5
```

Développement des chemins de fichiers

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

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

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

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

Jokers (« *Méta-caractères* »)

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

Exemples de modèles de chemins

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

Découpage de la ligne en commandes

- Les mots sont séparés par des **blancs** non protégés
  Un blanc est un caractère espace ou une tabulation

- Une commande est un mot quelconque :
  - situé en **première position** de la ligne
  - ou situé **juste après un séparateur** de commandes
  - éventuellement **suivies** par des paramètres (d'autres mots)

- Les commandes peuvent être séparées par :
  - des **points-virgules** ;
    Attendre la fin d'une commande avant de passer à la suivante
  - des **esperluètes** &
    Ne pas attendre la fin d'une commande pour passer à la suivante
  - des **tubes** |
    Démarrer les commandes en parallèle en les connectant

Opérateurs

Autres séparateurs possibles : opérateurs logiques séquentiels paresseux

- **ET** `cmd1 && cmd2`
  `cmd2` est exécutée si et seulement si `cmd1` a réussi
- **OU** `cmd1 || cmd2`
  `cmd2` est exécutée si et seulement si `cmd1` a échoué
- **Le shell essaie de faire réussir la ligne**
  - évaluation de gauche à droite
  - dès qu'on sait que la séquence ne peut pas réussir (ou qu'elle est déjà réussie) on arrête son évaluation

Définition de réussite (échec) dans le manuel

Recherche et exécution de la commande

1. Si la commande est interne elle est exécutée directement

2. Sinon
   1. Recherche répertoire et fichier
      - si la commande contient au moins un caractère `/`
        extraction du répertoire et du nom de fichier
      - sinon pour tous les répertoires définis dans la variable `$PATH`
        recherche d'un fichier correspondant à la commande
   2. Exécution ou erreur
      - si un fichier a été trouvé **et qu'il est exécutable**
        exécution du code qu'il contient dans un nouveau processus
      - sinon envoi d'un message d'erreur

Commandes internes

Les commandes internes (« *builtins commands* ») sont traités directement :

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

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

Documentées dans la page du manuel de `bash`

`bash(1)`

Commandes externes

- code stocké dans un **fichier régulier exécutable**
- rangée dans un répertoire de la hiérarchie du système
  la convention est d'utiliser des répertoire nommés `bin`
- recherchée dans une liste de répertoires
  répertoires séparés par des `:` dans la variable `PATH`
- exécutée dans un nouveau processus par le shell
- elles **ne peuvent pas** modifier le contexte d'exécution du shell

Quelques exemples :

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

Structure pour

```
for var in ⟨liste⟩
do
    ⟨cmds⟩
done
```

- ⟨*cmds*⟩ exécutés autant de fois qu'il y a d'élément dans ⟨*liste*⟩
- Pour chaque tour **`$var`** a comme valeur un des éléments de ⟨*liste*⟩.
- Éléments utilisés de la gauche vers la droite de la liste
- ⟨*liste*⟩ est définie après les développements du shell

Code de retour

Sous UNIX toutes les commandes ont un code de retour :

- invisible sur la sortie standard
- visible pour le shell via une variable (`$?`)
- convention :
  - si la commande **réussit** le code de retour **vaut 0**
  - si elle **échoue** le code de retour **est différent de 0**
- le code de retour vu comme le **nombre d'erreurs**

➠ possibilité de faire des actions conditionnées au résultat d'autres actions

Vrai ≡ **réussite** ≡ code de retour = 0
Faux ≡ **échec** ≡ code de retour ≠ 0

Structure si

if ⟨*cmd-si*⟩
then
    ⟨*cmds-if*⟩
elif ⟨*cmd-sinon-si*⟩
    ⟨*cmds-elif*⟩
else
    ⟨*cmds-else*⟩
fi

1. si ⟨*cmd-si*⟩ réussit alors ⟨*cmds-if*⟩ exécutés
2. sinon
    1. si ⟨*cmd-sinon-si*⟩ réussit alors ⟨*cmds-elif*⟩ exécutés
    2. sinon ⟨*cmds-else*⟩ est exécutés

Structure tant que

while ⟨*cmd-tq*⟩
do
    ⟨*cmds-while*⟩
done

1. ⟨*cmd-tq*⟩ est exécutée
2. si elle a réussit
    1. les ⟨*cmds-while*⟩ sont exécutées
    2. retour en 1

Structure case

```
case ⟨mot⟩ in
    ⟨val-A-1⟩|⟨val-A-2⟩)
        ⟨cmds-case-A⟩
    ;;
    ⟨val-B-1⟩|⟨val-B-2⟩)
        ⟨cmds-case-B⟩
    ;;
    *)
        ⟨cmds-case-default⟩
    ;;
esac
```

La valeur de la variable ⟨*mot*⟩ est comparée en séquence avec chacun des choix fournit. Si elle correspond à un des choix alors les commandes spécifiées sont exécutées et les choix suivants sont oubliés.

Les choix peuvent faire l'objet d'expansion des noms génériques, de sorte qu'il est souvent fait usage d'un choix placé en dernier correspondant au choix par défaut grâce au caractère `*`.

Modes de fonctionnement

Tous les shells ont 3 modes de fonctionnement :

1. login interactif ..... (connexion à la machine)
2. shell interactif ..... (appel de **`bash`**)
3. shell non-interactif ..... (script)

Initialisation

Certaines commandes sont exécutées au démarrage de chacun des modes :

1. dans le processus d'un login interactif
   1. les commandes du fichier `/etc/profile` sont lues et exécutées
   2. les commandes du fichier `${HOME}/.bash_profile` sont lues et exécutées
   3. les commandes du fichier `${HOME}/.profile` sont lues et exécutées
2. dans le processus d'un shell interactif les commandes du fichier `${HOME}/.bashrc` sont lues et exécutées
3. dans le processus d'un shell non interactif si la variable `BASH_ENV` a une valeur le shell considère que son contenu (après transformation) est le nom d'un fichier dont les commandes doivent être lues et exécutées

man dash

The shell is a command that reads lines from either a file or the terminal, interprets them, and generally executes other commands. It is the program that is running when a user logs into the system (although a user can select a different shell with the chsh(1) command). The shell implements a language that has flow control constructs, a macro facility that provides a variety of features in addition to data storage, along with built in history and line editing capabilities. It incorporates many features to aid interactive use and has the advantage that the interpretative language is common to both interactive and non-interactive use (shell scripts). That is, commands can be typed directly to the running shell or can be put into a file and the file can be executed directly by the shell.

Algorithme du shell

1. Si processus interactif alors envoyer un message sur la sortie standard
2. Lire une ligne
3. Interpréter cette ligne :
    1. Développer par étapes
    2. Exécuter :
    3. Pour chacune des commandes :
        1. Recherche de la commande
        2. Si la commande est trouvée exécution de la commande, sinon envoi d'un message d'erreur sur la sortie d'erreur
4. Retour en 1

Les commandes

1. **fonctions**
   - paramètres positionnels (sauf `$0`) remplacés
   - la fonction est exécutée sans démarrer de nouveau processus
   - paramètres positionnels initiaux refixés
2. **commandes internes**
   exécutées par le shell sans démarrer de nouveau processus
   `cd`, `pwd`, `exit`, `exec`, `shift`, etc.
   mais aussi `echo`, `test`, `true`
3. **commandes externes**
   1. recherche d'un fichier exécutable si pas de `/` dans le mot
   2. démarrage d'un processus héritant de l'environnement courant

Les fonctions

[`function`] ⟨*nom*⟩ `() { ` ⟨*commandes*⟩ `; }`

- appelée comme une commande (même syntaxe)
- exécutée comme une commande **mais** dans l'environnement courant
- paramètres de la fonction deviennent les paramètres de position
  - uniquement durant l'exécution de la fonction
  - à l'exception de `$0` qui demeure inchangé.
- statut de retour de la fonction est :
  - le retour de la dernière commande exécutée dans la fonction
  - ou l'argument de la commande **`return`** s'il est présent.

Les fonctions (exemples)

```
bash$ cat scriptTITI
#!/bin/bash
TITI=essai2
export TITI

bash$ function toto () { echo "coucou";}
bash$ toto
coucou
bash$ function defTOTO () { TOTO=essai; export TOTO; }
bash$ defTOTO
bash$ echo $TOTO
essai
bash$ scriptTITI
bash$ echo $TITI

bash$
```

Regroupement de commandes

Intéressant pour traiter plusieurs commandes comme un tout :

- la sortie standard
- le code de retour

2 regroupements possibles :

- Les commandes sont exécutés dans le shell courant :

```
{ commande1 ; commande2 ; ... ; commandeN ; }
```

- Les commandes sont exécutés dans un sous-shell (nouveau processus) :

```
( commande1 ; commande2 ; ... ; commandeN )
```

En pratique

- Les parenthèses sont des caractères spéciaux
- Les accolades sont des mots réservés

Expressions arithmétiques

`((` ⟨*expression*⟩ `))`

permet d'utiliser des opérateurs similaires à ceux du C

- Incrémentation, décrémentation : `++` et `--`
- Opérations arithmétiques : `+,-,*,/,%,**`
- Opérations logiques : `!,&&,||`
- Opérations bit à bit : `~,&,^,|,<<,>>`
- Comparaisons : `<= >= < > == !=`
- Expression conditionelle : `expr?expr:expr`
- Affectation : `=,*=,/=,%=,+=,-=,<<=,>>=,&=,^=,|=`

Retoune `1` si l'expression est nulle, `0` sinon.

Expressions arithmétiques (exemples)

```
bash$ toto=2
bash$ ((toto++))
bash$ echo $toto
3
bash$ ((2+3))
bash$ echo $?
0
bash$ ((titi=2+3))
bash$ echo $titi
5
bash$ (( 2 < 3 ))
bash$ echo $?
0
bash$ (( 3 < 2 ))
bash$ echo $?
1
```

Expressions conditionnelles

`[[` ⟨*expression*⟩ `]]`

- Permet d'évaluer des expressions combinées par `()`, `!`, `||`, `&&` (avec short cut)
- Utilise les expressions conditionnelles vues pour `test`
- Avec les opérateurs `==`, `!=`, la deuxième opérande est considérée comme un pattern, avec `=~` comme une expression régulière étendue
- retourne `0` pour vrai

Expressions conditionnelles (exemples)

```
bash$  [[ 1 -gt 2 ]]
bash$ echo $?
1
bash$  [[ 1 -lt 2 ]]
bash$ echo $?
0
bash$ [[(1 -lt 2) && ("aaa" == "aaa")]]
bash$ echo $?
0
bash$ [[($a -lt 2) && ($b == $b)]]
bash$ echo $?
0
bash$ [[ $b == a?a ]]
bash$ echo $?
0
```

Forcer l'évaluation

La commande interne suivante permet de « forcer » l'évaluation des arguments.

```
eval [argument ...]
```

- Les arguments sont évalués et concaténés en une seule commande qui est exécutée par le shell.
- La valeur de retour de la commande est retournée comme valeur du `eval`. S'il n'y a pas d'argument ou uniquement des arguments vides, la valeur de retour est 0.

Exemple.

```
bash$ x=2
bash$ y='$x'
bash$ echo $y
$x
bash$ eval echo $y
2
```

Tableaux

- Tableaux unidimensionnels
- Indexés à partir de `0`
- Affectation d'un élément : `tableau[indice]=valeur`
- Affection de plusieurs éléments : `tableau=([i1]=v1 [i2]=V2 ... )`
- Valeur d'un élément : `${tableau[indice]}`
- Valeur des éléments : `${tableau[*]}` ou `${tableau[@]}`
- Longueur : `${#tableau[indice]}` donne la taille de l'élément `indice`
- `${#tableau[*]}` donne la taille du tableau

Exemples

```
bash$ tab[3]=aaa
bash$ echo ${tab[3]}
aaa
bash$ tab=([0]=a [1]=b [2]=c [3]=d [4]=e)
bash$ echo ${tab[2]}
c
bash$ echo ${tab[*]}
a b c d e
bash$ t[1]=bbb
bash$ t=([0]=a [2]=c [3]=d [4]=e)
bash$ echo ${t[*]}
a c d e
bash$ t[1]=bbb
bash$ echo ${#t[1]}
3
bash$ echo ${#t[*]}
5
```

Une autre forme de boucle

for ((⟨*init*⟩ ; ⟨*cond*⟩ ; ⟨*inc*⟩))
do
    ⟨*commandes*⟩
done

- Structure identique à celle des langages C ou Java
- Les expressions sont évaluées selon les règles d'évaluation des expressions arithmétiques

```
bash$ for (( i=0 ; $i < 3; i++)) ; do echo "hello" ; done
hello
hello
hello
```

Signaux

La commande interne

`trap [argument] {numero_signal}`

- permet à un processus de spécifier le traitement qu'il veut exécuter quand un signal particulier lui est adressé ;
- le *numero_signal* spécifie le (ou les) signal concerné par le traitement ;
- l'argument spécifie le traitement à adopter :
  - si *argument* vaut `""` le signal est ignoré,
  - si *argument* vaut `-`, le comportement par défaut est réinstallé,
  - si *argument* est une commande ou une fonction, celle-ci sera exécutée à la réception du signal.

Exemples

```
bash$ essai > toto &
[1] 11697
bash$ ps
  PID TTY          TIME CMD
 5307 ttyp2    00:00:00 bash
11697 ttyp2    00:00:01 essai
11698 ttyp2    00:00:00 ps
bash$ kill -15 11697
bash$ cat toto
coucou
coucou
coucou
coucou
je suis tue
[1]+  Done                    essai >toto
```

```
#!/bin/bash
trap 'echo "je suis tue"; exit'
while test "" = ""
do
   echo "coucou"
done
```

find

`find` ⟨*répertoires*⟩... ⟨*critères*⟩...

- Recherche de fichiers dans une arborescence selon certains critères :
  - parcours **récursif** de tous les répertoires spécifiés
  - application de critères sur tous les fichiers rencontrés
    - critères appliqués de gauche à droite
    - si un critère ne peut pas être appliqué on passe au fichier suivant
- Exemple de critères

`-name GLOB`	le nom du fichier doit correspondre au modèle `GLOB`
`-regex REGEX`	le nom du fichier doit correspondre à l'expression régulière `REGEX`
`-type TYPE`	le fichier doit être de type `TYPE`
`-print`	affiche le nom du fichier
`-delete`	supprime le fichier
`-exec CMD \;`	exécute `CMD` sur le fichier qui doit être présent dans `CMD` sous la forme `{}`

sed

```
sed [-n] [[-e] REQUETES] [-f SCRIPT] [FICHIER...]
```

- `sed` est un *stream editor* : permet d'éditer un flot de données

  une seule ligne est mémorisé à la fois on peut donc traiter de très gros fichiers avec très peu de ressources.

- lit des lignes de caractères et applique une requête sur chacune

- Options de base
  - `-n` n'afficher que les lignes dont l'affichage est explicitement demandé
  - `-f SCRIPT` lire les requêtes dans le fichier `SCRIPT`
  - `-e REQUETES` traiter les commandes de la chaine `REQUETES`

Commandes `sed`

Forme

```
[ligne1[,ligne2][!]] CMD [ARGS]
```

Pour spécifier quelles seront les lignes à traiter :

- si aucune ligne n'est spécifiée, tout le fichier est traité ;
- si une ligne est spécifiée, elle est traitée ;
- si deux lignes sont spécifiées, le bloc délimité par ces deux lignes sera traité ;
- si le `!` est présent, l'ensemble complémentaire des lignes spécifiées sera traité.

Commandes `sed`

Les lignes sont spécifiées par :

`nombre`	le numéro de la ligne
`$`	la dernière ligne du fichier
`/REGEX/`	la première ligne qui vérifie l'expression régulière
`+nombre`	adressage relatif

Exemples :

`13`	la ligne 13
`/^mot/, +3`	la première ligne qui commence par « *mot* » et les trois lignes suivantes
`1,/mot/+2`	de la ligne 1 jusqu'à deux lignes après la première occurrence de « *mot* »

Commandes de base

- `p` (print) écrit le contenu du tampon sur la sortie standard.
- `n` (next) idem puis remplace le contenu du tampon par la prochaine ligne d'entrée.
- `=` écrit le numéro de la ligne courante sur la sortie standard.

Exemples.
`sed 'p' toto` affiche le contenu de toto en doublant les lignes.
`sed -n 'p' toto` ou `sed '' toto` affichent tel quel le contenu du fichier.
`sed -n -e '=' -e 'p' toto` affiche le numéro de ligne avant chaque ligne.
(Identique à `sed -e '=' toto`).

Autres commandes

- `d` (delete) détruit le contenu du tampon.
- `q` (quit) termine l'exécution, `sed` lit l'entrée standard sans la traiter.
- `y/chaine1/chaine2/` les deux chaînes doivent être de même longueur, traduit chaque occurrence du $i^{ème}$ caractère de `chaine1` par le $i^{ème}$ caractère de `chaine2`.
- `s/REGEX/CHAINE/[mod]` recherche une chaîne correspondant à l'expession régulière `REGEX` et la remplace par la chaîne `CHAINE`.
  `mod` peut être :
  - `p` (print) pour afficher le tampon s'il a subit une modification
  - `g` (global) pour effectuer la substitution sur tout la ligne

  Dans `CHAINE` on peut rappeler la valeur correspondant à l'expression régulière avec `&`.
- `i\TEXT` insère `TEXT` avant la ligne
- `a\TEXT` ajoute `TEXT` après la ligne

Toutes les commandes

```
man sed
```

```
info sed
```

Exemples

```
bash$ cat /tmp/toto
aaa
bbb
ccc

bash$ sed -e 'd' /tmp/toto

bash$ sed -e '=' -e 'd' /tmp/toto
1
2
3

bash$ sed -e 'y/abc/xyz/' /tmp/toto
xxx
yyy
zzz
```

Exemples

```
bash$ cat /tmp/titi
tralala il fait beau tralala
tralala le soleil brille tralala

bash$ sed -e 's/tralala/youpi/' /tmp/titi
youpi il fait beau tralala
youpi le soleil brille tralala

bash$ sed -e 's/tralala/youpi/g' /tmp/titi
youpi il fait beau youpi
youpi le soleil brille youpi
```

Expressions–Régulières – Quoi ? Pourquoi ?

- Les expressions régulières définissent des « *motifs* » qui permettent de rechercher des chaînes dans un texte.
- Elles sont utilisées par de nombreuses commandes comme `sed`, `grep`, `find`, ...
- Deux types d'expressions régulières sont définies en POSIX, les « *obsolètes* » qui sont les anciennes expressions régulières et les expressions dites « *étendues* ».
- On dit qu'une chaîne « *correspond* » ou « *matche* » ou « *est appariée à* » une expression régulière.
- Attention aux problèmes d'incompatibilités et de portage !

Exemple : afficher les lignes du fichier `/etc/services` qui commencent par un 't'.

```
egrep ^t /etc/services
```

Expression régulière – Définition

- Au plus haut niveau, une expression régulière (étendue) est une alternative (symbole `|`).
- Chaque opérande de l'alternative est une concaténation de « *pièces* ».

Exemples.
Les lignes commençant par 'f' ou 'g' :

```
egrep "^f|^g" /etc/services
```

Les lignes commençant par 'fo' ou 'gd' :

```
egrep "^fo|^gd" /etc/services
```

Pièce

- Une « *pièce* » est un atome (noté a ici) éventuellement suivi d'un symbole spécial :
  - `a*` correspond à la répétition de 0 ou plusieurs fois a ;
  - `a+` correspond à la répétition de 1 ou plusieurs fois a ;
  - `a?` correspond à la répétition de 0 ou 1 fois a ;
  - `a{n}` correspond à une séquence de n matches de a ;
  - `a{n,}` correspond à une séquence d'au moins n matches de a ;
  - `a{n,m}` avec $n \leq m$ correspond à une séquence k matches de a avec $n \leq k \leq m$;

Exemple

```
bash$ cat toto
aaa
abab
aaaaa
bbbbbb

bash$ egrep "(a|b){4}" toto
abab
aaaaa
bbbbbb

bash$ cat toto
acaaacc
aa
ababaaaaa
aaaaaaaaaa

bash$ egrep "a{3,5}" toto
acaaacc
ababaaaaa
aaaaaaaaaa
```

Atome

- Une expression régulière entre parenthèses (l'expression `()` correspond à la chaîne vide).
- Un caractère spécial :
  - `.` qui représente n'importe quel caractère (sauf entre `[` et `]`) ;
  - `^` qui représente un début de ligne lorsqu'il est le premier caractère d'une expression ;
  - `$` qui représente une fin de ligne lorsqu'il est le dernier caractère d'une expression ;
  - `\` suivi d'un caractère quelconque représente ce caractère ;
  - `\<` et `\>` correspondent respectivement au début et à la fin d'un mot. Un mot est une suite de caractères alpha-numériques et « *souligné* ».
- Un seul caractère sans signification spéciale.
- Une expression entre crochets.

Exemple

```
bash$ cat toto
.{aaaaaa}
.{aaaaaa}b
.{aabaaaa}
c.{aaaaaa}

bash$ egrep "()" toto
.{aaaaaa}
.{aaaaaa}b
.{aabaaaa}
c.{aaaaaa}
c{aaaaa}

bash$ egrep ^.{a*}$ toto
.{aaaaaa}
c{aaaaa}
```

Crochets

Une expression entre crochets correspond à un caractère de l'ensemble décrit. Elle peut être :

- `[`$c_1 c_2 \ldots c_n$`]` correspond à un des caractères c_i ;
- `[^`$c_1 c_2 \ldots c_n$`]` correspond à un caractère du complémentaire de l'ensemble $\{c_1, \ldots, c_n\}$;
- c_1`-`c_2 dans une suite de caractères décrivent tous les caractères compris entre c_1 et c_2 (inclus) ;
- Cas particuliers :
  - `]` dans une suite doit être placé en premier caractère,
  - `-` dans une suite doit être placé en premier ou en dernier caractère.

Exemple

```
bash$ cat toto
bonjour il fait beau !
tralala la lere
aaa[bbb]ccc

bash$ egrep b[aeiouy] toto
bonjour il fait beau !

bash$ egrep l[^ea] toto
bonjour il fait beau !

bash$ egrep [b-d] toto
bonjour il fait beau !
aaa[bbb]ccc

bash$ egrep []\!] toto
bonjour il fait beau !
aaa[bbb]ccc
```

Exemple

```
bash$ cat toto
Bonjour, il fait beau
C'est un grand jour !
operation : 2+2=4 !

bash$ egrep "\<jour\>" toto
C'est un grand jour !

bash$ egrep "[-+=]" toto
operation : 2+2=4 !

bash$ egrep "jour\>" toto
Bonjour, il fait beau
C'est un grand jour !
```

Classes de caractères

- `[:class:]` correspond (entre crochets) à un caractère de la classe de caractères ainsi désignée, `class` pouvant être :
  - `alnum` pour les caractères alphanumériques ;
  - `digit` pour les chiffres décimaux ;
  - `punct` pour les caractères de ponctuation ;
  - `alpha` pour les lettres ;
  - `graph` pour les caractères imprimables sauf espace ;
  - `space` pour les caractères d'espacement ;
  - `blank` pour espace ou tabulation ;
  - `lower` pour les lettres minuscules ;
  - `upper` pour les lettres majuscules ;
  - `cntrl` pour les caractères de contrôle ;
  - `print` pour les caractères imprimables ;
  - `xdigit` pour les chiffres hexadécimaux.

Exemple

```
bash$ cat toto
aaa 0xAF12 bbb
ABDFEZFFDFD
aCsDeFgBtHfD
afbv 12 fdlk 14

bash$ egrep 0x[[:xdigit:]]* toto
aaa 0xAF12 bbb

bash$ egrep "^[[:upper:]]*$" toto
ABDFEZFFDFD

bash$ egrep "^([[:lower:]][[:upper:]])*$" toto
aCsDeFgBtHfD
```

Sous-expressions

```
bash$ cat toto
aaa toto azerrt
aaa djfhldksfh aaa fkdlgjmfdl
sldkjfksfjk
toto titi titi toto
toto toto titi titi

bash$ egrep "(..)\1" toto
aaa toto azerrt
toto titi titi toto
toto toto titi titi

bash$ egrep "([[:alpha:]]{3,}).*\1" toto
aaa djfhldksfh aaa fkdlgjmfdl
toto titi titi toto
toto toto titi titi

bash$ egrep "([[:alpha:]]{4})[[:space:]]*\1" toto
toto titi titi toto
toto toto titi titi
```

Sous-expressions

```
bash$ cat toto
toto titi titi toto
toto toto titi titi
totititotitotitotitotito
totitototototo

bash$ egrep "(\<[[:alpha:]]{4}\>).*(\<[[:alpha:]]{4}\>).*\2.*\1" toto
toto titi titi toto

bash$ egrep "(..)(..)(\2\1){5}" toto
totititotitotitotitotito
```

Perl

Qu'est-ce que Perl ?

- Langage de programmation **très souple**
- Combine des fonctionnalités de `sh`, `sed` et `awk`
- Syntaxe et idées de « *Perl* » empruntées à `sh`, `sed`, `awk`, `C`, `C++`, `BASIC/PLUS`.
- Créé par Larry Wall.
- Initialement crée pour traiter des fichiers `log` pour extraire des données, produire des rapports puis enrichissement et diffusion sur Internet.
- Grande facilité pour traiter les fichiers `log`
  $\Rightarrow$ succès auprès des administrateurs système dès 1989.
- Aujourd'hui employé dans de nombreux domaines par tous ceux qui doivent manipuler ou analyser rapidement de nombreuses données, que ce soit des pages Web ou des séquences d'ADN.

Pourquoi l'apprendre ?

- « *Perl* » est un outil polyvalent (un seul outil au lieu de plusieurs avec des syntaxes différentes).
- « *Perl* » est portable (Unix, Windows, Mac, ...).
- Par rapport au shell :
  - langage « *semi-compilé* », plus rapide que les scripts shell ;
  - dispose d'un débogueur interactif ;
  - évite l'utilisation coûteuse des pipelines et commandes Unix
- Très bonne interface avec les systèmes facilitant l'écriture de démons, surtout pour les applications réseaux.
- En vrac : paquetages, modularité, peut traiter des fichiers textes ou binaires, pas de limitations arbitraires sur la taille des données, ... et ...langage objet !

Pourquoi faire ?

- Traitement de fichiers (optimisé pour fichiers texte mais peut manipuler fichiers binaires).
- Formatage de données.
- Écriture de démons.
- Maquettes de projets avant d'utiliser un langage compilé.
- Écriture de scripts (tout ce qui peut être fait avec un script shell).
- cgi
- ...

La richesse de « *Perl* » fait qu'il est toujours possible de traiter un problème de plusieurs façons d'où le slogan :

There's more than one way to do it.

Comment exécuter un programme *Perl* ?

- Écrire un script dont la première ligne est

  `#!/usr/bin/perl`,

  le rendre exécutable et . . . l'exécuter.
- Lancer *Perl* avec comme argument le script à exécuter et ses arguments :

  `perl monScript arg1 arg2 ...`
- Tester un ensemble d'instructions sans écrire de script en utilisant l'option `-e` :
  - `perl -e 'instruction1; ... ; instructionN;'`

Programme *Perl*

Perl manipule des objets grâce à des instructions.

- Les instructions peuvent prendre les formes suivantes :
  - `expression ;`
  - `expression modifieur;`
  - `[label] bloc`
  - `structure_de_controle`
- Un bloc est une suite d'instructions entre accolades.
- Un programme est un ensemble d'instructions.
- Les commentaires commencent par un caractère **#** et se terminent à la fin de la ligne.

Exemple.

```
bash$ perl -e 'print "coucou\n";'
coucou
```

Les objets manipulés par *Perl*

Les objets peuvent être manipulés de plusieurs manières, par :

- les constantes ;
- les variables ;
- les références.

Perl ne dispose que de quelques types de données prédéfinis :

- les **scalaires** (une seule valeur simple) ;
- les **tableaux** (liste ordonnée de scalaires) ;
- les **hachages** aussi appelés tableaux associatifs (ensemble non ordonné de paires clef/valeur, la chaîne clef donne accès à la valeur scalaire associée).

N.B. Voir aussi les handles de fichiers, sous-programmes, typeglobs et formats (qui peuvent être considérés commes des types de données).

Les variables

- Les variables sont toujours de l'un des trois types de base.
- Elles ne sont pas déclarées, elles sont créées dynamiquement à la demande.
- Le nom d'une variable commence par une lettre ou un caractère _ suivi de toute combinaison de lettres, chiffres et caractère _. La taille du nom est limitée à 255 caractères. Attention, « *Perl* » distingue les majuscules et les minuscules.
- Un préfixe est utilisé devant le nom pour indiquer le type de la variable :
  - `$nom` est un scalaire ;
  - `@nom` est un tableau ;
  - `%nom` est un hachage.

Les scalaires

Un scalaire contient toujours une seule valeur qui peut être un **nombre**, une **chaîne** ou une **référence**.

- Littéraux numériques :

123	entier
123.456	décimal
6.08E-2	notation scientifique
0xFFFF	hexadécimal
056	octal
4_456_345_544	entier avec séparateur

Les conversions nécessaires sont automatiques :

```
bash$ perl -e 'print 123e2 - 12300;'
0
```

Les scalaires (2)

- Littéraux chaînes de caractères :
  - délimités par des apostrophes (quotes), dans ce cas la chaîne est complètement protégée (excepté \\ et \' : qui permettent d'inclure un anti-slash ou une apostrophe dans la chaîne).
  - délimités par des guillemets, les anti-slashs et les variables (commençant par $ ou @) sont alors interprétés. Attention le caractère ‘ n'est PAS interprété.

  Il y a conversion automatique entre chaînes et nombres :

```
bash$ perl -e 'print 123 - "123", "\t", length(0xFF), "\n";'
0       3
bash$
```

- Valeurs booléennes :
  - `0`, `"0"`, `""` et `undef` valent **faux** ;
  - le reste vaut vrai.

Exemple

```
bash$  cat essai
#! /usr/bin/perl
$nb = 12;
$chaine='bonjour\'toto $nb \n';
printf $chaine;
$chaine="bonjour\'toto $nb \n";
printf $chaine;
$chaine="toto\@machin.fr \x41 D \ua \n";
printf $chaine;

bash$  essai
bonjour'toto $nb \nbonjour'toto 12
toto@machin.fr A D A
```

Les listes

Une liste est une collection de constantes scalaires de forme :

```
(element1, element2, ..., elementN)
```

- la taille de la liste peut varier de manière dynamique ;
- les différents éléments de la liste ne sont pas tous forcément de même nature ;
- la valeur d'une liste est son nombre d'éléments.

Exemple.

`(1,2,3)`	liste de trois éléments numériques
`('cd', '', 'ab')`	liste de trois chaînes
`()`	liste vide
`(1, ' ', "\n", 1e10)`	liste « *mixte* »

Contexte

Chaque opération *Perl* se déroule dans un contexte particulier. Le résultat d'une opération dépend du contexte dans lequel elle est évaluée. Il existe deux contextes principaux : scalaire ou liste.

```
bash$  cat essai
#! /usr/bin/perl
@liste=('a','b','c');
$l=('a','b','c');
print @liste, "\n", $l, "\n", scalar @liste, "\n";

bash$  essai
abc
c
3

bash$
```

Rq : les backquotes

De même qu'en shell, une commande Unix placée entre backquotes (caractère ‘) est exécutée et son résultat est renvoyé.

- L'interprétation de la chaîne entre backquotes est identique à celle d'une chaîne entre guillemets.
- Dans un contexte scalaire, le résultat est la sortie standard de la commande.
- Dans un contexte de liste, le résultat de la commande est découpé en lignes, chaque ligne formant un élément de la liste (selon la variable `$/`).

Variables : généralités

- Les variables sont crées dynamiquement à la demande. Un tableau grossit par ajout d'éléments.
- Une variable non définie s'évalue comme une chaîne vide (elle vaut donc faux).
- Les variables scalaires ne sont pas typées. En fonction du contexte, le contenu sera interprété comme une chaîne, un nombre ou un booléen.
- Le préfixe $, @, ou % est toujours nécessaire pour accéder à la variable.
- Il est possible d'avoir des variables de types de base différents de même nom.
- Il n'est pas nécessaire de mettre les variables chaînes entre guillemets pour protéger leur valeur (contrairement au shell).

Variables scalaires

- Modification et création par affectation : signe =.
  Exemples.
  `$n = 12;` nombre
  `$gn=5_294_967_295 ;` très grand nombre !
  `$nusers=‘who | wc -l‘;` exécution d’une commande
- Pour utiliser une variable :
  - `$nom;`
  - `${nom}` pour lever les ambiguités.

  Exemples.
  `${n}34` donne 1234
  `$n34` variable de nom n34

Variables tableaux

- Modification et création par affectation : signe =.
  Exemples.
  `@tab = ('el1', 'el2', 'el3');` affectation d'une liste
  `@tab = ();` liste vide
  `@tab = `ls`;` les éléments sont les noms de fichiers renvoyés par `ls`.
- Dans un contexte de liste, `@tab` vaut la liste de tous les éléments composant le tableau.
- Dans un contexte scalaire, `@tab` vaut le nombre d'éléments du tableau.
- Entre guillemets, `@tab` vaut une chaîne de caractères formée de ses éléments séparés par des espaces.

Exemples

```
bash$  cat essai
#! /usr/bin/perl

@tab = ('a','b', 1, 'c','d');
print @tab, "\n";
print "@tab", "\n";
print @tab + 0, "\n";

bash$  essai
ab1cd
a b 1 c d
5

bash$
```

Accès aux éléments d'un tableau

Syntaxe :

```
$tableau[expression]
```

Attention, il s'agit du symbole `$` !!!

- Le premier élément est à l'indice `0`.
- Le dernier élément est à l'indice `$#tableau`.
- Les indices négatifs permettent de numéroter les éléments à partir de la fin.
- Lorsque l'on crée l'élément d'indice `n`, tous les éléments manquant d'indice compris entre `0` et `n-1` sont créés avec la valeur `""` ou `0`.

Exemples

```
bash$  cat essai
#! /usr/bin/perl
@jours = ('lu', 'ma', 'me', 'je', 've', 'sa', 'di');
print $#jours, "\n";
print $jours[5], " ", $jours[-2], "\n";
$jours[4] = "ven";
$jours[10] = "ind";
print $#jours, "\t@jours\n";
$indice=4;
print "$jours[$indice+1]\n";

bash$  essai
6
sa sa
10      lu ma me je ven sa di    ind
sa

bash$
```

Accès à un sous-ensemble du tableau

Syntaxe `@tableau[liste-indices]`.

- La liste peut être le résultat de l'évaluation d'une expression ou une liste constante (parenthèses facultatives).
- Le même indice peut apparaître plusieurs fois dans la liste d'indices.

Exemples.

```
#! /usr/bin/perl
@jours = ('lu', 'ma', 'me', 'je', 've', 'sa', 'di');
@conges = (2,5,6);
print "Jours de conge : ", "@jours[@conges]", "\n";
@absences = (1,3,1,1);
print "Absences : ", "@jours[@absences]", "\n";

bash$  essai
Jours de conge : me sa di
Absences : ma je ma ma
```

Manipulation de listes

- Chaque élément d'une liste est un scalaire.
- Lors de la construction d'une liste, chaque élément est une expression à évaluer :
  - si le résultat est un scalaire, il est ajouté à la liste ;
  - si le résultat est une liste, tous les éléments de la liste sont insérés dans la liste en construction.
- Une liste de variables peut être utilisée en partie gauche d'une affectation.

Exemples.

`($x,$y) = ($y, $x);`	échange les valeurs de x et y
`($x,$y) = (1,2,3,4);`	3 et 4 sont ignorés
`($x,$y) = (1);`	y vaut 0 ou ""
`($x,@reste) = (1,2,3,4);`	x vaut 1, reste (2,3,4)
`(@tout,$x) = (1,2,3,4);`	x vaut 0 ou ""

Tableaux et listes

- Dans toute expression qui requiert une liste, on peut utiliser un tableau.
- On peut considérer une liste comme un tableau temporaire.
- Attention :
  - si on affecte une liste à un scalaire, c'est le dernier élément de la liste qui est affecté ;
  - si on affecte un tableau à un scalaire, c'est le nombre d'éléments du tableau qui est affecté.

Exemples.

```
$dernier=('a','b','c');
print $dernier, "\n";                                    affiche c
@tab=('a','b','c');
$scal = @tab;
print $scal, "\n";                                       affiche 3
```

Exemples

```
bash$  cat essai
#! /usr/bin/perl

$elem = (1,2,3,4)[2];
@lignes3et5 = (‘cat toto.txt‘) [3,5];

print $elem,"\n";
print @lignes3et5;

bash$  essai
3
ddddddddddddddddddd
fffffffffffff

bash$
```

Variables prédéfinies

Il existe de nombreuses variables prédéfinies parmi lesquelles :

- `$0` nom du script ;
- `$_` argument par défaut ;
- `$$` numéro du processus ;
- `$<` uid réel du processus ;
- `$(` gid réel du processus ;
- `$?` code de retour du dernier " ou pipe ou appel système ;
- `$"` séparateur des éléments d'un tableau dans une chaîne entre guillemets ;

Variables prédéfinies (2)

- `$,` séparateur des arguments écrits par `print` (rien par défaut) ;
- `$.` numéro de la dernière ligne lue ;
- `@ARGV` arguments passés au script. Attention, `$ARGV[0]` est le premier argument et pas le nom du script (comme en C) ;
- `@_` arguments dans une fonction ou un sous-programme ;
- `%ENV` environnement ;
- ...

Consulter le manuel pour plus de détails.

Les opérateurs

- Certains opérateurs s'appliquent à des termes scalaires, dans ce cas les opérandes sont évaluées dans un contexte scalaire.
- Certains opérateurs s'appliquent à des termes listes, dans ce cas les opérandes sont évaluées dans un contexte de liste.
- Certains opérateurs sont mixtes, dans ce cas leur action dépend du contexte souhaité.

Quelques exemples d'opérateurs :

- incrémentation et décrémentation :
  - **++** et **−** ;
- traitement de bits :
  - **~** complément à un,
  - **«** et **»** décalages de bits gauche et droite,
  - **&** « ***et*** » bit à bit,
  - **|** et **^** « ***ou*** » et « ***ou exclusif*** » bit à bit ;

Opérateurs (2)

- comparaisons :
  - `<`, `>`, `<=`, `>=`, comparaisons de nombres,
  - `lt`, `gt`, `le`, `ge`, comparaisons de chaînes,
  - `==`, `!=`, `<=>` égalité de nombres,
  - `eq`, `ne`, `cmp`, égalité de chaînes ;
- opérateurs logiques :
  - `!`, `&&` et ‖ négation, **et** et **ou** logiques,
  - `not`, `and`, `or` et `xor` négation, **et**, **ou** et **ou exclusif** logiques de priorité basse ;
- opérations arithmétiques :
  - `**` élévation à la puissance.
  - `+`, `-`, `*`, `/`, `%`, addition, soustraction, multiplication, division et modulo.
- opérateurs sur les chaînes :
  - `.` concaténation.

Opérateur de répétition

Syntaxe `expression-gauche x expression-droite`

- L'expression droite est évaluée en contexte scalaire.
- En contexte scalaire, l'expression de gauche est évaluée et son résultat est répété selon l'expression de droite (évaluée en contexte scalaire).
- En contexte de liste, si l'expression de gauche est une liste entre parenthèses, elle est répétée selon l'expression de droite (évaluée en contexte scalaire).

```
print '-' x 10;                                  affiche 10 tirets
print 'ab' x 3;                                  affiche ababab
@tab = 1 x 3;
print "@tab";                                     affiche 111
@tab = (1) x 3;
print "@tab";                                     affiche 1 1 1
@tab = (@tab) x 2;
print "@tab", "\n";                               1 1 1 1 1 1
```

Les modifieurs

Un modifieur est une forme que l'on peut ajouter à la fin d'une expression :

- le modifieur peut changer :
  - le fait que l'instruction va être exécutée ou non,
  - le nombre de fois où l'instruction va être exécutée ;
- un seul modifieur peut être utilisé, juste avant le ' ; ' ;
- les modifieurs possibles sont :
  - `if expression;`
  - `unless expression;`
  - `while expression;`
  - `until expression;`

Exemple

```
bash$  cat essai
#! /usr/bin/perl

$x = 2;
$y = 5;

print "x egal y \n" if $x == $y;
print "x different de y \n" if $x != $y;

print "x egal y \n" unless $x != $y;
print "x different de y \n" unless $x == $y;

bash$  essai
x different de y
x different de y
```

Instruction *if*

```
if (expression)
   BLOC
elsif (expression)
   BLOC
   ...
else
   BLOC
```

Les parties **elsif** et **else** sont facultatives. Comme les instructions sont forcément sous forme de BLOC (donc entre accolades), il n'y a pas d'ambiguité possible.
Exemple.

```
if ($x == $y) { print "x egal y \n";}
else { print "x different de y \n" ;}
```

Rq. On peut utiliser **unless** à la place de `if` dans une forme `if ...` ou `if ... else...`

Instruction *while*

```
LABEL: while (expression)
          BLOC
```

ou

```
LABEL: while (expression)
          BLOC1
       continue
          BLOC2
```

- Le label est optionnel, c'est une chaîne de caractères qui est, par convention en majuscules.
- Le `BLOC2` sera exécuté chaque fois que l'on termine le `BLOC1`, avant de recommencer l'évaluation de l'expression.
- **Rq.** On peut utiliser `until` à la place de `while`.

Exemple

```
bash$  cat essai
#! /usr/bin/perl
$i = 1;
until ($i == 3) {
    print "test until : bloc1 tour $i\n";
    $i = $i+1;
}
while ($i != 5) { print "test while : bloc1 tour $i\n";}
continue {print "test while : bloc2 tour $i\n"; $i = $i+1;}

bash$  essai
test until : bloc1 tour 1
test until : bloc1 tour 2
test while : bloc1 tour 3
test while : bloc2 tour 3
test while : bloc1 tour 4
test while : bloc2 tour 4
```

Instruction *for*

```
LABEL: for (expression; expression; expression)
           BLOC
```

- Les trois expressions correspondent respectivement à l'initialisation, la condition et la réinitialisation. Chacune est optionnelle.
- Les initialisation et réinitialisation peuvent concerner plusieurs variables (séparer les affectations ou autres par des virgules).
- Si la condition n'est pas présente, elle est considérée comme vraie.

Instruction *foreach*

```
LABEL: foreach var (liste)
          BLOC
```

- Parcourt la liste et assigne tour à tour chaque élément de la liste à la variable `var`.
- La variable est implicitement locale à la boucle, elle reprend sa valeur précédente à la sortie de la boucle.
- Si `liste` est un « *vrai* » tableau plutôt qu'une liste, chaque élément du tableau est modifiable par l'intermédiaire de la variable de boucle.
- Le bloc `continue` peut également être présent.

Exemples

```
bash$  cat essai
#! /usr/bin/perl
for ($i=0; $i<4; $i++)
   {@vect[$i] = $i+10;}
print "@vect\n";
foreach $nom (`ls`)
   {print $nom;}
@tab= (1,2,3,4);
foreach $elem (@tab)
   {$elem=$elem *2;}
print "@tab\n";

bash$  essai
10 11 12 13
fichier.txt
essai
2 4 6 8
```

Contrôle de boucles

Il existe des instructions permettant de rompre le flot « *normal* » des boucles :

- `last LABEL`
- `next LABEL`
- `redo LABEL`

- Le `LABEL` est falcutatif, dans ce cas, on se réfère à la boucle la plus interne.
- La commande **`last`** sort immédiatement de la boucle. Le bloc `continue`, s'il existe, n'est pas exécuté.
- La commande **`next`** permet de « *sauter* » à l'itération suivante. Le bloc `continue`, s'il existe, est exécuté avant que la condition ne soit réévaluée.
- La commande **`redo`** permet de redémarrer le bloc de boucle sans réévaluer la condition. Le bloc `continue`, s'il existe, n'est pas exécuté.

Exemples

Considérons le programme suivant en « *programmation classique* ».

```
bash$  cat essai
#! /usr/bin/perl
@tab1 = (2,0,5,8,5);
@tab2 = (3,0,2,10,5,12);
for ($i=0; $i < @tab1; $i++)
{
   $j=0;
   while ($j < @tab2 && $tab1[$i] <=  $tab2[$j]) {
       $tab1[$i] += $tab2[$j];
       $j++;
   }
}
print "@tab1\n";

bash$  essai
5 3 5 8 5
```

Exemples (2)

Le programme suivant est identique, dans un style « *plus Perl* ».

```
#! /usr/bin/perl
@tab1 = (2,0,5,8,5);
@tab2 = (3,0,2,10,5,12);

A: foreach $ceci (@tab1)
{
   B: foreach $cela (@tab2)
   {
       next A if $ceci > $cela;
       $ceci+=$cela;
   }
}
print "@tab1\n";
```

Les hachages

- Un hachage est un tableau associatif, c'est-à-dire un tableau dont les valeurs ne sont pas sélectionnées par des indices entiers mais par des indices quelconques.
- Les « indices » sont nommés « clés » et sont choisis par le programmeur.
- L'ordre interne des données n'est pas connu du programmeur, il permet de ne pas parcourir toute la liste pour accéder à une valeur. Y toucher risque de limiter l'efficacité d'une telle structure.

Accès aux éléments

- Une variable hachage est toujours préfixée par le symbole %.
- Chaque élément d'un hachage est un scalaire (comme pour les tableaux).
- On accède à un élément par une clé qui est également un scalaire (comme dans le cas des tableaux, l'élément étant un scalaire, l'accès se fait avec un symbole $).
- Comme pour les tableaux, on crée de nouveaux éléments en leur affectant une valeur.

Exemples.

```
$mon_hachage{"aaa"} = 3;                          clé aaa, valeur 3
$mon_hachage{3.14} = "Pi";                        clé 3.14, valeur Pi
$i = "toto";
$mon_hachage{$i} = 4;                             clé toto, valeur 4
print $mon_hachage{"toto"}, "\n";                 affiche 4
```

Variables et littéraux

Exemples.

```
print %mon_hachage, "\n";
```
affiche `aaa3toto43.14Pi`

```
@mon_tab = %mon_hachage;
print "@mon_tab", "\n";
```
affiche `aaa 3 toto 4 3.14 Pi`

- Il n'existe pas de « véritable » représentation littérale d'un hachage.
- Un hachage est représenté par une liste qui se lit de gauche à droite comme une suite de paires clé/valeur.
- On peut obtenir une copie d'un hachage par affectation.
- On peut inverser les rôles clé/valeur en utilisant l'opérateur reverse.

Exemples

```
bash$ cat essai
#!/usr/bin/perl
$mon_hachage{"aaa"} = 3;
$mon_hachage{3.14} = "Pi";
$mon_hachage{"toto"} = 4;

%copie = %mon_hachage;
print %copie, "\n";
%copie = reverse %copie;
print %copie, "\n";
print $copie{"Pi"},"\n";

bash$ essai
aaa33.14Pitoto4
Pi3.143aaa4toto
3.14

bash$
```

Fonction *keys*

- Fournit la liste de toutes les clés utilisées pour un hachage.
- Dans un contexte scalaire, fournit le nombre d'éléments du hachage (comme le hachage d'ailleurs).

```
bash$ cat essai
$hach{"aaa"} = 3;
$hach{3.14} = "Pi";
$hach{"toto"} = 4;
if (%hach){
    foreach $cle (keys(%hach)){
        print "cle : $cle, valeur : $hach{$cle}\n";
    }
}
else {print "Aucun element\n";}

bash$  essai
cle : aaa, valeur : 3
cle : toto, valeur : 4
cle : 3.14, valeur : Pi
```

Autres fonctions sur les hachages

- La fonction `values(%hach)` renvoie la liste de toutes les valeurs contenues dans le hachage (dans l'ordre correspondant à celui des clés renvoyées par `keys`).
- La fonction `delete($hach{$cle})` permet de supprimer un élément du hachage.
- La fonction `each(%hach)` permet d'accéder successivement (à chaque appel) à tous les couples clé/valeur du hachage sous forme de liste. Elle renvoie la liste vide lorsqu'il n'existe plus de couple.
- Il est possible d'accéder à une *tranche* de hachage : `@hach{$cle1, $cle2, $cle3}` représente la tranche du hachage `hach` correspondant aux clés citées.

Exemples

```
bash$ cat essai
#!/usr/bin/perl
%hach = ("cle1", 1, "cle2", 2, "cle3", 3, "cle4", 4);
delete $hach{"cle2"};
while (($cle,$valeur)=each (%hach)) {
     print "cle : $cle, valeur : $valeur\n";
}
@nouveau{"cle5","cle6","cle7"} = (5,6,7);
print $nouveau{"cle6"},"\n";
@nouveau{keys %hach} = values %hach;
print "@nouveau{keys %nouveau}","\n";

bash$  essai
cle : cle4, valeur : 4
cle : cle1, valeur : 1
cle : cle3, valeur : 3
6
4 5 6 7 1 3
```

Ouvrir un fichier

- Trois possibilités :
  - `open(MONHANDLE, "fichier.txt")` ouvre le fichier `fichier.txt` en **lecture** et lui associe le handle `MONHANDLE`.
  - `open(MONHANDLE, ">fichier.txt")` ouvre le fichier `fichier.txt` en **écriture** et lui associe le handle `MONHANDLE`.
  - `open(MONHANDLE, ">>fichier.txt")` ouvre le fichier `fichier.txt` en **ajout** et lui associe le handle `MONHANDLE`.
- Les trois retournent « vrai » en cas de succès et « faux » en cas d'échec.
- Pour fermer le fichier : `close(MONHANDLE)` (qui peut également échouer).

Lire un fichier

- Le fichier se lit ligne par ligne en utilisant le handle obtenu à l'ouverture entre les symboles <> (opérateur d'entrée).
- En contexte scalaire, à chaque évaluation, la ligne suivante du fichier est retournée, `undef` lorsqu'il n'y a plus de ligne disponible.
- En contexte de liste, retourne toutes les lignes restantes.
- Chaque fois qu'un test de boucle ne dépend que de l'opérateur d'entrée, la ligne lue est automatiquement affectée à la variable `$_`.
- Le handle de l'entrée standard est `STDIN`.

Exemples : équivalent de la commande `cat fichier`.

```
#!/usr/bin/perl
open(HAND, $ARGV[0]);
while (<HAND>) {print;}
```

Fonctions *chop et* chomp

Deux fonctions de manipulations des chaînes peuvent être utiles lors de la lecture d'un fichier :

- `chop($chaine)`
  - supprime le dernier caractère de la chaîne,
  - elle retourne le caractère supprimé,
  - si la chaîne est vide, la fonction ne retourne rien ;
- `chomp($chaine)`
  - si la chaîne se termine par un caractère `'\n'`, la fonction le supprime,
  - si la fonction a supprimé un caractère `'\n'`, elle retourne `1`, sinon, elle retourne `0`.

Écrire dans un fichier

Il suffit pour écrire dans un fichier de :

- de disposer d'un handle ouvert en écriture ou en ajout ;
- d'utiliser la fonction `print` avec, en premier argument, le handle du fichier.

Les handles de la sortie standard et de la sortie standard d'erreur dont `STDOUT` et `STDERR`.

Exemples : copie d'un fichier.

```
#!/usr/bin/perl
open(DE, "toto.txt");
open(VERS, ">toto2.txt");
while (<DE>) {
    print VERS;
}
close(DE);
close(VERS);
```

Fonction *die*

S'il n'est pas possible d'ouvrir un fichier (ou de le fermer), aucune erreur ne survient lorsque l'on tente de le lire ou d'y écrire. Il faut donc vérifier les retours des fonctions ou utiliser `die` :

- cette fonction prend une liste et l'envoie sur la sortie standard (comme `print`) ;
- si le message à afficher ne se termine pas par un `'\n'`, la fonction lui ajoute le nom et le numéro de ligne du fichier où s'est produite la sortie par `die` ;
- la fonction termine le processus (avec un code de retour non nul) ;
- il est également possible d'utiliser la variable `$!` qui contient le message d'erreur du dernier appel système.

Exemple

```
bash$ cat essai
#!/usr/bin/perl
$de = "t1.txt";
$vers = "t2.txt";
open(DE, $de) || die "Echec open $de : $!";
open(VERS, ">$vers") || die "Echec open $vers : $!";

while (<DE>) {
    print VERS;
}
close(DE) || die "Echec close $de : $!";
close(VERS)|| die "Echec close $vers : $!";

bash$  essai
Echec open t1.txt : No such file or directory at essai line 4.
```

Opérateur <>

- L'opérateur diamant **<>** fonctionne comme `<HANDLE>`.
- Il lit les données depuis les fichiers dont les noms figurent dans le tableau `@ARGV`.

Exemples.
Équivalent de la commande `cat f1... fN`.

```
#!/usr/bin/perl
while (<>) {
   print;
}
```

Afficher les lignes du fichier passé en argument en ordre inverse.

```
#!/usr/bin/perl
print reverse (<>);
```

Opérateurs de test sur les fichiers

Opérateur	Test
`-r`	en lecture par iud/gid effectifs
`-w`	en écriture par iud/gid effectifs
`-x`	exécutable par iud/gid effectifs
`-o`	appartenant à iud/gid effectifs
`-R`	en lecture par iud/gid réels
`-W`	en écriture par iud/gid réels
`-X`	exécutable par iud/gid réels
`-O`	appartenant à iud/gid réels
`-e`	existence
`-z`	existence et taille égale à 0
`-s`	existence et taille différente de 0

Opérateurs de test sur les fichiers (2)

Opérateur	Test
`-f`	fichier régulier
`-d`	répertoire
`-l`	lien symbolique
`-S`	socket
`-p`	tube nommé
`-b`	fichier bloc
`-c`	fichier caractère
`-t`	fichier associé à un terminal
`-u`	setuid bit positionné
`-g`	setgid bit positionné
`-k`	sticky bit positionné

Opérateurs de test sur les fichiers (3)

Opérateur	Test
`-T`	fichier « texte »
`-B`	fichier « binaire »
`-M`	Nombre de jours depuis dernière modif
`-A`	Nombre de jours depuis dernier accès
`-C`	Nombre de jours depuis dernière modif de l'inode

Exemple. Afficher les noms des fichiers lisibles dans le répertoire courant.

```
#!/usr/bin/perl
foreach (‘ls‘) {
    chomp;
    print "$_\n" if -r;
}
```

Expressions Rationnelles en Perl

- Les expressions rationnelles (telles que celles vue avec `sed`) peuvent être utilisées directement en *Perl*.
- Toutes les expressions rationnelles utilisées avec les différents outils UNIX tels de `sed`, `vi`, `emacs`, `grep`, ...peuvent également être décrites en *Perl*.
- L'utilisation des expressions rationnelles facilite la manipulation et la recherche dans les fichiers sans avoir besoin de faire appel à des commandes externes.
- Elles suivent un modèle étendu : « *Perl Compatible Regular Expressions* » Modèle `PCRE` utilisé dans beaucoup d'autres langages (PHP, Python, etc.)

Exemple. Afficher toutes les lignes d'un fichier qui contiennent le mot `toto`.

```
#!/usr/bin/perl
while (<>) {
    if (/toto/) { print; }
}
```

Caractères

Pour désigner un caractère dans une expression rationnelle, il est possible d'utiliser :

- **le caractère** ;
- `.` pour tout caractère excepté `'\n'` ;
- `[azfltof]` pour un caractère dans l'ensemble ;
- `[0-9]` pour un caractère de l'intervalle ;
- `^` pour le complémentaire ;
- `\^` pour un caractère (ici `^`) ayant une signification spéciale ;
- `\d` pour un chiffre et `\D` pour un non chiffre ;
- `\w` pour un caractère de mot et `\W` pour un caractère ne pouvant se trouver dans un identificateur ;
- `\s` pour un espacement `\S` pour un non espacement.

Caractères – Exemples

`[0-9]`	chiffre
`[0-9ab]`	chiffre ou 'a' ou 'b'
`[a-zA-Z_]`	lettre ou souligné
`[^a-z]`	tout sauf une minuscule
`[\s,?\.;:!]`	espacement ou ponctuation
`[\da-fA-F]`	chiffre hexadécimal
`[^\^]`	tout sauf un ^

Plusieurs caractères

Pour désigner une suite de caractères dans une expression rationnelle, il est possible d'utiliser :

- **la suite** elle-même ;
- `*` pour zéro ou plusieurs fois le caractère qui précède ;
- `+` pour une ou plusieurs fois le caractère qui précède ;
- `?` pour zéro ou une fois le caractère qui précède ;
- `6` pour six fois le caractère qui précède ;
- `3,8` pour 3 à 8 fois le caractère qui précède ;
- `5,` pour 5 fois ou plus le caractère qui précède ;
- les précédents motifs se remplacent de gauche à droite en englobant le plus de caractères possibles (« *gloutons* »), il est possible de les rendre « *paresseux* » en les faisant suivre d'un point d'interrogation.

Plusieurs caractères – Exemples

Expression	Mots
`ab?c`	`ac, abc`
`ab*c`	`ac, abc, abbc, ...`
`ab+c`	`abc, abbc, ...`
`a.b*`	`anbbb, a!bbbbb, ...`
`a.+.*`	`abcdefg` ($.+$)
`a.+?.*`	`a b cdefg` ($.+$, $.*$)

Autres

- Mémorisation d'une partie entre parenthèse et utilisation par `\n` si l'expression est la `n`$^{\text{ème}}$ entre parenthèses dans l'expression.
  Exemple : `(.)(.)\2\1`, les palindromes de 4 lettres.
- Alternatives séparées par des |. Exemple : `toto|titi|tutu`, l'un des mots au choix.
- Ancres `\b` pour une limite de mot, `^` pour un début de ligne (si bien situé), `$` pour une fin de ligne (si bien situé), ...
  Exemple : `^toto\b`, ligne commençant par le mot `toto` n'étant pas préfixe d'un autre mot.
- L'utilisation de `i` après l'expression permet d'ignorer majuscules et minuscules.
- Les variables sont évaluées dans les expressions rationnelles.

Substitution

L'opérateur de substitution ressemble à celui de `sed`

`s/expr/chaine/gi`

- remplace la première occurrence du motif décrit par l'expression par la chaîne donnée ;
- l'option `g` permet d'appliquer la substitution à toutes les occurrences ;
- l'option `i` permet d'ignorer les majuscules/minuscules ;
- il est possible d'utiliser des variables.

Exemple.
`s/toto/titi/gi` remplace toutes les occurrences de `toto` (en majuscules ou minuscules) par `titi`.

Variables

Après une comparaison réussie, des variables spéciales permettent de récupérer des parties de la chaîne (sinon, les variables sont indéfinies) :

- `$1`, `$2`, ...contiennent les valeurs de `\1`, `\2`, ...
- `$&` contient la partie de la chaîne qui a correspondu à l'expression rationnelle ;
- `$‘` contient la partie de la chaîne qui précède la partie qui a correspondu à l'expression rationnelle ;
- `$’` contient la partie de la chaîne qui suit la partie qui a correspondu à l'expression rationnelle.

Ces variables peuvent également être utilisées dans les substitutions.
Exemple.
`s/([0-9]+)/($1)/g` mettre les nombres entre parenthèses.

Opérateur =~

Il est possible d'appliquer les comparaisons et les substitutions à d'autres cibles que la variable $_.

Exemples.

```
bash$  cat essai1
#!/usr/bin/perl
$chaine= "Bonjour toto";
if ($chaine =~ /.*(..)\1/)
     { print "chaine valide\n";}
else { print "chaine non valide\n";}

bash$  cat essai2
#!/usr/bin/perl
while ($a = <STDIN>)
{
    $a =~ s/([0-9]+)/($1)/g;
    print $a;
}
```

Opérateur *split*

- Il permet de découper une chaîne, par exemple pour en extraire les champs.
- Il prend en arguments une expression rationnelle et une chaîne et retourne les parties de la chaîne qui ne correspondent pas au motif de l'expression rationnelle.

Exemple.

```
bash$ cat essai
#!/usr/bin/perl
$a = "bonjour, il fait   beau.";
@mots = split(/[, .]+/,$a);
print "@mots\n";

bash$ essai
bonjour il fait beau
```

Opérateur *join*

- Il permet de « coller » les éléments d'une liste en les séparant par une chaîne donnée.
- Attention, la chaîne n'est pas une expression rationnelle !

Exemple.

```
bash$ cat essai
#!/usr/bin/perl

@noms = ("dupont", "durant", "carpentier");
$liste = join(",", @noms);
print "$liste\n";

bash$  essai
dupont,durant,carpentier
```

Définition

Pour définir une « fonction utilisateur » :

```
sub nom_de_la_fonction {
instruction1 ;
...
instructionN ;
}
```

- Les définitions des fonctions sont globales (au paquetage).
- Le nom de la fonction est un nom quelconque, il n'y a pas de conflit avec les noms de variables.
- Le nom de la fonction est `&nom_de_la_fonction` ou simplement `nom_de_la_fonction`.
- Par défaut les variables utilisées dans une fonction sont globales.

Appel et Valeur de retour

Une fonction est appelée par son nom suivi de parenthèses

```
#! /usr/bin/perl
bonjour();
sub bonjour {
   printf "Bonjour\n";
}
```

Une fonction retourne toujours une valeur

- celle de l'instruction **`return`** ;
- sinon, celle de la dernière expression évaluée.

```
print deux_plus_deux(), "\t", trois_plus_trois(), "\n";}
sub deux_plus_deux {
   2 + 2 ;
}
sub trois_plus_trois {
   return 3 + 3;
}
```

Paramètres

- Une fonction peut être appelée avec des paramètres, il suffit de faire suivre l'invocation de la fonction de la liste des paramètres :

```
somme (2,4,6);
```

- La liste des paramètres effectifs est affectée à la variable `@_` durant l'exécution de la fonction :
  - `@_` est un tableau contenant les paramètres ;
  - les éléments du tableau sont les paramètres dans l'ordre, une tentative de lecture des éléments suivants donne `undef` ;
  - la variable `@_` est **locale** à la fonction.

Exemple

```
fonction1(1,2,3,4,5);
sub fonction1 {
    print "@_\n";
    fonction2(6,7);
    for ($i; $i < 6; $i++) {
        print ":$_[$i]:\t";
    }
    print "\n";
}
sub fonction2 {
    print "@_\n";
    for ($j; $j < 6; $j++) {
        print ":$_[$j]:\t";
    }
    print "\n";
}
```

Exécution :

```
bash$ essai.pl
1 2 3 4 5
6 7
:6:  :7:  ::  ::  ::  ::
:1:  :2:  :3:  :4:  :5:  ::
```

Variables locales

L'opérateur `my` permet de créer des variables locales :

- il prend une liste de variables en paramètre ;
- sauvegarde leur valeur courante si nécessaire ;
- la valeur initiale des variables locales est `undef`.

Exemple.

```
bash$ cat essai.pl
$a = 1;
$b = 2;
print "result = ", fonction(), ", a = $a, b = $b\n";
sub fonction {
    my ($a) = 10;
    $b = $b + $a;
    return $b;
}
```

Exécution :

```
bash$ essai.pl
result = 12, a = 1, b = 12
```

Variables semi-locales

`local` permet de définir des variables « moins locales » que celles définies avec `my` :

- les variables crées par `local` sont visibles dans toutes les fonctions appelées par la fonction dans laquelle elles ont été définies ;
- `my` ne permet que de définir des variables de nom « normal », cette restriction ne s'applique pas à `local` ;
- `my` est plus rapide.

Exemple

```
fonction ("toto", `ls`);
sub fonction {
    local ($mot) = $_[0];
    my (@fichiers) = (@_[1...$#_]);
    foreach (@fichiers) {
       print "********* fin $_";
       traiter_fichier ($_) if -f ;
       print "********* fin $_";
   }
}
sub traiter_fichier {
    local $_;
    my ($fichier) = $_[0];
    open (FICH, $fichier) || die ("ouverture $fichier");
    while (<FICH>) { print if (/$mot/); }
    close (FICH) || die ("fermeture $fichier");
}
```

Avertissement

- Ceci n'est pas un cours sur les sockets, simplement des indications pour utiliser les sockets en Perl.
- Nous nous intéressons uniquement à TCP.
- Nous utiliserons le module `Socket`, il en existe d'autres de plus haut niveau.

Principe

- Un programme A « écoute » sur un port P (+/- le serveur).
- Un autre programme B connaissant l'adresse IP de la machine sur laquelle A s'exécute et le numéro du port va essayer de se connecter (+/- le client).
- Si tout se passe bien, A et B vont pouvoir communiquer jusqu'à fermeture de la connexion par l'un des deux.
- La communication est bi-directionnelle.
- Le principe de TCP est d'établir un « circuit virtuel » : les paquets sont délivrés au destinataire dans leur ordre d'émission et sans perte.

Création d'une socket

La fonction

```
socket(SOCKET, DOMAINE, TYPE, PROTOCOLE)
```

- permet de créer une socket et de l'attacher au handle `SOCKET` ;
- le `DOMAINE` désigne un domaine de communications, `PF_INET` désigne le protocol `IPv4` ;
- le `TYPE` spécifie le type de communication, `SOCK_STREAM` désigne une connexion fiable, séquencée, bi-directionnelle en flot d'octets (à utiliser pour TCP) ;
- le `PROTOCOLE` désigne le protocole.

Remarques.

- Utiliser la fonction `getprotobyname` pour convertir le nom du protocole (par exemple `"tcp"`) en son numéro.
- La plupart des fonctions utilisées par la suite sont succeptibles de produire des erreurs : il FAUT tester les codes de retour.

Connecter la socket

Une fois la socket créée, il faut la connecter à la machine distante. La fonction

```
connect (SOCKET, NAME)
```

- permet d'initier la connexion, à condition qu'un processus soit en train d'attendre « à l'autre bout » ;
- `SOCKET` est le handler d'une socket qui a été créée dans le bon mode ;
- `NAME` est une adresse qui doit être au bon format pour la socket en question ;
- retourne vrai en cas de succès, faux sinon.

Remarque. Utiliser la fonction `sockaddr_in(port, nom)` pour fabriquer une adresse valide dans notre cas.

Exemple – client

```
use Socket;

socket (SERVEUR, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

$adresse = inet_aton ("localhost") || die ("inet_aton");
$adresse_complete = sockaddr_in("3000",$adresse)
                                || die ("sockaddr_in");
connect (SERVEUR, $adresse_complete) || die ("connect");

print "OK\n";
while (<SERVEUR>) {
    print "test } ",$_;
}
close (SERVEUR);
```

bind

La fonction

```
bind (SOCKET, SOCKADDR)
```

- permet d'attacher une adresse à une socket déjà créée ;
- l'adresse `SOCKADDR` doit être une adresse valide pour le type de socket considéré ;
- dans notre cas, l'adresse est donnée par `sockaddr_in($port, INADDR_ANY)` dans lequel `$port` désigne le numéro du port sur lequel le « serveur » doit attendre et `INADDR_ANY` n'importe quelle adresse.

Un certain nombre d'options peuvent être précisées `setsockopt (SOCKET, SOL_SOCKET, SO_REUSEADDR, 1)` permet de réutiliser l'adresse immédiatement après l'avoir libérée (voir `man 2 setsockopt` pour plus de détails).

Mise en attente sur la socket

La fonction

```
listen (SOCKET, QUEUESIZE)
```

- signale au système que le « serveur » est prêt à accepter les connexions sur `SOCKET` ;
- indique la taille de la queue, c'est-à-dire le nombre de connexions qui peuvent être en attente ;
- retourne vrai en cas de succès, faux sinon.

Accepter les connexions

La fonction

```
accept(NEWSOCKET, SOCKET)
```

- permet d'établir une connexion avec un client ;
- `SOCKET` désigne une socket qui a été créée et correctement initialisée (`bind`, `listen`, ...) ;
- est bloquante jusqu'à ce qu'un client demande une connexion ;
- `NEWSOCKET` est un nouveau handle attaché à la nouvelle connexion, il va permettre au « serveur » de dialoguer avec le « client » dont il vient d'accepter la connexion et uniquement celui-là ;
- `SOCKET` n'est pas modifiée et d'autres clients peuvent continuer à faire des demandes de connexion.

Exemple – Serveur

```
use Socket;

socket (SERVEUR, PF_INET, SOCK_STREAM, getprotobyname('tcp'));
setsockopt (SERVEUR, SOL_SOCKET, SO_REUSEADDR, 1);

$mon_adresse = sockaddr_in ("3000", INADDR_ANY);

bind(SERVEUR, $mon_adresse) || die ("bind");

listen (SERVEUR, SOMAXCONN) || die ("listen");

accept (CLIENT, SERVEUR) || die ("accept");
select (CLIENT);
print "bonjour\n";
close (CLIENT);
close (SERVEUR);
```

Processus

- Objet dynamique correspondant à l'exécution d'un programme.
- Un processus possède un espace d'adressage qui définit l'ensemble des objets qui lui sont propres (instructions et données).
- Le processus peut s'exécuter dans deux modes différents :
  - en **mode utilisateur**, le processus exécute des instructions du programme et accède aux données de son espace d'adressage.
  - en **mode noyau**, le processus exécute des instructions du noyau et a accès à l'ensemble des données du système (par exemple lors des appels système).
- Chaque processus possède un espace d'adressage de données propres, plusieurs processus peuvent partager le même programme (code réentrant).

Naissance des processus

- Tout processus peut créer de nouveaux processus.
- Tout processus (sauf le premier) est créé par un appel à la primitive `fork`.
- La primitive `fork` a pour effet de dupliquer le processus appelant.
- Les processus sont organisés en arborescence en fonction de leur processus créateur appelé **père**.
- Le noyau du système a en charge la gestion des différents processus et le partage des ressources entre-eux, en particulier l' **ordonnancement** des processus : choisir parmi les processus en attente celui qui doit être activé.

Caractéristiques d'un processus

- Identité du processus.
- Identité du père du processus.
- Liens avec les utilisateurs :
  - propriétaire réel du processus (`uid` de l'utilisateur qui a « lancé » le processus) ;
  - propriétaire effectif du processus (différent du propriétaire réel par exemple lorsqu'un processus correspond à l'exécution d'un programme dont le bit `u` est positionné) ;
  - groupe réel du processus ;
  - groupe effectif du processus (différent du groupe réel par exemple lorsqu'un processus correspond à l'exécution d'un programme dont de bit `g` est positionné).

Remarque. Un processus ayant des droits privilégiés peut modifier ses propriétaires et groupes réels ou effectifs (procédure utilisée à la connexion d'un utilisateur).

Caractéristiques (suite)

- Le répertoire de travail du processus.
- Le groupe de processus et la session auxquels le processus appartient.
- La date de création du processus.
- Les temps CPU consommés par le processus en modes utilisateur et noyau ainsi que par ses fils terminés.
- Le masque de création des fichiers.
- La table des descripteurs de fichiers.
- L'état du processus.
- L'événement attendu par le processus s'il est à l'état endormi.
- Les informations pour le traitement des signaux.
- Les verrous sur les fichiers.
- ...

État d'un processus

Au cours de sa vie, un processus passe par différents états :

- état **transitoire** à sa création ou lors de la création d'un fils ;
- état **actif** (en mode noyau ou utilisateur) (état **R** donné par `ps`) ;
- état **prêt**, en attente de la CPU (état **R** donné par `ps`) ;
- état **endormi**, en attente d'un événement (attente d'entrées/sorties, attente de terminaison d'un processus, attente d'un signal, ...) (état **S** ou **D** donné par `ps`) ;
- état **suspendu** (état **T** donné par `ps`) ;
- état **zombi**, processus terminé mais dont le père n'a pas encore pris connaissance de la terminaison (état **Z** donné par `ps`).

Organisation mémoire

Le processus est constitué de 4 segments mémoire :

- le **bloc de contrôle** qui contient les informations utiles au système. Cette partie n'est pas directement accessibles aux utilisateurs. Les éléments du bloc de contrôle sont de deux natures :
  - les informations utiles lorsque le processus est actif (descripteurs de fichiers, signaux, ...), elles appartiennent à l'espace d'adressage du processus,
  - les informations utiles au système même lorsque le processus n'est pas actif (identité, priorité, ...), elles n'appartiennent pas à l'espace d'adressage du processus.

  Le bloc de contrôle a une taille fixe quel que soit le processus ;

Organisation mémoire (2)

- Les **instructions** qui appartiennent à l'espace d'adressage du processus et peuvent être partagées entre plusieurs processus si le code est réentrant, de taille fixe pour un programme donné ;
- Les **données** manipulées par le programme qui appartiennent par excellence à l'espace d'adressage du processus et dont la taille varie au grès des allocations mémoire ;
- La **pile** dont la taille varie en fonction de l'imbrication des appels de fonctions (allocation des variables locales, sauvegarde des contextes, ...).

Appel système `fork`

La fonction **`fork`** invoque l'appel système `fork` qui permet de dupliquer un processus :

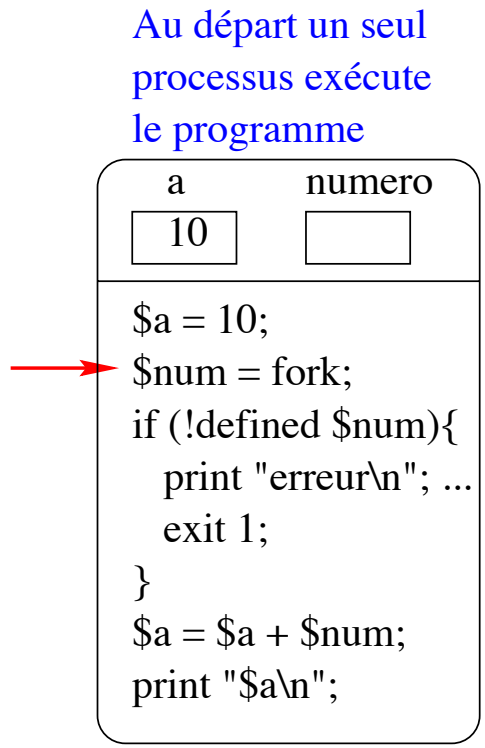
- elle permet de créer un nouveau processus (appelé fils) à partir de celui qui réalise l'appel (appelé père) ;
- après la création, les deux processus semblent avoir exécuté l'appel à la primitive **`fork`**, chacun des processus continue son exécution à partir de l'instruction qui suit le `fork` ;
- la valeur de retour de `fork`, permet de différencier les processus père et fils :
  - `0` dans le fils,
  - le `pid` du fils dans le père ;
- la fonction retourne `undef` en cas d'échec, dans ce cas, il n'y a pas création d'un nouveau processus.

Exemple 1

```
bash$ cat essai.pl
$retour_du_fork = fork;
$mon_pid = $$;
if ($retour_du_fork != 0) {
   print "je suis le pere de pid : $mon_pid,"
         " mon fils est : $retour_du_fork\n";
}
elsif (defined $retour_du_fork) {
   print "je suis le fils de pid : $mon_pid\n";
}
else {
   print STDERR "pas de fork";
}
bash$ essai.pl
je suis le pere de pid : 859 mon fils est : 860
bash$ je suis le fils de pid : 860
bash$
```

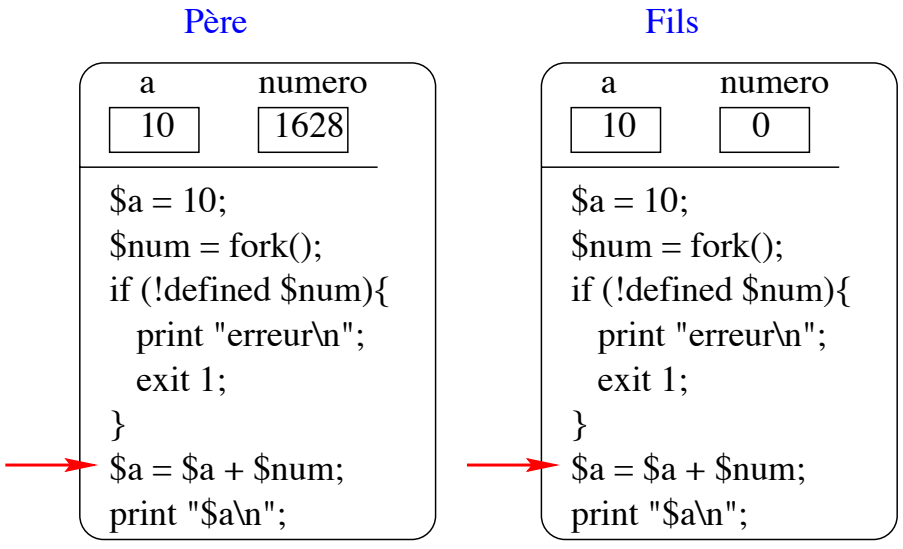
Exemple 2

```
#! /usr/bin/perl
$a= 10;
$num = fork();
if (! defined $num) {
      print "Erreur du fork\n";
      exit 1;
}
$a = $a+$num;
print "$a\n";
```

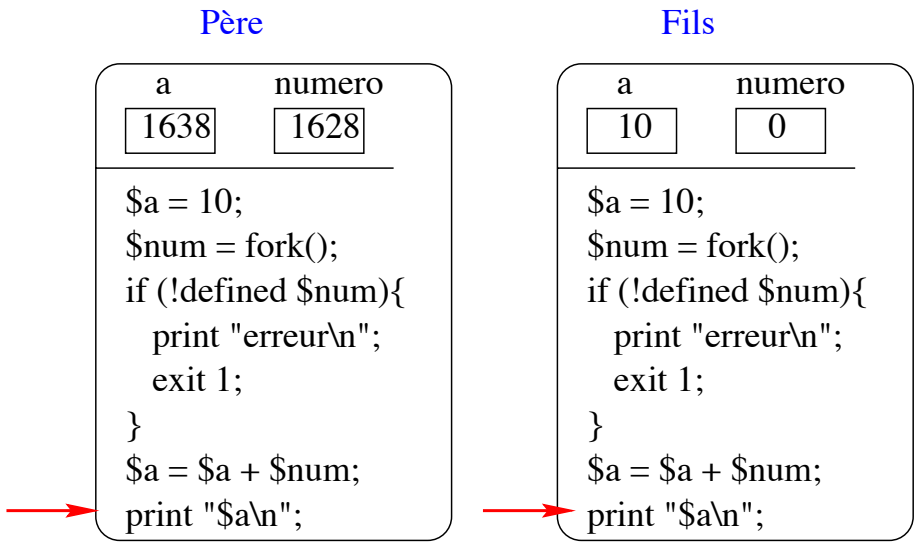



```
bash$ essai.pl
10
1638
```

Exemple 2 – suite



Après exécution du fork, deux processus continuent à exécuter le programme "comme s'ils venaient tous deux d'exécuter l'appel à fork.



Les deux processus ont chacun leur propre environnement et leurs propres variables.

Génétique de processus

Le processus fils hérite les caractéristiques de son père excepté :

- le `pid` du fils est différent de celui du père (le `pid` étant l'identifiant d'un unique processus) ;
- le `pid` du père ;
- les temps CPU (ils sont mis à `0` pour le fils) ;
- les verrous sur les fichiers ;
- les signaux pendants (voir cours sur les signaux) ;
- la priorité (la priorité est utilisée pour l'ordonnancement, la priorité du fils est initialisée à une valeur standard lors de sa création).

Terminaison des processus

- Tout processus UNIX possède une valeur de retour (valeur de retour de la fonction main, ou code utilisé pour la fonction `exit`) à laquelle son père peut accéder.
- Tout processus se terminant passe dans l'état **zombi** (état `Z` indiqué par commande `ps`), jusqu'à ce que son père prenne connaissance de sa terminaison.
- Le mécanisme de processus `zombi` permet à un processus d'accéder au code de retour de ses processus fils de manière asynchrone.

Remarque. Si le processus père se termine sans avoir pris connaissance de la terminaison d'un de ses fils, celui-ci est adopté par le processus de `pid 1` qui prend connaissance du code de retour du fils et lui permet ainsi de se terminer.

Exemple

```
#! /usr/bin/perl

$pid = fork();
if ($pid != 0) {
   print "Processus de pid ", $$, " de pere ", getppid(), "\n";
   print "Fin du pere dont le fils est $pid \n";
}
else {

  sleep(2);

   print "Processus de pid ", $$, " de pere " ,getppid(), "\n";
   print "Fin du fils, valeur du fork $pid\n";
}
```

Exemple – suite

Exécution sans l'instruction `sleep(2)` :

```
bash$ essai
Processus de pid 1189 de pere 679
Processus de pid 1190 de pere 1189
bash$ Fin du pere dont le fils est 1190
Fin du fils, valeur du fork 0
```

Exécution avec l'instruction `sleep(2)` :

```
bash$ essai
Processus de pid 1105 de pere 679
Fin du pere dont le fils est 1106
bash$ Processus de pid 1106 de pere 1
Fin du fils, valeur du fork 0
```

Un processus orphelin est adopté par le processus de `pid 1`.

Terminaison d'un fils

Il peut parfois être nécessaire d'attendre la terminaison d'un fils, parce qu'on a besoin de se synchroniser ou simplement pour prendre connaissance de sa terminaison et l'éliminer (sinon il reste « zombi »).

- La fonction **`wait`** permet d'attendre la terminaison d'un fils. Elle retourne le `pid` du fils terminé, (attention, le système choisit le fils) ou `-1` s'il n'y a pas de fils à attendre.
- La fonction **`waitpid`** permet d'attendre un fils au choix avec des options :
  - 1^{er} paramètre : le `pid` du fils attendu ou `-1` pour « tout fils », ...
  - $2^{ème}$ paramètre : option par exemple `WNOHANG` pour le mode non bloquant ;
  - retourne le `pid` du fils terminé ou `0` en mode non bloquant s'il n'y a pas de fils terminé, ou `-1` s'il n'y a pas de fils à attendre.

Dans les deux cas, le status de terminaison du fils se trouve dans `$?`.

Exemple – wait

```
$pid = fork();
if ($pid != 0) {
   print "Processus de pid ", $$, " de pere ", getppid(), "\n";
   if (($n = wait()) == -1) {
       print "erreur du wait\n";
       exit 1;
   }
   print "Recuperation du fils $n\n";
}
else {
   print "Processus de pid ", $$, " de pere " ,getppid(), "\n";
}
bash$ essai.pl
Processus de pid 1205 de pere 679
Processus de pid 1206 de pere 1205
Recuperation du fils 1206
```

Exemple – waitpid

```
#! /usr/bin/perl
use POSIX ":sys_wait_h";
$pid = fork();
if ($pid != 0) {
   print "Processus de pid ", $$, " de pere ", getppid(), "\n";
   while (($n = waitpid($pid, &WNOHANG)) == 0)
       { print "toujours pas mort\n";}
   if ($n == -1)
       { print "erreur waitpid\n"; exit 1;}
   print "Recuperation du fils $n\n";
}
else {
   print "Processus de pid ", $$, " de pere " ,getppid(), "\n";
   sleep(1);
   print "Fin du fils\n";
}
```

Exemple – waitpid

```
bash$ essai.pl
Processus de pid 1259 de pere 679
toujours pas mort
...
toujours pas mort
toujProcessus de pid 1260 de pere 1259
Fin du fils
ours pas mort
toujours pas mort
toujours pas mort
Recuperation du fils 1260
```

Fonction `system`

La fonction `system` permet de lancer une commande dans un shell :

- un nouveau shell est lancé et est chargé d'exécuter la commande ;
- les entrée, sortie et sortie d'erreur sont héritées du processus Perl ;
- toute chaîne qui convient au shell peut être utilisée ;
- le processus Perl attend la fin de l'exécution de la commande avant de poursuivre sa propre exécution (sauf utilisation de &).

Exemple.

```
if (system ("ls") != 0) {
   print "erreur\n";
}
```

Les « backquotes »

Lorsqu'une commande est placée en backquotes, un processus est lancé :

- l'entrée et la sortie d'erreur sont héritées du processus Perl ;
- la sortie standard est interceptée ;
- le processus Perl attend la fin de l'exécution de la commande avant de poursuivre sa propre exécution.

Exemple.

```
$result = `ls`;
print "$result***\n";
foreach (`ls`) { chomp; print if -d; }
bash$ essai.pl
rep
titi.txt
toto.txt
***
rep
```

Processus et handles de fichiers

Il est possible de créer un processus et de le manipuler comme un handle de fichier, sur le principe des tubes :

- `open (HANDLE1, "ls|");` crée un processus qui exécute la commande `ls`. Le handle `HANDLE1` est ouvert en lecture et permet de lire la sortie standard de la commande ;
- `open (HANDLE2, "|cat");` crée un processus qui exécute la commande `cat`. Le handle `HANDLE2` est ouvert en écriture et permet d'envoyer des données à l'entrée standard de la commande ;
- les deux processus (celui qui fait l'appel à `open` et celui qui est créé) sont indépendants ;
- la fermeture des handles (`close`) oblige le processus appelant à attendre la terminaison du processus associé au handle ;
- les autres entrées/sorties standards sont héritées du processus Perl.

Recouvrement de code

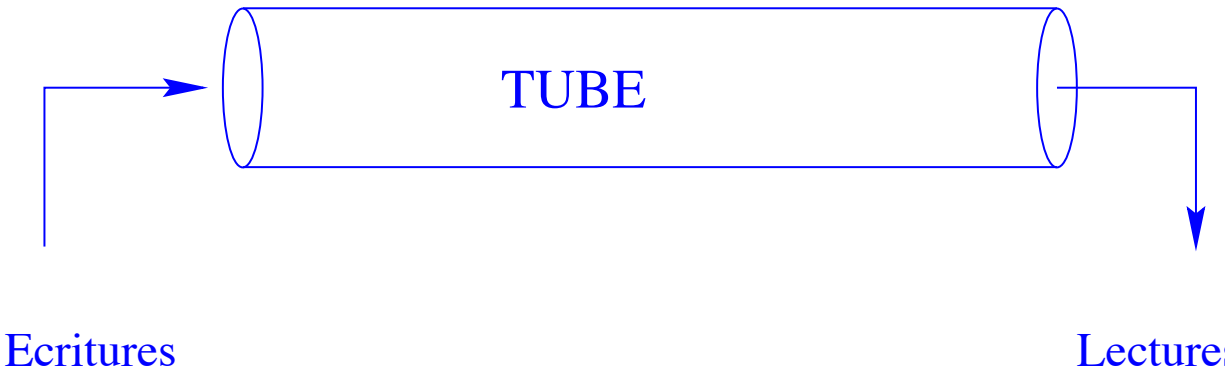
L'appel à la fonction `exec` entraîne un recouvrement de code :

- recouvrement de code : le code du processus en cours (celui qui a exécuté le `exec`) est remplacé par celui du programme à exécuter. Si l'appel réussi, il n'y a JAMAIS de retour.
- en Perl : ressemble à la fonction `system`. Le paramètre est une commande à faire exécuter par un shell.

```
bash$ cat essai.pl
#! /usr/bin/perl

print "Bonjour\n";
exec ("ls");
print "Au revoir\n";
bash$ essai.pl
Bonjour
essai.pl  rep  titi.txt  toto.sh  toto.txt
```

Rappels

- Les tubes sont des mécanismes permettant aux processus de communiquer entre-eux.
- Les tubes appartiennent au système de fichiers UNIX, *i.e.* ils sont décrits par un i-nœud.
- Les tubes peuvent donc être manipulés par l'intermédiaire de descripteurs de fichiers et par les primitives usuelles sur les fichiers.
- Les tubes sont des moyens de communication **unidirectionnels**.

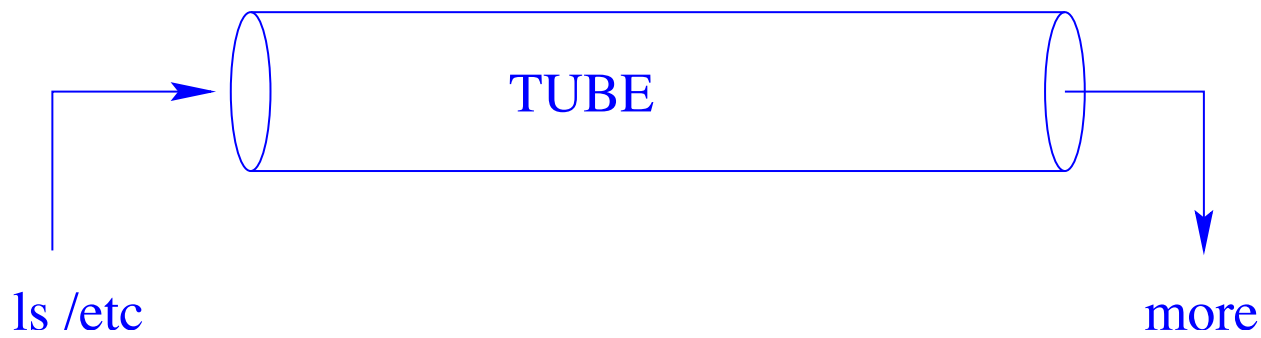


Les tubes dans les commandes `shell`

```
$ ls /etc | more
DIR_COLORS
HOSTNAME
X11/
a2ps-site.cfg
a2ps.cfg
adjtime
-More-
```

☛ **frappe d'un Ctrl-z par l'utilisateur**

```
Suspended
[7] 21369 21370
$ fg
ls -F /etc | more
aliases
```



Les tubes et `open`

Lorsque l'on crée un processus et qu'on le manipule comme un handle de fichier, on utilise les tubes (voir cours sur les processus) :

- l'appel `open(HANDLE,"|com");` (resp. `open(HANDLE,"com|");`) crée un processus exécutant la commande `com` et un tube permettant de communiquer du processus appelant vers le processus exécutant la commande `com` (resp. du processus exécutant la commande `com` vers le processus appelant) ;
- l'appel de `close` force le processus appelant à attendre la terminaison du processus exécutant la commande.

```
bash$ cat essai.pl
open (HANDLE, "echo *.pl|");
@a = <HANDLE> ;
close (HANDLE);
print STDOUT "@a\n";
bash$ essai.pl
client.pl client_http.pl essai.pl ser.pl serveur.pl
```

Tubes et `open` (2)

La forme

```
open (H, "|-")
```

- permet de créer en même temps un fils et un tube pour communiquer avec lui ;
- `open` retourne (comme `fork`) le `pid` du fils dans le père et `0` dans le fils ;
- si le deuxième paramètre est `"|-"`, le handle `H` permet au père d'écrire dans le tube et le tube est connecté à l'entrée standard du fils ;
- si le deuxième paramètre est `"-|"`, le handle `H` permet au père de lire dans le tube et le tube est connecté à la sortie standard du fils.

Exemple

```
$pid = open (VERS, "|-");
if (! defined ($pid)) { print "fork rate\n"; exit (1);}
if ($pid == 0) {
   print "je suis le fils de pid $$\n";
   while (<STDIN>) { print "mon pere ecrit : $_";}
}
else {
   autoflush VERS 1;
   print "je suis le pere de pid $$ mon fils est $pid\n";
   $a =<STDIN>;
   while ($a ne "fin\n") {
       print VERS $a;
       $a = <STDIN>;
   }
   close(VERS);
}
```

Exemple

```
bash$ essai.pl
je suis le pere de pid 7573 mon fils est 7574
je suis le fils de pid 7574
aaa
mon pere ecrit : aaa
zzz
mon pere ecrit : zzz
eee
mon pere ecrit : eee
fin
bash$ printf "aaa\nzzz\neee\nfin\n" | essai.pl
je suis le pere de pid 7571 mon fils est 7572
je suis le fils de pid 7572
mon pere ecrit : aaa
mon pere ecrit : zzz
mon pere ecrit : eee
```

Créer un tube

La fonction

`pipe(LECTURE, ECRITURE)`

- permet de créer un tube ;
- `LECTURE` est le handler sur l'entrée en lecture du tube ;
- `ECRITURE` est le handler sur l'entrée en écriture du tube ;
- permet d'obtenir directement les descripteurs sur le tube, il n'est pas nécessaire d'utiliser par la suite des opérations d'ouverture.

Exemple

```
pipe (LECTURE, ECRITURE) || die "erreur pipe";
$pid = fork;
if (!defined $pid) {
    print ("erreur du fork");
    exit 1;
}
if ($pid == 0) {
    close (LECTURE);
    print ECRITURE "bonjour\n";
    close(ECRITURE);
}
else {
    close (ECRITURE);
    $a = <LECTURE>;
    print "recu $a";
    close (LECTURE);
}
```

```
bash$ essai.pl
recu bonjour
```

Généralités

- Un tube correspond au plus à deux entrées dans la table des fichiers ouverts (une entrée en lecture et une entrée en écriture).
- Les données ne sont pas formatées, elles apparaissent comme un flot de caractères. Le tube est géré en file, *i.e.* la première donnée écrite dans le tube est également la première donnée lue.
- Attention, un tube a une capacité finie !
- Le nombre de lecteurs d'un tube est le nombre de descripteurs associés à l'entrée en lecture sur le tube. Si ce nombre est nul, il est impossible d'écrire dans le tube.
- Le nombre de rédacteurs d'un tube est le nombre de descripteurs associés à l'entrée en écriture sur le tube. Si ce nombre est nul, les fonctions de lecture détectent une fin de fichier.

Tubes vs Fichiers

- Comme les tubes n'ont pas de noms, il est impossible de les ouvrir grâce à la primitive `open`. En conséquence, un processus peut acquérir un descripteur sur un tube, soit en le créant, soit par héritage.
- Seuls le processus ayant créé le tube et sa descendance peuvent y accéder. Si un processus perd son accès au descripteur sur le tube (par exemple par un appel à `close`), il n'a aucun moyen de le récupérer par la suite.
- Les lectures dans un tube sont « effaçantes », il s'agit bien d'une extraction des données du tube et non d'une consultation.

Lectures

- Le descripteur en lecture permet d'extraire des données du tube.
- La lecture est par défaut bloquante :

```
bash$ cat essai.pl
#! /usr/bin/perl
pipe (LECTURE, ECRITURE) || die "erreur pipe";
$ligne = <LECTURE>;
print ("processus fini\n");
bash$ essai.pl
```

☛ le processus ne se termine pas !

- Si le nombre d'écrivains est nul, les fonctions permettant de lire détectent la fin de fichier.

Pourquoi utiliser `close` ?

```
pipe (LECTURE, ECRITURE) || die "erreur pipe";
$pid = fork;
if (!defined $pid) print ("erreur du fork"); exit 1;
if ($pid == 0) {
 close (LECTURE);
 print ECRITURE "bonjour\n";
 print ECRITURE "coucou\n";
 print ECRITURE "hello\n";
 close(ECRITURE);
 print ("fils fini\n");
} else {
 close (ECRITURE);                                    #instruction à tester
 while (<LECTURE>)  print "recu $_";
 close (LECTURE);
 print ("pere fini\n");
}
```

Pourquoi utiliser `close` ? (2)

- Avec le `close(ECRITURE)` :

```
bash$ essai.pl
fils fini
recu bonjour
recu coucou
recu hello
pere fini
bash$
```

- Sans le `close(ECRITURE)` :

```
bash$  essai.pl
fils fini
recu bonjour
recu coucou
recu hello
```

☛ le processus père ne se termine pas !

Écritures

- Le descripteur en écriture permet d'écrire des données dans le tube.
- Si le nombre de lecteurs est nul, le processus qui tente une écriture reçoit un SIGPIPE.
- Si le nombre de lecteurs n'est pas nul, les données sont envoyées dans le tube.

```
bash$ cat essai.pl
#! /usr/bin/perl
pipe (LECTURE, ECRITURE) || die "erreur pipe";
close (LECTURE);
print ECRITURE "bonjour\n";
print ECRITURE "coucou\n";
print ECRITURE "hello\n";
close(ECRITURE);
print ("processus fini\n");
```

```
bash$ essai.pl
Broken pipe
```

Les tubes nommés

Les tubes nommés ou fifo ont été introduits dans la version III d'UNIX.

- Leur but est de permettre à des processus sans lien de parenté particulier de communiquer par l'intermédiaire de tubes.
- Ils ont toutes les caractéristiques des tubes et ont en plus une référence dans le système de fichiers.
- Tout processus connaissant la référence d'un tube peut l'ouvrir avec la primitive `open` (modulo vérification des droits comme pour tout fichier).
- Les fichiers correspondant à des tubes sont identifiés par `ls`.

Exemple.

```
bash$ ls -l
...
prw-r-r- 1 ryl ens 0 Oct 29 19:14 serveur|
```

Création d'un tube nommé

La fonction :

```
mkfifo (<nom>, <droits>)
```

- permet de créer un tube nommé ;
- `nom` désigne le nom (nom relatif ou nom absolu) du fichier associé au tube ;
- `mode`, exprimé en octal, désigne le mode du fichier ainsi créé (attention au `umask`) ;
- attention, le tube est simplement **créé**, il faut ensuite l'**ouvrir** pour pouvoir l'utiliser.

Utilisation d'un tube nommé

- Un processus connaissant le « nom » d'un tube peut l'ouvrir grâce à la primitive `open` (si les droits du tube le permettent).
- Attention, `open` est dans ce cas bloquant (une ouverture en lecture est bloquante tant qu'il n'y a aucun écrivain sur le tube et vice et versa).
- Le descripteur de fichier obtenu par `open` peut ensuite être utilisé comme tout autre descripteur.
- Il ne faut pas oublier de supprimer les fichiers associés aux tubes lorsque l'on en a plus besoin :

  ```
  unlink(LISTE_DE_FICHIERS)
  ```

  permet de supprimer « proprement » ces fichiers du système de fichiers (lire **`man 2 unlink`**).

Exemple

```
bash$ cat p1.pl
#! /usr/bin/perl
use POSIX qw(mkfifo);
mkfifo ("/tmp/toto", 0644) || die "mkfifo impossible";
open (ECRITURE, ">>/tmp/toto") || die "open en lecture";
print ECRITURE "Bonjour\n";
close(ECRITURE);
unlink ("/tmp/toto");
bash$ cat p2.pl
#! /usr/bin/perl
open (LECTURE, "/tmp/toto") || die "open en ecriture";
$mot = <LECTURE>;
print "recu : $mot";
close (LECTURE);
```

Signal

- Le signal joue le rôle d'une sonnerie d'alerte qui est « entendue » par un processus et qui signale un événement particulier.
- Les signaux sont des moyens de communiquer avec les processus, un signal peut être envoyé :
  - par une commande ;
  - par un appel système dans un programme.
- Différents signaux signalent différents événements. Cependant, le signal peut être déclenché « artificiellement » même si l'événement ne s'est pas produit. La seule information dont le processus recevant le signal dispose est le nom de celui-ci, il ne possède aucun moyen de savoir si l'événement associé s'est réellement produit.
- Le signal peut être vu comme une interruption logicielle.

État des signaux

Lorsqu'un signal est envoyé à un processus, celui-ci doit le gérer correctement. Un signal peut être dans différents états :

- un signal **pendant** est un signal qui a été envoyé à un processus mais qui n'a pas encore été pris en compte par celui-ci. **Attention**, un seul signal de chaque type peut être pendant, si un deuxième signal du même type arrive, il est perdu ;
- un signal est **délivré** au processus lorsque celui-ci en prend connaissance. Le processus exécute alors la fonction de traitement du signal ;
- un signal peut être **bloqué** ou **masqué**, dans ce cas, il ne sera pas délivré au processus.

Noms des signaux

- Pour augmenter la portabilité des applications, les signaux sont nommés par des constantes ayant un nom « évocateur » (par exemple `SIGTERM` ou `SIGILL`).
- Les signaux peuvent provenir d'un événement extérieur au processus, par exemple envoyés par un utilisateur ou par un autre processus, ou provenir d'un événement intérieur au processus ayant provoqué une erreur (division par zéro, violation d'une zone mémoire, ...).

Exemple. L'envoi du signal `SIGSEGV` à un processus provoque sa terminaison ainsi que la création d'un fichier `core`. Il est utilisé lors d'une violation mémoire par le processus mais peut aussi être envoyé par l'utilisateur sans qu'aucune violation de la mémoire ne se soit produite.

Principaux signaux

La liste des signaux peut être obtenue par la commande `kill -l`. Utiliser `man 7 signal` pour obtenir la liste des signaux ainsi que leur code.

Quelques exemples :

- `SIGILL` détection d'une instruction illégale ;
- `SIGKILL` signal de terminaison ;
- `SIGTERM` signal de terminaison ;
- `SIGUSR1` signal réservé utilisateur ;
- `SIGUSR2` signal réservé utilisateur ;
- `SIGCHLD` signal de terminaison d'un fils ;
- `SIGSTOP` signal de suspension ;
- `SIGINT` signal d'interruption ;
- `SIGCONT` signal de continuation pour un processus suspendu.

Envoyer un signal

La fonction

```
kill(SIGNAL, LISTE_DE_PID)
```

- permet d'envoyer le signal demandé à la liste des processus indiqués à condition qu'ils appartiennent au même utilisateur ou que la fonction soit utilisée par le super-utilisateur ;
- le paramètre `SIGNAL` est le numéro du signal ou le nom du signal privé de `'SIG'` ;
- si le premier paramètre vaut `0`, la fonction permet de tester la possibilité d'envoyer un signal *i.e.* aucun signal n'est envoyé, la fonction teste si les processus indiqués existent et appartiennent au bon utilisateur ;
- retourne le nombre de processus qui ont pu être atteints.

Exemple

```
bash$ cat essai.pl
#! /usr/bin/perl
$pid = fork;
if ($pid == 0) {
    while (1) {
        print "fils OK\n";
        sleep (1);
    }
}
else {
    print "pere demarre\n";
    sleep (1);
    kill (15, $pid);          # kill ('TERM', $pid);
    print "signal envoye\n";
}
```

```
bash$ essai.pl
pere demarre
fils OK
fils OK
signal envoye
```

Recevoir un signal

- Lorqu'un processus reçoit un signal, il doit le traiter, c'est-à-dire réaliser l'action prévue pour ce signal.
- À chaque signal est associé un **handler** définissant le comportement par défaut d'un processus recevant ce signal.
- Le comportement dépend du type de signal reçu, différents comportements sont possibles :
  - terminaison du processus (ex : `SIGKILL`) ;
  - terminaison du processus avec création d'un fichier `core` (ex : `SIGSEGV`) ;
  - suspension du processus (ex : `SIGSTOP`) ;
  - reprise d'un processus suspendu (ex : `SIGCONT`).
- Certains signaux sont ignorés (par défaut `SIGCHLD` est ignoré) ou masqués.

Traitement du signal

À la prise en compte d'un signal par un processus, le traitement associé à ce signal est exécuté, le déroulement normal du processus est donc interrompu et reprend éventuellement après traitement du signal.

- Le signal est délivré au processus lorsque celui-ci passe du mode noyau au mode utilisateur, le processus ne peut pas être interrompu lorsqu'il se trouve en mode noyau.
- Si le processus est endormi à un niveau de priorité interruptible (ex : attente de signal), le processus passe dans l'état prêt et il reçoit le signal lorsqu'il repasse en état actif. Si le signal est ignoré, le processus reprend le cours de son exécution (il peut éventuellement se rendormir).
- Si le processus est stoppé, `SIGKILL` et `SIGTERM` le terminent, `SIGCONT` le réveille, les autres signaux lui sont délivrés au réveil.

Associer un handler à un signal

L'utilisateur peut définir lui-même les traitements qu'il veut voir associer aux différents signaux :

- les traitements associés aux signaux (noms des fonctions) sont stockés dans le hachage `%SIG` ;
- les fonctions de traitement de signal doivent être « simples ». En général, positionnement d'une variable globale ou quelques instructions avant de quitter (voir problèmes de réentrance) ;
- les valeurs prédéfinies `IGNORE` et `DEFAULT` permettent respectivement d'indiquer qu'un signal doit être ignorer et que le comportement par défaut doit être restauré ;
- les comportements associés à certains signaux ne peuvent être redéfinis (`SIGKILL`, `SIGSTOP`) ;
- si les signaux `SIGFPE`, `SIGILL` ou `SIGSEGV` sont ignorés, le comportement du programme est indéterminé.

Exemple

```
#! /usr/bin/perl
sub fin {
    print "fin cpt = $cpt\n";
    exit (1);
}
$pid = fork;
if ($pid != 0) {
    $drapeau=0;
    $cpt = 0;
    $SIG{'INT'} = 'fin';
    $SIG{'USR1'} = sub {$drapeau = ($drapeau + 1) %2; $cpt ++;};
    while (1) {
        sleep (1);
        if ($drapeau == 0) { print "drapeau a 0\n";}
        else { print "drapeau a 1\n";}
    }
}
```

Exemple

```
else
 print "fils demarre\n";
 $resultat = kill ('USR1', getppid());
  sleep (1);
 $resultat = kill ('USR1', getppid());
  sleep (1);
 $resultat = kill ('USR1', getppid());
  sleep (1);
 $resultat = kill ('INT', getppid());
 print "signal de fin envoye $resultat \n";
```

Exemple

- Avec les instructions `sleep(1);` :

```
bash$  essai.pl
fils demarre
drapeau a 1
drapeau a 0
drapeau a 1
signal de fin envoye 1
fin cpt = 3
```

- Sans les instructions `sleep(1);` :

```
bash$ essai.pl
fils demarre
signal envoye de fin 1
fin cpt = 1
```

Problématique

- Point de vue « utilisateur » : sauvegarder des données, les organiser, y avoir accès facilement. Essentiellement caractérisé par :
  - un nom ;
  - éventuellement une localisation.
- Point de vue « système » :
  - gestion des ressources disques et autres ;
  - différentes informations comme la taille, la date de création, date de dernière modification, ...
- Point de vue « système multi-utilisateurs » :
  - partage des ressources entre les utilisateurs ;
  - protection des fichiers.

Les « fichiers » Unix

« En Unix, tout est fichier. »

Le terme de « fichier » désigne des ressources :

- matérielles – disque, disquettes, terminal, ...
- logicielles – fichiers disques « classiques » contenant des données mémorisées sur le disque.

Les primitives génériques d'accès aux fichiers permettent de réaliser des opérations de lecture et d'écriture sur toutes les ressources du système.

En interne, chaque fichier est associé à une structure décrivant ses caractéristiques appelée **i-nœud** ou **i-node**.

Organisation logique

- Les « fichiers » UNIX peuvent être des fichiers disques « classiques » ou des fichiers « ressources ».
- Plusieurs disques peuvent être connectés à une machine ainsi que de nombreuses ressources.
- Chaque disque physique peut également être partitionné en plusieurs disques logiques.
- À chaque fichier correspond un i-nœud (i-node) qui contient entre-autres, l'identification du disque logique du fichier et son numéro d'identification dans ce disque logique.

Tous les fichiers apparaissent à l'utilisateur dans une arborescence unique.

Les i-nœuds

Un fichier n'est pas seulement repéré par son nom : à chaque fichier est associé un **i-nœud**. Chaque i-nœud contient les informations suivantes :

- identification du propriétaire et du groupe propriétaire du fichier ;
- type et droits du fichier ;
- taille du fichier en nombre de caractères (si possible) ;
- nombre de liens physiques du fichier ;
- trois dates (dernier accès au fichier, dernière modification du fichier, dernière modification du i-nœud) ;
- adresse des blocs utilisés sur le disque pour ce fichier (pour les fichiers disques) ;
- identification de la ressource associée (pour les fichiers spéciaux).

Rappel. Le type détermine les opérations autorisées sur un fichier.

Lecture d'un i-nœuds

La fonction

`stat(FICHIER)`

- `FICHIER` est un nom de fichier ou un handle ouvert ;
- en contexte scalaire, retourne vrai si l'appel a réussi ;
- en contexte de liste retourne les statistiques du fichier (voir page suivante).

Remarque. La fonction `stat` appliquée à un lien symbolique fournit les informations sur le fichier « réel » pointé par le lien. Pour obtenir les informations sur l'inode représentant le lien symbolique lui-même, utiliser `lstat`.

Informations fournies par `stat`

0	$dev	numéro du disque
1	$ino	numéro de l'inode
2	$mode	mode du fichier (type et droits)
3	$nlink	nombre de liens physiques sur le fichier
4	$uid	`uid` du propriétaire
5	$gid	`gid` du propriétaire
6	$rdev	identification du device (pour les fichiers spéciaux)
7	$size	taille en octets
8	$atime	date du dernier accès
9	$mtime	date de dernière modification
10	$ctime	date de dernière modification de l'inode
11	$blksize	taille de bloc I/O
12	$blocks	nombre de blocs alloués

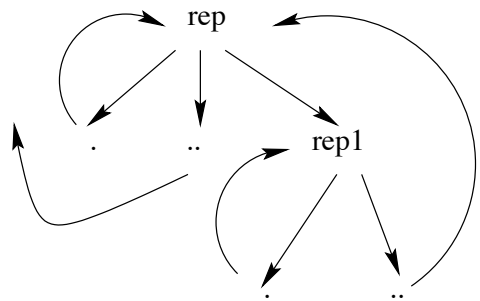
Exemple

```
bash$ cat essai.pl
#! /usr/bin/perl
 ($dev, $ino, $mode, $nlink,$uid,$gid,$rdev, $size, $atime,
     $mtime, $ctime, $blksize, $blocks) = stat ($ARGV[0]);

print "dev : $dev, ino : $ino, mode : $mode, size : $size\n";
printf "droits  %o\n", $mode & 0777;

bash$ essai.pl
dev : 775, ino : 32686, mode : 33188, size : 168
droits 644
```

Les liens physiques

Plusieurs entrées de répertoires peuvent représenter le même inode, on dit qu'il y a des **liens physiques** sur ce fichier.

- Le nombre de liens physiques du fichier peut être vu comme le nombre de « noms » du fichier, le nombre de noms (chaînes de caractères) associés au même i-nœud.
- Un répertoire n'est jamais complètement vide, il possède au moins deux liens :



- La commande `ln fichier lien` permet de créer des liens physiques. Le fichier spécifié ne peut être un répertoire et le répertoire dans lequel le lien est créé doit appartenir au même disque logique que le fichier.

Les liens symboliques

- Les liens symboliques sont des fichiers dont le contenu est interprété comme un nom de fichier.
- Les fonctions usuelles « suivent » les liens symboliques *i.e.* les rendent transparents à l'utilisateur.
- Création d'un lien symbolique :
  - par la commande `ln` avec l'option `-s` ;
  - consultation des caractéristiques : `lstat`

Exemple

```
bash$ ls -li
 121968 -rw-r--r-- 1 smith iut_f  4 Jan 19 18:51 essai.txt
bash$ ln -s essai.txt lien1
bash$ ls -li
 121968 -rw-r--r-- 1 smith iut_f  4 Jan 19 18:51 essai.txt
 121981 lrwxrwxrwx 1 smith iut_f  9 Jan 19 18:52 lien1
                                                 -> essai.txt
bash$ ls
essai.txt  lien1@
```

- Les numéro d'i-nœud sont différents.
- La commande `ls` identifie clairement les liens :
  - par des symboles `->` ou `@` après le nom de fichier ;
  - par le type de fichier `l`.

Lecture d'un lien symbolique

La fonction

```
readlink (NOM_DE_FICHIER)
```

- permet de lire le nom du fichier sur lequel pointe le lien ;
- `NOM_DE_FICHIER` est le nom d'un fichier :
  - si c'est un lien symbolique le nom du fichier associé est retourné,
  - si ce n'est pas un lien symbolique, la fonction retourne `undef`.

Exemple

```
bash$ cat essai.pl
#! /usr/bin/perl
$nom = readlink $ARGV[0];
print "Le fichier $ARGV[0] pointe sur $nom\n";
bash$ ls
essai.pl toto.txt
bash$ ln -s toto.txt toto
bash$ ln -s /bin/ls  monLS
bash$ ls -li
-rwxr-xr-x    1 ryl   neg    556 Dec 10 09:50 essai.pl*
lrwxrwxrwx    1 ryl   neg      7 Dec 10 09:54 monLS -> /bin/ls*
lrwxrwxrwx    1 ryl   neg      8 Dec 10 09:50 toto -> toto.txt
-rw-r--r--    1 ryl   neg     14 Dec  3 10:28 toto.txt
bash$ essai.pl toto
Le fichier toto pointe sur toto.txt
bash$ essai.pl monLS
Le fichier monLS pointe sur /bin/ls
```

Les entrées/sorties sur les répertoires

- Un répertoire est un fichier du système de fichiers ayant un type particulier.
- La structure des répertoires dépend du système de fichiers, de nombreuses structures différentes existent.
- Un répertoire peut être modifié par effet de bord de différentes fonctions : création/supressions de fichiers, liens, répertoires, etc...
- Un répertoire peut être « lu » en utilisant les commandes de base tels que `ls`.
- Un répertoire peut être « lu » en utilisant l'interface standard du système.

Lire un répertoire

- La fonction `opendir(DIRHANDLE, NOM)` permet à d'ouvrir un répertoire et retourne `true` si l'ouverture réussit.
- La fonction `readdir(DIRHANDLE)` permet de lire les entrées du répertoire :
  - en contexte scalaire, elle retourne l'entrée suivante du répertoire ou `undef` s'il n'y en a plus ;
  - en contexte de liste, elle retourne la liste de toutes les entrées restantes, une liste nulle s'il ne reste rien.
- La fonction `closedir(DIRHANDLE)` permet de fermer le handle.

Exemple

```
bash$ cat essai.pl
#! /usr/bin/perl
opendir (REP,$ARGV[0]) || die ("ouverture impossible");
while ($name = readdir (REP)) { print "$name\t";}
print "\n";
closedir (REP);
bash$ essai.pl tmp
.       ..      toto    titi    abc
bash$ cat essai.pl
#! /usr/bin/perl
opendir (REP,$ARGV[0]) || die ("ouverture impossible");
for $name (sort readdir (REP)) { print "$name\t";}
print "\n";
closedir (REP);
bash$ essai.pl tmp
.       ..      abc     titi    toto
```