INSTITUT Mines-Télécom

TELECOM SudParis

CSC 3102 – COURS
Introduction aux systèmes d'exploitation

Année 2022-2023

Coordinateurs :
Mathieu Bacou et Élisabeth Brunet

http://www-inf.telecom-sudparis.eu/COURS/CSC3102/Supports/

CSC 3102 – Introduction aux systèmes d'exploitation

Annexe shell

Commandes utiles utilisées dans le cours

`cat fich`	Affiche le contenu du fichier `fich` sur la sortie standard.
`echo args`	Affiche les arguments `args` sur la sortie standard.
`expr expression`	Calcule le résultat de l'expression arithmétique indiquée en argument et affiche le résultat sur la sortie standard. Attention, ne travaille qu'en arithmétique entière. Typiquement, vous pouvez incrémenter un compteur `i` avec l'expression suivante `i=$(expr $i + 1)`.
`grep [opt] motif fich`	Affiche sur la sortie standard les lignes de `fich` correspondant au motif `bash motif`. Si l'option `-v` est précisée, les lignes ne correspondant pas au motif `motif` sont affichées. Si aucun nom de fichier n'est donné, fait la recherche à partir de l'entrée standard. Les motifs utilisés par `grep` sont différents de ceux utilisés par `bash` et nous invitons les étudiants à consulter la page de `man`. Pour `grep`, le caractère `^` indique le début de ligne et le caractère `$` indique la fin de ligne.
`du [opt] fich`	Affiche la taille du fichier `fich` en nombre de blocs suivi du nom du fichier (un bloc correspond à 512 octets). L'option `-h` permet d'afficher la taille en utilisant des unités de mesure plus usitées par les humains.
`read var`	Copie la ligne de caractères lue sur l'entrée standard dans la variable `var`. Typiquement, pour lire une ligne d'un fichier, on utilise `read var < fichier`. Pour lire un fichier ligne à ligne, on utilise `while read var; do echo $var; done < fichier`.
`touch fich`	Crée le fichier `fich` si il n'existait pas.
`which cmd`	Indique le chemin absolue de la commande `cmd` après avoir recherché `cmd` dans le `PATH`.
`alias cmd='...'`	`cmd` devient un alias pour la commande `...`. `alias` sans autres options donnent la liste des alias utilisés par `bash`.
`unalias cmd`	Supprime l'alias associé à `cmd`.
`source script.sh` ou `. script.sh`	Exécute les instructions de script.sh dans le processus courant.

Motifs `bash`

`*`	N'importe quelle séquence de caractères.
`?`	N'importe quel caractère
`[wtz]`	Le caractère `w` ou le caractère `t` ou le caractère `z`.
`[d-r]`	Tous les caractères compris entre `d` et `r`.
`[[:lower:]]/[[:upper:]]`	Alphabet (caractères accentués compris) minuscule/majuscule.
`[[:digit:]]`	Chiffres décimaux.
`[[:alpha:]]`	Tous les caractères alphnumérique, c'est-à-dire toutes les lettres de l'alphabet (caractères accentués compris) et tous les chiffres décimaux.

Système de fichiers

`cd [chemin]`	Positionne le répertoire courant à "chemin" ou au répertoire d'accueil (HomeDirectory) si aucun paramètre n'est donné.

d'accueil (HomeDirectory) si aucun paramètre n'est donné.

`cp f1 f2`	Copie le fichier `f1` dans le fichier `f2`. Si `f2` existait déjà, son ancien contenu est perdu.
`ls [opt] [liste]`	Pour chaque élément `e` de `liste`, affiche le contenu du répertoire `e` si c'est un répertoire ou le nom du fichier `e` si c'est un fichier. Un certain nombre d'options peuvent être indiquées (opt). En voici quelques unes : • `-l`: affiche un certain nombre d'informations relatives aux éléments (type, taille, date...). • `-a`: visualise tous les éléments, y compris les fichiers et répertoires cachés (sous Unix, les éléments cachés commencent par le caractère "."). • `-d`: affiche le nom du répertoire et non son contenu.
`tree [chem]`	Affiche l'arborescence des fichiers accessibles à partir de `chem`. Si `chem` n'est pas spécifié, utilise le répertoire courant.
`mkdir [opt] rep`	Crée le répertoire `rep` dans le répertoire courant. Si l'option `-p` est passée en argument, les répertoires intermédiaires sont aussi créés.
`mv f1 f2`	Déplace le fichier `f1` dans le fichier `f2`. Cette commande peut aussi s'appliquer à des répertoires. Si `f2` existait déjà, son ancien contenu est perdu. Après l'opération, `f1` n'existe plus.
`pwd`	Permet de connaître le chemin absolu du répertoire courant (celui dans lequel on se trouve).
`rm f1`	Détruit le fichier f1.
`rmdir rep`	Détruit le répertoire `rep`, à condition que celui-ci soit vide.

Variables notables

`HOME`	Chemin vers le répertoire de connexion.
`PS1`	Prompt utilisé par `bash`. Par exemple `PS1="\w$ "` permet d'avoir un prompt qui indique le répertoire courant.
`PS2`	Prompt en cas de commandes sur plusieurs lignes.
`PATH`	Liste de chemins séparés par des « : » dans lesquels `bash` recherche les commandes.

Redirection des entrées-sorties

`exec n<> fichier`	Ouvre le flux de numéro `n` en écriture à partir de `fichier` (en écrasant son contenu). Par exemple : `exec 3<>mon-tube.pipe`. Dans la suite du script, on peut alors exécuter `echo coucou >&3` pour lire une donnée ou `read x <&3` pour écrire une donnée.
`exec n> fichier`	Ouvre le flux de numéro `n` en écriture à partir de `fichier` (en écrasant son contenu).
`exec n>> fichier`	Ouvre le flux de numéro `n` en écriture à partir de `fichier` (en ajout).
`exec n< fichier`	Ouvre le flux de numéro `n` en lecture à partir de `fichier`.
`cmd >&n`	Redirige la sortie standard de la commande `cmd` dans le flux numéro `n`.
`cmd <&n`	Redirige l'entrée standard de la commande `cmd` à partir du flux numéro `n`.
`cmd > fichier`	Ouvre un flux en écriture à partir de `fichier` (en écrasant son contenu), avant de rediriger la sortie de `cmd` dans le flux.
`cmd >> fichier`	Ouvre un flux en écriture à partir de `fichier` (en ajout à la fin), avant de rediriger la sortie de `cmd` dans le flux.
`cmd < fichier`	Ouvre un flux en lecture à partir de `fichier`, avant de rediriger l'entrée de `cmd` à partir du flux.

Processus

`;`	Permet de séparer deux commandes exécutées en séquentiel.
`&`	Exécute la commande à gauche en arrière plan.
`$$`	Identifiant (PID) du processus en cours d'exécution.
`$PPID`	Identifiant (PID) du processus père du processus en cours d'exécution.
`$!`	Identifiant (PID) du dernier processus lancé en arrière plan.
`$?`	Valeur de retour de la dernière commande exécutée en premier plan.
`exit [n]`	Interrompre l'exécution d'un script avec la valeur de retour `n`. Par convention, une commande renvoie une valeur différente de `0` en cas d'erreur et `0` sinon.
`export var`	Crée une copie de la variable `var` lorsque le processus engendre des enfants. La variable sera aussi exportée par les enfants.
`wait [n1 n2 ...]`	Bloque le processus courant tant que les fils dont les identifiants sont passés en paramètre ne sont pas terminés. Si aucun paramètre n'est donné, le processus attend la fin de tous ses fils.
`$(xyz)`	Le shell substitue à $(xyz) le texte produit sur la sortie standard par l'exécution de la commande xyz.
`ps [opt]`	Affiche la liste des processus en cours d'exécution. Un certain nombre d'options peuvent être indiquées (opt). En voici quelques unes : • `l`: affiche un certain nombre d'informations relatives aux éléments (PPID, état, priorité...). • `e`: affiche la liste de tous les processus (pas seulement ceux attachés au terminal courant). • `u`: affiche le nom de l'utilisateur qui a lancé chaque processus.
`pstree`	Affiche l'arborescence des processus.
`top`	Fournie une vue dynamique temps réel des processus en cours d'exécution.

Arguments

`$#`	Nombre de paramètres de la ligne de commande.
`$n` avec `n ∈ [0,9]`	Valeur du n^{e} paramètre. `$0` correspond au nom de la commande invoquée. `$n` correspond à la chaîne vide s'il y a moins de `n` paramètres.
`"$@"`	Donne l'ensemble des paramètres à partir de $1 en préservant les espaces à l'intérieur de chaque paramètre. Équivalent à écrire `"$1" "$2" ...`.
`shift [n]`	Décale d'un cran les paramètres du script (pas de valeur `n`) ou bien de "`n` crans" (si `n` est précisé). Par exemple, dans le cas où `n` n'est pas spécifié, `$2` devient `$1`, `$3` devient `$2` etc... ***Attention :*** *Le comportement n'est pas spécifié si le nombre de paramètres n'est pas suffisant.*

Chaînes et protections de caractères

`\x`	Un `\` protège n'importe quel caractère ayant normalement un sens particulier pour le shell. Par exemple, `\*` est interprété comme le caractère `*` et non comme le motif `*`. `\\` correspond au caractère `\`.
`"xyz"`	Protège la chaîne `xyz` du découpage sur les espaces lors de la séparation des arguments. Empêche l'interprétation des méta-caractères du langage, sauf `$`, `` ` `` et `\`. En particulier, les variables (`$mavar`) et les substitutions de commandes (`$(xyz)`) sont

($mavar) et les substitutions de commandes ($(xyz)) sont interprétées.

`'xyz'`	Protège la chaîne `xyz` du découpage, comme les guillemets, mais n'effectue aucune substitution. Aucun caractère autre que `'` n'a de sens particulier dans une chaîne entre apostrophes, pas même `\`.

Tests

`[ -z ch1 ]`	Renvoie vrai si la chaîne de caractères `ch1` est vide.
`[ -n ch1 ]`	Renvoie vrai si la chaîne de caractères `ch1` est non vide.
`[ ch1 = ch2 ]`	Renvoie vrai si les chaînes de caractères `ch1` et `ch2` sont égales.
`[ ch1 != ch2 ]`	Renvoie vrai si les chaînes de caractères `ch1` et `ch2` sont différentes.
`[ n1 -eq n2 ]`	Renvoie vrai si les nombres `n1` et `n2` sont égaux. Par exemple, `[ "01" -eq "1" ]` renvoie vrai puisque les nombres sont égaux, alors que `[ "01" = "1" ]` renvoie faux puisque les chaînes de caractères sont différentes.
`[ n1 -ne n2 ]`	Renvoie vrai si les nombres `n1` et `n2` sont différents.
`[ n1 -lt n2 ]`	Renvoie vrai si le nombre `n1` est strictement inférieur au nombre `n2`.
`[ n1 -le n2 ]`	Renvoie vrai si le nombre `n1` est inférieur ou égal au nombre `n2`.
`[ n1 -gt n2 ]`	Renvoie vrai si le nombre `n1` est strictement supérieur au nombre `n2`.
`[ n1 -ge n2 ]`	Renvoie vrai si le nombre `n1` est supérieur ou égal au nombre `n2`.
`[ -e chemin ]`	Renvoie vrai si le chemin `chemin` existe dans le système de fichier (fichier, répertoire, lien symbolique...).
`[ -f chemin ]`	Renvoie vrai si `chemin` correspond à un fichier.
`[ -d chemin ]`	Renvoie vrai si `chemin` correspond à un répertoire.
`[ -L chemin ]`	Renvoie vrai si `chemin` correspond à un lien symbolique.
`! test`	Renvoie vrai si `test` renvoie faux. Par exemple `! [ 42 -eq 67 ]` renvoie vrai.
`test1 && test2`	Renvoie vrai si `test1` et `test2` renvoie vrai.
`test1 || test2`	Renvoie vrai si `test1` ou `test2` renvoie vrai.

Structures de contrôle

`if test1; then`
`  corps1`
`elif test2; then`
`  corps2`
`else`
`  corps3`
`fi`
	Execute `corps1` si le test `test1` renvoie vrai. Dans le cas où `test1` renvoie faux, exécute `corps2` si `test2` renvoie vrai et `corps3` sinon. Les parties `elif` et `else` sont optionnelles.

`while test; do`
`  corps`
`done`
	Execute `corps` tant que le test `test` renvoie vrai.

`for var in liste; do`
`  corps`
`done`
	Pour `var` prenant successivement les différentes valeurs de la liste (une liste est une suite de mots séparés par des espaces), exécute `corps`.

`case $var in`
`  motif1) corps1;;`
`  motif2) corps2;;`
`  ...`
`  motifn) bodyn;;`
`esac`
	Trouve le premier `motif` correspondant à `$var` et exécute le `corps` associé.

`( expression )`	Permet de regrouper des commandes. Utile pour donner des priorités aux opérateurs `!` `||` et `&&` dans les tests. Par exemple `[`

priorités aux opérateurs `;`, `||` et `&&` dans les tests. Par exemple `[ 42 -eq 65 ] || ( [ 65 -eq 65 ] && [ 63 -eq 63 ] )` renvoie vrai. Peut aussi être utilisé pour rediriger les entrées/sorties de plusieurs commandes simultanément. Par exemple `( ls; cat /etc/passwd ) > fic`

Communication

`|`	Tube de communication anonyme entre deux processus : connecte la sortie standard de la commande de gauche à l'entrée standard de la commande de droite.
`kill -sig pid`	Envoie le signal `sig` (`sig` est un numéro ou un nom de signal, avec ou sans le `SIG` de début de chaîne) au processus d'identifiant `pid`.
`kill –l`	Donne la liste des signaux (nom et valeur numérique).
`mkfifo chemin`	Crée le tube nommé de nom `chemin`.
`trap [arg] sig`	Exécute `arg` lors du traitement du signal `sig` (`sig` est un numéro ou un nom de signal, avec ou sans le `SIG` de début de chaîne). Si `arg` est une chaîne vide, le signal est ignoré. Si `arg` est absent, le traitement par défaut du signal est rétabli.

INSTITUT Mines-Télécom

TELECOM SudParis

Introduction aux systèmes d'exploitation

CSC 3102

Introduction aux systèmes d'exploitation

Elisabeth Brunet et Amina Guermouche

Présentation du cours

- Contexte du cours :
  - Introduire notre objet d'étude : les systèmes d'exploitation
- Objectifs :
  - Comprendre ce qu'est un ordinateur
  - Comprendre ce que sont un logiciel et un programme
  - Comprendre ce qu'est un système d'exploitation
  - Comprendre ce qu'est un processus
- Notions abordées :
  - Ordinateur, mémoire, processeur, périphérique, système d'exploitation, processus, communication, programme, logiciel

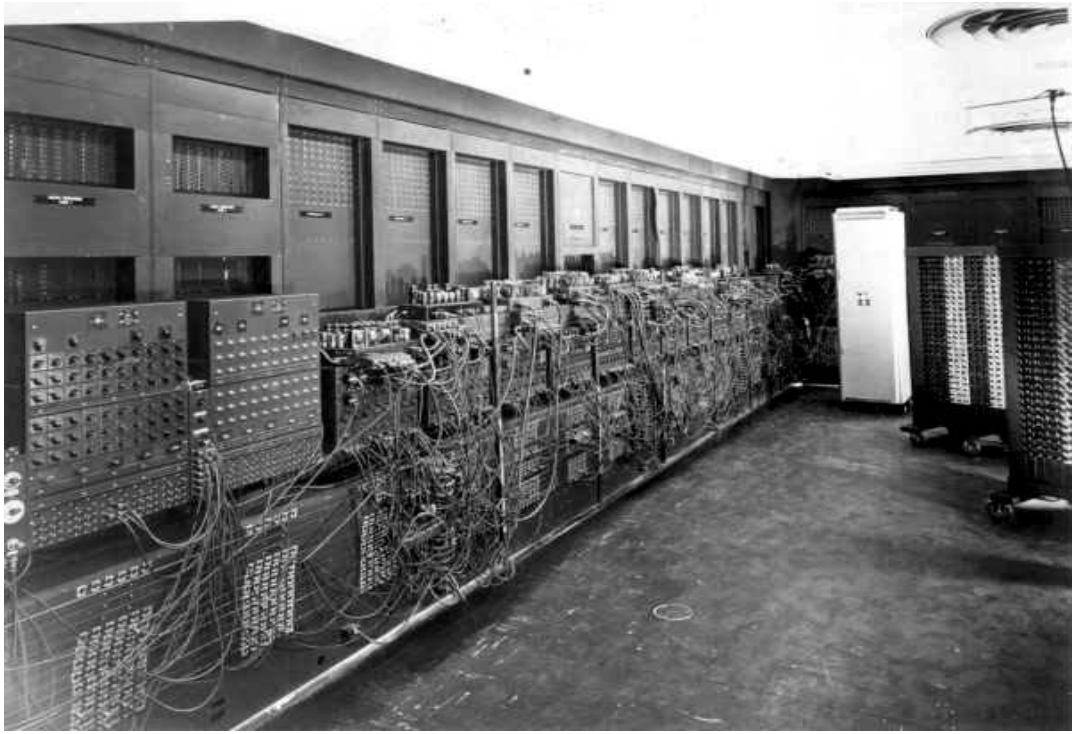
INSTITUT Mines-Télécom

TELECOM SudParis

I. Qu’est ce qu’un ordinateur ?

Définition d'un ordinateur

- Machine électronique capable d'exécuter des instructions effectuant des opérations sur des nombres



- 1946 : ENIAC (calculateur à tubes 30 tonnes, 72m² pour 330 mult/s)
*- Un processeur actuel (Intel i5) : 5.28*10⁶ opérations (à virgule flottante) par seconde*

Définition d'un ordinateur

- Machine électronique capable d'exécuter des instructions effectuant des opérations sur des nombres

- Janv 1948 : SSEC (premier ordinateur chez IBM) avec une capacité mémoire de 150 nombres
- Ordinateur récent avec 4 Go de mémoire : 10^6 nombres entiers

Définition d’un ordinateur

- Machine électronique capable d’exécuter des instructions effectuant des opérations sur des nombres

Schéma de haut niveau d'un ordinateur

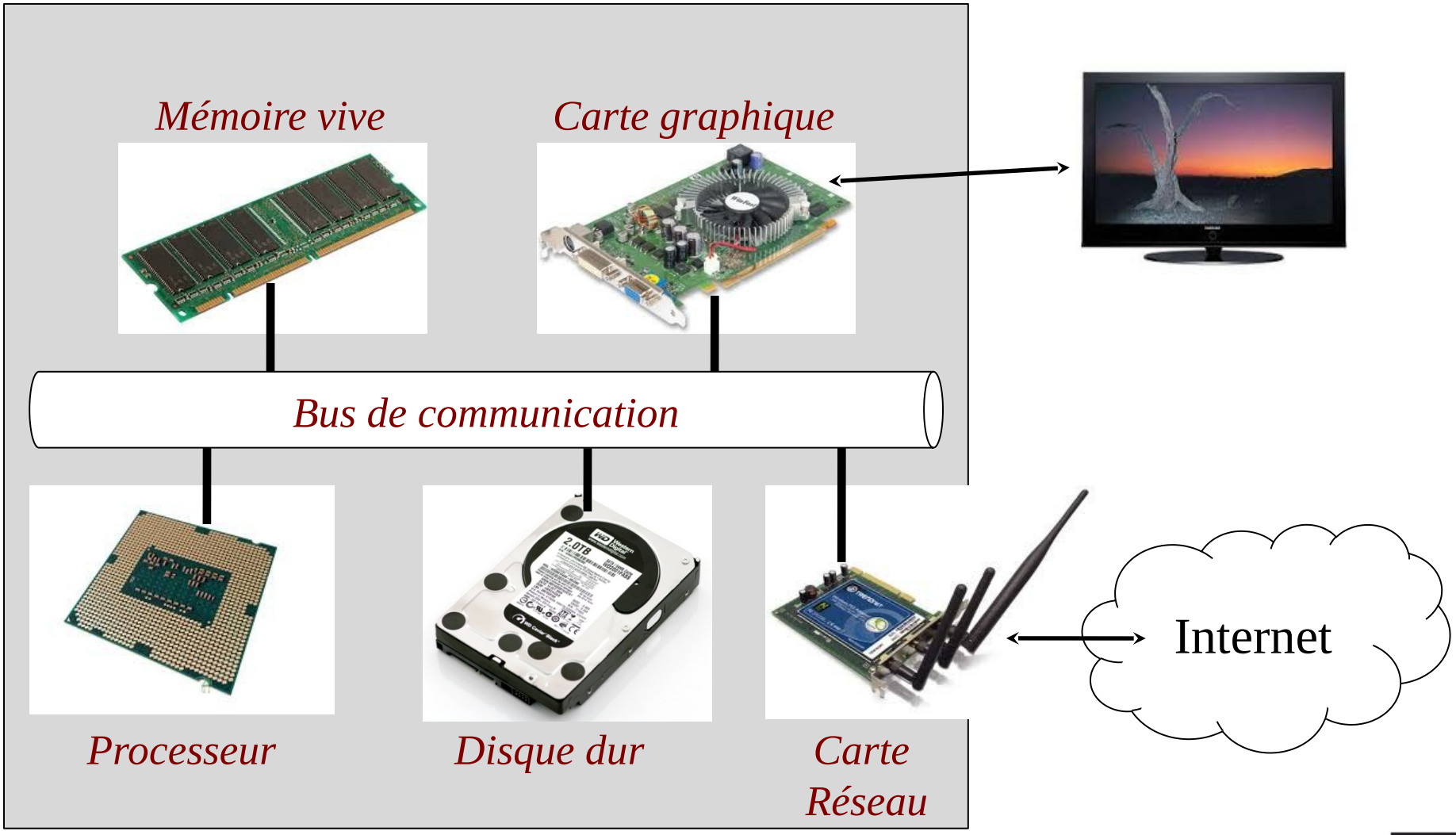


Schéma de haut niveau d'un ordinateur

- Processeur : unité capable d'effectuer des calculs
- Mémoire vive : matériel stockant des données directement accessibles par le processeur
  - Accès rapide, données perdues en cas de coupure électrique.
    - Par exemple : SDRAM (*Synchronous Dynamic Random Access Memory*)
- Périphériques : matériel fournissant ou stockant des données secondaires
  - Réseau, disque dur, souris, clavier, carte graphique, carte son…
- Bus de communication : bus interconnectant le processeur, la mémoire vive et les périphériques

Fonctionnement d'un processeur

- Un processeur exécute des instructions qui peuvent
  - Effectuer des calculs
  - Accéder à la mémoire
  - Accéder aux autres périphériques
  - Sélectionner l'instruction suivante à exécuter (saut)

- Le processeur identifie une instruction par un numéro (Par exemple : 1 = additionne, 2 = soustrait, etc.)

Fonctionnement d’un ordinateur

Et c’est tout!

Un ordinateur ne sait rien faire de mieux que des calculs

Ce qu'il faut retenir

- Une machine est constituée d'un processeur, d'une mémoire vive et de périphériques, le tout interconnecté par un bus
- Un processeur exécute de façon séquentielle des instructions qui se trouvent en mémoire
- Chaque instruction est identifiée par un numéro, elle peut
  - Effectuer une opération sur des variables internes (registres)
  - Lire ou écrire en mémoire ses registres
  - Accéder à un périphérique
  - Modifier la prochaine instruction à effectuer (saut)

INSTITUT Mines-Télécom

TELECOM SudParis

II. Logiciels et programmes

L’ordinateur vu par l’utilisateur

- L’utilisateur installe des **logiciels**

  Microsoft office, Chrome, Civilization V…

- Logiciel = ensemble de fichiers
  - Fichiers ressources : images, vidéos, musiques…
  - Fichiers programmes : fichier contenant des données et des instructions destinées à être exécutées par un ordinateur

- *In fine*, l’utilisateur lance l’exécution de **programmes**

  Excel, Word, Chrome, Civilization V, CivBuilder (permet de construire des cartes pour civilization V)…

INSTITUT Mines-Télécom

TELECOM SudParis

III. Processus et système

Du programme au processus

- Un **processus** est un programme en cours d'exécution
  - Contient bien sûr les opérations du programme
  - Mais aussi son état **à un instant donné**
    - Données en mémoire manipulées par le programme
    - État des périphériques (fichiers ouverts, connexions réseaux…)
    - ...

Gestion des processus

- Le **système d'exploitation** est un logiciel particulier qui gère les processus
  - Un noyau de système

    (Le noyau du système est le seul programme qu'on n'appelle pas processus quand il s'exécute)
  - Un ensemble de programmes utilitaires

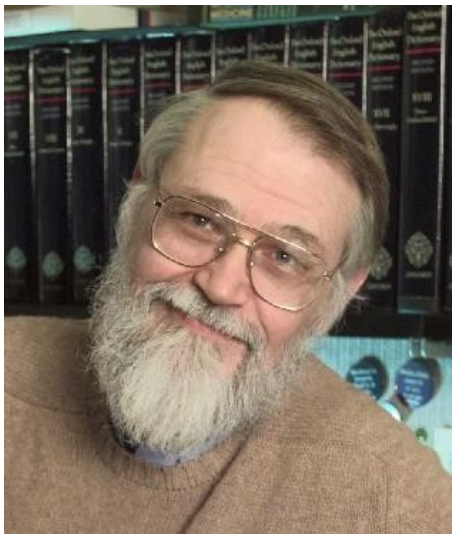
- Rôle du système d'exploitation
  - Démarrer des processus
  - Arrêter des processus
  - Offrir une vision de haut niveau du matériel aux processus
  - Offrir des mécanismes de communication inter-processus

Naissance des premiers systèmes UNIX

- 1969 : première version d'UNIX en assembleur
- 1970 : le nom UNIX est créé
- 1971 : invention du langage de programmation C pour réécrire UNIX dans un langage de haut niveau

Ken Thompson

Dennis Ritchie



Brian Kernighan

Utilité d'un système d'exploitation ?

- Est-ce bien utile ?
  - Ca doit bien l'être vu qu'il y en a sur nos ordinateurs et nos téléphones
  - Mais il y a des plates-formes qui n'en ont pas
    - dans ce cas, une seule application tourne
    - mais on veut tout faire en même temps (travailler, écouter de la musique, …)
- L'utilité du système d'exploitation est donc de permettre le multi-tâches
  - Vidéo sur les systèmes d'exploitation à regarder

Comment gérer le multi-tâches ?

- Comment le système d'exploitation reprend la main s'il y a un processus qui s'exécute ?
  a. Ordonnancement

- Comment les processus communiquent-ils ?
  a. Les tubes
  b. Les signaux

- Comment protège-t-il les données utilisées par un processus interrompu ?
  a. La concurrence

Objectif du module

- Étude des systèmes Unix par l'exemple
- À l'aide du langage `bash` (CI1)
  - Langage interprété par le programme `bash`
  - Langage spécialisé dans la gestion de processus
- Comprendre
  - La notion de fichier (CI2 à 4)
  - La notion de processus (CI5)
  - Quelques mécanismes de communication inter-processus (CI6 à 9)

INSTITUT Mines-Télécom

TELECOM SudParis

Le shell bash

CSC3102 - Introduction aux systèmes d'exploitation

Élisabeth Brunet et Gaël Thomas

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Le terminal

- Porte d'entrée d'un ordinateur

- Un terminal offre :
  - un canal pour entrer des données (clavier, souris, écran tactile…)
  - un canal pour afficher des données (écran, imprimante, haut-parleur…)

Le terminal

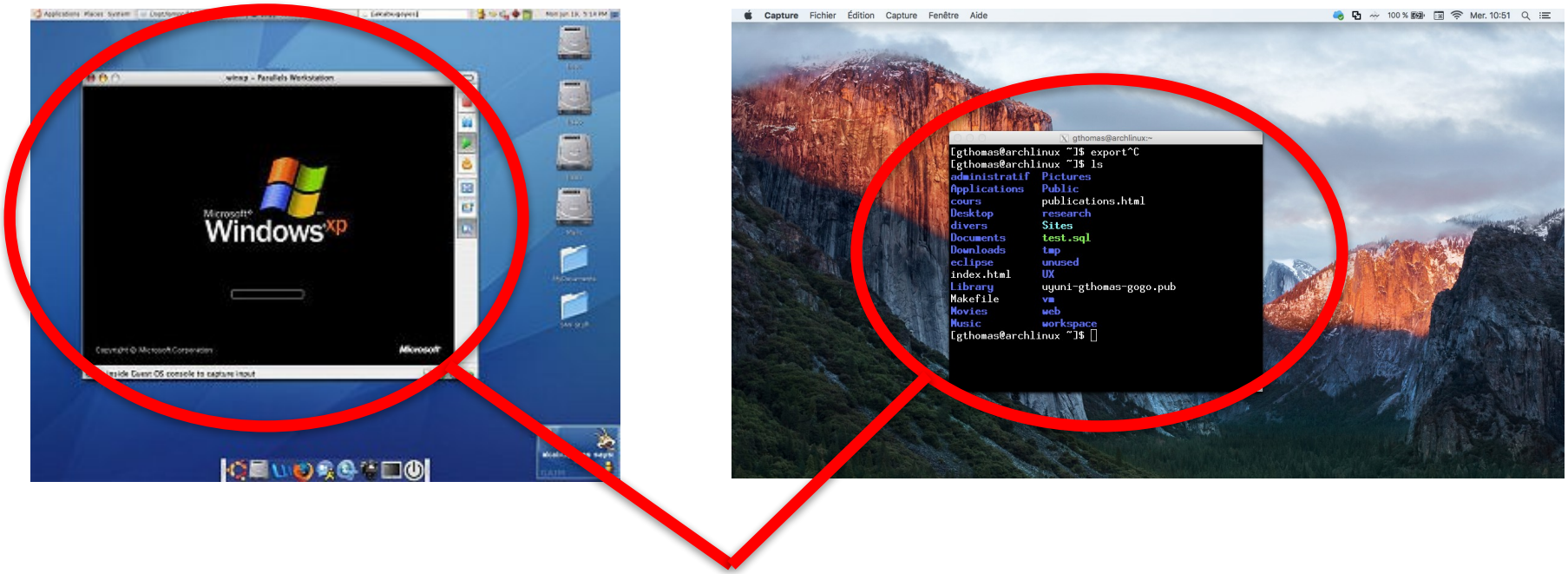
Un ordinateur n'a pas toujours un terminal intégré

Bien que ce soit souvent le cas

(smartphone, tablette, ordinateur portable…)

Un terminal peut être virtualisé

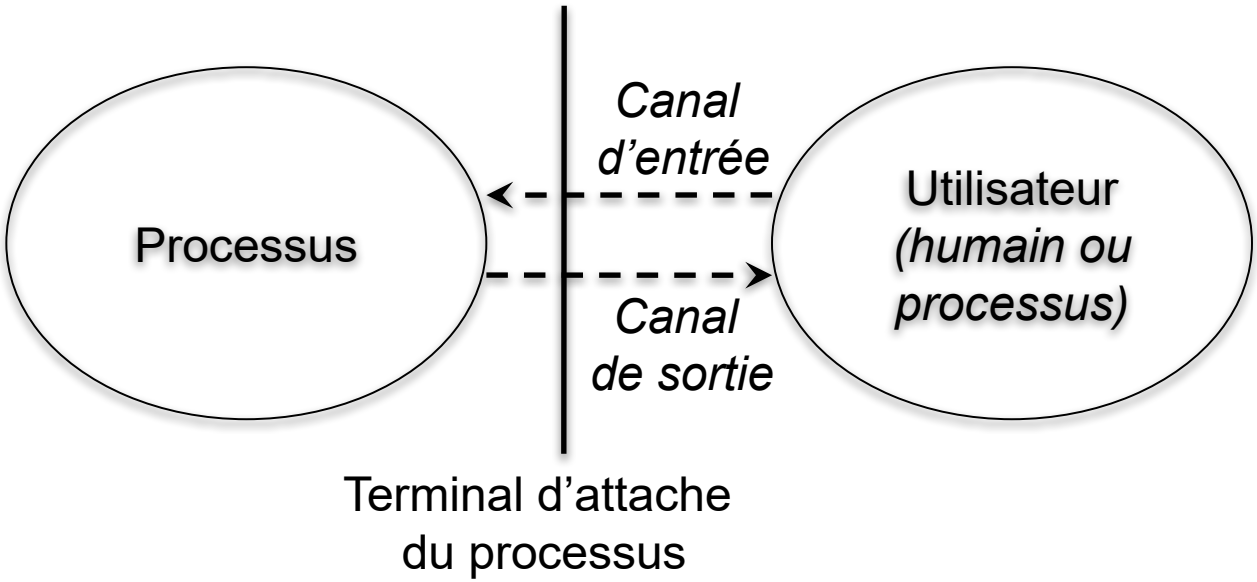
- Un terminal virtuel émule le comportement d'un terminal physique dans un autre terminal (virtuel ou physique)



Terminaux virtuels

Un processus communique avec l'utilisateur via un terminal

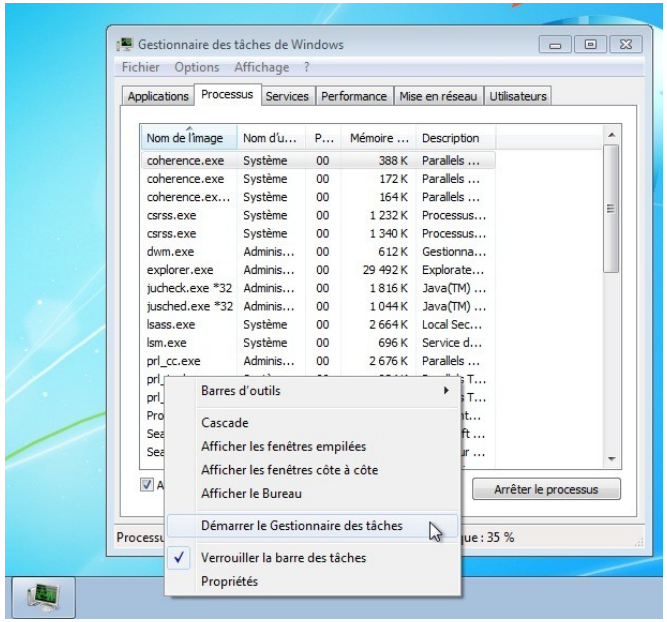
- On dit que le processus est attaché à un (et un seul) terminal



Remarque : lorsqu'un terminal est fermé,
tous les processus attachés au terminal sont détruits

Le shell

Le shell est un programme permettant d'interagir avec les services fournis par un système d'exploitation



Shell en mode graphique
(Bureau windows, X-windows…)

```
[gthomas@archlinux ~]$ ls
administratif  Pictures
Applications   Public
cours          publications.html
Desktop        research
divers         Sites
Documents      test.sql
Downloads      tmp
eclipse        unused
index.html     UX
Library        uyuni-gthomas-gogo.pub
Makefile       vm
Movies         web
Music          workspace
[gthomas@archlinux ~]$
```

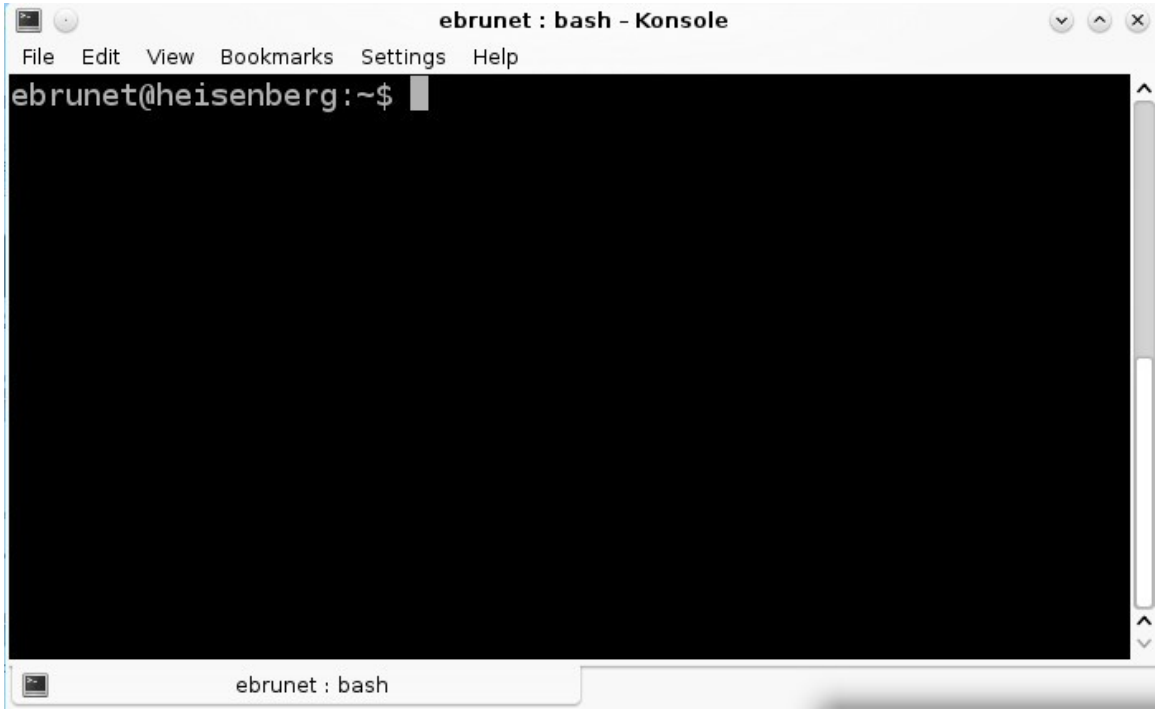
Shell en mode texte
(bash, tcsh, zsh, ksh, cmd.exe…)

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Le Bourne-Again Shell (`bash`)

- Dans ce cours, nous étudions le shell en **mode texte** `bash`
  - En mode texte car permet d'écrire des scripts !

- Attaché à un terminal virtuel en mode texte



Remarque importante

Dans la suite du cours, nous utiliserons souvent le terme « shell » pour désigner le « Bourne-Again shell »

Mais n'oubliez pas que `bash` n'est qu'un shell parmi de nombreux autres shells (`bash`, `tcsh`, `zsh`, `ksh`, `cmd.exe`...)

Bash

- Interpréteur de commandes
  - Lit des commandes (à partir du terminal ou d'un fichier)
  - Exécute les commandes
  - Écrit les résultats sur son terminal d'attache

- `Bash` définit un langage, appelé le langage bash
  - Structures de contrôle classiques
    (if, while, for, etc.)
  - Variables

- Accès rapide aux mécanismes offert par le noyau du système d'exploitation (tube, fichiers, redirections, …)

Un texte `bash`

- Un **texte** est formé de **mots `bash`**

- Un **mot `bash`** est
  - Formé de **caractères** séparés par des **délimiteurs** (délimiteurs : espace, tabulation, retour à la ligne)
    Exemple : `Coucou=42!*` est un unique mot
  - Exceptions :
    - `;` `&` `&&` `|` `||` `(` `)` `` ` `` sont des mots ne nécessitant pas de délimiteurs
    - Si une chaîne de caractères est entourée de `""` ou `''`, bash considère un unique mot

`bash` est sensible à la casse (c.-à-d., minuscule ≠ majuscule)

Un texte bash

- Un **texte** est formé de **mots**

```
Ici    nous    avons   5         mots

" En bash, ceci est un unique "mot" y compris mot milieu"

Voici, trois, mots

" zip "@é§èçà°-_¨^$%ù£,.:+=' est un autre unique mot'

Nous|avons;NEUF&&mots&ici
```

Un texte `bash`

- Un **texte** est formé de **mots**

Attention :

Ce n'est pas parce qu'on écrit des mots que ces mots ont un sens pour bash

Exemple : `echo yop!3:bip` est constitué de deux mots mais n'est pas compréhensible par `bash`

Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2…` **`cmd`** `arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$
```

Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo Salut tout le monde
```

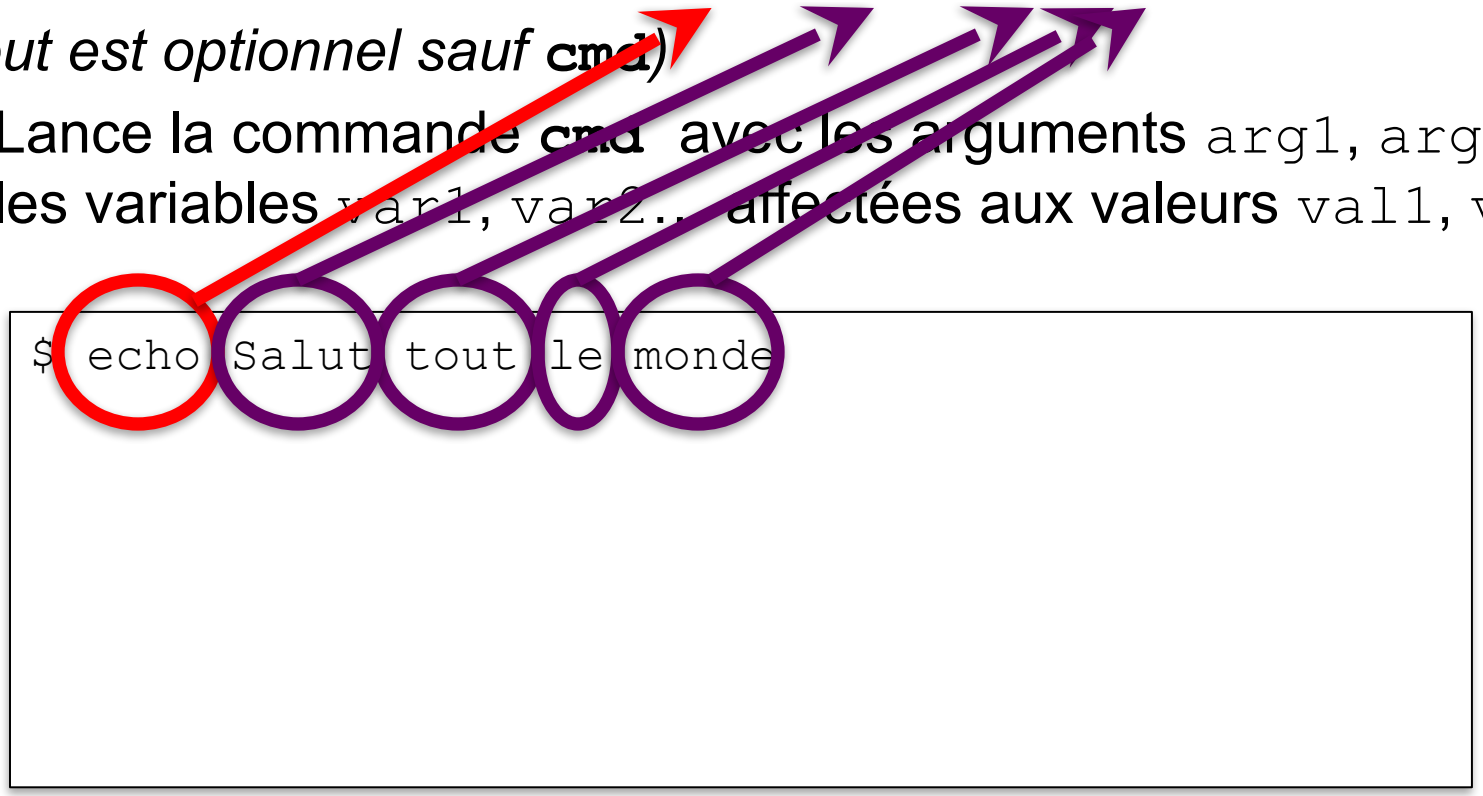
Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…



Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo Salut tout le monde
Salut tout le monde
```

Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo "Salut   tout le   monde"
```

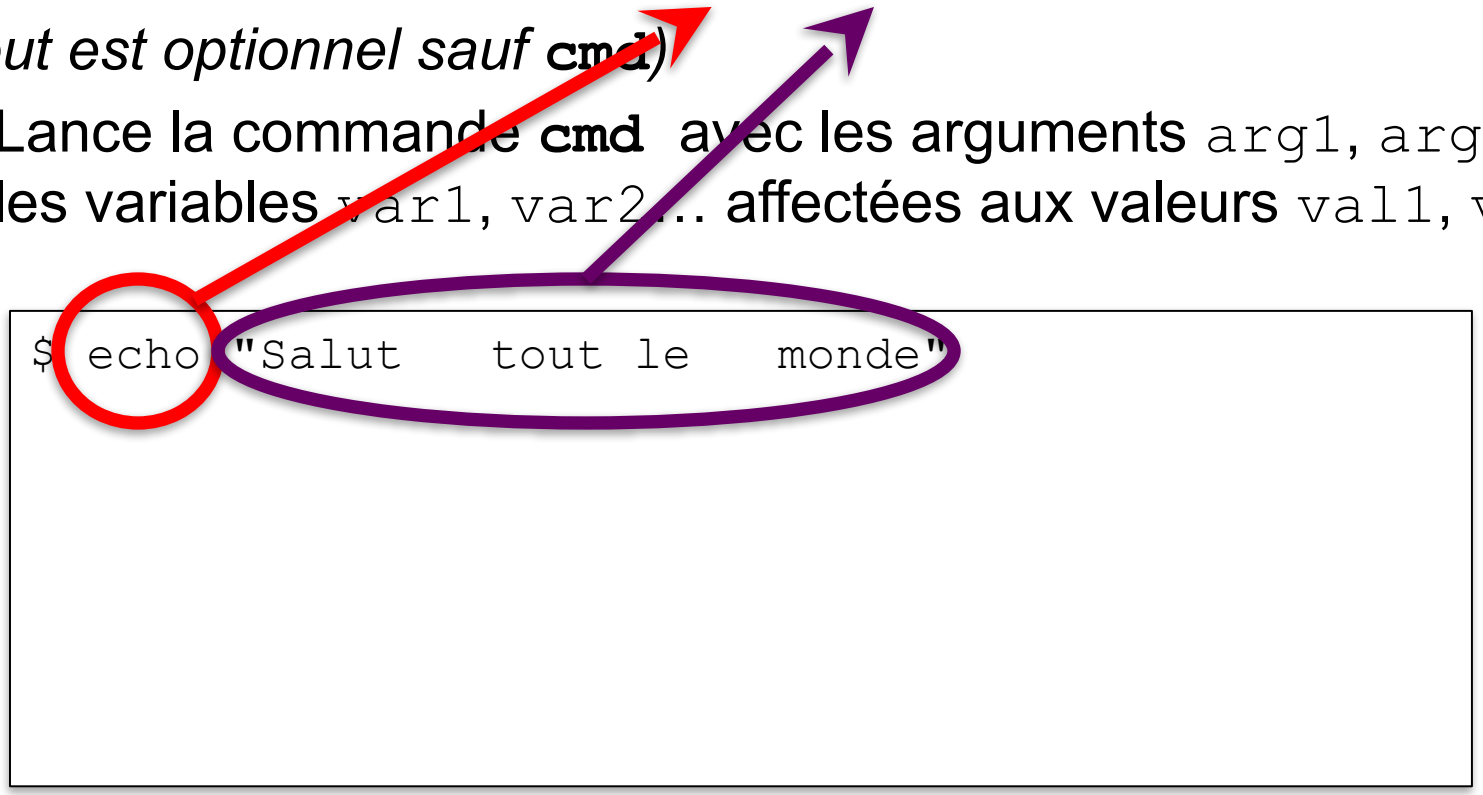
Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…



Invocation d'une commande `bash`

- Invocation d'une commande :

  `var1=val1 var2=val2… cmd arg1 arg2…`

  (tout est optionnel sauf **`cmd`***)*

  - Lance la commande **`cmd`** avec les arguments `arg1`, `arg2`… et les variables `var1`, `var2`… affectées aux valeurs `val1`, `val2`…

```
$ echo "Salut   tout le   monde"
Salut   tout le   monde
```

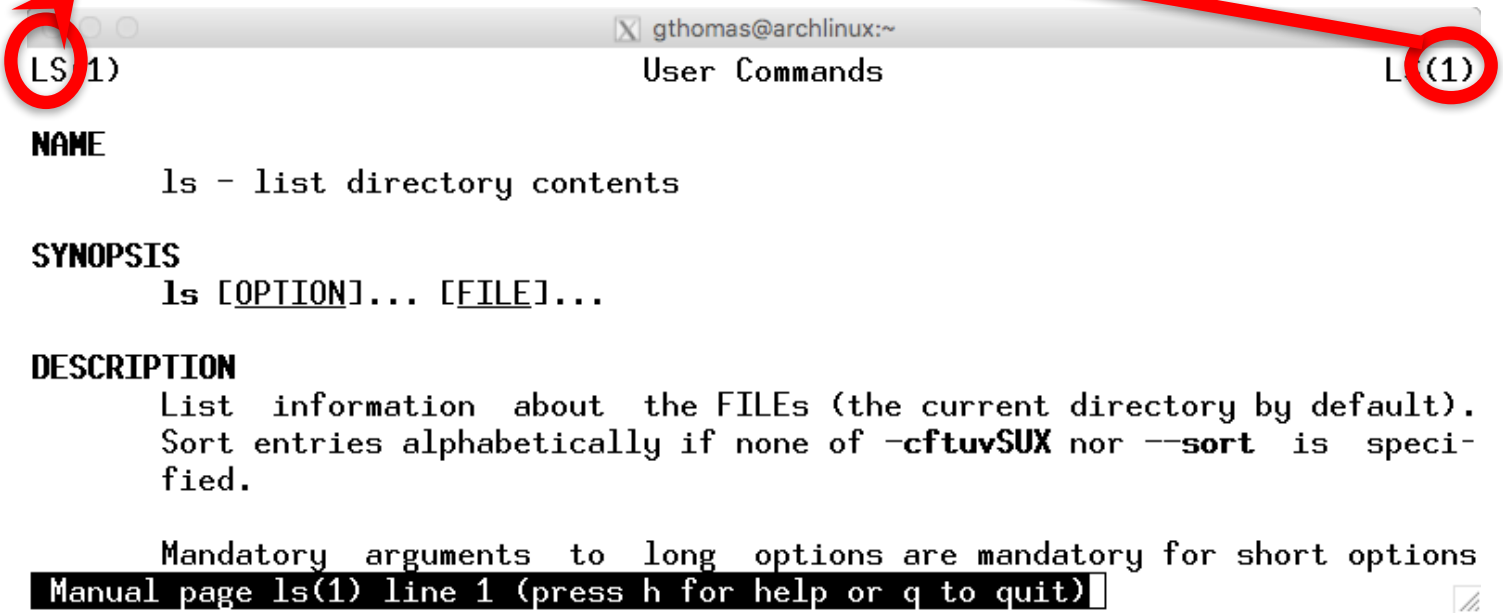
La première commande à connaître

- `man 1 cmd`
  - `man` pour manuel : donne de l'aide
  - `1` (optionnel) indique la section d'aide de la commande
    - `1` : commandes
  - `cmd` est la commande dont on veut consulter le manuel

```
$ man ls
```

La première commande à connaître

- `man 1 cmd`
  - `man` pour manuel : donne de l'aide
  - `1` (optionnel) indique la section d'aide de la commande
    - `1` : commandes
  - `cmd` est la commande dont on veut consulter le manuel



Caractères spéciaux de `bash`

- Caractères spéciaux
  - `\ ' ` " > < $ # * ~ ? ; ( ) { }`
    (`'` est appelé quote ou apostrophe
    alors que `` ` `` est appelé antiquote ou accent grave)
  - Explication de chacun donnée dans la suite du cours

- Désactiver l'interprétation des caractères spéciaux
  - `\` désactive l'interprétation spéciale du caractère suivant
  - `'…'` ⇒ désactive l'interprétation dans toute la chaîne
  - `"…"` ⇒ seuls sont interprétés les caractères $ \ ` (accent grave)

Script `bash`

- Programme `bash` = texte `bash` dans un fichier texte
  - Interprétable par `bash` au lancement par l'utilisateur
  - Modifiable par un éditeur de texte (p. ex. `emacs`, `vi`, mais pas `word` !)
  - Un programme bash doit être rendu exécutable avec :

    `chmod u+x mon_script.sh`
    (notion vue dans le CI2 sur le système de fichiers)
  - Par convention, les noms de script sont suffixés par l'extension « .sh »
    - p. ex., `mon_script.sh`

- Invocation du script nommé `mon_script.sh` avec
  - `./mon_script.sh`
  - Avec ses arguments :

    `./mon_script.sh  arg1 arg2`

`./` indique que le script se trouve dans le répertoire courant (notion vue dans le CI2)

Structure d'un script bash

- Première ligne : `#! /bin/bash`
  - `#!` : indique au système que ce fichier est un ensemble de commandes à exécuter par l'interpréteur dont le chemin suit
    - par exemple : `/bin/sh`, `/usr/bin/perl`, `/bin/awk`, etc.
  - `/bin/bash` lance `bash`

- Puis séquence structurée de commandes shell

```
#! /bin/bash

commande1
commande2
...
```

mon_script.sh

- Sortie implicite du script à la fin du fichier
  - Sortie explicite avec la commande `exit`

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Variables `bash`

- Déclaration/affectation avec = (exemple `ma_var=valeur`)
- Consultation en préfixant du caractère `$` (exemple `$ma_var`)
- Saisie interactive : `read var1 var2 ... varn`
  - Lecture d'une ligne saisie par l'utilisateur (jusqu'au retour chariot)
  - Le premier mot va dans `var1`
  - Le second dans `var2`
  - Tous les mots restants vont dans `varn`

Variables `bash`

- Déclaration/aff
- Co
- Sa
  -
  -
  - L
  - Tous les mots restants vont dans `varn`

Attention :

Pas de blanc dans `ma_var=valeur`
Pas de blanc dans `$ma_var`

(dans les deux cas, `bash` interprète de façon spéciale un unique mot)

r chariot)

Variables `bash` - exemple

```
$
```

Variables `bash` - exemple

```
$ a=42
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$
```

Saisi par l'utilisateur

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$ read x y
Ceci est une phrase
$
```

Saisi par l'utilisateur

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$ read x y
Ceci est une phrase
$ echo $x
Ceci
$
```

Variables `bash` - exemple

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$ read x y
Ceci est une phrase
$ echo $x
Ceci
$ echo $y
est une phrase
```

Premier mot

Tous les mots qui suivent

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Schéma algorithmique séquentiel

- Suite de commandes les unes après les autres
  - Sur des lignes séparées
  - Sur une même ligne en utilisant le caractère point virgule (`;`) pour séparateur

Schéma alternatif (`if`)

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

Conditions de test

- Tests sur des valeurs numériques
  - `[ n1 -eq n2 ]` : vrai si `n1` est égal à `n2`
  - `[ n1 -ne n2 ]` : vrai si `n1` est différent de `n2`
  - `[ n1 -gt n2 ]` : vrai si `n1` supérieur strictement à `n2`
  - `[ n1 -ge n2 ]` : vrai si `n1` supérieur ou égal à `n2`
  - `[ n1 -lt n2 ]` : vrai si `n1` inférieur strictement à `n2`
  - `[ n1 -le n2 ]` : vrai si `n1` est inférieur ou égal à `n2`
- Tests sur des chaînes de caractères
  - `[ mot1 = mot2 ]` : vrai si `mot1` est égale à `mot2`
  - `[ mot1 != mot2 ]`: vrai si `mot1` n'est pas égale à `mot2`
  - `[ -z mot ]` : vrai si `mot` est le mot vide
  - `[ -n mot ]` : vrai si `mot` n'est pas le mot vide

Conditions de test

- Tests sur des valeurs numériques
  - `[ n1 -eq n2 ]` …

Attention :

Les blancs sont essentiels !

- Tes…
  - … à `mot2`
  - `[` … `!= mot2 ]` : vrai si `mot1` n'est pas égale à `mot2`
  - `[ -z mot ]` : vrai si `mot` est le mot vide
  - `[ -n mot ]` : vrai si `mot` n'est pas le mot vide

Remarque sur les conditions

- `[ cond ]` est un raccourci pour la commande **`test`** `cond`

- **`test`** est une commande renvoyant vrai (valeur 0) ou faux (valeur différente de 0) en fonction de l'expression qui la suit

```
if [ $x -eq 42 ]; then
   echo coucou
fi
```

Équivaut à

```
if test $x -eq 42; then
   echo coucou
fi
```

Schéma alternatif (`if`)

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

```
x=1
y=2
if  [ $x -eq $y ]; then
   echo "$x = $y"
elif [ $x -ge $y ]; then
   echo "$x >= $y"
else
  echo "$x < $y"
fi
```

Schéma alternatif (case)

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

```
case mot in
   motif1)
    …;;
   motif2)
    …;;
   *)
     …;;
esac
```

- Schéma alternatif multiple
  - Si `mot` vaut `motif1` …
    Sinon si `mot` vaut `motif2` …
    Sinon …
  - Motif : chaîne de caractères pouvant utiliser des méta-caractères (voir CI3)
  - *) correspond au cas par défaut

Schéma alternatif (case)

- Schéma alternatif simple
  - Si alors … sinon (si alors … sinon …)
  - `elif` et `else` sont optionnels

```
if cond; then
   cmds
elif cond; then
   cmds
else
  cmds
fi
```

- Schéma alternatif multiple
  - Si `mot` vaut `motif1` …
    Sinon si `mot` vaut `motif2` …
    Sinon …
  - Motif : chaîne de caractères pouvant utiliser des méta-caractères (voir CI3)
  - *) correspond au cas par défaut

```
case mot in
   motif1)
    …;;
   motif2)
    …;;
   *)
     …;;
esac
```

```
res="fr"
case $res in
   "fr")
     echo "Bonjour";;
   "it")
     echo "Ciao";;
   *)
     echo "Hello";;
esac
```

Schémas itératifs

```
while cond; do
    cmds
done
```

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle

Schémas itératifs

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle

```
while cond; do
    cmds
done
```

```
x=10
while [ $x -ge 0 ]; do
   read x
   echo $x
done
```

Schémas itératifs

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle
  - **for**
    - Pour chaque … dans … faire …
    - `var` correspond à la variable d'itération
    - `liste` : ensemble sur lequel `var` itère

```
while cond; do
    cmds
done
```

```
x=10
while [ $x -ge 0 ]; do
    read x
    echo $x
done
```

```
for var in liste; do
    cmds
done
```

Schémas itératifs

- Boucles
  - **while**
    - Tant que … faire …
    - Mot clé `break` pour sortir de la boucle
  - **for**
    - Pour chaque … dans … faire …
    - `var` correspond à la variable d'itération
    - `liste` : ensemble sur lequel `var` itère

```
while cond; do
    cmds
done
```

```
x=10
while [ $x -ge 0 ]; do
    read x
    echo $x
done
```

```
for var in liste; do
    cmds
done
```

```
for var in 1 2 3 4; do
    echo $var
done
```

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Arguments d'une commande

- `mon_script.sh arg1 arg2 arg3 arg4 …`
  ⇒ chaque mot est stocké dans une variable numérotée

mon_script.sh	arg1	arg2	arg3	arg4	…
"$0"	"$1"	"$2"	"$3"	"$4"	…

- `"$0"` : toujours le nom de la commande
- `"$1" … "$9"` : les paramètres de la commande
- `$#` : nombre de paramètres de la commande
- `"$@"` : liste des paramètres : `"arg1" "arg2" "arg3" "arg4" …`
- `shift` : décale d'un cran la liste des paramètres

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$
```

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$./mon_echo.sh
$
```

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$./mon_echo.sh
$./mon_echo.sh toto titi
toto
titi
$
```

Arguments d'une commande

```
#! /bin/bash
for i in "$@"; do
  echo $i
done
```

mon_echo.sh

```
$./mon_echo.sh
$./mon_echo.sh toto titi
toto
titi
$./mon_echo "fin de" la demo
fin de
la
demo
$
```

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

```
$ date
lundi 27 juillet 2015, 12:47:06 (UTC+0200)
$
```

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

```
$ date
lundi 27 juillet 2015, 12:47:06 (UTC+0200)
$ echo "Nous sommes le $(date). "
Nous sommes le lundi 27 juillet 2015, 12:47:06
(UTC+0200).
$
```

Rappel : avec "…", seuls sont interprétés les caractères $ \ `

Imbrication de commandes

- Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères
  - `$(cmd)`
  - Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

```
$ date
lundi 27 juillet 2015, 12:47:06 (UTC+0200)
$ echo "Nous sommes le $(date). "
Nous sommes le lundi 27 juillet 2015, 12:47:06
(UTC+0200).
$ echo "Nous sommes le $date."
Nous sommes le .
$
```

Attention, récupère la variable date et non le résultat de la commande date

- Terminal et shell
- Le langage bash
- Les variables
- Les structures algorithmiques
- Arguments d'une commande
- Commandes imbriquées

Conclusion

- Concepts clés
  - Terminal, shell
  - Interpréteur de commande `bash`
    - Commandes, langage `bash`
  - Documentation
  - Caractères spéciaux de `bash`
  - Script `bash`
- Commandes clés
  - `man, bash, echo, read`
- Commandes à connaître
  - `date`

En route pour le TP !

INSTITUT Mines-Télécom

TELECOM SudParis

Système de Fichiers

CSC3102 – Introduction aux systèmes d'exploitation

Élisabeth Brunet & Gaël Thomas

Système de Fichiers

- Besoin de mémoriser des informations
  - Photos, PDF, données brutes, exécutables d'applications, le système d'exploitation lui-même, etc.
- Organisation du stockage sur mémoire de masse
  - Localisation abstraite grâce à un chemin dans une arborescence
  - Unité de base = fichier
- Exemples de types de systèmes de fichiers
  - NTFS pour Windows, ext2, ext3, ext4 pour Linux, HFSX pour Mac-OS
  - FAT pour les clés USB, ISO pour les CD
  - … et des myriades d'autres types de systèmes de fichiers

- Le système de fichiers vu par un processus
- Le système de fichiers sur disque
- Les commandes utilisateurs
- Les droits d'accès

Qu'est-ce qu'un fichier

- Un fichier est la réunion de
  - Un contenu, c'est-à-dire un ensemble ordonné d'octets
  - Un propriétaire
  - Des horloges scalaires (création, dernier accès, dernière modif)
  - Des droits d'accès (en lecture, en écriture, en exécution)

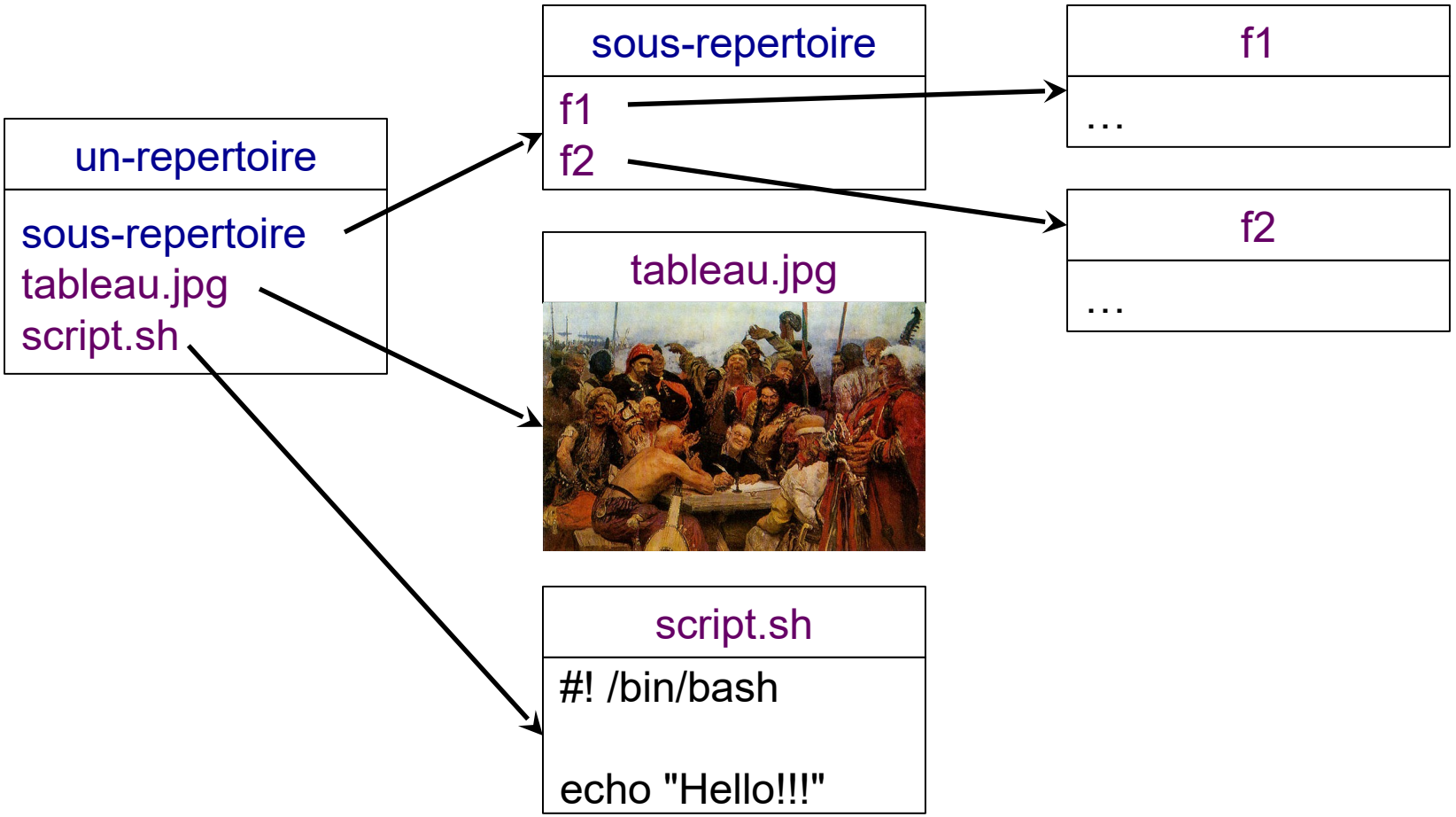
- Attention : c'est inattendu, mais un fichier est indépendant de son nom (c.-à-d., le nom ne fait pas parti du fichier et un fichier peut avoir plusieurs noms)

On stocke de nombreux fichiers

- Facilement plusieurs centaines de milliers de fichiers dans un ordinateur
  - Plusieurs milliers gérés/utilisés directement par l'utilisateur
  - Plusieurs centaines de milliers pour le système et les applications
- Problème : comment retrouver facilement un fichier parmi des centaines de milliers ?
- Solution : en rangeant les fichiers dans des répertoires (aussi appelés dossiers)

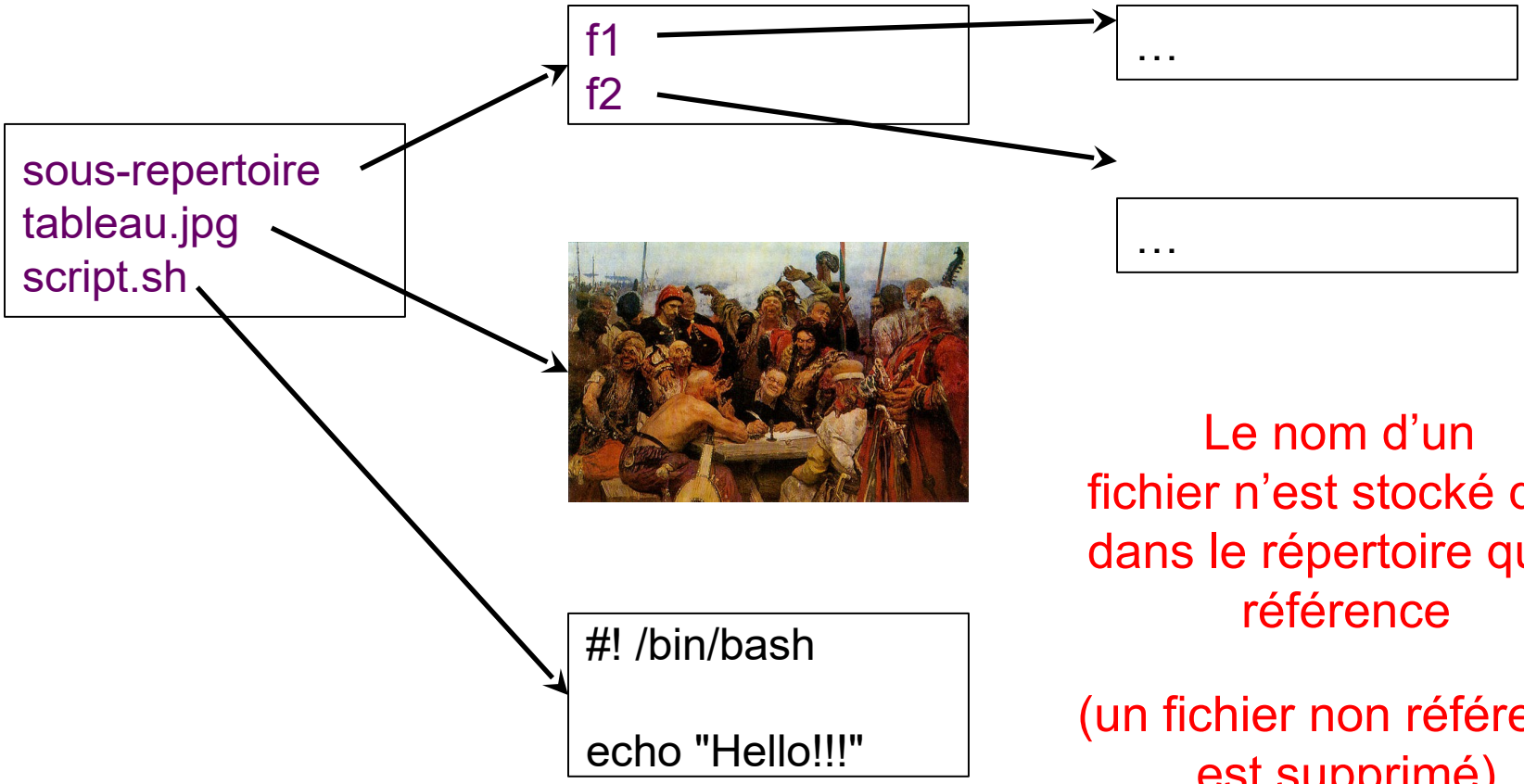
Organisation en répertoires

- Répertoire = fichier spécial qui associe des noms à des fichiers



Organisation en répertoires

- Répertoire = fichier spécial qui associe des noms à des fichiers

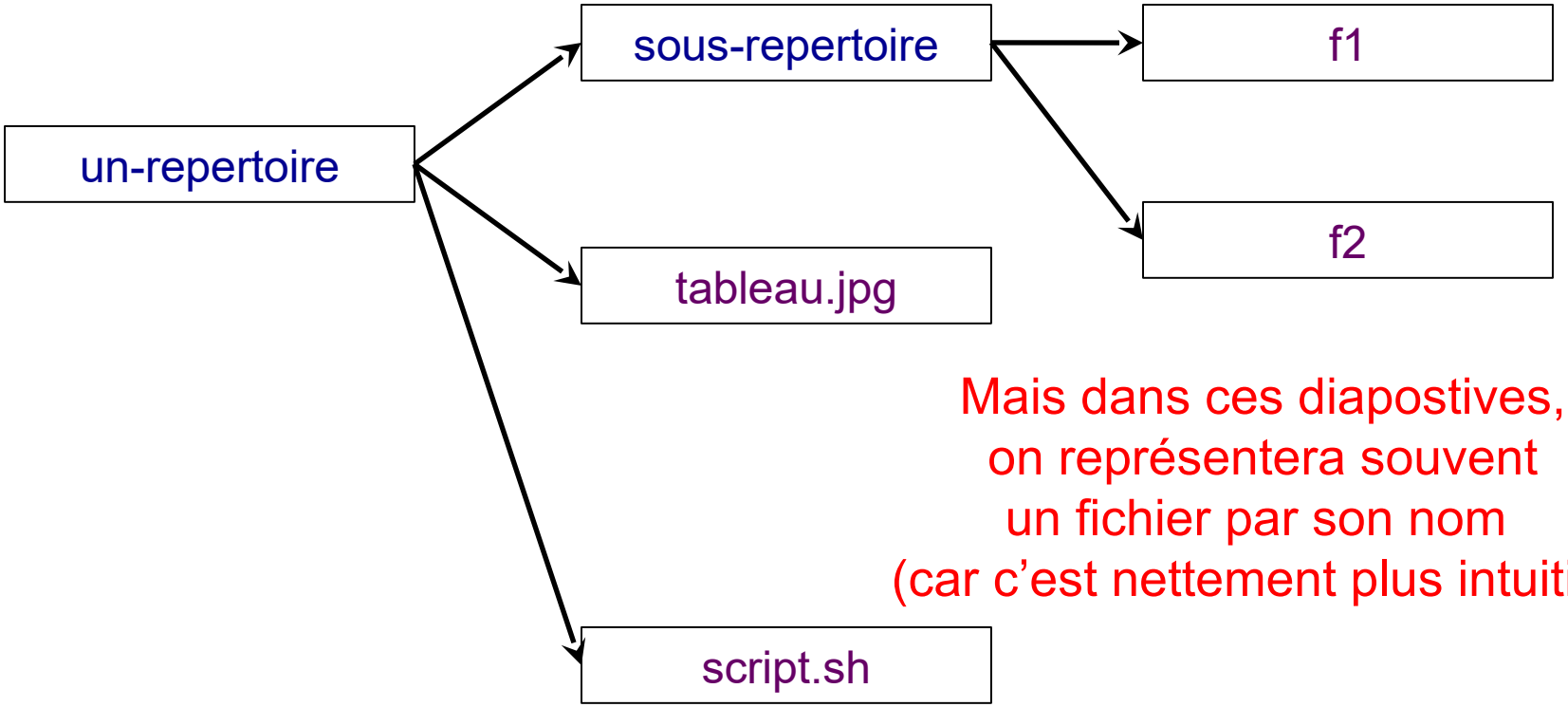


Le nom d'un fichier n'est stocké que dans le répertoire qui le référence

(un fichier non référencé est supprimé)

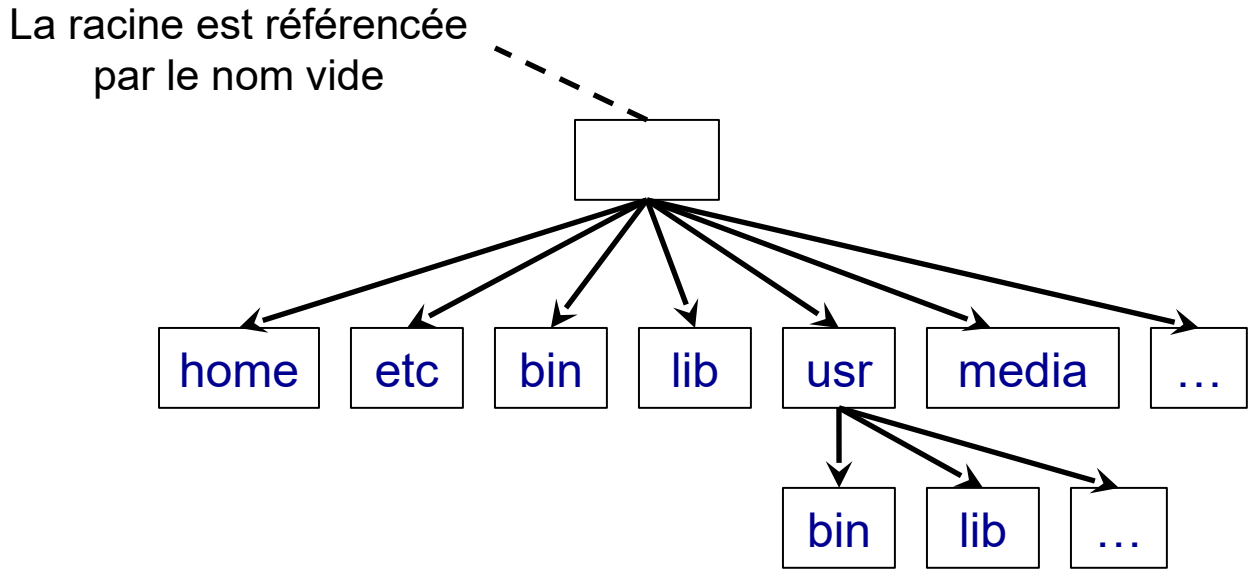
Organisation en répertoires

- Répertoire = fichier spécial qui associe des noms à des fichiers



Mais dans ces diapostives,
on représentera souvent
un fichier par son nom
(car c'est nettement plus intuitif !)

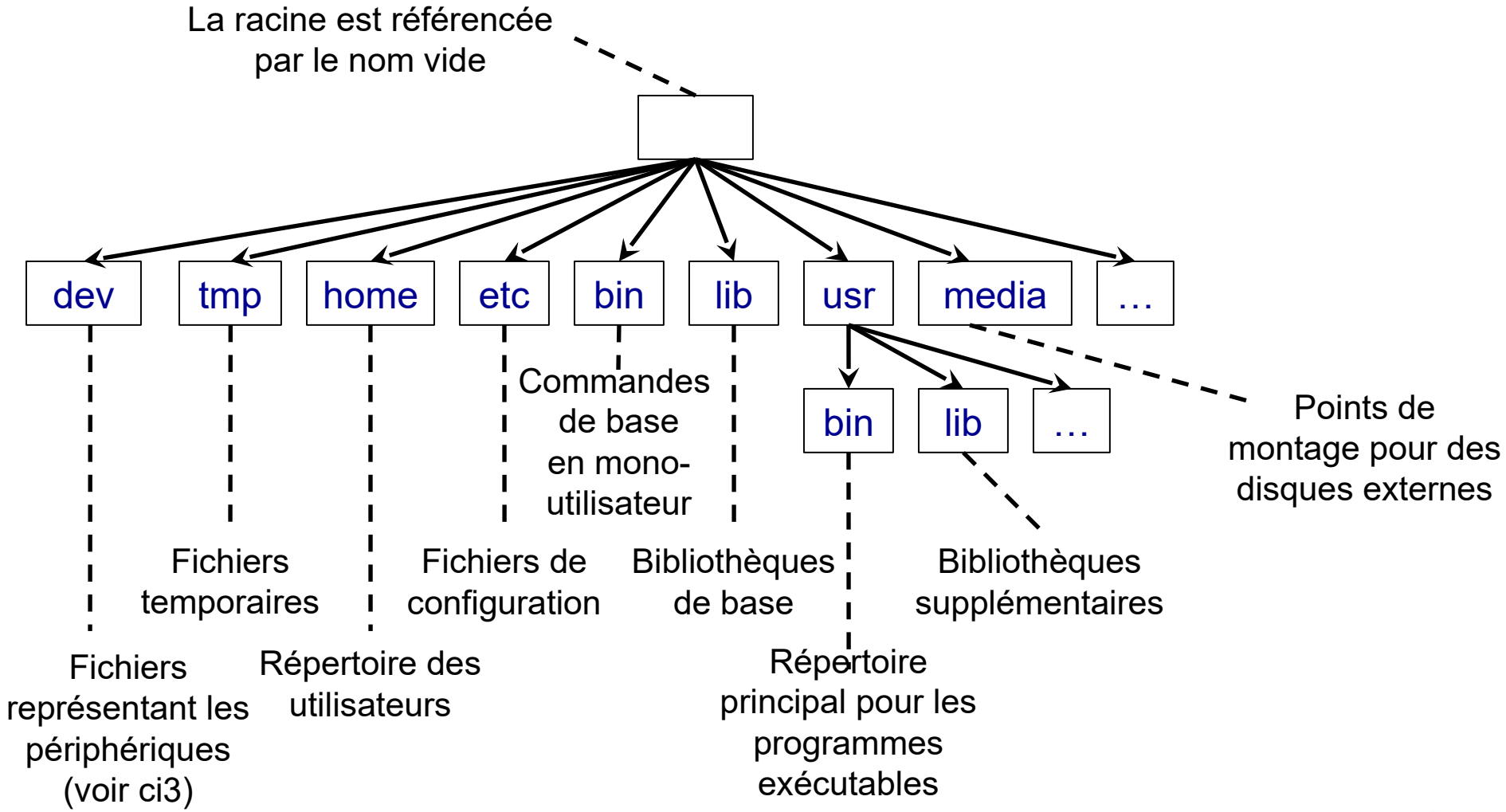
Arborescence standard des systèmes d'exploitation UNIX



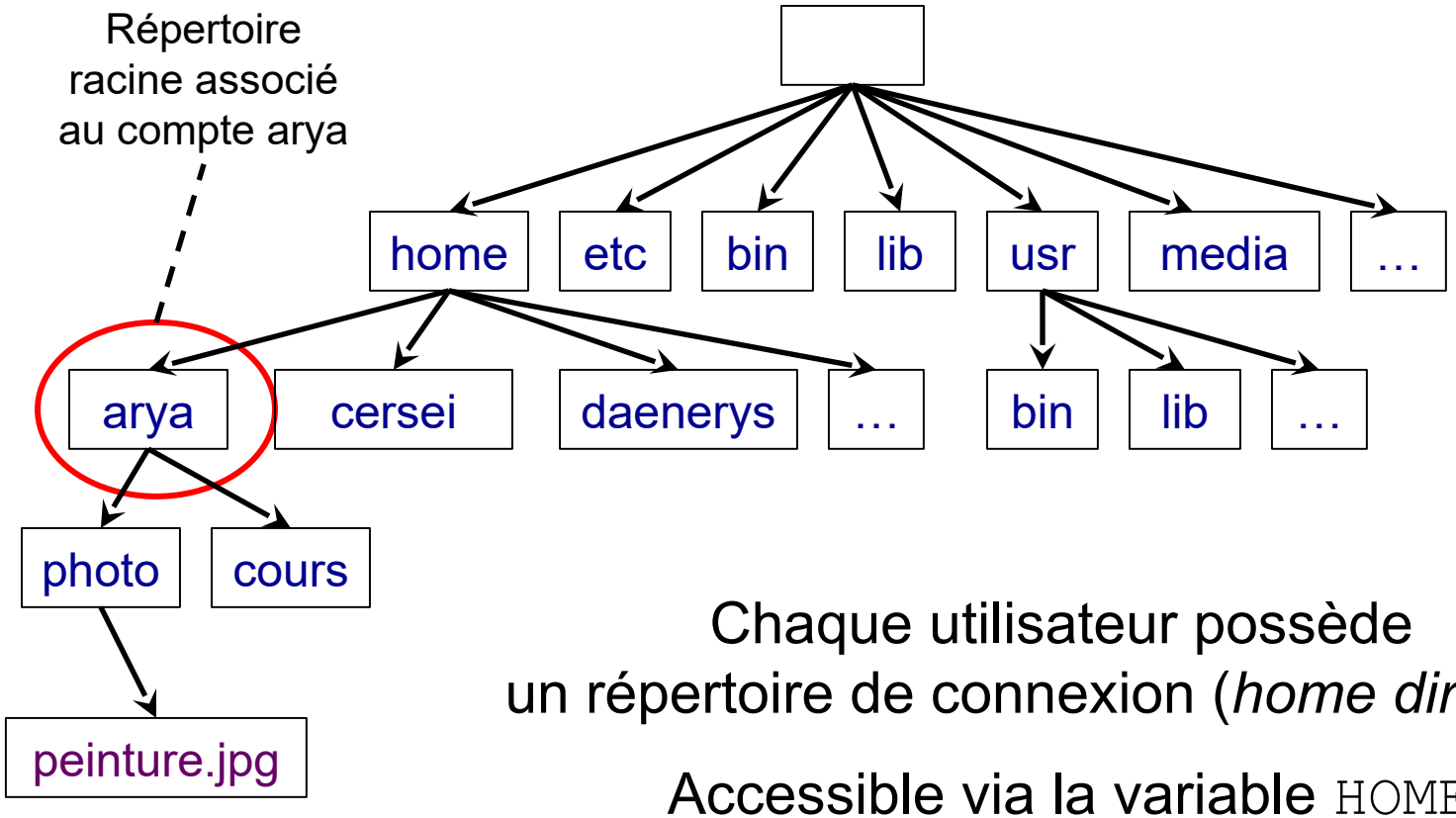
La plupart des systèmes d'exploitation Unix (GNU/Linux, BSD, MacOS…) utilisent une arborescence de base standardisée
(seul Windows utilise une arborescence réellement différente)

Vous pouvez la consulter en faisant : `man hier` (pour hierarchy)

Arborescence standard des systèmes d'exploitation UNIX



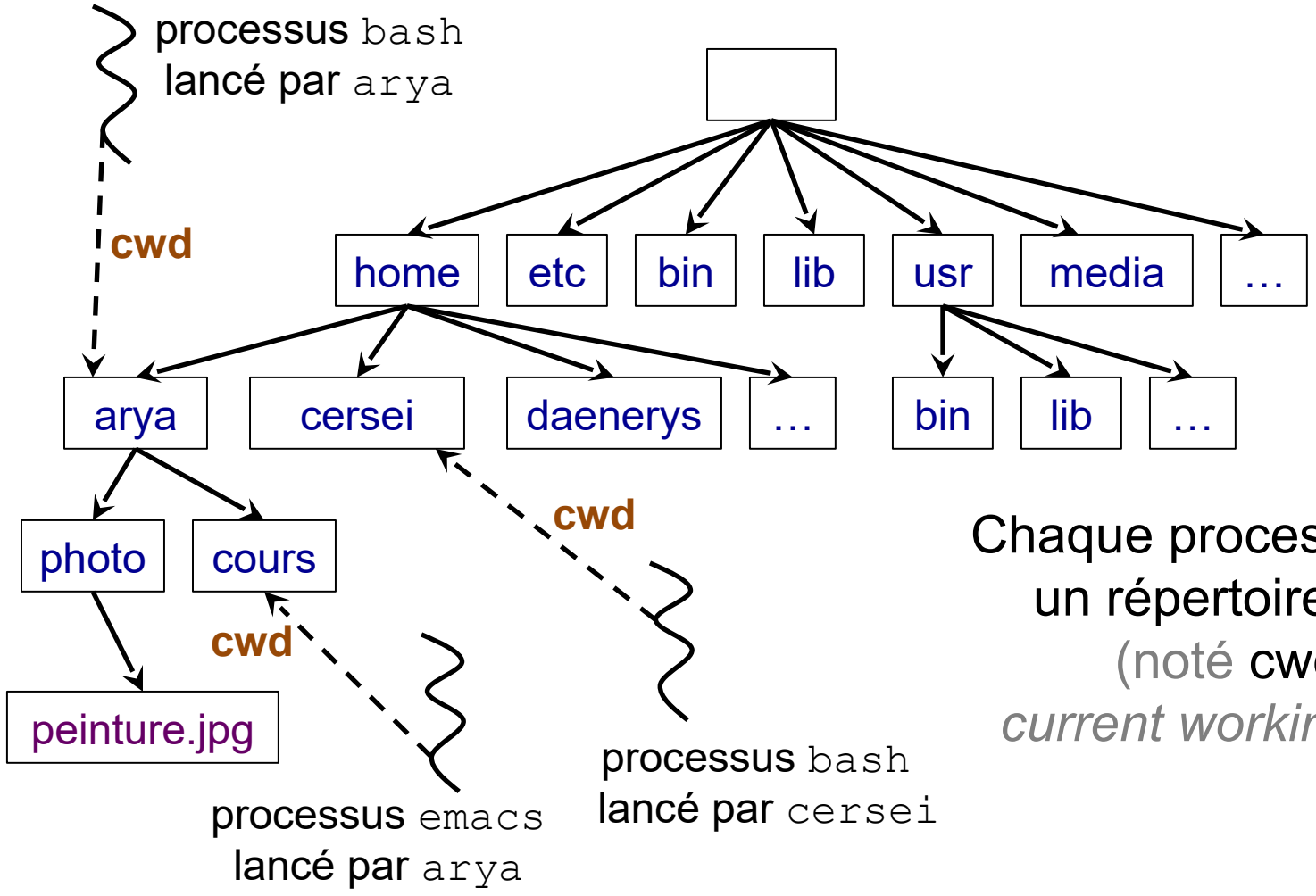
Arborescence standard des systèmes d'exploitation UNIX



Chaque utilisateur possède un répertoire de connexion (*home directory*)

Accessible via la variable `HOME`

Notion de répertoire de travail



Chaque processus possède un répertoire de travail (noté cwd pour *current working directory*)

Notion de chemin

- En `bash`, le séparateur de répertoires est le caractère `/`

- Un chemin s'écrit sous la forme `a/b/c` qui référence
  - le fichier `c`
  - se trouvant dans le répertoire `b`
  - se trouvant lui même dans le répertoire `a`

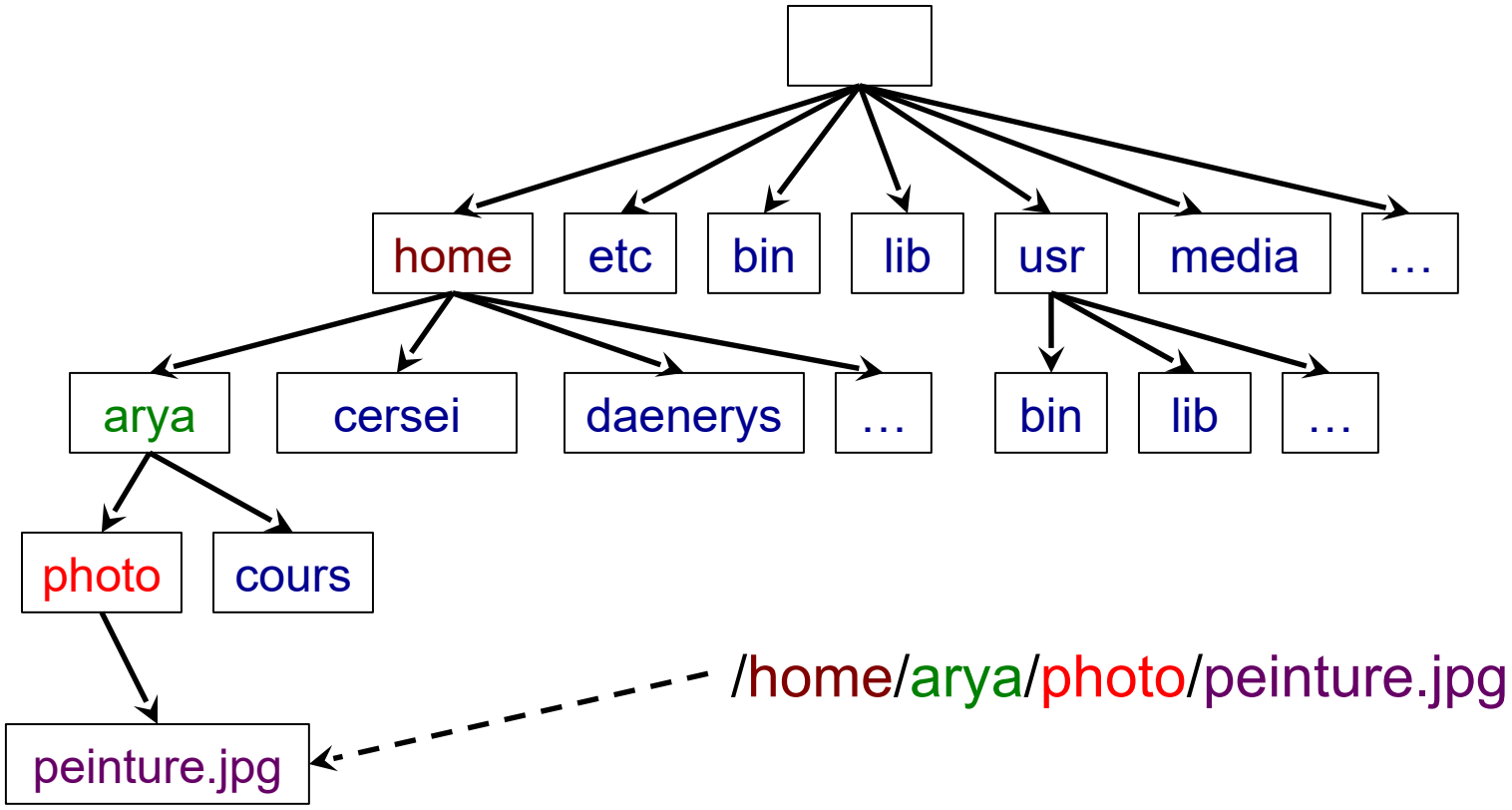
- Un chemin absolu part de la racine du système de fichiers

  Commence par le nom vide (racine), par exemple `/a/b/c`

- Un chemin relatif part du répertoire de travail du processus

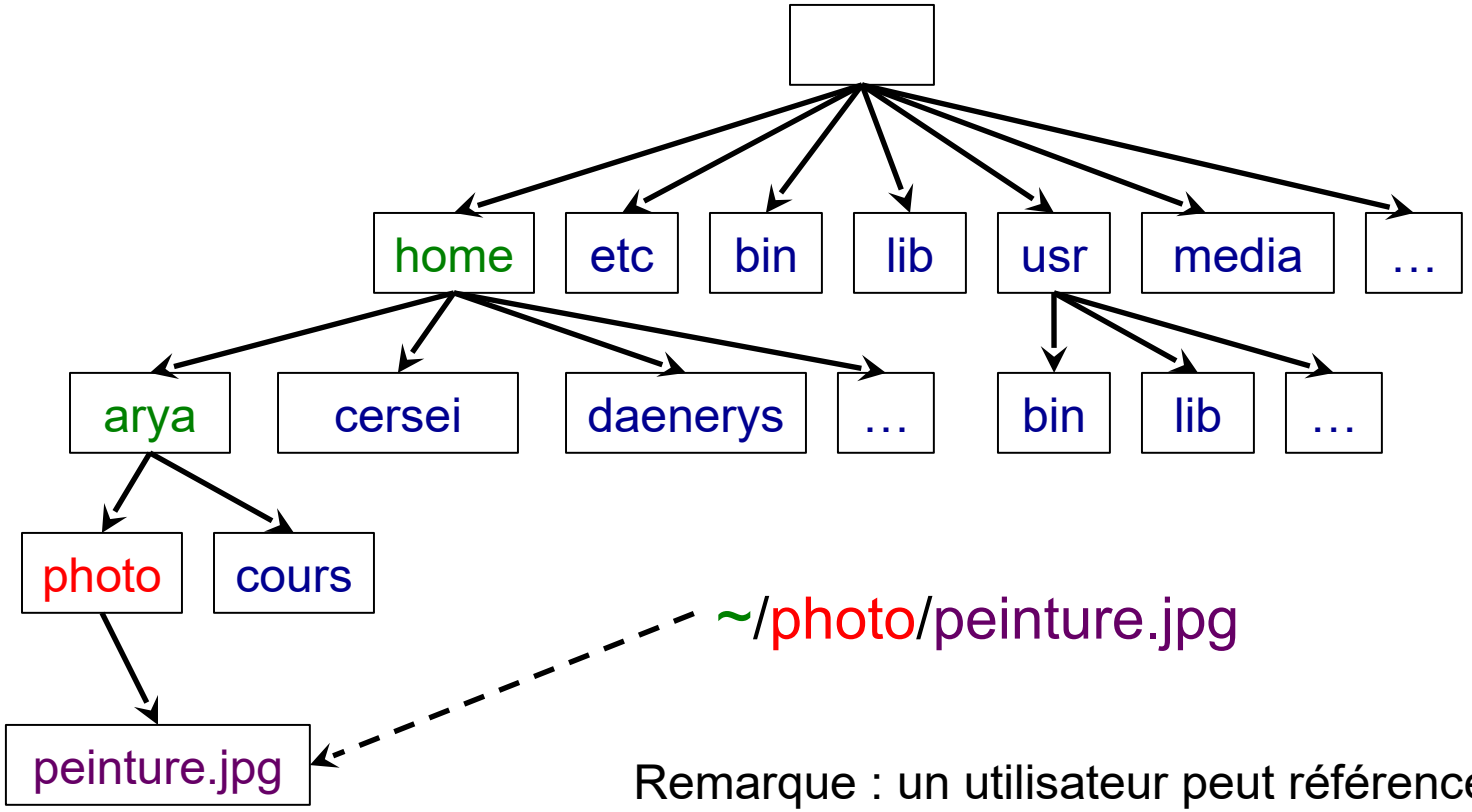
  Commence par un nom non vide, par exemple `a/b/c`

Exemple de chemin absolu (1/2)



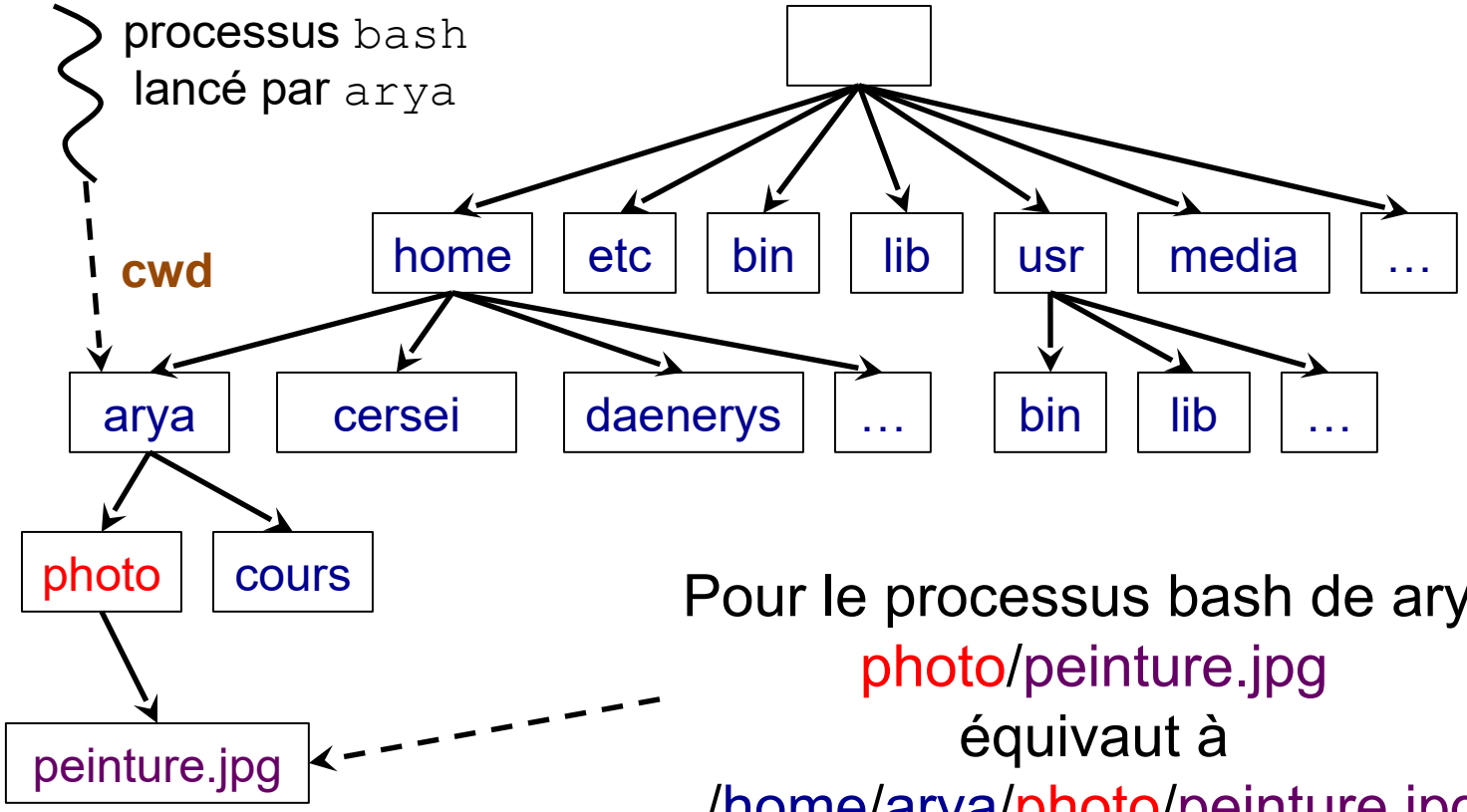
Exemple de chemin absolu (2/2)

Un utilisateur peut utiliser ~ comme raccourci pour son répertoire de connexion



Remarque : un utilisateur peut référencer le répertoire de connexion d'un autre utilisateur avec `~name` (par exemple `~arya/photo/peinture.jpg`)

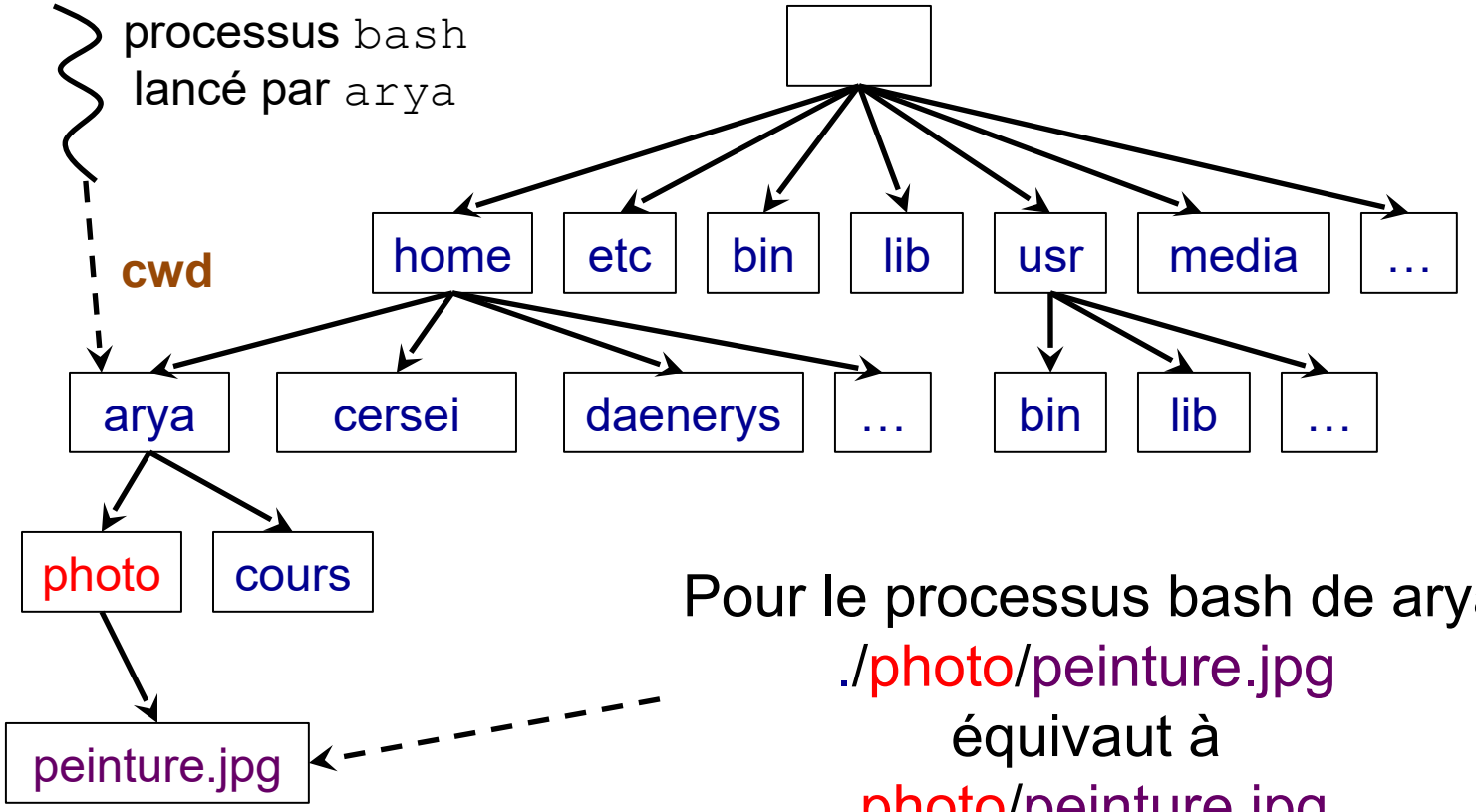
Exemple de chemin relatif (1/3)



Pour le processus bash de arya
photo/peinture.jpg
équivaut à
/home/arya/photo/peinture.jpg

Exemple de chemin relatif (2/3)

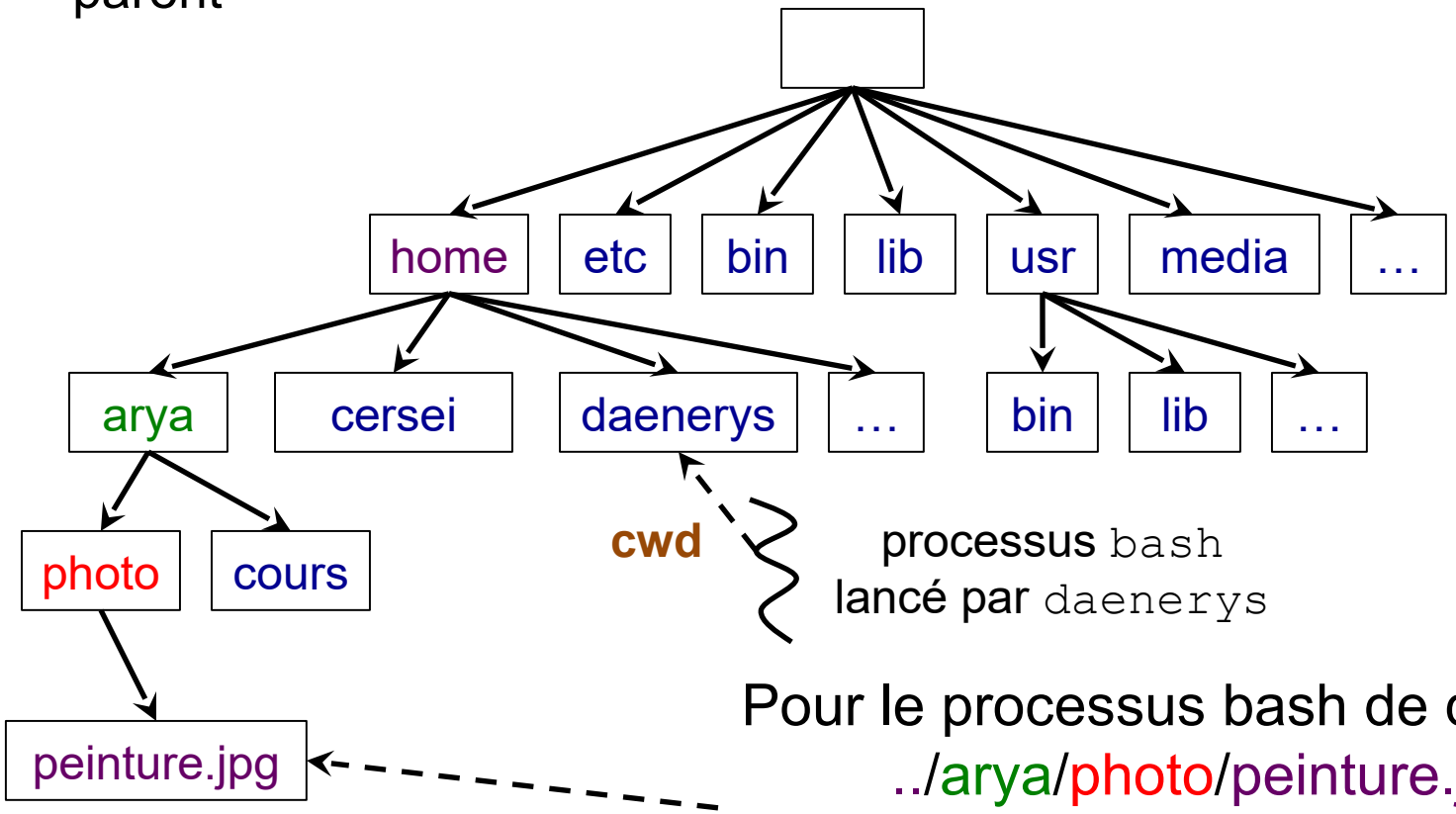
Chaque répertoire possède un fichier nommé `.` s'auto-référençant



Pour le processus bash de arya
./photo/peinture.jpg
équivaut à
photo/peinture.jpg

Exemple de chemin relatif (3/3)

Chaque répertoire possède un fichier nommé `..` référençant son parent



Pour le processus bash de daenerys

../arya/photo/peinture.jpg

équivaut à

/home/arya/photo/peinture.jpg

Remarque

- Dans `bash`, quand vous écrivez `./script.sh`, vous référencez le fichier `script.sh` du répertoire de travail du processus `bash` de votre terminal

Exemple

```
#! /bin/bash



echo "Bonjour, vous êtes dans le répertoire $PWD"
echo "Votre maison se trouve en $HOME"
echo "Et vous avez lancé le script $0"
```

/home/gael/tmp/script.sh

```
$ ./script.sh
Bonjour, vous êtes dans le répertoire /home/gael/tmp
Votre maison se trouve en /home/gael
Et vous avez lancé le script ./script.sh
$
```

Explorer l'arborescence de fichiers

- `cd chem` : *change directory*

  ⇒ change le répertoire courant vers `chem`

  Exemple : `cd ../cersei; cd /home/arya/photo`

  (sans argument, `cd` va dans votre répertoire de connexion)

- `pwd` : *print working directory*

  ⇒ affiche le répertoire de travail (⇔ `echo $PWD`)

Explorer l'arborescence de fichiers

- `ls chem` : *list*

  ⇒ liste le chemin `chem`

  - Si `chem` absent : affiche le contenu du répertoire courant
  - Si `chem` répertoire : affiche le contenu du répertoire `chem`
  - Sinon si `chem` est un fichier : affiche le nom du fichier
  - Options utiles :
    - `-a` : affiche les fichiers cachés (c.-à.d., commençant par '.')
    - `-l` : affichage long (propriétaire, droits d'accès, taille etc.)
    - `-d` : affiche le informations sur un répertoire au lieu de son contenu

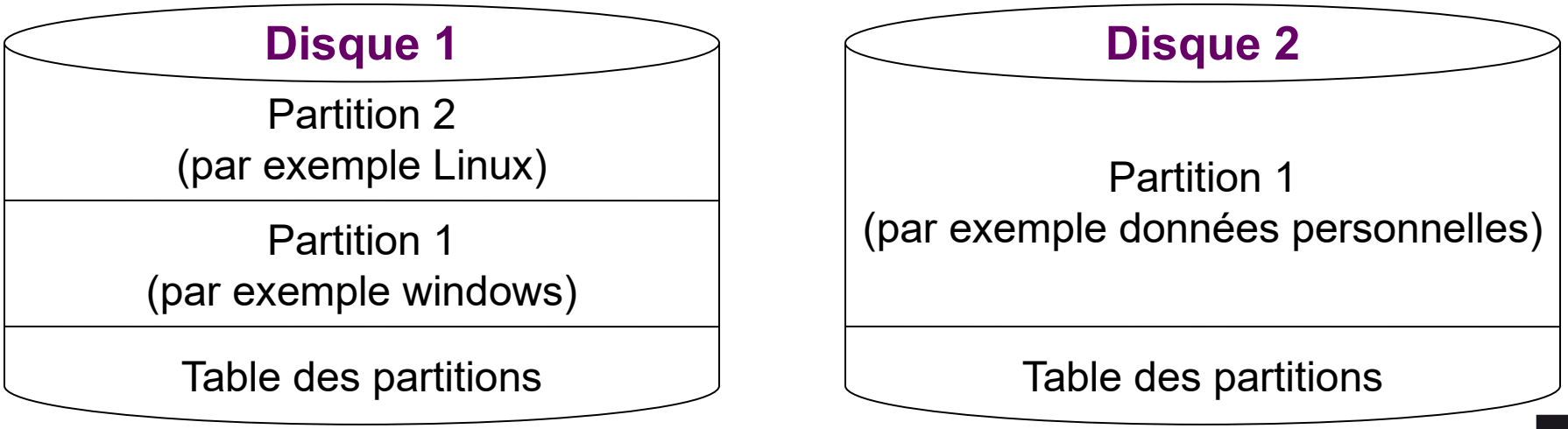
- Le système de fichiers vu par un processus
- Le système de fichiers sur disque
- Les commandes utilisateurs
- Les droits d'accès

Organisation des disques

- Une machine peut posséder plusieurs disques
- Et chaque disque peut être scindé en plusieurs partitions

  Utile pour installer plusieurs systèmes d'exploitation ou pour augmenter l'indépendance entre les données utilisateurs et le système d'exploitation

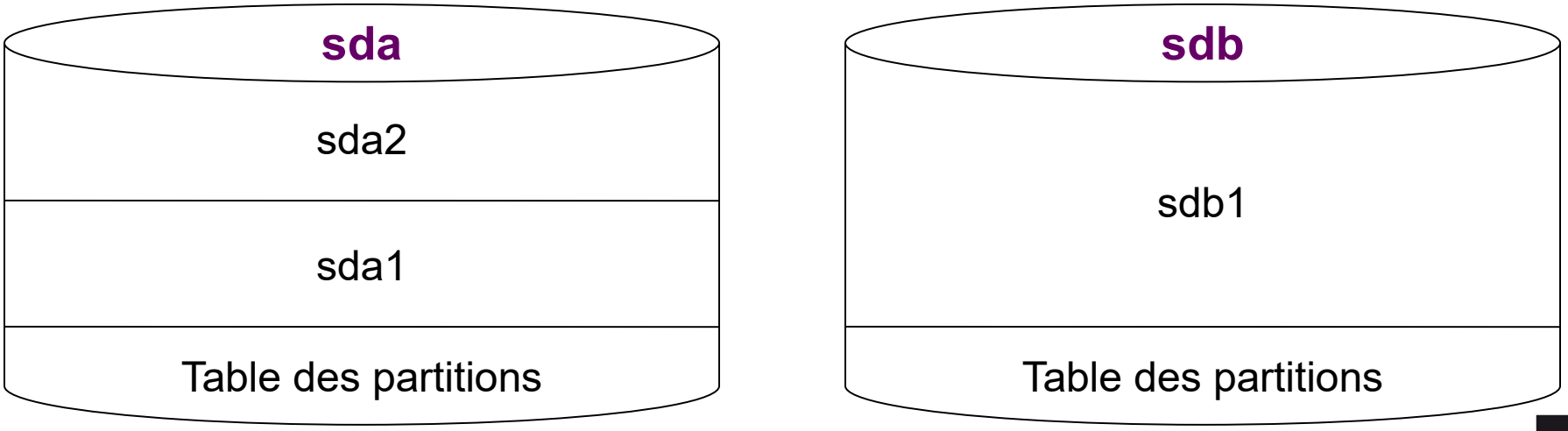
  - Chaque partition possède son système de fichiers indépendant



Les partitions dans les systèmes UNIX

- Un disque est identifié par le préfixe `sd` (*scsi drive*)
- Les disques sont numérotés `a`, `b`, `c`…
- Les partitions sont numérotées `1`, `2`, `3`…

  (vous pouvez voir les disques/partitions en faisant `ls /dev`)



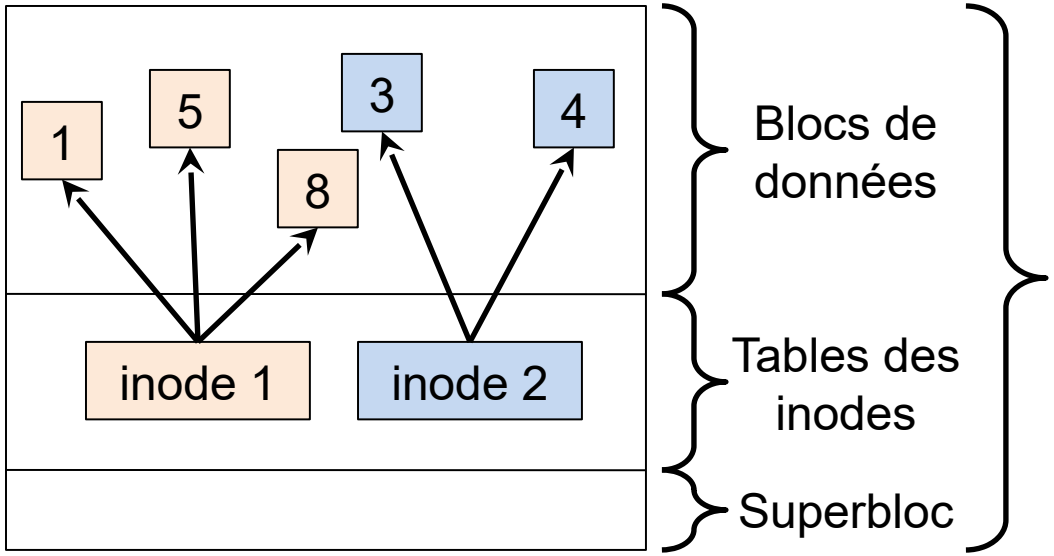
Le système de fichiers sur disque (1/2)

- 3 concepts fondamentaux
  - Le bloc : unité de transfert entre le disque et la mémoire
    (souvent 4096 octets)
  - L'inode (*index node*) : descripteur d'un fichier
    - Type de l'inode (fichier ordinaire, répertoire, autres)
    - Propriétaire, droits, dates de création/modification/accès
    - Taille
    - Liste des blocs du contenu du fichier
    - …
  - Donc, dans ce cours : fichier = inode + blocs du fichier

Le système de fichiers sur disque (2/2)

- Avec `ext`, utilisé sous GNU/Linux, trois zones principales
  - Le superbloc, au début, décrit les autres zones
  - La table des inodes contient les inodes (inode 0 = racine)
  - La zone des blocs de données contient les données des fichiers

Par exemple,
contenu de inode 1 :
4096 octets du bloc 1 puis
4096 octets du bloc 5 puis
312 octets du bloc 8



Montage d'une partition (1/2)

- Le système maintient une table des montages qui associe des chemins (points de montage) et des disques
  - / ⇒ sda1
  - /home ⇒ sdb1
  - /mnt/windows ⇒ sdb2

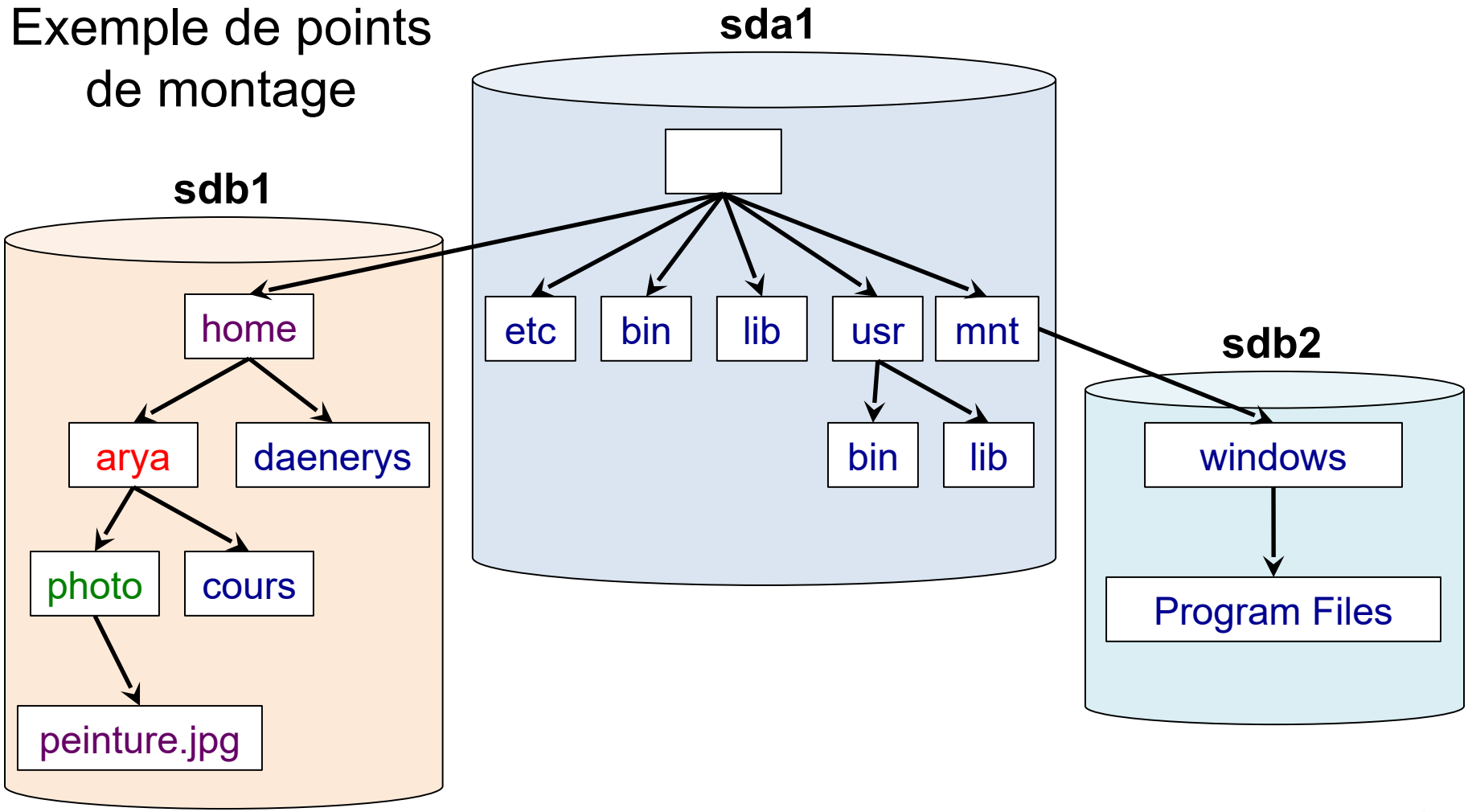
  Remarque : les partitions du disque dur peuvent se trouver sur une autre machine
  (typiquement Network File System, comme en salle TP, voir https://doc.ubuntu-fr.org/nfs)

- Lorsqu'un processus accède à un point de montage, il accède à l'inode racine du disque indiqué dans la table des montages

  Par exemple `cd /mnt/windows` accède à l'inode racine de `sdb2`

Montage d'une partition (2/2)

Exemple de points de montage



Lien direct (1/2)

- Le nom d'un inode dans un répertoire s'appelle un lien direct
  (*hard link* en anglais, aussi appelé parfois lien dur, physique ou matériel)

- On peut créer plusieurs liens directs vers le même inode

  Commande `ln chem-cible chem-lien`

  - Aucune différence entre le nom original et le nouveau nom
  - Facilite l'accès à des fichiers à partir d'emplacements connus

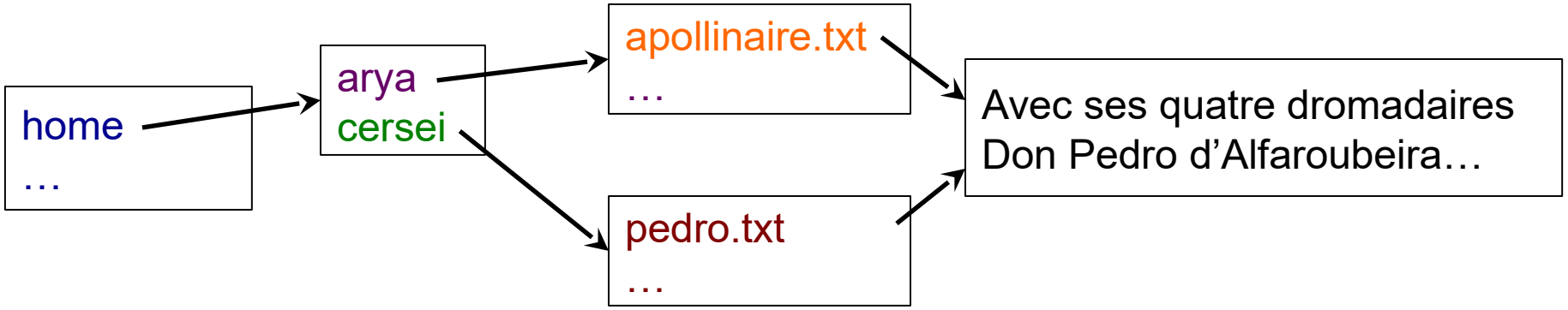
Lien direct (1/2)

- Le nom d'un inode dans un répertoire s'appelle un lien direct
  (*hard link* en anglais, aussi appelé parfois lien dur, physique ou matériel)

- On peut créer plusieurs liens directs vers le même inode

  Commande `ln chem-cible chem-lien`

  - Aucune différence entre le nom original et le nouveau nom
  - Facilite l'accès à des fichiers à partir d'emplacements connus




```
ln /home/arya/apollinaire.txt /home/cersei/pedro.txt
```

Lien direct (2/2)

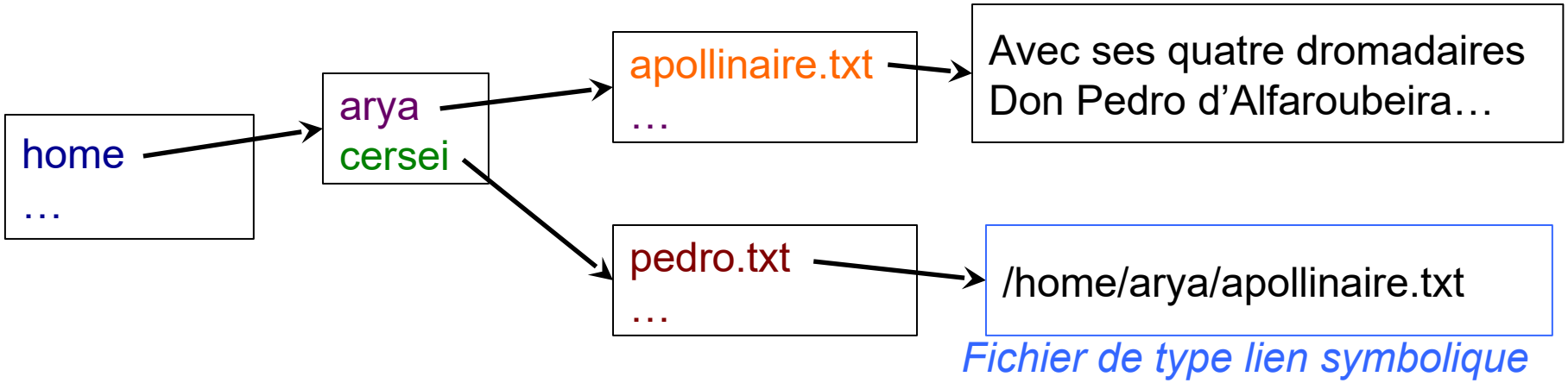
- Mais faire de multiples liens directs pour faire des raccourcis peut poser problème
  - Pour supprimer un fichier, il faut supprimer tous les liens directs vers son inode, mais les utilisateurs sont distraits et en oublient
  - Un lien direct ne peut référencer qu'un inode de la même partition (du même système de fichiers)

Notion de lien symbolique (1/2)

- Pour faire des raccourcis on utilise aussi des liens symboliques

  Comme `ln -s chem-cible chem-lien`

  - Fichier spécial (type lien) dont le contenu est un chemin cible
  - Lorsque le système doit ouvrir le fichier, il ouvre la cible à la place de l'original




```
ln -s /home/arya/apollinaire.txt /home/cersei/pedro.txt
```

Notion de lien symbolique (2/2)

- Avantages des liens symboliques
  - Dès que le fichier cible est détruit, son espace est libéré (ouvrir le lien symbolique engendre alors une erreur)
  - Un lien symbolique peut référencer un fichier quelconque, y compris appartenant à une autre partition

- Principal inconvénient des liens symboliques
  - En cas de déplacement du fichier cible, le lien symbolique peut devenir invalide

Il existe de nombreux types de fichiers

- Fichier ordinaire
- Répertoire
- Lien symbolique
- Device : un fichier qui représente un périphérique
  (disque dur, carte son, carte réseau, …)
  - Par exemple `/dev/sda1`
- Tube nommé : fichier spécial étudié en CI6
- Socket : fichier spécial proche des tubes (non étudié dans ce cours)

- Le système de fichiers vu par un processus
- Le système de fichier sur disque
- Les commandes utilisateurs
- Les droits d'accès

Commandes utilisateur

- Commandes de base sur les fichiers
  - Création
  - Suppression
  - Copie
  - Déplacement / renommage
  - Consultation
  - Recherche

- Commandes utilitaires bien pratiques
  - Principales vues en TP

Création d'un fichier

- Création d'un fichier ordinaire :
  - Au travers de logiciels
    - en particulier des éditeurs : emacs, vi, gedit, etc…
  - `touch chem` : crée fichier vide + mise à jour heures modif.
- Création d'un répertoire :
  - `mkdir rep` : *make directory*
- Création d'un lien :
  - Lien dur : `ln chem-cible chem-lien`
  - Lien symbolique : `ln -s chem-cible chem-lien`

Suppression d'un fichier (1/5)

- Supprimer un fichier (tout type, sauf répertoire)

  `rm chem` : *remove*

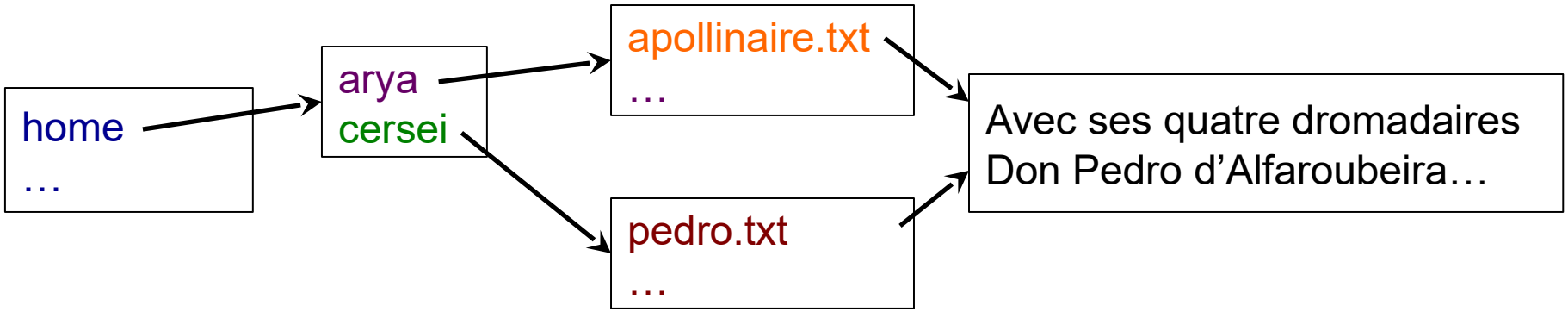
  - Suppression de l'entrée associée au chemin dans le répertoire parent
    - Décrémentation du compteur de liens directs de l'inode
    - Libère le fichier (inode + données) si compteur tombe à zéro

Suppression d'un fichier (2/5)

- Supprimer un fichier (tout type, sauf répertoire)

  `rm chem` : *remove*

  - Suppression de l'entrée associée au chemin dans le répertoire parent
    - Décrémentation du compteur de liens directs de l'inode
    - Libère le fichier (inode + données) si compteur tombe à zéro

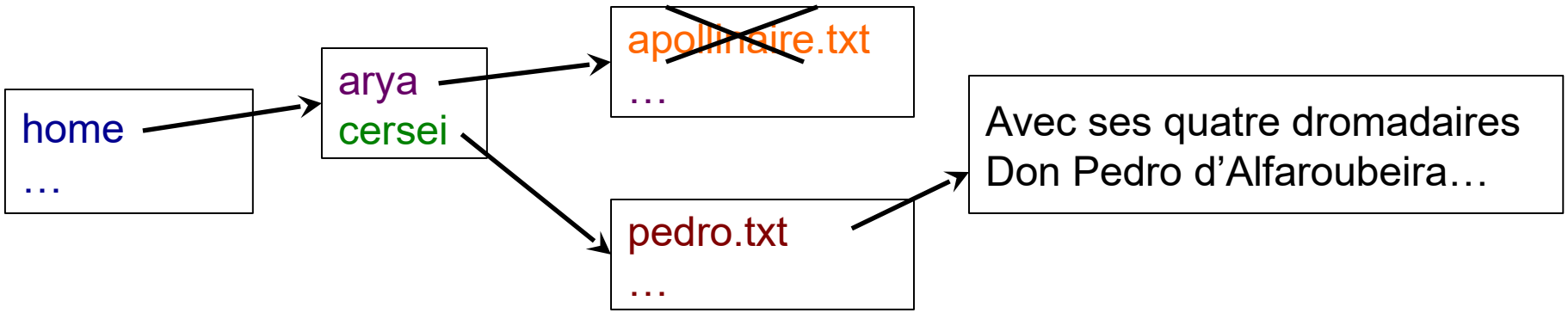


Suppression d'un fichier (3/5)

- Supprimer un fichier (tout type, sauf répertoire)

  `rm chem` : *remove*

  - Suppression de l'entrée associée au chemin dans le répertoire parent
    - Décrémentation du compteur de liens directs de l'inode
    - Libère le fichier (inode + données) si compteur tombe à zéro



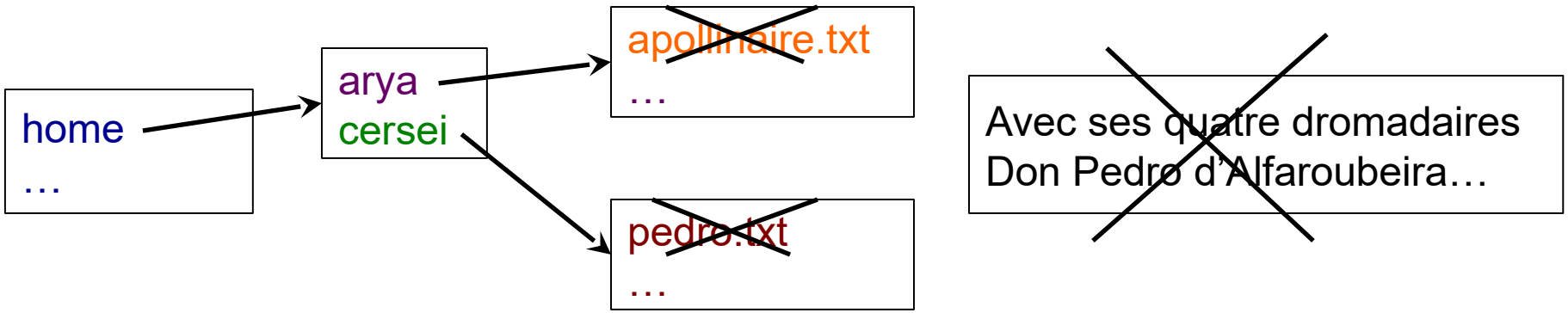
`rm /home/arya/apollinaire.txt`

Suppression d'un fichier (4/5)

- Supprimer un fichier (tout type, sauf répertoire)

  `rm chem` : *remove*

  - Suppression de l'entrée associée au chemin dans le répertoire parent
    - Décrémentation du compteur de liens directs de l'inode
    - Libère le fichier (inode + données) si compteur tombe à zéro




```
rm /home/cersei/pedro.txt
```

Suppression d'un fichier (5/5)

- Supprimer un fichier (tout type, sauf répertoire)

  `rm chem` : *remove*

  - Suppression de l'entrée associée au chemin dans le répertoire parent
    - Décrémentation du compteur de liens directs de l'inode
    - Libère le fichier (inode + données) si compteur tombe à zéro

- Supprimer un répertoire
  - `rmdir <rep>` : suppression d'un répertoire vide
  - `rm -r <rep>` : suppression récursive d'un répertoire et de tous les sous-fichiers (sous-répertoires inclus)

    (faites très attention avec cette commande !)
  - `rm -i <rep>` : demande confirmation avant suppression (utile !)

Copie d'un fichier (1/3)

- `cp src dest` : *copy*

  Création d'un nouvel inode et duplication des blocs de données
  - `src` correspond au chemin du fichier à copier
  - `dest`, au chemin où doit être copiée `src`

- Deux fonctionnements différents
  - Si `dest` est un répertoire, copie `src` dans le répertoire `dest` (dans ce cas, multiples copies possibles avec `cp fic1 fic2 … rep`)
  - Sinon, copie `src` sous le nom `dest`

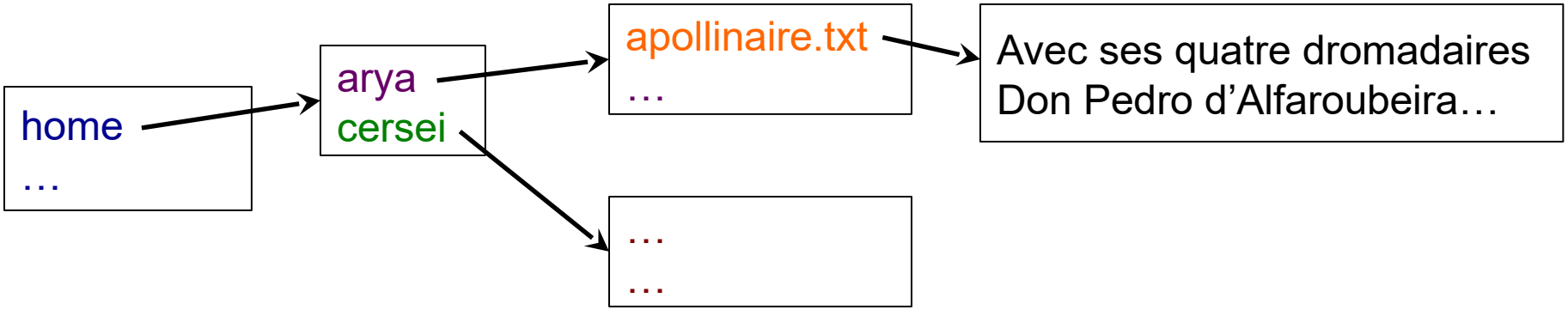
- L'option `-r` permet de copier récursivement un répertoire (sans `-r`, si `src` est un répertoire, erreur)

Copie d'un fichier (2/3)

- `cp src dest` : *copy*

  Création d'un nouvel inode et duplication des blocs de données

  - `src` correspond au chemin du fichier à copier
  - `dest`, au chemin où doit être copiée `src`

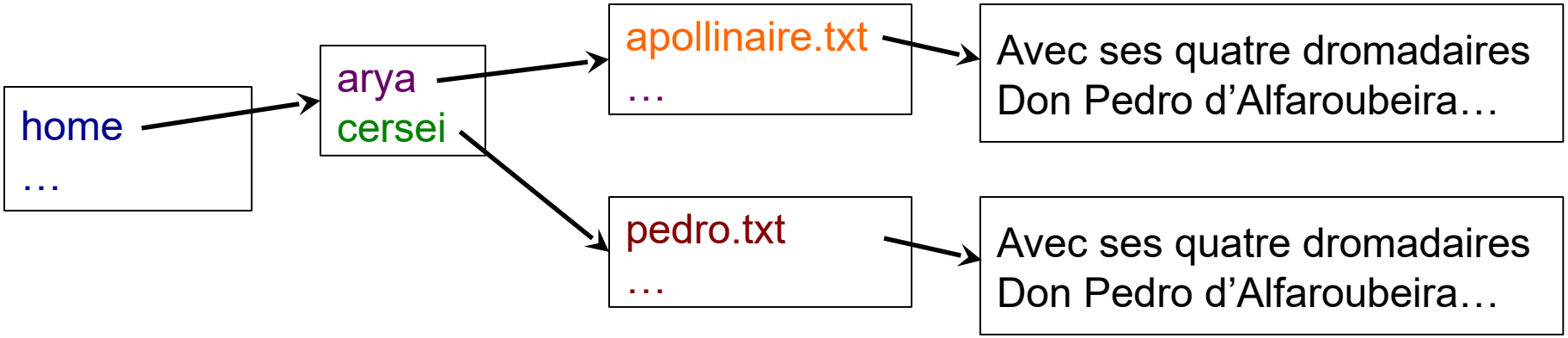


Copie d'un fichier (3/3)

- `cp src dest` : *copy*

  Création d'un nouvel inode et duplication des blocs de données

  - `src` correspond au chemin du fichier à copier
  - `dest`, au chemin où doit être copiée `src`




```
cp /home/arya/apollinaire.txt /home/cersei/pedro.txt
```

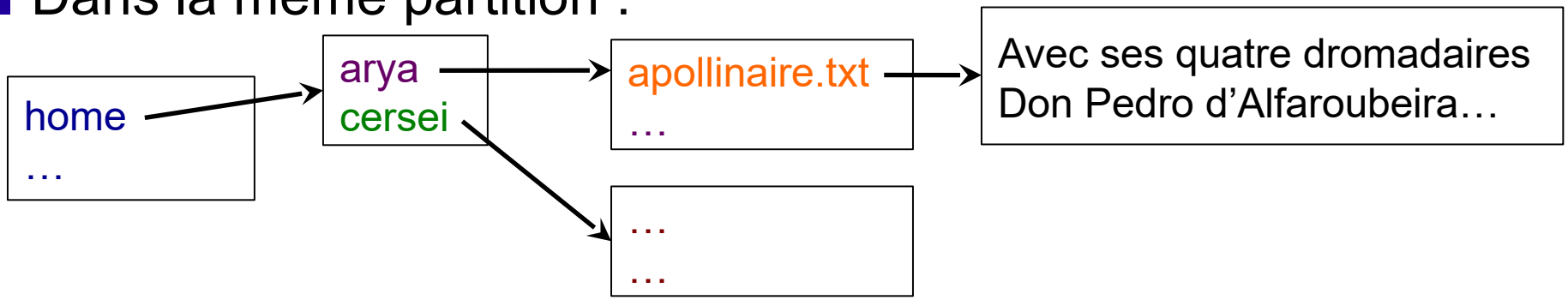
Déplacement d'un fichier (1/7)

- `mv src dest` : *move* (déplace ou renomme)
  - *`src` : fichier de type quelconque*
  - Si `dest` est un répertoire, déplace `src` dans le répertoire `dest` (dans ce cas, multiples déplacements possibles avec `mv fic1 fic2 … rep`)
  - Sinon, déplace `src` sous le nom `dest`
    - Si `dest` est dans le même répertore : renommage

- Fonctionnement :
  - Déplacement dans la même partition
    - Crée un lien direct à partir de `dest` puis supprime `src`
  - Déplacement sur une autre partition
    - Copie `src` vers `dest` puis supprime `src`

Déplacement d'un fichier (2/7)

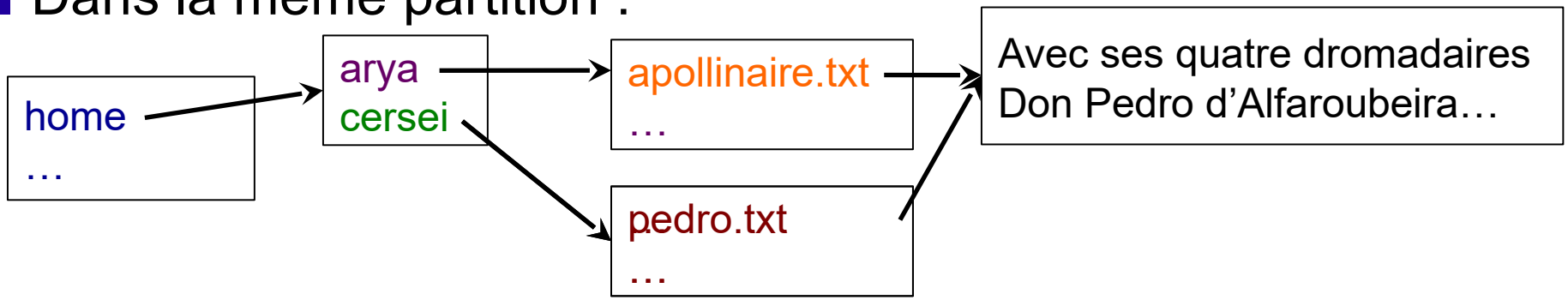
- `mv src dest` : *move* (déplace ou renomme)
  - *`src` : fichier de type quelconque*
  - Si `dest` est un répertoire, déplace `src` dans le répertoire `dest` (dans ce cas, multiples déplacements possibles avec `mv fic1 fic2 … rep`)
  - Sinon, déplace `src` sous le nom `dest`
    - Si `dest` est dans le même répertore : renommage
- Dans la même partition :




```
mv /home/arya/apollinaire.txt /home/cersei/pedro.txt
```

Déplacement d'un fichier (3/7)

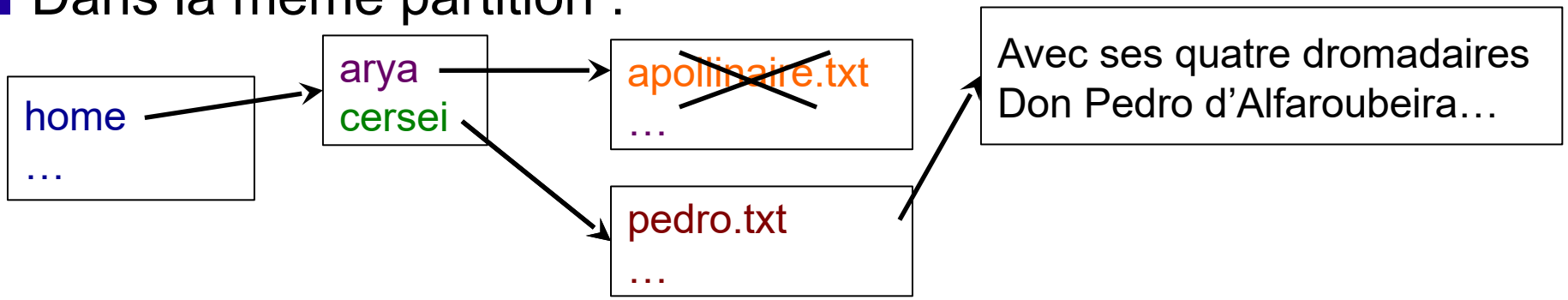
- `mv src dest` : *move* (déplace ou renomme)
  - *`src` : fichier de type quelconque*
  - Si `dest` est un répertoire, déplace `src` dans le répertoire `dest` (dans ce cas, multiples déplacements possibles avec `mv fic1 fic2 … rep`)
  - Sinon, déplace `src` sous le nom `dest`
    - Si `dest` est dans le même répertore : renommage
- Dans la même partition :



`mv /home/arya/apollinaire.txt /home/cersei/pedro.txt`

Déplacement d'un fichier (4/7)

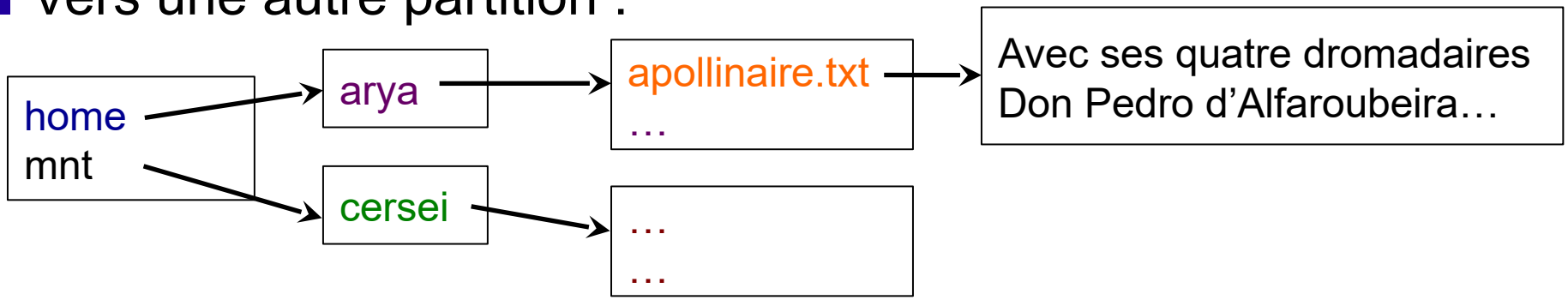
- `mv src dest` : *move* (déplace ou renomme)
  - *`src` : fichier de type quelconque*
  - Si `dest` est un répertoire, déplace `src` dans le répertoire `dest` (dans ce cas, multiples déplacements possibles avec `mv fic1 fic2 … rep`)
  - Sinon, déplace `src` sous le nom `dest`
    - Si `dest` est dans le même répertore : renommage
- Dans la même partition :



`mv /home/arya/apollinaire.txt /home/cersei/pedro.txt`

Déplacement d'un fichier (5/7)

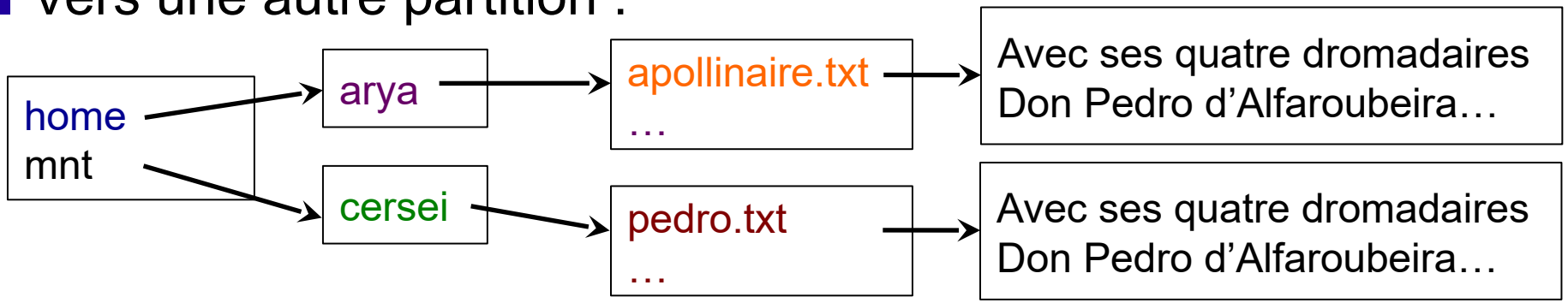
- `mv src dest` : *move* (déplace ou renomme)
  - *`src` : fichier de type quelconque*
  - Si `dest` est un répertoire, déplace `src` dans le répertoire `dest` (dans ce cas, multiples déplacements possibles avec `mv fic1 fic2 … rep`)
  - Sinon, déplace `src` sous le nom `dest`
    - Si `dest` est dans le même répertore : renommage
- Vers une autre partition :




```
mv /home/arya/apollinaire.txt /mnt/cersei/pedro.txt
```

Déplacement d'un fichier (6/7)

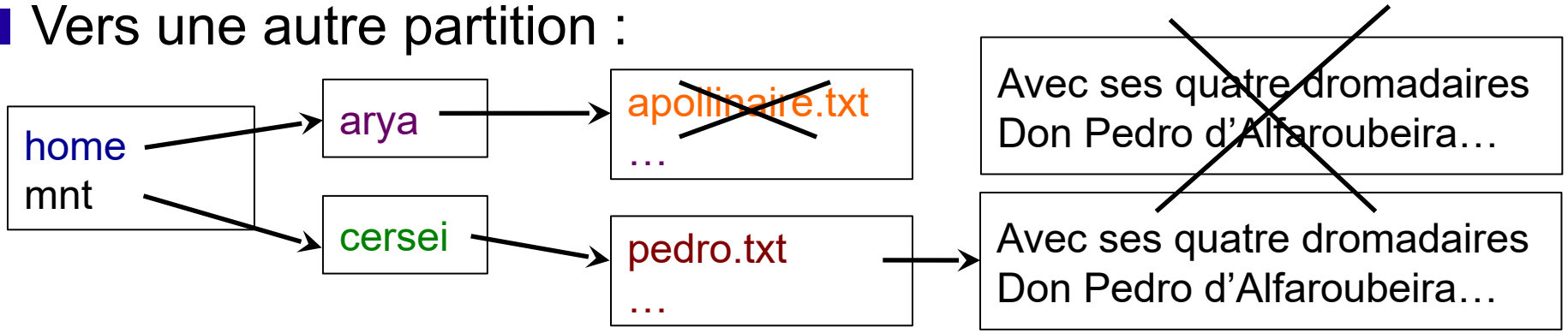
- `mv src dest` : *move* (déplace ou renomme)
  - *`src` : fichier de type quelconque*
  - Si `dest` est un répertoire, déplace `src` dans le répertoire `dest` (dans ce cas, multiples déplacements possibles avec `mv fic1 fic2 … rep`)
  - Sinon, déplace `src` sous le nom `dest`
    - Si `dest` est dans le même répertore : renommage
- Vers une autre partition :




```
mv /home/arya/apollinaire.txt /mnt/cersei/pedro.txt
```

Déplacement d'un fichier (7/7)

- `mv src dest` : *move* (déplace ou renomme)
  - *`src` : fichier de type quelconque*
  - Si `dest` est un répertoire, déplace `src` dans le répertoire `dest` (dans ce cas, multiples déplacements possibles avec `mv fic1 fic2 … rep`)
  - Sinon, déplace `src` sous le nom `dest`
    - Si `dest` est dans le même répertore : renommage
- Vers une autre partition :



`mv /home/arya/apollinaire.txt /mnt/cersei/pedro.txt`

- Le système de fichiers vu par un processus
- Le système de fichiers sur disque
- Les commandes utilisateurs
- Les droits d'accès

Droits d'accès

- Toute opération sur un fichier est soumise à droits d'accès
  - Message d'erreur « *Permission non accordée* »

- 3 types d'accès
  - `r` : droit de lecture
    - Si répertoire, consultation de ses entrées (c.-à-.d, `ls` autorisé)
    - Sinon, consultation du contenu du fichier
  - `w` : droit d'écriture
    - Si répertoire, droit de création, de renommage et de suppression d'une entrée dans le répertoire
    - Sinon, droit de modification du contenu du fichier
  - `x` :
    - si répertoire, droit de traverser (c.-à-.d., `cd` autorisé)
    - sinon, droit d'exécution

Droits d'accès

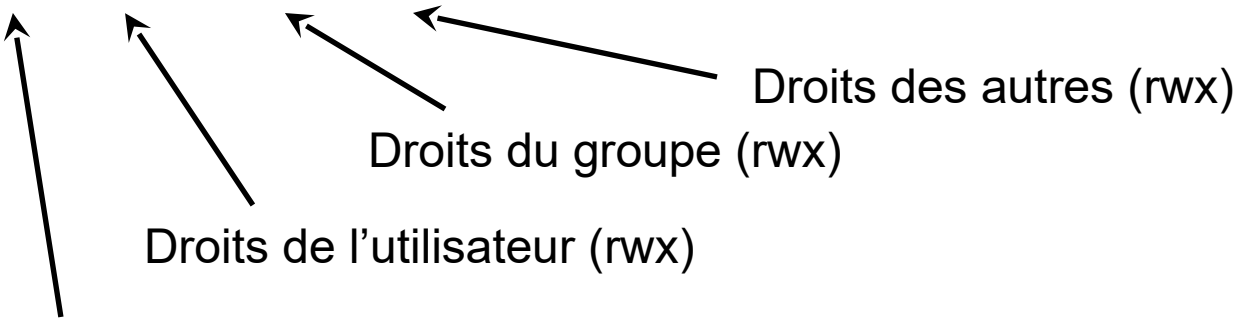
- 3 catégories d'utilisateurs :
  - Propriétaire (`u`)
  - Groupe propriétaire (`g`)
  - Tous les autres (`o`)

- Chaque catégorie possède ses types d'accès `r w x`

Droits d'accès – consultation

- `ls -ld` ⇒ donne les droits des fichiers

- Format de sortie de `ls -l`
  - `- --- --- ---`



Type du fichier :
d : répertoire
l : lien symbolique
- : fichier ordinaire

```
$ ls -l fichier
- rwx r-- --- fichier
$
```

Droits d'accès – modification

- Modification sur un fichier existant

  `chmod droit fichier` : change mode

- Droits à **!appliquer!** au fichier
  - Catégories : `u`, `g`, `o` ou `a` (= `all` c.-à-.d., `ugo`)
  - Opérations : Ajout (+), retrait (−), affectation (=)

```
$
```

Droits d'accès – modification

- Modification sur un fichier existant

  `chmod droit fichier` : change mode

- Droits à **!appliquer!** au fichier
  - Catégories : `u`, `g`, `o` ou `a` (= `all` c.-à-.d., `ugo`)
  - Opérations : Ajout (+), retrait (−), affectation (=)

```
$ ls -ld fichier
-rwx r-- --- fichier
$
```

Droits d'accès – modification

- Modification sur un fichier existant

  `chmod droit fichier` : change mode

- Droits à **!appliquer!** au fichier
  - Catégories : `u`, `g`, `o` ou `a` (= `all` c.-à-.d., `ugo`)
  - Opérations : Ajout (+), retrait (−), affectation (=)

```
$ ls -ld fichier
-rwx r-- --- fichier
$ chmod u-x fichier
$ ls -ld fichier
-rw- r-- --- fichier
$
```

Droits d'accès – modification

- Modification sur un fichier existant

  `chmod droit fichier` : change mode

- Droits à **!appliquer!** au fichier
  - Catégories : `u`, `g`, `o` ou `a` (= `all` c.-à-.d., `ugo`)
  - Opérations : Ajout (+), retrait (−), affectation (=)

```
$ ls -ld fichier
-rwx r-- --- fichier
$ chmod u-x fichier
$ ls -ld fichier
-rw- r-- --- fichier
$ chmod u+x fichier
$ ls -ld fichier
-rwx r-- --- fichier
```

Démonstration

```
$ cp /etc/passwd .
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$ cat passwd
cat: passwd: Permission non accordée
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$ cat passwd
cat: passwd: Permission non accordée
$ mkdir rep
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$ cat passwd
cat: passwd: Permission non accordée
$ mkdir rep
$ ls -l
total 8
--w-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
drwxr-xr-x 2 gthomas users   68 19 juil.  2016 rep
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$ cat passwd
cat: passwd: Permission non accordée
$ mkdir rep
$ ls -l
total 8
--w-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
drwxr-xr-x 2 gthomas users   68 19 juil.  2016 rep
$ cd rep/
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$ cat passwd
cat: passwd: Permission non accordée
$ mkdir rep
$ ls -l
total 8
--w-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
drwxr-xr-x 2 gthomas users   68 19 juil.  2016 rep
$ cd rep/
$ cd ..
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$ cat passwd
cat: passwd: Permission non accordée
$ mkdir rep
$ ls -l
total 8
--w-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
drwxr-xr-x 2 gthomas users   68 19 juil.  2016 rep
$ cd rep/
$ cd ..
$ chmod u-x rep
$
```

Démonstration

```
$ cp /etc/passwd .
$ ls -l
total 4
-rw-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
$ chmod u-r passwd
$ cat passwd
cat: passwd: Permission non accordée
$ mkdir rep
$ ls -l
total 8
--w-r--r-- 1 gthomas users 1120 19 juil.  2016 passwd
drwxr-xr-x 2 gthomas users   68 19 juil.  2016 rep
$ cd rep/
$ cd ..
$ chmod u-x rep
$ cd rep
-bash: cd: rep: Permission non accordée
```

Droits d'accès initiaux

- Masque de droits d'accès **!retirés!** à la création de tout fichier
  - Commande `umask` (*user mask*)
  - Le masque est donné en octal (base 8) avec 3 chiffres (`u`, `g`, `o`)
  - En standard, masque par défaut = `022`
    - `r = 100` en binaire = 4 en octal, `w = 010 = 2`
    - Si droits retirés `--- -w- -w-`, alors droits appliqués `rw- r-- r--`
    - Le droit `x` est déjà retiré par défaut en général

  - Modification du masque grâce à la commande `umask`
    - Attention : `umask` sans effet rétroactif sur les fichiers préexistants
    - Attention : `umask` n'a d'effet que sur le `bash` courant

Démonstration

```
$ touch fichier_umask_defaut
$
```

Démonstration

```
$ touch fichier_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0 oct. 2 10:49 fichier_umask_defaut
$
```

Tous les fichiers sont créés avec des droits par défaut

Démonstration

```
$ touch fichier_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0 oct. 2 10:49 fichier_umask_defaut
$ mkdir repertoire_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0     oct. 2 10:49 fichier_umask_defaut
drwxrwxr-x 2 amina amina   4,0K  oct.  2 10:50 repertoire_umask_defaut
$
```

Et les répertoires aussi. Les droits des fichiers et des répertoires sont souvent différents

Démonstration

```
$ touch fichier_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0 oct. 2 10:49 fichier_umask_defaut
$ mkdir repertoire_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0     oct. 2 10:49 fichier_umask_defaut
drwxrwxr-x 2 amina amina 4,0K  oct. 2 10:50 repertoire_umask_defaut
$ umask 007
$
```

Ici, umask ne retire aucun droit au propriétaire et au groupe. Il retire tous les droits aux utilisateurs « other »

Démonstration

```
$ touch fichier_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0 oct. 2 10:49 fichier_umask_defaut
$ mkdir repertoire_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0     oct. 2 10:49 fichier_umask_defaut
drwxrwxr-x 2 amina amina   4,0K  oct.  2 10:50 repertoire_umask_defaut
$ umask 007
$ touch fichier_umask_nouveau
$ ls -lh
-rw-rw-r-- 1 amina amina 0     oct. 2 10:49 fichier_umask_defaut
-rw-rw---- 1 amina amina 0     oct. 2 10:52 fichier_umask_nouveau
drwxrwxr-x 2 amina amina 4,0K  oct. 2 10:50 repertoire_umask_defaut
$
```

A partir de là, tous les fichiers et répertoires créés n'ont plus les droits retirés par umask. Les droits des fichers existants ne changent pas

Démonstration

```
$ touch fichier_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0 oct. 2 10:49 fichier_umask_defaut
$ mkdir repertoire_umask_defaut
$ ls -lh
-rw-rw-r-- 1 amina amina 0     oct. 2 10:49 fichier_umask_defaut
drwxrwxr-x 2 amina amina   4,0K  oct.  2 10:50 repertoire_umask_defaut
$ umask 007
$ touch fichier_umask_nouveau
$ ls -lh
-rw-rw-r-- 1 amina amina 0     oct. 2 10:49 fichier_umask_defaut
-rw-rw---- 1 amina amina 0     oct. 2 10:52 fichier_umask_nouveau
drwxrwxr-x 2 amina amina 4,0K  oct. 2 10:50 repertoire_umask_defaut
$ mkdir repertoire_umask_nouveau
$ ls -lh
-rw-rw-r-- 1 amina amina 0     oct. 2 10:49 fichier_umask_defaut
-rw-rw---- 1 amina amina 0     oct. 2 10:52 fichier_umask_nouveau
drwxrwxr-x 2 amina amina 4,0K  oct. 2 10:50 repertoire_umask_defaut
drwxrwx--- 2 amina amina 4,0K  oct. 2 10:53 repertoire_umask_nouveau
```

Conclusion

- Concepts clés :
  - Arborescence, racine du système de fichier, répertoire de connexion, répertoire de travail
  - Chemin absolu, chemin relatif
  - Droits d'accès
  - Partition, inode
  - Fichier, répertoire, liens (direct et symbolique)

- Commandes clés :
  - `pwd`, `cd`, `ls`
  - `chmod`, `umask`
  - `mkdir`, `ln`, `rm`, `rmdir`, `cp`, `mv`

En route pour le TP !

INSTITUT Mines-Télécom

TELECOM SudParis

Compléments sur bash

CSC3102 – Introduction aux systèmes d'exploitation

Elisabeth Brunet & Gaël Thomas

Plan

- Variables notables
- Code de retour d'un processus
- Alias de commandes
- Fichier de configuration bash
- Filtrage de fichiers par motif

Variables notables

- `Bash` définit des variables d'environnement notables :
  - `HOME` : chemin absolu du répertoire de connexion
    - `cd`, `cd ~` et `cd $HOME` sont des commandes équivalentes
  - `PS1` : prompt (défaut `$`)
  - `PS2` : prompt en cas de commande sur plusieurs lignes (défaut `>`)

```
$ if
>
```

Variables notables

- `Bash` définit des variables d'environnement notables :
  - `HOME` : chemin absolu du répertoire de connexion
    - `cd`, `cd ~` et `cd $HOME` sont des commandes équivalentes
  - `PS1` : prompt (défaut `$`)
  - `PS2` : prompt en cas de commande sur plusieurs lignes (défaut >)

```
$ if
> [ 0 == 0 ]; then echo 'yes!'; fi
yes!
$
```

Variables notables

- `Bash` définit des variables d'environnement notables :
  - `HOME` : chemin absolu du répertoire de connexion
    - `cd`, `cd ~` et `cd $HOME` sont des commandes équivalentes
  - `PS1` : prompt (défaut `$`)
  - `PS2` : prompt en cas de commande sur plusieurs lignes (défaut >)

```
$ if
> [ 0 == 0 ]; then echo 'yes!'; fi
yes!
$ PS2="++++ "
$
```

Variables notables

- `Bash` définit des variables d'environnement notables :
  - `HOME` : chemin absolu du répertoire de connexion
    - `cd`, `cd ~` et `cd $HOME` sont des commandes équivalentes
  - `PS1` : prompt (défaut `$`)
  - `PS2` : prompt en cas de commande sur plusieurs lignes (défaut >)

```
$ if
> [ 0 == 0 ]; then echo 'yes!'; fi
yes!
$ PS2="++++ "
$ if
++++
```

Variables notables

- `Bash` définit des variables d'environnement notables :
  - `HOME` : chemin absolu du répertoire de connexion
    - `cd`, `cd ~` et `cd $HOME` sont des commandes équivalentes
  - `PS1` : prompt (défaut `$`)
  - `PS2` : prompt en cas de commande sur plusieurs lignes (défaut >)

```
$ if
> [ 0 == 0 ]; then echo 'yes!'; fi
yes!
$ PS2="++++ "
$ if
++++ [ 0 == 0 ]; then echo 'yes!'; fi
yes!
$
```

Variables notables

- `Bash` définit des variables d'environnement notables :
  - `HOME` : chemin absolu du répertoire de connexion
    - `cd`, `cd ~` et `cd $HOME` sont des commandes équivalentes
  - `PS1` : prompt (défaut `$`)
  - `PS2` : prompt en cas de commande sur plusieurs lignes (défaut >)

```
$ if
> [ 0 == 0 ]; then echo 'yes!'; fi
yes!
$ PS2="++++ "
$ if
++++ [ 0 == 0 ]; then echo 'yes!'; fi
yes!
$ PS1="ceci est un prompt: "
ceci est un prompt:
```

La variable d'environnement PATH

- `PATH` : ensemble de chemins séparés par des deux points (`:`)

  Typiquement : `PATH=/bin:/usr/bin`

- Lorsque `bash` essaye d'exécuter `cmd`
  - Si `cmd` contient un `/`, lance l'exécutable de chemin `cmd`
    Exemple : `./truc.sh`, `/bin/truc.sh`
  - Sinon
    - Si `cmd` est une commande interne
      (c.-à-.d, directement exécutable par bash), exécute la commande
      Exemple : en général, les commandes `read` ou `echo`
    - Sinon, `bash` cherche `cmd` dans les répertoires du `PATH`
      Exemple : `test.sh` ⇒ `/bin/test.sh` puis `/usr/bin/test.sh`
    - Sinon, `bash` affiche `Command not found`

La variable d'environnement PATH

- La commande `which` indique où se trouve les commandes
  - Dans Bash, `which` ne fonctionne pas sur les alias (vus plus loin)

`which cmd` : indique le chemin complet de `cmd` en utilisant `PATH`

La variable d'environnement PATH

Attention : il est fortement déconseillé de mettre `.` dans `PATH`
(surtout si `.` est en tête du `PATH`)

- Avantage : mettre `.` dans `PATH` évite le `./` pour trouver les commandes du répertoire de travail
  (`$ script.sh au lieu de $ ./script.sh`)
- Mais n'importe quel virus/malware peut alors créer un cheval de troie en :
  - Plaçant un script nommé `ls` dans le répertoire `/tmp`
  - Attendant tranquillement que l'administrateur entre dans `/tmp`
  - Attendant ensuite que l'administrateur lance `ls` dans `/tmp`,
    => lancement du `ls` du malware avec les droits administrateurs

La malware a pris le contrôle de la machine !

Plan

- Variables notables
- Code de retour d'un processus
- Alias de commandes
- Fichier de configuration bash
- Filtrage de fichiers par motif

Code de retour d'un processus

- Un script peut renvoyer un code de retour avec `exit n`
  - Ce code de retour peut être utilisé dans les `if` et `while`

    0 ⇒ vrai (ou ok), autre ⇒ faux (ou problème)
  - Sémantique du code de retour parfois cryptique ⇒ utiliser `man`
- Code de retour dernière commande stocké dans la variable `$?`

```
$
```

```
#! /bin/bash

exit $1
```

replay.sh

Code de retour d'un processus

- Un script peut renvoyer un code de retour avec `exit n`
  - Ce code de retour peut être utilisé dans les `if` et `while`

    0 ⇒ vrai (ou ok), autre ⇒ faux (ou problème)
  - Sémantique du code de retour parfois cryptique ⇒ utiliser `man`
- Code de retour dernière commande stocké dans la variable `$?`

```
$ ./replay.sh 42
$
```

```
#! /bin/bash

exit $1
```

replay.sh

Code de retour d'un processus

- Un script peut renvoyer un code de retour avec `exit n`
  - Ce code de retour peut être utilisé dans les `if` et `while`

    0 ⇒ vrai (ou ok), autre ⇒ faux (ou problème)
  - Sémantique du code de retour parfois cryptique ⇒ utiliser `man`
- Code de retour dernière commande stocké dans la variable `$?`

```
$ ./replay.sh 42
$ echo $?
42
$
```

```
#! /bin/bash

exit $1
```

replay.sh

Code de retour d'un processus

- Un script peut renvoyer un code de retour avec `exit n`
  - Ce code de retour peut être utilisé dans les `if` et `while`

    0 ⇒ vrai (ou ok), autre ⇒ faux (ou problème)
  - Sémantique du code de retour parfois cryptique ⇒ utiliser `man`
- Code de retour dernière commande stocké dans la variable `$?`

```
$ ./replay.sh 42
$ echo $?
42
$ if ./replay.sh 0; then echo coucou; fi
coucou
$
```

```
#! /bin/bash

exit $1
```

replay.sh

Code de retour d'un processus

- Un script peut renvoyer un code de retour avec `exit n`
  - Ce code de retour peut être utilisé dans les `if` et `while`

    0 ⇒ vrai (ou ok), autre ⇒ faux (ou problème)
  - Sémantique du code de retour parfois cryptique ⇒ utiliser `man`
- Code de retour dernière commande stocké dans la variable `$?`

```
$ ./replay.sh 42
$ echo $?
42
$ if ./replay.sh 0; then echo coucou; fi
coucou
$ if ./replay.sh 1; then echo coucou; fi
$
```

```
#! /bin/bash

exit $1
```

replay.sh

Code de retour d'un processus

- Un script peut renvoyer un code de r
- C

Attention : contrairement à `bash`, dans quasiment tous les autres langages de programmation, la valeur faux vaut 0 et la valeur vrai vaut autre chose que 0

(car dans le cas 0 = faux/1 = autre, le `ou logique` est une simple addition dans $\mathbb{Z}$ et le `et logique` est une simple multiplication dans $\mathbb{Z}$)

```
$ .
$ ec
42
$ if
couc
$ if
$
```

replay.sh

Plan

- Variables notables
- Code de retour d'un processus
- Alias de commandes
- Fichier de configuration bash
- Filtrage de fichiers par motif

Alias de commande

- Sert à (re)définir le nom d'une commande
  - Pour créer des noms abrégés ou passer des options

- Création :

```
alias cmd='…'
```

- Suppression :

```
unalias cmd
```

- Consultation :

```
alias
```

Alias de commande

- Sert à (re)définir le nom d'une commande
  - Pour créer des noms abrégés ou passer des options
- Création :
  `alias cmd='…'`
- Suppression :
  `unalias cmd`
- Consultation :
  `alias`

```
$ ls
d        f1       test.sh
$
```

Alias de commande

- Sert à (re)définir le nom d'une commande
  - Pour créer des noms abrégés ou passer des options
- Création :
  `alias cmd='…'`
- Suppression :
  `unalias cmd`
- Consultation :
  `alias`

```
$ ls
d        f1       test.sh
$ alias ls='ls -a'
$
```

Alias de commande

- Sert à (re)définir le nom d'une commande
  - Pour créer des noms abrégés ou passer des options
- Création :
  `alias cmd='…'`
- Suppression :
  `unalias cmd`
- Consultation :
  `alias`

```
$ ls
d       f1      test.sh
$ alias ls='ls -a'
$ ls
.       ..      d       f1      test.sh
$
```

Alias de commande

- Sert à (re)définir le nom d'une commande
  - Pour créer des noms abrégés ou passer des options
- Création :
  `alias cmd='…'`
- Suppression :
  `unalias cmd`
- Consultation :
  `alias`

```
$ ls
d       f1      test.sh
$ alias ls='ls -a'
$ ls
.       ..      d       f1      test.sh
$ alias
alias ls='ls -a'
$
```

Alias de commande

- Sert à (re)définir le nom d'une commande
  - Pour créer des noms abrégés ou passer des options
- Création :
  `alias cmd='…'`
- Suppression :
  `unalias cmd`
- Consultation :
  `alias`

```
$ ls
d       f1      test.sh
$ alias ls='ls -a'
$ ls
.       ..      d       f1      test.sh
$ alias
alias ls='ls -a'
$ unalias ls
$
```

Alias de commande

- Sert à (re)définir le nom d'une commande
  - Pour créer des noms abrégés ou passer des options
- Création :
  `alias cmd='…'`
- Suppression :
  `unalias cmd`
- Consultation :
  `alias`

```
$ ls
d       f1      test.sh
$ alias ls='ls -a'
$ ls
.       ..      d       f1      test.sh
$ alias
alias ls='ls -a'
$ unalias ls
$ ls
d       f1      test.sh
$
```

Plan

- Variables notables
- Code de retour d'un processus
- Alias de commandes
- Fichier de configuration bash
- Filtrage de fichiers par motif

Fichiers de configuration bash

- Exécutés automatiquement au démarrage de `bash`
  - La prise en compte d'une modification de configuration impose le redémarrage de `bash` (ou l'utilisation de la **`source ~/.bashrc`**)
- Configuration
  - Globale du système d'exploitation par l'administrateur
    - Fichier `/etc/profile`
  - Pour son compte par l'utilisateur
    - Fichier `~/.bashrc` (+ d'autres fichiers non étudiés dans ce cours)
- Opérations typiquement réalisées :
  - Affectation de variables : `PATH`, `PS1`, etc.
  - Déclaration de variables liées à des logiciels installés en sus
  - Création d'alias
  - Positionnement du masque des droits d'accès
  - Etc.

Plan

- Variables notables
- Code de retour d'un processus
- Alias de commandes
- Fichier de configuration bash
- Filtrage de fichiers par motif

Filtrage de fichiers par motif (1/3)

- `Bash` peut filtrer des noms de fichiers en suivant un motif
  - `*` ⇒ une chaîne de caractères quelconque (même vide)
  - `?` ⇒ substitue **un** caractère quelconque

```
$ ls                    # contenu du répertoire
CSC3101 CSC3102 CSC3601 CSC3602 NET3101 NET3102
$
```

Filtrage de fichiers par motif (1/3)

- `Bash` peut filtrer des noms de fichiers en suivant un motif
  - `*` ⇒ une chaîne de caractères quelconque (même vide)
  - `?` ⇒ substitue **un** caractère quelconque

```
$ ls                    # contenu du répertoire
CSC3101 CSC3102 CSC3601 CSC3602 NET3101 NET3102
$ echo CSC*1            # les cours CSC se terminant par 1
CSC3101 CSC3601
$
```

Filtrage de fichiers par motif (1/3)

- `Bash` peut filtrer des noms de fichiers en suivant un motif
  - `*` ⇒ une chaîne de caractères quelconque (même vide)
  - `?` ⇒ substitue **un** caractère quelconque

```
$ ls                  # contenu du répertoire
CSC3101 CSC3102 CSC3601 CSC3602 NET3101 NET3102
$ echo CSC*1          # les cours CSC se terminant par 1
CSC3101 CSC3601
$ echo CSC?1??        # les cours CSC de semestre 1
CSC3101 CSC3102
$
```

Filtrage de fichiers par motif (2/3)

- Filtre suivant un ensemble de caractères
  - `[...]` → un caractère dans l'ensemble donné
  - `[!...]` → un caractère hors de l'ensemble donné

- Ensemble
  - Liste de caractères : `[aeiouy] [!aeiouy]`
  - Un intervalle : `[0-9] [a-zA-Z] [!A-F]`
  - Ensembles prédéfinis :
    - `[[:alpha:]]` : caractères alphabétiques
    - `[[:lower:]]` / `[[:upper:]]` : alphabet minuscule / majuscule
    - `[[:digit:]]` : chiffres décimaux `[0-9]`

Filtrage de fichiers par motif (2/3)

```
$ ls
CSC3102 CSC3501 CSC4501 CSC5001 NET3101 NET3102
$ echo CSC[45]*     # cours de 2A et 3A
CSC4502 CSC5001
```

Concepts clés

- Bash présente des variables d'environnement
  - `PATH` configure la localisation des exécutables des commandes
- Communication inter-processus avec le code de retour (`$?`)
- Alias de commandes
- Motifs pour écrire des sélections complexes de fichiers
  - `*`, `?`, `[...]`
  - Interprétés par Bash

INSTITUT Mines-Télécom

TELECOM SudParis

Les flux

CSC 3102

Introduction aux systèmes d'exploitation

Gaël Thomas

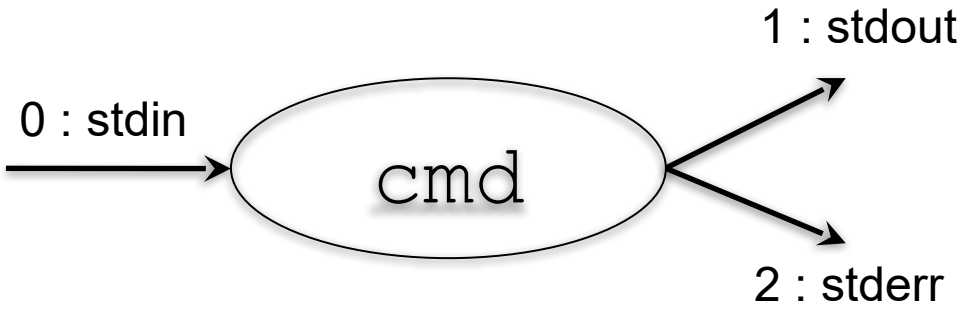
Notion de flux

- Pour accéder aux données d'un fichier (écran, clavier, fichier ordinaire…), le système d'exploitation définit une notion de flux

- Un flux est définit par :
  - Un fichier
  - Une fonction de lecture : permet d'extraire des données du flux
  - Une fonction d'écriture : permet d'ajouter des données au flux
  - Une tête de lecture/écriture : position dans le fichier pour les lectures/écritures

- Un flux est représenté par un numéro

Par défaut un processus possède 3 flux



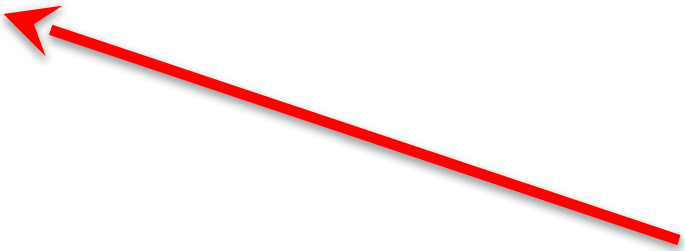
- `stdin (0)` : *standard input*
  - canal de lecture, par défaut clavier du terminal (celui de `read`)
- `stdout (1)` : *standard output*
  - canal de sortie, par défaut écran du terminal (celui d'`echo`)
- `stderr (2)` : *standard error*
  - canal de sortie pour les erreurs, par défaut écran du terminal
  - pour le moment, on n'utilise pas ce canal

Les flux par défaut du terminal

```
$ read a b
Salut tout le monde!!!
$ echo $a
Salut
$ echo $b
tout le monde!!!
$
```

Lecture à partir du clavier

Le terminal affiche les caractères saisis au clavier

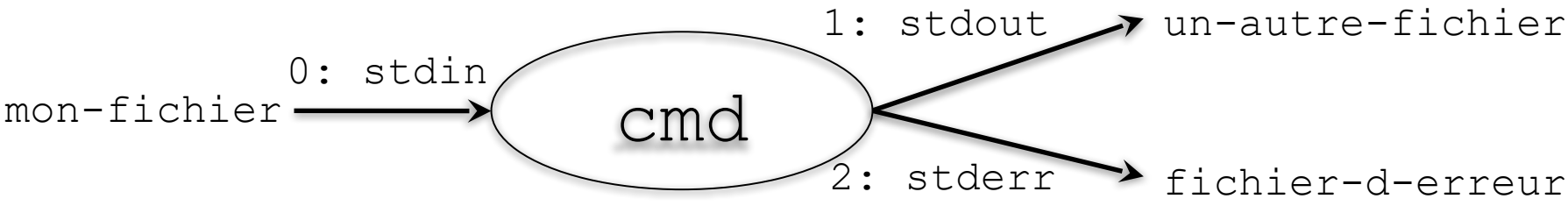


Tête de lecture/écriture dans le flux de sortie (écran associé au terminal)

1. Redirections simples
2. Redirections avancées
3. Les tubes
4. Fichiers associés aux périphériques

Redirections simples

- Toute commande peut être lancée en redirigeant les flux du processus vers un fichier



- À ce moment :
  - `echo` écrit dans `un-autre-fichier`
  - `read` lit à partir de `mon-fichier`

3 paramètres pour rediriger un flux

- Un **fichier** associé au nouveau flux
- Le **numéro** du flux à rediriger
- Un **mode** d'ouverture

	Lecture	Écriture	Tête de lecture	Remarque
<	Oui	Non	Au début	
>	Non	Oui	Au début	Ancien contenu effacé
>>	Non	Oui	À la fin	
<>	Oui	Oui	Au début	

Remarque : lors d'une ouverture en écriture,
le fichier est toujours créé s'il n'existe pas

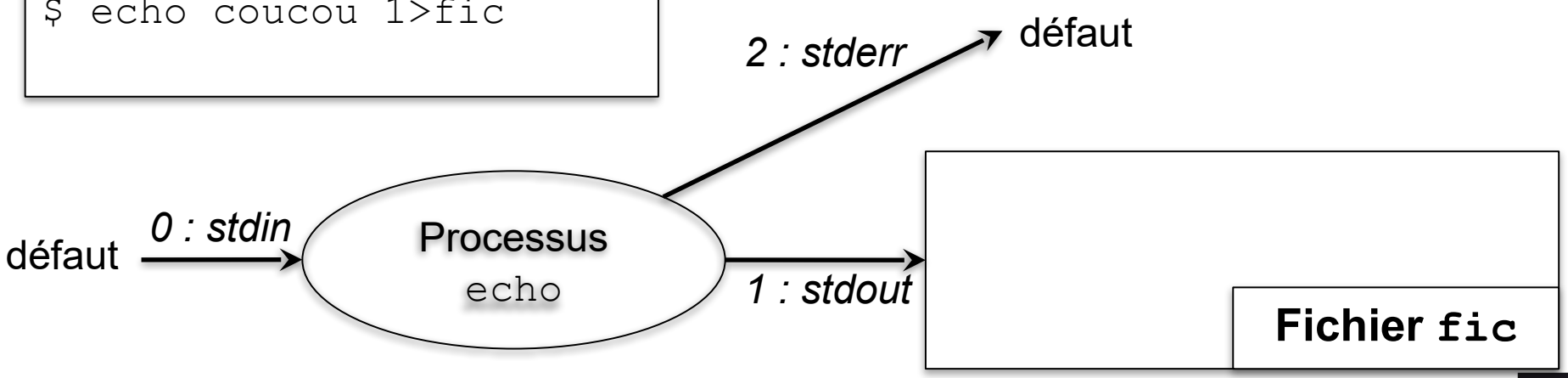
Redirection de flux

- Lancement d'une commande en redirigeant un flux

```
cmd n[<,>,>>,<>]fic
```

Lance la commande `cmd` dans un nouveau processus **après** avoir ouvert le flux numéro `n` associé au fichier `fic` avec le mode idoine

```
$ echo coucou 1>fic
```



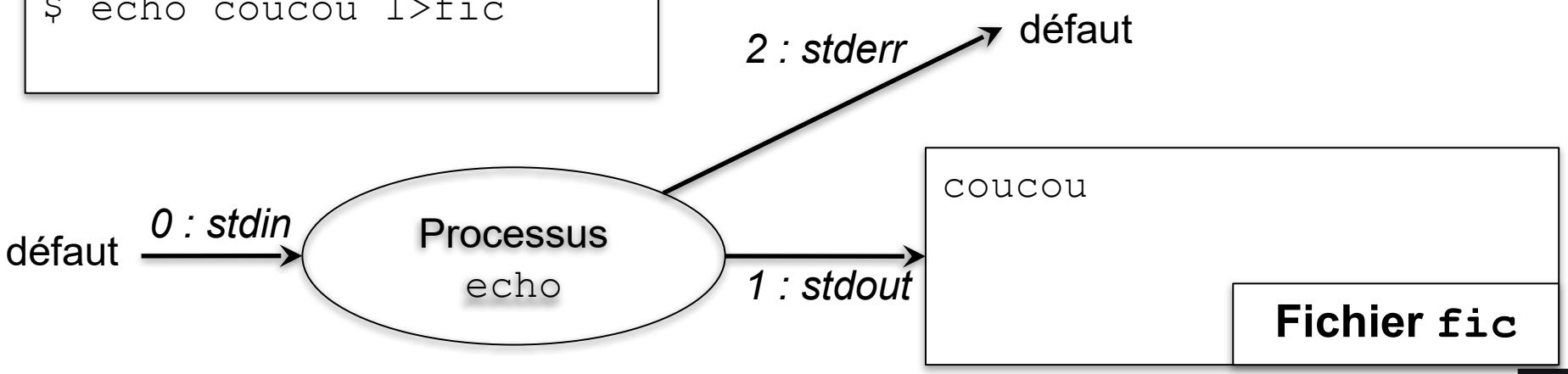
Redirection de flux

- Lancement d'une commande en redirigeant un flux

```
cmd n[<,>,>>,<>]fic
```

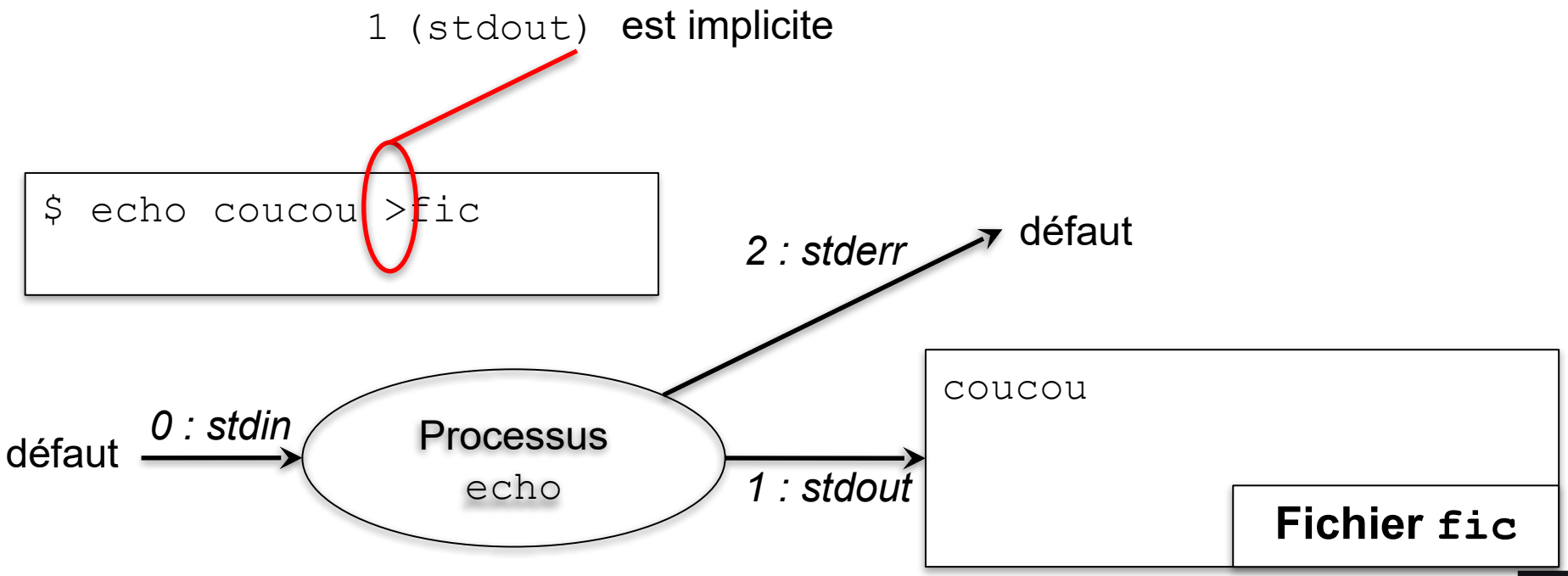
Lance la commande `cmd` dans un nouveau processus **après** avoir ouvert le flux numéro `n` associé au fichier `fic` avec le mode idoine

```
$ echo coucou 1>fic
```



Redirection de flux

- Si le numéro de flux n'est pas indiqué
  - Utilise `1 (stdout)` si en écriture
  - Utilise `0 (stdin)` si en lecture ou lecture/écriture



Redirection de flux

```
$ echo Coucou vous >fic
$
```

Exécute `echo` en redirigeant
la sortie standard
(redirection en écriture)

```
Coucou vous
```

Fichier `fic`

Redirection de flux

```
$ echo Coucou vous >fic
$ read x y <fic
$
```

Exécute `read` en redirigeant l'entrée standard (redirection en lecture)

```
Coucou vous
```

Fichier `fic`

Redirection de flux

```
$ echo Coucou vous >fic
$ read x y <fic
$ echo $x
Coucou
$
```

```
Coucou vous
```

Fichier `fic`

Redirection de flux

```
$ echo Coucou vous >fic
$ read x y <fic
$ echo $x
Coucou
$ echo $y
vous
$
```

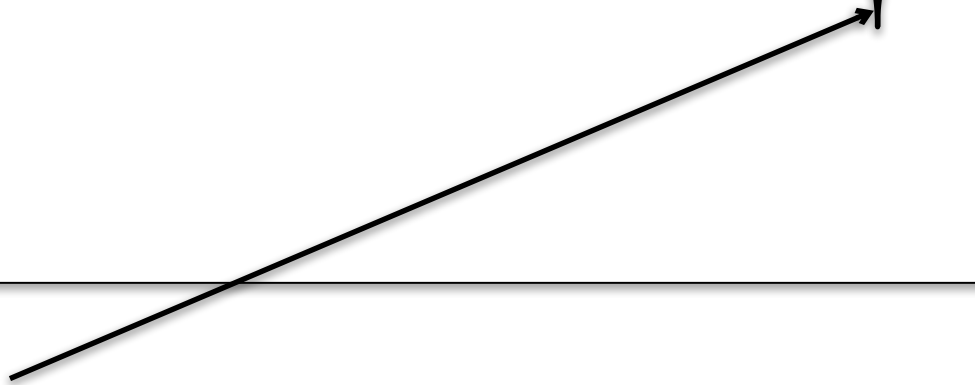
```
Coucou vous
```

Fichier `fic`

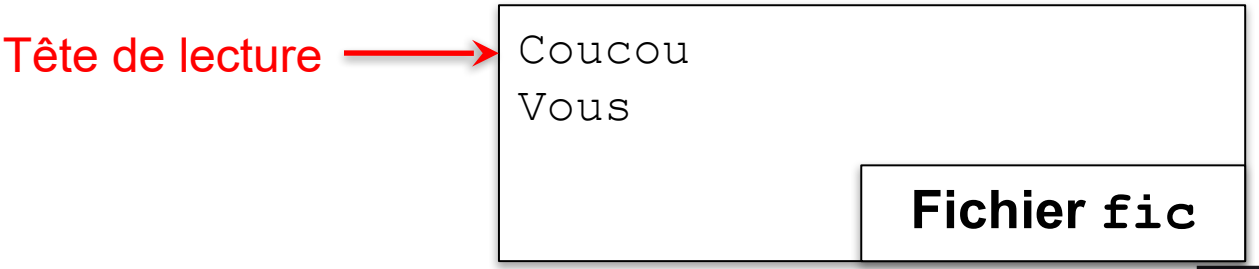
Redirection de flux et regroupement

- Toute expression `bash` peut être redirigée

```
$ ( read x; echo $x; read y; echo $y; ) <fic
```



Étape 1 : associe le flux `0 (stdin)` à `fic` en lecture

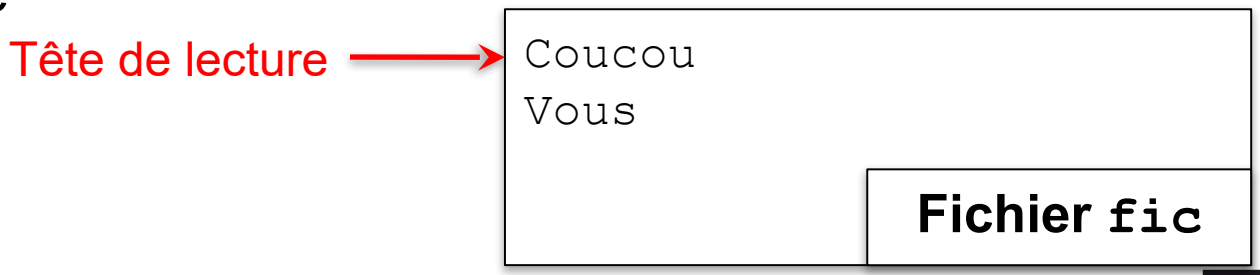


Redirection de flux et regroupement

- Toute expression `bash` peut être redirigée

```
$ ( read x; echo $x; read y; echo $y; ) <fic
```

Étape 2 : lance l'exécution du regroupement (nouveau processus, cf. CI5) en redirigeant son entrée

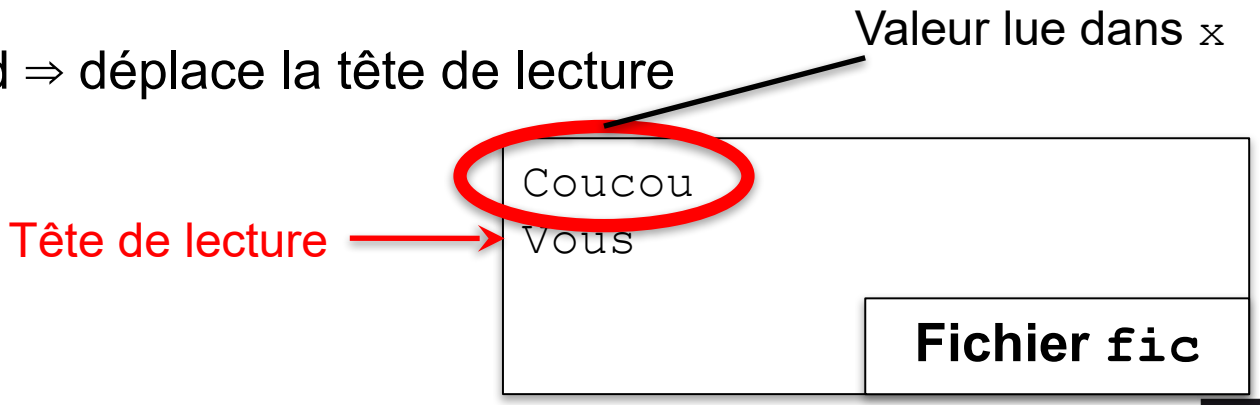


Redirection de flux et regroupement

- Toute expression `bash` peut être redirigée

```
$ ( read x; echo $x; read y; echo $y; ) <fic
```

Étape 3 : exécute read ⇒ déplace la tête de lecture

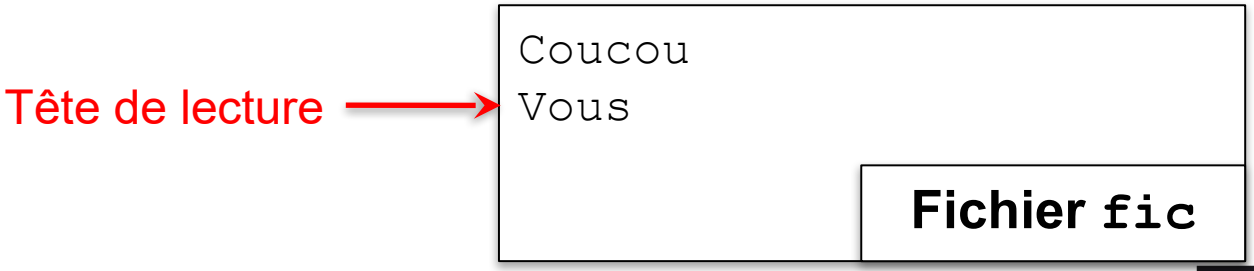


Redirection de flux et regroupement

- Toute expression `bash` peut être redirigée

```
$ ( read x; echo $x; read y; echo $y; ) <fic
Coucou
```

Étape 4 : affiche la variable `x`

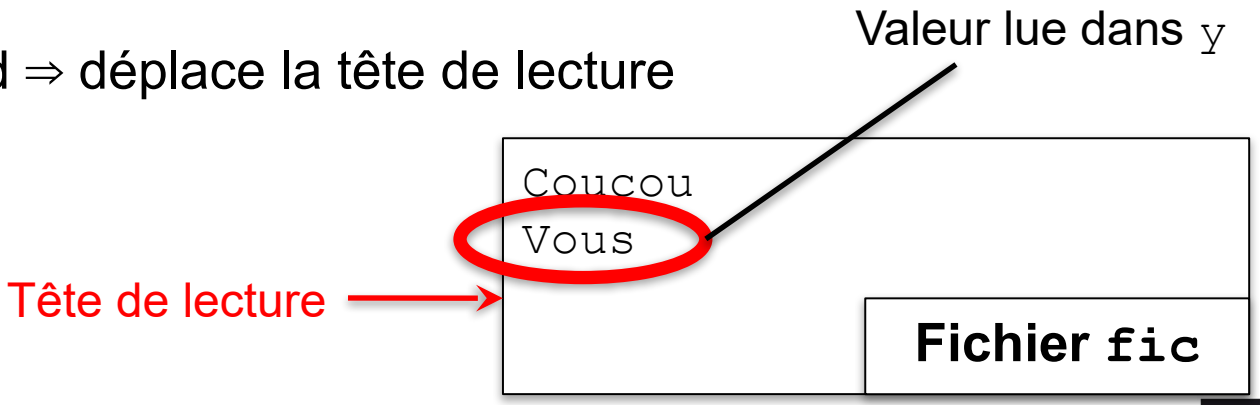


Redirection de flux et regroupement

- Toute expression `bash` peut être redirigée

```
$ ( read x; echo $x; read y; echo $y; ) <fic
Coucou
```

Étape 5 : exécute read ⇒ déplace la tête de lecture

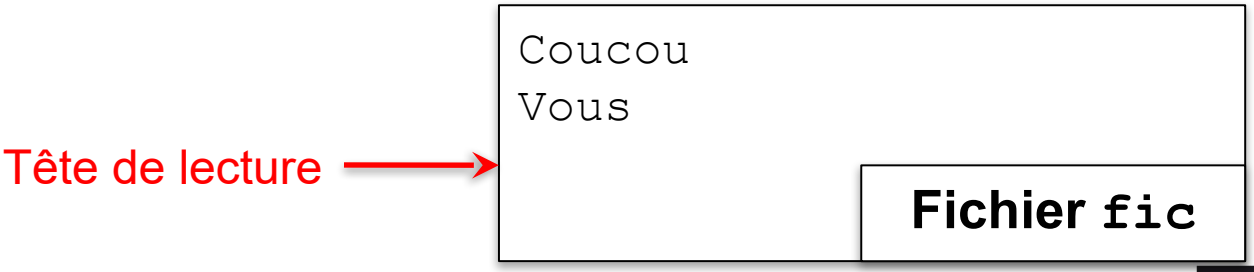


Redirection de flux et regroupement

- Toute expression `bash` peut être redirigée

```
$ ( read x; echo $x; read y; echo $y; ) <fic
Coucou
Vous
```

Étape 6 : affiche la variable `y`



Redirection de flux et regroupement

- Toute expression `bash` peut être redirigée

```
$ for x in 1 2 3; do
>   echo $x
> done >fic
```

```
1
2
3
```

Fichier `fic`

Lecture d'un flux ligne à ligne

- `read` lit une ligne d'un flux et avance la tête de lecture
  ⇒ `read` dans une boucle permet de lire un fichier ligne à ligne

- Il faut aussi détecter la fin d'un flux pour terminer la boucle
  - Fin de flux indiquée par un code EOF (end-of-file)
    - Généré sur le terminal lorsque l'utilisateur saisie `Control+d`
    - Généré automatiquement lorsque la tête de lecture atteint la fin d'un fichier
  - Lecture de EOF indiquée dans le code de retour de `read`
    - `read` retourne faux si lecture renvoie EOF
    - `read` retourne vrai sinon

Lecture d'un flux ligne à ligne

```
$
```

```
#! /bin/bash

while read line; do
  echo ":: $line"
done <fic
```

Fichier `script.sh`

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Fichier `fic`

Lecture d'un flux ligne à ligne

```
$ ./script.sh
```

Ouverture du flux associé
à `fic` avant d'exécuter la boucle

```
#! /bin/bash

while read line; do
  echo ".. $line"
done <fic
```

Fichier `script.sh`

Tête de lecture →

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Fichier `fic`

Lecture d'un flux ligne à ligne

```
$ ./script.sh
```

`read` renvoie vrai car pas fin de fichier

```
#! /bin/bash

while read line; do
  echo ":: $line"
done <fic
```

Fichier `script.sh`

Tête de lecture →

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Fichier `fic`

Lecture d'un flux ligne à ligne

```
$ ./script.sh
:: Avec ses quatre dromadaires
```

```
#! /bin/bash

while read line; do
  echo ":: $line"
done <fic
```

Fichier `script.sh`

Tête de lecture →

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Fichier `fic`

Lecture d'un flux ligne à ligne

```
$ ./script.sh
:: Avec ses quatre dromadaires
```

`read` renvoie vrai car pas fin de fichier

```
#! /bin/bash

while read line; do
  echo ":: $line"
done <fic
```

Fichier `script.sh`

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Tête de lecture →

Fichier `fic`

Lecture d'un flux ligne à ligne

```
$ ./script.sh
:: Avec ses quatre dromadaires
:: Don Pedro d'Alfaroubeira
```

```
#! /bin/bash

while read line; do
  echo ":: $line"
done <fic
```

Fichier `script.sh`

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Tête de lecture →

Fichier `fic`

Lecture d'un flux ligne à ligne

```
$ ./script.sh
:: Avec ses quatre dromadaires
:: Don Pedro d'Alfaroubeira
```

`read` renvoie faux car fin de fichier
(valeur de `line` indéfinie)

```
#! /bin/bash

while read line; do
  echo ":: $line"
done <fic
```

Fichier `script.sh`

Tête de lecture →

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Fichier `fic`

Lecture d'un flux ligne à ligne

```
$ ./script.sh
:: Avec ses quatre dromadaires
:: Don Pedro d'Alfaroubeira
$
```

Termine la boucle, ferme le flux puis termine le processus

```
#! /bin/bash

while read line; do
  echo ":: $line"
done <fic
```

Fichier `script.sh`

```
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
```

Fichier `fic`

1. Redirections simples
2. Redirections avancées
3. Les tubes
4. Fichiers associés aux périphériques

Redirections avancées

- La commande `exec` redirige les flux du processus courant (au lieu de lancer un nouveau processus)

```
exec n[<,>,>>,<>]fic
```

⇒ **ouvre** le flux `n` associé à `fic` avec le mode idoine

- Et une redirection peut se faire vers n'importe quel flux ouvert

```
cmd n[<,>,<>]&k
```

⇒ lance `cmd` en redirigeant le flux `n` vers le flux `k`

Redirections avancées

```
$ echo coucou >fic
```

```
$ exec 3>fic
$ echo coucou >&3
```

Ouverture du flux `3` en écriture vers le fichier `fic`

Redirection dans le flux `3` préalablement ouvert

Intérêt des redirections avancées

- Permet de lire et écrire dans plusieurs fichiers simultanément

```
#! /bin/bash

exec 3>redoublants  # flux 3 pour les redoublants

while read etudiant note; do
  if [ "$note" -lt 10 ]; then
    echo $etudiant >&3
  fi
  echo "Etudiant $etudiant a eu $note/20"
done <fichier-de-notes
```

script.sh

Les redoublants sont ajoutés à la fin du fichier `redoublants`

1. Redirections simples
2. Redirections avancées
3. Les tubes
4. Fichiers associés aux périphériques

Les tubes

- On peut rediriger la sortie d'une commande dans l'entrée d'une autre

  `cmd1 | cmd2`

  - Exécute `cmd1` et `cmd2` en parallèle
  - La sortie de `cmd1` est redirigée dans l'entrée de `cmd2`

- À gros grain, comportement proche de
  - `cmd1 >temp-file`
  - `cmd2 <temp-file`
  - `rm temp-file`

  (la mise en œuvre est différente et repose sur des concepts vus dans les prochains cours)

Les tubes par l'exemple

- Chaînage de deux commandes utiles (vues au cours 4)
  - `cat fic` : affiche le contenu de `fic` sur la sortie standard
  - `grep motif` : lit ligne à ligne l'entrée et n'affiche que celles qui contiennent `motif`

```
$ cat dromadaire.txt
Avec ses quatre dromadaires
Don Pedro d'Alfaroubeira
Courut le monde et l'admira.
Il fit ce que je voudrais faire
Si j'avais quatre dromadaires.
$ cat dromadaire.txt | grep Pedro
Don Pedro d'Alfaroubeira
```

1. Redirections simples
2. Redirections avancées
3. Les tubes
4. Fichiers associés aux périphériques

Redirection et fichiers de périphérique

- Le système définit un fichier par périphérique
  - Périphérique matériel connecté à l'unité centrale
    - /dev/sda : premier disque dur
    - /dev/input/mice : souris
    - …
  - Périphérique logiciel appelé pseudo-périphérique
    - /dev/null : lecture donne une chaîne vide et écriture ne fait rien
    - /dev/tty : terminal
    - /dev/urandom : générateur de nombres aléatoire
    - …

Redirection et fichiers de périphérique

- On peut utiliser les redirections avec les périphériques
  - Lecture d'une ligne à partir du générateur de nombres aléatoires

```
$ read a </dev/urandom
$ echo $a
?.ó???oJu 檄 xȆ4????F?s{?6
```

Ligne de caractères aléatoires

  - Suppression d'un affichage

```
$ echo "Je ne veux pas voir ça" >/dev/null
$
```

Concepts clés

- Un flux est la réunion de
  - Un numéro représentant le flux
  - Un mode d'ouverture (lecture/écriture, ajout/écrasement)
  - Un fichier associé au flux

- Tout flux peut être redirigé avec
  - `cmd n[<,>,>>,<>]fic` où fic est un fichier
    (la commande `exec` ouvre le flux dans le processus courant)
  - `cmd n[<,>,<>]&k` où k est un numéro de flux ouvert

- Le tube (`|`) permet de chaîner une sortie et une entrée
  `cmd1 | cmd2`

INSTITUT Mines-Télécom

TELECOM SudParis

Outils indispensables

CSC3102 – Introduction aux systèmes d'exploitation

Elisabeth Brunet

Plan

- Outils incontournables
  - Nature d'une entrée
  - Pour les fichiers texte : affichage, tri, recherche de motif
  - Occupation disque
  - Archivage de fichiers
  - Recherche de fichiers

Nature d'une entrée du système de fichiers

- Traitement applicable à un fichier dépend de sa nature
  - Est-ce un fichier texte ? Une image ? Une archive ? Un pdf ?

- Commande `file` : affiche la nature d'une entrée
  - Si texte, précise le type de codage
    - ASCII s'il n'y a que des caractères, UTF-8 si caractères accentués, etc.

```
$ file *
TP3                directory
TP3.html     exported SGML document, UTF-8 Unicode text
Ci3.pdf        PDF document, version 1.5
Ci3.pptx      Microsoft Powerpoint 2010
Notes.txt    ASCII text
Pedagogie.txt    UTF-8 Unicode text
```

Nature d'une entrée du système de fichiers

- La commande `test` teste aussi la nature d'un fichier
  (rappel `test cond` peut s'écrire `[ cond ]` avec `bash`)

  - `[ -e fichier ]` : vrai si fichier existe
  - `[ -f fichier ]` : vrai si fichier existe et est normal
  - `[ -d fichier ]` : vrai si fichier existe et est répertoire
  - `[ -L fichier ]` : vrai si fichier existe et est un lien symbolique

    (remarque : les autres tests suivent les liens symboliques)

Taille de l'occupation disque

- `df` : connaître l'état d'occupation des partitions

- `ls -lh chem …` : taille des chemins cibles
  - Si répertoire, donne la taille nécessaire au stockage de sa table d'entrées mais n'inclut pas celle de ses sous-entrées
  - Si lien symbolique, donne sa taille, i.e. l'espace nécessaire au stockage du chemin vers sa cible, ce qui correspond au nombre de caractères de ce chemin

- `du` : totalise l'occupation disque d'une entrée
  - Si répertoire, parcours récursif de son arborescence
  - Par défaut, donne le nombre de blocs occupés
    - Option `-h`, pour afficher l'équivalent de ce nombre de blocs de manière « lisible pour un humain » en o/K/M/G
    - Option `-d0` pour éviter l'affichage des tailles des sous-répertoires

Archivage

- Commande `tar` (pour **t**ape **ar**chive) ⇒ manipuler des archives

  Archive = rassemblement d'une arborescence de fichiers en un seul fichier

  - `tar -czf fic.tgz rep` : crée l'archive `fic.tgz` à partir de `rep`
  - `tar -xf fic.tgz` : extrait l'archive `fic.tgz`
  - `tar -tf fic.tgz` : liste le contenu de l'archive `fic.tgz`

  - Option `-c chem`, pour créer l'archive à partir du chemin `chem`
  - Option `-v`, pour un affichage en mode verbeux
  - Option `-z`, pour une compression des données au format gzip
  - Option `-f nom.tgz`, pour préciser le nom de l'archive voulue
    - Par convention, extension `.tgz` ou `.tar.gz`
  - Option `-x`, pour extraire (`-z`, pour la décompression via gzip)
    ⇒ décompression dans le répertoire courant
  - Option `-t`, pour lister

Affichage d'un fichier en mode texte

- Consultation du contenu d'un fichier ordinaire

- `more fichier`
- `less fichier`

} affichage simple page par page

- `head -n k <fichier>` : affichage des `k` premières lignes
- `tail -n k <fichier>` : affichage des `k` dernières lignes

- `cat fic1 fic2…` : affiche la concaténation des fichiers indiqués

- `wc fic` : compte les lignes, mots et caractères du fichier
  - Option `-l`, uniquement les lignes ; `-w`, les mots ; `-c`, les caractères

À propos de `cat` et des commandes qui suivent

- Pour les commandes qui suivent : si aucun fichier n'est donné en argument, la commande s'applique sur l'entrée standard
  - Rappel : `ctl-d` génère un code de fin de fichier (`EOF`)

- Par exemple :
  - `cat fic` : affiche le contenu de `fic`
  - `echo coucou | cat` : affiche `coucou`
  - `cat > fic` : écrit ce qui est tapé en entrée standard dans fic

Extraire des parties de lignes

- `cut -c plage fic` : extrait des caractères de chaque ligne de `fic`
  - `plage` : `num` ou `num1, num2, …` ou `num1-num2`

  Exemple : `cut -c3-7 fic.txt`
  ⇒ extrait les caractères 3 à 7 de `fic.txt`

- `cut -d car -f plage fic` : extraits des champs
  - `-d car` : `car` = séparateur de champs (tabulation par défaut)
  - `plage` comme avec `-c`

  Exemple : `cut -d' ' -f2,4 fic.txt`
  ⇒ extrait les 2ième et 4ième mots de chaque ligne de `fic.txt`

Supprimer ou transformer des caractères

- `tr s1 s2` : transforme chaque caractère de l'ensemble `s1` en ceux de l'ensemble `s2`

  (la commande ne prend pas de fichier en argument : toujours à partir de `stdin`)

  - Exemple : `cat fic | tr '\n ' 'ab'`
    ⇒ transforme les retours à la ligne en a et les espaces en b
  - Exemple : `cat fic | tr '\n ' 'a'`
    ⇒ transforme les retours à la ligne et espaces en a

- `tr -d s` : élimine chaque caractère de la chaîne s
  - Exemple : `cat fic | tr -d 'aeiouy'`
    ⇒ élimine les voyelles de `fic`

Trier les lignes de fichiers texte

- `sort fic…`
  - Par défaut, tri lexicographique
    - Option `-n` pour un tri numérique
  - Par défaut, tri appliqué en tenant compte de toute la ligne
    - Option `-k x,x` pour un tri selon le champs `x`
      - `sort -k 2,2 fic` : tri selon le 2ème champ de chaque ligne
      - `sort -k 2,2 -k 3,3 fic` : tri selon les 2ème et 3ème champs
      - Remarque : pour un tri non numérique, `-k x,y` pour champs `x` à `y`
  - Par défaut, le séparateur de champs est l'espace
    - Option `-t <caractère>` pour changer le séparateur
  - Option `-r` pour inverser l'ordre du tri appliqué
  - Peut s'appliquer sur un ensemble de fichiers
  - D'autres options à consulter dans la page du manuel

Recherche d'un motif dans du texte (1/3)

- `grep motif fic1 fic2 …`
  - Affiche les lignes des fichiers qui contiennent le motif
    - Peut aussi lire l'entrée standard : `cat fic | grep motif`
  - Le motif est une expression régulière (ou rationnelle)
    - `grep` = *global regular expression print*
  - Pour CSC3102, seul un sous-ensemble d'expressions régulières GNU
    - Chaînes de caractères
    - Attention ! Les méta-caractères de `grep` sont différents de ceux de `bash` !
      - `.` : n'importe quel caractère
      - `*` : répétition du caractère précédent, 0 ou plusieurs fois
      - `[…]` (/ `[^…]`) : met en correspondance un caractère de (/hors) l'ensemble
      - `^` / `$` : ancre le motif au début / à la fin de la ligne
    - **Attention : mettre le motif entre guillemets simples (« ' »)**

Recherche d'un motif dans du texte (2/3)

- Options :
  - `-v` : inverse le motif (affiche les lignes qui ne le contiennent pas)
  - `-r` : traite récursivement les fichiers dans le dossier passé en argument
  - `-i` : ignore la casse
  - `-q` : n'affiche rien, seul le code de retour indique le résultat
    - Utile pour seulement tester la présence du motif
    - Code de retour `0` ⇔ motif trouvé
- D'autres options à consulter dans la page du manuel

Recherche d'un motif dans du texte (3/3)

- Exemples :
  - `grep warning *.log`
    - Affiche les lignes contenant `warning` de tous les fichiers .log du dossier courant
  - `grep -i warning *.log`
    - Comme ci-dessus, mais en ignorant la casse
  - `grep -v '^[mn]' fic`
    - Affiche les lignes de `fic` ne commençant pas par `m` ou `n`
  - `grep '(.*)$' fic`
    - Afficher les lignes qui se terminent par des caractères quelconques entre parenthèses

Recherche dans une arborescence

- `find` : recherche des entrées satisfaisants un ensemble de critères de sélection dans une arborescence
  - Parcourt récursivement et teste les critères sur chaque entrée
  - `find rep_de_recherche liste des critères`
    - `-name "<motif>"` : précise le nom des entrées à rechercher
      - `<motif>` est motif conforme à `bash` à l'exception des crochets `[ … ]`
      - **Attention : mettre le motif entre guillemets (« " »)**
    - `-type <type>` : précise le type des entrées à rechercher
      - `f` : fichier normal ; `d` : dossier ; `l` : lien symbolique
    - `-print` : permet l'affichage des résultats (par défaut)
    - Exemple : `find . -name core -print`

      → affiche les chemins des entrées nommées `core` de mon répertoire courant

      `find /usr -name "*.c" -print`

      → affiche les chemins des entrées dont le nom se terminent par `.c` sous `/usr`

Conclusion

- Commandes clés :
  - `more`, `less`, `head`, `tail`, `cat`, `wc`
  - `cut`, `tr`, `sort`, `grep`
  - `df`, `du`, `ls -lh`
  - `tar`
  - `find`,

En route pour le TP!

INSTITUT Mines-Télécom

TELECOM SudParis

Petit bilan à mi-module

CSC3102 – Introduction aux systèmes d'exploitation

Gaël Thomas

Le langage bash

- Des variables : `x=42; echo $x`
- Des structures algorithmiques : `if, for, while`
- Des paramètres : `shift, "$@", "$0", "$1", "$2"...`
- Des codes de retour : `exit n`
- Des imbrications de commandes : `x=$(expr $x + 1)`

Des commandes de base

- Lecture/écriture :
  - `echo, read`
- Le système de fichier :
  - `ls, rm, cp, mv, ln,`
  - `find, df, du, tar`
- Le contenu d'un fichier :
  - `cat, grep, cut, sort, tr`
- Le calcul :
  - `expr`

Des commandes de base

- Lecture/écriture :
  - `echo, read`
- Le système de fichier :
  - `ls, rm, cp, mv, ln,`
  - `find, df, du, tar`
- Le contenu d'un fichier :
  - `cat, grep, cut, sort, tr`
- Le calcul :
  - `expr`

Nota bene :

`expr` affiche son résultat sur la sortie standard

```
$ expr 1 + 2
3
$
```

Des commandes de base

- Lecture/écriture :
  - `echo, read`
- Le système de fichier :
  - `ls, rm, cp, mv, ln,`
  - `find, df, du, tar`
- Le contenu d'un fichier :
  - `cat, grep, cut, sort, tr`
- Le calcul :
  - `expr`

Nota bene :

`expr` affiche son résultat sur la sortie standard

```
$ expr 1 + 2
3
$ x=$(expr 1 + 2)
$
```

Des commandes de base

- Lecture/écriture :
  - `echo, read`
- Le système de fichier :
  - `ls, rm, cp, mv, ln,`
  - `find, df, du, tar`
- Le contenu d'un fichier :
  - `cat, grep, cut, sort, tr`
- Le calcul :
  - `expr`

Nota bene :

`expr` affiche son résultat sur la sortie standard

```
$ expr 1 + 2
3
$ x=$(expr 1 + 2)
$ echo $x
3
$
```

Interprétation de commandes

- Quand `bash` interprète une commande, il exécute, dans l'ordre :
  - Analyse « déclaration de variables » – « commande et arguments » – « redirections »
  - Substitution des variables et des motifs
  - Puis ouverture des flux si redirections
  - Puis exécution de la commande

Les entrées/sorties

- Des redirections
  - `echo coucou >fic`
  - `read line <fic`
  - `exec 3>fic; echo coucou >&3`
- Des tubes anonymes
  - `cat /etc/passwd | grep root | cut -d':' -f3`

Les entrées/sorties

- Des redirections
  - `echo coucou >fic`
  - `read line <fic`
  - `exec 3>fic; echo coucou >&3`

```
$ ls
bap     bip
$
```

Les entrées/sorties

- Des redirections
  - `echo coucou >fic`
  - `read line <fic`
  - `exec 3>fic; echo coucou >&3`

```
$ ls
bap     bip
$ ls -l >plop
$
```

Le flux, et donc le fichier `plop`, sont créés avant de lancer la commande `ls`

Les entrées/sorties

- Des redirections
  - `echo coucou >fic`
  - `read line <fic`
  - `exec 3>fic; echo coucou >&3`

```
$ ls
bap     bip
$ ls -l >plop
$ cat plop
total 96
-rw-r--r--  1 gthomas  staff   5925 11 oct 16:38 bap
-rwxr-xr-x  1 gthomas  staff  38512 11 oct 16:38 bip
-rw-r--r--  1 gthomas  staff      0 11 oct 16:39 plop
$
```

Le flux, et donc le fichier `plop`, sont créés avant de lancer la commande `ls`

⇒ `plop` apparaît dans `plop` !

Les entrées/sorties

- Des redirections
  - `echo coucou >fic`
  - `read line <fic`
  - `exec 3>fic; echo coucou >&3`

```
$ ls
bap     bip
$ ls -l >plop
$ cat plop
total 96
-rw-r--r--  1 gthomas  staff   5925 11 oct 16:38 bap
-rwxr-xr-x  1 gthomas  staff  38512 11 oct 16:38 bip
-rw-r--r--  1 gthomas  staff      0 11 oct 16:39 plop
$ echo * >plip
$ cat plip
bap bip plop
```

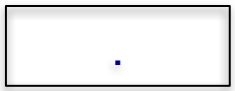
Le motif est évalué avant la redirection

⇒ `plip` n'apparaît pas dans `plip` !

Motifs `bash`, motifs de commandes

- `echo rep/19[7-9][[:digit:]][-_][[:upper:]]*`

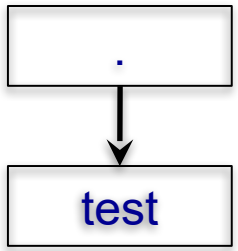
```
$ ls
$
```



Motifs `bash`, motifs de commandes

- `echo rep/19[7-9][[:digit:]][-_][[:upper:]]*`

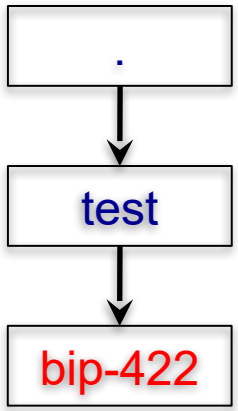
```
$ ls
$ mkdir test
$
```



Motifs `bash`, motifs de commandes

- `echo rep/19[7-9][[:digit:]][-_][[:upper:]]*`

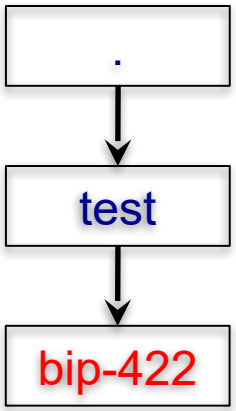
```
$ ls
$ mkdir test
$ touch test/bip-422
$
```



Motifs `bash`, motifs de commandes

- `echo rep/19[7-9][[:digit:]][-_][[:upper:]]*`

```
$ ls
$ mkdir test
$ touch test/bip-422
$ find . -name bip*
./test/bip-422
$
```

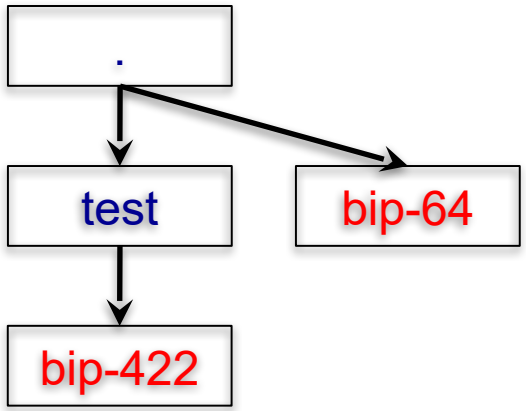


`bash` commence par chercher le motif `bip*`
⇒ pas de correspondance trouvée dans le répertoire courant
⇒ `bash` conserve `bip*`
⇒ `bash` exécute `find` avec comme paramètre `bip*`

Motifs `bash`, motifs de commandes

- `echo rep/19[7-9][[:digit:]][-_][[:upper:]]*`

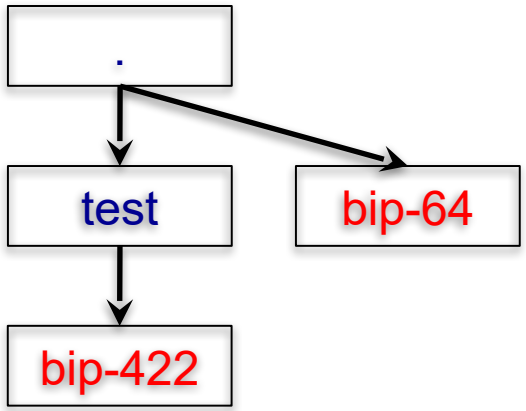
```
$ ls
$ mkdir test
$ touch test/bip-422
$ find . -name bip*
./test/bip-422
$ touch bip-64
$
```



Motifs `bash`, motifs de commandes

- `echo rep/19[7-9][[:digit:]][-_][[:upper:]]*`

```
$ ls
$ mkdir test
$ touch test/bip-422
$ find . -name bip*
./test/bip-422
$ touch bip-64
$ find . -name bip*
./bip-64
$
```

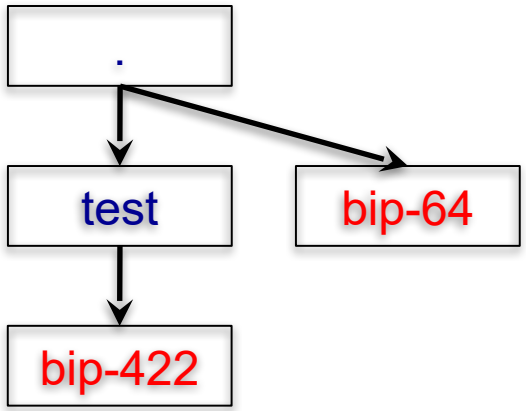


`bash` commence par chercher le motif `bip*`
⇒ une correspondance trouvée dans le répertoire courant
⇒ `bash` transforme `bip*` en `bip-64`
⇒ `bash` execute `find` avec comme paramètre `bip-64` !

Motifs `bash`, motifs de commandes

- `echo rep/19[7-9][[:digit:]][-_][[:upper:]]*`

```
$ ls
$ mkdir test
$ touch test/bip-422
$ find . -name bip*
./test/bip-422
$ touch bip-64
$ find . -name bip*
./bip-64
$ find . -name "bip*"
./bip-64
./test/bip-422
$
```



`bash` n'interpère pas le motif `bip*` car entre guillements

⇒ `bash` execute `find` avec comme paramètre `bip*`

INSTITUT Mines-Télécom

TELECOM SudParis

Les Processus

CSC3102 – Introduction aux systèmes d'exploitation

François Trahay & Gaël Thomas

Présentation du cours

- Contexte :
  - Des dizaines de processus s'exécutent simultanément sur une machine
- Objectifs :
  - Savoir observer les processus s'exécutant sur une machine
  - Manipuler un processus en cours d'exécution
  - Comprendre comment sont ordonnancés les processus
- Notions clés :
  - Arborescence de processus, états d'un processus, ordonnancement

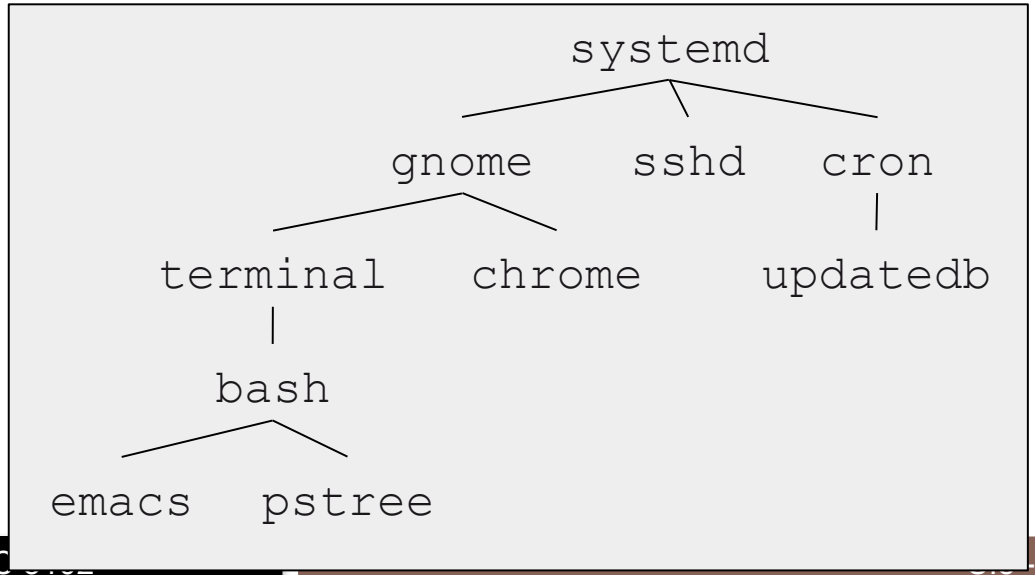
Notion de processus

- Processus = programme en cours d'exécution
  - Un espace mémoire + contexte d'exécution (fichiers ouverts, etc.)
- Caractéristiques statiques
  - PID : Process Identifier (identifie le processus)
  - PPID : Parent Processus Identifier (identifie le parent)
  - Utilisateur propriétaire
  - Droits d'accès aux ressources (fichiers, etc.)
- Caractéristiques dynamiques
  - Priorité, environnement d'exécution, etc.
  - Quantité de ressources consommées (temps CPU, etc.)

1. Observer un processus
2. Processus en avant et arrière plan
3. Cycle de vie d'un processus
4. Variables et processus
5. Gestion des processus dans le système d'exploitation

Arborescence de processus

- Chaque processus possède un processus parent
  - Sauf le premier processus (`systemd` ou `init`, PID=1)
    ⇒ arborescence de processus
- Deux types de processus :
  - Processus utilisateurs (attachés à un terminal)
  - Daemons : processus qui assurent un service (détachés de tout terminal)



Observer les processus

- `ps` : affiche les processus s'exécutant à un instant donné

```
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 22995  1403  0  80   0 -  6285 -      pts/1    00:00:00 bash
0 S  1000 29526 22995  0  80   0 - 128631 -     pts/1    00:00:05 emacs
0 S  1000 29826 22995  0  80   0 - 51571 -      pts/1    00:00:00
oosplash
0 S  1000 29843 29826  1  80   0 - 275029 -     pts/1    00:00:48
soffice.bin
0 R  1000 30323 22995  0  80   0 -  2790 -      pts/1    00:00:00 ps
```

  - `ps PID` : affiche les information du processus avec ce `PID`

Observer les processus (suite)

- `pstree` : affiche l'arborescence des processus

```
$ pstree -pA
systemd(1)-+-ModemManager(535)-+-{gdbus}(675)
           |                   `-{gmain}(580)
           |-NetworkManager(552)-+-dhclient(27331)
           |                     |-{NetworkManager}(673)
           |                     |-{gdbus}(756)
           |                     `-{gmain}(733)
           |-acpid(692)
           |-konsole(1403)-+-bash(22995)-+-emacs(29526)-+-{dconf worker}(29529)
                           |             |              |-{gdbus}(29528)
                           |             |              `-{gmain}(29527)
                           |             `-pstree(30412)
                           `-{QProcessManager}(1411)
```

Observer les processus (suite)

- `top` : affiche dynamiquement des processus

```
$ top
top - 15:52:18 up 5 days,  2:04,  3 users,  load average: 0,19, 0,12, 0,13
Tasks: 176 total,   1 running, 175 sleeping,   0 stopped,   0 zombie
%Cpu(s):  6,0 us,  1,3 sy,  0,1 ni, 92,5 id,  0,1 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   8099392 total,  5840956 used,  2258436 free,   494524 buffers
KiB Swap: 10157052 total,        0 used, 10157052 free.  3114404 cached Mem
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  866 root      20   0  731892 377196 346672 S   6,4  4,7  21:01.97 Xorg
 1375 trahay     9 -11  651480  11108   8052 S   6,4  0,1  23:23.48 pulseaudio
    1 root      20   0  176840   5420   3144 S   0,0  0,1   0:02.57 systemd
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.01 kthreadd
    3 root      20   0       0      0      0 S   0,0  0,0   0:04.34 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0,0  0,0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S   0,0  0,0   0:30.37 rcu_sched
```

Variables relatives aux processus

- Chaque processus `bash`, y compris les scripts, définissent :
  - **`$$`** : `PID` du `bash` courant
  - **`$PPID`** : `PID` du parent du `bash` courant

```
$ echo $$
20690
$ echo $PPID
20689
$
```

Variables relatives aux processus

- Chaque processus `bash`, y compris les scripts, définissent :
  - **`$$`** : `PID` du `bash` courant
  - **`$PPID`** : `PID` du parent du `bash` courant

```
$ echo $$
20690
$ echo $PPID
20689
$ ps -p 20689,20690
  PID TTY           TIME CMD
20689 ??         0:11.69 xterm -r
20690 ttys004    0:01.32 bash
$
```

Détail d'un processus

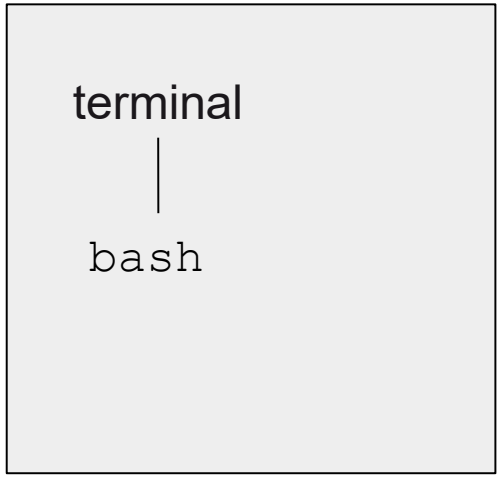
- `/proc/$PID/` contient :
  - cmdline : texte de la ligne de commande ayant lancé le processus
  - exe : lien vers le fichier exécutable du programme
  - environ : contenu de l'environnement
  - fd : liens vers les fichiers ouverts
  - …

```
$ ls /proc/29526
attr          coredump_filter  gid_map     mountinfo   oom_score      sessionid  task
autogroup     cpuset           io          mounts      oom_score_adj  smaps      timers
auxv          cwd              limits      mountstats  pagemap        stack      uid_map
cgroup        environ          loginuid    net         personality    stat       wchan
clear_refs    exe              map_files   ns          projid_map     statm
cmdline       fd               maps        numa_maps   root           status
comm          fdinfo           mem         oom_adj     sched          syscall
```

1. Observer un processus
2. Processus en avant et arrière plan
3. Cycle de vie d'un processus
4. Variables et processus
5. Gestion des processus dans le système d'exploitation

Processus en avant-plan

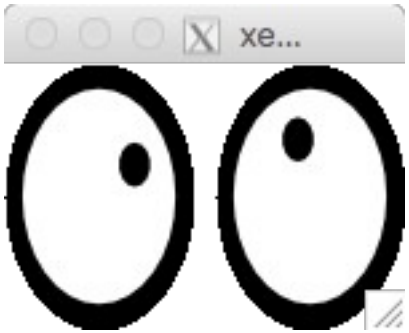
- Par défaut, une commande s'exécute en avant-plan (en anglais, *foreground*)
  - `bash` crée un processus enfant et attend qu'il termine
  - Le processus enfant exécute le programme

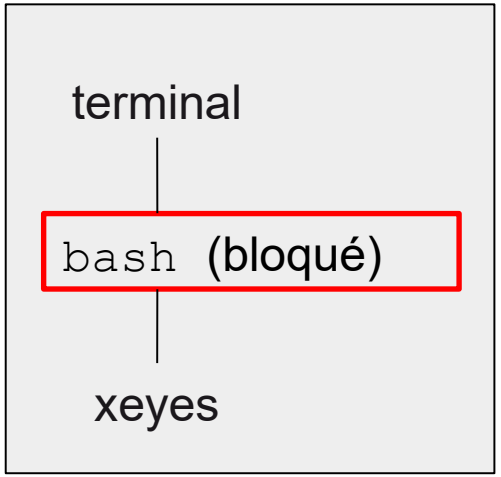


Processus en avant-plan

- Par défaut, une commande s'exécute en avant-plan (en anglais, *foreground*)
  - `bash` est bloqué tant que le processus fils s'exécute

```
$ xeyes
```

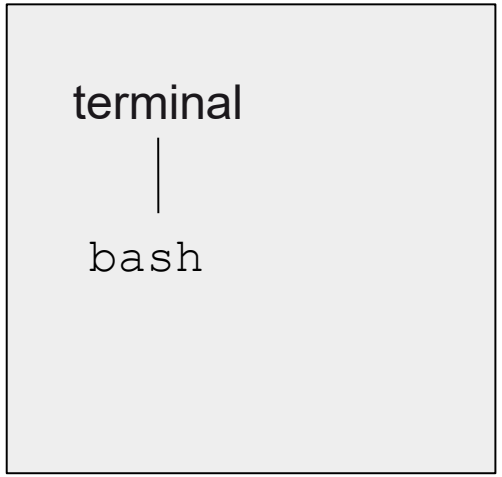




Processus en avant-plan

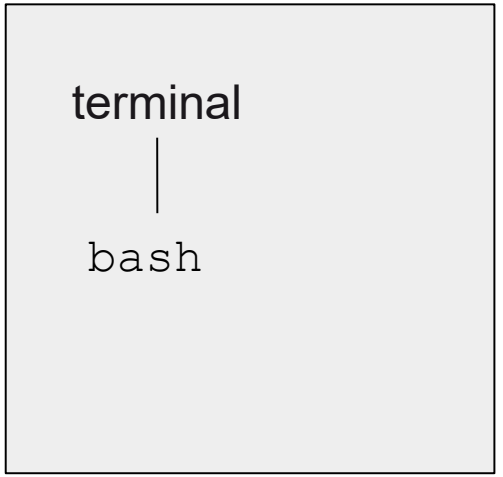
- Par défaut, une commande s'exécute en avant-plan (en anglais, *foreground*)
  - Quand le processus fils se termine, `bash` reprend son exécution

```
$ xeyes
$
```



Processus en arrière-plan

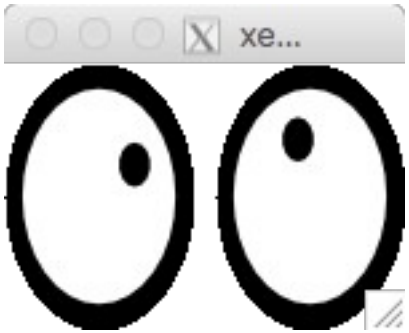
- Pour exécuter une commande arrière-plan (en anglais, *background*)
  - Terminer la commande par « `&` »

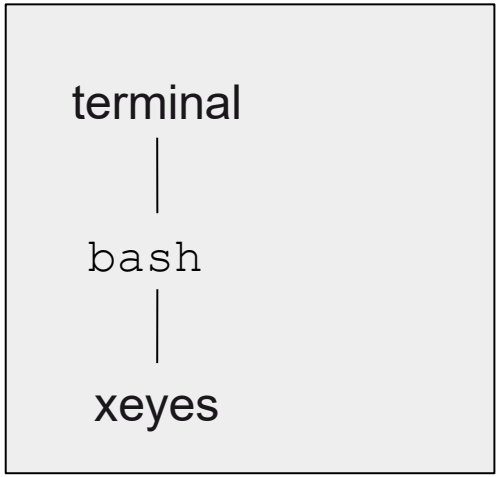


Processus en arrière-plan

- Commande en arrière-plan (en anglais, *background*)
  - `bash` crée un enfant et n'attend pas qu'il se termine
  - `bash` affiche le numéro de job (JobID) et le PID du fils
  - Le processus enfant exécute le programme

```
$ xeyes &
[1] 35794
$
```

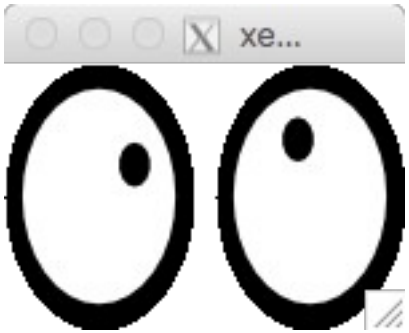


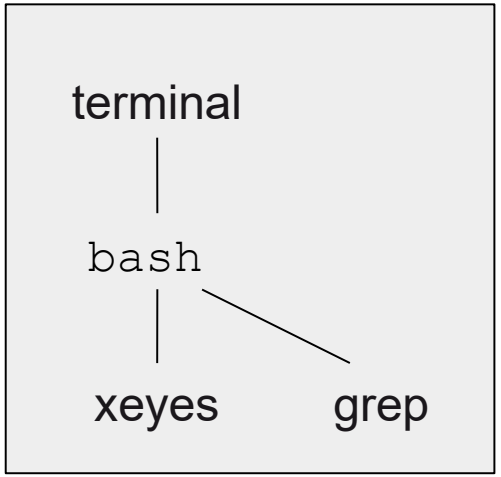


Processus en arrière-plan

- Commande en arrière-plan (en anglais, *background*) :
  - `bash` et le processus fils s'exécutent en parallèle
  - `bash` peut donc exécuter d'autres commandes

```
$ xeyes &
[1] 35794
$ grep c bjr.txt
coucou
```

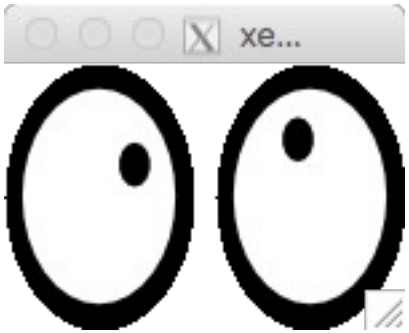


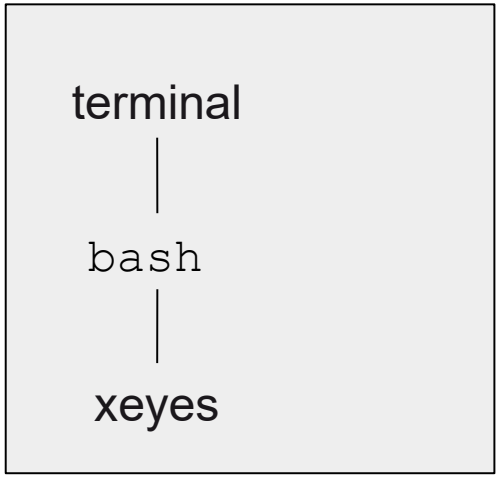


Processus en arrière-plan

- Commande en arrière-plan (en anglais, *background*) :
  - `bash` et le processus fils s'exécutent en parallèle
  - `bash` peut donc exécuter d'autres commandes

```
$ xeyes &
[1] 35794
$ grep c bjr.txt
coucou
$
```



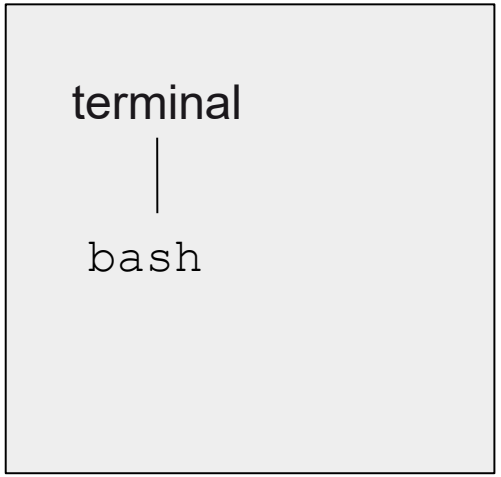


Processus en arrière-plan

- Commande en arrière-plan (background) :
  - Quand le fils se termine, le système d'exploitation informe `bash`

```
$ xeyes &
[1] 35794
$ grep c bjr.txt
coucou
$
[1]+  Done    xeyes
$
```

JobID



PID du dernier processus lancé

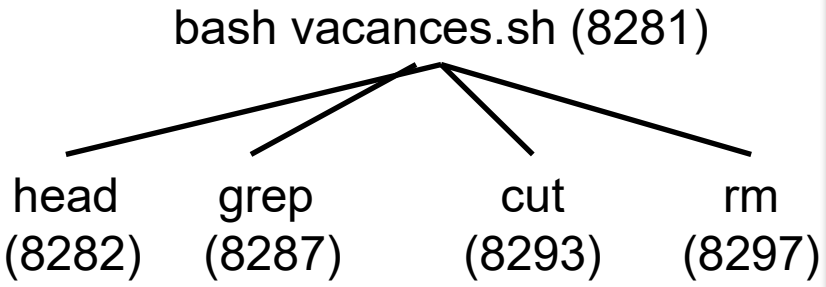
- Le PID du dernier processus lancé en arrière-plan est dans la variable `$!`

```
$ xeyes &
[1] 35794
$ xeyes &
[2] 35795
$ echo $!
35795
$ echo $!
35795
```

1. Observer un processus
2. Processus en avant et arrière plan
3. Cycle de vie d'un processus
4. Variables et processus
5. Gestion des processus dans le système d'exploitation

Commandes et processus

- Chaque commande crée un processus
  - Sauf pour les commandes internes qui sont directement interprétées par `bash` (`exit`, `source`...)



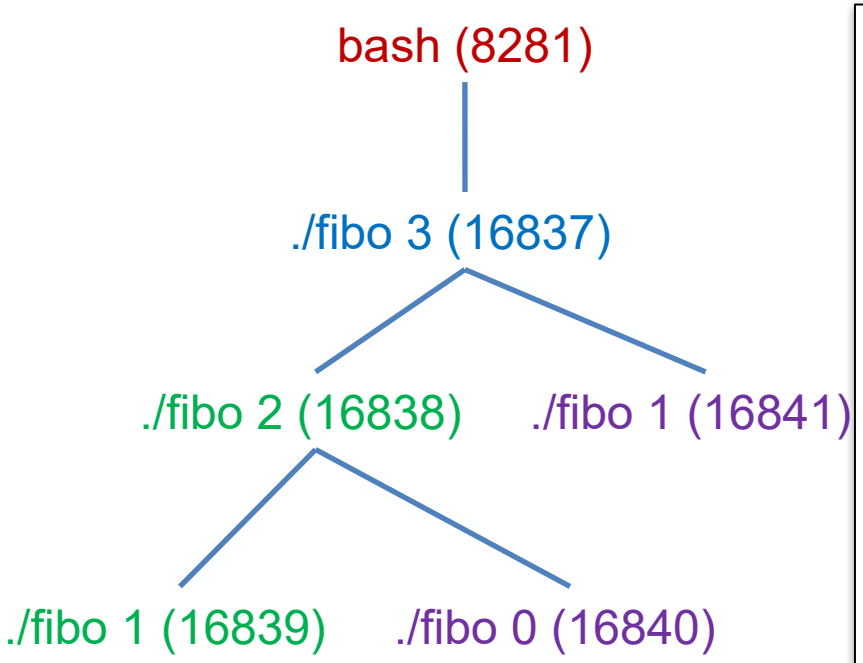

```
#! /bin/bash

head -n 30 itineraire > debut_iti
grep plage debut_iti> baignade
cut -d' ' -f3 baignade > tresor
rm baignade
```

vacances.sh

Scripts et processus

- Par défaut, un script est lancé dans un processus enfant




```
#! /bin/bash
if [ $1 -eq 0 ] || [ $1 -eq 1 ];
then
    echo 1
else
    n=$1
    fib1=$(./fibo $(expr $n - 1))
    fib2=$(./fibo $(expr $n - 2))
    echo $(expr $fib1 + $fib2 )
fi
```

fibo

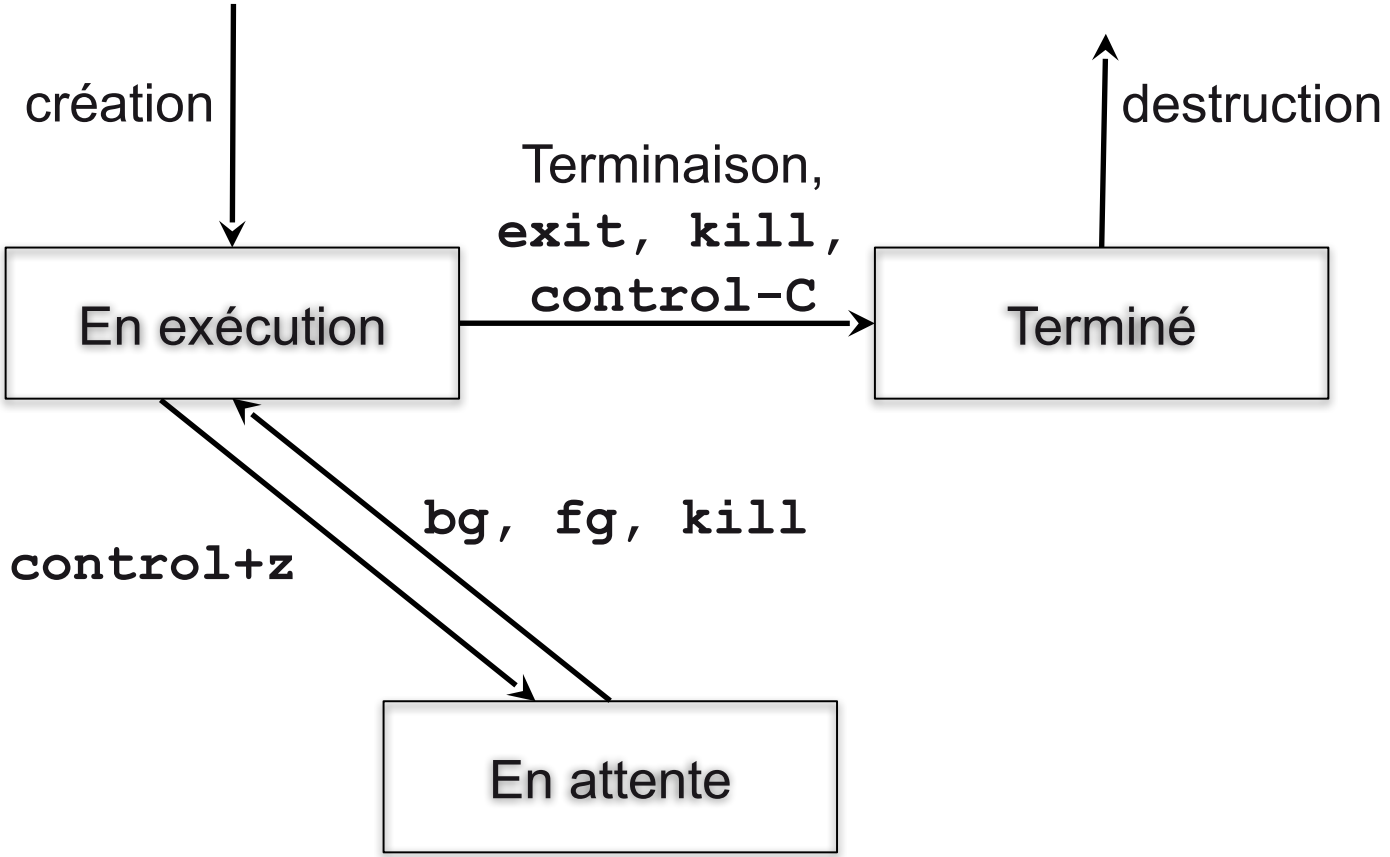
Suspendre un processus

- Suspendre un processus en avant-plan : `control+z`
  - Le processus est placé « en attente »
- Reprendre un processus en attente
  - Pour le mettre en avant-plan : `fg` (*foreground*)
    - `fg N` : mettre en avant-plan le job `N`
  - Pour le mettre en arrière-plan : `bg` (*background*)
    - `bg N` : mettre en arrière-plan le job `N`

Suppression d'un processus

- Un processus se termine s'il atteint sa dernière instruction
- Ou s'il appelle `exit`
- Ou s'il reçoit un signal (voir CI6)
  - `control-c` : tue le processus en avant plan (avec `SIGINT`)
  - `kill` ou `killall` : tue un processus (avec `SIGTERM`)
    - `kill %JobID` : tue le processus de numéro de job `JobID`
    - `kill PID` : tue le processus d'identifiant `PID`
    - `killall prog` : tue tous les processus dont le chemin du programme est `prog`
  - Remarque : vous verrez en CI6 que les processus peuvent résister à `control-c`, `kill` ou `killall`. Si c'est le cas, ajoutez -9 (`SIGKILL`) après `kill`/`killall` pour forcer leur mort

États d'un processus



Attendre la fin d'un processus

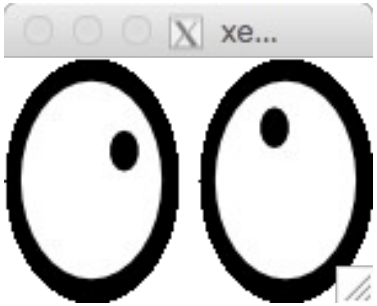
- La commande `wait` permet d'attendre la fin d'un fils
  - `wait` sans argument : attend la fin de tous les fils
  - `wait %jobid1 %jobid2...` ou `wait pid1 pid2...` : attend la fin des processus passés en argument

Attendre la fin d'un processus

```
$
```

Processus en exécution

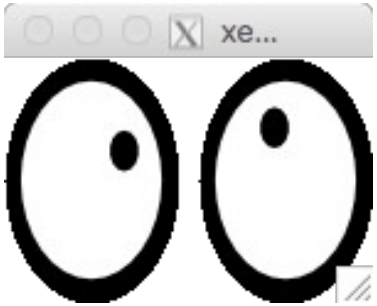
Attendre la fin d'un processus




```
$ xeyes &
$
```

Processus en exécution — → Création de processus

Attendre la fin d'un processus



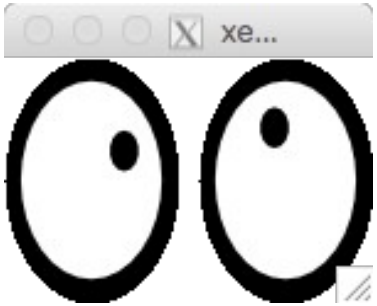

```
$ xeyes &
$ grep gthomas /etc/passwd
```

Processus en exécution

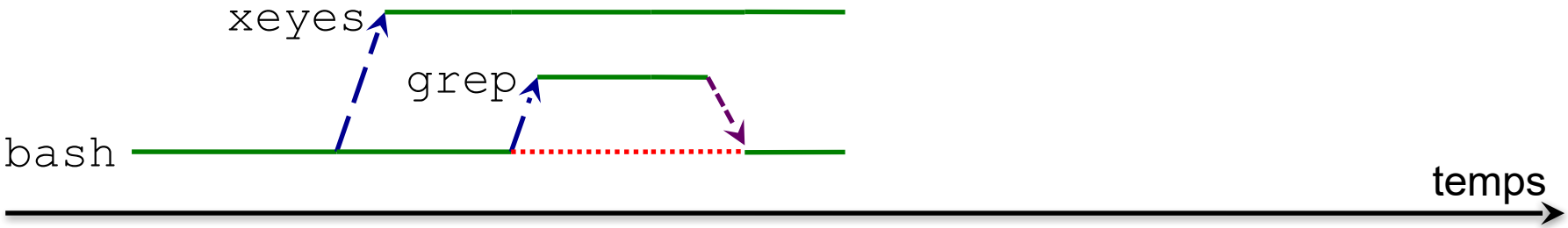
Processus en attente

Création de processus

Attendre la fin d'un processus




```
$ xeyes &
$ grep gthomas /etc/passwd
gthomas:x:501:20::/home/gthomas:/bin/bash
$
```



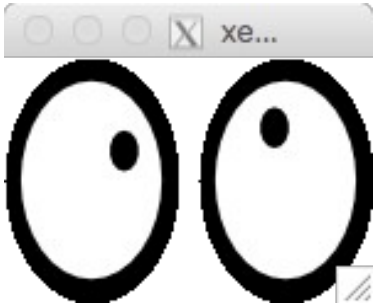
Processus en exécution

Processus en attente

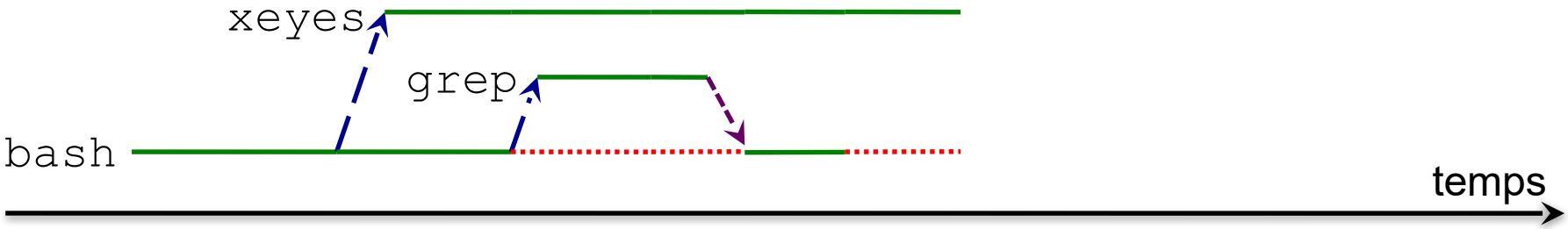
Création de processus

Notification de fin de processus

Attendre la fin d'un processus




```
$ xeyes &
$ grep gthomas /etc/passwd
gthomas:x:501:20::/home/gthomas:/bin/bash
$ wait
```



Processus en exécution

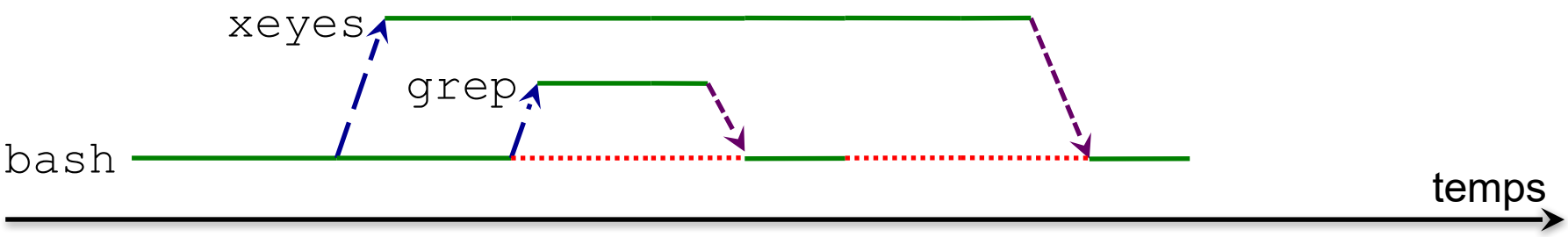
Processus en attente

Création de processus

Notification de fin de processus

Attendre la fin d'un processus

```
$ xeyes &
$ grep gthomas /etc/passwd
gthomas:x:501:20::/home/gthomas:/bin/bash
$ wait
[1]+  Done                    xeyes
$
```



Processus en exécution

Processus en attente

Création de processus

Notification de fin de processus

1. Observer un processus
2. Processus en avant et arrière plan
3. Cycle de vie d'un processus
4. Variables et processus
5. Gestion des processus dans le système d'exploitation

Variables `bash` et processus

- Une variable est toujours locale à un processus
  ⇒ les modifications sont toujours locales

- Une variable peut être exportée chez un enfant
  - La variable et sa valeur sont recopiées chez l'enfant à la création
  - Les variables du père et du fils sont ensuite indépendantes
  - Par défaut une variable n'est pas exportée
  - Marquer une variable comme exportée : `export var`
  - Arrêter d'exporter une variable : `unset var`
    (détruit aussi la variable)

Portée des variables

```
$ a="existe"
$
```

```
#! /bin/bash

b="existe"
echo "a: $a"
echo "b: $b"
a="autre chose"
```

variable.sh

```
#! /bin/bash

export b
b="existe"
echo "a: $a"
echo "b: $b"
```

variable_exportee.sh

Portée des variables

```
$ a="existe"
$ ./variable.sh
a:
b: existe
$
```

```
#! /bin/bash

b="existe"
echo "a: $a"
echo "b: $b"
a="autre chose"
```

variable.sh

```
#! /bin/bash

export b
b="existe"
echo "a: $a"
echo "b: $b"
```

variable_exportee.sh

Portée des variables

```
$ a="existe"
$ ./variable.sh
a:
b: existe
$ export a
$
```

```
#! /bin/bash

b="existe"
echo "a: $a"
echo "b: $b"
a="autre chose"
```

variable.sh

```
#! /bin/bash

export b
b="existe"
echo "a: $a"
echo "b: $b"
```

variable_exportee.sh

Portée des variables

```
$ a="existe"
$ ./variable.sh
a:
b: existe
$ export a
$ ./variable.sh
a: existe
b: existe
$
```

```
#! /bin/bash

b="existe"
echo "a: $a"
echo "b: $b"
a="autre chose"
```

variable.sh

```
#! /bin/bash

export b
b="existe"
echo "a: $a"
echo "b: $b"
```

variable_exportee.sh

Portée des variables

```
$ a="existe"
$ ./variable.sh
a:
b: existe
$ export a
$ ./variable.sh
a: existe
b: existe
$ echo "a: $a - b: $b"
a: existe - b:
$
```

```
#! /bin/bash

b="existe"
echo "a: $a"
echo "b: $b"
a="autre chose"
```

variable.sh

```
#! /bin/bash

export b
b="existe"
echo "a: $a"
echo "b: $b"
```

variable_exportee.sh

Portée des variables

```
$ a="existe"
$ ./variable.sh
a:
b: existe
$ export a
$ ./variable.sh
a: existe
b: existe
$ echo "a: $a - b: $b"
a: existe - b:
$ ./variable_exportee.sh
a: existe
b: existe
$
```

```
#! /bin/bash

b="existe"
echo "a: $a"
echo "b: $b"
a="autre chose"
```

variable.sh

```
#! /bin/bash

export b
b="existe"
echo "a: $a"
echo "b: $b"
```

variable_exportee.sh

Portée des variables

```
$ a="existe"
$ ./variable.sh
a:
b: existe
$ export a
$ ./variable.sh
a: existe
b: existe
$ echo "a: $a - b: $b"
a: existe - b:
$ ./variable_exortee.sh
a: existe
b: existe
$ echo "b: $b"
b:
$
```

```
#! /bin/bash

b="existe"
echo "a: $a"
echo "b: $b"
a="autre chose"
```

variable.sh

```
#! /bin/bash

export b
b="existe"
echo "a: $a"
echo "b: $b"
```

variable_exportee.sh

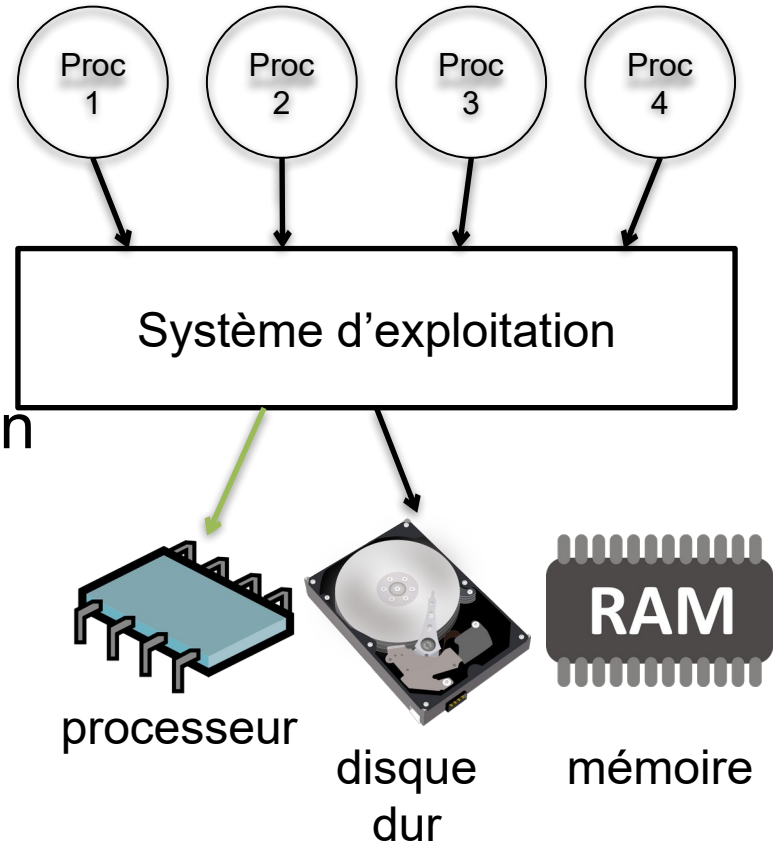
Variables d'environnement

- Une variable exportée s'appelle une variable d'environnement
  - Par convention, son nom est en majuscules
- Certaines variables sont souvent dans l'environnement :
  - `HOME` : chemin absolu du répertoire de connexion
    - `cd` , `cd ~` et `cd $HOME` sont des commandes équivalentes
  - `PS1` : prompt (par défaut `$`)
  - `PATH` : liste des répertoires de recherche des commandes
    - Rappel : entre chaque chemin, séparateur « `:` »
- La commande `env` liste toutes les variables de l'environnement courant
- La commande `source` charge un script (et ses variables !) dans le processus bash courant
  - Exemple pour recharger la configuration Bash : `source ~/.bashrc`

1. Observer un processus
2. Processus en avant et arrière plan
3. Cycle de vie d'un processus
4. Variables et processus
5. Gestion des processus dans le système d'exploitation

Partage de ressources

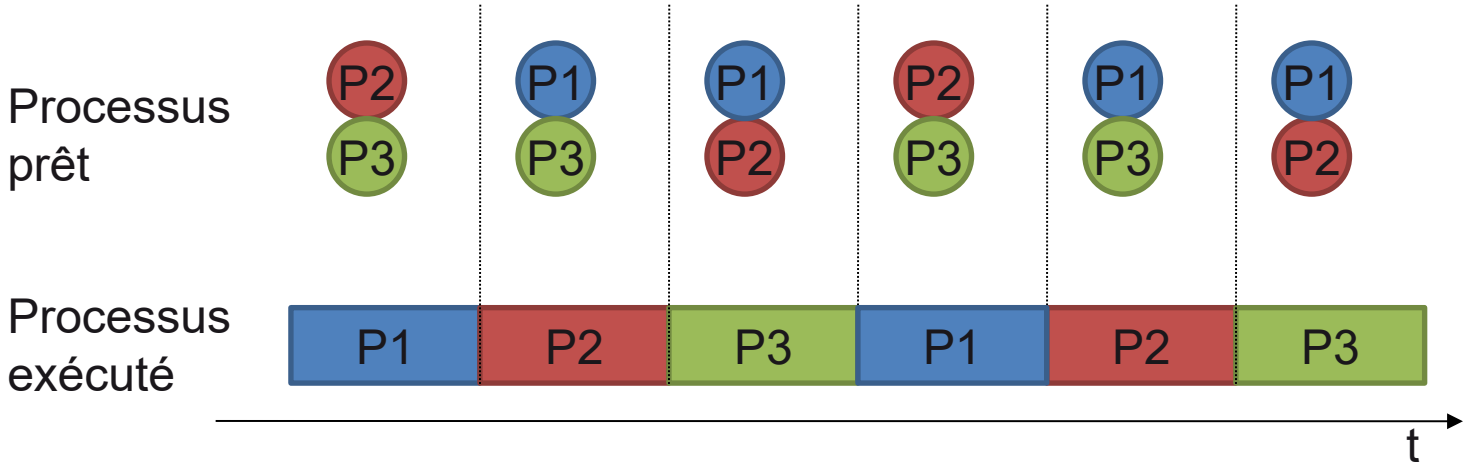
- Ressources partagées par les processus
  - CPU (cœur d'un processeur)
  - Mémoire
  - Entrées-sorties



- Gestion par le Système d'Exploitation
  - Exclusion mutuelle
  - Contrôle de l'accès au matériel
  - Droits d'accès
  - Non-dépassement des limites

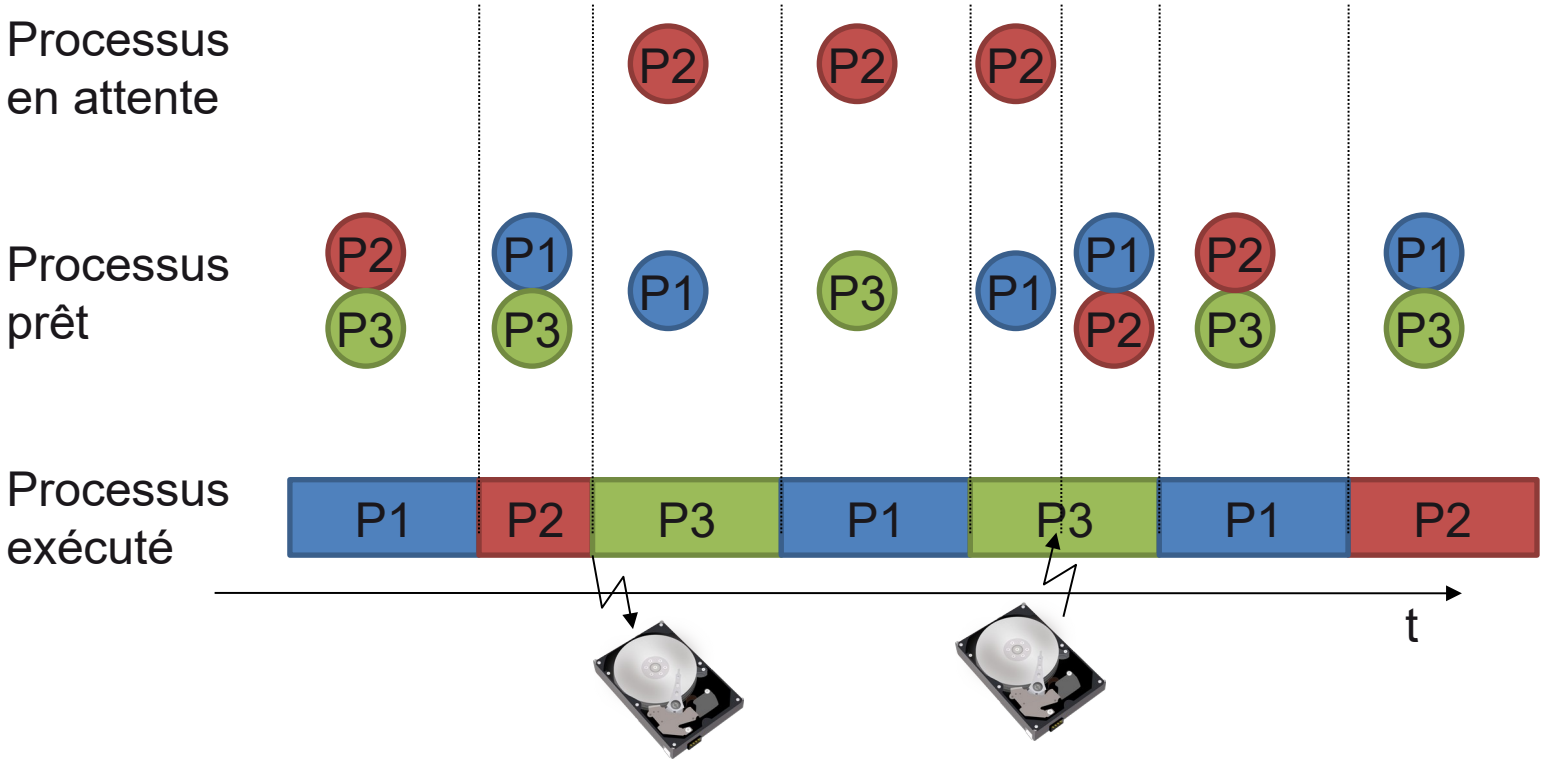
Partage du CPU

- À un instant donné, le CPU n'exécute qu'un processus
  - Les autres processus attendent
- L'ordonnanceur partage le CPU par « quantum de temps » (en anglais, *timeslice*)
  - À la fin du *timeslice*, l'ordonnanceur préempte le processus s'exécutant et choisit un autre processus



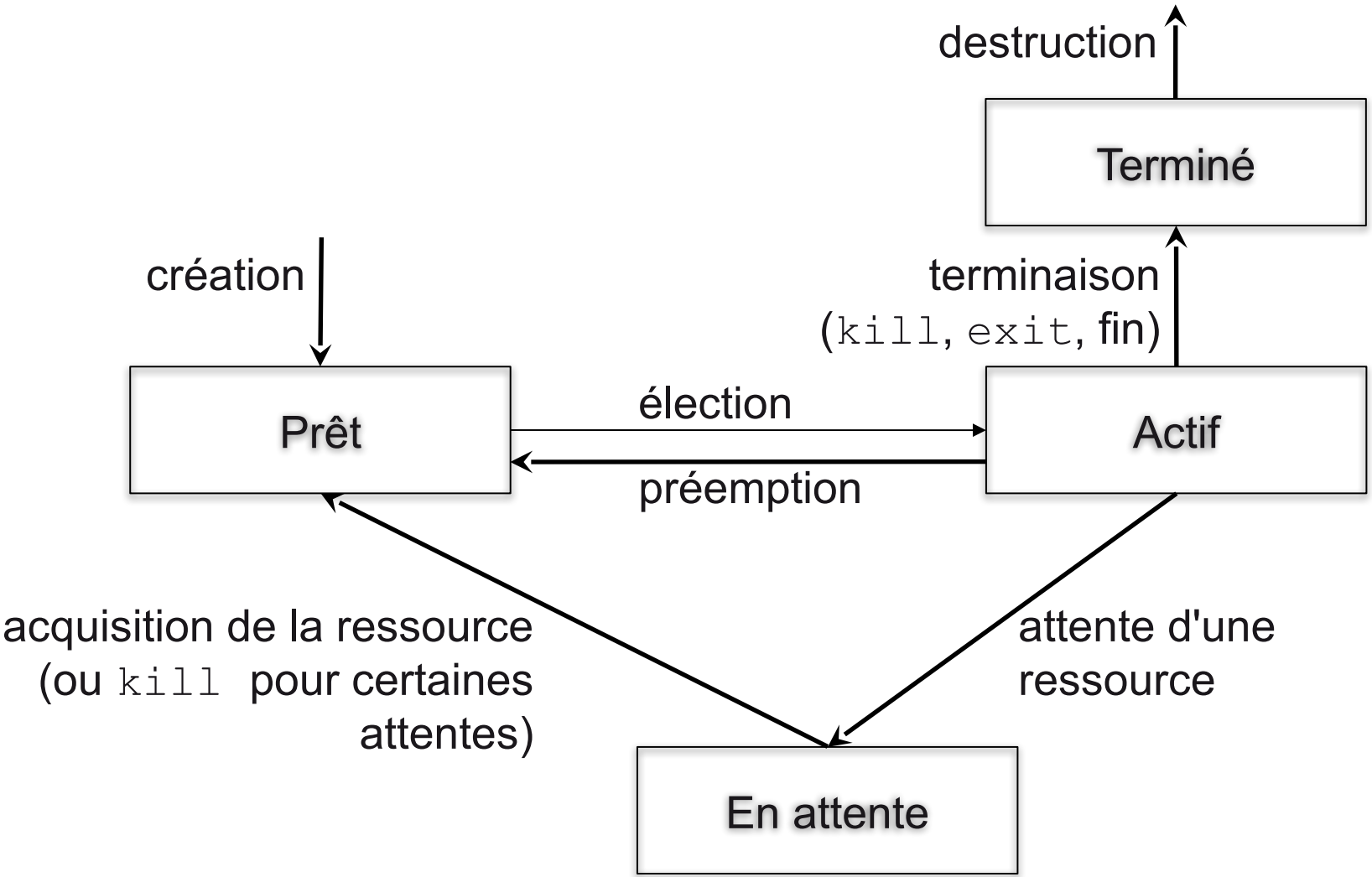
Partage du CPU et entrées/sorties

- Entrées/sorties ⇒ attente d'une ressource (disque, carte réseau, écran, etc.)
- Libération du CPU en attendant la ressource



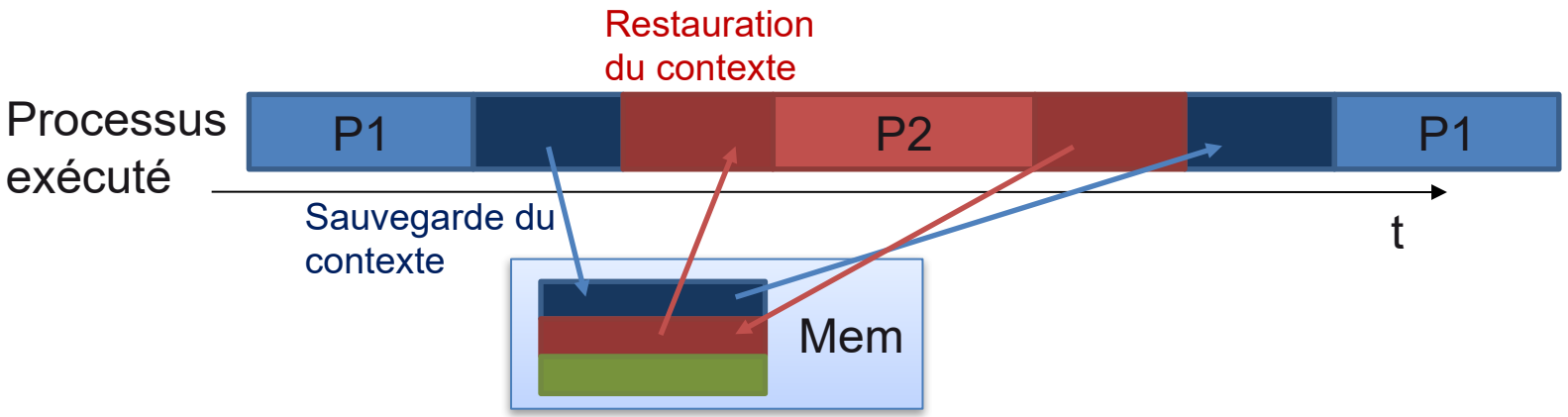
États d'un processus

Le point de vue du système d'exploitation



Commutation de processus

- La commutation a lieu lors de l'élection d'un processus :
  - Sauvegarde du contexte du processus évincé
  - Chargement du contexte du processus élu
- Contexte : ensemble des infos associées au processus
  - Valeur des registres
  - Informations mémoire (emplacement, etc.)

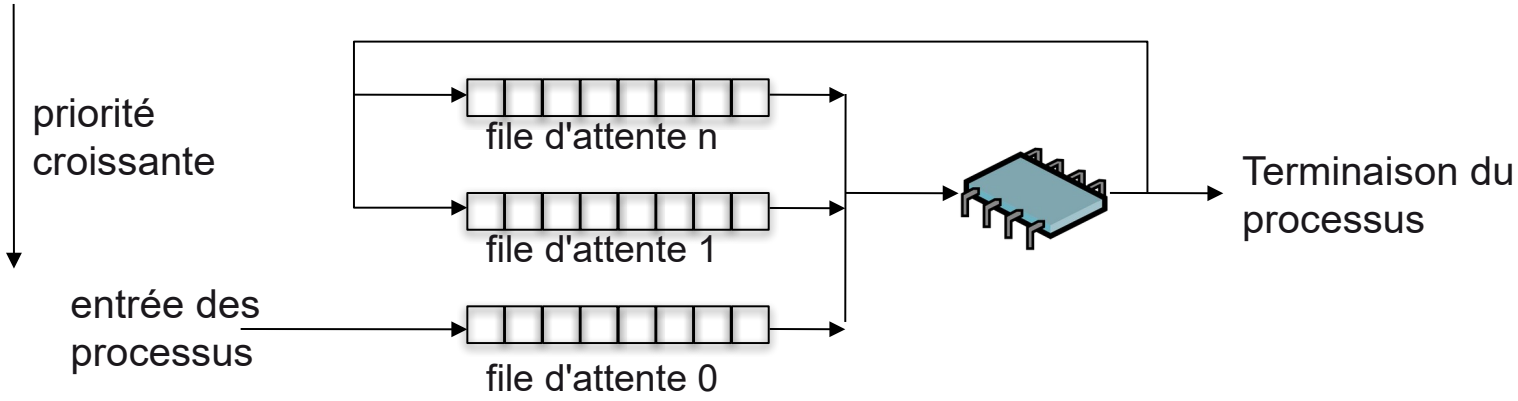


Ordonnancement de processus

Exemple d'algorithme d'ordonnancement à priorité

- Une file d'attente des processus prêts par niveau de priorité
- L'ordonnanceur choisit plus souvent les processus de forte priorité
- Ajustement de la priorité d'un processus au court de son exécution

- Exemple d'algorithme d'ordonnancement
  - Choisir un processus de la file d'attente non vide de plus haute priorité
  - Si un processus consomme tout son *timeslice* : `priorité--`
  - Régulièrement : `priorité++` pour les processus non élus



Changer la priorité d'un processus

- Possibilité de changer manuellement la priorité d'un processus
  - Exemple: baisser la priorité d'un programme qui indexe le contenu d'un disque dur
- Lancer un programme avec une certaine priorité
  - `$ nice –n priorité commande`
- Changer la priorité d'un processus déjà lancé
  - `$ renice –n priorité PID`

Introduction à la concurrence

- Accès concurrent à une ressource gérée par l'OS
  - Disque dur, imprimante, sortie du terminal, …
- L'OS assure l'exclusion mutuelle de ses ressources
  - À tout moment, seul un processus manipule la ressource

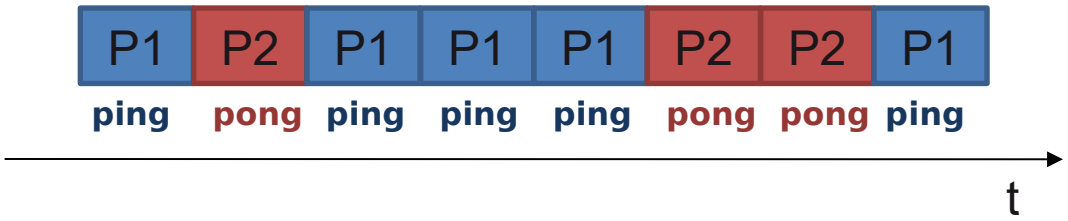
```
$ ./do_ping.sh & ./do_pong.sh
ping
pong
ping
pong
ping
pong
ping
pong
pong
pong
ping
ping
ping
ping
pong
```

```
#!/bin/bash
while true; do
    echo ping
done
```

do_ping.sh

```
#!/bin/bash
while true; do
    echo pong
done
```

do_pong.sh



Conclusion

- Concepts clés
  - Processus
    - Caractéristiques statiques et dynamiques
    - Processus parent, processus enfant
    - Exécution en avant-plan, arrière-plan, suspension/reprise de processus
  - Ordonnancement de processus
    - Quantum de temps, préemption
    - changement de contexte
- Commandes clés
  - `ps, pstree, top`
  - `CTRL+Z, fg, bg`
  - `CTRL+C, kill, killall`

En route pour le TP !!

INSTITUT Mines-Télécom

TELECOM SudParis

Les signaux

CSC 3102

Introduction aux systèmes d'exploitation

Gaël Thomas

Présentation du cours

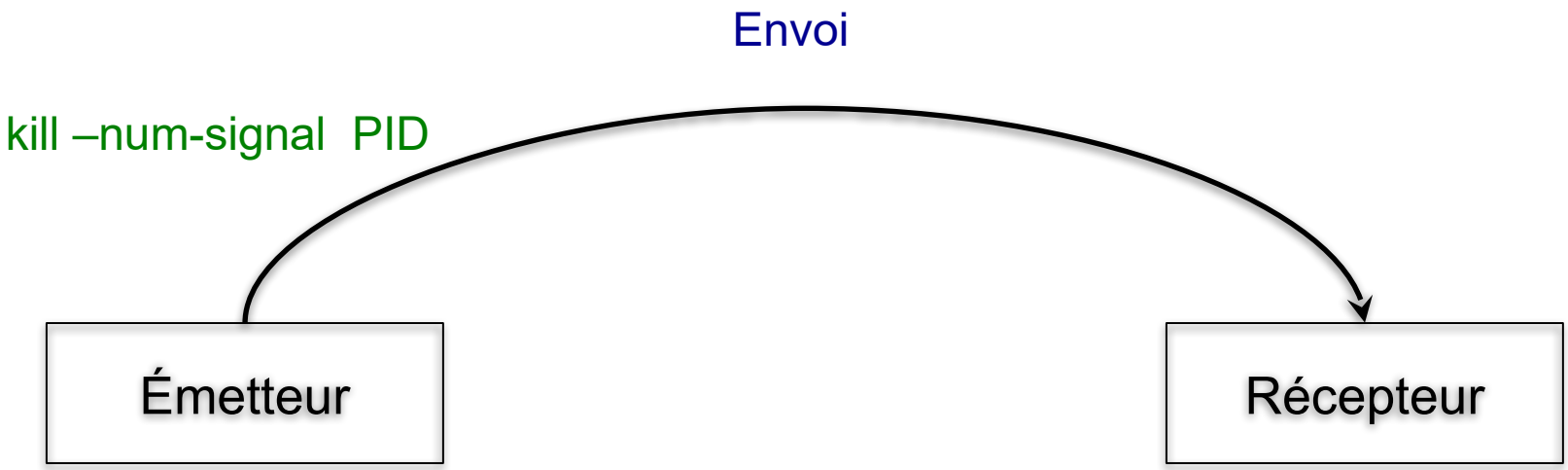
- Contexte : comprendre un mécanisme de communication inter-processus
- Objectif : Savoir utiliser les signaux
- Notion clé : Signaux (`kill`, `trap`)

Les signaux

- Signal = un mécanisme de communication inter-processus
  - Communication simple par envoi de message direct
  - Message = un entier `n` entre 1 et 31
  - Perte de message possible (si sig. `n` envoyé 2x avant réception)
  - Ordre de réception aléatoire (différent de l'ordre d'émission)

- Souvent utilisé pour
  - Arrêter un processus (par exemple, `control-c`)
  - Notifier un processus lorsque sa configuration change
  - Prévenir un processus qu'il effectue une opération invalide (accès mémoire invalide, division par zéro…)

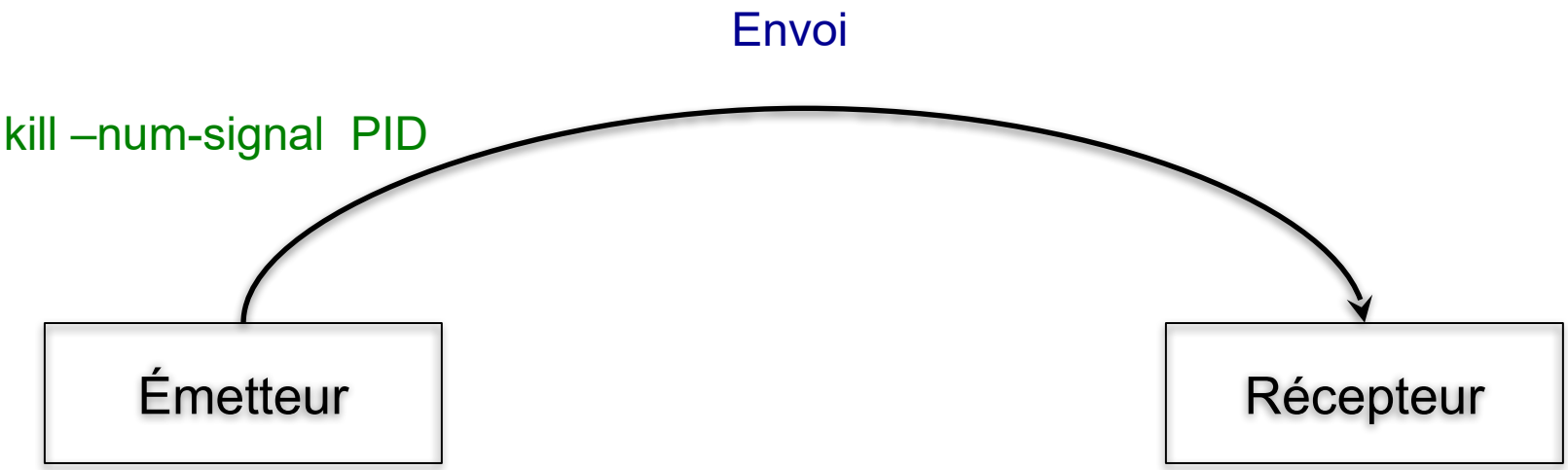
Principe de fonctionnement



- Émetteur envoie un message à un processus
- Une routine de réception est automatiquement invoquée chez le récepteur dès que le signal arrive
- Par défaut, cette routine tue le récepteur

  (sauf pour les signaux SIGCHLD, SIGTSTP, SIGSTOP, SIGCONT)

Principe de fonctionnement



- Nota bene :
  - Un message est limité à un nombre compris entre 1 et 31
  - Tout signal émis est livré
    (sauf si le même numéro de signal est émis une seconde fois avant réception – dans ce cas le deuxième signal est perdu)
  - Ordre de réception aléatoire

Les signaux

Quelques exemples (man 7 signal) :

- SIGHUP (1) : fermeture terminal ⇒ à tous les processus attachés
- SIGINT (2) : control-c dans un terminal ⇒ au processus au premier plan
- SIGQUIT (3) : souvent control-d, généré par un processus à lui-même
- SIGILL (4) : instruction illégale (envoyé par le noyau)
- SIGFPE (8) : division par 0 (envoyé par le noyau)
- SIGKILL (9) : pour terminer un processus
- SIGSEGV (11) : accès mémoire invalide (envoyé par le noyau)
- SIGTERM (15) : argument par défaut de la commande kill
- SIGCHLD (17) : envoyé par le noyau lors de la mort d'un fils
- SIGCONT (18) : redémarre un procesus suspendu (avec bg ou fg)
- SIGTSTP (20) : suspend un processus (généré par control-z)
- SIGUSR1 (10) : libre, sémantique définie pour chaque processus
- SIGUSR2 (12) : libre, sémantique définie pour chaque processus

Les signaux

- Deux signaux bien utiles
  - `SIGTSTP` : demande au système de suspendre un processus
  - `SIGCONT` : demande au système de le redémarrer

- `Bash` utilise ces signaux :
  - `control-z` : envoie un `SIGTSTP` au processus au premier plan
  - `bg` et `fg` : envoient un `SIGCONT` au processus stoppé
    (rappel : `bg` background, `fg` foreground)

Les signaux

- Un processus peut attacher un gestionnaire dit de signal avec

  `trap 'expression' sig`

  ⇒ exécution de `expression` lors de la réception de `sig`

  À faire avant de recevoir le signal (en gén., au début du programme)

- Un processus peut envoyer un signal à un destinataire avec

  `kill -sig pid`

- Où
  - `expression` : expression quelconque bash
  - `sig` : numéro de signal (nombre ou symbole comme `USR1`)
  - `pid` : `PID` du processus destinataire

Les signaux

Attention : n'oubliez pas les apostrophes !

- Un processus peut attacher un gestionnaire dit de signal avec

  `Trap 'expression' sig`

  ⇒ exécution de `expression` lors de la réception de `sig`

  À faire avant de recevoir le signal (en grl., au début du programme)

- Un processus peut envoyer un signal à un destinataire avec

  `kill -sig pid`

- Où
  - `expression` : expression quelconque bash
  - `sig` : numéro de signal (nombre ou symbole comme `USR1`)
  - `pid` : `PID` du processus destinataire

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$
```

Terminal 1

```
$
```

Terminal 2

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$
```

Terminal 1

```
$ ./recepteur.sh
```

Terminal 2

Terminal 2 : lancement de `recepteur.sh`

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$
```

Terminal 1

```
$ ./recepteur.sh
```

Terminal 2

`recepteur.sh` attache le gestionnaire `'echo coucou'` à `USR1`

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$
```

Terminal 1

```
$ ./recepteur.sh
PID: 52075
```

Terminal 2

`recepteur.sh` affiche son `PID`

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$
```

Terminal 1

```
$ ./recepteur.sh
PID: 52075
```

Terminal 2

`recepteur.sh` exécute la boucle infinie

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$ ./emetteur.sh 52075
```

Terminal 1

```
$ ./recepteur.sh
PID: 52075
```

Terminal 2

Terminal 1 : lancement de `emetteur.sh`

Principe de fonctionnement

USR1 en attente

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

Envoi du signal

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$ ./emetteur.sh 52075
```

Terminal 1

```
$ ./recepteur.sh
PID: 52075
```

Terminal 2

emetteur.sh envoie le signal USR1 à recepteur.sh

Principe de fonctionnement

USR1 en attente

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$ ./emetteur.sh 52075
$
```

Terminal 1

```
$ ./recepteur.sh
PID: 52075
```

Terminal 2

emetteur.sh termine

(l'ordre entre emetteur.sh et recepteur.sh est aléatoire)

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$ ./emetteur.sh 52075
$
```

Terminal 1

```
$ ./recepteur.sh
PID: 52075
coucou
```

Terminal 2

`recepteur.sh` reçoit le signal

⇒ le système déroute l'exécution de `recepteur.sh` vers le gestionnaire

⇒ affiche `coucou`

Principe de fonctionnement

```
#! /bin/bash

kill -USR1 $1
```

emetteur.sh

```
#! /bin/bash

trap 'echo coucou' USR1

echo "PID: $$"

while true; do
    sleep 1
done
```

recepteur.sh

```
$ ./emetteur.sh 52075
$
```

Terminal 1

```
$ ./recepteur.sh
PID: 52075
coucou
```

Terminal 2

À la fin du gestionnaire du signal, l'exécution reprend là où elle s'était arrêtée

Notions clés

- Les signaux
  - Mécanisme de communication à base de messages
  - Message = nombre entre 1 et 31
  - Ordre de réception aléatoire
  - Perte possible en cas d'envoi multiple du même numéro de signal
  - `kill -sig pid` : envoie un signal `sig` à `pid`
  - `trap 'expr' sig` : associe `expr` à la réception d'un signal `sig`

À vous de jouer!

INSTITUT Mines-Télécom

TELECOM SudParis

Les tubes

CSC 3102

Introduction aux systèmes d'exploitation

Gaël Thomas

Présentation du cours

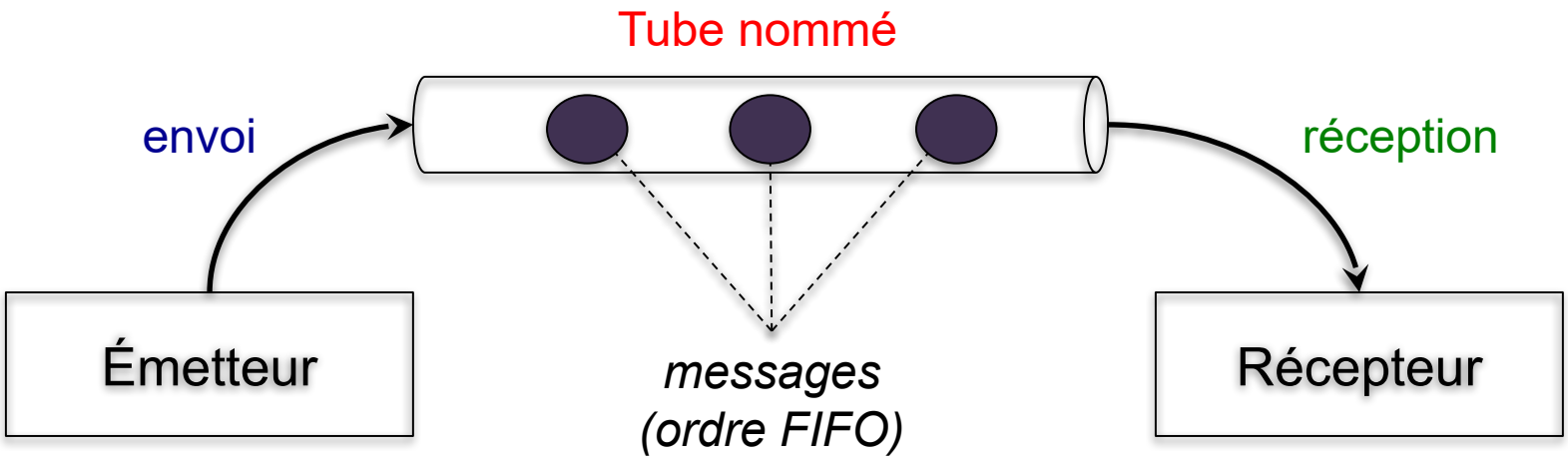
- Contexte : comprendre comment les processus interagissent
- Objectifs :
  - Savoir utiliser les tubes
- Notions clés :
  - Les tubes (`mkfifo`, redirections avancées, `|`)

Les tubes

- Tube = mécanisme de communication par envoi de messages
  - Via un canal de communication
  - Message = données quelconques
  - Pas de perte de message,
  - Réception dans l'ordre d'émission
    - (Réception dite FIFO pour First In First Out)

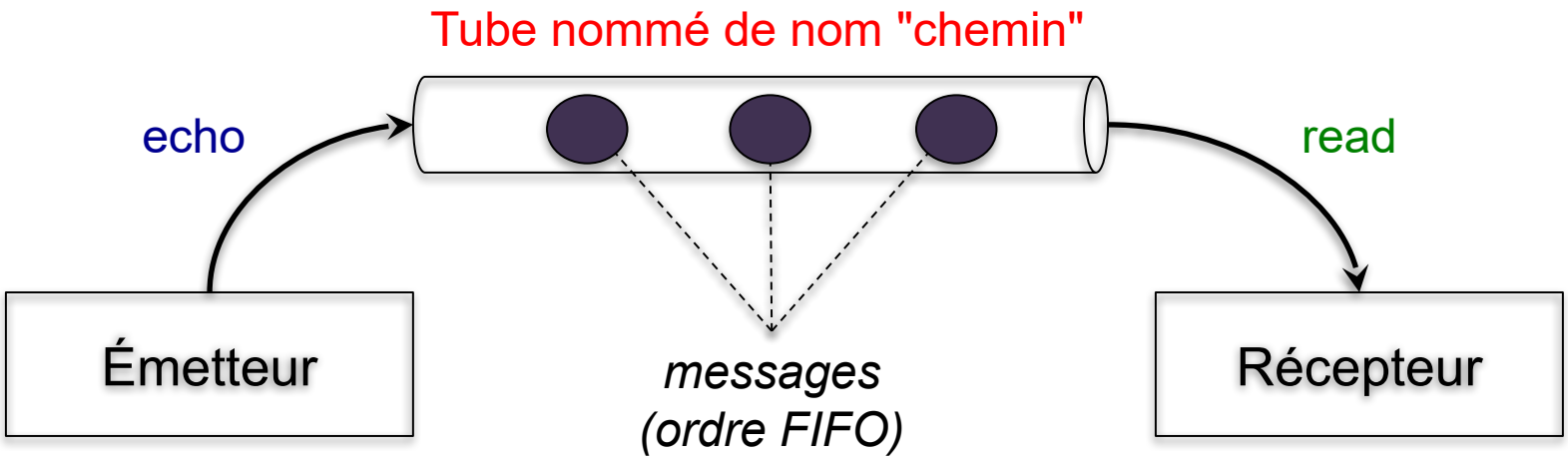
- Échange de messages complexes entre processus
  - Base de données + serveur Web
  - Processus d'affichage de notifications pour les autres processus
  - De façon générale, pour mettre en place une architecture de type client/serveur

Tubes nommés



- Principe :
  - Un tube est un fichier spécial dans le système de fichiers
  - L'émetteur écrit dans le tube
  - Le récepteur lit dans le tube

Principe de mise en œuvre



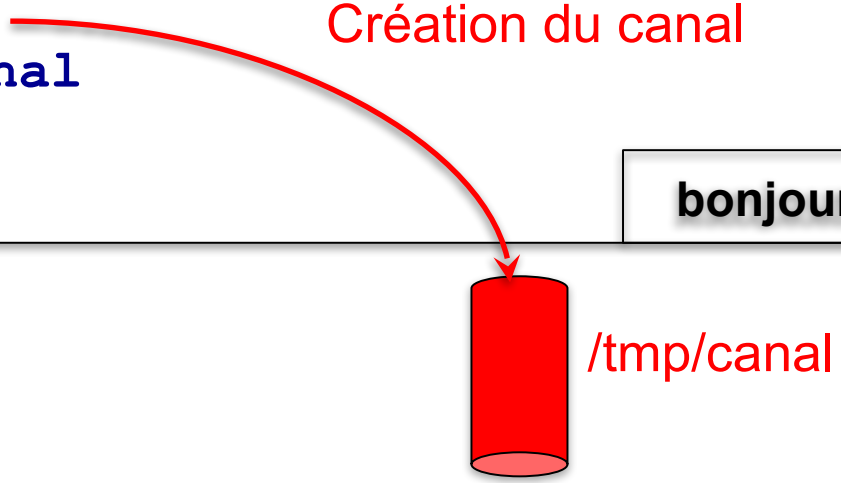
- Principe de mise en œuvre :
  - `mkfifo chemin` : crée un tube nommé (visible avec `ls`)
  - `echo message > chemin` : écrit dans le tube nommé
  - `read message < chemin` : lit à partir du tube nommé
    (bloque si pas encore de message dans le tube)
  - `rm chemin` : détruit le tube nommé

Principe de mise en œuvre

```
#!/bin/bash

mkfifo /tmp/canal
echo "Bonjour" >/tmp/canal
read line </tmp/canal
echo "Reçoit $line"
```

bonjour.sh




```
#!/bin/bash

read line </tmp/canal
echo "Reçoit $line"
echo "Au revoir" >/tmp/canal
```

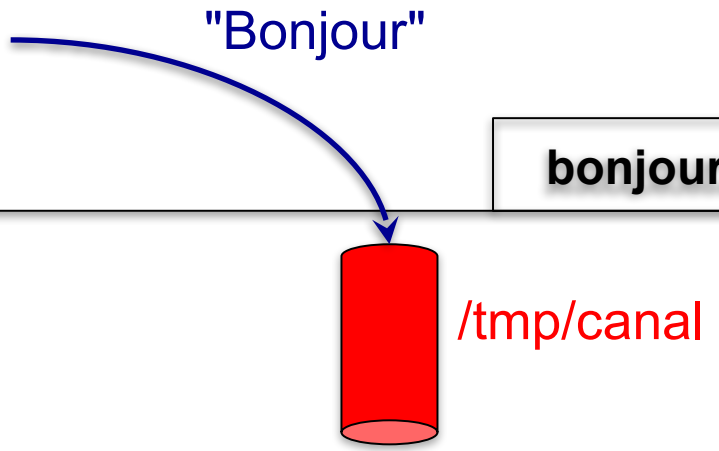
au-revoir.sh

Principe de mise en œuvre

```
#!/bin/bash

mkfifo /tmp/canal
echo "Bonjour" >/tmp/canal
read line </tmp/canal
echo "Reçoit $line"
```

bonjour.sh




```
#!/bin/bash

read line </tmp/canal
echo "Reçoit $line"
echo "Au revoir" >/tmp/canal
```

au-revoir.sh

Principe de mise en œuvre

```
#!/bin/bash

mkfifo /tmp/canal
echo "Bonjour" >/tmp/canal
read line </tmp/canal
echo "Reçoit $line"
```

bonjour.sh

/tmp/canal

"Bonjour"

```
#!/bin/bash

read line </tmp/canal
echo "Reçoit $line"
echo "Au revoir" >/tmp/canal
```

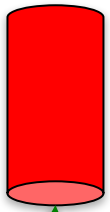
au-revoir.sh

Principe de mise en œuvre

```
#!/bin/bash

mkfifo /tmp/canal
echo "Bonjour" >/tmp/canal
read line </tmp/canal
echo "Reçoit $line"
```

bonjour.sh



/tmp/canal

```
#!/bin/bash

read line </tmp/canal
echo "Reçoit $line"
echo "Au revoir" >/tmp/canal
```

"Au revoir"

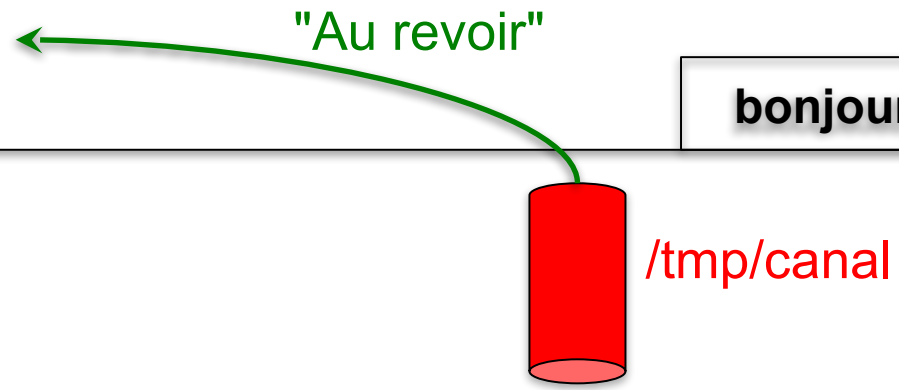
au-revoir.sh

Principe de mise en œuvre

```
#!/bin/bash

mkfifo /tmp/canal
echo "Bonjour" >/tmp/canal
read line </tmp/canal
echo "Reçoit $line"
```

bonjour.sh

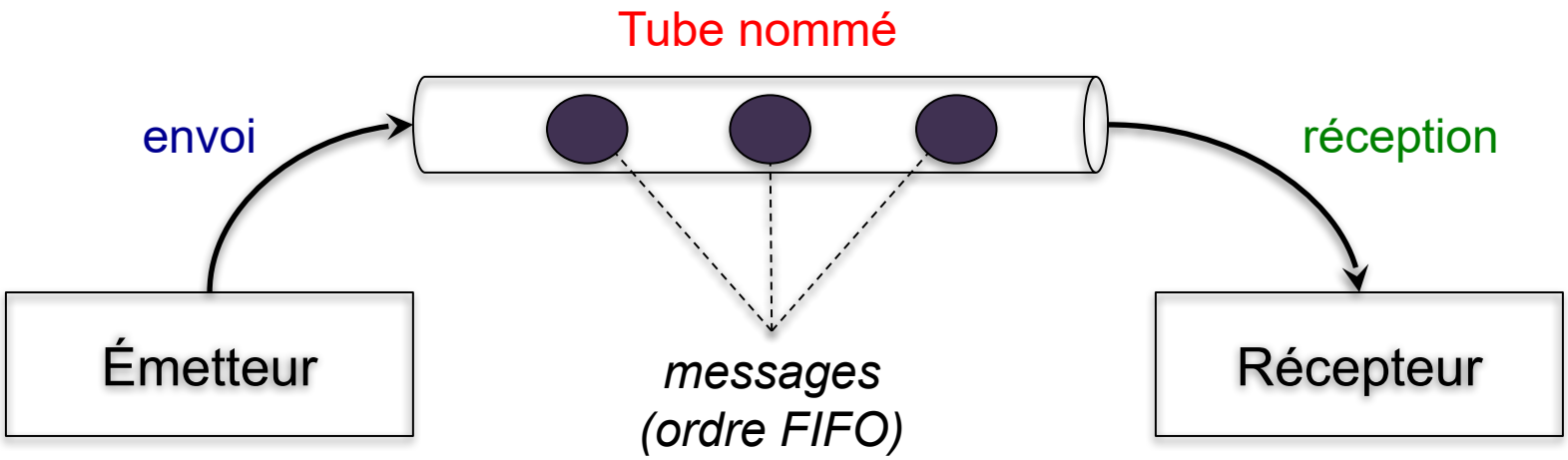



```
#!/bin/bash

read line </tmp/canal
echo "Reçoit $line"
echo "Au revoir" >/tmp/canal
```

au-revoir.sh

La vie est malheureusement complexe !



- Les contraintes
  - Le récepteur bloque en attendant un message
  - Ouverture bloque si interlocuteur n'existe pas

    `read line < f` bloque en attendant `echo msg > f` et vice-versa
  - Erreur si émission alors qu'il n'existe plus de récepteur

    (voir l'exemple donné dans les prochaines diapositives)

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

Démarre `emetteur_tube.sh` puis `recepteur_tube.sh`

- La plupart des tours de boucle s'exécutent sans problème
  - L'émetteur émet "yes"
  - Le récepteur reçoit le "yes"
- Mais parfois...

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
```

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
                                        2. Ouvre tube
```

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
                                        2. Ouvre tube
3. Écrit "yes"
```

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
                                        2. Ouvre tube
3. Écrit "yes"
4. Ferme tube
```

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
                                        2. Ouvre tube
3. Écrit "yes"
4. Ferme tube
5. Ré-ouvre tube(ok car ∃ récepteur)
```

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
                                        2. Ouvre tube
3. Écrit "yes"
4. Ferme tube
5. Ré-ouvre tube(ok car ∃ récepteur)
                                        6. Lit yes
```

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
                                                    2. Ouvre tube
3. Écrit "yes"
4. Ferme tube
5. Ré-ouvre tube(ok car ∃ récepteur)
                                                        6. Lit yes
                                                    7. Ferme tube
```

La vie est malheureusement complexe !

- On fait souvent des envois/réceptions dans des boucles

```
#!/bin/bash
mkfifo tube
while true; do
  echo "yes" >tube
done
```

emetteur_tube.sh

```
#!/bin/bash

while true; do
  read line <tube
done
```

recepteur_tube.sh

```
1. Ouvre tube
                                        2. Ouvre tube
3. Écrit "yes"
4. Ferme tube
5. Ré-ouvre tube(ok car ∃ récepteur)
                                            6. Lit yes
                                        7. Ferme tube
8. Écriture ⇒ plantage (silencieux) : pas de récepteur
```

Tube et redirection avancée

- Pour éviter ces fermetures intempestives de tubes

On préconise dans ce cours de toujours ouvrir un tube avec une redirection avancée en lecture/écriture

- Si le récepteur ferme le tube, l'émetteur agissant comme récepteur, plus de plantage

- Effet connexe : ni l'émetteur ni le récepteur ne bloquent pendant l'ouverture s'il n'existe pas encore d'interlocuteur

Tube et redirection avancée

- **Toujours ouvrir un tube avec une redirection avancée en lecture/écriture**

```
#!/bin/bash
mkfifo tube
exec 3<>tube
while true; do
  echo "yes" >&3
done
```

emetteur_tube_exec.sh

```
#!/bin/bash

exec 3<>tube
while true; do
  read line <&3
done
```

recepteur_tube_exec.sh

(redirection avancée pas nécessaire chez le récepteur dans ce cas, mais bonne habitude à prendre car souvent, un récepteur est aussi un émetteur)

Retour sur les tubes anonymes

- Tube anonyme : comme tube nommé, mais sans nom

- Le « `|` » entre deux processus shell crée un tube anonyme

```
cmd_gauche | cmd_droite
```

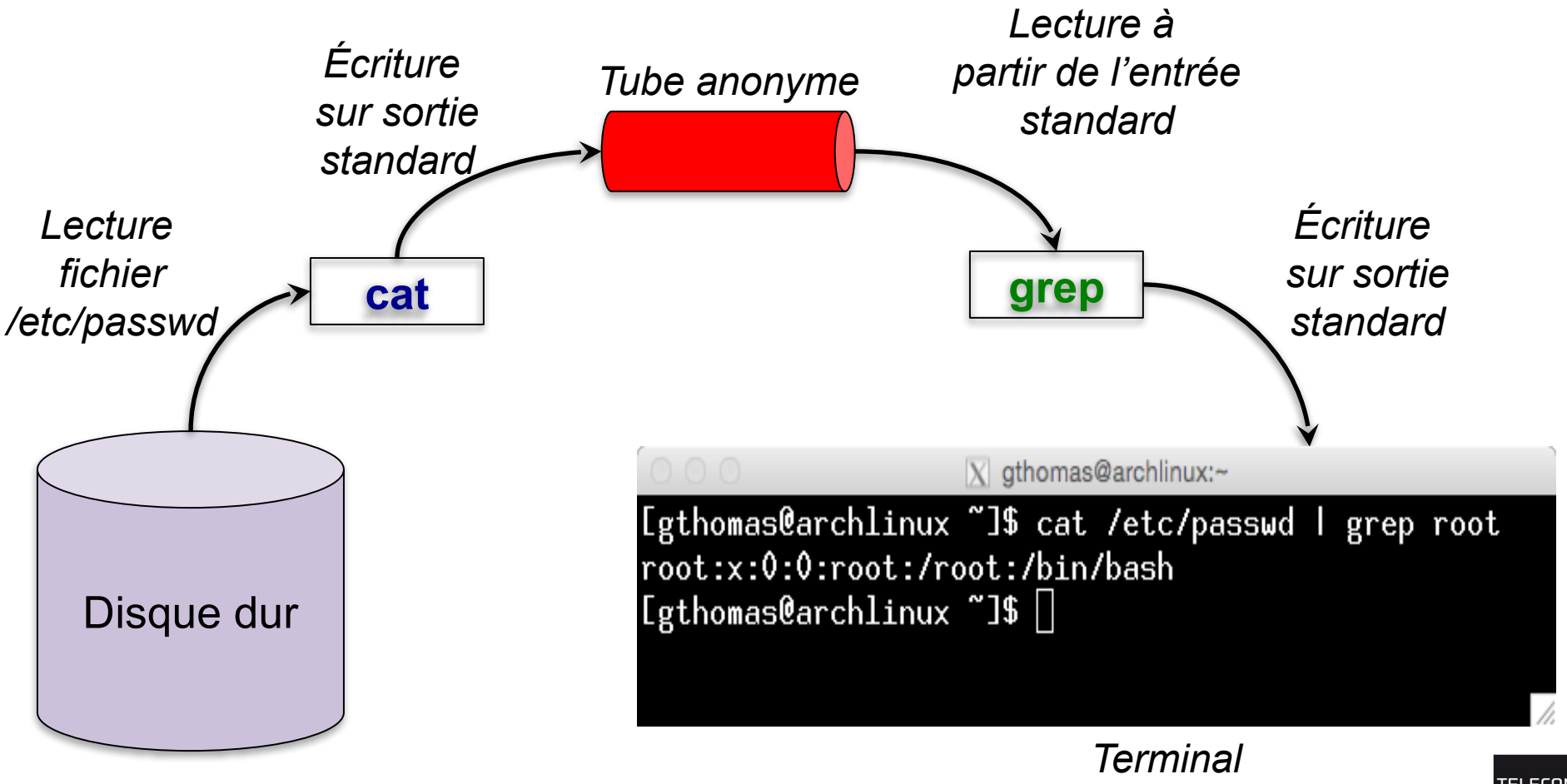
  - Sortie standard de `cmd_gauche` connectée au tube
  - Entrée standard de `cmd_droite` connectée au tube

- À haut niveau, un peu comme si on exécutait

```
mkfifo tube-avec-nom
cmd_gauche > tube-avec-nom &
cmd_droite < tube-avec-nom
```

Retour sur les tubes anonymes

```
cat /etc/passwd | grep root
```

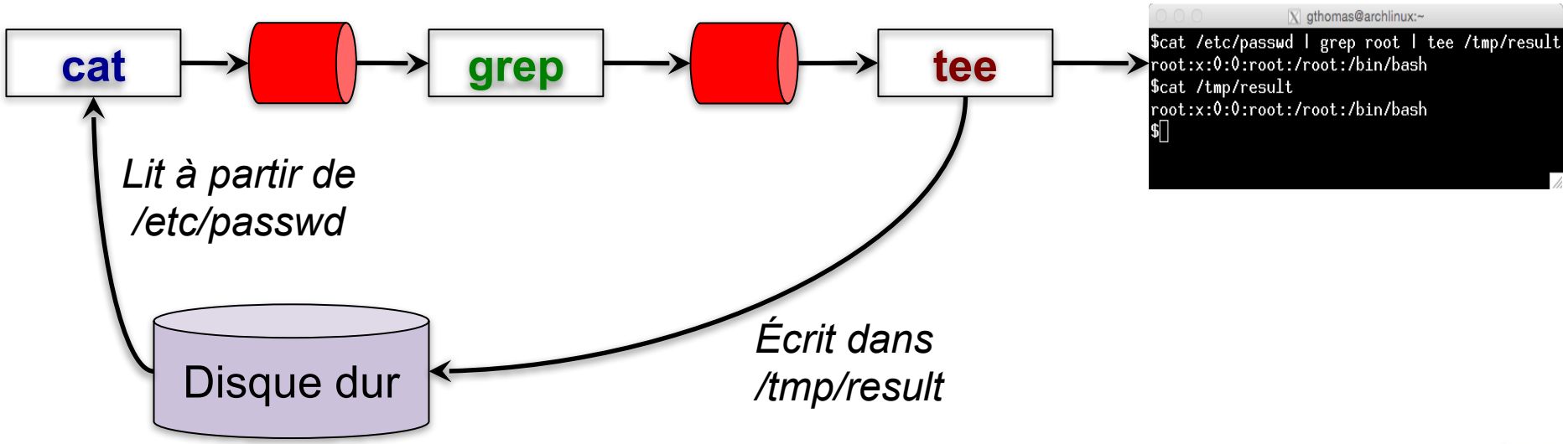


Terminal

Commande utile : `tee`

- Écrit les données lues à partir de l'entrée standard
  - Sur la sortie standard
  - Et dans un fichier passé en argument

- Exemple : **`cat`** `/etc/passwd | ` **`grep`** `root | ` **`tee /tmp/result`**



Notions clés

- Tube nommé
  - Fichier spécial dans le système de fichiers
  - Envoi de messages de taille quelconque, ordre de réception = ordre d'émission, pas de perte de message
  - `mkfifo nom-tube` : crée un tube nommé mon-tube
  - Toujours utiliser des redirections avancées

- Tubes anonyme (|)
  - Comme un tube nommé, mais sans nom dans le système de fichier

À vous de jouer !

INSTITUT Mines-Télécom

TELECOM SudParis

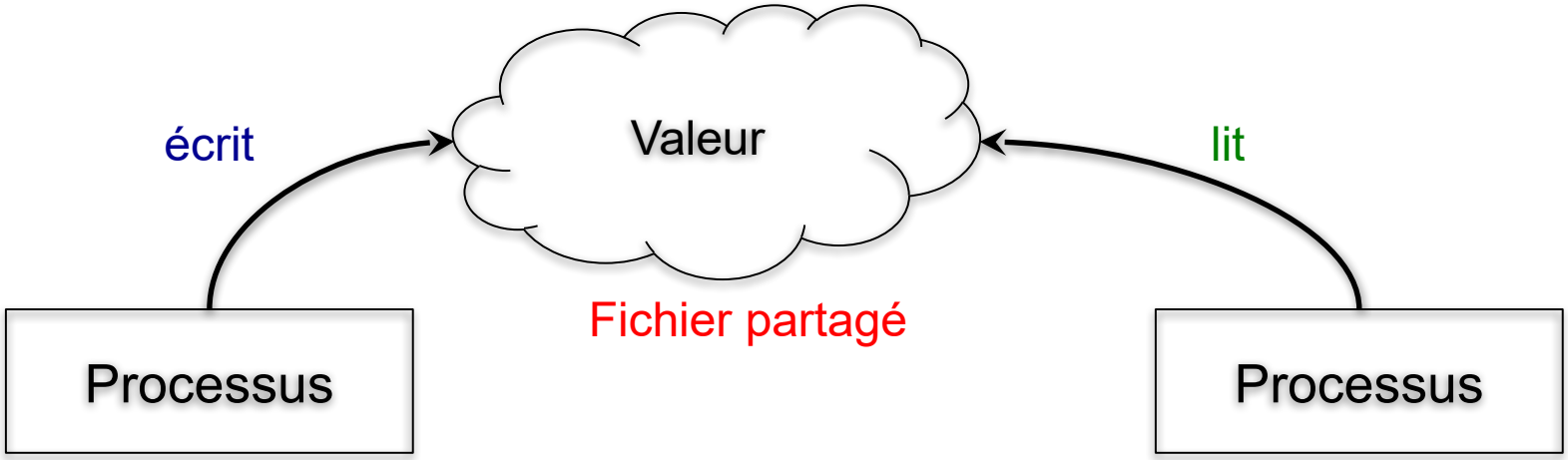
Communication entre processus : communication par fichiers partagés

CSC 3102

Introduction aux systèmes d'exploitation

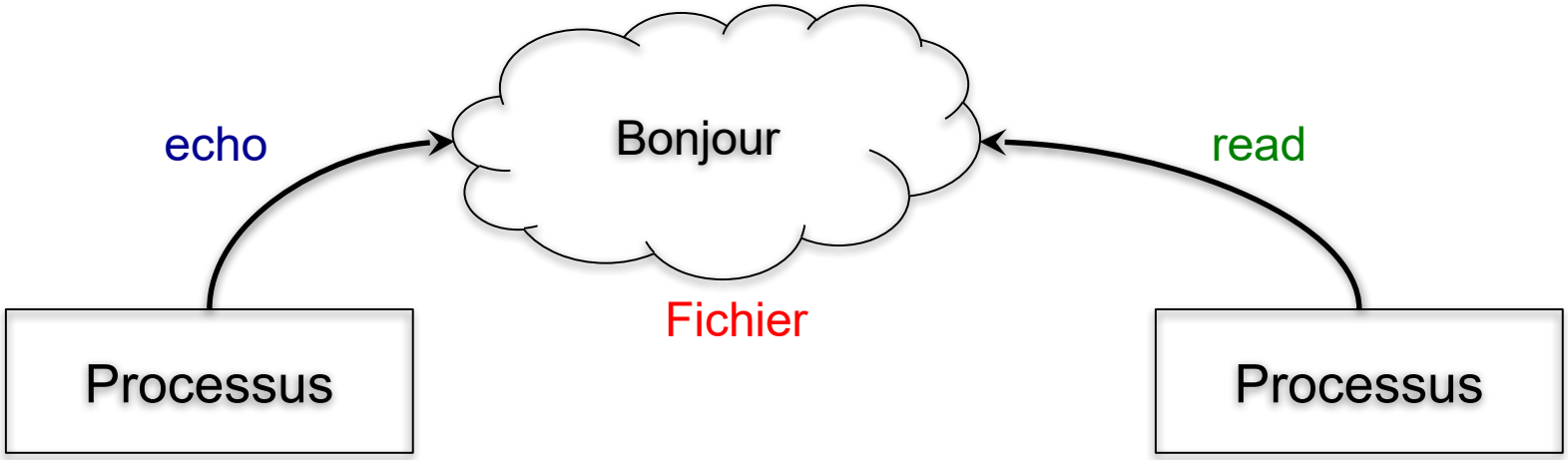
Gaël Thomas

Communication par fichiers partagés



- Des processus écrivent dans et lisent un fichier partagé

Communication par fichiers partagés



- Exemple
  - P1 exécute : `echo "Bonjour" > f1`
  - P2 exécute : `read a < f1`
- Différence entre tube nommé et fichier
  - Tube nommé : messages supprimés après la lecture
  - Fichier partagé : données non supprimées après la lecture

Le problème des fichiers partagés

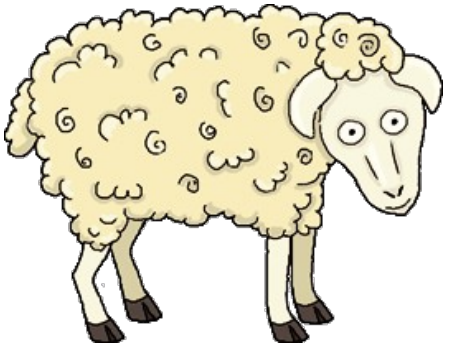
- Les fichiers peuvent être mis à jour concurremment
- Les accès concurrents aux fichiers partagés peuvent mener à des incohérences

Problème de la mise à jour concurrente

Deux enfants dessinent sur un tableau

Enfants 1 puis 2

Enfants 1 et 2 en parallèle



Enfants 2 puis 1

Donnée incohérente!

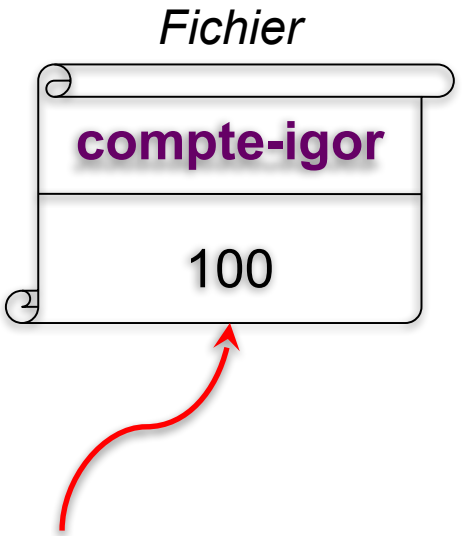
Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

```
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
```

P2 : débite le compte d'Igor de 100 euros

```
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
```

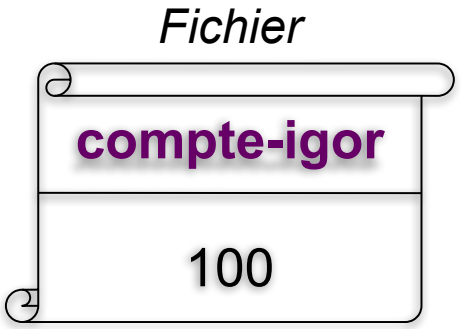


Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

```
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
```

a : 100



P2 : débite le compte d'Igor de 100 euros

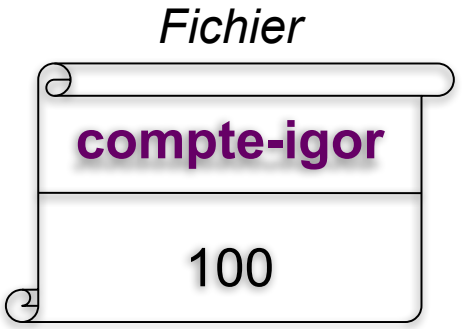
```
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
```

Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

```
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
```

a : 102



P2 : débite le compte d'Igor de 100 euros

```
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
```

Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

```
read a < compte-igor
a=$(expr $a + 2)
```

a : 102

```
echo $a > compte-igor
```

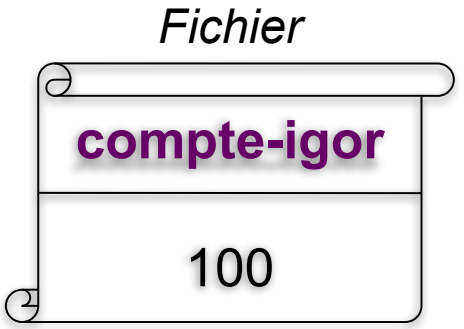
Commutation de P1 vers P2 avant le `echo`

P2 : débite le compte d'Igor de 100 euros

```
read b < compte-igor
b=$(expr $b - 100)
```

b : 100

```
echo $b > compte-igor
```

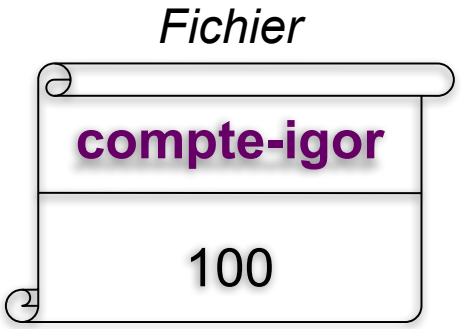


Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

```
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
```

a : 102



P2 : débite le compte d'Igor de 100 euros

```
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
```

b : 0

Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

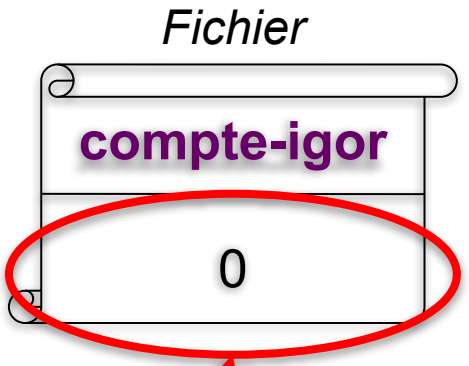
```
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
```

a : 102

P2 : débite le compte d'Igor de 100 euros

```
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
```

b : 0



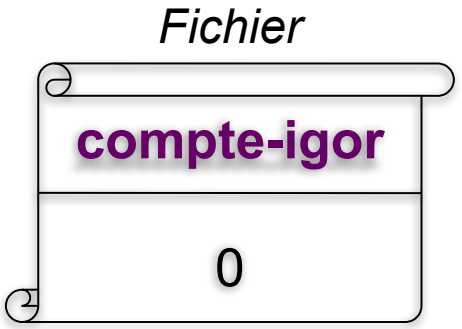
Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

```
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
```

a : 102

P2 se termine

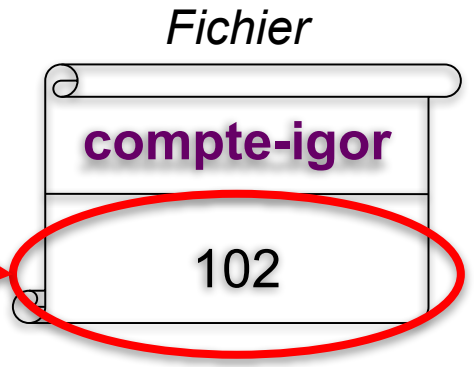


Problème de la mise à jour concurrente

P1 : crédite le compte d'Igor de 2 euros

```
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
```

a : 102



P2 se termine
⇒ commutation de P2 vers P1
⇒ P1 exécute le `echo`

Le retrait de 100 euros a été perdu!

Principe de solution

Éviter que deux sections de code accédant au même fichier partagé puissent s'exécuter en même temps

⇒ on parle de sections de code en ***exclusion mutuelle***

Sections critiques : sections de code en exclusion mutuelle

⇒ les sections critiques s'exécutent entièrement l'une après l'autre

Remarque : une section critique est souvent en exclusion mutuelle avec elle-même

Mise en œuvre de l'exclusion : le verrou

- Mutex : verrou (lock) en exclusion mutuelle

- Principe :
  - Verrouille le verrou avant d'entrer en section critique
  - Déverrouille le verrou à la sortie d'une section critique

- Deux opérations atomiques

  Atomique : semble s'exécuter instantanément
  - P.sh : attend que le verrou soit déverrouillé et le verrouille

    P comme « puis-je? » *(Proberen = tester en Néerlandais)*
  - V.sh : déverrouille le verrou

    V comme « vas-y » *(Verhogen = incrémenter en Néerlandais)*

Mise en œuvre de l'exclusion : le verrou

- Le verrou a été introduit par Edsger W. Dijkstra en 1965

  E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. Commun. ACM 8, 9, pp. 569-569.

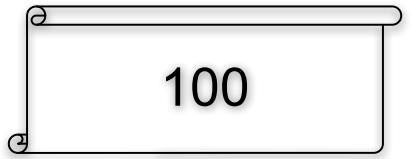
- Puis généralisé par Edsger W. Dijkstra avec les sémaphores

  E. W. Dijkstra. 1968. The structure of the "THE"-multiprogramming system. Commun. ACM 11, 5, pp. 341-346.

Exemple

P1 : crédite le compte d'Igor de 2 euros

```
P.sh compte-igor.lock
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
V.sh compte-igor.lock
```



compte-igor

compte-igor.lock

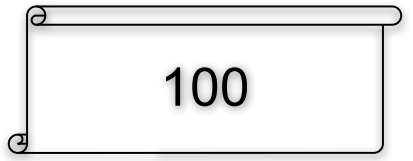
P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

Exemple

P1 : crédite le compte d'Igor de 2 euros

```
P.sh compte-igor.lock
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
V.sh compte-igor.lock
```



compte-igor

compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

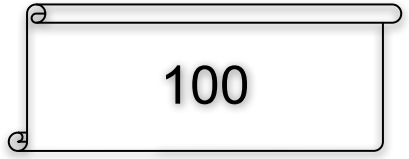
```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

Exemple

P1 : crédite le compte d'Igor de 2 euros

```
P.sh compte-igor.lock
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
V.sh compte-igor.lock
```

a : 100



compte-igor

compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

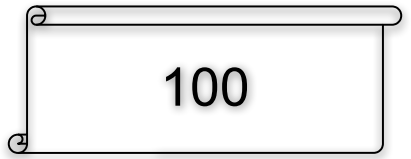
```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

Exemple

P1 : crédite le compte d'Igor de 2 euros

```
P.sh compte-igor.lock
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
V.sh compte-igor.lock
```

a : 102



compte-igor

compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

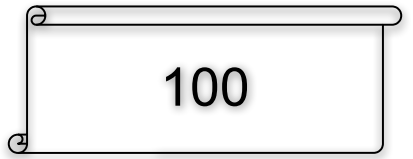
```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

Exemple

P1 : crédite le compte d'Igor de 2 euros

```
P.sh compte-igor.lock
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
V.sh compte-igor.lock
```

a : 102



compte-igor

compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

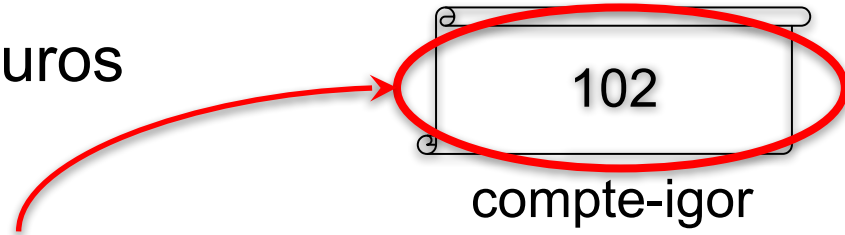
```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

Commutation de P1 vers P2
P2 bloque car verrou pris

Exemple

P1 : crédite le compte d'Igor de 2 euros

```
P.sh compte-igor.lock
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
V.sh compte-igor.lock
```



a : 102

compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

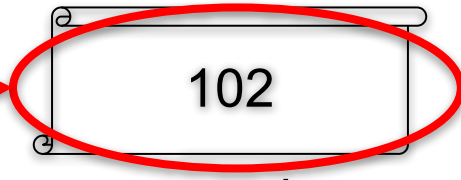
Commutation de P1 vers P2
P2 bloque car verrou pris
⇒ réélection de P1

Exemple

P1 : crédite le compte d'Igor de 2 euros

```
P.sh compte-igor.lock
read a < compte-igor
a=$(expr $a + 2)
echo $a > compte-igor
V.sh compte-igor.lock
```

a : 102



compte-igor

compte-igor.lock

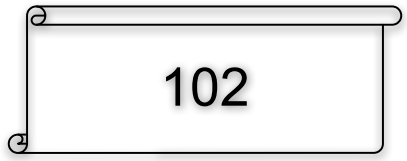
P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

P1 a terminé sa section critique

⇒ déverrouille le verrou

Exemple



compte-igor

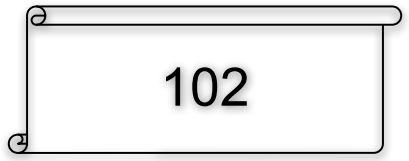
compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

Fin P1
⇒ commutation vers P2
Et P2 prend le verrou

Exemple



compte-igor

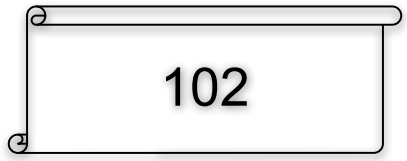
compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

b : 102

Exemple



compte-igor

compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

b : 2

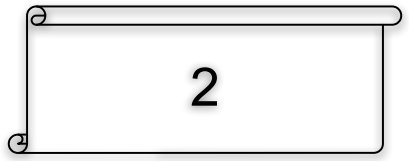
Exemple

P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

b : 2

Exemple



compte-igor

compte-igor.lock

P2 : débite le compte d'Igor de 100 euros

```
P.sh compte-igor.lock
read b < compte-igor
b=$(expr $b - 100)
echo $b > compte-igor
V.sh compte-igor.lock
```

b : 2

Remarques

Un mutex ne sert à rien si une section critique ne le prend pas!

Si on oublie de déverrouiller le mutex en fin de section critique, l'application reste bloquée

Attention à l'inter-blocage

- Inter-blocage : des processus concurrents s'attendent mutuellement
- Blocage définitif de l'avancée du programme
- Exemples :
  - Un processus s'attend lui-même (`wait` sur son PID…)
  - Un processus P1 verrouille V1 puis attend V2, qui est pris par P2 qui attend V1 *(voir diapositives suivantes)*

Attention à l'inter-blocage

- Processus P1

```
P.sh v1.lock
P.sh v2.lock
…
V.sh v2.lock
V.sh v1.lock
```

- Processus P2

```
P.sh v2.lock
P.sh v1.lock
…
V.sh v1.lock
V.sh v2.lock
```

v1.lock

v2.lock

Attention à l'inter-blocage

- Processus P1
  - `P.sh v1.lock`
    `P.sh v2.lock`
    …
    `V.sh v2.lock`
    `V.sh v1.lock`
- Processus P2
  `P.sh v2.lock`
  `P.sh v1.lock`
  …
  `V.sh v1.lock`
  `V.sh v2.lock`


v1.lock


v2.lock

Attention à l'inter-blocage

- Processus P1
  - P.sh v1.lock
  - P.sh v2.lock
  - …
  - V.sh v2.lock
  - V.sh v1.lock
- Processus P2
  - P.sh v2.lock
  - P.sh v1.lock
  - …
  - V.sh v1.lock
  - V.sh v2.lock

Commutation de P1 vers P2

v1.lock

v2.lock

Attention à l'inter-blocage

- Processus P1
  - P.sh v1.lock
  - P.sh v2.lock
  - …
  - V.sh v2.lock
  - V.sh v1.lock
- Processus P2
  - P.sh v2.lock
  - P.sh v1.lock
  - …
  - V.sh v1.lock
  - V.sh v2.lock

P2 bloqué car v1.lock est pris par P1

v1.lock

v2.lock

Attention à l'inter-blocage

- Processus P1

```
P.sh v1.lock
P.sh v2.lock
…
V.sh v2.lock
V.sh v1.lock
```

- Processus P2

```
P.sh v2.lock
P.sh v1.lock
…
V.sh v1.lock
V.sh v2.lock
```

Commutation de P2 vers P1

P1 bloqué car v2.lock est pris par P2

v1.lock

v2.lock

P2 bloqué car v1.lock est pris par P1

⇒ ni P1, ni P2 ne peuvent progresser…

Règle pour éviter l'inter-blocage

Il faut toujours prendre les verrous dans le même ordre dans tous les processus

Notions clés

- Section critique : section de code en exclusion mutuelle
  - Deux sections critiques ne peuvent pas s'exécuter en parallèle

- Mise en œuvre des sections critiques avec des mutex :
  - `P.sh` : entrée en section critique

    Bloque tant qu'il existe un processus en section critique
  - `V.sh` : sortie de section critique

- Attention aux inter-blocages : toujours prendre les mutex dans le même ordre dans tous les processus

À vous de jouer!