Système et programmation système

Julien BERNARD Eric MERLET

Université de Franche-Comté – UFR Sciences et Technique
Licence Informatique – 2è année

2023 – 2024

Première partie

Généralités

Plan de ce cours

Plan

UE Système

Organisation

Équipe pédagogique

Éric Merlet : CM, TD, TP (eric.merlet@univ-fcomte.fr)

Volume

- Cours : 12 x 1h30
- TD : 12 x 1h30
- TP : 12 x 1h30

UE Système

Comment ça marche ?

Mode d'emploi

1. Prenez des notes ! Posez des questions !
2. Comprendre plutôt qu'apprendre

Niveau d'importance des transparents

	trivial	pour votre culture
$\star$	intéressant	pour votre compréhension
$\star\star$	important	pour votre savoir
$\star\star\star$	vital	pour votre survie

Note : les contrôles portent sur *tous* les transparents !

UE Système

Contenu pédagogique

Objectif

Comprendre et manipuler les principes de base d'un système de type Unix

- Système d'exploitation
- Shell, ligne de commande, scripts shell
- Programmation C
- Fichiers
- Processus

UE Système

Evaluation

Évaluation

- 2 devoirs surveillés
- TP : à terminer pour la prochaine séance et à rendre sur Moodle
- 2 projets en TP : projet shell, projet C

2e chance

Le calcul de la moyenne intégrera directement la 2e chance, en modulant les coefficients des deux DS de façon à faire peser le plus grand poids sur le meilleur des deux. Le calcul se fera comme suit :

- 35% sur la note du meilleur des deux DS
- 25% sur la note du moins bon des deux DS
- 40% sur les TP et les projets

UE Système

Bibliographie

Andrew Tanenbaum.
Systèmes d'exploitation.
3è édition, 2008, Pearson

Christophe Blaess.
Développement système sous Linux.
4è édition, 2016, Eyrolles.

Jean-Marie Rifflet, Jean-Baptiste Yunès.
UNIX Programmation et communication.
2003, Dunod.

Plan

Système d'exploitation

Définition

★★★

Définition (Système d'exploitation)

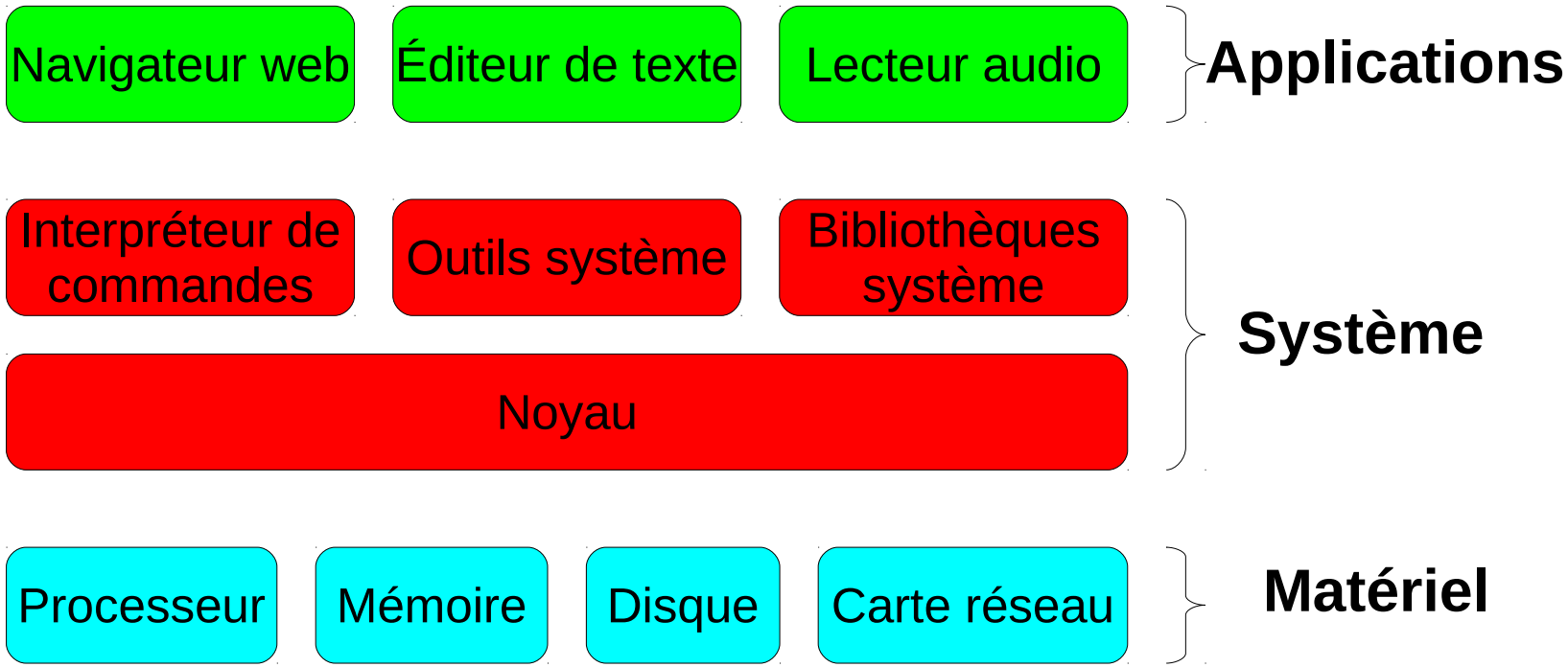
Le système d'exploitation, abrégé SE (en anglais operating system, abrégé OS), est l'ensemble de programmes central d'un appareil informatique qui se place à l'interface entre le matériel et les logiciels applicatifs .
(Source : Wikipedia)

- permet de libérer l'utilisateur de la complexité de la programmation du matériel
- propose une gestion flexible et optimisée des ressources d'un ordinateur (processeur, mémoire centrale, stockage, communication)

Système d'exploitation

Entre les applications et le matériel

★★★



Système d'exploitation

Noyau

★★★

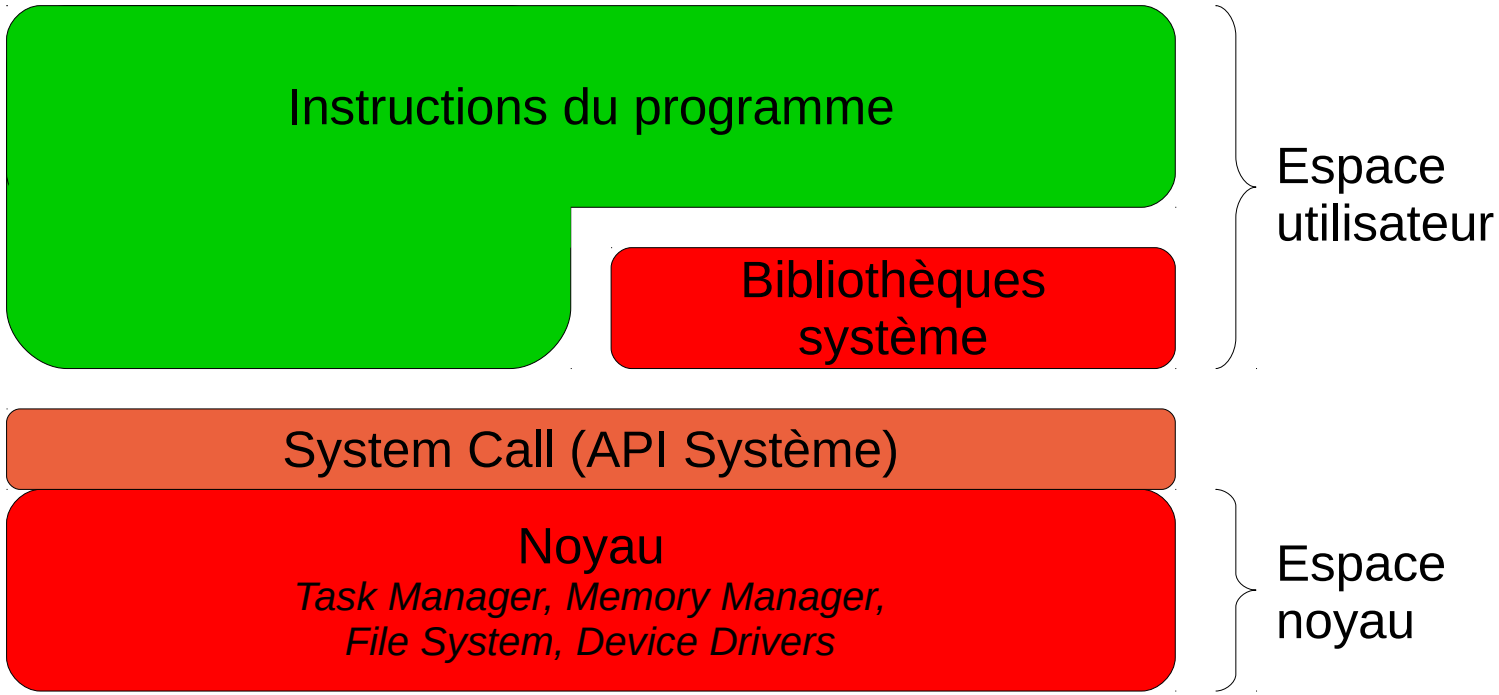
Définition (Noyau)

Le noyau (en anglais kernel) est un ensemble cohérent de routines fournissant des services aux applications, en s'assurant de l'**intégrité** du système.

- L'existence d'un noyau présuppose une partition virtuelle de la mémoire vive physique en deux régions disjointes, l'une étant réservée au noyau (l'espace noyau) et l'autre aux applications (l'espace utilisateur).
- Le noyau fournit des points d'entrée : les appels système.
  L'invocation d'un appel système est une opération assez **coûteuse** car elle implique une commutation du contexte d'exécution du processus.

Système d'exploitation

Espace noyau et espace utilisateur



Système d'exploitation

Typologie

- *multi-tâches* : exécution simultanée de plusieurs programmes
- *multi-utilisateurs* : utilisation simultanée par plusieurs usagers
- *multi-processeurs* : capable d'exploiter plusieurs processeurs
- *temps réel* : temps d'exécution des programmes garanti

Exemples

- DOS : mono-utilisateur, mono-tâche
- Windows 95, OS/2 : mono-utilisateur, multi-tâches
- NT, Linux, Solaris : multi-tâches, multi-utilisateurs, multi-processeurs
- QNX : multi-tâches, multi-utilisateurs, temps réel

Plan

Unix

Logiciel gratuit/payant, libre/propriétaire

Logiciel gratuit vs logiciel payant

Un logiciel gratuit, ou freeware, est un logiciel qui ne nécessite pas de contrepartie financière pour son utilisation.

Logiciel libre vs logiciel propriétaire

Un logiciel est libre si et seulement si son code source est accessible, modifiable, redistribuable , selon les termes d'une licence libre.

Unix

Quelques dates-clefs

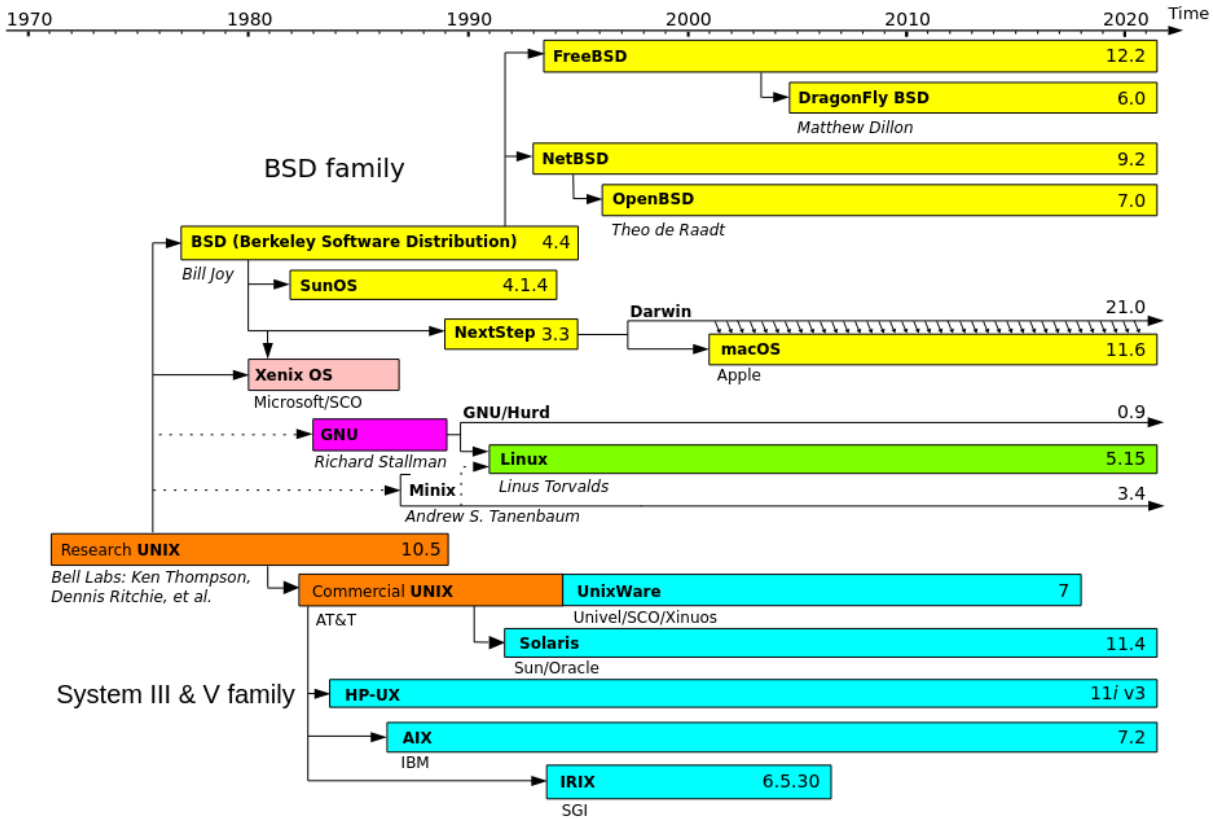
- 1969 : Création d'Unix, Bell Labs (AT&T)
  Ken Thompson, Dennis Ritchie, Brian Kernighan
- 1973 : Création de C et réécriture d'Unix en C (portabilité)
- 1977 : 1BSD (Berkeley Software Distribution), Bill Joy
- 1982 : Sortie de Unix System V (AT&T)
- 1982 : Création de Sun Microsystems, Bill Joy
- 1983 : Création de GNU (GNU's Not Unix), Richard Stallman
- 1987 : Création de Minix, Andrew Tanenbaum
- 1988 : Spécifications POSIX
- 1991 : Création de Linux, Linus Torvalds
- 1994 : Sortie de la 1ère version de GNU/Linux
- 1994 : Sortie de 4.4BSD
- 2001 : Mac OS X, Apple, Steve Jobs

Unix

Unix Timeline

Exemples

- Propriétaire : IBM AIX, Sun Solaris, HP/UX, Mac OS X
- Libre : GNU/Linux, FreeBSD, OpenBSD, NetBSD, Minix



Unix

Philosophie

Définition (Philosophie d'Unix, 1994, Mike Gancarz)

1. La concision est merveilleuse.
2. Écrivez des programmes qui font une *seule chose* mais qui le font bien.
3. Concevez un prototype dès que possible.
4. Préférez la portabilité à l'efficacité.
5. Stockez les données en ASCII.
6. Utilisez les scripts shell pour améliorer la portabilité.
7. Évitez les interfaces utilisateur captives.
8. Faites de chaque programme un filtre.

GNU/Linux

Qu'est-ce que c'est ?

Dates importantes

- 1983 : Création de GNU (GNU's Not Unix), Richard Stallman
- 1984 : Création de la FSF (Free Software Foundation)
- 1991 : Création de Linux, Linus Torvalds

But

Offrir un système d'exploitation libre compatible POSIX

Composants

- GNU : outils système, bibliothèques système
- Linux : noyau

GNU/Linux

Distributions

Définition (Distribution)

Une distribution GNU/Linux est un ensemble cohérent de logiciels (libres) assemblés autour du système d'exploitation GNU/Linux. (Source : Wikipedia)

- Distributions généralistes vs Distributions spécifiques
- Distributions commerciales vs Distributions communautaires

Exemples

- Debian, Ubuntu
- Red Hat, Fedora, OpenSUSE
- Mageia, Gentoo, Arch Linux

GNU/Linux

Paquetage

Définition (Paquetage)

Un paquetage est une archive comprenant les fichiers informatiques, les informations et procédures nécessaires à l'installation d'un logiciel sur un système d'exploitation. (Source : Wikipedia)

Caractéristiques

- Méta-données : description, version, dépendances
- Dépôt : serveur qui stocke les paquets d'une distribution
- Gestionnaire de paquetage : cohérence fonctionnelle du système
- Paquet source vs Paquet binaire

Exemples

- `deb`, dpkg, apt/aptitude

```
# apt update
# apt upgrade
```

- `rpm`, rpm, dnf/yum

Plan

Interfaces

GUI vs CLI

★★

Interface graphique (Graphical User Interface)

- Paradigme WIMP : «window, icon, menu, pointing device»
- Métaphore du bureau
- Communication par mouvement

Interpréteur de commande (Command Line Interface)

- Mode texte
- Communication par langage
- Avantages :
  - Contrôle à distance (via ssh)
  - Automatisation des tâches
  - Faible consommation de ressources

Shell Unix

Familles

sh et dérivés

- sh : Bourne SHell, 1977, Steve Bourne, Version 7 Unix
- ksh : Korn SHell, 1983, David Korn
- bash : Bourne-Again SHell, 1987, Brian Fox, GNU
- ash : Almquist SHell, 1989, Kenneth Almquist
- zsh : Z SHell, 1990, Paul Falstad
- dash : Debian Almquist SHell, 1997, Herbert Xu, Debian

csh et dérivés

- csh : C SHell, 1978, Bill Joy, BSD
- tcsh : TENEX C SHell, 1979, Ken Greer

Interpréteur de commandes

★★★

Principes

Fonctionnement d'un interpréteur de commandes

1. Attente de la commande avec le prompt ($ ou #)
2. Saisie de la commande
3. Analyse puis exécution de la commande avec affichage du résultat

Format des commandes

- Commande élémentaire :
  `$ commande [option]... [fichier_ou_donnée]...`
- Succession de commandes :
  `$ commande1; commande2`

Exemple

```
$ echo -n "Bonjour, nous sommes le "; date
```

Interpréteur de commandes

Erreurs courantes

★★

Exemples

- Commande introuvable

```
$ cag
cag: not found
```

- Aucun fichier ou dossier de ce type

```
$ cat inconnu.txt
cat: inconnu.txt: No such file or directory
```

- Option non valide

```
$ cat -w
cat: invalid option - 'w'
```

- Permission non accordée

```
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Interpréteur de commandes

★★

Quelques commandes de base

Exemples

- `$ whoami`
  Afficher le nom de l'utilisateur courant
- `$ uname [-snrvma]`
  Afficher des informations sur le système
- `$ uptime`
  Indiquer depuis quand le système a été mis en route
- `$ date`
  Afficher ou configurer la date et l'heure du système
- `$ cal [[month] year]`
  Afficher un calendrier
- `$ echo [-n] [string]...`
  Afficher une ligne de texte

Plan

Page de manuel

★★★

Qu'est-ce que c'est ?

Définition (Page de manuel)

Une page de manuel (manpage) est une documentation complète relative à un programme, une fonction, un fichier, etc. Les pages de manuel sont rangées dans différentes sections. Le nom d'une page de manuel est de la forme `page(section)`.
RTFM !

Commande `man`

```
$ man [section] page
```
Afficher la page de manuel `page` de la section `section`

Exemples

```
$ man mkdir
```
Afficher la page de manuel `mkdir(1)` de la commande `mkdir`
```
$ man 2 mkdir
```
Afficher la page de manuel `mkdir(2)` de l'appel système `mkdir`
```
$ man send
```
Afficher la page de manuel `send(2)` de l'appel système `send`

Page de manuel

Sections de pages de manuel

★★

Sections des pages de manuel

1. Commandes utilisateur
2. Appels système
3. Fonctions de bibliothèque
4. Fichiers spéciaux
5. Formats de fichier
6. Jeux
7. Divers
8. Administration système
9. Interface du noyau Linux

Page de manuel

Exemple : $ man cp

Exemple

```
CP(1)                                                    User Commands

NAME
  cp - copy files and directories

SYNOPSIS
  cp [OPTION]... [-T] SOURCE DEST
  cp [OPTION]... SOURCE... DIRECTORY
  cp [OPTION]... -t DIRECTORY SOURCE...

DESCRIPTION
  Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

  Mandatory arguments to long options are mandatory for short options too
  -R, -r, --recursive
        copy directories recursively
  -t, --target-directory=DIRECTORY
        copy all SOURCE arguments into DIRECTORY
```

Aide sur les commandes

--help et autres commandes

--help

Option disponible pour (presque) toutes les commandes. Exemple :

```
$ cp --help
Usage: cp [OPTION]... [-T] SOURCE DEST
  or:  cp [OPTION]... SOURCE... DIRECTORY
  or:  cp [OPTION]... -t DIRECTORY SOURCE...
Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY. etc...
```

Autres commandes d'aide

- `$ whatis name`
  Recherche les pages de manuel dont le nom correspond à name, et affiche leur numéro de section et leur description courte
- `$ apropos keyword`
  Chercher dans le nom et la description courte des pages de manuel
- `$ info [item]`
  Lire le document Info (alternative à `man`)

Deuxième partie

Système de fichiers

Plan de ce cours

4. Système de fichiers
   - Fichiers
   - Répertoires
   - Information et navigation
   - Inodes et structure sur le disque
   - Opérations de base
   - Utilisateurs et permissions

Plan

Qu'est-ce qu'un fichier ?

★★★

Définition (Fichier)

Un fichier est un lot d'*informations* portant un *nom*.

Tout est fichier

Sous Unix, tout est fichier :

- Les fichiers «physiques» ou fichiers réguliers
- Les fichiers «virtuels»
- Les répertoires
- Les périphériques (disques durs, terminaux, etc.)
- Les tubes (Voir le chapitre «Shell et scripts shell»)
- Les sockets (Voir le cours «Réseaux» en L3)

Opérations sur les fichiers ⋆⋆

Opérations

Sur tous les fichiers, on peut réaliser les opérations suivantes :

- Ouverture : prévient le système qu'on commence à utiliser un fichier
- Fermeture : prévient le système qu'on a terminé d'utiliser un fichier
- Lecture : lit des informations depuis un fichier
- Écriture : écrit des informations dans un fichier

Voir le chapitre «Manipulation de fichiers»

Nom de fichier

★★

Nom de fichier

Un nom de fichier est formé d'un *nom* décrivant généralement son contenu et d'une (ou plusieurs ou zéro) *extension* qui indique son type. L'ensemble du nom doit être inférieur à 255 caractères. Il y a une différence entre majuscule et minuscule !

Fichier caché

Un fichier dont le nom commence par '.' est un fichier caché.

Bonnes pratiques de nommage

- Utiliser uniquement des lettres, des chiffres et '_', '-', '.'
- Ne pas utiliser d'espace !
- Ne pas utiliser d'accent !
- Ne pas nommer deux fichiers de manière identique à la casse près !

Exemples de fichiers

Exemples

- `README`
  fichier texte (sans extension) souvent présent pour décrire le contenu du répertoire courant et des sous-répertoires
- `.bashrc`
  fichier caché de configuration de `bash` présent dans le répertoire privé de l'utilisateur
- `systeme.pdf`
  le fichier qui me sert pour cette présentation
- `gmp-5.0.1.tar.bz2`
  archive (`.tar`) compressée à l'aide de `bzip2` (`.bz2`) des sources de la bibliothèque GMP (GNU Multiple Precision Arithmetic Library) en version 5.0.1 $\rightarrow$ il y a donc ici deux extensions et non quatre.

$\rightarrow$ on ne peut pas faire confiance à l'extension pour connaître le type d'un fichier régulier, il faut utiliser `file(1)`.

Plan

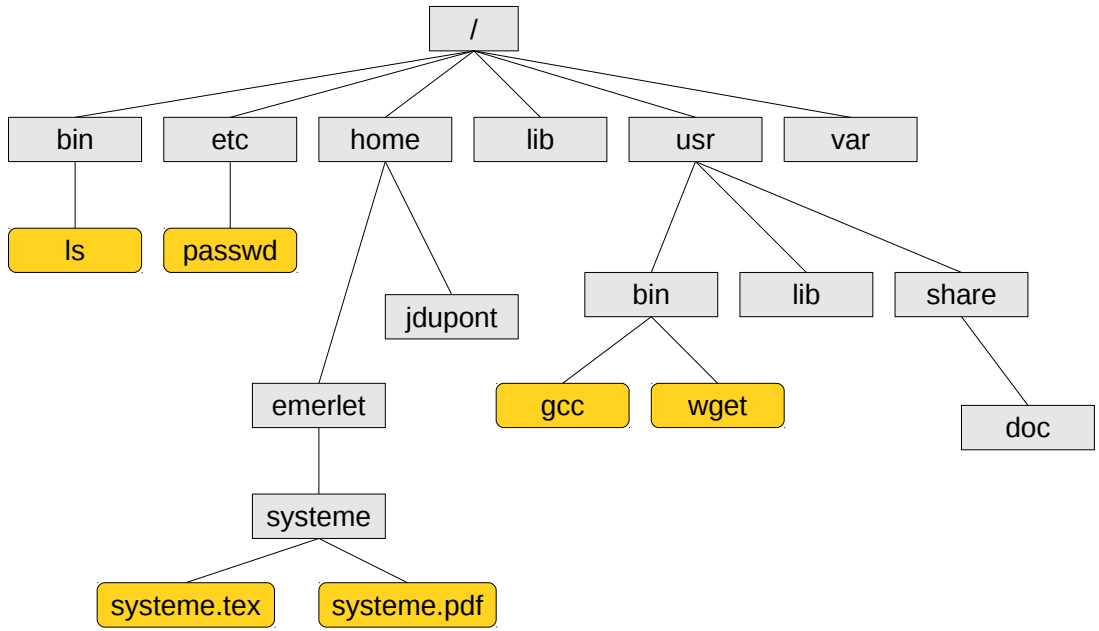
Arborescence des fichiers d'un système

★★★

Généralités

Arborescence des fichiers d'un système

Tous les fichiers d'un système Unix, éventuellement constitué de plusieurs systèmes de fichiers (i.e. partitions), sont "rassemblés" dans un seul arbre dont la racine est le répertoire /. Le séparateur de répertoire est /.



Filesystem Hierarchy Standard ⋆⋆

Définition (Filesystem Hierarchy Standard)

Le Filesystem Hierarchy Standard définit l'arborescence et le contenu des principaux répertoires des systèmes de fichiers de GNU/Linux et de la plupart des Unix. Exception notable : Mac OS X.

Répertoires dans /

- `/bin/` (*BINaries*) : commandes(exécutables) de base pour les utilisateurs nécessaires en mode mono-utilisateur pour démarrer ou réparer le système.
- `/boot/` : chargeur d'amorçage (*bootloader*) et noyau
- `/dev/` (*DEVice*) : fichiers de périphériques physiques et virtuels
- `/etc/` (*Editing Text Configuration*) : fichiers de configuration
- `/home/` : répertoires privés des utilisateurs
- ...

Filesystem Hierarchy Standard ⋆⋆

Répertoires dans /

- `/lib/` (*LIBrary*) : bibliothèques nécessaires pour `/bin` et `/sbin`
- `/mnt/` (*MouNT*) : points de montage temporaire
- `/opt/` (*OPTional*) : logiciels optionnels
- `/proc/` (*PROCess*) : informations sur les processus et le noyau
- `/root/` : c'est le répertoire privé de l'utilisateur `root`
- `/run/` : fichiers transitoires
- `/sbin/` (*System BINaries*) : commandes pour les administrateurs
- `/tmp/` (*TeMPorary*) : fichiers temporaires
- `/usr/` (*User System Resources*) : similaire à `/`
- `/var/` (*VARiable*) : fichiers de données variables

RTFM : `hier(7)`

Filesystem Hierarchy Standard

`/dev/`

Fichiers de périphériques physiques

- `/dev/dsp` : carte son
- `/dev/fd0` : lecteur de disquettes
- `/dev/sd*` ou `/dev/hd*` : disques durs, partitions
- `/dev/sr*` : lecteurs CDROM/DVDROM
- `/dev/tty*` : terminaux en dehors d'une interface graphique

Fichiers de périphériques virtuels

- `/dev/random` : générateur de nombres (vraiment) aléatoires
- `/dev/urandom` : générateur de nombres (pas tout à fait) aléatoires
- `/dev/null` : «trou noir» (plutôt en écriture)
- `/dev/zero` : flux de zéros (plutôt en lecture)
- `/dev/pts/*` : terminaux au sein d'une interface graphique

RTFM : `random(4)`, `null(4)`

Filesystem Hierarchy Standard

/etc/

/etc/

Quelques fichiers importants :

- /etc/passwd : informations sur les comptes utilisateurs. Format :

  login:password:UID:GID:name:home:shell

- /etc/shadow : mots de passe chiffrés
- /etc/group : groupes du système. Format :

  group_name:password:GID:user_list

- /etc/fstab : informations sur les systèmes de fichiers
- /etc/services : liste des services réseau Internet
- /etc/issue : message de bienvenue lors d'une connexion

RTFM : passwd(5), shadow(5), group(5), fstab(5), services(5), issue(5)

Filesystem Hierarchy Standard

/home/

Répertoire personnel d'un utilisateur

Le répertoire personnel d'un utilisateur est le répertoire où l'utilisateur stocke ses fichiers. Sous Unix, il est généralement situé dans le répertoire /home. C'est le répertoire dans lequel on se trouve après connexion. Exemple : le répertoire personnel de l'utilisateur emerlet est /home/emerlet/

Notations

- ~/ (ou $HOME) sont des notations pour le répertoire personnel de l'utilisateur connecté
- ~jdupont/ est une notation pour le répertoire personnel de Jean Dupont (généralement, /home/jdupont/)

Filesystem Hierarchy Standard

/usr/

/usr/

Arborescence de répertoires et fichiers utilitaires accessibles aux utilisateurs "normaux" du système.

- /usr/bin/, /usr/lib/, /usr/sbin/ : arborescence similaire à celle de /
- /usr/include/ : fichiers d'entête des bibliothèques (Voir le chapitre «Développement en C»)
- /usr/share/ : fichiers indépendants de la plateforme, notamment :
  - /usr/share/man/ : emplacement des manpages
  - /usr/share/doc/ : documentation du système
- /usr/local/ : comme /usr/ pour les programmes spécifiques à la machine
  - /usr/local/bin/, /usr/local/lib/, /usr/local/sbin/, etc.

Filesystem Hierarchy Standard

/var/

/var/

- /var/log/ : fichiers de journalisation
- /var/lock/ : fichiers de verrouillage
- /var/run/ : fichiers temporaires des logiciels en cours d'exécution
- /var/mail/ : boîtes aux lettres utilisateurs
- /var/spool/ : files d'attente des services
- /var/spool/mail/ : mails en cours de transit sur la machine
- /var/www/ : répertoire web

Répertoires . et . . et chemin ★★★

Les répertoires cachés . et . .

Dans tous les répertoires, il existe deux répertoires cachés :

- . représente le répertoire courant ;
- . . représente le répertoire parent, c'est-à-dire le répertoire qui contient le répertoire courant.

Cas particulier de la racine /

Le répertoire parent de / est lui-même.

Chemin relatif et chemin absolu ★★★

Définition (Chemin)

Le chemin (*path*) d'un fichier (ou d'un répertoire) est une chaîne de caractères décrivant sa position dans le système de fichier. Il existe :

- les chemins **relatifs**, c'est-à-dire exprimés par rapport au répertoire courant ;
- les chemins **absolus**, c'est-à-dire exprimés par rapport à la racine.

Distinction entre chemin relatif et chemin absolu

- Un chemin absolu est un chemin qui commence par un `/`.
  Exemple : `/rep1/rep2/rep3`
  → Le répertoire `rep1` doit appartenir au répertoire racine `/`.
- Un chemin relatif est un chemin qui ne commence pas par un `/`.
  Exemple : `rep1/rep2/rep3`
  → Le répertoire `rep1` doit appartenir au répertoire courant.

Exemples de chemin

Exemples

On considère que le répertoire courant est `/home/emerlet/`

- `/bin/ls` est un chemin absolu vers la commande (exécutable) `ls`
- `./` est un chemin relatif vers le répertoire courant : `/home/emerlet/`
- `../` est un chemin relatif vers le répertoire parent du répertoire courant : `/home`
- `../../bin/ls` est un chemin relatif vers la commande `ls`
- `./systeme/../../emerlet/.././emerlet/` est un chemin relatif (inutilement compliqué) vers le répertoire courant : `/home/emerlet/`

Plan

Système de fichiers

Information

Commandes d'information

- `pwd` : *Print Working Directory*
  Affiche le chemin absolu canonique du répertoire courant
- `ls [OPTION]... [FILE]...` : *LiSt*
  Liste les informations des fichiers (par défaut, elle liste le contenu (les fichiers non cachés) du répertoire courant)
  - `ls -a` : liste tous les fichiers, y compris les fichiers cachés
  - `ls -l` : affiche les détails des fichiers. Une ligne par fichier avec :
    - un premier caractère qui indique le type du fichier
    - les permissions liées à ce fichier (les 9 caractères suivants)
    - le nombre de liens durs qui pointent sur l'inode
    - l'utilisateur et le groupe propriétaires du fichier
    - la taille du fichier
    - la date et l'heure de la dernière modification
    - le nom du fichier

Système de fichiers

Information

Commande `ls -l` : signification du premier caractère (le type de fichier)

- `-` : fichier régulier
- `d` (*Directory*) : répertoire
- `b` (*Block*) : fichier de périphérique en mode bloc
- `c` (*Character*) : fichier de périphérique en mode caractère
- `l` (*Link*) : lien symbolique
- `p` (*Pipe*) : tube nommé (ou fifo)
- `s` (*Socket*) : socket

Commande `ls` utilisée avec un répertoire `dir`

- `ls dir` : liste les fichiers (non cachés) contenus dans le répertoire `dir`
- `ls -d dir` : l'option `-d` permet d'afficher les répertoires avec la même présentation que les fichiers réguliers, sans lister leur contenu.

Système de fichiers

Information

Exemples

```
$ pwd
/home/emerlet/Bazaar/emerlet/lectures/systeme/cours
$ ls -l .
total 392
drwxr-xr-x 2 emerlet emerlet   4096  5 janv. 00:33 bits
drwxr-xr-x 2 emerlet emerlet   4096 16 janv. 12:51 contrib
-rw-r--r-- 1 emerlet emerlet    266 18 janv. 15:39 Makefile
drwxr-xr-x 4 emerlet emerlet   4096 18 janv. 17:18 media
-rw-r--r-- 1 emerlet emerlet 374865 19 janv. 16:20 systeme.pdf
-rw-r--r-- 1 emerlet emerlet   1957 19 janv. 11:35 systeme.tex
$ ls -ld .
drwxr-xr-x 5 emerlet emerlet   4096  5 janv. 00:33 .
$ ls -l Makefile
-rw-r--r-- 1 emerlet emerlet    266 18 janv. 15:39 Makefile
```

Système de fichiers

Navigation

Commandes de navigation

- `cd [dir]` : *Change Directory*
  Se déplacer dans le répertoire `dir` (le chemin peut être relatif ou absolu)
- `cd` : *Change Directory*
  Sans répertoire passé en argument ⇒ se déplacer dans le répertoire personnel de l'utilisateur connecté (équivalent à `cd ~`)

Exemples

```
$ cd
$ pwd
/home/emerlet
$ ......
$ pwd
/home
$ ......
$ pwd
/usr/bin
```

Plan

Système de fichiers

Définition

★★

Définition (Système de fichiers)

Une partition d'un disque dur (i.e. disque logique) constitue un *système de fichiers* qui est caractérisé par son *type*. Le type d'un système de fichiers définit l'organisation du disque logique.
Chaque système de fichiers est organisée en arborescence.

Exemples (Types de système de fichiers)

ext2, ext3, ext4, ntfs, fat

Contenu d'un système de fichiers de type *Unix*

- un bloc d'initialisation
- un superbloc
- des blocs de données
- une *table des inodes*

Inode

Définition

★★★

Définition (Inode)

Un inode (*index node*) est une structure de données contenant des informations concernant un fichier stocké dans un système de fichiers. À chaque fichier correspond un numéro d'inode (*i-number*) dans le système de fichiers dans lequel il réside, **unique** au périphérique sur lequel il est situé.
Les inodes sont créés lors de la création du système de fichiers.

Numéro d'inode d'un fichier

Pour connaître le numéro d'inode d'un fichier,

- `ls -i file` :
  Donne uniquement le numéro d'inode du fichier `file`
- `stat file` : *STATus*
  Donne le numéro d'inode et des informations contenues dans l'inode du fichier `file`

RTFM : `stat(1)`

Inode

Informations contenues dans un inode (1/2)

★★

Informations contenues dans un inode (1/2)

- Identifiant du périphérique
- Numéro d'inode
- Permissions du fichier
- Compteur indiquant le nombre de liens physiques (durs) sur cet inode.
- Identifiant de l'utilisateur propriétaire du fichier
- Identifiant du groupe propriétaire du fichier
- Taille du fichier
- Trois dates :
  - Date de dernier accès `atime` (*Access TIME*)
  - Date de dernière modification des données `mtime` (*Modify TIME*)
  - Date de dernière modification du inode `ctime` (*Change TIME*)
- Des adresses de blocs de données contenant les données du fichier

Inode

Informations contenues dans un inode (2/2)

★★

Informations contenues dans un inode (2/2)

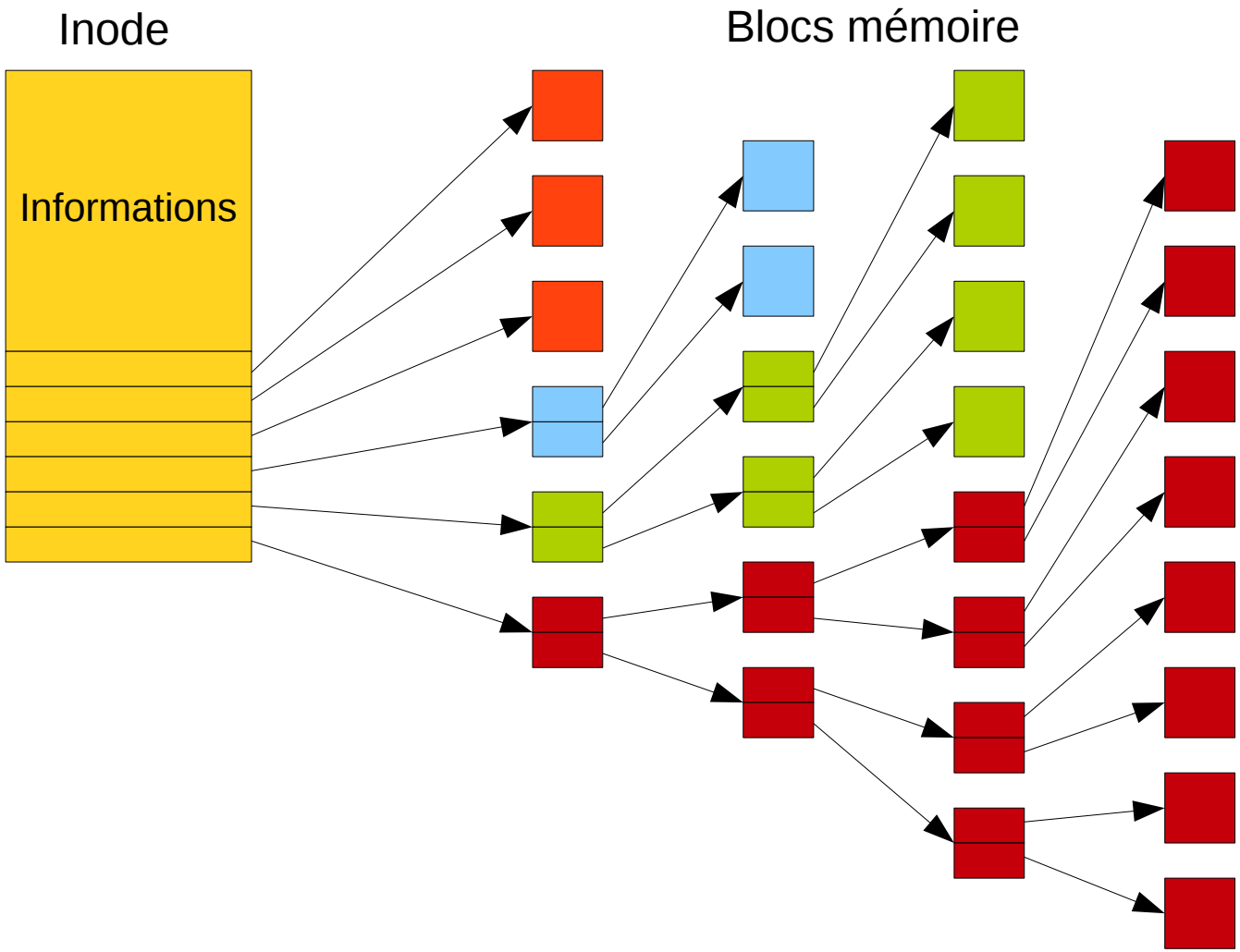
Dans système de fichier de type ext2, un inode de fichier contient 15 adresses de blocs de données :

- Les 12 premiers blocs pointées contiennent "directement" des données du fichier
- Le 13è bloc pointé est un bloc d'indirection simple qui contient des adresses de blocs de données du fichier
- Le 14è bloc pointé est un bloc de double-indirection qui contient des adresses de blocs d'indirection simple
- Le 15è bloc pointé est un bloc de triple-indirection qui contient des adresses de blocs de double-indirection

→ Il n'y a pas le nom du fichier !

Inode

Repésentation des 15 adresses de blocs mémoire



Inode de fichier régulier

★★

Inode de fichier régulier

Les fichiers réguliers (ou ordinaires) contiennent des octets sans organisation particulière. La structuration de ce contenu est laissée aux applications. Une caractéristique essentielle d'un fichier régulier est sa **taille** : c'est par son intermédiaire que le système détecte la fin du fichier.

Inode d'un répertoire ★★★

Un inode de répertoire contient des liens physiques

Les répertoires sont également des "fichiers de données", mais leur contenu a une *structure logique*. Un inode de répertoire pointe vers un bloc de données constitué d'une liste d'"entrées" : chaque entrée est un lien physique qui associe un numéro d'inode avec un nom. Ce nom est un "nom de fichier" pour le fichier identifié par le numéro d'inode.

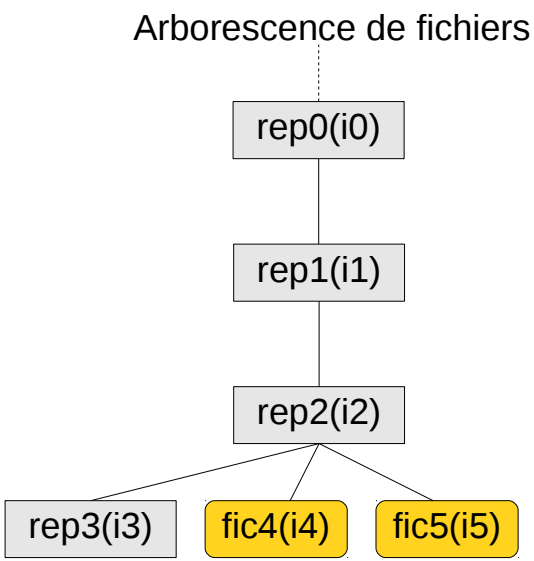
Retour sur les répertoires `.` et `..`

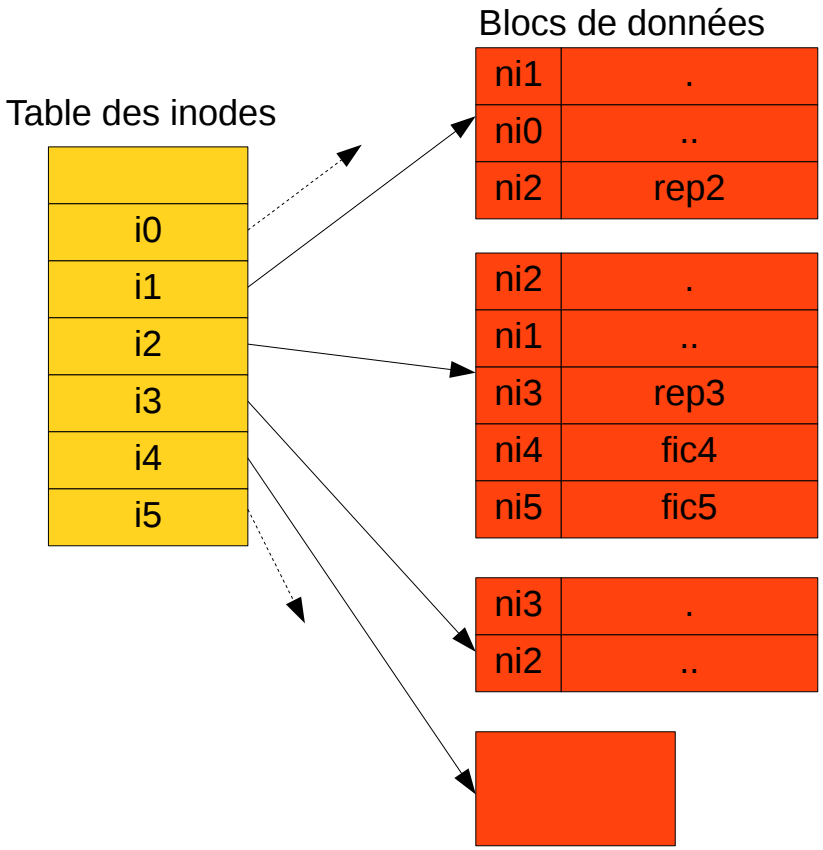
Un répertoire Unix nommé "`rep`" n'est jamais vide au sens physique du terme, puisqu'il contient toujours au moins les liens physiques `.` et `..` :

- `.` est associé au numéro d'inode du répertoire `rep`.
- `..` est associé au numéro d'inode du répertoire père du répertoire `rep`.

Inode d'un répertoire

Exemple





Inode d'un répertoire

Exemple (commentaires)

Nombre de liens durs pointant sur un inode de répertoire

- Deux liens pointent sur l'inode du répertoire vide i3 :
  - (ni3, `rep3`) dans son répertoire père
  - (ni3, `.`) dans lui-même
- Trois liens pointent sur l'inode du répertoire i2 :
  - (ni2, `rep2`) dans son répertoire père
  - (ni2, `.`) dans lui-même
  - (ni2, `..`) dans son répertoire fils rep3.

$\rightarrow$ Le nombre de liens (pointant) sur un inode de répertoire contenant n répertoires fils est égal à n+2.

"Visualisation" des entrées contenues dans un répertoire

La commande `ls -ia dir` permet d'avoir une "image" du contenu du répertoire `dir`.

Montage de systèmes de fichiers ⋆⋆

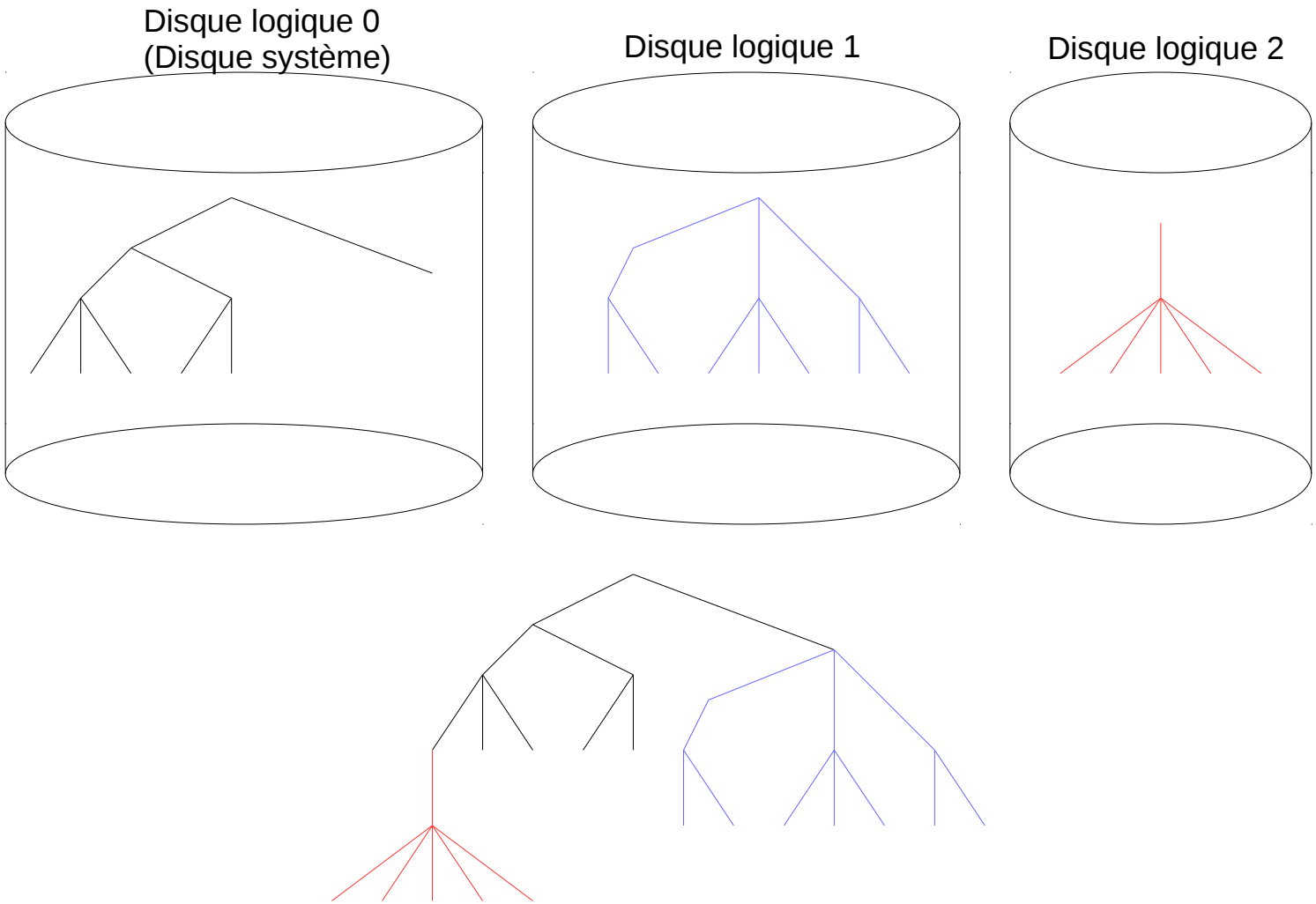
Montage de systèmes de fichiers

- Chaque système de fichiers constitue en lui-même une arborescence de fichiers.
- Parmi tous les disques logiques, le *disque système* est privilégié : lors du démarrage de la machine, il est monté en premier et sa racine est la racine absolue de l'arborescence des fichiers du système. (/).
- Les arborescences des autres diques logiques sont reliées entre elles par le mécanisme de *montage* : il consiste à greffer en un point, accessible depuis la racine absolue (/), la racine d'une arborescence pas encore montée.

Voir la partie de ce cours : «Administration système»

Montage de systèmes de fichiers

Illustration



Plan

Lien dur/Lien symbolique ★★★

Définition (Lien dur)

Rappel : un lien physique (*hard link*) est un nom associé à un numéro d'inode pour former une entrée de répertoire. Un lien physique pointe sur l'inode dont le numéro lui est associé.

Définition (Lien symbolique)

Un lien symbolique (*symlink*) est un fichier sur disque dont le contenu est interprété comme un chemin vers un fichier, qui permet de créer un alias vers le fichier.

Commande de création de lien

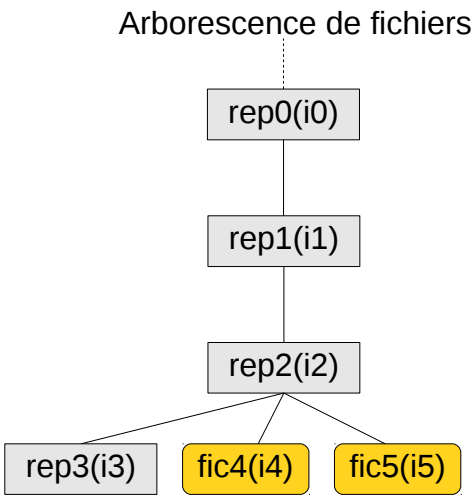
- `ln target link` : *LiNk*
  Crée un lien physique appelé `link` vers l'inode du fichier `target`
- `ln -s target link` : *LiNk*
  Crée un lien symbolique appelé `link` vers `target`

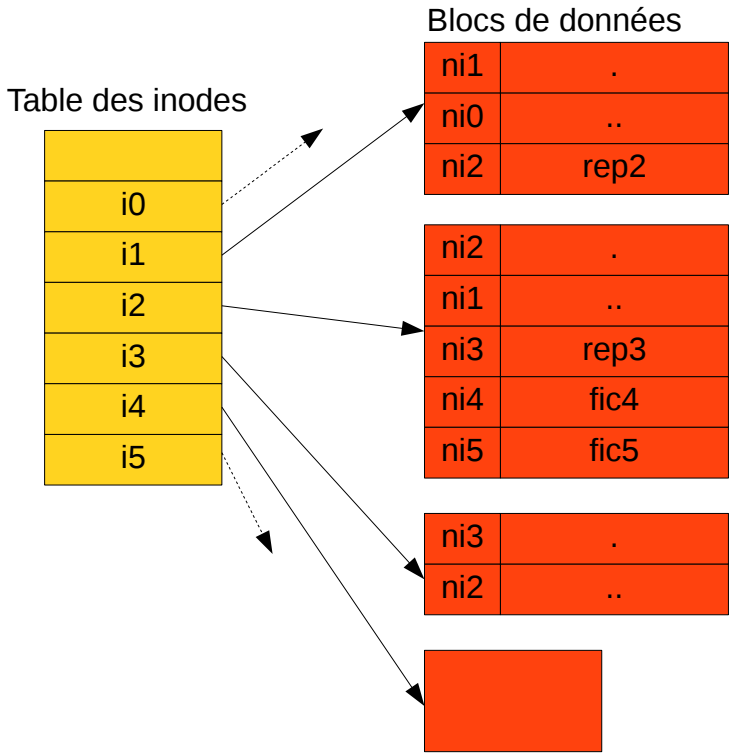
RTFM : `ln(1)`

Création de liens (1/3)

Avant création de liens

```
$ cd ~/rep0/rep1/rep2/; ls -il
total 4
20186858 -rw-rw-r--. 1 eric eric    0 9 janv. 8:19 fic4
20186859 -rw-rw-r--. 1 eric eric    0 9 janv. 8:19 fic5
20186857 drwxrwxr-x. 2 eric eric 4096 9 janv. 8:22 rep3
```

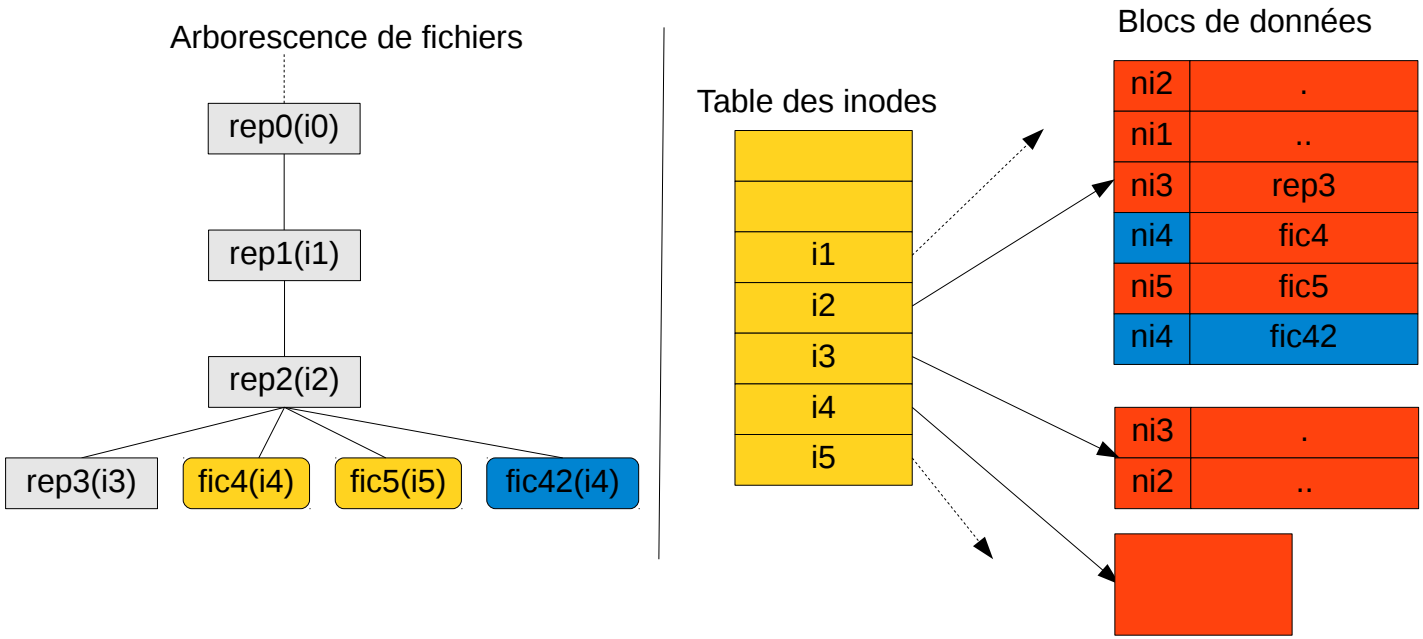




Création de liens (2/3)

Création d'un lien dur vers l'inode de fic4 (de numéro ni4)

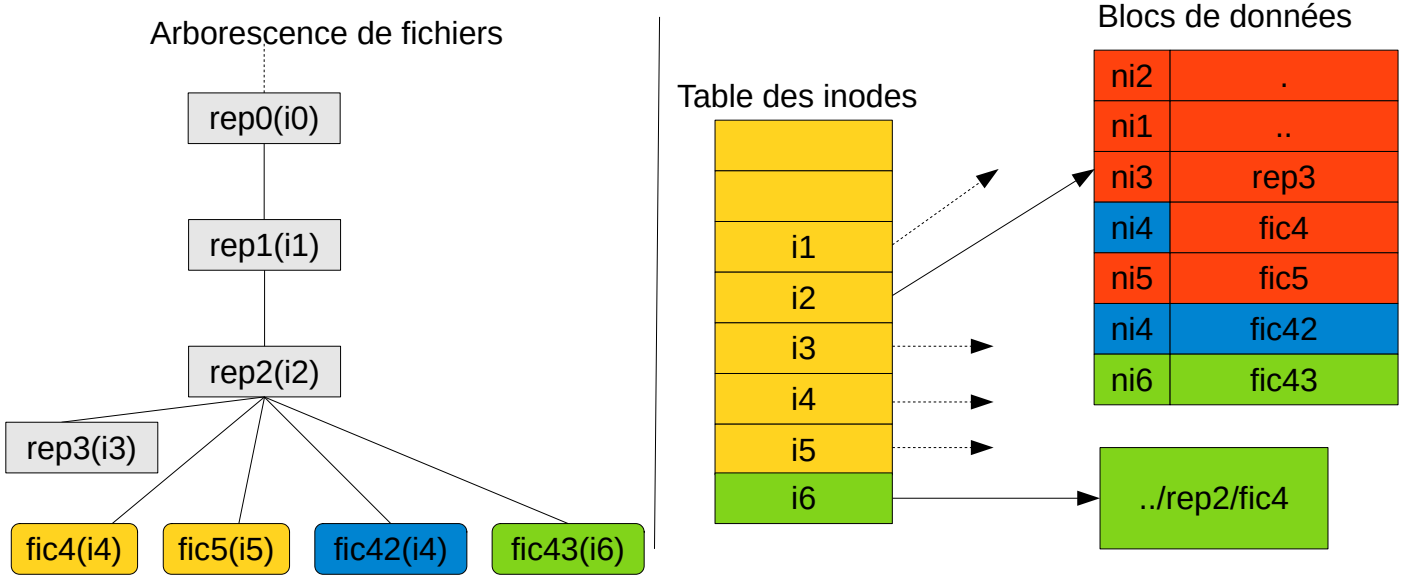
```
$ ln fic4 fic42
$ ls -il
total 4
20186858 -rw-rw-r--. 2 eric eric    0 9 janv. 8:19 fic4
20186858 -rw-rw-r--. 2 eric eric    0 9 janv. 8:19 fic42
20186859 -rw-rw-r--. 1 eric eric    0 9 janv. 8:19 fic5
20186857 drwxrwxr-x. 2 eric eric 4096 9 janv. 8:22 rep3
```



Création de liens (3/3)

Création d'un lien symbolique vers fic4

```
$ ln -s ../rep2/fic4 fic43
$ ls -il
total 4
20186858 -rw-rw-r--. 2 eric eric    0 9 janv. 8:19 fic4
20186858 -rw-rw-r--. 2 eric eric    0 9 janv. 8:19 fic42
20186707 lrwxrwxrwx. 1 eric eric   12 9 janv. 8:50 fic43 -> ../rep2/fic4
20186859 -rw-rw-r--. 1 eric eric    0 9 janv. 8:19 fic5
20186857 drwxrwxr-x. 2 eric eric 4096 9 janv. 8:22 rep3
```



Gestion de fichiers (1/2) ⋆⋆

Commandes de gestion de fichiers

- `cp [OPTION]... [-T] SOURCE DEST` : *CoPy*
  Copie le fichier `SOURCE` vers le fichier `DEST`
- `cp [OPTION]... SOURCE... DIRECTORY` : *CoPy*
  Copie les fichiers `SOURCE`...dans le répertoire `DIRECTORY`
- `mv [OPTION]... [-T] SOURCE DEST` : *MoVe*
  Déplace/renomme le lien physique `SOURCE` en `DEST`
  Si le nouveau lien appartient au même disque logique que l'ancien, l'inode correspondant n'est en aucun cas déplacé : il s'agit d'un simple "jeu" sur les liens physiques.
- `mv [OPTION]... SOURCE... DIRECTORY` : *MoVe*
  Déplace les fichiers `SOURCE`...dans le répertoire `DIRECTORY`

RTFM : `cp(1)`, `mv(1)`

Gestion de fichiers (2/2) ★★

Commandes de gestion de fichiers

- `unlink file` :
  Supprime le lien physique `file` et éventuellement supprime physiquement le fichier associé (s'il s'agit du dernier lien physique pointant sur l'inode correspondant).
  La suppression physique d'un fichier correspond à la libération du inode et à celle des blocs de données qui lui sont alloués.
- `rm [OPTION]... [file]...` : *ReMove*
  Idem `unlink`
  La commande `rm` possède plus d'options.
- `touch [OPTION]... file...`
  Modifie la date d'accès et la date de modification de chaque fichier indiqué, avec la date courante.
  Les fichiers n'existant pas sont créés (sauf si l'option -c est utilisée).

RTFM : `unlink(1)`, `rm(1)`, `touch(1)`

Système de fichiers

Gestion de répertoires

★★

Commandes de gestion de répertoires

- `mkdir dir` : *MaKe DIRectory*
  Crée le répertoire `dir` dans le répertoire courant
- `mkdir -p dir` : *MaKe DIRectory*
  Crée le répertoire `dir` et tous les répertoires intermédiaires
- `rmdir dir` : *ReMove DIRectory*
  Efface le répertoire vide `dir` du répertoire courant
- `rmdir -p dir` : *ReMove DIRectory*
  Efface le répertoire vide `dir` et tous les répertoires intermédiaires

RTFM : `mkdir(1)`, `rmdir(1)`

Plan

4. Système de fichiers
   - Fichiers
   - Répertoires
   - Information et navigation
   - Inodes et structure sur le disque
   - Opérations de base
   - Utilisateurs et permissions

Fichiers, utilisateurs, groupes et permissions ★★★

Fichiers, utilisateurs, groupes et permissions

Chaque fichier (ou répertoire) du système de fichiers possède :

- un utilisateur propriétaire et un groupe propriétaire du fichier
- un ensemble de permissions relatives à la lecture, l'écriture et l'éxécution du fichier

Exemple

```
$ ls -l Makefile
-rw-r--r-- 1 emerlet emerlet    266 18 janv. 15:39 Makefile
```

→ Comment s'organisent ces éléments les uns avec les autres ?

Utilisateurs et groupes

★★★

Utilisateur et UID

- Toute entité (personne physique ou programme) devant interagir avec un système Unix est authentifiée par un utilisateur (*user*).
- Un nom d'utilisateur et un numéro unique appelé numéro d'utilisateur (UID, *User ID*) sont associés à chaque utilisateur.
- La correspondance entre le nom d'utilisateur et le numéro d'utilisateur se trouve dans le fichier `/etc/passwd`.

Groupe et GID

- Chaque utilisateur fait partie d'un (ou plusieurs) groupe(s) (*group*).
- Un nom de groupe et un numéro unique appelé numéro de groupe (GID, *Group ID*) sont associés à chaque groupe.
- La correspondance entre le nom de groupe et le numéro de groupe se trouve dans le fichier `/etc/group`.

Le super-utilisateur root ⋆⋆

Le super-utilisateur root

Il existe un utilisateur particulier appelé super-utilisateur ou root. Il possède tous les pouvoirs sur le système :

- Il peut lire, écrire et exécuter n'importe quel fichier.
- Il peut prendre l'identité de n'importe quel utilisateur.

L'utilisateur root a pour UID le numéro 0.

Le groupe root

Le groupe du super-utilisateur root se nomme également root et a pour GID le numéro 0. Généralement, seul l'utilisateur root fait partie du groupe root.

Permissions

★★★

Les trois permissions

Chaque fichier ou répertoire possède trois types de permissions :

- Lecture (`r`, *Read*)
  - Fichier régulier : lire le contenu du fichier
  - Répertoire : lister le contenu du répertoire
- Écriture (`w`, *Write*)
  - Fichier régulier : écrire le contenu du fichier
  - Répertoire : créer, supprimer, changer le nom des fichiers du répertoire
- Exécution (`x`, *eXecute*)
  - Fichier régulier : exécuter le fichier
  - Répertoire : accéder au répertoire (ie. le faire figurer dans un chemin, ou s'y positionner (en faire son répertoire de travail))

Représentation symbolique

On représente les permissions par un triplet de lettres représentant la permission (`rwx`) ou un tiret (`-`) pour l'absence de permission.

Permissions

★★★

Application des permissions

Pour chaque fichier, les systèmes Unix permettent de fixer les permissions (`Read, Write, eXecute` : `rwx`) :

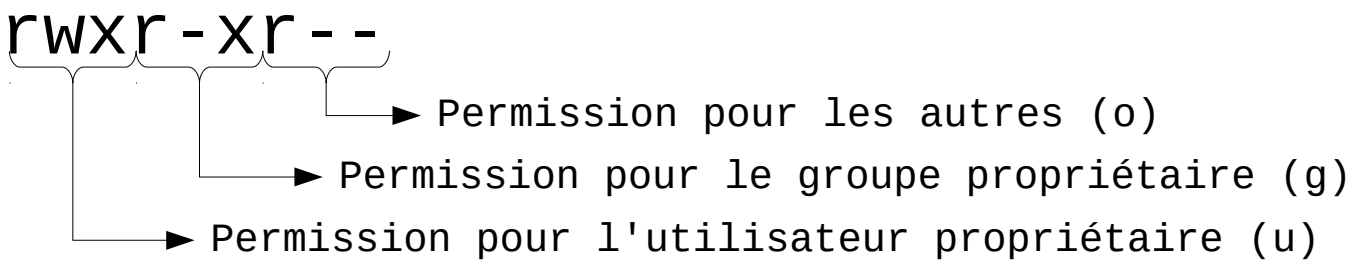
- de l'utilisateur propriétaire du fichier : (`u`, *User*)
- des membres du groupe propriétaire du fichier, excepté le propriétaire : (`g`, *Group*)
- des autres (c'est à dire des utilisateurs qui ne sont ni propriétaire, ni membre du groupe propriétaire du fichier : (`o`, *Other*)

Remarque : ces 3 "ensembles" d'utilisateur(s) sont disjoints.

Permissions

Représentation symbolique des permissions

On représente les permissions associées à un fichier par trois triplets de lettres ou un tiret (-) pour l'absence de permission.



Pour connaître les permissions d'un utilisateur quelconque sur un fichier, il faut commencer par déterminer à quel "ensemble" (u, g ou o) il appartient pour ce fichier.
Rappel : un utilisateur peut appartenir à plusieurs groupes.

Permissions

Exécution d'un fichier exécutable par un utilisateur

- Remarque : pour être autorisé à lancer l'exécution du fichier exécutable, il faut que l'utilisateur ait le droit d'exécuter le fichier.
- Si l'utilisateur possède ce droit, un nouveau processus est créé pour exécuter le fichier, et il a par défaut les mêmes droits que l'utilisateur qui le lance. Plus précisément, le processus a pour UID (effectif) l'UID de l'utilisateur qui le lance, et il appartient aux mêmes groupes que cet utilisateur.
- Les permissions spéciales `SUID` et `SGID` permettent de modifier les permissions du processus qui exécute le fichier exécutable.

Permissions spéciales

Les permissions spéciales

- SUID (`s` ou `S`, *Set User ID*) : `--s------` ou `--S------`
  Permission d'exécution spéciale : le processus qui exécute le fichier exécutable a pour UID (effectif) l'UID du propriétaire du fichier
  `s` remplace `x` et `S` remplace `-`.
- SGID (`s` ou `S`, *Set Group ID*) : `-----s---` ou `-----S---`
  Permission d'exécution spéciale : le processus qui exécute le fichier exécutable devient membre du groupe propriétaire du fichier
  `s` remplace `x` et `S` remplace `-`.
- *Sticky Bit* (`t` ou `T`) : `--------t` ou `--------T`
  Sur un répertoire, il interdit la suppression d'un fichier qu'il contient à tout autre utilisateur que le propriétaire du fichier et le propriétaire du répertoire
  `t` remplace `x` et `T` remplace `-`.
  Exemple classique : `/tmp`

Exécutables avec et sans permission spéciale SUID ⋆⋆

Exemples (commandes cp et passwd)

```
[eric@dir_300 coursn]$ ls -l /usr/bin/cp
-rwxr-xr-x. 1 root root 141392 10 nov.   2017 /usr/bin/cp
[eric@dir_300 coursn]$ cp sys.pdf /etc
cp: impossible de créer le fichier standard '/etc/sys.pdf': Permission
denied

[eric@dir_300 coursn]$ ls -l /etc/shadow /usr/bin/passwd
-rw-r----- 1 root shadow  1490 nov.   3 22:55 /etc/shadow
-rwsr-xr-x 1 root root   59976 nov.  24 13:05 /usr/bin/passwd
[eric@dir_300 coursn]$ cat /etc/group | grep shadow
shadow:x:42:
[eric@dir_300 coursn]$
```

→ Les permissions spéciales SUID et SGID sont à utiliser avec beaucoup de précautions.

Permissions

Représentation octale

★★

Représentation octale

Un triplet de permission peut être représenté par un chiffre octal (0 à 7), chaque bit représentant une des permissions :

$$\texttt{r} = 4 = 100_2 \mid \texttt{w} = 2 = 010_2 \mid \texttt{x} = 1 = 001_2$$

0	`---`	pas de permission
1	`--x`	exécution
2	`-w-`	écriture
3	`-wx`	écriture et exécution
4	`r--`	lecture
5	`r-x`	lecture et exécution
6	`rw-`	lecture et écriture
7	`rwx`	lecture, écriture et exécution

Permissions

Représentation octale

★★

Représentation octale des permissions

On représente l'ensemble des permissions d'un fichier avec trois chiffres en octal (équivalents au trois triplets). Un quatrième chiffre octal peut être ajouté pour représenter les permissions spéciales.

Représentation octale des permissions spéciales

- SUID : 4000_8
- SGID : 2000_8
- Sticky Bit : 1000_8

Exemples

- `rw-r--r--` correspond à 644
- `rwsr-xr-x` correspond à 4755

Commandes relatives aux utilisateurs/groupes ⋆

Commandes d'identification

- `id` : *IDentify*
  Afficher les identifiants de l'utilisateur et de ses groupes

Commandes de modifications d'utilisateurs/groupes

- `chown [user] [file]` : *CHange OWNer*
  Change l'utilisateur propriétaire du fichier `file` pour qu'il appartienne désormais à l'utilisateur `user`. Seul `root` peut généralement appeler cette commande.
- `chgrp [group] [file]` : *CHange GRouP*
  Change le groupe propriétaire du fichier `file` pour qu'il appartienne désormais au groupe `group`. Le propriétaire d'un fichier peut changer le groupe d'un fichier uniquement vers un groupe auquel il appartient.

RTFM : `id(1)`, `chown(1)`, `chgrp(1)`

Commandes relatives aux permissions ⋆⋆

`umask`

Le *umask*

Le *umask* est une valeur sur quatre chiffres en octal qui est utilisée pour déterminer les permissions initiales d'un fichier ou d'un répertoire :

- Pour un fichier régulier : `permission` $= 0666_8$ & $\sim$*umask*
- Pour un répertoire : `permission` $= 0777_8$ & $\sim$*umask*

Généralement, le *umask* est à 0022_8, ce qui donne les permissions :
$\rightarrow 0644_8$ pour les fichiers et 0755_8 pour les répertoires.

Commande `umask`

- `umask` :
  Affiche le *umask* courant
- `umask [mask]` :
  Définit le *umask* courant à `mask`

Commandes relatives aux permissions ⋆⋆

`chmod`

Commande `chmod`

- `chmod [mode] [file]` : *CHange MODe*
  Définit les permissions du fichier `file` à `mode` (en notation octale)
- `chmod [symbol] [file]` : *CHange MODe*
  Change les permissions du fichier `file` selon la notation `symbol`

Notation symbolique

La notation symbolique comprend 3 champs :

- La ou les parties concernées : une ou plusieurs lettres parmi `ugoa`
- L'action : + pour activer, - pour désactiver, = pour définir
- La ou les permissions : une ou plusieurs lettres parmi `rwxXst`

RTFM : `chmod(1)`

Questions de la fin

À propos des inodes de fichier

Admettons que chaque bloc mémoire fasse 2048 octets et que chaque adresse de bloc soit codée sur 4 octets. Quelle est la taille maximale d'un fichier ?

Troisième partie

Shell et scripts shell

Plan de ce cours

5. Commandes
   - Code de retour
   - Enchaînement de commandes
   - Flux standard
   - Variables
   - Commandes internes/externes
   - Exécution/résolution d'une commande
   - Groupement de commandes
   - Développement de la ligne de commande
   - Chaînes de caractères
   - Dév ligne de commande, chaînes de caractères
6. Programmation Shell
   - Principes
   - Tests et structures de contrôle
   - Fonctions

Plan

Code de retour

Signification

★★★

Code de retour

Chaque programme renvoie un **code de retour** qui indique sa réussite ou son échec :

- 0 en cas de bon fonctionnement : le programme a réussi (associé à TRUE).
- $\neq 0$ en cas d'erreur(s) : le programme a echoué (associé à FALSE).

En cas d'erreur(s), le code de retour renvoyé dépend du programme.

$\rightarrow$ il est > 0 et toujours ≤ 255.

On peut récupérer le code de retour de la dernière commande exécutée en utilisant `$?`

Exemple

```
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$ echo $?
1
```

Plan

Enchaînement séquentiel simple ⋆⋆

Enchaînement séquentiel simple

Une ligne de commande de la forme :

$$\$\ \texttt{commande}_1\ ;\ \texttt{commande}_2\ ;\ \ldots\ ;\ \texttt{commande}_n$$

exécute en séquence, dans l'ordre et en mode synchrone les commandes spécifiées. La $\texttt{commande}_{i+1}$ n'est lancée que lorsque la $\texttt{commande}_i$ s'est terminée.

Exemples

```
$ echo -n "Bonjour, nous sommes le : " ; date +%d/%m/%y
Bonjour, nous sommes le : 25/01/24
$ echo $?
0
$ ls nonexistant
ls: impossible d'accéder à 'nonexistant': No such file or directory
$ echo $?
2
```

Enchaînement séquentiel conditionnel

★★

&& et || : opérateurs AND et OR

Enchaînement séquentiel conditionnel

```
$ commande1 && commande2
$ commande1 || commande2
```

- Avec &&, $\texttt{commande}_1$ est exécutée , et ensuite, quand l'exécution de $\texttt{commande}_1$ est terminée, $\texttt{commande}_2$ est exécutée si et seulement si $\texttt{commande}_1$ a réussi ($\Leftrightarrow$ le code de retour de $\texttt{commande}_1$ est égal à 0 (associé à TRUE)).
- || est similaire, mais $\texttt{commande}_2$ est exécutée si et seulement si $\texttt{commande}_1$ a échoué ($\Leftrightarrow$ le code de retour de $\texttt{commande}_1$ est différent de 0 (associé à FALSE)).

Pour un shell, les 2 opérateurs ont la même priorité.

Enchaînement séquentiel conditionnel

★★

&& et || : opérateurs AND et OR

La décision d'exécuter ou non une commande située après un opérateur && ou || dépend du code de retour de la **dernière commande exécutée**.

Exemples

```
$ test -d /etc && echo "C'est un répertoire"
C'est un répertoire
$ test -d /etc/passwd && echo "C'est un répertoire"
$
$ test -f /etc || echo "Ce n'est pas un fichier"
Ce n'est pas un fichier
$ test -f /etc && echo 'Oui' || echo 'Non'
Non
$ test -f /etc/passwd && echo 'Oui' || echo 'Non'
Oui
```

Enchaînement séquentiel conditionnel

★★

&& et || : opérateurs AND et OR

$? permet d'accéder au code de retour de la dernière commande exécutée

Exemples

```
$ touch fic1 fic2 fic3 fic4
$ ls fic1 && ls fic2 || ls fic3 && ls fic44
fic1
fic2
ls:impossible d'accéder à 'fic44':No such file or directory
$ echo $?
2
$ (ls fic1 && ls fic2) || (ls fic3 && ls fic44)
fic1
fic2
$ echo $?
0
```

Plan

Flux standard

Notion de flux

★★★

Définition (Flux)

Un **flux** est une séquence d'octets. Généralement, un flux est sous forme textuelle ($\neq$ binaire). On peut voir un flux comme un «tuyau de données» reliant un **processus** à un **fichier**.

Autre vision d'un programme

«Faites de chaque programme un filtre.» = un programme peut être vu comme un filtre qui manipule des flux.

Flux standard

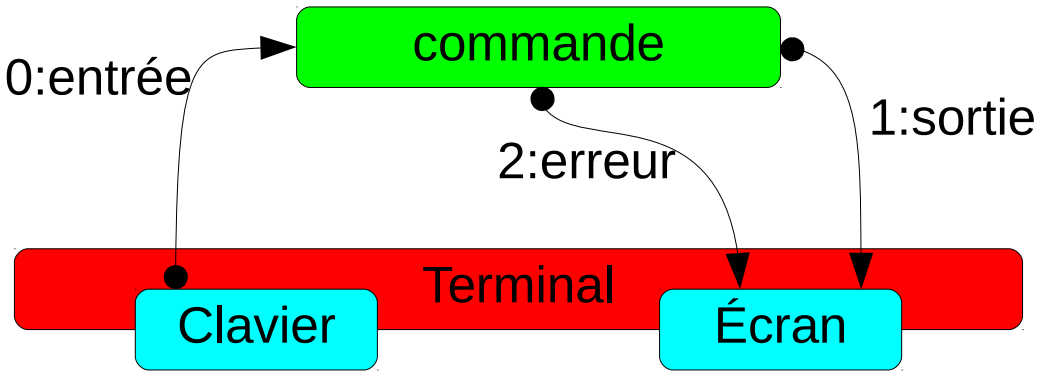
Flux standard

★★★

Flux standard

- Entrée standard (0, *standard input*, *stdin*) : flux d'entrée par lequel du texte ou toute autre donnée peut être entré dans un programme.
- Sortie standard (1, *standard output*, *stdout*) : flux de sortie dans lequel les données sont écrites par le programme.
- Erreur standard (2, *standard error*, *stderr*) : le flux de sortie permettant aux programmes d'émettre des messages d'erreur.

Par défaut, ces 3 flux sont connectés au terminal de lancement de la commande.



Redirection

Principes

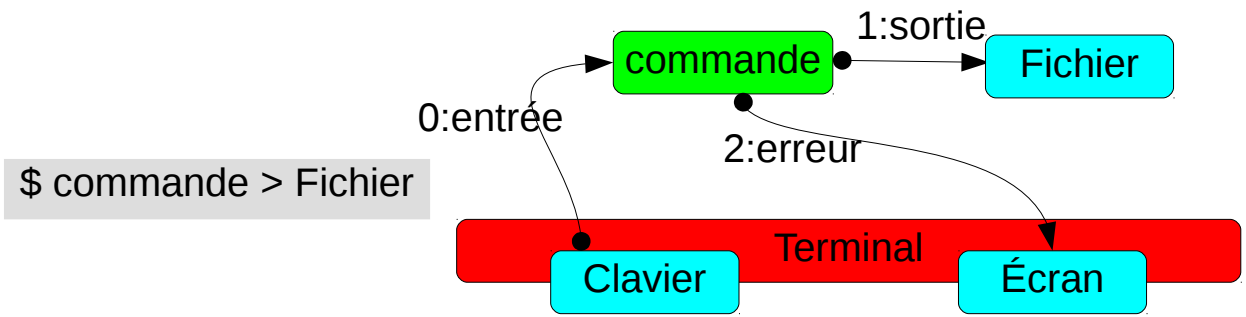
★★

Définition (Redirection)

Une **redirection** est un mécanisme qui permet de dérouter un flux, c'est à dire modifier le **fichier** auquel il est connecté.

Types de redirection

- `$ commande < fichier`
  Redirige l'entrée standard sur `fichier`. Les données seront lues depuis `fichier` plutôt que depuis le terminal (le clavier).
- `$ commande > fichier`
  Redirige la sortie standard sur `fichier`. Les données seront écrites dans `fichier` plutôt que sur le terminal.



Redirection

★★

Principes

Types de redirection

- `$ commande >> fichier`
  Redirige la sortie standard sur `fichier` sans écrasement (mode "append" : les données sont mises à la suite).
- `$ commande < fichier1 > fichier2`
  Les deux à la fois.
- `$ commande 2> fichier`
  Redirige l'erreur standard sur `fichier`. Les messages d'erreur seront écrits dans `fichier` plutôt que sur le terminal.
- `$ commande 2>> fichier`
  Redirige l'erreur standard dans `fichier` sans écrasement (mode "append").
- `$ commande 2>&1`
  Redirige l'erreur standard sur le fichier désigné par le descripteur 1 (&1) : la sortie standard.
- `$ commande 1>&2` ou `$ commande >&2`
  Redirige la sortie standard sur l'erreur standard.

Redirection

Exemples

Exemples

- `$ man man > man.txt`
  Redirige la page de manuel `man(1)` dans le fichier `man.txt`
- `$ cat fichier.txt > copie.txt`
  Mauvaise manière de copier un fichier.
- `$ cat fichier1.txt fichier2.txt fichier3.txt > tous.txt`
  Concatène les fichiers et met le résultat dans `tous.txt`
- `$ cat > fichier1.txt`
  Les caractères saisis au clavier sont écrits dans fichier1.txt
- `$ commande1 > tmp; commande2 < tmp; rm tmp`
  `commande2` utilise (en entrée) les résultats produits par `commande1`.
  Ce mécanisme nécessite la suppression du fichier temporaire créé.
  Il serait préférable d'utiliser un tube.

Tube

Principe d'un tube

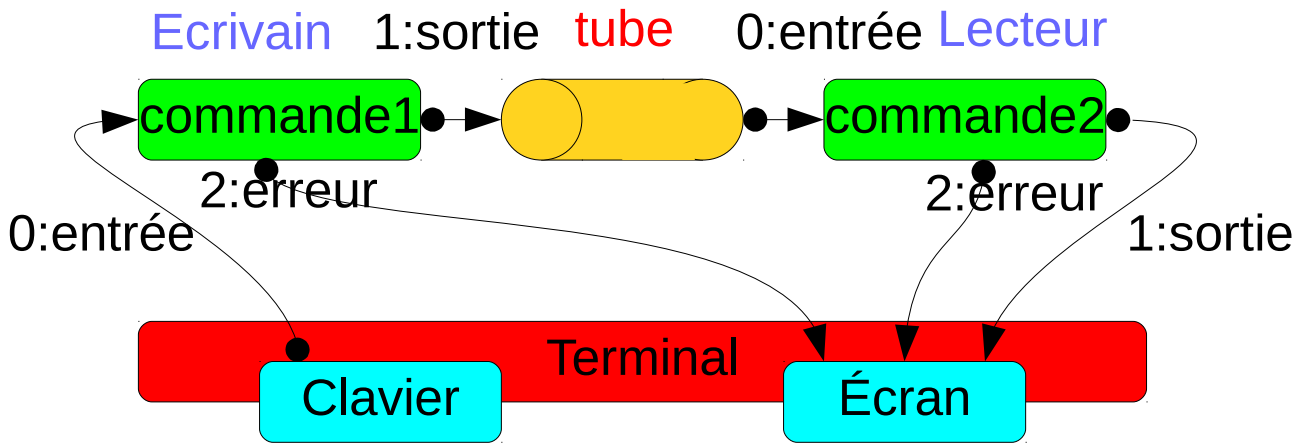
★★★

Définition (Tube)

Un **tube** (*pipeline* ou *pipe*) est un mécanisme qui permet de chaîner la sortie standard d'une commande à l'entrée standard d'une autre commande. On utilise le caractère | pour créer un tube.

```
$ commande1 | commande2
```

Un tube est un "tuyau" unidirectionnel.
La sortie standard de `commande1` est redirigée vers l'entreé du tube.
L'entrée standard de `commande2` est redirigée vers la sortie du tube.



Tube

★★★

Exemples et explications

Exemples

- `$ ps -elH | less`
  Affiche la liste de tous les processus page par page.
- `$ ls -1 | wc -l`
  `ls -1` écrit sur sa sortie standard les fichiers présents dans le répertoire courant (un par ligne).
  `wc -l` compte le nombre de lignes qu'elle reçoit sur son entrée standard.
  → compte et affiche le nombre de fichiers du répertoire courant.

Chaque commande précédente entraîne la création de 2 processus qui sont exécutées de façon **concurrente** (ils existent simultanément dans le système et sont en concurrence pour l'allocation du processeur). Le système assure la synchronisation :

- Il bloque le processus lecteur du tube lorsque le tube est vide ;
- Comme un tube a une capacité limitée, il bloque le processus ecrivain si le tube est plein ;
- Quand le processus qui exécute `wc -l` se termine-t-il ?
  Réponse : quand le processus détecte une fin de fichier sur son entrée standard, c'est à dire quand le tube est vide et que le processus écrivain est terminé

Interception de flux

La commande tee

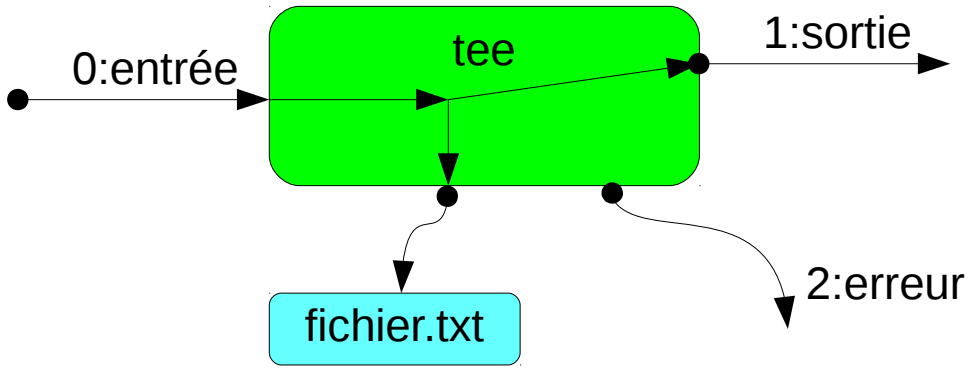
Interception de flux

Il est possible d'intercepter un flux et de le rediriger dans un fichier en insérant la commande `tee` entre deux commandes.

```
$ commande1 | tee fichier.txt | commande2
```

Exemple

```
$ echo "Vivent les tubes." | tee fichier.txt | wc
 1 3 18
$ cat fichier.txt
Vivent les tubes.
```



Redirections

★★★

Mise en place par le shell

Mise en place par le shell

Les redirections sont réalisées par le shell **avant** que les exécutables, correspondants aux commandes, soient chargés en mémoire.

Exemples

```
$ cat fic          $ wc fic          $ wc < fic
aa bb ccc          1 3 10 fic        1 3 10
```

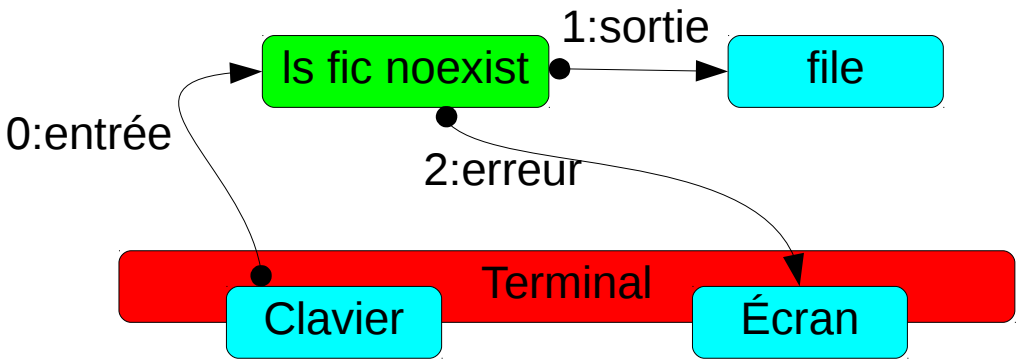
Avec `wc fic`, le programme exécuté (`/bin/wc`) reçoit `fic` en argument. Sa fonction main() reçoit `argc=2`, `argv[0]="wc"` et `argv[1]="fic"`. `argv[1]` est interprété par le programme comme un chemin vers un fichier. Il ouvre le fichier, le lit et écrit ses résultats sur sa sortie standard (qui indiquent bien qu'il connaît le fichier). Il n'utilise pas son entrée standard.
Avec `wc < fic`, la fonction main() du programme `/bin/wc` reçoit `argc=1` et `argv[0]="wc"` (les mots `"< fic"` sont analysés par le shell et ne lui sont pas transmis). Il lit donc ses données sur son entrée standard et n'a donc pas "conscience" de travailler sur un fichier disque : les résultats affichés ne font pas apparaître le nom du fichier.

Redirection

Exemple

Exemple (Redirection de la sortie standard)

```
$ ls
fic
$ ls fic noexist > file
ls: impossible d'accéder à 'noexist': No such file or directory
$ cat file
fic
$
```

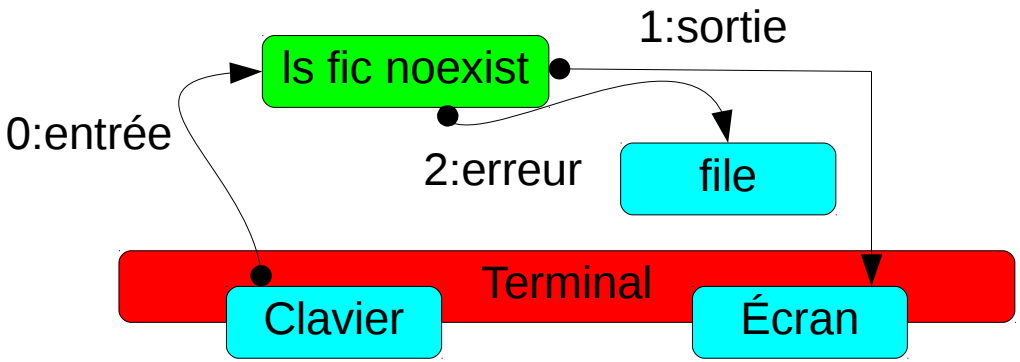


Redirection

Exemple

Exemple (Redirection de l'erreur standard)

```
$ ls
fic
$ ls fic noexist 2> file
fic
$ cat file
ls: impossible d'accéder à 'noexist': No such file or directory
$
```



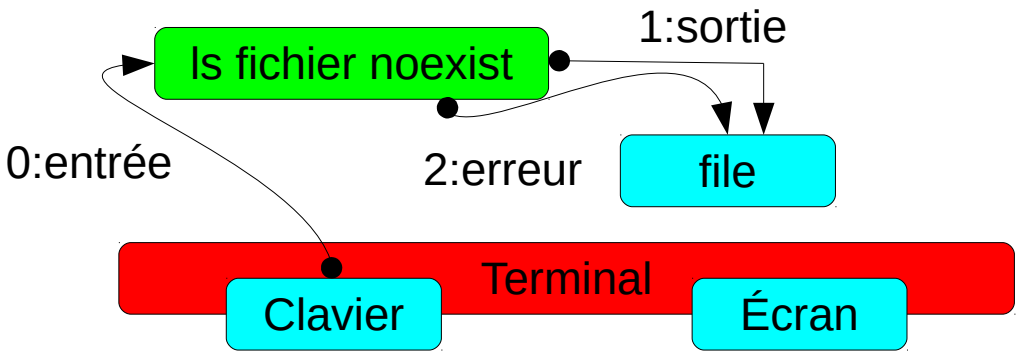
Redirections

Exemple

★★

Exemple (Redirection de la sortie standard et de l'erreur standard vers le même fichier (KO))

```
$ ls
fichier
$ ls fichier noexist > file 2> file
$ cat file
fichier
ssible d'accéder à 'noexist': No such file or directory
$
```



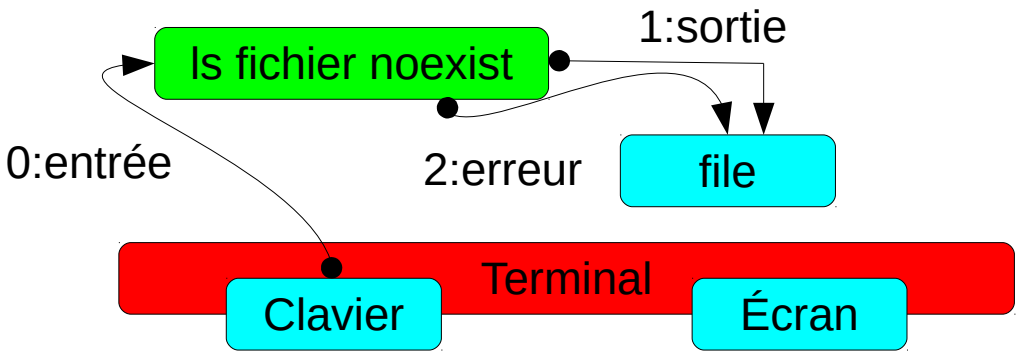
Redirections

Exemple

★★

Exemple (Redirection de la sortie standard et de l'erreur standard vers le même fichier (OK))

```
$ ls
fic
$ ls fic noexist > file 2>&1
$ cat file
ls: impossible d'accéder à 'noexist': No such file or directory
fic
$
```

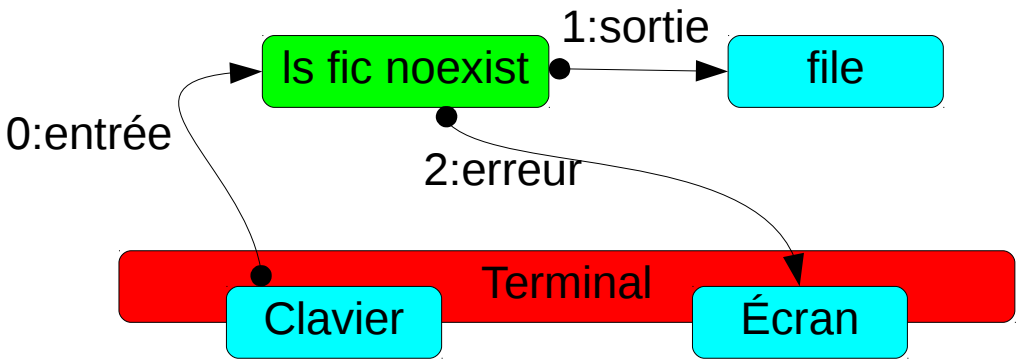


Redirections

Exemple

Exemple (Redirection de ...)

```
$ ls
fic
$ ls fic noexist 2>&1 > file
ls: impossible d'accéder à 'noexist': No such file or directory
$ cat file
fic
$
```



Redirections et tube

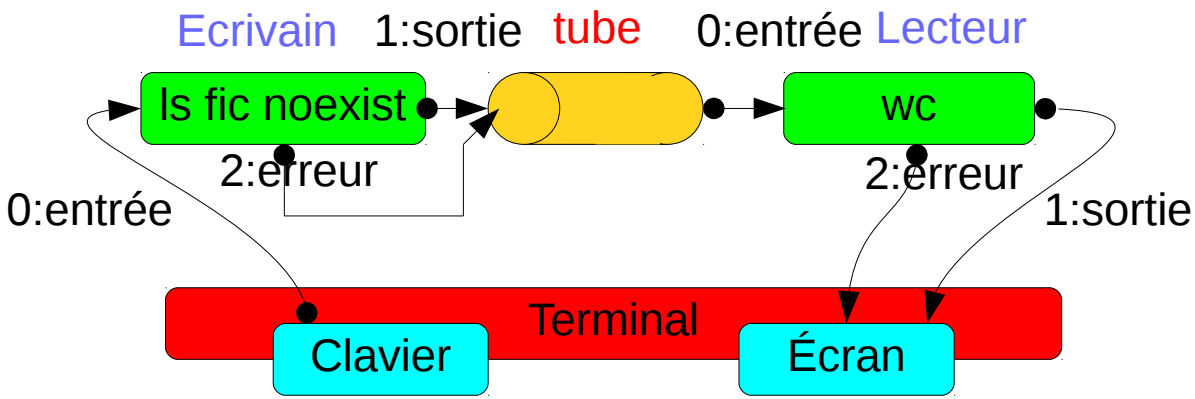
Exemple

★★

Exemple (Redirections et tube)

```
$ ls
fic
$ ls fic noexist  2>&1 | wc
      2       11       70
$
```

La redirection de la sortie standard de la commande `ls` sur l'entrée du tube a lieu avant la redirection de l'erreur standard sur la sortie standard. Donc l'erreur standard est redirigée elle aussi vers l'entrée du tube.



Plan

Variables

Affectation dans une variable

★★

Variable de shell

Le shell permet l'utilisation de variables. Une variable de shell est une variable non-typée, dont le nom contient uniquement des lettres (généralement en majuscule par convention), des chiffres et le caractère _. Le nom d'une variable ne peut pas commencer par un chiffre.
Exemple : `PATH`

Affectation dans une variable

L'affectation d'une valeur à une variable se fait de la manière suivante :
`$ VAR=`*value*

- Attention ! Pas de $ devant le nom de la variable
- Pas d'espace entre le nom de la variable et le =, ni après le =

Exemple

```
$ PI=3.1415
$ MY_NAME=Eric
```

Variables

Accès au contenu d'une variable

Accès au contenu d'une variable

On lit le contenu d'une variable à l'aide de $. On peut éventuellement délimiter le nom de la variable en utilisant des accolades ({ et }).

```
$ echo $VAR
$ echo ${VAR}
```

Exemple

```
$ echo PI
PI
$ echo $PO

$ echo $PI
3.1415
$ echo "${PI}957"
3.1415957
```

Environnement

★★★

Définition (Environnement)

Une variable d'un shell possède un attribut (booléen) qui peut la rendre *héritable* ou ***exportée*** :

- l'ensemble des variables exportées du shell constituent l'**environnement** du shell.
- une variable non exportée est une variable interne du shell.

Exemples

- La variable d'environnement `PATH` contient une liste de répertoires séparés par ":". Le shell recherche les exécutables des commandes externes dans ces répertoires.
  Exemple : `/usr/local/bin:/usr/bin:/bin:/usr/games`
- La variable d'environnement `PWD` contient le chemin absolu du répertoire courant.

Variables

Manipulation de variables

★★

Manipulation de variables

- `NOM=`*value* : affectation dans une variable
  S'il s'agit d'une création, la variable créée n'appartient pas à l'environnement.
- `set` : visualisation de toutes les variables définies (internes au shell et environnement)
- `env` : visualisation de toutes les variables d'environnement
- Exportation :
  - `export NOM`
    → Exportation d'une variable (déjà créée) vers l'environnement
  - `export NOM=`*value*
    → Création et exportation d'une variable vers l'environnement
- `unset NOM` : destruction d'une variable

RTFM : `env(1)`

Variables

Lecture sur l'entrée standard

Lecture sur l'entrée standard

La commande `read` permet de lire une ou plusieurs variables sur l'entrée standard.

```
$ read VAR
$ read VAR1 VAR2 VAR3
```

C'est la variable non exportée IFS (*Internal Field Separator*) qui désigne les caractères qui servent de séparateurs dans la ligne saisie

→ IFS contient par défaut la valeur ' \t\n'.

Exemple (Sans modif d'IFS)

```
$ read J M A
14;07;1789
$ echo "Date: $J/$M/$A"
Date: 14;07;1789//
```

Exemple (Avec modif d'IFS)

```
$ IFS=';'
$ read J M A
14;07;1789
$ echo "Date: $J/$M/$A"
Date: 14/07/1789
```

Plan

Commandes internes/externes

Commandes internes

★★

Définition (Commande interne)

Une **commande interne** est une commande dont le code est implémenté au sein du shell et qui est donc exécutée dans le processus qui exécute le shell : elle ne nécessite pas la création d'un nouveau processus pour son exécution.
Elle a donc un accès direct à l'ensemble des variables définies dans le shell.
L'environnement du shell peut être modifié.
Exemples : `echo`, `test`, `umask`, `...`

Commandes internes/externes

Commandes externes

★★

Définition (Commande externe)

Une **commande externe** est une commande fournie par un programme du système ⇒ le shell crée un nouveau processus (fils du shell) pour exécuter la commande.

- Le processus créé hérite d'une **copie** de l'environnement.
- ⇒ Il ne peut donc **pas modifier** l'environnement du shell.
- Il n'a pas accès aux variables non exportées (i.e. internes) du shell.

→ Il est nécessaire que la commande `cd` soit interne car elle doit être capable de modifier le répertoire de travail du shell (et donc la variable d'environnement `PWD`)
→ Certains shell possèdent une version "internalisée" de quelques commandes habituellement externes, et ce essentiellement pour des raisons de performance.

Plan

Exécution d'une commande ⋆⋆

Exécution d'une commande

Les alias, les fonctions et les commandes internes s'appellent directement par leur nom. Les commandes externes, quant à elles, peuvent s'appeler par leur nom ou en utilisant un chemin les ciblant :

1. `$ mon_alias arg1 arg2`
2. `$ ma_fonction arg1 arg2`
3. `$ commande_interne arg1 arg2`
4. `$ commande_externe arg1 arg2`
5. `$ /usr/bin/commande_externe arg1 arg2`
6. `$ ./commande_externe arg1 arg2`

Avec un chemin contenant par définition au moins un "/" (lignes 5 et 6 dans l'exemple ci-dessus), il n'y a pas d'ambiguïté : on sait de façon certaine que la commande appelée est une commande externe et on sait aussi où elle se trouve exactement dans le système de fichiers.

Résolution d'une commande ⋆⋆

Étapes de résolution d'un nom de commande

Quand la commande est simplement désignée par son nom avec
`$ ma_commande arg1 arg2`
le shell la recherche, dans l'ordre, en s'arrêtant dès qu'il la trouve, dans :

1. la liste des alias
2. la liste des fonctions shell
3. la liste des commandes internes
4. l'ensemble des commandes externes, c'est-à-dire **dans les répertoires du `PATH`**

Si au bout de ces 4 étapes le shell n'a toujours pas trouvé la commande, il affiche une erreur du type `"ma_commande: commande introuvable"`.

Commande interne ou commande externe

Les commandes type et which

★★

La commande type

- La commande interne type cherche à résoudre le nom de commande reçu en argument en utilisant le même algorithme que le shell (décrit sur la diapo précédente).
- Si le nom de commande correspond à une commande externe, la commande interne type affiche un chemin absolu vers l'exécutable.

La commande which(1)

La commande externe which(1) ne réalise que l'étape 4 de l'algorithme décrit sur la diapo précédente : elle recherche la commande reçue en argument uniquement dans les répertoires du PATH. Elle affiche un chemin absolu vers l'exécutable, si elle le trouve.

→ Si type indique qu'une commande est une commande interne, cela ne signifie pas qu'il n'en existe pas une version externe. Pour le savoir, il faut utiliser which(1)

Résolution d'une commande

★★

Exemples

```
$ type ls
ls is aliased to ‘ls --color=auto’
$ type echo
echo is a shell builtin
$ which echo          # pour savoir s'il existe aussi une version externe
/usr/bin/echo
$ type type
type is a shell builtin
$ which type          # pour savoir s'il existe aussi une version externe
$ type fdsdsfds
bash: type: fdsdsfds: not found
$ type mkdir
mkdir is /usr/bin/mkdir
$ echo bonjour            # version utilisée ?
bonjour
$ /usr/bin/echo bonjour   # version utilisée ?
bonjour
$ man 1 echo              # man page de la commande interne ou externe ?
```

Plan

Groupement de commandes ⋆⋆

Parenthésage des groupements de commandes

Il existe 2 types de parenthèses pour les groupements de commandes :

- Une séquence entre parenthèses () est exécutée dans un **sous-shell** (nouveau processus) qui dispose d'une **copie de toutes les variables du shell**.
- Une séquence entre accolades {} (où l'accolade ouvrante doit être suivie d'un espace et l'accolade fermante précédée d'un ;) est exécutée dans le shell courant.

Un groupement de commandes peut être vu comme une seule commande, notamment du point de vue des redirections.

Exemple

```
$ { echo -n 'Hello' ; echo ' World' ;} > message.txt
```

Groupement de commandes

★★

Exemple

```
$ pwd; MOT=bonjour
/home/eric
$ (echo $MOT; pwd; cd ..; MOT='AU REVOIR'; pwd; echo $MOT)
bonjour
/home/eric
/home
AU REVOIR
$ pwd; echo $MOT
/home/eric   → C'est le répertoire de travail du sous-shell qui a été modifié
bonjour      → C'est la copie de MOT dans le sous-shell qui a été modifiée
$ { echo $MOT; pwd; cd ..; MOT='AU REVOIR'; pwd; echo $MOT;}
bonjour
/home/eric
/home
AU REVOIR
$ pwd; echo $MOT
/home
AU REVOIR
$
```

Plan

Développement de la ligne de commande

word expansion

Développement de la ligne de commande

Le développement de la ligne de commande consiste en un ensemble d'opérations réalisées par le shell **avant** d'exécuter réellement la commande.

1.
   - Développement du tilde
     → ~ est remplacé par le nom du répertoire personnel de l'utilisateur connecté
   - Substitution des paramètres et des variables
     → les variables sont remplacées par leur contenu
   - Substitution de commandes
   - Développement arithmétique
2. Découpage en mots des champs générés lors de l'étape 1, suivant les caractères contenus dans `IFS`
3. Développement des chemins
4. Suppression des quotes

Développement de la ligne de commande

Substitution de commande

Substitution de commande

La substitution de commande permet de remplacer une commande par son résultat, c'est-à-dire par **ce qu'elle écrit sur sa sortie standard**. Il existe deux formes :

- `` `commande` `` $\rightarrow$ *il s'agit de backquotes*
- `$(commande)`

Exemple

```
$ echo "Nombre d'utilisateurs : $(cat /etc/passwd | wc -l)"
Nombre d'utilisateurs : 32
$ DIR=$(pwd)
```

Développement de la ligne de commande

Développement arithmétique

Développement arithmétique

Le développement arithmétique permet de remplacer une expression arithmétique par le résultat de son évaluation. Son format est : $((expression))

Exemple

```
$ A=3+4
$ echo $A
3+4
$ A=$((3+4))
$ echo $(($A-2))
5
```

Développement de la ligne de commande ⋆⋆

Développement des chemins

Développement des chemins

Le développement des chemins permet la substitution de certains motifs (*pattern*) par une liste de chemins.

Pour écrire un motif, on dispose de méta-caractères :

- `*` correspond à n'importe quelle chaîne de caractères (y compris vide)
- `?` correspond à n'importe quel caractère (*un seul*)
- `[aei]` correspond à *un seul* caractère qui peut être a, e ou i
- `[!aei]` correspond à *un seul* caractère qui ne peut pas être a, e ou i
- `[c-k]` correspond à *un seul* caractère qui peut être c, d, ..., k
- `[[:classe:]]` correspond à *un seul* caractère d'une classe parmi `alnum alpha ascii blank cntrl digit graph lower print punct space upper xdigit`
  Remarque : `[[:alnum:]]` est équivalent à `[a-zA-Z0-9]`

RTFM : `glob(7)`, `isalpha(3)`

Développement de la ligne de commande

Développement des chemins

Chaque motif est remplacé par l'ensemble des chemins correspondants qui désignent un fichier existant.

Exemples

- `*` est remplacé par tous les fichiers non cachés du répertoire courant
- `dir/.*` est remplacé par tous les fichiers cachés du répertoire `dir`
- `*.c` est remplacé par tous les fichiers non cachés, du répertoire courant, dont l'extension est `.c`
- `exemple?` est remplacé par tous les fichiers, du répertoire courant, dont le nom commence par `exemple` suivi de n'importe quel caractère
- `*.[Jj][Pp][Gg]` est remplacé par tous les fichiers non cachés JPEG, du répertoire courant, dont l'extension peut être écrite en majuscule ou en minuscule
- `/var/log/mysql.log.[[:digit:]].gz`

Plan

Chaînes de caractères ⋆⋆

Délimitation des chaînes de caractères

Les châines de caractères peuvent être délimitées par différents types de quotes :

- simple quote (') : tous les caractères à l'intérieur de la chaîne sont protégés : ils perdent, s'il en ont un, leur aspect spécial.
  → le seul caractère qui ne peut pas appartenir à une telle chaîne est le caractère simple quote (') lui même
- double quote (") : à l'intérieur de la chaîne, les caractères $, \, et ‘ sont encore des caractères spéciaux :
  → la chaîne peut contenir le caractère ", il suffit de le faire précéder d'un \
  → pas de découpage en mots (utilisant la valeur de `IFS`)
  → pas de développement des chemins
- backquote (‘) : la chaîne est interprétée comme une commande et le résultat de la commande lui est substitué.

Chaînes de caractères

★★

Longueur et extraction de sous-chaîne

Longueur

`${#VAR}` donne la longueur en nombre d'octets de la valeur de `VAR`

Extraction de sous-chaîne

Les motifs sont des "patterns shell" : ils utilisent les mêmes méta-caractères que le développement des chemins.

- `${VAR%motif}` renvoie une chaîne après avoir supprimé le plus **petit suffixe** correspondant avec `motif`
- `${VAR%%motif}` renvoie une chaîne après avoir supprimé le plus **grand suffixe** correspondant avec `motif`
- `${VAR#motif}` renvoie une chaîne après avoir supprimé le plus **petit préfixe** correspondant avec `motif`
- `${VAR##motif}` renvoie une chaîne après avoir supprimé le plus **grand préfixe** correspondant avec `motif`

Chaînes de caractères

★★

Longueur et extraction de sous-chaîne

Exemples

```
$ VAR='toto:titi:tata'; I='*:'
$ echo ${#VAR}
14          # la valeur de VAR ne contient pas les quotes
$ VAR2=${VAR#*:}
$ echo $VAR2
titi:tata
$ echo ${VAR##$I}
tata
$ echo ${VAR#a*:}
toto:titi:tata
$ echo ${VAR##*}

$ echo ${VAR#t*:}
titi:tata
```

Plan

Dév ligne de commande, chaînes de caractères (1/2) ⋆⋆

Exemples

```
$ touch fic1 fic2 fic3; ls
fic1  fic2  fic3

    # simple développement de chemins
$ ls fic[[:digit:]]
fic1  fic2  fic3

    # quand le développement de chemins échoue, le motif est transmis tel
    # quel à la commande
$ ls ficccc[[:digit:]]
ls: impossible d'accéder à 'ficccc[[:digit:]]': No such file or directory

    # suppression de quotes
$ ls 'fic1' \'fic1
ls: impossible d'accéder à "'fic1": No such file or directory
 fic1

$ A='fic1 fic2'; echo ${#A}
9   #la valeur de A ne contient pas les quotes
```

Dév ligne de commande, chaînes de caractères (2/2) ⋆⋆

Exemples (Suite diapo précédente)

```
    # substitution de variable + découpage en mots suivant IFS
$ ls $A
fic1  fic2

    # pas de substitution de variable entre simples quotes
    # suppression de quotes
$ ls '$A'
ls: impossible d'accéder à '$A': No such file or directory

    # substitution de variable, pas de découpage en mots suivant IFS
    # suppression de quotes
$ ls "$A"
ls: impossible d'accéder à 'fic1 fic2': No such file or directory

    # pas de développement de chemins entre double quotes
    # suppression de quotes
$ ls "fic[[:digit:]]"
ls: impossible d'accéder à 'fic[[:digit:]]': No such file or directory
```

Plan

Scripts shell

Définition

★★

Définition (Script shell)

Un **script shell** est un fichier texte contenant des **commandes** à exécuter. On lui donne généralement l'extension `.sh`. Un script shell commence obligatoirement par un **shebang** (`#!`) suivi du chemin du shell qui va interpréter les commandes du fichier : `#!/bin/sh`

Pour exécuter un script shell (après lui avoir attribué la permission d'exécution) :

`$ . ./script.sh` → Interprétation dans le shell courant

`$ ./script.sh` → Interprétation dans un nouveau shell : un shell **invoqué** ($\neq$ sous-shell car il hérite uniquement d'une copie de l'environnement)

Shell et script shell

Le shell implémente un langage de programmation assez complet qu'il est possible d'utiliser sur la ligne de commande ou dans un script. Tout ce qui est faisable dans un script est faisable sur la ligne de commande et vice-versa (à condition d'utiliser le même shell !).

Scripts shell

Premiers exemples

Exemple (Hello World en shell)

```
#!/bin/sh
echo 'Hello World'
```

Exemple (script1.sh)

```
#!/bin/dash
echo "VAR1=$VAR1"
echo "VAR2=$VAR2"
echo "PATH=$PATH"
```

Exemple (Exécutions de script1.sh)

```
$ export VAR1=toto
$ VAR2=tata
$ ./script1.sh
VAR1=toto
VAR2=
PATH=/usr/local/bin:/usr/local/sbin:/usr/bin
$ . ./script1.sh
VAR1=toto
VAR2=tata
PATH=/usr/local/bin:/usr/local/sbin:/usr/bin
```

Paramètres du script ⋆

Paramètres du script

Les paramètres du script sont dans les variables (paramètres positionnels) $0, $1, $2, ...En particulier la variable $0 contient le chemin utilisé pour lancer le script (ou le nom du shell courant s'il est interprété dans celui-ci). La commande `shift` permet de décaler tous les paramètres vers la gauche, l'ancien paramètre $1 est alors perdu ($0 ne change pas).

Exemple

```
$ cat script.sh
#!/bin/sh
echo "$0 $1"
$ ./script.sh toto
./script.sh toto
$ . ./script.sh tata
bash tata
```

Variables spéciales

Variables spéciales

- `$#` : nombre de paramètres positionnels
- `$*` ou `$@` : liste des paramètres positionnels
- `$?` : code de retour de la dernière commande exécutée
- `$$` : PID (Processus ID) du shell invoqué
- `$PPID` : PID du processus père du shell invoqué

Exemple

```
$ ps -H
  PID TTY          TIME CMD
 2489 pts/0    00:00:00 bash
 3292 pts/0    00:00:00   ps
$ echo "$$ $PPID"
2489 2483
```

Plan

Tests : commande test

★★

test ou [

- `-d FICHIER` : `FICHIER` existe et est un répertoire
- `-f FICHIER` : `FICHIER` existe et est un fichier ordinaire
- `-n CHAINE` : `CHAINE` est non-vide
- `CHAINE1 = CHAINE2` : les deux chaînes sont égales
- `ENTIER1 OP ENTIER2` : comparaison de `ENTIER1` avec `ENTIER2`
  → OP est dans la liste -eq, -ne, -lt, -le, -gt, ou -ge.
- `! EXPR` : `EXPR` est fausse
- `EXPR1 -a EXPR2` : `EXPR1` et `EXPR2` sont vraies (utilisez plutôt &&)
- `EXPR1 -o EXPR2` : `EXPR1` ou `EXPR2` sont vraies (utilisez plutôt ||)

Exemple

```
$ test -d "$1" && test -x "$1"
```

Teste si $1 est un répertoire ET si l'utilisateur courant y a accès

RTFM : `dash(1)`, `test(1)`

Commande test Piège à éviter ⋆⋆

Exemple

```
$ A=''
$ test -d $A && echo "'$A' est un répertoire"
'' est un répertoire
$ test -d "$A" && echo "'$A' est un répertoire"
$

# sans les double quotes, quand A est non définie, ou
quand c'est une chaîne vide, $A est remplacé par rien
du tout => la commande test -d ne reçoit pas d'opérande,
ce qui conduit à un résultat inattendu !
# avec les double quotes, quand A est non définie, ou
quand c'est une chaîne vide, la commande test -d reçoit
une chaîne vide en opérande, et la commande test -d
échoue : une chaîne vide n'est pas un chemin vers un
répertoire.
```

Structures de contrôle

Structure conditionnelle

if

```
if commande
then
  commandes_1
else      # optionnel
  commandes_2
fi
```

if

```
if commande; then
  commandes_1
else      # optionnel
  commandes_2
fi
```

Exemple

```
if test -d '/etc/passwd'
then
  echo '/etc/passwd is a directory'
else
  echo '/etc/passwd is not a directory'
fi
```

Structures de contrôle

else if vs elif

Exemple (else if)

```
if [ $# -eq 0 ]; then
  echo '0'
else
  if [ $# -eq 1 ]; then
    echo '1'
  else
    echo '>1'
  fi
fi #2ème fi nécessaire
```

Exemple (elif)

```
if [ $# -eq 0 ]; then
  echo '0'
elif [ $# -eq 1 ]; then
  echo '1'
else
  echo '>1'
fi # un seul fi
```

Conclusion

- Il faut fermer chaque `if` avec un `fi`
- Un `elif` doit "correspondre" à un `if`, et ne doit pas être fermé avec un `fi`

Structures de contrôle

Structure conditionnelle case

Exemple

```
case "$1" in
  'chien' | 'chat' )
    echo 'est un chien ou un chat'
    ;;
  [aeiouy]* )
    echo 'commence par une voyelle'
    ;;
  * )
    echo 'on est dans le cas par défaut'
    ;;
esac
```

Fonctionnement

- si la condition est vérifiée, tout ce qui suit est exécuté jusqu'au prochain double point-virgule, puis branchement après esac
- La commande case utilise les "patterns shell" (ceux utilisés par le mécanisme de développement de chemins)

Structures de contrôle

Structure répétitive `for`

★★

for

```
for VAR in list
do
  commandes
done
```

Exemple

```
for I in $(seq 0 9); do
  echo "${I} is a digit"
done
```

Structures de contrôle

Structure répétitive `while`

while

```
while commande; do
  commandes
done
```

Exemple

```
REP=""
while test -z "$REP"
do
  read REP
done
```

Structures de contrôle

Structure répétitive `until`

until

```
until commande
do
  commandes
done
```

Exemple

```
REP=""
until [ -n "$REP" ]; do
  read REP
done
```

→ Possiblité d'utiliser les commandes internes :

- `true` et `false` (leurs codes de retour sont respectivement égaux à 0 et à 1).
- `break [ num ]` et `continue [ num ]`

Plan

Fonctions

★★★

Transmission de paramètres

Fonctions et transmission de paramètres

Une fonction est définie dans un script shell de la manière suivante :

`nom () commande`

où *commande* est une commande ou un **groupement de commandes**.
Elle peut avoir des paramètres auxquels elle accède via les variables $1, $2, ...Une fonction s'appelle de la même manière qu'une commande normale (sans parenthèses !).
Au retour de l'appel, les valeurs initiales des paramètres positionnels sont restaurées.

Exemple

```
#!/bin/sh
hello () {
  echo "Hello $1";
}
hello 'World'
```

Fonctions

★★

Variables, return et exit

Fonctions et variables

Par défaut les variables d'un shell sont globales. Pour déclarer qu'une variable est locale à une fonction, il faut utiliser la commande interne `local`. Cette déclaration doit être placée au début de la fonction.

Fonctions et la commande interne `return`

- Utilisée dans une fonction, la commande interne `return` provoque la fin de la fonction. Le comportement est identique si on délimite la fonction avec des {} ou avec des ().
- On peut récupérer le code de retour renvoyé avec $?

Fonctions et la commande interne `exit`

- `exit` provoque la fin du shell (sous-shell ou shell invoqué) dans lequel cette commande est utilisée. Donc, si on exécute cette commande dans le shell invoqué, le script se termine.
- on peut récupérer le code de retour renvoyé avec $?

Fonctions

Variables

★★

Exemple (script2.sh)

```
#!/bin/dash
f () {
  local VAR1
  ps -H
  echo "Begin f:$VAR1,$VAR2"
  VAR1='V3'
  VAR2='V4'
  echo "End f:$VAR1,$VAR2"
}
VAR1='V1'
VAR2='V2'
f
echo "End main:$VAR1,$VAR2"
```

Pour la 2ème exécution, les {} ont été remplacées par des ()

Exemple (Exécutions du script)

```
$ ./script2.sh
  PID TTY          TIME CMD
12379 pts/0    00:00:00 bash
18162 pts/0    00:00:00   script2.sh
18163 pts/0    00:00:00     ps
Begin f:V1,V2
End f:V3,V4
End main:V1,V4
$ ./script2.sh
  PID TTY          TIME CMD
12379 pts/0    00:00:00 bash
18170 pts/0    00:00:00   script2.sh
18171 pts/0    00:00:00     script2.sh
18172 pts/0    00:00:00       ps
Begin f:V1,V2
End f:V3,V4
End main:V1,V2
```

Fonctions

Commandes internes `return` et `exit`

Exemple (`script3.sh`)

```
#!/bin/dash
func1 () (
  return 15
)
func2 () (
  exit 23
)
func1
echo "After 1, $?"
func2
echo "After 2, $?"
```

Exemple (Exécutions du script)

```
$ ./script3.sh
After 1, 15
After 2, 23
$ echo $?
0

## après remplacement ##
## des () par des {}  ##

$ ./script3.sh
After 1, 15
$ echo $?
23
```

Fonctions

Substitution de commande

★★

Substitution de commande

Comment récupérer au retour d'une fonction autre chose qu'un entier compris dans [0, 255] ?
Réponse : en utilisant le mécanisme de substitution de commande.

Exemple (script4.sh)

```
#!/bin/dash
func () {
  ps -H  >&2
  echo 'valret'
}
RET=$(func)
echo "RET=$RET"
```

Exemple (Exécution du script)

```
$ ./script4.sh
  PID TTY          TIME CMD
12379 pts/0    00:00:00 bash
17825 pts/0    00:00:00   script4.sh
17826 pts/0    00:00:00     script4.sh
17827 pts/0    00:00:00       ps
RET=valret
```

→ Que se passe-t-il si on remplace les {} par des () ?

Questions de la fin

À propos de la sortie standard d'erreur

Comment faire pour n'afficher aucune erreur quand on exécute une commande ?

À propos des commandes ?

Comment faire pour pouvoir exécuter n'importe quel programme du répertoire courant sans avoir à préfixer le nom du programme par `./` ?

Quatrième partie

Commandes (shell) de manipulation de fichiers texte

Plan de ce cours

7. Généralités
8. Les expressions rationnelles
9. Filtres
   - `grep(1)` : sélection de lignes
   - `cut(1)` : sélection de colonnes
   - `sed(1)` : filtrage et transformation de texte
   - `awk(1)` : manipulation des données contenues dans un fichier
   - `tr(1)` : transposition ou suppression de caractères
   - `sort(1)` : tri d'un fichier
   - `uniq(1)` : élimination des lignes répétées
   - `head(1)`, `tail(1)` : affichage des début/fin d'un fichier
   - `more(1)`, `less(1)` : affichage page par page
   - `wc(1)` : comptage du nombre de lignes/mots/octets

Fichier texte et filtres ⋆⋆

Fichier texte

Le fichier texte peu structuré est le format d'échange préféré quand on utilise les utilitaires de manipulation de fichiers texte. Peu structuré signifie que les informations sont rangées par ligne, éventuellement avec un caractère séparateur pour les différents champs sur chaque ligne.

Commandes filtres

- Toutes les commandes qui suivent (sauf la commande `tr`) peuvent prendre en argument un (ou plusieurs) fichier(s) texte. Si ce fichier est omis, alors c'est l'entrée standard qui est lue (qui peut donc être connectée à la sortie d'un tube).
- Toutes les commandes qui suivent ont une fonction simple et identifiée. Pour réaliser des opérations plus complexes, on associera plusieurs de ces commandes grâce à des tubes.

Les expressions rationnelles ⋆⋆

Les expressions rationnelles

Une expression rationnelle (regular expression) est un motif (`pattern`) qui permet de décrire un ensemble de chaînes lui correspondant. Dans les systèmes Unix, il existe deux versions différentes pour la syntaxe des expressions rationnelles :

- les expressions rationnelles ***simples (basic)*** : utilisées par défaut par les commandes `grep`, `sed`, ...
- les expressions rationnelles ***étendues (extended)*** : utilisées par les commandes `grep` avec l'option `-E`, `sed` avec l'option `-E`, `awk`, `less`, ...

→ Les expressions rationnelles étendues utilisent un certain nombre de méta-caractères : `.` (point), `*`, `+`, `?`, `[`, `]`, `{`, `}`, `(`, `)`, `\`, `^`, `-`, `$`
→ les expressions rationnelles et le développement de chemins n'utilisent pas les mêmes méta-caractères, voire ne leur donnent pas le *même sens* (exemples : `*`, `?`)
→ Dans la suite, on va traiter principalement les expressions rationnelles étendues.

Les expressions rationnelles

Expression atomique (1/2)

★★

Définition (Expression atomique)

Une expression atomique est une expression qui représente **un seul caractère**.

Ecriture d'une expression atomique (simple ou étendue)

Pour cela, on peut utiliser :

- un caractère ordinaire : représente lui-même
- `.` : méta-caractère qui représente n'importe quel caractère
- `\.` : représente le caractère `.`
- `\\` : représente le caractère `\`
- `[aei]` : représente `a`, `e` ou `i`
- `[^aei]` : représente un caractère qui ne peut pas être `a`, `e`, ou `i`

Les expressions rationnelles

Expression atomique (2/2)

★★

Ecriture d'une expression atomique (simple ou étendue)

Pour écrire une expression atomique, on peut aussi utiliser :

- `[b-k]` : représente un caractère $\geq b$ et $\leq k$
- `[^b-k]` : représente un caractère qui n'appartient pas à `[b-k]`
- `[[:classe:]]` : représente *un seul* caractère d'une classe parmi `alnum alpha blank cntrl digit graph lower print punct space upper xdigit`

RTFM : `isalpha(3)`

Les expressions rationnelles (simples ou étendues) ⋆

Exemples

Exemples (On cherche l'ensemble des correspondances avec le motif)

`bonjour`	une seule correspondance : la suite de caractères "bonjour"
`[[:digit:]]`	un caractère qui est un chiffre
`[0-9]`	un caractère qui est un chiffre
`[^[:digit:]]`	un caractère qui n'est pas un chiffre
`[[:alnum:]]`	un caractère alphanumérique
`[a-zA-Z0-9]`	un caractère alphanumérique
`[]-]`	un caractère qui doit être] ou -
`[]^-]`	un caractère qui doit être], ^ ou -
`\[]`	une seule correspondance : un [suivi par un un]
`[0-9]]`	un chiffre suivi par un un]

Dans une liste délimitée entre crochets, pour insérer :

- un caractère] littéral, il faut le placer en première position dans la liste
- un caractère ^ littéral, il faut le placer partout sauf en première position
- un caractère - littéral, il faut le placer en première ou dernière position

Les expressions rationnelles

★★

Ancres, sous-motif et opérateur |

Ancres

- ^ : marque le début d'une ligne lorsqu'il est placé en début d'une expression
- $: marque la fin de ligne lorsqu'il est placé en fin d'expression

Elles sont utilisables avec les expressions rationnelles simples et étendues. Elles ne "consomment" pas de caractère.

Sous-motif

Les parenthèses (et) permettent de répérer un sous-motif. Cette syntaxe ne fonctionne qu'avec les expressions rationnelles étendues.

Opérateur |

L'opérateur | permet d'indiquer un choix entre motifs ou sous-motifs. Il n'est utilisable qu'avec les expressions rationnelles étendues.

Les expressions rationnelles

Itérations

★★

Itérations (avec les expressions rationnelles étendues)

Nombre d'itérations (de l'expression atomique ou du **sous-motif** qui précède) :

- `?` : 0 ou 1
- `*` : 0 ou plus
- `+` : 1 ou plus
- `{n}` : n fois
- `{n,m}` : entre n et m fois
- `{n,}` : un nombre de fois $\geq n$

Les expressions rationnelles étendues

Exemples

Exemples (On cherche l'ensemble des correspondances avec le motif)

`un|une`	les mots "un" et "une"
`une?`	les mots "un" et "une"
`^[a-zA-Z]{4}[0-9]{5,}$`	une ligne constituée de 4 lettres et d'un nombre de chiffres ≥ 5
`^A+B`	un mot situé en début de ligne commençant par un ou plusieurs `A` suivi par un `B`
`^A*`	correspondance qui peut être de taille nulle : tout début de ligne correspond avec ce motif
`^A\*`	une seule correspondance : un caractère `A` suivi par une `*`, situés en début de ligne
`^$`	une seule correspondance : une ligne vide
`#.*`	un # et tous les caractères qui suivent

Les expressions rationnelles étendues ⋆

Exemples (On cherche l'ensemble des correspondances avec le motif)

`^(From|Subject):`	`"From:"` ou `"Subject:"` situés en début de ligne

`an? (simple |easy )?problem`	`"a simple problem"`
	`"an simple problem"`
	`"a easy problem"`
	`"an easy problem"`
	`"a problem"`
	`"an problem"`

`(^| )le([^a-z]|$)`	- une ligne contenant uniquement `"le"` - un `"le"` situé en début de ligne suivi par un caractère qui n'est pas une minuscule - un `"le"` précédé d'un espace, suivi par un caractère qui n'est pas une minuscule - un `"le"` précédé d'un espace, situé en fin de ligne
`([[:digit:]][[:lower:]]{2}){3}`	3 répétitions d'un sous-motif délimité par () et constitué d'un chiffre suivi de 2 lettres minuscules

Plan

Sélection de lignes

grep(1)

★★

grep(1)

`grep [OPTION]... PATTERN [FILE]... : Global Regular Expression Print`

Par défaut, écrit sur sa sortie standard les lignes contenant une correspondance avec un motif (`pattern`)

- `-E` : interpréter le `pattern` comme une expression rationnelle étendue
- `-G` : interpréter le `pattern` comme une expression rationnelle simple (comportement par défaut)

Exemples

- `$ grep -E 'bon' fic.txt`
- `$ grep -E '(^| )[Ll]e([^a-z]|$)' fic.txt`
- `$ grep -E '(^| )[Ll]e([^a-z]|$)' < fic.txt`
- `$ grep -E '([[:digit:]][[:lower:]]{2}){3}' fic.txt`

Plan

Sélection des colonnes

★★

cut(1)

Elle est utilisée avec des fichiers où chaque ligne contient des champs séparés par un *caractère séparateur*.

cut(1)

`cut OPTIONS... [FILE]...`

Sélectionner des colonnes d'un fichier (i.e. supprimer une partie de chaque ligne)

- `-d SEP` : utiliser `SEP` en tant que séparateur
- `-f LIST` : sélectionner les champs de `LIST`
  - `N` : N_{ieme} champs
  - `N,M` : N_{ieme} et M_{ieme} champs
  - `N-` : du N_{ieme} champs jusqu'à la fin de la ligne
  - `N-M` : du N_{ieme} champs jusqu'au M_{ieme} champs

Exemple

- `$ cut -d: -f7 /etc/passwd`
- `$ cut -d: -f7 < /etc/passwd`

Plan

Edition de flux

sed(1)

★★

sed(1) : Stream EDitor

`sed [OPTION]... {script-only-if-no-other-script} [input-file]...`
sed est un "éditeur de flux" *non intéractif*. Le flux en entrée (fichier(s) ou entrée standard) est traité *ligne par ligne*. sed lui applique des *commmandes* et écrit le résultat du traitement sur sa sortie standard.

- `-E` : utilisation des expressions rationnelles étendues
- `-n` : suppression de l'affichage
- `-i[SUFFIX]` : modifie directement le fichier, fait un backup si un `SUFFIX` est fourni

Quelques commandes

- `s/pattern/replacement/[indicator]` : remplace des occurrences correspondant au `pattern` par `replacement`. Sans indicateur, seule la première occurrence du `pattern` sur chaque ligne est remplacée. L'indicateur `g` permet de remplacer, sur chaque ligne, toutes les occurrences correspondant au `pattern`.
- `d` : supprime la ligne
- `p` : affiche la ligne (utilisée en général avec l'option `-n`)

Edition de flux

sed(1)

★★

Exemples

```
$ sed '4,6 d' test.txt
```

La sortie contient toutes les lignes sauf les 4ème, 5ème et 6ème lignes

```
$ sed -i '4 d;7 d' test.txt
```

Suppression des lignes 4 et 7 du fichier. Pas de backup. Pas de sortie

```
$ sed -n '4 p' test.txt
```

La sortie produite contient uniquement la 4ème ligne

```
$ sed -n '4,6 p' < test.txt
```

La sortie produite contient uniquement les 4ème, 5ème et 6ème lignes

```
$ sed -E 's/connection/connexion/' mondevoir.txt
```

Remplace, sur la sortie produite, la 1ère occurence du mot connection par connexion dans toutes les lignes

```
$ sed -E '4,7 s/\t/    /g' < fichier
```

Remplace, sur la sortie produite, toutes les tabulations par 4 espaces pour les lignes de numéros compris entre 4 et 7

Plan

Pattern scanning and processing language ⋆⋆

awk(1)

awk(1)

```
awk [OPTION]... [ -- ] program-text [input-file]...
awk [OPTION]... -f program-file [ -- ] [input-file]...
```

awk voit le texte comme une ensemble d'enregistrements (**records**) constitués de champs (**fields**), et permet d'appliquer des **actions** aux enregistrements sélectionnés par un **pattern**.

- les enregistrements sont par défaut séparés par un \n, ou sinon par l'expression rationnelle RS (Record Separator).
- les champs :
  - Ils sont par défaut séparés par un espace, ou sinon par l'expression rationnelle FS (Field Separator).
  - On accède à chaque champ de l'enregistrement courant avec les variables $1, $2, ... $NF. $0 correspond à l'enregistrement complet. La variable NF contient le nombre de champs de l'enregistrement courant, la variable $NF correspond donc au dernier champ de l'enregistrement courant.
  - Si FS est égal à null, chaque enregistrement est divisé en un champ par caractère ($1 contient le premier caractère, $2 le second, etc.)

Structure d'un programme

awk(1)

★★

pattern {action}

Un programme awk est une suite de "commandes" de la forme :

```
pattern {action}
```

Le pattern permet de sélectionner sur quels enregistrements (lignes si RS = \n) l'action est appliquée.

- S'il n'y a pas d'action, les enregistrements sélectionnés sont affichés.
- S'il n'y a pas de pattern, tous les enregistrements "matchent", et le bloc action est donc appliqué à tous les enregistrements.

Les patterns optionnels BEGIN et END

```
BEGIN {action}
pattern {action}
END {action}
```

Ils permettent respectivement de définir des actions à effectuer *avant* le traitement du premier enregistrement, et *après* le traitement du dernier enregistrement.

Pattern scanning and processing language ⋆⋆

awk(1)

Exemples (Commandes équivalentes)

```
    # affiche la 1ère colonne du fichier
    # en prenant l'espace comme séparateur
    # (une action, pas de pattern)
$ awk '{print $1}' fic
$ cut -d' ' -f1 fic

    # affiche les colonnes 1 et 3 du fichier
    # en prenant l'espace comme séparateur
$ awk '{print $1, $3}' fic
$ cut -d' ' -f1,3 fic

    # le pattern peut être une expression rationnelle délimitée entre /
    # Affiche toutes les lignes qui commencent par "toto"
    # (un pattern, pas d'action)
$ awk '/^(to){2}/' fic
$ grep -E '^(to){2}' fic
```

Variables

awk(1)

★★

Built in variables

Le programme awk dispose de variables d'environnement internes :

- NF : le nombre de champs de l'enregistrement courant
- NR : le nombre total d'enregistrements lus jusqu'à présent
- FS : l'expression rationnelle utilisée pour séparer les champs en entrée ; peut également être fixée avec l'option -F. Par défaut un espace.
- RS : le séparateur d'enregistrements utilisé en entrée. Par défaut un \n.
- OFS : le séparateur de champs utilisé en sortie. Par défaut un espace.
- ORS : le séparateur d'enregistrements utilisé en sortie. Par défaut un \n.

Variables "développeur"

Vous pouvez aussi définir vos propres variables "développeur" dans les blocs d'actions.

L'option -v var=val

Elle permet d'assigner la valeur val à la variable nommée var avant le début de l'exécution du programme.

Une syntaxe proche du langage C

awk(1)

Opérateurs

- Arithmétiques : +, -, *, /, %
- D'affectations simple et étendues : =, +=, -=, *=, /=, %=

Structures de contrôle

- Alternatives : `if(...){...}[else{...}]`
- Boucles :
  - `while(...) {...}`
  - `do {...} while(...);`
  - `for(...;...;...) {...}`

Des fonctions prédéfinies

- Numériques : `cos(x)`, `rand()`, `sin(x)`, `sqrt(x)`, ...
- Sur les chaînes de caractères : `index(s,t)`, `length(s)`, `match(s,r)`, `substr(s,i[,n])`, ...
- `printf()` pour les écritures formatées

Pattern scanning and processing language ⋆⋆

awk(1)

Exemples (Concaténation de chaînes de caractères)

var3 = var2 var1 ;
var3 = var3 " -- " var1 ;

Exemples

```
    # affiche les 2ème, 3ème et 4ème lignes du fichier
    # (pas d'action pour la 1ère commande)
    # (pas de pattern pour la dernière commande)
$ awk 'NR >= 2 && NR <= 4' fic
$ awk 'NR >= 2 && NR <= 4 {print $0}' fic
$ awk '{if (NR >= 2 && NR <= 4) print $0}' fic

$ echo '5.17:15.68' | awk -F: -v var='3' '{var+=$1+$2; print var;}'

$ awk 'BEGIN {nb_mots = 0; sum = 0;}
{ nb_mots +=  NF;
  for(i=1; i <= NF; ++i) sum += length($i); }
END {OFS=" -- "; print NR, nb_mots, sum}' < fic
```

Plan

Transposer ou éliminer des caractères ★★

tr(1)

tr(1)

tr copie son entrée standard sur sa sortie standard en effectuant au moins une des manipulations suivantes :

- transposition de caractères
- -d --delete : suppression de caractères
- -s --squeeze-repeats : élimination des répétitions de caractères

Exemples

```
$ tr 'abc' 'ABC' < fichier.txt
```

les caractères a, b et c sont remplacés respectivement par les caractères A, B et C

```
$ tr 'a-z' 'A-Z' < fichier.txt
$ tr '[:lower:]' '[:upper:]' < fichier.txt
```

conversion (transposition) de minuscules en majuscules

```
$ tr -s '\n' < test.txt
```

convertir les séquences de sauts de lignes en un seul saut de ligne (supprime les lignes vides)

```
$ tr -d '[:digit:]' < test.txt
```

supprime les chiffres

Plan

Tri d'un fichier

sort(1)

★★

sort(1)

`sort [OPTION]... [FILE]...`

`sort` trie les lignes des fichiers indiqués. Si aucun fichier n'est fourni, ou si le nom '-' est mentionné, la lecture se fera depuis l'entrée standard.

`sort` utilise l'ordre lexicographique par défaut.

- `-n` : ordre numérique
- `-r` : ordre inverse

Exemple

```
$ sort -n notes.txt
```

Plan

Élimination des lignes répétées

★★

uniq(1)

uniq(1)

`uniq [OPTION]... [INPUT [OUTPUT]]`

Sans l'option `-u`, `uniq` ne conserve qu'un seul exemplaire des lignes dupliquées **qui se suivent**.

`uniq` réclame donc que le fichier d'entrée soit trié car il ne compare que les lignes consécutives.

- `-c` : préfixer les lignes par le nombre d'occurrences
- `-u` : n'afficher que les lignes uniques

S'utilise, dans 95% des cas, en sortie de `sort(1)`

Exemple

```
$ echo 'mauvais\nbon\nmauvais' | sort | uniq -c
   1 bon
   2 mauvais
```

Plan

Affichage des début/fin d'un fichier ⋆⋆

head(1) et tail(1)

Affichage du début d'un fichier

`head [OPTION]... [FILE]...`
Affiche par défaut les 10 premières lignes.

- `-n K` : affiche les `K` premières lignes

Exemple

```
$ head -n 20 fic
```

Affichage de la fin d'un fichier

`tail [OPTION]... [FILE]...`
Affiche par défaut les 10 dernières lignes.

- `-n K` : affiche les `K` dernières lignes ;
- `-n +K` : la sortie commence à la ligne `K` (affiche la ligne `K` et les suivantes)

Exemple

```
$ tail -n +3 < fic
```

Plan

Affichage page par page

more(1) or less(1)

more(1)

more [OPTION]... [FILE]...
Affiche le texte page par page.

less(1)

less [OPTION]... [FILE]...
Affiche le texte page par page mais mieux que more
Quelques commandes disponibles :

<SPACE>	page suivante
b	page précédente
<NL>	ligne suivante
y	ligne précédente
/<pattern>	recherche avant d'une correspondance avec <pattern>
?<pattern>	recherche arrière d'une correspondance avec <pattern>
n	recherche de la correspondance suivante dans un sens
N	recherche de la correspondance suivante dans l'autre sens

Plan

Afficher le nombre de lignes/mots/octets ⋆⋆

wc(1)

wc(1)

`wc [OPTION]... [FILE]...`

Par défaut, compte le nombre de lignes/mots/octets du ou des fichier(s)

- `-l, --lines` : affiche le nombre de lignes
- `-w, --words` : affiche le nombre de mots
- `-c, --bytes` : affiche le nombre d'octets

Autre option :

- `-m, --chars` : affiche le nombre de caractères

Exemple

```
$ wc < systeme.tex
142 287 3596
$ echo -n 'éric' | wc -cm
4   5
```

Cinquième partie

Développement en C

Plan de ce cours

10. Construction d'un programme
    - Unité de compilation
    - Production d'un exécutable
    - Préprocesseur
    - Edition de liens
    - Construction de bibliothèques
    - Options de gcc

11. Make
    - Principe d'un Makefile
    - Makefile avancé
    - Générateur de Makefile

Plan

Problématique de la production de programmes ⋆⋆

Situation actuelle

Jusqu'à présent, vos programmes sont constitués d'un seul fichier. Généralement, votre programme n'a pas besoin d'autres fonctions que celles qui sont définies dans votre fichier source (en dehors des fonctions de la bibliothèque standard).

Problématique

Vous allez avoir à produire des programmes de plus en plus complexes, utilisant de nombreuses fonctionnalités. Cela pose deux problèmes :

- Comment produire des programmes constitués de plusieurs fichiers ?
- Comment mutualiser des fonctions utilisées par plusieurs programmes ?

Unité de compilation

★★★

Qu'est-ce que c'est ?

Définition (Unité de compilation)

Une unité de compilation est un ensemble cohérent de fonctions dont la définition est dans un fichier source `.c` et la déclaration (quand elle est nécessaire) est dans un fichier d'en-tête `.h`.

À quoi ça sert ?

Au moment de la conception, un programme est découpé de manière logique en plusieurs morceaux. Chaque morceau est alors implémenté dans une unité de compilation. Un exécutable est alors construit à partir de plusieurs unités de compilation.

Recommandations

Une unité de compilation ne doit pas avoir plus de 20 fonctions. Il faut toujours chercher à avoir des unités de compilation simples de manière à pouvoir retrouver une fonction facilement parmi toutes les unités.

Unité de compilation

Fichier d'en-tête

★★

Définition (Fichier d'en-tête)

Un fichier d'en-tête (*header*) d'une unité de compilation est un fichier contenant :

- les déclarations des types associés aux fonctions de l'unité
- les déclarations (ou prototypes) des fonctions de l'unité
- éventuellement des macrodéfinitions de symboles
- éventuellement des déclarations de variables globales

Pourquoi ?

Dans un fichier source, quand une fonction est appelée, il faut que cet appel soit précédé, soit de sa définition (corps), soit de sa déclaration (prototype).
Un fichier d'en-tête sert donc à déclarer les fonctions pour d'autres unités de compilation. Il a vocation à être **inclus** avec une directive `#include`.

Unité de compilation

Remarques et exemple pratique

Remarques

Dans un même fichier source :

- Une fonction peut avoir plusieurs déclarations, à condition qu'elles soient identiques, mais une seule définition.
- Un type ne peut être déclaré qu'une seule fois.

Exemple (Jusqu'à présent)

```
#include <stdio.h>
void say_hello(char *who);
int main() {
  say_hello("world");
  return 0;
}
void say_hello(char *who) {
  printf("Hello %s!\n", who);
}
```

Unité de compilation

Exemple pratique avec une unité de compilation

★★

Exemple (hello.h)

```
#ifndef HELLO_H
#define HELLO_H

void say_hello(char *who);

#endif /* HELLO_H */
```

Exemple (hello.c)

```
#include "hello.h"
#include <stdio.h>

void say_hello(char *who) {
  printf("Hello %s!\n", who);
}
```

Exemple (main.c)

```
#include "hello.h"
int main() {
  say_hello("world");
  return 0;
}
```

Plan

Production d'un exécutable à partir de plusieurs fichiers ⋆⋆

Étapes de production

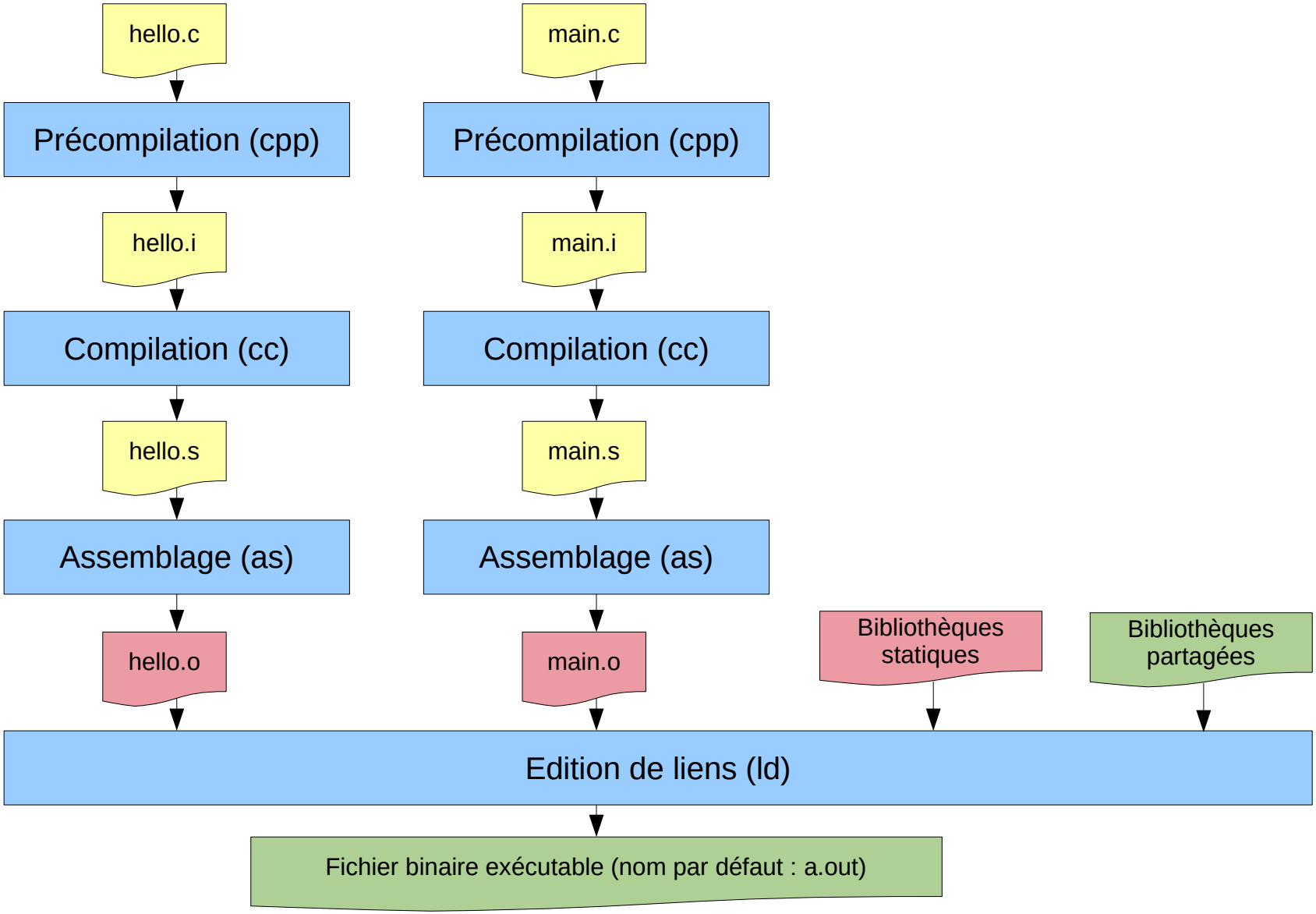
Étapes de production

1. Précompilation (`cpp(1)`) : fichier source `.c` → fichier source `.i`.
   Son rôle : traiter les **directives**.
2. Compilation (`cc(1)`) : fichier source `.i` → fichier assembleur `.s`.
3. Assemblage (`as(1)`) : fichier assembleur `.s` → fichier objet `.o`.
   Un fichier objet `.o` est un *fichier binaire* contenant du langage machine directement interprétable par le processeur.
4. Édition de liens (`ld(1)`) : liaison aux bibliothèques externes requises.

Pilote de compilation

gcc est un pilote de compilation (*compiler driver*), c'est-à-dire qu'il va appeler successivement tous ces programmes. Il est possible d'arrêter le processus après chacune des étapes via les options respectives : `-E`, `-S`, `-c`.

Production d'un exécutable : les 4 étapes ★★★



Production d'un exécutable : remarques ⋆⋆

Remarques

- Les fichiers d'entête `.h` ne doivent être soumis à aucune étape. Ils sont destinés à être inclus dans un ou plusieurs autre(s) fichier(s).
- Une bibliothèque (`.a` ou `.so`) est constituée d'un ensemble de *modules objets* (fichiers binaires) contenant les codes binaires des fonctions qu'elle définit. Il ne faut pas confondre une bibliothèque avec le ou les fichier(s) d'entête associé(s) à la bibliothèque. Un fichier d'entête d'une bibliothèque un fichier texte et constitue son *interface de programmation "utilisateur"*.
- Par exemple, les fichiers `<stdio.h>` et `<stdlib.h>` sont des fichiers d'entête de la bibliothèque standard du langage C appelée (`libc`). Cette bibliothèque est automatiquement liée à tous les programmes C.

Production d'un exécutable à partir de plusieurs fichiers ⋆⋆

Comment faire ?

Comment faire ?

Pour créer un exécutable à partir de plusieurs fichiers, il faut :

1. Compiler séparément les différents fichiers `.c` en fichiers objets `.o`.
2. Lier tous les fichiers `.o` ensemble pour créer l'exécutable

Exemple

Le programme `helloworld` est constitué de deux fichiers sources : `hello.c` et `main.c`.

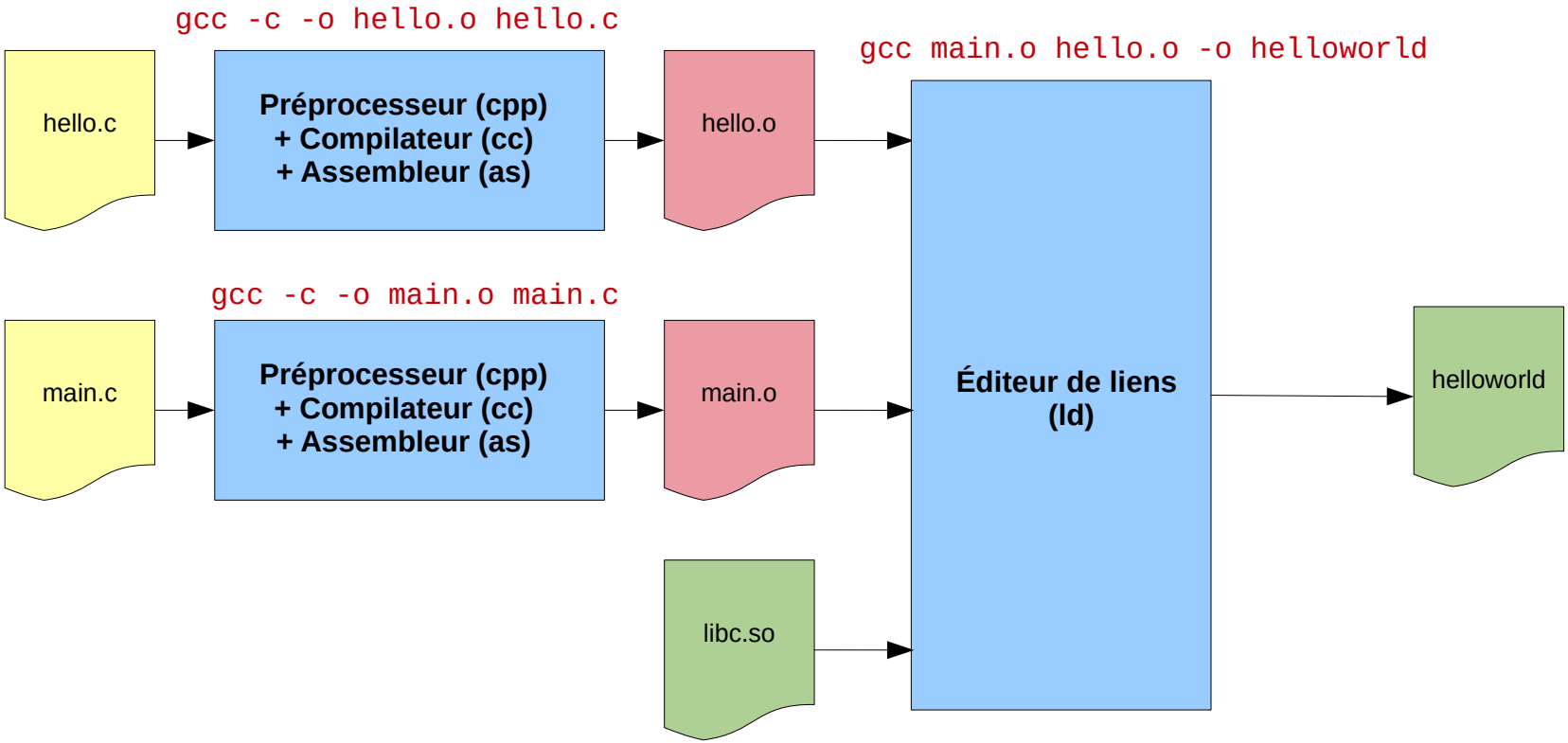
On le produit de la manière suivante :

```
$ gcc -c -o hello.o hello.c
$ gcc -c -o main.o main.c
$ gcc main.o hello.o -o helloworld
```

Production d'un exécutable à partir de plusieurs fichiers

★★★

Synthèse



Plan

Préprocesseur

Qu'est-ce que c'est ?

★★

Définition (Préprocesseur)

Le préprocesseur est un programme qui procède à des transformations sur un code source, avant l'étape de compilation. Il interprète des directives qui commencent par le caractère # et les remplace par du texte qui est ensuite envoyé au compilateur.

Remarques

- Il n'y a pas besoin d'appeler le préprocesseur directement, gcc s'en charge tout seul.
- Si on veut voir le fichier source à la sortie du préprocesseur, il est possible de passer l'option `-E` à gcc qui arrêtera le processus juste après l'étape de précompilation.

Préprocesseur

Macrodéfinition de symboles

★★

Définition de symboles avec `#define`

La directive `#define` permet de macrodéfinir des symboles qui seront remplacées par leurs valeurs dans le code source. Il s'agit d'une simple substitution de texte.

```
#define M_PI 3.14159265358979323846
#define NDEBUG
```

Exemple (Dans le code source)

```
double x = M_PI / 2.;
```

Exemple (Après le passage du préprocesseur)

```
double x = 3.14159265358979323846 / 2.;
```

Symboles prédéfinis particuliers

Le préprocesseur définit les symboles `__FILE__`, `__LINE__` qui représentent le fichier courant et la ligne courante dans le fichier.

Préprocesseur

Définition de macro

★★

Macros

La directive `#define` permet de définir une macro. Une macro ressemble à une fonction mais n'en est pas une !

```
#define OP(a,b) ((a) + 3 * (b))
```

Exemple (Dans le code source)

```
int x = OP(4,6);
```

Exemple (Après le passage du préprocesseur)

```
int x = ((4) + 3 * (6));
```

Préprocesseur

Les pièges classiques des macros

★★

Règles incontournables de définition d'une macro

- Une macro ne doit pas dépasser une ligne (sauf exception)
- Chaque paramètre est parenthésé dans la définition de la macro
- La définition de la macro doit elle-même être parenthésée

Exemple (Sans parenthèses autour des paramètres)

```
#define OP(a,b) (a + 3 * b)
int x = OP(4, 5 + 6);
int x = (4 + 3 * 5 + 6); /* COIN! */
```

Exemple (Sans parenthèses autour de la macro)

```
#define OP(a,b) (a) + 3 * (b)
int x = 2 * OP(4, 6);
int x = 2 * (4) + 3 * (6); /* COIN! */
```

Préprocesseur

Différences macros/fonctions

★★

Différences macros/fonctions

- Une fonction a un code binaire en mémoire, une macro n'en a pas, c'est une simple substitution de texte avec paramètres.
- Une fonction évalue une seule fois ses paramètres, une macro peut évaluer plusieurs fois ses paramètres.

Exemple (Évaluation multiple)

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
MAX(f(),g());
```

Deux évaluations de `f()` ou `g()`.

Préprocesseur

Compilation conditionnelle

★★

Compilation conditionnelle

Les directives `#if`, `#ifdef` et `#ifndef` permettent de compiler conditionnellement, c'est-à-dire de laisser après passage du préprocesseur uniquement le code qui remplit une certaine condition.

```
#if COND
#else
#endif
```

Exemples

```
#if LIB_VERSION >= 100
#if defined(DEBUG)
#ifdef DEBUG
#ifndef DEBUG
```

Préprocesseur

Exemple de définition simplifiée de la macro assert(3)

Exemple

```
#ifdef NDEBUG
#define assert(expr)
#else
#define assert(expr) if(!expr){fprintf(stderr, "Assertion
        failed : file %s, line %d \n", __FILE__, __LINE__);
        abort();}
#endif
```

Préprocesseur

Inclusion de fichier

★★

Inclusion de fichier avec `#include`

La directive `#include` permet d'inclure le contenu d'un fichier **d'en-tête** dans un autre fichier (source ou d'en-tête). Il y a deux types d'inclusion :

- `#include "fichier.h"` // pour les fichiers locaux
  Le fichier est recherché successivement dans :
  1. le répertoire courant
  2. puis dans la liste des répertoires indiqués avec l'option `-I` de la commande cpp
  3. et pour terminer dans le répertoire standard `/usr/include`
- `#include <fichier.h>` // pour les fichiers systeme
  Idem, sauf que le fichier n'est pas recherché dans le répertoire courant.

Exemples

```
#include "hello.h" // fichier d'en-tête "local"
#include <assert.h>  // fichier d'en-tête système
#include <stdio.h> // fichier d'en-tête système
```

Préprocesseur

include guard (1/2)

★★

Problèmes des inclusions

Dans un fichier source donné, le contenu d'un fichier d'entête ne doit être inclus qu'une seule fois.

→ Un fichier d'entête peut être inclus dans un autre fichier d'entête

Solution : *include guard*

Exemple (foo.h)

```
#ifndef FOO_H
#define FOO_H

#include "bar.h"

/* types and functions
 declarations */

#endif
```

Exemple (bar.h)

```
#ifndef BAR_H
#define BAR_H
/* types and functions
 declarations */
#endif
```

Exemple (foo.c)

```
#include "foo.h"
#include "bar.h"
/* functions definitions */
```

Préprocesseur

include guard (2/2)

★★

Problèmes des inclusions

Dans un fichier source donné, le contenu d'un fichier d'entête ne doit être inclus qu'une seule fois.

→ des boucles d'inclusion récursives (un fichier d'entête `A` qui inclut un fichier d'entête `B` qui inclut le fichier `A`) peuvent exister.

Solution : *include guard*

Exemple (foo.h)

```
#ifndef FOO_H
#define FOO_H

#include "bar.h"

/* types and functions
 declarations */

#endif
```

Exemple (bar.h)

```
#ifndef BAR_H
#define BAR_H

#include "foo.h"

/* types and functions
 declarations */

#endif
```

Préprocesseur

★★

Validité, constitution et inclusion d'un .h local

Validité d'un .h

Pour être valide, tous les types utilisés dans un .h doivent être "connus" (i.e. déclarés) avant leur utilisation, par exemple dans un prototype de fonction.

Constitution d'un .h

Attention cependant à inclure dans un .h le nombre **minimal** de fichiers d'entête requis pour déclarer les types qu'il utilise.

Inclusion d'un .h dans le .c du même nom

Pour vérifier qu'un .h est valide, il faut l'inclure **en premier** dans le .c correspondant (par exemple, foo.h doit être le premier fichier d'entête inclus dans foo.c).

Exemple

Pour pouvoir utiliser le type `size_t`, il suffit d'inclure le fichier d'entête système `stddef.h`.

Préprocesseur

Autres fonctionnalités

Suppression d'une macrodéfinition

La directive `#undef FOO` permet de supprimer une macrodéfinition à partir d'un certain point d'un fichier source.

Autres fonctionnalités

Il existe des fonctionnalités plus avancées, notamment pour les macros.
Mais attention ! Le préprocesseur est un outil à la fois puissant s'il est bien utilisé et dangereux s'il est mal utilisé. À utiliser de manière très raisonnée !

Plan

Fichier objet

★★

Définition, taille, table des symboles

Définition (Fichier objet)

Un fichier objet est un fichier contenant le code machine correspondant au fichier source ainsi que des informations sur les symboles (fonctions, variables globales) utilisés.
Un symbole utilisé dans un fichier objet (fichier binaire obtenu en sortie de `gcc -c`) peut être **défini** ou **non défini** dans ce fichier. Les symboles non définis sont dits **non résolus**.

Taille d'un fichier binaire

La commande `size(1)` fournit pour le ou les fichier(s) binaire(s) passé(s) en argument(s) la taille des différentes sections (ou segments) :
→ *segment de code* (le programme) : segment ***t**ext*
→ *segment des données* de classe statique *initialisées* : segment ***d**ata*
→ *segment des données* de classe statique *non initialisées* : segment ***b**ss*

Table des symboles d'un fichier binaire

La commande `nm(1)` permet de lister les symboles d'un fichier binaire. Pour chaque symbole, la commande fournit son nom, sa valeur (i.e. son adresse) et son type (**T** pour programme, **D** ou **B** pour données, **U** pour undefined, ...).

Taille, table des symboles

Exemples

★★

Exemples (Table des symboles de `hello.o` et `main.o`)

```
$ nm hello.o
                 U printf
0000000000000000 T say_hello
$ nm main.o
0000000000000000 T main
                 U say_hello
```

Exemples (Tailles de `hello.o`, `main.o` et `helloworld`)

```
$ size *.o helloworld
   text    data     bss     dec     hex filename
    104       0       0     104      68 hello.o
     83       0       0      83      53 main.o
   1230     532       4    1766     6e6 helloworld
```

Programme et bibliothèque ⋆⋆

Définition (Programme)

Un programme (ou application) est un exécutable qu'on peut appeler depuis la ligne de commande : il contient une fonction `main`.

Définition (Bibliothèque)

- Une bibliothèque est un fichier constitué d'un ensemble de *modules objets* (fichiers binaires) qui regroupe un ensemble de fonctions mais qui ne contient pas de fonction `main`.
- Intérêt : les fonctions d'une bibliothèque peuvent être utilisées par plusieurs programmes.
- Il existe deux types de bibliothèque : statique et dynamique. Le nom d'une bibliothèque commence toujours par `lib`.

Liaison

★★★

Définition (Liaison)

La liaison est un processus qui permet de résoudre les symboles, c'est-à-dire d'associer une adresse (ou une définition) aux symboles utilisés. Dans un exécutable, tous les symboles doivent donc être résolus. Plus précisément, chaque symbole doit avoir au plus une définition dans l'ensemble des modules objets à lier, ou à défaut au moins une définition dans l'une des bibliothèques. La liaison peut s'effectuer de deux manières :

- soit en trouvant le symbole dans un autre fichier objet, ou dans une bibliothèque statique `.a`, on parle alors de liaison statique
- soit en trouvant le symbole dans une bibliothèque dynamique `.so`, on parle alors de liaison dynamique

Liaison statique ⋆⋆

Liaison statique

- Ce type de liaison consiste à extraire le code binaire de la fonction et à le recopier dans le fichier binaire exécutable cible.
- Si tous les symboles d'un exécutable sont liés de façon statique, on obtient un exécutable *autonome* : le chargement du programme en mémoire, en vue de son exécution, pourra alors être réalisé directement, toutes les informations nécessaires à son exécution étant contenues (par construction) dans le fichier binaire.
- C'est la technique la plus simple offerte depuis les origines d'Unix.
- Inconvénients :
  - Avec cette technique, le code de la fonction `printf()`, par exemple, est chargée en mémoire centrale autant de fois qu'il y a de processus qui l'utilisent.
  - Les applications sont figées : si une bibliothèque est mise à jour, les applications déjà construites utilisent toujours l'ancienne version.

Liaison dynamique ⋆⋆

Liaison dynamique

- Lors d'une liaison dynamique, l'éditeur de liens ne résout plus totalement les symboles : dans la table des symboles, chaque *symbole "manquant"* est associé au nom de la bibliothèque où il a été trouvé.
- Au lancement du programme, la résolution est achevée : le chargeur de programme, appelé `ld.so(8)`, effectue les actions suivantes :
  1. il cherche et charge les bibliothèques dynamiques requises par le programme
  2. il résout les symboles contenus dans le programme grâce aux bibliothèques dynamiques
  3. il lance le programme
- Ce type de liaison évite les inconvénients de la liaison statique :
  → Il économise à la fois de l'espace disque et de l'espace mémoire (le cas échéant, le code de la fonction `printf()` n'est chargé qu'une seule fois, quelque soit le nombre de processus qui l'utilisent).
  → Les nouvelles versions des bibliothèques sont prises en compte automatiquement (i.e. sans nécessité de reconstruire l'exécutable).

Liaisons statique et dynamique

Exemples

Exemples (Table des symboles)

```
$ gcc main.o hello.o -o helloworld
$ gcc -static main.o hello.o -o helloworld-s
$ ls -l helloworld helloworld-s
-rwxrwxr-x. 1 eric eric   8592  5 févr. 16:02 helloworld
-rwxrwxr-x. 1 eric eric 913680  5 févr. 16:02 helloworld-s
$ nm helloworld | grep -E 'printf|say| main'
0000000000400527 T main
                 U printf@@GLIBC_2.2.5
000000000040053c T say_hello
$ nm helloworld-s | grep -E ' printf$|say| main$'
0000000000400add T main
000000000040f3e0 T printf
0000000000400af2 T say_hello
```

Plan

Illustration

Ajout des 2 fichiers suivants

Exemple (bye.h)

```
#ifndef BYE_H
#define BYE_H

void say_bye(char *who);

#endif /* BYE_H */
```

Exemple (bye.c)

```
#include "bye.h"
#include <stdio.h>

void say_bye(char *who) {
  printf("Bye %s!\n", who);
}
```

Exemples (hello.h, hello.c, main.c)

Les fichiers `hello.h`, `hello.c`, `main.c` ne sont pas modifiés.

Bibliothèque statique

★★

Définition (Bibliothèque statique)

Une bibliothèque statique est une simple archive composée de modules objets (`.o`). Quand une édition de liens est réalisée avec une bibliothèque statique, seuls les modules objets nécessaires en sont extraits (i.e. recopiés) pour former l'exécutable. Une bibliothèque statique a généralement l'extension `.a`.

Production d'une bibliothèque statique

Pour produire une bibliothèque statique, on fait appel à la commande `ar(1)` (*ARchive*) qui sert à créer des archives. Elle dispose de nombreuses options de manipulation des archives (très proches de celles de `tar(1)`). Les options utiles pour la création d'une bibliothèque statique sont `cr`. Lors de la construction d'une bibliothèque statique, aucune édition de liens n'est réalisée.

Exemple

```
$ ar cr libhello-static.a hello.o bye.o
```

RTFM : `ar(1)`

Bibliothèque dynamique

★★

Définition (Bibliothèque dynamique)

Une bibliothèque dynamique est une bibliothèque qui est liée dynamiquement avec les programmes qui l'utilisent (le code binaire de ses fonctions n'est pas copié dans l'exécutable). Une bibliothèque dynamique a généralement l'extension `.so` (*Shared Object*).

Production d'une bibliothèque dynamique

- Pour pouvoir construire une bibliothèque dynamique avec des modules objets, ils doivent être compilés avec l'option `-fPIC` (Position Independant Code).
- Pour produire une bibliothèque dynamique, on fait appel à la commande `gcc` (qui appelle `ld`) avec l'option `-shared`. Une édition de liens est donc réalisée : les symboles sont résolus.

Exemple

```
$ gcc -fPIC -c -o hello.o hello.c
$ gcc -fPIC -c -o bye.o bye.c
$ gcc -shared hello.o bye.o -o libhello.so
```

Construction de bibliothèques

Exemples

Exemples (Détails des fichiers et table des symboles)

```
$ ls -l lib*
-rwxrwxr-x. 1 eric eric 8176  5 févr. 17:44 libhello.so
-rw-rw-r--. 1 eric eric 3250  5 févr. 17:44 libhello-static.a
$ nm libhello-static.a

hello.o:
                 U printf
0000000000000000 T say_hello

bye.o:
                 U printf
0000000000000000 T say_bye
$ nm libhello.so | grep -E 'say| printf'
                 U printf@@GLIBC_2.2.5
00000000000006e7 T say_bye
00000000000006c0 T say_hello
```

Utilisation d'une bibliothèque ⋆⋆

Comment utiliser `libhello-static.a` ou `libhello.so` ?

Pour utiliser la bibliothèque (statique ou dynamique), il faut :

- inclure l'en-tête dans le fichier source de manière à pouvoir utiliser (appeler) les fonctions de la bibliothèque
  → l'option `-I` permet d'ajouter un répertoire non-standard dans la liste des répertoires de recherche des en-têtes
- au moment de l'édition de liens, lier l'exécutable à la bibliothèque avec l'option `-l` suivi du nom de la bibliothèque sans le préfixe `lib`
  → l'option `-L` permet d'ajouter un répertoire non-standard dans la liste des répertoires de recherche des bibliothèques

```
$ gcc -L${PWD} main.o -lhello-static -o helloworld-d1
$ gcc -L${PWD} main.o -lhello -o helloworld-d2
```

Tables des symboles des exécutables et exécutions ⋆

Exemples (Tables des symboles)

```
$ nm helloworld-d1 | grep -E 'printf|say| main'
00000000004004d7 T main
                 U printf@@GLIBC_2.2.5
00000000004004ec T say_hello


$ nm helloworld-d2 | grep -E 'printf|say| main'
00000000004005a7 T main
                 U say_hello
    # Le symbole say_hello semble non résolu alors qu'il l'est.
    # Sinon la compilation échouerait.
```

Exemples (Exécutions)

```
$ ./helloworld-d1
Hello world!
$ ./helloworld-d2
./helloworld-d2: error while loading shared libraries:
libhello.so: cannot open shared object file: No such file or directory
```

Liste des dépendances dynamiques d'un exécutable ⋆

La commande ldd

La commande `ldd(1)` (List Dynamic Dependencies) permet de connaître la liste des bibliothèques partagées nécessaires pour un programme ou une bibliothèque.

Exemples

```
$ ldd helloworld-d1
    linux-vdso.so.1 (0x00007ffd915d9000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f0624b30000)
    /lib64/ld-linux-x86-64.so.2 (0x000055e8c8434000)
$ ldd helloworld-d2
    linux-vdso.so.1 (0x00007fff2a865000)
    libhello.so => not found
    libc.so.6 => /lib64/libc.so.6 (0x00007fed73ab7000)
    /lib64/ld-linux-x86-64.so.2 (0x0000561a26cac000)
```

→ Il s'agit donc d'un problème de **localisation** de la bibliothèque dynamique, qui apparaît lorsque l'on souhaite **exécuter** le programme.

Une solution pour localiser la bibliothèque `libhello.so` ⋆

La variable d'environnement `LD_LIBRARY_PATH`

Elle contient une liste de répertoires séparés par le caractère : dans lesquels la recherche de bibliothèques dynamiques est effectuée en priorité.

Exemples

```
$ export LD_LIBRARY_PATH=$PWD
$ ldd helloworld_d2
    linux-vdso.so.1 (0x00007ffd6d14f000)
    libhello.so => /home/.../ex/libhello.so (0x00007f86350dc000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f8634cfd000)
    /lib64/ld-linux-x86-64.so.2 (0x000055da5deca000)
$ ./helloworld-d2
Hello world!
```

Plan

Options de gcc

Préprocesseur

Options transmises au préprocesseur cpp

Les options suivantes sont transmises au préprocesseur par `gcc` :

- `-DFOO[=3]` : le préprocesseur s'exécute comme si la directive `#define FOO [3]` apparaissait dans le fichier source.
  Exemple : `$ gcc -DNDEBUG -c -o fichier.o fichier.c`
- `-UFOO` : le préprocesseur s'exécute comme si la directive #undef FOO apparaissait dans le fichier source.
- `-I dir` : permet d'ajouter le répertoire non-standard `dir` dans la liste des répertoires de recherche des fichiers d'en-têtes.

Options de gcc

Editeur de liens

Options transmises à l'éditeur de liens `ld`

Les options suivantes sont transmises à l'éditeur de liens par `gcc` :

- `-lnom` : permet d'ajouter la bibliothèque de nom `libnom.ext` (ou `ext` peut prendre la valeur `a` ou `so`) en entrée de l'éditeur de lien. Il ne faut pas placer cette option n'importe où dans la ligne de commande (**elle doit en général être placée à la fin**)
- `-LDIR` : permet d'ajouter le répertoire `DIR` dans la liste des répertoires de recherche des bibliothèques.
- `-shared` : produire un objet partagé qui peut être lié avec d'autres objets pour former un exécutable.
- `-static` : produire un exécutable autonome.

Plan

Makefile

Définition (Makefile)

Un Makefile est un fichier qui décrit les différentes actions nécessaires à la production d'un logiciel à partir de données sources. La commande `make(1)` permet d'exécuter les actions du Makefile situé dans le répertoire courant. Un Makefile permet d'automatiser la production d'un logiciel, c'est-à-dire d'éviter de taper les commandes de production à chaque changement dans le code source : la commande `make(1)` se charge de tout !

Différence avec un script shell

La principale différence avec un script shell est l'optimalité de la production : on ne recompile que les fichiers qu'il est nécessaire de recompiler !

Makefile

Fonctionnement

★★★

Fonctionnement

Un Makefile est composé d'une suite de règles qui ont la forme suivante :

```
cible: dépendances
⟼actions
```

où les dépendances sont les fichiers nécessaires à la production de la cible et les actions sont les commandes nécessaires pour construire la cible à partir des dépendances. À l'appel de `make(1)` :

1. les dépendances sont analysées récursivement : pour une règle donnée, et avant toute décision, la commande `make` tente de (re)construire les dépendances si d'autres règles l'indiquent (chaque dépendance devenant à son tour cible).
2. si une dépendance est plus récente que la cible, ou si la cible n'existe pas, alors on exécute les actions de manière à produire la cible.

Makefile

Exemple

★★

Exemple (Makefile)

```
helloworld: hello.o main.o
	gcc -g hello.o main.o -o helloworld
hello.o: hello.c hello.h
	gcc -Wall -std=c99 -g -c -o hello.o hello.c
main.o: main.c hello.h
	gcc -Wall -std=c99 -g -c -o main.o main.c
```

Exécution des règles

`$ make`

Par défaut, exécute la première règle, donc `helloworld`

`$ make hello.o`

Exécute la règle `hello.o` → ne fait rien car "`hello.o`" est à jour

`$ touch hello.c; make`

Exécute la règle `helloworld` → reconstruit "`hello.o`" puis "`helloworld`"

Makefile

Bonnes pratiques

★★

Bonnes pratiques

- Il existe toujours une règle `all` : règle par défaut qui regroupe dans ses dépendances tous les exécutables à produire
- Il existe toujours une règle `clean` qui sert à nettoyer tous les fichiers "intermédiaires" générés au cours de la production, à l'exception du fichier final
- Il existe parfois une règle `mrproper` qui sert à nettoyer tous les fichiers générés au cours de la production, y compris le fichier final
- Il existe souvent une règle `install` qui permet d'installer le logiciel (programmes, bibliothèques, en-têtes) sur le système

Makefile

Exemple amélioré

Exemple (Makefile)

```
all: helloworld

helloworld: hello.o main.o
	gcc -g hello.o main.o -o helloworld
hello.o: hello.c hello.h
	gcc -Wall -std=c99 -g -c -o hello.o hello.c
main.o: main.c hello.h
	gcc -Wall -std=c99 -g -c -o main.o main.c
clean:
	rm -f *.o
mrproper: clean
	rm -f helloworld
```

Plan

Makefile

Variables

★★

Variables dans un Makefile

On définit des variables de la même manière que dans un script shell

Variables classiques

- `CC` : compilateur C (généralement `gcc`)
- `CFLAGS` : options de compilation (utilisée avec `gcc -c`)
- `LDFLAGS` : options de liaison sauf `-l` (utilisée avec `gcc` pour l'édition de lien)
- `LDLIBS` : bibliothèque(s) liée(s) (utilisée avec `gcc` pour l'édition de lien)
  Exemple de valeur : `-lm`

Options classiques

- `-g` : inclut les symboles de debug
- `-Wall` : active tous les warnings (obligatoire !)
- `-O3` : optimisation de niveau 3 (maximum)
- `-Os` : optimisation de la taille

Makefile

Exemple avec variables

Exemple (Makefile)

```
CC=gcc
CFLAGS=-Wall -std=c99 -g
LDFLAGS=-g
TARGET=helloworld

all: $(TARGET)

$(TARGET): hello.o main.o
	$(CC) $(LDFLAGS) hello.o main.o -o $(TARGET)
hello.o: hello.c hello.h
	$(CC) $(CFLAGS) -c -o hello.o hello.c
main.o: main.c hello.h
	$(CC) $(CFLAGS) -c -o main.o main.c
clean:
	rm -f *.o
mrproper: clean
	rm -f $(TARGET)
```

Makefile

Variables spéciales

★★

Variables spéciales

Elles sont utilisées dans les *actions*.
Une variable spéciale permet de remplacer un ou plusieurs éléments de la règle de manière générique :

- `$@` : nom de la cible
- `$<` : première dépendance
- `$^` : liste de toutes les dépendances
- `$?` : liste de toutes les dépendances plus récentes que la cible
- `$*` : nom de la cible sans extension

Makefile

Exemple avec variables spéciales

Exemple (Makefile)

```
CC=gcc
CFLAGS=-Wall -std=c99 -g
LDFLAGS=-g
TARGET=helloworld

all: $(TARGET)

$(TARGET): hello.o main.o
	$(CC) $(LDFLAGS) $^ -o $@
hello.o: hello.c hello.h
	$(CC) $(CFLAGS) -c -o $@ $<
main.o: main.c hello.h
	$(CC) $(CFLAGS) -c -o $@ $<
clean:
	rm -f *.o
mrproper: clean
	rm -f $(TARGET)
```

Règles d'inférence ⋆⋆

Définition (Règle d'inférence)

Une règle d'inférence est une règle générique de production qui permet de mutualiser les règles de production utilisées habituellement. On utilise le symbole % pour désigner un nom générique dans la cible et dans les dépendances.

Règle d'inférence prédéfinies

Il existe des règles d'inférence prédéfinies par `make(1)` :

```
%.o: %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

%: %.o
	$(CC) $(LDFLAGS) $^ $(LOADLIBES) $(LDLIBS) -o $@
...
```

Makefile

Exemple avec règles d'inférence

Exemple (Makefile)

```
CC=gcc
CFLAGS=-Wall -std=c99 -g
LDFLAGS=-g
TARGET=helloworld

all: $(TARGET)

$(TARGET): hello.o main.o
	$(CC) $(LDFLAGS) $^ -o $@
hello.o main.o: hello.h

# redéfinition non nécessaire dans ce cas d'une règle d'inférence prédéfinie
%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ $<
clean:
	rm -f *.o
mrproper: clean
	rm -f $(TARGET)
```

Plan

Problématique

Pourquoi générer un Makefile ?

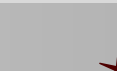
Il est difficile de prendre en compte certaines spécificités du système ou certains choix de développement (debug, production) via les Makefile. Il est alors nécessaire d'avoir un script de configuration (généralement appelé `configure`) qui va analyser le système et les arguments pour générer un Makefile adapté à partir d'un fichier `Makefile.in`.

Exemple

Pour installer une application à partir de son code source, on procède généralement de la manière suivante :

```
$ ./configure
$ make
# make install
```

Autotools



Autotools

Les autotools sont un ensemble de programmes capables de générer un script `configure` portable et le `Makefile.in` associé :

- `autoconf` : prend un `configure.ac` et génère un script `configure`
- `automake` : prend des `Makefile.am` et génère des `Makefile.in`

Les autotools sont basés sur le shell (pour le script `configure`) et sur le langage de script M4 (pour les fichiers `Makefile.am` et `configure.ac`)

- Principal avantage : Le script `configure` généré peut être exécuté sur n'importe quel Unix (portabilité)
- Principal inconvénient : Cet ensemble d'outils est très complexe à prendre en main et à maîtriser

CMake

CMake

CMake est un programme qui remplace les autotools et le script `configure`. Il génère des Makefile mais est capable de générer d'autres formats de projets. Il utilise des fichiers `CMakelists.txt` qui utilisent un langage très simple avec de très nombreuses commandes pour faire des tâches de base.

Exemple (CMakelists.txt)

```
project(HELLO)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall -std=c99 -g")
add_executable(helloworld hello.c main.c)
```

Sixième partie

Les entrées-sorties en C

Plan de ce cours

12 Les entrées-sorties en C
- Généralités
- Descripteur de fichier
- Flux `FILE*` et `DIR*`

Plan

Fichiers en C ★★★

Fichiers en C

- «Tout est fichier» : fichiers réguliers, répertoires, périphériques, etc
- Le langage C est lié à Unix : créé au départ pour implémenter Unix
- «Faites de chaque programme un filtre»

Les 2 API

Deux niveaux d'abstraction pour les fichiers :

- Descripteurs de fichier : appartiennent à l'interface du noyau, manipulation de **bas niveau** via des *appels systèmes*
- Flux : appartiennent à la bibliothèque standard du langage C →encapsulation des appels systèmes dans des fonctions de plus **haut niveau**

→ Le noyau n'a aucune connaissance de la notion de flux, il ne connait que les descripteurs de fichier.

Les 2 API

★★★

Choix

La plupart du temps, le programmeur se tournera vers les **flux** pour manipuler des fichiers pour les raisons suivantes :

- *portabilité* : l'utilisation de l'API descripteur de fichiers limite la portabilité aux seuls systèmes conformes à POSIX, alors que l'API flux est disponible sur tous les systèmes supportant le C standard.
- *performance* : les flux utilisent des **buffers** pour différer et limiter le nombre d'opérations d'écriture et de lecture dans l'espace du noyau ($\Rightarrow$ diminution du nombre d'appels systèmes sous-jacents)
- *simplicité* : la large panoplie de fonctions d'entrée et de sortie disponibles pour les flux n'existe pas pour les descripteurs de fichiers (qui ne permettent que des lectures et des écritures de blocs mémoire sans formatage des données)

$\rightarrow$ Certaines fonctionnalités sont spécifiques aux descripteurs de fichiers et nécessitent leur emploi (utilisés en programmation système, exemple : redirection vers une extrémité d'un tube anonyme)

Fichier texte / fichier binaire ⋆⋆

Définition (Fichier texte)

Un fichier texte est un fichier binaire, qui utilise un **charset**, dans lequel on ne stocke que des caractères imprimables ainsi que des "espaces blancs" (espace (' '), newline ('\n'), carriage return ('\r'), tabulation horizontale ('\t') ...). Un fichier texte qui peut être lu par un simple éditeur de texte.

Définition (Fichier binaire)

Un fichier binaire est un fichier informatique contenant des données sous forme d'octets pouvant prendre n'importe quelle valeur et qui n'ont donc de sens que pour le logiciel qui les utilise (et non pour les utilisateurs finaux). Si par stricte définition tout fichier est binaire, l'usage veut que l'on qualifie un fichier de binaire pour indiquer qu'il ne s'agit pas d'un fichier texte.

Exemples (Fichiers binaires)

images, fichier objet, video, fichier MP3, ficher .odt,...

Gestion des erreurs

errno(3)

★★★

Echec d'un appel système

Conventionnellement, les appels système qui renvoient un entier, **renvoient -1 en cas d'echec** ; et les appels systèmes qui renvoient un pointeur, **renvoient NULL en cas d'échec**.
Problème : comment avoir une information plus précise sur l'erreur rencontrée ?

La variable globale `errno`

Le fichier d'en-tête `<errno.h>` définit la variable globale entière `errno`, qui est renseignée par les appels système et quelques fonctions de la bibliothèque standard, **uniquement en cas d'échec**, pour **décrire la nature de l'erreur rencontrée**.

→ Un appel système ne modifie la variable `errno` qui si il échoue.
(Lorsqu'un appel système réussit, il ne modifie pas la variable `errno`).
RTFM : `errno(3)`

Gestion des erreurs

Symboles macrodéfinis

Exemples (Quelques erreurs)

Le fichier <errno.h> fournit la liste des erreurs potentielles (i.e. potentiellement affectées dans errno), ainsi que les messages standard associés. Extrait :

```
#define EPERM        1 /* Operation not permitted */
#define ENOENT       2 /* No such file or directory */
#define ESRCH        3 /* No such process */
#define EINTR        4 /* Interrupted system call */
#define EIO          5 /* Input/output error */
#define ECHILD      10 /* No child processes */
#define EAGAIN      11 /* Try again */
#define ENOMEM      12 /* Cannot allocate memory */
#define EACCES      13 /* Permission denied */
#define EFAULT      14 /* Bad address */
#define EPIPE       32 /* Broken pipe */
```

Gestion des erreurs ★★★

perror(3)

```
void perror(const char *s);
```

Affiche, sur l'erreur standard, `s`, « : », suivi d'un *message standard* associé à la valeur contenue dans `errno`, et un saut de ligne (pour faire cela, la fonction lit le contenu de la variable globale `errno`).

Exemple (ex_perror.c)

```
int main(int argc, char *argv[]) {
  int ret = mkdir("/toto", 0755);
  if (ret == -1){
    fprintf(stderr, \
        "errno = %d\n", errno);
    perror("mkdir toto");
  }
  return 0;
}
```

Exemple (Exécution)

```
$ ./ex_perror
errno = 13
mkdir toto: Permission denied
```

TODO

- **TOUJOURS** tester les valeurs de retour des appels systèmes
- **en cas d'erreur**, utiliser la fonction `perror(3)` ou directement `errno` pour déterminer son origine.

Plan

Descripteur de fichier ★★★

Définition (Descripteur de fichier)

Un descripteur de fichier est un entier (`int`) qui est associé à un fichier dans le système de fichiers. Cet entier est en fait un index dans la table des descripteurs de fichier (chaque processus dispose de sa propre table).

Descripteurs spéciaux

- 0 : entrée standard (`STDIN_FILENO`)
- 1 : sortie standard (`STDOUT_FILENO`)
- 2 : erreur standard (`STDERR_FILENO`)

Ouverture d'un fichier

★★

open(2)

open(2)

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Ouvre le fichier identifié par `pathname` et renvoie le descripteur de fichier correspondant ou -1 en cas d'échec.
Le paramètre `flags` une combinaison de plusieurs symboles macrodéfinis assemblés par un *OU* binaire (|).
Tout d'abord, il faut impérativement utiliser une (et une seule) des 3 constantes suivantes :

- `O_RDONLY` : Ouvre le fichier en lecture seule
- `O_WRONLY` : Ouvre le fichier en écriture seule
- `O_RDWR` : Ouvre le fichier en lecture et écriture

Ensuite, on peut préciser le mode d'ouverture avec par exemple :

- `O_APPEND` : Ouvre le fichier en ajout (toutes les écritures ont lieu en fin de fichier)
- `O_TRUNC` : Tronque le fichier
- `O_CREAT` : Crée le fichier s'il n'existe pas (⇒ 2è version avec permissions)

Exemple d'ouverture d'un fichier

Exemple

```
int fd; /* file descriptor */

fd = open("foo.txt", O_RDONLY);

if (fd == -1) {
  fprintf(stderr, "Error while opening %s!\n", "foo.txt");
  perror("open foo.txt");
  exit(EXIT_FAILURE);
}
```

Création d'un fichier

★★

creat(2)

creat(2)

`int creat(const char *pathname, mode_t mode);`

Crée le fichier identifié par `pathname`. Équivalent à `open(2)` avec les `flags` suivants : `O_CREAT | O_WRONLY | O_TRUNC`

Exemple

```
int fd = creat("bar.h", 0644);

if (fd == -1) {
  fprintf(stderr, "Error while creating %s!\n", "bar.h");
  perror("create bar.h");
  exit(EXIT_FAILURE);
}
```

Création d'un fichier

Le paramètre mode

Le paramètre mode

Il est est filtré à travers le umask du processus. Les constantes symboliques suivantes sont disponibles pour le paramètre mode :

```
S_IRWXU  00700 L'utilisateur (propriétaire du fichier) a les
               autorisations de lecture, écriture, exécution.
S_IRUSR  00400 L'utilisateur a l'autorisation de lecture.
S_IWUSR  00200 L'utilisateur a l'autorisation d'écriture.
S_IXUSR  00100 L'utilisateur a l'autorisation d'exécution.
S_IRWXG  00070 Le groupe a les autorisations de lecture,
               écriture, exécution.
S_IRGRP  00040 Le groupe a l'autorisation de lecture.
S_IWGRP  00020 Le groupe a l'autorisation d'écriture.
S_IXGRP  00010 Le groupe a l'autorisation d'exécution.
S_IRWXO  00007 Les autres ont les autorisations de lecture,
               écriture, exécution.
S_IROTH  00004 Les autres ont l'autorisation de lecture.
S_IWOTH  00002 Les autres ont l'autorisation d'écriture.
S_IXOTH  00001 Les autres ont l'autorisation d'exécution.
```

Entrées sorties

Tables du système

★★

Table des descripteurs d'un **processus**

Cette table comprend pour chaque descripteur divers attributs (comme celui de fermeture sur recouvrement) et un pointeur sur une structure `file`

Table des fichiers ouverts du **système**

Chaque entrée de cette table a la structure `file`. On y trouve :

- le nombre total de descripteurs qui pointent sur l'entrée
- le mode d'ouverture du fichier
- **l'offset (ou position courante)** : repère le prochain octet à lire ou à écrire dans le fichier
- un pointeur sur une entrée de la table des inodes en mémoire

Entrées sorties

Tables du système

★★

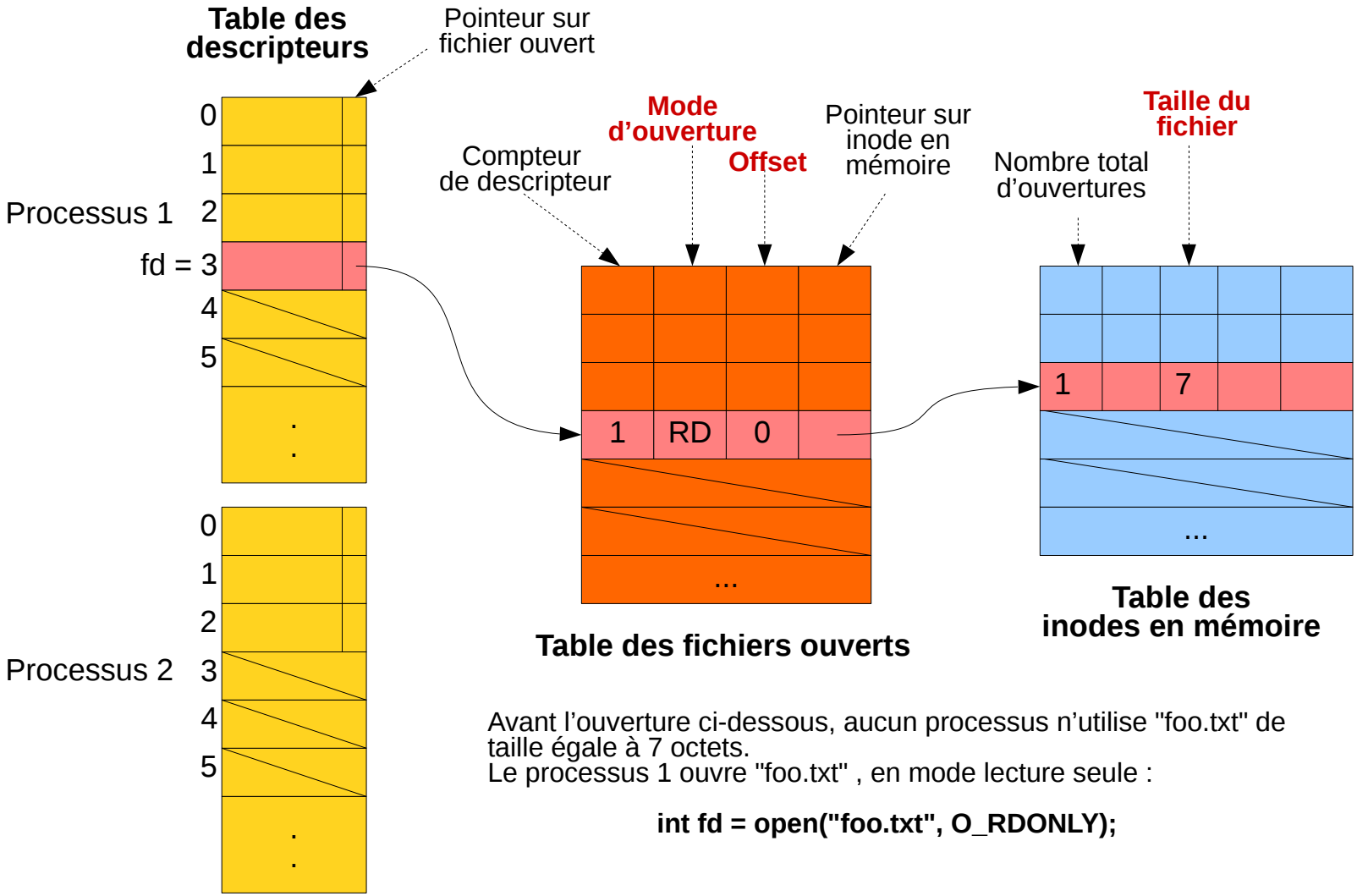
Table des inodes en mémoire du **système**

Tout inode manipulé par le système y est chargé. Chaque entrée de cette table contient :

- le nombre total d'ouvertures (nombre d'entrées de la table des fichiers ouverts qui pointe sur l'inode en mémoire)
- les données correspondant au inode (identifiant du périphérique, numéro d'inode, compteur de liens physiques, permissions, **taille**, adresses des blocs mémoire, ...)
- le vecteur de fonctions de manipulation du inode en mémoire

Entrées sorties

Tables du système (1/3)

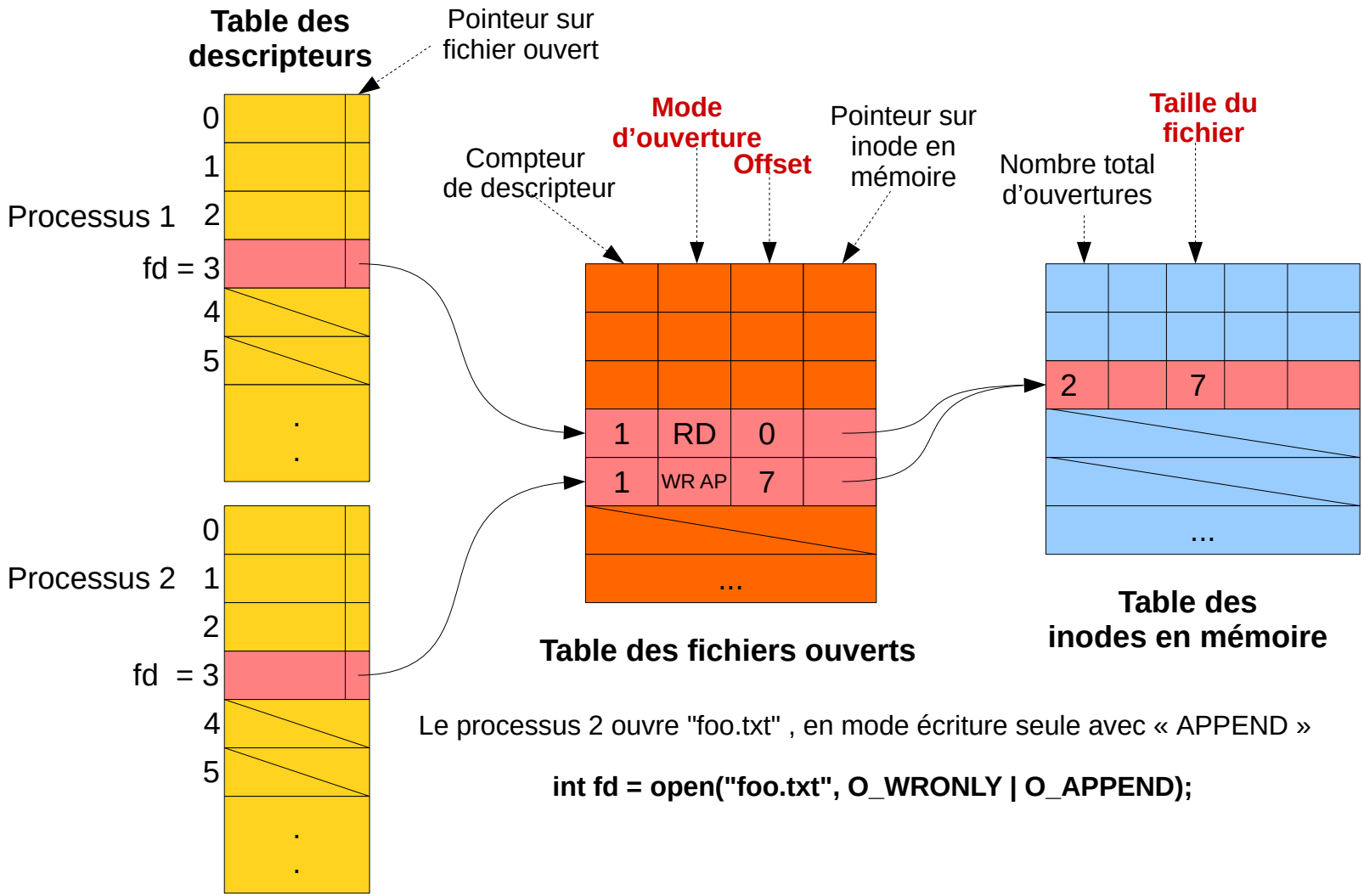


Avant l'ouverture ci-dessous, aucun processus n'utilise "foo.txt" de taille égale à 7 octets.
Le processus 1 ouvre "foo.txt" , en mode lecture seule :

int fd = open("foo.txt", O_RDONLY);

Entrées sorties

Tables du système (2/3)

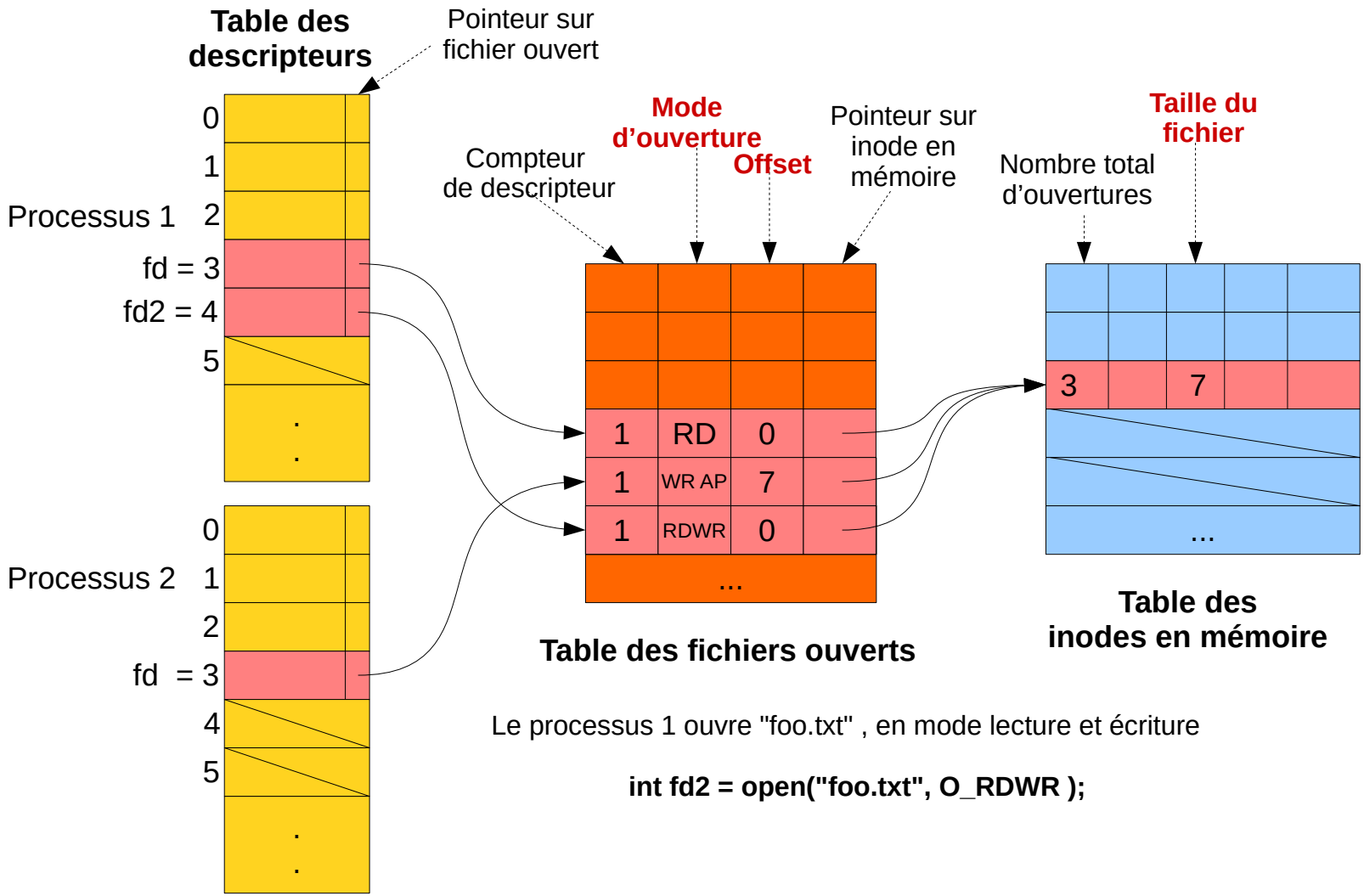


Le processus 2 ouvre "foo.txt" , en mode écriture seule avec « APPEND »

int fd = open("foo.txt", O_WRONLY | O_APPEND);

Entrées sorties

Tables du système (3/3)



Le processus 1 ouvre "foo.txt" , en mode lecture et écriture

int fd2 = open("foo.txt", O_RDWR);

Fermeture d'un fichier

★★

close(2)

close(2)

```
int close(int fd);
```

Ferme le descripteur de fichier `fd`. Il est indispensable de tester sa valeur de retour, car c'est la dernière chance de détecter une éventuelle erreur !

Exemple

```
int fd = open("baz.c", O_WRONLY);
if (fd == -1) {
  fprintf(stderr, "Error while opening %s!\n", "baz.c");
  perror("open");
  exit(EXIT_FAILURE);
}
do_something_useful_with(fd);
int ret = close(fd);
if (ret == -1) { perror("close"); }
```

Fermeture d'un fichier

★★

Effets en cascade

Effets en cascade d'un appel de `close(2)`

- le descripteur correspondant est libéré dans la table des descripteurs du processus ;
- le compteur de descripteur de la structure `file` correspondante dans la table des fichiers ouverts est décrémenté ;
- si ce compteur devient nul, l'entrée dans la table des fichiers ouverts est libérée, et le compteur d'ouverture du inode en mémoire correspondant est à son tour décrémenté ;
- si ce compteur devient lui aussi nul, le inode en mémoire est libéré ; et dans le cas où le compteur de liens physiques de ce inode est également nul, le fichier est physiquement supprimé : les blocs de données et le inode sont alors libérés.

Lecture dans un fichier ordinaire

★★★

Lecture non formatée

`read(2)`

```
ssize_t read(int fd, void *buf, size_t count);
```

Demande à lire dans le fichier associé au descripteur de fichier `fd`, jusqu'à `count` octets, et place les octets lus dans le tampon `buf`

Algorithme de lecture

- la fonction renvoie -1 si une erreur s'est produite (descripteur inconnu, lecture non autorisée, ...) ⇒ il faut consulter `errno`
- sinon
  - Si l'**offset** est inférieur à la taille du fichier, la primitive lit des octets dans le fichier, à partir de l'**offset**, jusqu'à ce que le nombre d'octets demandés (`count`) soit atteint, ou que la fin de fichier soit atteinte. La primitive renvoie alors le nombre d'octets effectivement lus, et la valeur de l'**offset** est augmentée de ce nombre.
  - Si l'*offset* est supérieur ou égal à la **taille** du fichier, la primitive **renvoie 0**.

Lecture dans un fichier ordinaire

★★

`read(2)`

Exemple (ex_read.c)

```
char buf[5]="23";//0

int fd = open("fic", O_RDONLY);
if (fd == -1){
  perror("open");
  exit(EXIT_FAILURE);
}

ssize_t s = read(fd,buf,5);//1
s = read(fd, buf, 5);//2
s = read(fd, buf, 5);//3
s = read(fd, buf, 5);//4

int ret = close(fd);
if (ret == -1) { perror("close"); }
```

Exemple (Exécution)

```
$ cat fic
abcdefghijkl
$ od -A x -t x1 fic
61 62 63 64 65 66 67 68 69 6a
6b 6c 0a
```

	buf					s
0	32	33	0	0	0	X
1	61	62	63	64	65	**5**
2	66	67	68	69	6A	**5**
3	6B	6C	0A	69	6A	**3**
4	6B	6C	0A	69	6A	**0**

Exemple : lecture séquentielle dans un fichier ordinaire ⋆⋆⋆

Exemple (Lecture séquentielle d'un fichier ordinaire)

```
char buf[BUFSIZE];
ssize_t sz;
int fd = open("file.txt", O_RDONLY);
if (fd == -1){
  perror("open");
  exit(EXIT_FAILURE);
}
while ((sz = read(fd, buf, BUFSIZE)) > 0) {
  do_something_with(buf, sz);
}
if (sz == -1) { perror("read"); }
int ret = close(fd);
if (ret == -1) { perror("close"); }
```

Écriture dans un fichier ordinaire ⋆⋆

Ecriture non formatée

`write(2)`

```
ssize_t write(int fd, const void *buf, size_t count);
```

Correspond à une demande d'écriture, dans le fichier associé au descripteur `fd`, de `count` octets lus à l'adresse `buf` dans l'espace d'adressage du processus.

Algorithme d'écriture

- la fonction renvoie -1 si une erreur s'est produite (descripteur inconnu, écriture non autorisée, dépassement de la taille maximale autorisée pour un fichier, ...) ⇒ il faut consulter la variable globale `errno`
- sinon, il y a écriture, dans le fichier, à partir de l'**offset** courant, d'octets lus à partir de l'adresse `buf`. Si le fichier a été ouvert avec le flag `O_APPEND`, l'**offset** est positionné en la fin du fichier avant l'écriture. Le nombre d'octets écrits est renvoyé : une valeur positive inférieure à `count` indique un problème (disque plein par exemple), mais ne constitue pas une erreur. Le retour de la fonction signifie que l'écriture a été réalisée dans la *mémoire cache* du noyau. L' **offset** courant est augmenté du nombre d'octets écrits.

Exemple : lecture/écriture séquentielle dans un fichier ⋆⋆

Exemple (ex_read_write.c)

```
char buf[2] = {'y', 'z'};
int fd = open("fic", O_RDWR);
if (fd == -1){
  perror("open");
  exit(EXIT_FAILURE);
}

ssize_t s = read(fd,buf,2);
printf("s = %zd; %c %c\n", \
          s, buf[0], buf[1]);
write(fd, "012", 3);

int ret = close(fd);
if (ret == -1) { perror("close"); }
```

Exemple (Exécution)

```
$ echo abcd > fic
$ ./ex_read_write
s = 2; a b
$ cat fic
ab012$
```

Positionnement dans un fichier ★★

`lseek(2)`

`lseek(2)`

`off_t lseek(int fd, off_t offset, int whence);`
permet à un processus de modifier l'**offset** (ou **position courante**) de l'entrée de la table des fichiers ouverts associée au descripteur `fd` sans effectuer de lecture ni d'écriture. La nouvelle position courante est obtenue en ajoutant le paramètre entier relatif `offset` à une *valeur déduite* du paramètre `whence`.
Valeurs possibles pour `whence` :

`Constante`	**Valeur déduite**
`SEEK_SET`	0 (début du fichier)
`SEEK_CUR`	Position courante
`SEEK_END`	Taille (fin) du fichier

Cet appel système renvoie l'**offset**, mesuré en octets depuis le début du fichier, ou -1 en cas d'erreur.
→Pour connaître la valeur de l'**offset**, il suffit d'utiliser :

```
off_t pos = lseek(fd, 0, SEEK_CUR);
```

RTFM : `lseek(2)`

Positionnement dans un fichier ⋆⋆

Exemple (ex_lseek.c)

```
char buf[5]="to"; off_t pos; ssize_t nb_read;
int fd = open("fic.txt", O_RDONLY);
pos = lseek(fd, 0, SEEK_END);
printf("1.Taille du fichier : %ld\n", pos);
nb_read = read(fd, buf, 2);
printf("2.nb_read : %zd : %c %c\n",\
    nb_read, buf[0], buf[1]);
pos = lseek(fd, 1, SEEK_SET);
nb_read = read(fd, buf, 2);
printf("3.nb_read : %zd : %c %c\n",\
    nb_read, buf[0], buf[1]);
pos = lseek(fd, 0, SEEK_CUR);
printf("4.pos : %ld\n", pos);
pos = lseek(fd, -1, SEEK_END);
printf("5.pos : %ld\n", pos);
nb_read = read(fd, buf, 2);
printf("6.nb_read : %zd : %c %c\n",\
    nb_read, buf[0], buf[1]);
```

Exemple (Exécution)

```
$ echo abcdef > fic.txt
$ ./ex_lseek
1.Taille du fichier : 7
2.nb_read : 0 : t o
3.nb_read : 2 : b c
4.pos : 3
5.pos : 6
6.nb_read : 1 :
c
$
```

Plan

Flux

★★★

Définition (Flux)

Un flux (*stream*) est un pointeur sur une structure opaque de type `FILE` qui représente un fichier dans le système de fichier. La structure `FILE` encapsule un descripteur de fichier. En plus du descripteur de fichier, un flux dispose d'un **buffer** (ou **mémoire tampon**), et de deux indicateurs.

Flux spéciaux de type `FILE *`

- `stdin` : entrée standard
- `stdout` : sortie standard
- `stderr` : erreur standard

RTFM : `stdin(3)`

Ouverture d'un fichier

★★

`fopen(3)`

`fopen(3)`

```
FILE *fopen(const char *path, const char *mode);
```

Ouvre le fichier dont le chemin est contenu dans la chaîne pointée par `path` et lui associe un flux. L'attribut `mode` est une des chaînes de caractères suivantes et précise le mode d'ouverture :

- `"r"` : lecture seule, le fichier doit exister
- `"r+"` : lecture et écriture, le fichier doit exister, il n'est pas tronqué
- `"w"` : écriture seule, tronque le fichier s'il existe, le crée sinon
- `"w+"` : lecture et écriture, tronque le fichier s'il existe, le crée sinon
- `"a"` : écriture en fin de fichier, le fichier est créé s'il n'existe pas
- `"a+"` : écriture en fin de fichier, plus lecture n'importe où

Équivalence mode de fopen(3) vs flags de open(2) ⋆⋆

Extrait de la man page de fopen(3)

fopen(3) **mode**	open(2) **flags**
"r"	O_RDONLY
"w"	O_WRONLY | O_CREAT | O_TRUNC
"a"	O_WRONLY | O_CREAT | O_APPEND
"r+"	O_RDWR
"w+"	O_RDWR | O_CREAT | O_TRUNC
"a+"	O_RDWR | O_CREAT | O_APPEND

Exemple d'ouverture d'un fichier

Exemple

```
FILE *fp = fopen("foo.txt", "r");

if (fp == NULL) {
  fprintf(stderr, "Error while opening %s!\n", "foo.txt");
  perror("fopen");
  exit(EXIT_FAILURE);
}
```

Fermeture d'un fichier

★★

`fclose(3)`

`fclose(3)`

```
int fclose(FILE *fp);
```

Ferme le flux pointé par `fp` :

- vide le **buffer** (associé au flux) en appelant `fflush(3)`
- ferme le descripteur de fichier sous-jacent

Renvoie EOF en cas d'échec, tout en positionnant `errno`.

Exemple

```
FILE *fp = fopen("baz.c", "w");
if (fp == NULL) {
  perror("fopen baz.c"); exit(EXIT_FAILURE);
}
do_something_useful_with(fp);
int ret = fclose(fp);
if (ret == EOF) { perror("fclose baz.c"); }
```

Lecture dans un fichier

Lecture non formatée

fread(3)

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

Demande la lecture, dans le fichier associé à `stream`, de `nmemb` éléments de données, chaque élément ayant une taille de `size` octet(s), et stocke les élements lus à partir de l'adresse `ptr`.
Renvoie le nombre d'éléments lus.

Exemple

```
#define BUFSIZE 256
char buf[BUFSIZE]; /* buffer */
FILE *fp = fopen("passwd", "r");

size_t n = fread(buf, sizeof(char), BUFSIZE, fp);
printf("I read %zu char(s) from 'passwd'.\n", n);
```

Écriture dans un fichier

★★

Ecriture non formatée

fwrite(3)

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

Écrit, dans le fichier associé à `stream`, `nmemb` éléments de données récupérés à partir de l'adresse `ptr` dans l'espace d'adressage du processus, chaque élément ayant une taille de `size` octet(s).
Renvoie le nombre d'éléments écrits.

Exemple

```
char *str = "blabla\n"; int n = 15;
FILE *fp = fopen("fic.txt", "w");
size_t n = fwrite(str, sizeof(char), strlen(str), fp);
printf("I wrote %zu char(s) to 'fic.txt'.\n", n);
n = fwrite(&n, sizeof(int), 1, fp);
printf("I wrote %zu int to 'fic.txt'.\n", n);
```

Écriture dans un fichier

Ecriture formatée

fprintf(3)

```
int fprintf(FILE *stream, const char *format, ...);
```

Idem `printf(3)`, mais permet d'écrire dans n'importe quel flux.
Permet de créer des **fichiers texte**.

Exemple

```
fprintf(stdout, "%s a %d ans et mesure %.2f m\n",
    "Alice", 20, 1.70);

fprintf(stderr, "ERROR! WARNING!\n");

FILE *fp = fopen("README", "w");
fprintf(fp, "README generated by %s\n", __FILE__);
fclose(fp);
```

Vider le buffer d'un flux ouvert en écriture ⋆⋆

fflush(3)

```
int fflush(FILE *stream);
```

Lorsqu'on appelle `fflush()`, la bibliothèque C invoque l'appel système `write()` sur les données présentes dans le **buffer** associé à `stream` qui n'étaient pas encore transmises.

Ce **buffer** est également vidé lorsqu'on ferme le flux ou suivant divers critères de vidage.

La fonction `fflush()` n'a d'effet que sur les flux ouverts en écriture.

Critères de vidage du buffer

- buffer complet : lorsque le buffer est plein
- buffer de ligne : lorsqu'il contient le caractère Line Feed ('\n') ou qu'il est plein

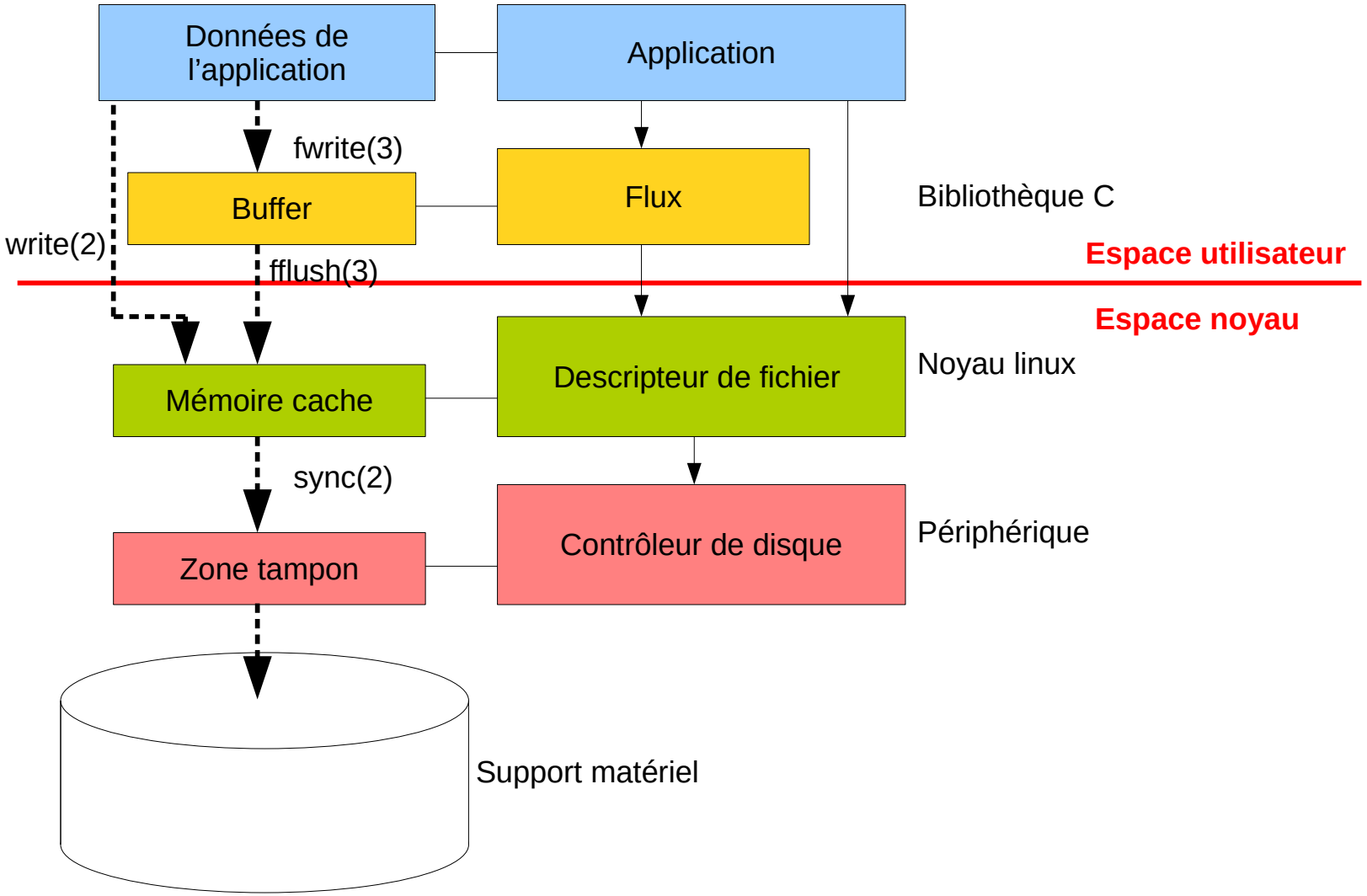
Remarque : un flux peut ne pas avoir de buffer.

RTFM : `fflush(3)` `setbuf(3)`

Buffers associés aux flux

★★★

3 buffers succeptibles de différer l'écriture



Etat d'un flux

★★

Indicateur de fin de fichier atteinte, et indicateur d'erreur

Chaque flux dispose de 2 indicateurs.

feof(3)

```
int feof(FILE *stream);
```

La fonction `feof()` renvoie une valeur non nulle si la fin de fichier a été atteinte. Attention, l'indicateur correspondant n'est **positionné** que lors d'une **tentative de lecture qui échoue**, c'est à dire lorsqu'on essaie de lire un ou plusieurs octets **après** le dernier octet du fichier. Si on lit uniquement le dernier octet du fichier sans tenter de lire un octet après ce dernier octet, il n'est pas positionné.

ferror(3)

```
int ferror(FILE *stream);
```

La fonction `ferror()` renvoie une valeur non nulle si une erreur s'est produite. Dans ce cas la variable globale `errno` peut être utilisée pour le diagnostic.

Exemple complet de lecture d'un fichier ★★★

Exemple

```
char buf[BUFSIZE];
size_t sz = 0;
FILE *fp = fopen("file.txt", "r");
if (fp == NULL){
  fprintf(stderr, "Error while opening %s!\n", "file.txt");
  perror("fopen");
  exit(EXIT_FAILURE);
}
while (!feof(fp)) {
  sz = fread(buf, sizeof(char), BUFSIZE, fp);
  if (ferror(fp)) { perror("fread"); break;}
  do_something_useful_with(buf, sz);
}
int ret = fclose(fp);
if (ret == EOF) { perror("fclose"); }
```

Comparaison des deux API

descripteur *vs* flux

Comparaison des deux API

descripteur	flux	rôle
`open(2)`	`fopen(3)`	ouverture
`close(2)`	`fclose(3)`	fermeture
`read(2)`	`fread(3)`	lecture non formatée
	`fscanf(3)`	lecture formatée (fichier texte)
	`fgets(3)`	lecture d'une ligne (fichier texte)
`write(2)`	`fwrite(3)`	écriture non formatée
	`fprintf(3)`	écriture formatée (fichier texte)
`lseek(2)`	`fseek(3)`	modification de l'offset
	`rewind(3)`	Mise à 0 de l'offset
`lseek(2)`	`ftell(3)`	consultation de l'offset
	`feof(3)`	indicateur de fin de fichier positionné ?
	`ferror(3)`	indicateur d'erreur positionné ?

Flux de répertoire ⋆⋆

Définition

Un flux de répertoire est un flux particulier, dédié au parcours d'un répertoire. C'est un pointeur sur une structure opaque de type `DIR` qui permet d'accéder aux données (i.e. liens) d'un répertoire. La structure `DIR` encapsule un descripteur de fichier.

Ouverture d'un flux de répertoire

★★

opendir(3)

opendir(3)

```
DIR *opendir(const char *name);
```

Ouvre un flux de répertoire associé au répertoire name

Exemple

```
DIR *dir; /* directory */

dir = opendir("/etc");

if (dir == NULL) {
  fprintf(stderr, "Failed to open: %s\n", "/etc");
  perror("opendir");
  exit(EXIT_FAILURE);
}
```

Fermeture d'un flux de répertoire

★★

closedir(3)

closedir(3)

```
int closedir(DIR *dirp);
```

Ferme le flux de répertoire dirp

Exemple

```
DIR *dir = opendir("/usr/bin");
if (dir == NULL) {
  fprintf(stderr, "Failed to open: %s\n", "/usr/bin");
  perror("opendir");
  exit(EXIT_FAILURE);
}
do_something_with(dir);
int ret = closedir(dir);
if (ret == -1) { perror("closedir"); }
```

Lecture des entrées d'un répertoire ⋆⋆

readdir(3)

readdir(3)

```
struct dirent *readdir(DIR *dirp);
```

Renvoie un pointeur sur une structure dirent représentant l'entrée suivante du répertoire associé à dirp. Elle renvoie NULL a la fin du répertoire, ou en cas d'erreur.

Exemple

```
DIR *dir = opendir("/home");

struct dirent *info = readdir(dir);
while (info != NULL) {
  printf("%s\n", info->d_name);
  info = readdir(dir);
}
closedir(dir);
```

Relations avec les descripteurs de fichiers ⋆

Depuis un flux : fileno(3)

```
int fileno(FILE *stream);
```

Renvoie le descripteur de fichier associé au flux `stream`

Depuis un flux de répertoire : dirfd(3)

```
int dirfd(DIR *dirp);
```

Renvoie le descripteur de fichier associé au flux de répertoire `dirp`

Depuis un descripteur de fichier : fdopen(3)

```
FILE *fdopen(int fd, const char *mode);
```

Renvoie un flux associé au descripteur de fichier `fd`

Attention !

Ne jamais mélanger des opérations sur les flux et des opérations sur les descripteurs de fichiers !

Septième partie

Processus

Plan de ce cours

13 Processus
- Création et exécution d'un processus
- Signaux
- Session et groupe de processus
- Contrôle des jobs
- Contrôle des processus

Plan

Définition d'un processus

★★★

Définition (Processus)

Un processus est une entité **dynamique** qui correspond à l'exécution d'un programme. Un processus est caractérisé par :

- un **espace d'adressage** qui définit l'ensemble des adresses (instructions du programme, données manipulées dont la pile et le tas) auxquelles il peut accéder.
- un **état** et l'ensemble des valeurs des registres du processeur correspondant à l'avancement de son exécution
- des informations liant le processus avec l'extérieur :
  - un **PID** (*Process IDentifier*) : nombre entier
  - un PPID (*Parent Process IDentifier*)
  - des propriétaires réel et effectif, des groupes propriétaires réel et effectif
  - un répertoire de travail, une table des descripteurs de fichier, un éventuel terminal de contrôle, ...

Création d'un processus ★★★

Création d'un processus

Un processus peut créer des processus, appelés processus fils et a lui-même un processus père dont l'identifiant est appelé PPID (*Parent Process IDentifier*). Les processus sont ainsi organisés en arbre.

`/sbin/init`

La racine de l'arbre est le processus `init` (`systemd(1)` : system daemon) qui est lancé par le noyau au démarrage. Il a le PID 1.

Voir le chapitre «Administration système».

RTFM : `pstree(1)`

Exécution d'un processus, ordonnancement ⋆⋆

Exécution

Un processus peut s'exécuter dans 2 modes différents :

- dans le **mode utilisateur**, il exécute des instructions ordinaires de son programme en ne manipulant que des données de son espace d'adressage ;
- dans le **mode système**, ou **noyau**, il exécute des instructions qui appartiennent au noyau du système, ce qui autorise un accès à des données extérieures à son espace d'adressage (en particulier les tables du système). Le passage en mode système a lieu suite à un **appel système**.

Ordonnancement

Sur les systèmes d'exploitation multi-tâches, il est possible d'exécuter plusieurs processus «en même temps». L'un des rôles du système est de permettre à tous les processus d'avancer dans leur exécution. Le module du noyau appelé **ordonnanceur** (*scheduler*) est chargé d'allouer le ou les processeurs aux différents processus.

Ordonnancement ⋆⋆

Ordonnancement

- Multi-tâche **coopératif** : les processus décident eux-même de rendre la main au noyau pour qu'il choisisse le prochain processus à exécuter.
- Multi-tâche **préemptif** : le noyau peut interrompre le processus en cours d'exécution pour en exécuter un autre. Deux stratégies :
  - **Temps partagé** : le noyau associe à chaque processus une priorité dynamique (recalculée régulièrement). Les processus prêts à être exécutés sont placées dans une *file d'attente avec priorité*. Quand un processeur est libre, l'ordonnanceur l'attribue au processus en tête de la file pendant une certaine durée.
  - **Temps réel** : impose pour chaque tâche des *délais temporels* (ou temps de réponse)

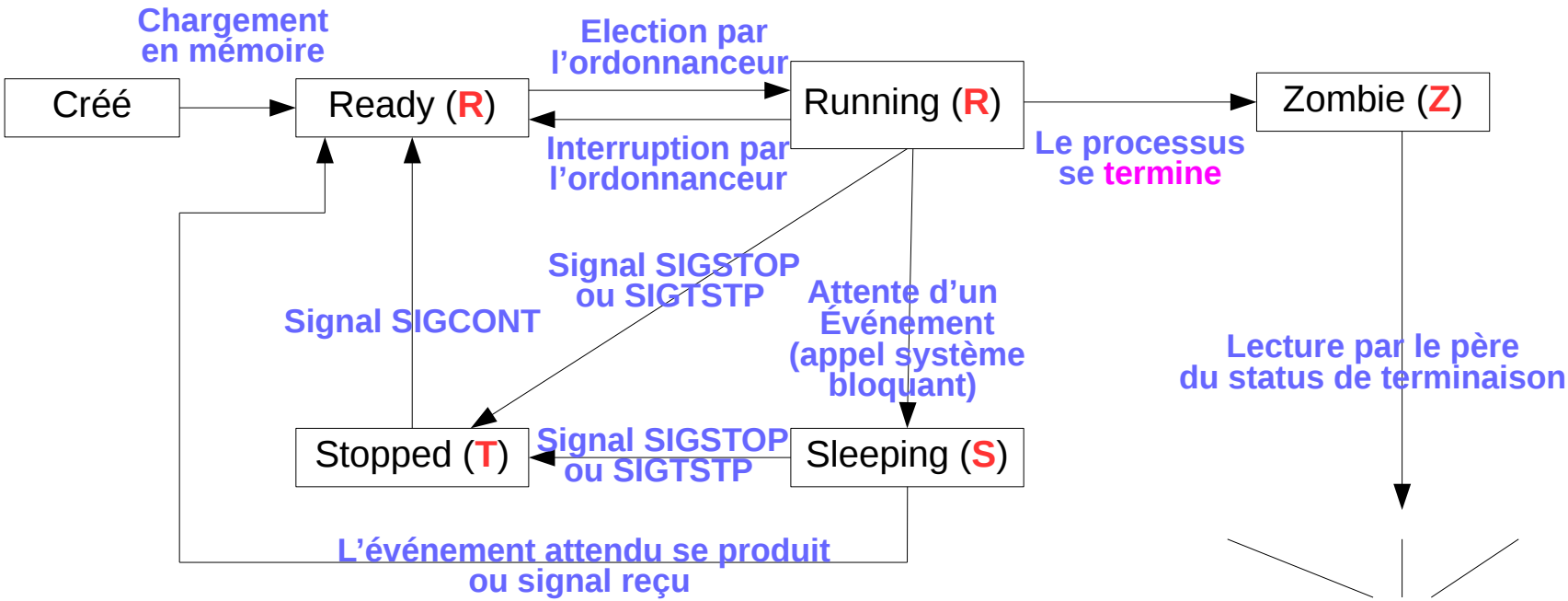
États d'un processus

★★★

États d'un processus

- **Créé** : le processus est créé et attend le passage dans l'état "Prêt" (automatique en temps partagé)
- **Prêt** (Ready `R`) : le processus est chargé en mémoire, il est éligible par l'ordonnanceur et attend l'allocation d'un processeur
- **En exécution** (Running `R`) : le processus est en cours d'exécution sur un processeur (en mode utilisateur, ou en mode système)
- **En sommeil** (Sleeping `S`) : le processus est en attente d'un événement extérieur (entrée/sortie, etc). Il ne consomme pas de temps CPU. En général, le sommeil est "interruptible" car le processus peut être réveillé prématurément si il reçoit un signal.
- **Arrêté** (Stopped `T` (Traced)) : le processus a été temporairement arrêté (ou *suspendu*) par un signal. Il ne s'exécute plus et ne réagira qu'à un signal de redémarrage.
- **Zombie** (`Z`) : le processus est terminé, soit normalement, soit anormalement, mais son père n'a pas pris connaissance de son status de terminaison.

États d'un processus



En **rouge** : les lettres utilisées par la commande **ps** pour représenter l'état du processus (champ **S**)

Fin d'un processus, zombies et orphelins ★★★

Fin d'un processus, zombies

Il est primordial de pouvoir déterminer si un processus a réussi à effectuer son travail correctement ou non. C'est pourquoi, quand un processus se termine, le noyau mémorise son **status de terminaison** (valeur "retournée" dans le cas d'une terminaison normale), et le processus passe automatiquement dans l'état zombie (`<defunct>`), en attendant que son processus père lise son status de terminaison. Un processus zombie n'a plus d'espace d'adressage, mais a toujours son PID (il occupe toujours une place dans les tables du système). Si le processus père ne lit pas le status de terminaison de son fils, celui-ci peut rester indéfiniment à l'état zombie. La seule façon d'éliminer un processus zombie, c'est que son père lise son status de terminaison.

Définition (Processus orphelin)

Un processus orphelin est un processus dont le père est terminé. Il est alors automatiquement adopté par le processus de PID 1, qui prend toutes les dispositions permettant l'élimination de ses fils zombies.

RTFM : `exit(2)`, `exit(3)`

Le système de fichiers /proc ⋆⋆

/proc

Le système de fichier /proc est un système de fichiers virtuel qui fournit une interface avec les structures de données du noyau. Il contient des informations sur le matériel, la configuration du noyau et sur les processus en cours d'exécution (un répertoire par processus nommé selon le PID) :

- cmdline : fichier contenant la ligne de commande complète du programme exécuté par le processus
- cwd : lien symbolique vers le répertoire courant du processus
- environ : l'environnement du processus
- exe : lien symbolique vers la commande en cours d'exécution (permet de la relancer)
- status : informations sur le processus
- fd : sous-répertoire contenant un lien pour chaque descripteur de fichier ouvert par le processus
- ...

RTFM : proc(5)

Processus, utilisateurs et permissions ⋆⋆

UID / GID réels et effectifs

Chaque processus possède un UID réel, un UID effectif, un GID réel et un GID effectif. L'UID réel et le GID réel du processus sont ceux de l'utilisateur ayant lancé le programme. L'UID effectif et le GID effectif du processus fixent les permissions accordées au processus.

SUID et SGID non positionnés

Pour ces applications, l'UID effectif est le même que l'UID réel. Idem pour le GID effectif. Les processus correspondants ont les mêmes droits que l'utilisateur qui les lancent.

SUID ou SGID positionnés

- SUID : quand un programme a le bit SUID positionné, l'UID effectif du processus est égal à l'UID du *propriétaire* du fichier et donc le processus s'exécute "sous l'identité" du propriétaire du fichier.
- SGID : quand un programme a le bit SGID positionné, le GID effectif du processus est égal au GID du *groupe propriétaire* du fichier.

Processus, utilisateurs et permissions ⋆⋆

Exemples

Exemple (Commandes rm et su)

Les 2 fichiers binaires exécutables appartiennent à l'utilisateur root et au groupe root. Seul su a le bit SUID positionné. Quand un utilisateur λ utilise la commande rm, les droits du processus sont ceux de λ. Quand il utilise la commande su, les droits du processus sont ceux de root.

Exemple (ex-getuid.c)

```
int main(void) {
  printf("UID reel = %u\n",\
         getuid());
  printf("UID effectif = %u\n",\
         geteuid());
  return 0; }
```

Exemple (Exécution)

```
$ make
# chown root:root ex-getuid
$ ls -l ex-getuid
-rwxrwxr-x. 1 root root ...
$ ./ex-getuid
UID reel = 1000
UID effectif = 1000
$

# chmod u+s ex-getuid
$ ls -l ex-getuid
-rwsrwxr-x. 1 root root ...
$ ./ex-getuid
UID reel = 1000
UID effectif = 0
$
```

Processus et descripteurs de fichiers ⋆⋆

Processus et descripteurs de fichiers

- Chaque processus possède sa propre table des descripteurs de fichiers.
- Chaque processus possède 3 descripteurs ouverts à sa création : 0 (entrée standard), 1 (sortie standard) et 2 (erreur standard).
- Un processus fils hérite, à sa naissance, de tous les descripteurs de son père.

Plan

Définition d'un signal

Définition (Signal)

Un signal est une forme limitée de communication inter-processus. Il s'agit d'une notification qui peut être envoyée à un processus pour le prévenir qu'un événement (souvent grave) s'est produit.

RTFM : `signal(7)`

Envoi d'un signal

Envoi d'un signal

Un signal peut être envoyé par :

- directement par le noyau
- un autre processus via l'appel système `kill(2)`
- un utilisateur via :
  - la commande `kill(1)` (il existe une version interne au shell et une version externe)
  - la frappe de caractères (de contrôle) sur le terminal de contrôle d'un processus

Tout envoi de signal "passe" par le **noyau**.

RTFM : `signal(7)`

Réception d'un signal ★★★

Réception d'un signal

Quand il reçoit un signal :

- le processus "s'interrompt" : il arrête d'"avancer" dans son code exécutable ;
- l'**action** (ou la **disposition**) courante associée au signal est exécutée ;
  - chaque signal est associé à une **action par défaut**
  - si l'action courante est un gestionnaire de signal (i.e. une fonction définie dans le programme), on dit alors que le signal est capté, capturé, intercepté ou attrapé)
- si l'action courante ne termine pas le processus et ne le stoppe pas, le processus reprend son exécution là où il avait été "interrompu".

RTFM : `signal(7)`

Action par défaut des signaux ⋆⋆

Action par défaut des signaux

Il existe un certain nombre d'actions (ou dispositions) par défaut :

- `Term` : Terminer le processus
- `Ign` : Ignorer le signal
- `Core` : Créer un fichier `core` et terminer le processus
- `Stop` : Arrêter le processus
- `Cont` : Continuer le processus s'il est actuellement arrêté

Définition (Terminaison d'un processus)

La terminaison d'un processus entraîne son passage dans l'état **`Zombie`**.

RTFM : `signal(7)`, `core(5)`

Les principaux signaux

Signaux d'interruption d'un processus

★★

Définition (Interruption d'un processus)

L'interruption d'un processus est la terminaison depuis un terminal.

SIGINT

- Combinaison de touche `CTRL+C`
- Action par défaut : `Term` ⇒ interromp le processus

SIGQUIT

- Combinaison de touche `CTRL+\`
- Action par défaut : `Core` ⇒ interromp le processus

Rq : ces signaux ne sont envoyés qu'au groupe de processus en avant-plan (contrôlé par le terminal)

Les principaux signaux

★★

Signaux "liés" à la terminaison d'un processus

SIGTERM

- Demande de terminer le processus
- Action par défaut : `Term`

SIGKILL (9)

- Tue le processus (le termine immédiatement, brutalement)
  → à utiliser en dernier recours
- Action par défaut : `Term` mais impossible à modifier !

SIGCHLD

- Signale au processus courant qu'un de ses processus fils est terminé
- Action par défaut : `Ign`

Les principaux signaux

★★

Signaux d'arrêt et de reprise d'un processus

Définition (Arrêt d'un processus)

L'arrêt d'un processus est la suspension en attendant une reprise.

SIGSTOP

- Arrête le processus temporairement
- Action par défaut : `Stop` mais impossible à modifier !

SIGTSTP

- Arrête le processus temporairement depuis un terminal par la combinaison `CTRL+Z`
- Action par défaut : `Stop`

SIGCONT

- Reprend le processus arrêté
- Action par défaut : `Cont`

Les principaux signaux

★★

Signaux d'exception

SIGBUS

- Erreur de bus (mémoire) c'est-à-dire problème d'alignement mémoire, problème d'adresse physique, etc.
- Action par défaut : `Core`

SIGFPE

- Erreur dans une opération arithmétique («Floating Point Exception»), flottante ou non, c'est-à-dire division par zéro, dépassement de capacité d'un entier signé, etc.
- Action par défaut : `Core`

SIGSEGV

- Erreur de segmentation («Segfault»), c'est-à-dire problème d'adresse virtuelle, lecture et/ou écriture à l'adresse 0 (`NULL`)
- Action par défaut : `Core`

Synthèse sur les signaux ★★

Synthèse sur les principaux signaux

Signal	Défaut	Gest.	Clavier
SIGINT	Term		CTRL+C
SIGQUIT	Core		CTRL+\
SIGTERM	Term		
SIGKILL	Term	Non	
SIGCHLD	Ign		
SIGSTOP	Stop	Non	
SIGTSTP	Stop		CTRL+Z
SIGCONT	Cont		
SIGBUS	Core		
SIGFPE	Core		
SIGSEGV	Core		

Plan

Session et groupe de processus ⋆⋆

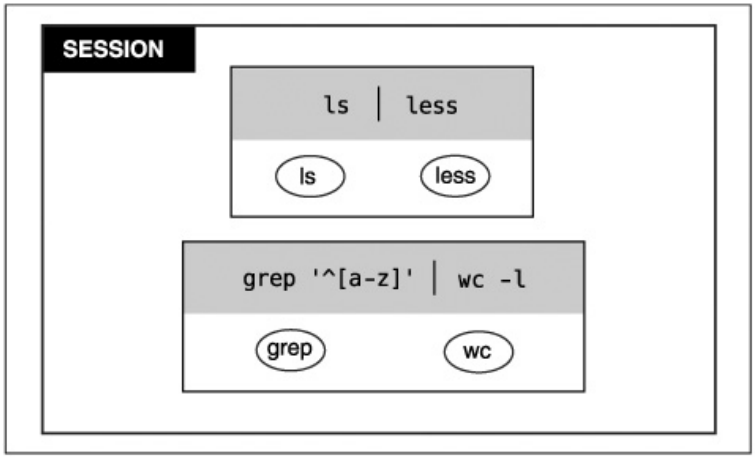
Session et groupe de processus

Les processus sont organisés hiérarchiquement en sessions et en groupes :

- Un processus appartient à un groupe de processus
- Un groupe de processus appartient à une session
- Une session peut être attaché à un terminal. Si c'est le cas, tous les processus de la session sont sous le contrôle de ce terminal.

Ce concept sert surtout aux shells pour implémenter le contrôle des jobs.

RTFM : `credentials(7)`



Identifiants de session et de groupe ⋆

Identifiants de session et de groupe

Chaque processus a un identifiant de session (SID, *Session ID*) et un identifiant de groupe de processus (PGID, *Process Group ID*), de type `pid_t`.

`getsid(2)`

```
pid_t getsid(pid_t pid);
```

Obtenir l'identifiant de session du processus dont le PID est *pid*

`getpgid(2)`

```
pid_t getpgid(pid_t pid);
```

Obtenir l'identifiant de groupe du processus dont le PID est *pid*

Session

Session

Une session est un ensemble de processus ayant le même SID.

- Le processus dont le PID est égal à son SID est le leader de session
- Dans le cas d'un shell, le shell est leader de session et cette session est attachée à un terminal de contrôle.
- Quand on ferme un terminal, le signal `SIGHUP` est envoyé à tous les processus sous son contrôle. Par défaut, ce signal les termine.
- Une session est créée avec `setsid(2)`, le processus créateur devient le leader de la session

Groupe de processus ⋆⋆

Groupe de processus

Un groupe de processus (ou *job*) est un ensemble de processus ayant le même PGID.

- Pour exécuter une commande, le shell crée un groupe de processus (qui peut contenir plusieurs processus concurrents communiquant via un ou plusieurs tubes)
- Le processus dont le PID est égal à son PGID est le leader du groupe de processus
- Tous les processus d'un même groupe appartiennent à la même session

Plan

Lancement de commandes en arrière plan ⋆⋆

Lancement de commandes en arrière plan

- Par défaut, les commandes lancées dans un shell interactif le sont en avant plan, c'est-à-dire que le shell rend la main quand le processus (ou les processus) correspondant(s) est (ou sont) terminé(s).
- Il est possible de lancer une commande en arrière plan en ajoutant `&` à la fin de la ligne. Le shell intéractif rend alors immédiatement la main à l'utilisateur
- Le numéro de `job` est indiqué, suivi du PID du processus
- Attention à ne pas confondre `&` et `&&` !

Exemple

```
$ ./Bc1.sh | ./Bc2.sh &
[1] 8897
$ ps -jH
  PID  PGID   SID TTY          TIME CMD
 3060  3060  3060 pts/1    00:00:00 bash
 8896  8896  3060 pts/1    00:00:11   Bc1.sh
 8897  8896  3060 pts/1    00:00:11   Bc2.sh
 8906  8906  3060 pts/1    00:00:00   ps
```

Jobs et terminal de contrôle ⋆⋆

Les différents "états" d'un tâche (ou job)

- en **avant plan** (*foreground*) : dans une session, il y au plus un job en avant plan. Son ou ses processus peuvent lire et écrire sur le terminal, et reçoivent le cas échéant respectivement les signaux SIGINT, SIGQUIT et SIGTSTP lors de la frappe des combinaisons de touches `CTRL+C`, `CTRL+\` et `CTRL+Z`. Si aucune tâche de la session n'est en avant plan, c'est le processus shell qui constitue à lui-seul le groupe de processus en avant-plan.
- en **arrière plan** (*background*) : les processus appartenant à un tel job ne peuvent pas lire sur le terminal, voire ne pas y écrire non plus. Ils ne sont pas affectés par la frappe des caractères de contrôle précédents (les signaux correspondants ne leur sont pas envoyés).
- **stoppée** ou **suspendue** : il est possible de suspendre (ou arrêter) un job qui s'exécute en avant plan en lui envoyant le signal `SIGTSTP` via la combinaison de touches `CTRL+Z`

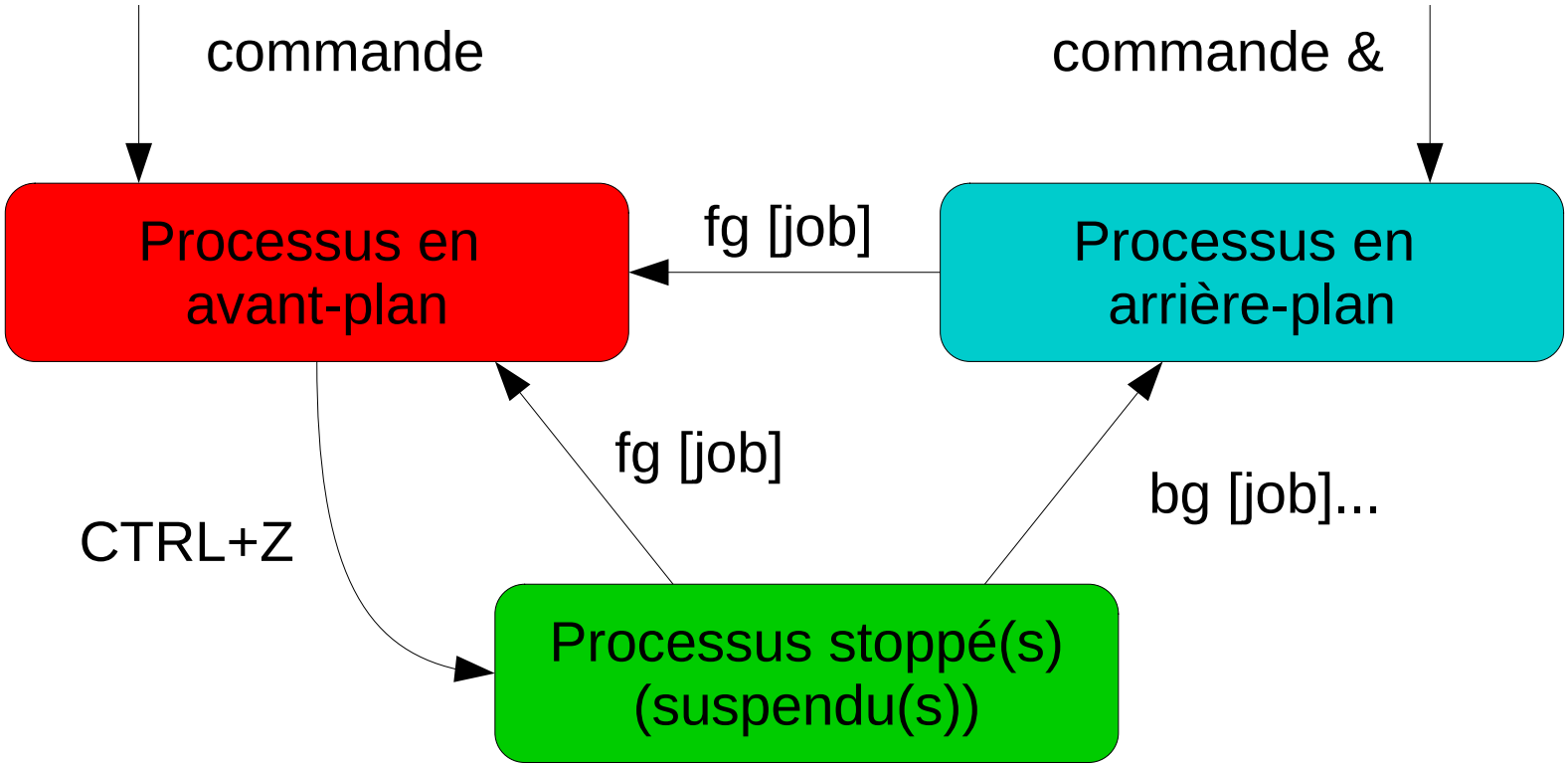
Commandes et changements d'"état" d'un job ⋆⋆

Commandes *internes* `jobs`, `bg`, `fg`

- `jobs [-l]` :
  affiche la liste de jobs actifs. L'option `-l` affiche les PIDs, en plus des informations normales. Le job repéré par le caractère + est le job courant (c'est lui qui sera traité lorsque l'argument correspondant à l'identification de la tâche est omis dans une des commandes suivantes).
- `bg [job]...` : *BackGround*
  relance en arrière-plan chaque `job` suspendu (en lui envoyant le signal `SIGCONT`), comme s'il avait été lancé avec `&`. Si aucun job n'est précisé, le job courant est basculé en arrière-plan.
- `fg [job]` : *ForeGround*
  place le `job` en avant-plan. Si aucun job n'est mentionné, le job courant est placé en avant-plan. Si le job considéré est suspendu, le signal `SIGCONT` lui est envoyé.

Commandes, changements d'"état" des processus d'un job

★★



Remarques : - la terminaison des processus n'est pas représentée dans ce diagramme
- la commande interne **kill** permet l'envoi d'un signal à tous les processus d'un job (cf. parties suivantes)

Commandes et changements d'"état" de jobs ⋆⋆

Exemple

```
$ ./Bc.sh | cat &
[1] 6653
$ sleep 500
^Z
[2]+  Stoppé                 sleep 500
$ ps -lH
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  5948  2662  0  80   0 - 31042 -      pts/1    00:00:00 bash
0 R  1000  6652  5948 93  80   0 -  1127 -      pts/1    00:00:14   Bc.sh
0 S  1000  6653  5948  0  80   0 - 28690 -      pts/1    00:00:00   cat
0 T  1000  6665  5948  0  80   0 - 28655 -      pts/1    00:00:00   sleep
0 R  1000  6672  5948  0  80   0 - 36267 -      pts/1    00:00:00   ps
$ fg %1
./Bc.sh | cat
^C
$ jobs -l
[2]+  6665 Stopped                 sleep 500
$
```

Plan

La commande ps(1) ⋆⋆

ps(1)

`ps [options]`

Affiche *diverses informations* sur les processus en cours

- `ps aux` : affiche tous les processus en cours (syntaxe BSD)
- `ps -ef` : affiche tous les processus en cours (syntaxe Posix)
- `ps -u eric` : affiche les processus ayant `eric` pour utilisateur effectif
- `ps -t /dev/pts/1` : affiche les processus sous le contrôle du `terminal /dev/pts/1`

Exemple

```
$ ps -lH
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3051  2258  0  80   0 - 30888 wait   pts/1    00:00:00 bash
0 R  1000  4845  3051 96  80   0 -  1284 -      pts/1    00:00:05   Bc.sh
0 R  1000  4852  3051  0  80   0 - 36968 -      pts/1    00:00:00   ps
```

La commande top(1)

top(1)

top
Affiche diverses informations sur les processus *dynamiquement* (toutes les secondes)

Exemple

```
top - 23:58:51 up  4:35,  3 users,  load average: 0.09, 0.18, 0.21
Tasks: 138 total,   2 running, 136 sleeping,   0 stopped,   0 zombie
Cpu(s): 14.0%us, 2.7%sy, 0.0%ni, 79.7%id, 3.7%wa, 0.0%hi, 0.0%si, 0.0%st
Mem:   3955660k total,  1907672k used,  2047988k free,    51984k buffers
Swap:  6530384k total,        0k used,  6530384k free,   752684k cached

  PID USER   PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2444 eric   20   0 1398m 565m  29m S   26 14.6  49:00.30 firefox-bin
 1320 root   20   0  246m  36m 8036 S    4  1.0  11:29.09 Xorg
 2255 eric   20   0  326m  31m  16m S    1  0.8   0:16.07 konsole
...
```

La commande `kill`

La commande externe

La commande externe `kill(1)`

`kill -l|-L`

Affiche la liste des signaux connus sur le système

`kill [-signal|-s signal] pid...`

Envoie le signal indiqué au(x) processus ou au(x) groupes de processus spécifié(s). Par défaut, le signal envoyé est `SIGTERM`.

- `-s KILL`, `-KILL`, `-9` : envoie le signal `SIGKILL`
- La liste des processus destinataires du signal peut être composée d'une liste où :
  - un entier `n` > 0 désigne le processus de pid `n`
  - un entier `n` < 0 désigne tous les processus du groupe `-n`

La commande kill

★★

La commande interne de bash, exemples

La commande interne kill (bash(1))

`kill -l` → affiche la liste des signaux connus sur le système

`kill [-signal|-s signal] pid|job...`

Envoie le signal indiqué aux processus spécifiés par pid ou par job. S'il n'y a pas d'indication de `signal`, le signal envoyé est `SIGTERM`.

Exemple

- Utilisation de la commande interne :
  - `$ kill -l` → affiche la liste des signaux connus
  - `$ kill 2997` → envoie SIGTERM au processus de PID 2997
  - `$ kill -9 -2990` → envoie SIGKILL aux processus du groupe 2990
  - `$ kill -- -299` → envoie SIGTERM aux processus du groupe 299
  - `$ kill -9 %2` → envoie SIGKILL aux processus du job %2
- Utilisation de la commande externe :
  - `$ $(which kill) -KILL -2990`

La commande killall(1)

killall(1)

`killall [-s,--signal signal] command...`

Envoie le signal indiqué au(x) processus en train d'exécuter les commandes mentionnées. Si plusieurs commandes ont été lancées, un signal est envoyé à tous les processus correspondants. Par défaut, le signal envoyé est SIGTERM.

- `-s KILL`, `-KILL`, `-9` : envoie le signal `SIGKILL`

Exemple

```
$ killall kate
```

La commande pidof(1)

pidof(1)

`pidof program...`
Donne le ou les PID du ou des processus qui exécutent *program*.

Exemple

```
$ pidof kate
23854 23851 22497
$ pidof chrome firefox
3903 2206 9714 8795 8774 8694
```

La commande nice(1)

nice(1)

`nice [-n N] command [arg]...`
Exécute une commande avec une priorité modifiée.
Ajoute `N` à la priorité par défaut (0).
Si l'option `-n` est absente, ajoute 10 à la priorité par défaut (0).
Les priorités vont de -20 (haute priorité) à 19 (basse priorité) et interviennent dans l'ordonnancement des processus (qui ont une politique "temps partagé").

Exemple

```
$ nice -n 19 ./gros-calcul
$ nice ./gros-calcul
# nice -n -10 ./tache-prioritaire
```

Questions de la fin

À propos de `killall(1)`

Comment implémenter une version simplifié de `killall(1)` en shell à l'aide de la commande interne `kill` et de `pidof(1)` ?

Huitième partie

Programmation système

Plan de ce cours

14 Programmation système
- Création de processus
- Exécution d'un programme
- Terminaison d'un processus
- Attente de la fin d'un processus fils
- Duplication de descripteur
- Tube anonyme
- Signaux
- Daemons

Plan

Création (ou duplication) de processus ★★★

Principes

Le premier processus du système (`init`) ainsi que quelques autres sont créés directement par le noyau au démarrage. Ensuite, pour créer un nouveau processus, il faut utiliser la fonction système `fork(2)`

`fork(2)`

```
pid_t fork(void);
```

Crée un nouveau processus, désigné sous le nom de **processus fils**, qui s'exécute de façon **concurrente** avec le processus qui l'a créé.

Héritage du père : le processus créé est une copie de son père pour une grande partie de ses attributs :

- il exécutera le même programme que son père sur une copie des données de celui-ci (notamment de la pile d'exécution) au moment de l'appel de `fork(2)`
- il possédera une copie des descripteurs de son père (notamment les mêmes entrées-sorties standard)
- il héritera d'une copie de l'environnement de son père
- UID GID réels et effectifs, répertoire de travail, gestionnaires de signaux ...

Création (ou duplication) de processus

★★★

`fork(2)`

`fork(2)` : un appel, deux retours (un dans le fils, un autre dans le père)

Une fois le nouveau processus créé, tout se passe comme si les processus père et fils avaient tous les deux réalisé un appel à `fork(2)`. Ainsi, à leur reprise d'exécution, les 2 processus effectuent un retour de l'appel de `fork(2)`. C'est donc le code de retour de `fork(2)` qui indique si on est dans le fils ou dans le père. Les codes de retour possibles sont :

- -1 : indique une erreur, c'est-à-dire aucun fils n'a été créé.
- 0 : indique qu'on est dans le fils.
- $p > 0$: indique qu'on est dans le père, le code de retour correspondant au PID du fils.

Obtention de PID

★★

getpid(2) et getppid(2)

getpid(2)

```
pid_t getpid(void);
```

Renvoie le PID du processus courant.

getppid(2)

```
pid_t getppid(void);
```

Renvoie le PID du processus père (du processus courant)

Exemple

```
pid_t pid = getpid();
pid_t ppid = getppid();

printf("I am %d and my father is %d.\n", pid, ppid);
```

Création (ou duplication) de processus ⋆⋆

Exemple basique

Exemple

```
int main (void) {
  pid_t pid_fils = fork();
  if (pid_fils == -1) {
    perror("fork"); return 1;
  }
  if (pid_fils == 0) { /* processus fils */
      fprintf(stdout, "Fils : PID=%d, PPID=%d\n",
              getpid(), getppid());
      return 0;
  }
  /* processus pere */
  fprintf(stdout, "Pere : PID=%d, PPID=%d, PID fils=%d\n",
          getpid(),getppid(),pid_fils);
  return 0;
}           /*      $ ./exemple-fork
                    Pere : PID=3589, PPID=2907, PID fils=3590
                    Fils : PID=3590, PPID=3589 */
    -> Quel le processus père du père ?
```

Création de processus : copie des données (1/2) ⋆

Exemple (ex-forkDonnees.c)

```
int n = 1000;
int main() {
  int m = 500;
  printf("1: Père : m et n : %d %d\n", m, n);
  switch(fork()) {
    case -1 : perror("fork"); exit(2);
    case 0 : /* fils */
      printf("2: Fils : m et n : %d %d\n", m, n);
      m *= 2; n *= 2;
      printf("3: Fils : m et n : %d %d\n", m, n);
      sleep(3); printf("6: Fils : m et n : %d %d\n", m, n);
      printf("6: Fils : &n=%p\n", &n); printf("6: Fils : &m=%p\n", &m);
      exit(0);
    default : /* père */
      sleep(2);
      printf("4: Père : m et n : %d %d\n", m, n); m *= 3; n *= 3;
      printf("5: Père : m et n : %d %d\n", m, n);
      printf("5: Père : &n=%p\n", &n); printf("5: Père : &m=%p\n", &m);
      sleep(2); } return 0; }
```

Création de processus : copie des données (2/2) ⋆

Exemple (Execution)

```
$ ./ex-forkDonnees
1: Père : m et n : 500 1000
2: Fils : m et n : 500 1000
3: Fils : m et n : 1000 2000
4: Père : m et n : 500 1000
5: Père : m et n : 1500 3000
5: Père : &n=0x601044
5: Père : &m=0x7ffc0fe7cedc
6: Fils : m et n : 1000 2000
6: Fils : &n=0x601044
6: Fils : &m=0x7ffc0fe7cedc
$
```

Remarques

- les adresses affichées ici sont des adresses virtuelles. Chaque processus dispose d'une table de pages qui permettent de les traduire en adresses physiques.
- Sous Linux, l'appel de `fork(2)` utilise la méthode "Copy-on-write". Toutes les données qui doivent être dupliquées pour chaque processus ne sont pas immédiatement recopiées. Tant qu'aucun des 2 processus n'a pas modifié des informations dans ses pages mémoires, il n'y en a qu'un seul exemplaire sur le système. Par contre, dès qu'un processus réalise une écriture dans une zone, le noyau assure la véritable copie des données.

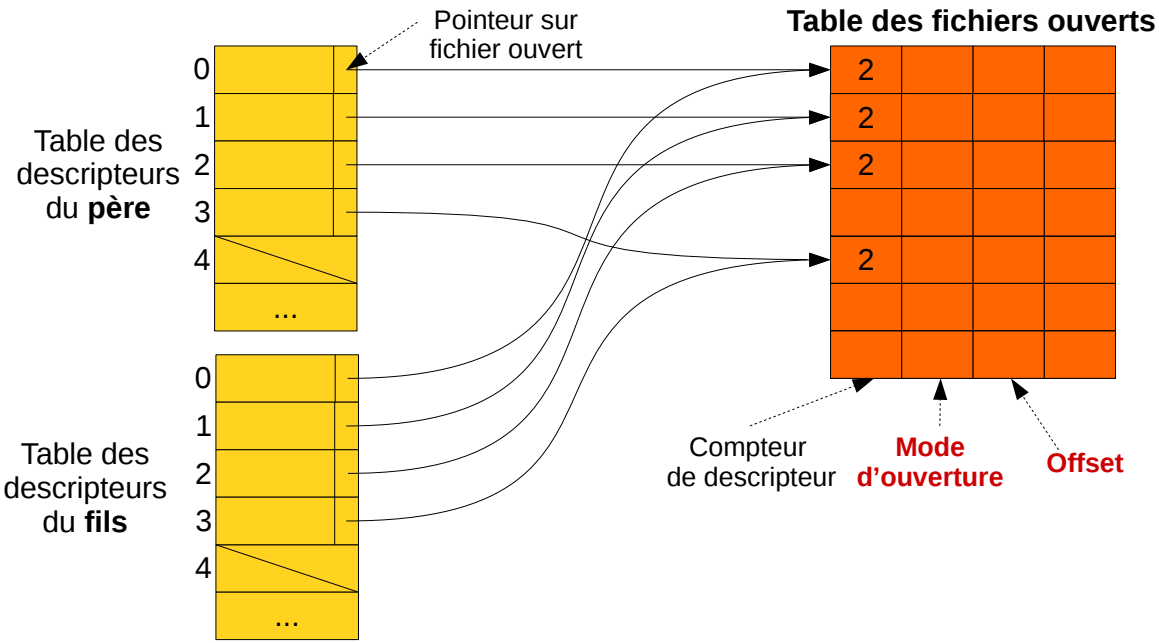
Création (ou duplication) de processus

Héritage de la table des descripteurs de fichiers (1/2)

★★

Héritage de la table des descripteurs de fichier

Tout entier correspondant à un descripteur de fichier du processus père avant l'appel de `fork(2)` correspondra à un descripteur de fichier du processus fils au retour de cet appel. C'est par ce mécanisme que les entrées-sorties standard se transmettent entre un processus père et ses fils. Il est essentiel de noter que les **descripteurs** correspondants du père et du fils **pointent exactement sur la même entrée de la table des fichiers ouverts**.



Création (ou duplication) de processus

★★

Héritage de la table des descripteurs de fichiers (2/2)

Exemple (ex-forkDescripteurs.c)

```
int main() {
  char *buf = calloc(8, sizeof(char));
  int desc;
  desc=open("toto",O_RDWR);
  if(fork()>0) {
    read(desc,buf,2);
    printf("Lu par pere : %s\n",buf);
    sleep(2);
    write(desc, "ab",2);
  }
  else {
    read(desc,buf,2);
    printf("Lu par fils : %s\n",buf);
    sleep(1);
    write(desc, "AB",2);
  }
  free(buf); return 0; }
```

Exemple (Exécution)

```
$ echo 0123456789 > toto
$ ./ex-forkDescripteurs
Lu par pere : 01
Lu par fils : 23
$ cat toto
0123ABab89
$
```

→ il manque les contrôles des valeurs de retour des appels système.

→ l'appel de `free` doit être fait dans le père et dans le fils

Plan

Lancement d'un nouveau programme ★★★

Principes

Le seul moyen de créer un nouveau processus, c'est d'appeler `fork(2)`, ce qui duplique le processus appelant. Seul le code de retour de `fork(2)` peut servir à exécuter des actions différentes. On va voir à présent comment charger un nouveau fichier exécutable en mémoire.

La seule façon de le faire est d'appeler une fonction de recouvrement, i.e. une fonction de la famille `exec`.

L'appel de l'une de ces fonctions permet de **remplacer l'espace mémoire du processus appelant par le code et les données du nouveau programme exécutable**. Ces fonctions ne reviennent qu'en cas d'erreur.

Il n'y a **pas de création d'un nouveau processus** au cours d'un recouvrement : la plupart des attributs du processus sont conservés (PID, PPID, ...)

L'intérêt de séparer "création du processus" et "recouvrement" est de permettre de réaliser, entre les 2, des opérations spécifiques telles que celles réalisées par un shell (redirections, ignorance de signaux, création d'une session ou d'un groupe de processus, ...).

Recouvrement de processus

★★

execve(2)

execve(2)

```
int execve(const char *filename, const char * argv[],
const char * envp[]);
```

Charge le programme correspondant au fichier *filename*. L'espace mémoire du processus appelant est remplacé par le code et les données du nouveau programme exécutable. Donc cette fonction ne revient jamais, à moins d'une erreur, auquel cas la fonction renvoie -1 ($\Rightarrow$ appeler `perror(3)`)

- `argv` correspond à un tableau de pointeurs de caractères, pointant sur les arguments (chaînes) **transmis à la fonction `main`** du nouveau programme. Son dernier élément doit être égal à `NULL`.
- `envp` correspond à un tableau de pointeurs de caractères, pointant sur les variables d'environnement (chaînes sous la forme `NOM=VALEUR`) **transmises** au nouveau programme. Son dernier élément doit être égal à `NULL`.

Recouvrement de processus

Exemple

Exemple

```
char *arg[] =
    { "cp", "/etc/passwd", "/tmp/passwd", NULL };

char *env[] = { NULL };

execve("/bin/cp", arg, env);
fprintf(stderr,"Error!\n");
perror("exec");
```

Recouvrement de processus

Variantes (1/2)

★★

La famille exec*

En plus de l'appel système `execve(2)`, il existe, dans la bibliothèque standard C, des fonctions, qui appellent en interne `execve(2)`, et qui permettent de "simplifier" le recouvrement de processus :

- `int execl(const char *filename, const char *arg, ...);`
- `int execlp(const char *filename, const char *arg, ...);`
- `int execle(const char *filename, const char *arg, ..., char * const envp[]);`
- `int execv(const char *filename, char *const argv[]);`
- `int execvp(const char *filename, char *const argv[]);`

RTFM : `exec(3)`

→ Les listes et les tableaux de pointeurs de caractères doivent se terminer par `NULL`

Recouvrement de processus

★★

Variantes (2/2)

La famille `exec*`

- Transmission des arguments `argv[i]` à la fonction `main` :
  - Les fonctions dont le suffixe commencent par un "l" (l comme liste) utilisent une liste d'arguments `arg0, arg1, ..., argn, NULL`
  - celles dont le suffixe commencent par un "v" (v comme vecteur) utilise un tableau de pointeurs de caractères à la manière du paramètre `argv` (`char *argv[]`) de la fonction `main`
- Les fonctions qui se terminent par un "e" transmettent explicitement l'environnement dans un tableau `envp` ; alors qu'avec les autres, l'environnement reste inchangé (la valeur de la variable globale `environ` est transmise à `execve(2)`)
- Recherche du fichier exécutable `filename` dans le système de fichiers :
  - Les fonctions se finissant par un "p" le recherchent dans les répertoires spécifiés dans la variable d'environnement `PATH` (si `filename` ne contient pas de `/`)
  - Les autres utilisent le chemin fourni (les chemins relatifs doivent être exprimés à partir du répertoire courant du processus appelant)

Recouvrement de processus

Exemple 2

★★

Exemple (prog.c)

```
extern char **environ;

int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < argc; i++){
    printf("%s\n", argv[i]);
  }
  i = 0;
  while (environ[i] != NULL){
    printf("%s\n", environ[i]);
    ++i;
  }
  exit(EXIT_SUCCESS);
}
```

Exemple (recouvre.c)

```
int main(int argc,char *argv[]) {
  char *envp[3];
  envp[0] = "X=AAAAA";
  envp[1] = "Y=BBB";
  envp[2] = NULL;
  execle( "./prog", "-prog",
          "/tmp/ddd", NULL, envp);
  perror("execle");
  exit(EXIT_FAILURE);  }
```

Exemple (Exécution)

```
$ ./recouvre
-prog
/tmp/ddd
X=AAAAA
Y=BBB
$
```

Recouvrement de processus ⋆

Attributs d'un processus après recouvrement

Les valeurs des attributs d'un processus sont conservés au cours d'un recouvrement à l'exception des suivants dans les conditions spécifiées :

- Propriétaire effectif : si le SUID est positionné sur l'exécutable chargé, le propriétaire de ce fichier devient le propriétaire effectif du processus
- Groupe propriétaire effectif : si le SGID est positionné sur l'exécutable chargé, le groupe propriétaire de ce fichier devient le groupe propriétaire effectif du processus
- Gestionnaires de signaux : le gestionnaire de signal par défaut est réinstallé pour les signaux captés (les dispositions des signaux ignorés ne sont pas modifiées)
- Descripteurs : si le bit `FD_CLOEXEC` de fermeture automatique en cas d'`exec` d'un descripteur a été positionné, ce descripteur est fermé (les autres descripteurs restent ouverts)

Plan

Terminaison normale / anormale ★★★

Terminaison normale / anormale

La terminaison d'un processus peut survenir :

- soit à demande du processus lui même (terminaison dite **normale**) en revenant de la fonction `main`, ou en appelant n'importe où `exit(3)` ou `_exit(2)`
- soit du fait de la délivrance d'un signal qui le tue brutalement (terminaison dite **anormale**)

Les shells `bash` et `dash` permettent de récupérer le code de retour de la dernière commande exécutée en utilisant `$?` :

- en cas de terminaison normale : `$?` = `status` transmis à `return` (fonction `main`), `exit(3)`, ou `_exit(2)`
- en cas de terminaison anormale : `$?` = 128 + numéro du signal ayant tué le processus

$\rightarrow$ `$?` est toujours < 256

Terminer un processus normalement ★★

`exit(3)`

`exit(3)`

```
void exit(int status);
```

Lorsqu'elle est appelée, la bibliothèque C :

- appelle toutes les fonctions de rappels (*callback*) enregistrées avec `atexit(3)` et `on_exit(3)`
- ferme tous les flux d'entrée-sortie, en vidant les buffers
- invoque l'appel système `_exit(2)` en lui transmettant `status`

→ L'appel de `exit(3)` ne revient jamais.

→ Conseil : toujours choisir `status` ≥ 0 et ≤ 127 (0 en cas de succès)

→ "`return status;`" depuis la fonction `main` entraîne l'appel de "`exit(status);`"

Terminer un processus normalement ⋆⋆

`_exit(2)`

`_exit(2)`

```
void _exit(int status);
```

Cet appel système exécute les tâches suivantes :

- il ferme les descripteurs de fichiers (en transférant les données aux périphériques)
- il libère les ressources système utilisées par le processus
- Tous ses processus fils sont adoptés par le processus de PID 1 (`init`)
- Le processus père reçoit un signal `SIGCHLD`
- Le status de terminaison du processus est **sauvegardé** dans son bloc de contrôle, et le processus devient **zombie**, jusqu'à ce que son père y accède

→ L'appel de `_exit(2)` ne revient jamais.

Terminer un processus normalement

Utilisation de atexit(3)

atexit(3)

```
int atexit(void (*function)(void));
```

Enregistre la fonction de rappel `function` : elle sera automatiquement appelée si le programme se termine normalement en appelant `exit(3)` (ou lors d'un `return` depuis le fonction `main`). Les fonctions ainsi enregistrées sont invoquées dans l'ordre inverse de leur enregistrement.

Exemple

```
void bye(void) {
  printf("Bye!\n");
}
int main() {
  atexit(bye);
  exit(EXIT_SUCCESS);
}
```

Terminaison anormale d'un processus ⋆⋆

Généralités

Certaines actions (exécution d'une instruction illégale, tentative de déréférencement d'un pointeur `NULL`, ...) déclenchent l'envoi d'un signal qui, par défaut, tue le processus.
Autres signaux qui, par défaut, tuent le processus destinataire : `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGKILL`, ...

`abort(3)`

```
void abort(void);
```

déverrouille (i.e. démasque) le signal `SIGABRT`, puis l'émet pour le processus appelant, ce qui, par défaut, le tue. L'appel d'`abort(3)` ne revient jamais.

La macro assert(3)

```
void assert(scalar expression);
```

si le symbole macrodéfini `NDEBUG` n'est pas défini et si `expression` est fausse (égale à zéro), affiche un message d'erreur sur la sortie d'erreur et termine l'exécution du processus en appelant `abort(3)`

Terminaison normale et anormale de processus ⋆

Exemple

Exemple (term.c)

```
int main (int argc,
          char *argv[])
{
  if (argc == 1){
    abort();
  }
  if (argc == 2){
    assert (argc != 2);
  }
  sleep(5);
  return 4;
}
```

Exemple (Exécution)

```
$ ./term
Aborted (core dumped)
$ echo $?
134
$ ./term fd
term: term.c:19: main:
Assertion ‘argc != 2’ failed.
Aborted (core dumped)
$ echo $?
134
$ ./term fd lm
^C
$ echo $?
130
$ ./term fd lm
$ echo $?
4
$
```

Plan

Attente de la fin d'un processus fils

★★★

Généralités

Généralités

- Les appels système `wait(2)` et `waitpid(2)` permettent l'élimination des processus (fils) zombies avec récupération des informations relatives à la terminaison normale ou anormale de ces processus.
- Ces appels système permettent aussi de *synchroniser* un processus avec la terminaison de ses fils.

Attente de la fin d'un processus fils ⋆⋆

`wait(2)`

`wait(2)`

```
pid_t wait(int *status);
```

- si le processus appelant ne possède aucun fils, la primitive renvoie `-1` et la variable `errno` a comme valeur `ECHILD`
- si le processus appelant possède au moins un fils zombie, la primitive renvoie le PID de l'un d'entre eux (c'est le système qui le choisit) et si l'adresse `status` est non `NULL`, `*status` fournit des informations sur la terminaison de ce processus zombie, qui disparaît des tables du système.
- si le processus appelant possède des fils, mais aucun fils zombie, il est **bloqué** jusqu'à ce qu'un de ses fils devienne zombie, ou bien jusqu'à ce que l'appel système soit interrompu

→ pour interpréter `*status`, il faut **utiliser des macros**
→ la primitive `wait(2)` supprime un zombie sans qu'il soit possible d'en choisir un en particulier

Attente de la fin d'un processus fils

★★

`waitpid(2)`

`waitpid(2)`

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Permet en bloquant ou non le processus appelant, d'attendre (en mode bloquant) ou de tester (en mode non bloquant) un changement d'état d'un (ou plusieurs) processus fils désignés par le paramètre *pid*.

- *pid* permet sélectionner le (ou les) processus attendu(s)
- *status* a le même rôle que pour `wait(2)`.
- *options* permet de spécifier l'attente d'autres types de changements d'état du fils (en plus de la terminaison) et de configurer l'appel pour qu'il soit non bloquant

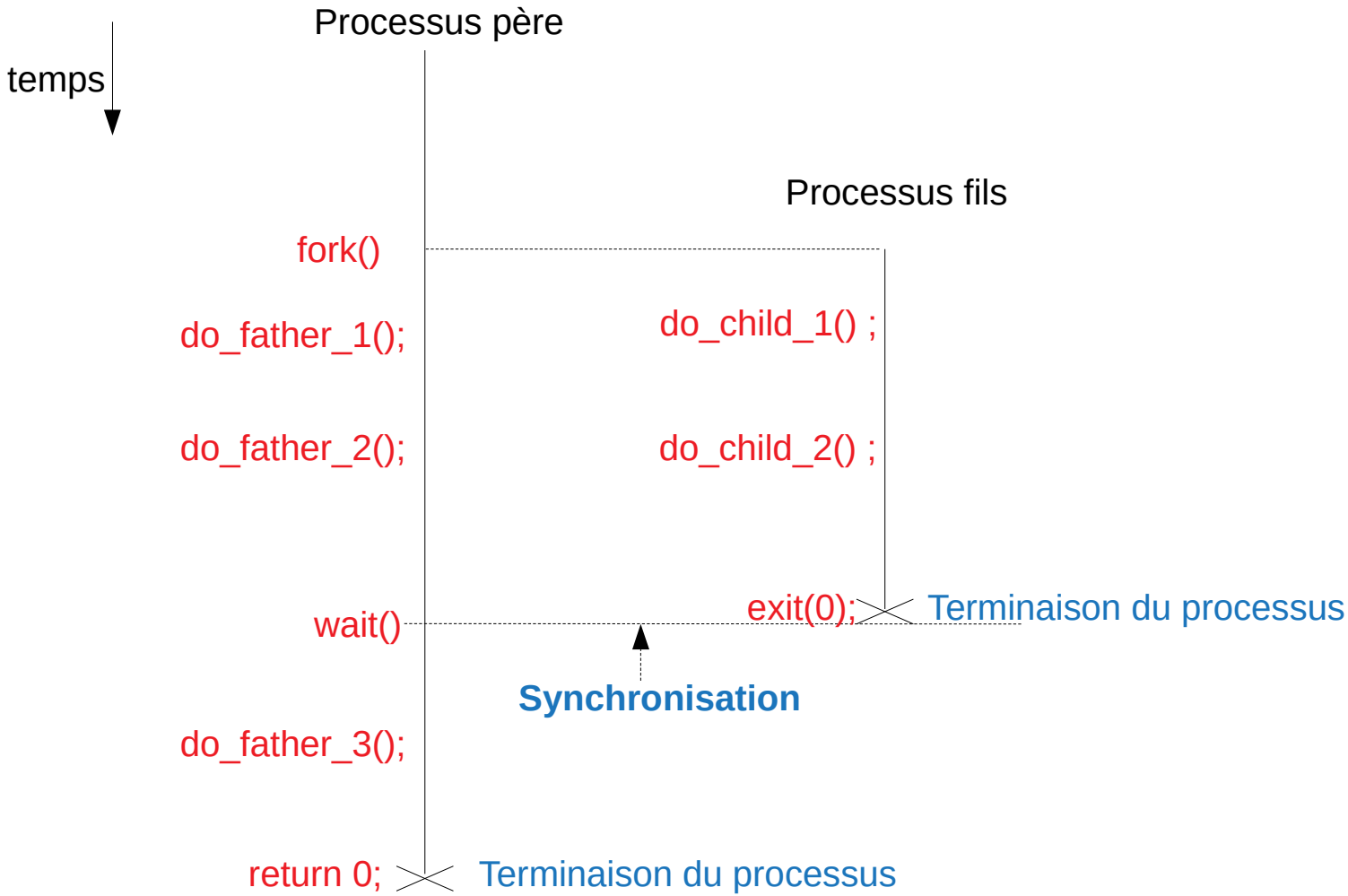
RTFM : `wait(2)`

Exemple complet (1/3) ⋆⋆

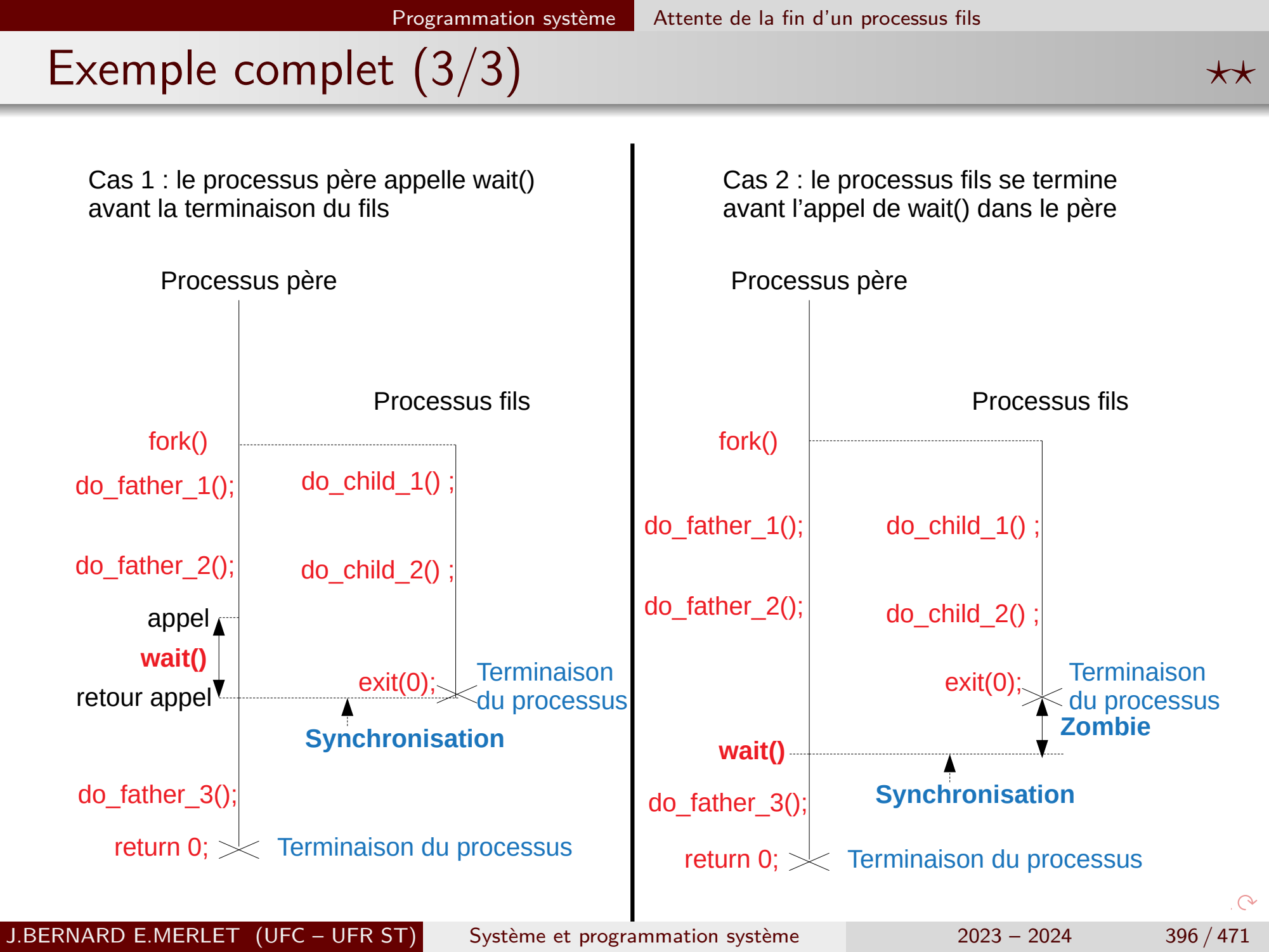
Exemple

```
int main() {
  pid_t pid = fork();
  if (pid == -1) {
    fprintf(stderr,"Error\n");
    perror("fork");
    exit(1);
  }
  if (pid == 0) {
    do_child_1();
    do_child_2();
    exit(0);
  }
  do_father_1();
  do_father_2();
  int status;
  pid = wait(&status);
  do_father_3();
  return 0;
}
```

Exemple complet (2/3) ⋆⋆



Exemple complet (3/3)



Plan

Duplication de descripteur ⋆⋆

Objectifs

La duplication de descripteur permet de réaliser des **redirections**, i.e. **modifier les associations entre descripteurs et fichiers physiques**.
Les shells l'utilisent couramment pour rediriger les entrées-sorties standard.
Pour exécuter une commande externe nécessitant une redirection, le shell crée un processus fils (qui hérite des descripteurs de son père), modifie l'association physique des descripteurs, puis le processus créé se recouvre par la commande (le recouvrement ne modifie pas la table des descripteurs (sauf quand le champ FD_CLOEXEC est positionné)).

Définition (Descripteurs synomymes)

La duplication de descripteur permet à un processus d'acquérir un descripteur synomyme d'un descripteur qu'il possède déjà. Etre **synomymes** signifie pour des descripteurs qu'ils **pointent sur la même entrée de la table des fichiers ouverts**.

Dupliquer un descripteur de fichier

★★

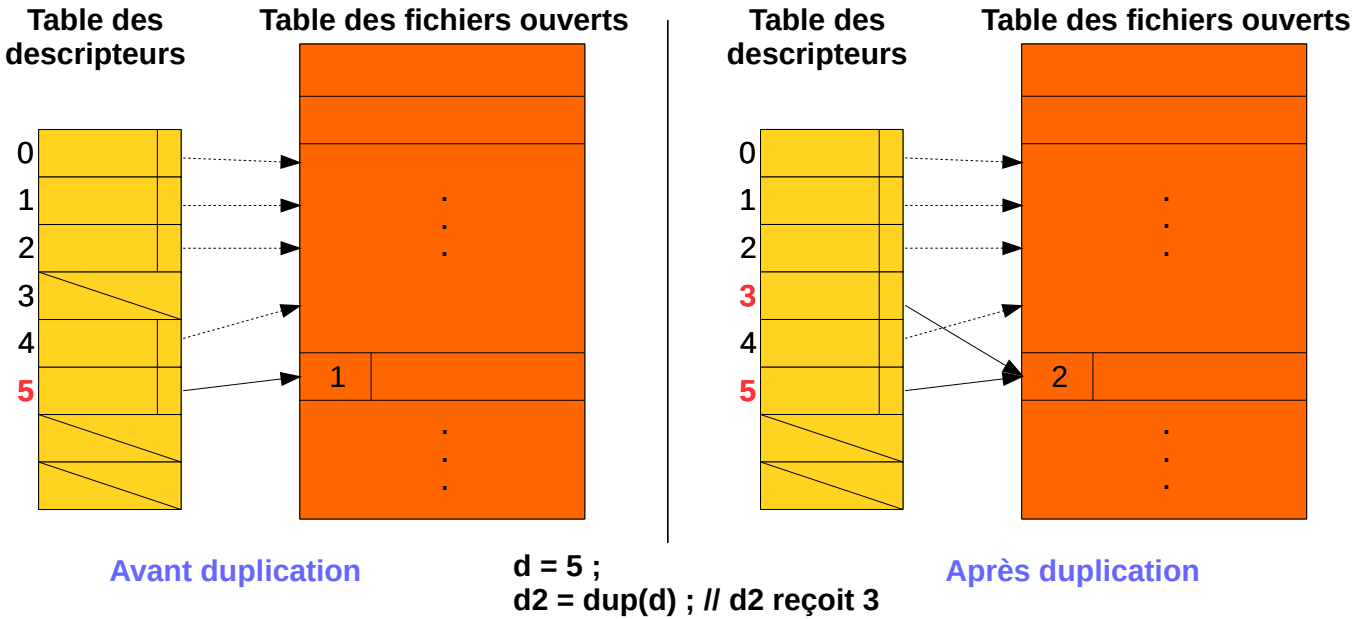
dup(2)

dup(2)

```
int dup(int oldfd);
```

Recherche le plus petit descripteur disponible dans la table des descripteurs du processus et en fait un synomyme du descripteur *oldfd*.

En cas de réussite, le compteur de descripteurs de l'entrée associée dans la table des fichiers ouverts est incrémenté, et la valeur retournée est le descripteur synomyme. Sinon, en cas d'erreur, *-1* est retournée.



dup(2)

Exemple

★

Exemple (ex-dup.c)

```
int main (int argc, char *argv[])
{
  int d1, d2; char c;
  d1 = open(argv[1], O_RDWR);
  if (d1 == -1){ perror("open");exit(1);}
  d2 = dup(d1);
  if (d2 == -1){ perror("dup");exit(1);}
  printf("d1 = %d, d2 = %d\n", d1, d2);
  read(d1, &c, 1); write(1, &c, 1);
  read(d2, &c, 1); write(1, &c, 1);
  write(1, "\n", 1);
  write(d1, "C", 1);
  write(d2, "D", 1);
  close(d1);
  close(d2);
}
```

Exemple (Exécution)

```
$ echo 012345 > toto
$ ./ex-dup toto
d1 = 3, d2 = 4
01
$ cat toto
01CD45
$
```

Dupliquer un descripteur de fichier

★★

dup2(2)

dup2(2)

```
int dup2(int oldfd, int newfd);
```

Permet de choisir le descripteur cible dont on souhaite qu'il devienne un synomyme d'un descripteur donné.

Force le descripteur *newfd* à devenir un synomyme du descripteur *oldfd*. Si *newfd* est déjà alloué, le système réalise au préalable une opération de fermeture (comme *close(newfd) ;*). La valeur renvoyée est *newfd* en cas de réussite, et *-1* en cas d'erreur.

Comment faire une redirection avec dup2(2) ? ⋆⋆

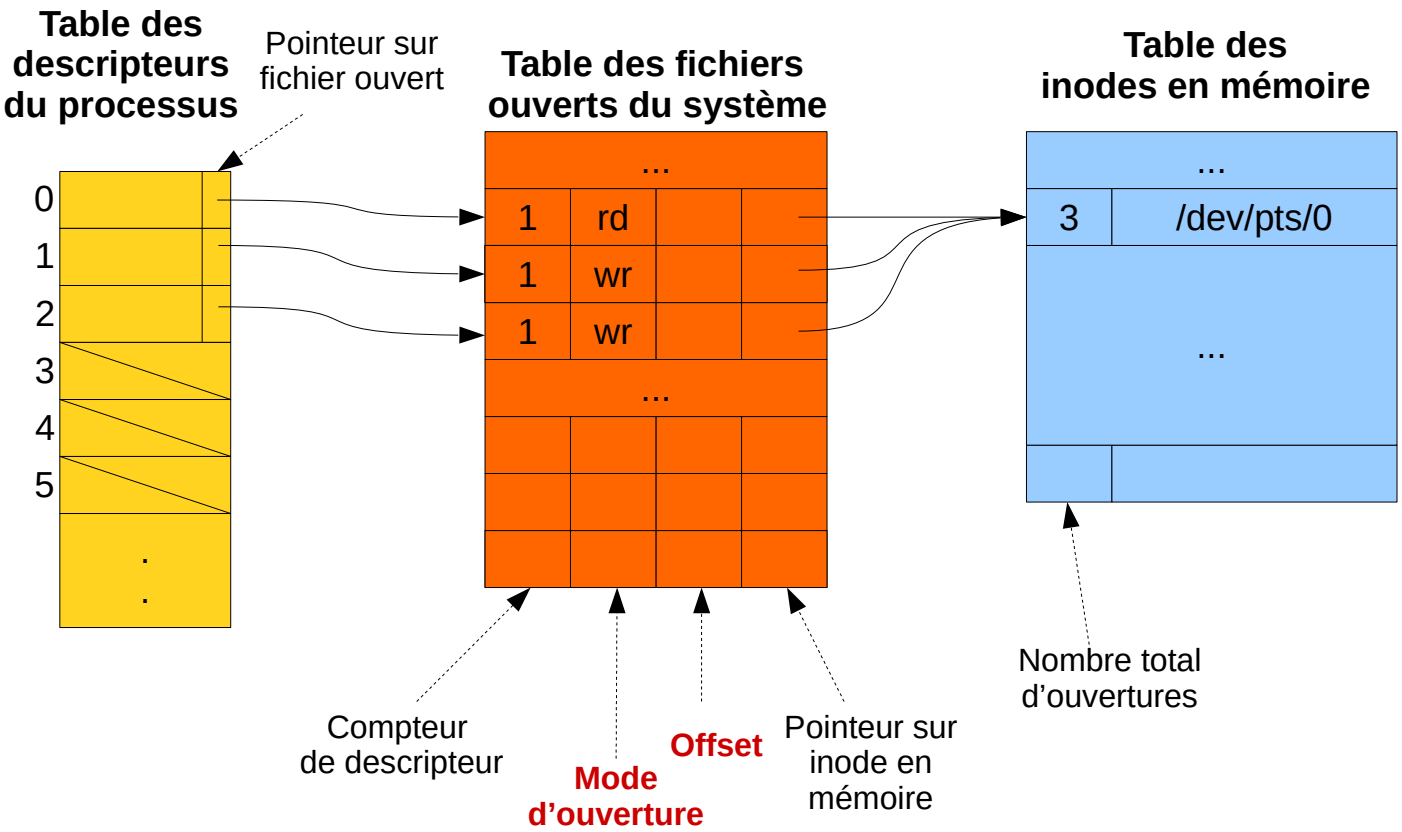
Exemple (ex-dup2.c)

```
int main (int argc, char *argv[])
{
  int d = open("out.txt",
               O_WRONLY | O_CREAT | O_TRUNC, 0666);
  if (d == -1) {perror("open"); exit(1);}
  int ret = dup2(d,1);
  if (ret != 1) {perror("dup2"); exit(1);}
  ret = close(d);
  if (ret != 0) {perror("close"); exit(1);}
  execl("/bin/ps", "ps", "-lH", NULL);
  perror("execl"); exit(1);
}
```

Comment faire une redirection avec dup2(2) ? ⋆⋆

Exemple (ex-dup2.c : tables du système - 1/4)

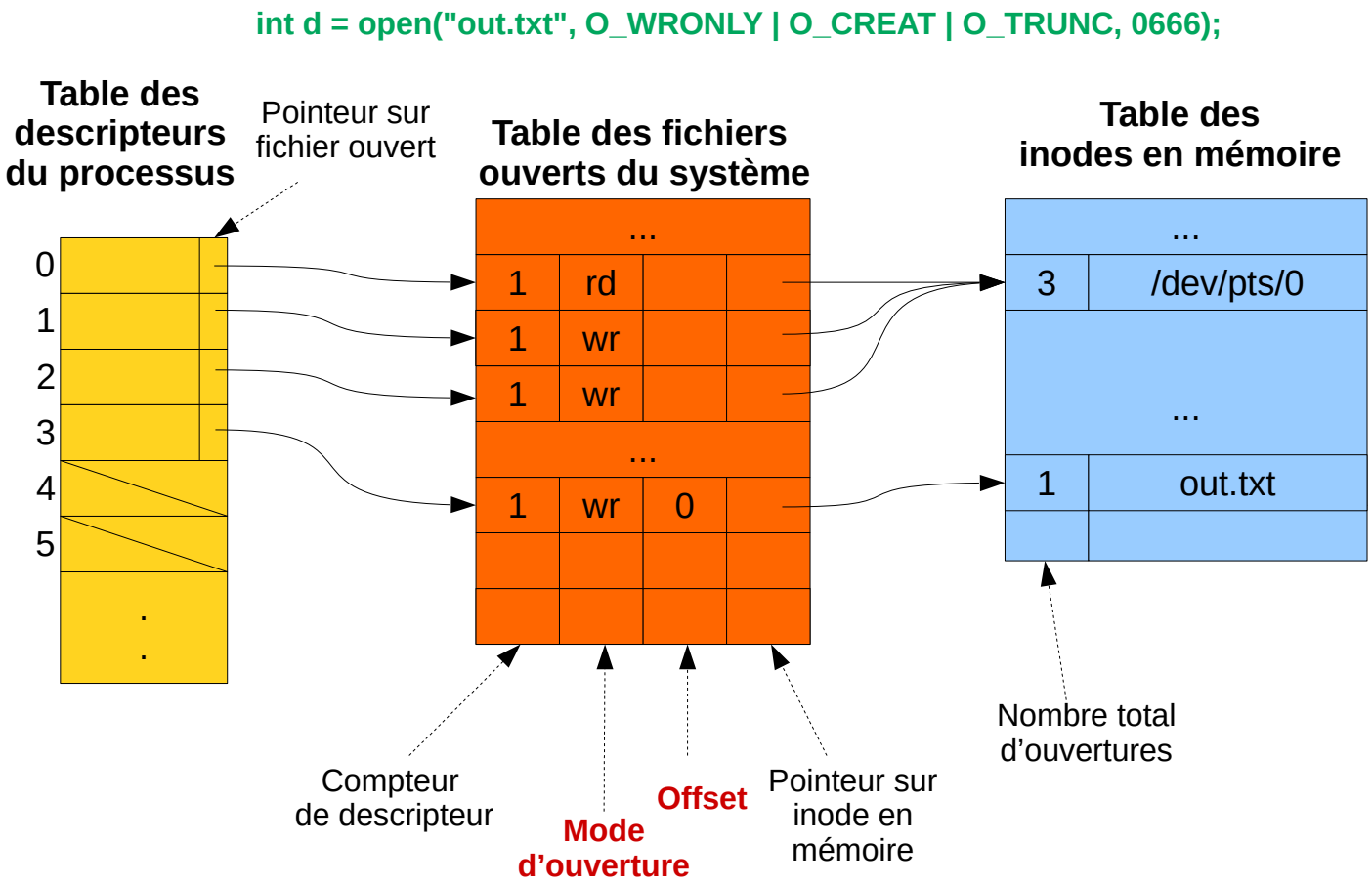
Avant appel de open() :



Comment faire une redirection avec dup2(2) ? ⋆⋆

Exemple (ex-dup2.c : tables du système - 2/4)

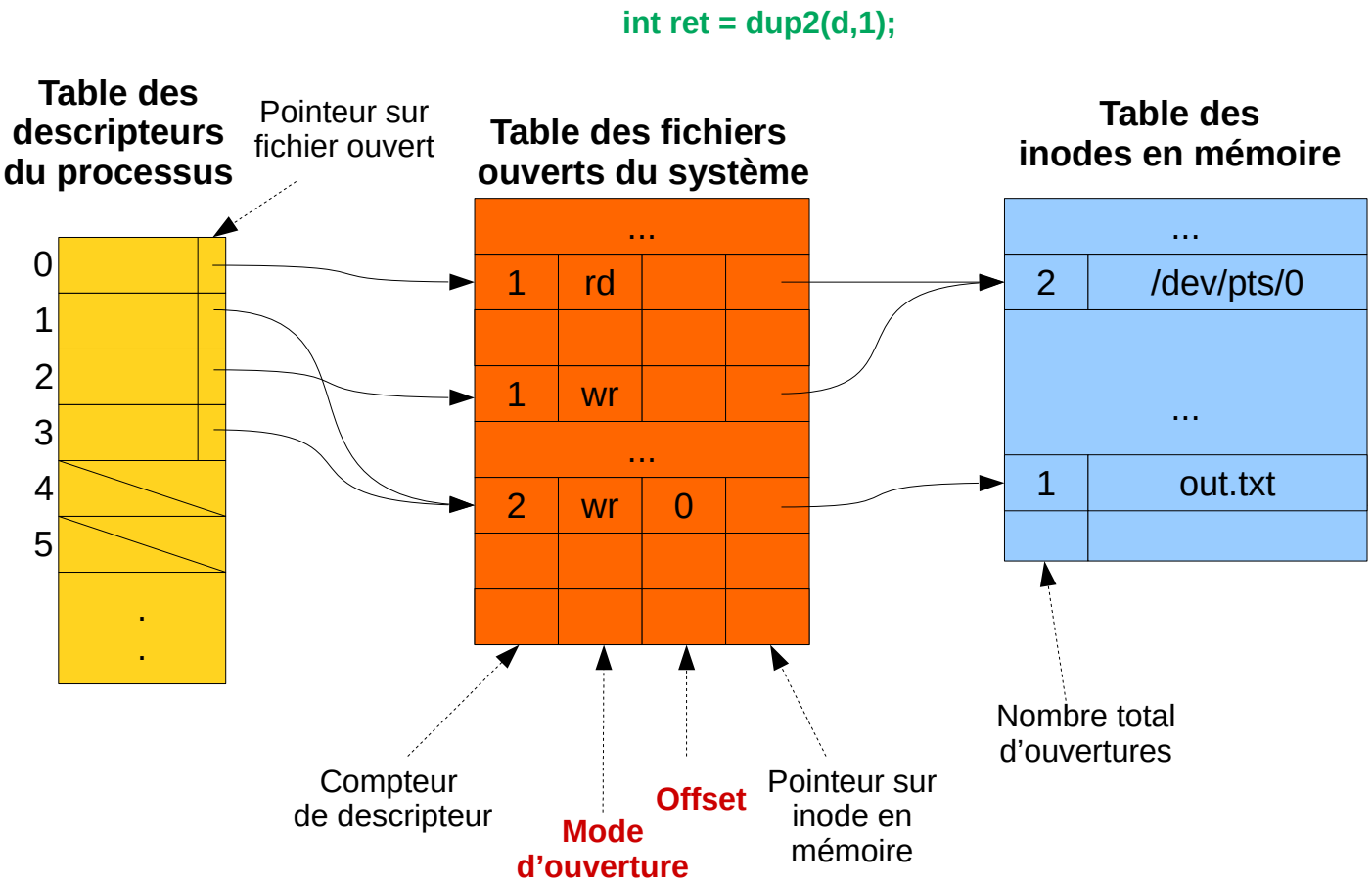
Après appel de open(), avant appel de dup2() :



Comment faire une redirection avec dup2(2) ? ⋆⋆

Exemple (ex-dup2.c : tables du système - 3/4)

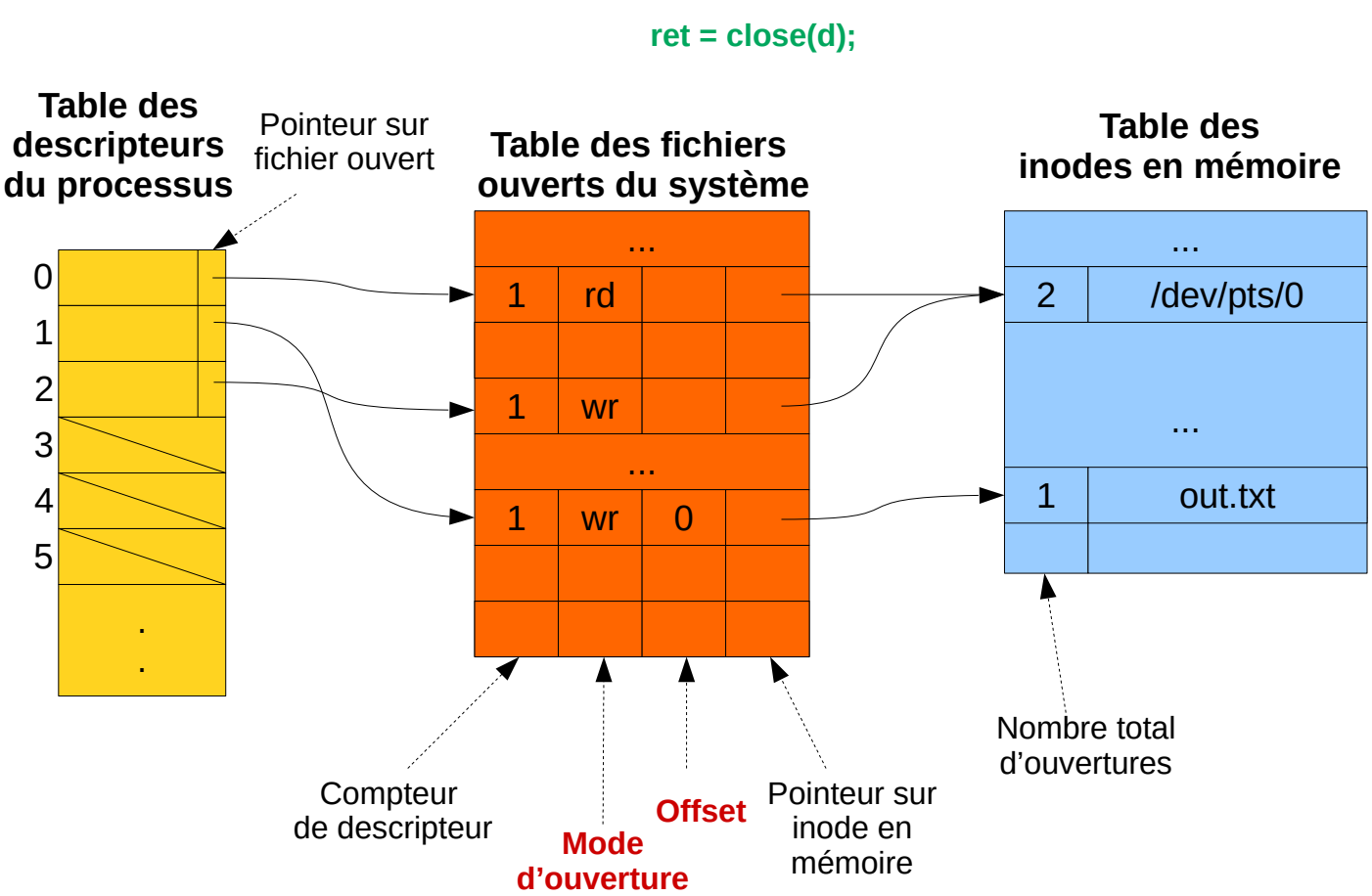
Après appel de dup2(), avant appel de close() :



Comment faire une redirection avec dup2(2) ? ⋆⋆

Exemple (ex-dup2.c : tables du système - 4/4)

Après appel de close() :



Plan

Tubes anonyme

Généralités

★★

Généralités

- Les tubes fournissent un **canal de communication interprocessus unidirectionnel**. Un tube a une entrée et une sortie. Les données écrites à l'entrée du tube peuvent être lues à sa sortie.
- La gestion des tubes est assurée en mode **FIFO**. Le premier octet écrit en entrée du tube sera le premier octet lu en sortie du tube.
- Le canal de communication fourni par un tube est un flot d'octets : il n'y a pas de notion de limite entre messages.
- Un tube a une capacité limitée (par défaut de 65536 octets sous Linux) $\Rightarrow$ un tube peut être **plein**.
- Les tubes anonymes ne permettent une communication qu'entre processus ayant un **lien de parenté**.

RTFM : `pipe(7)`

Créer un tube anomyme

★★

`pipe(2)`

`pipe(2)`

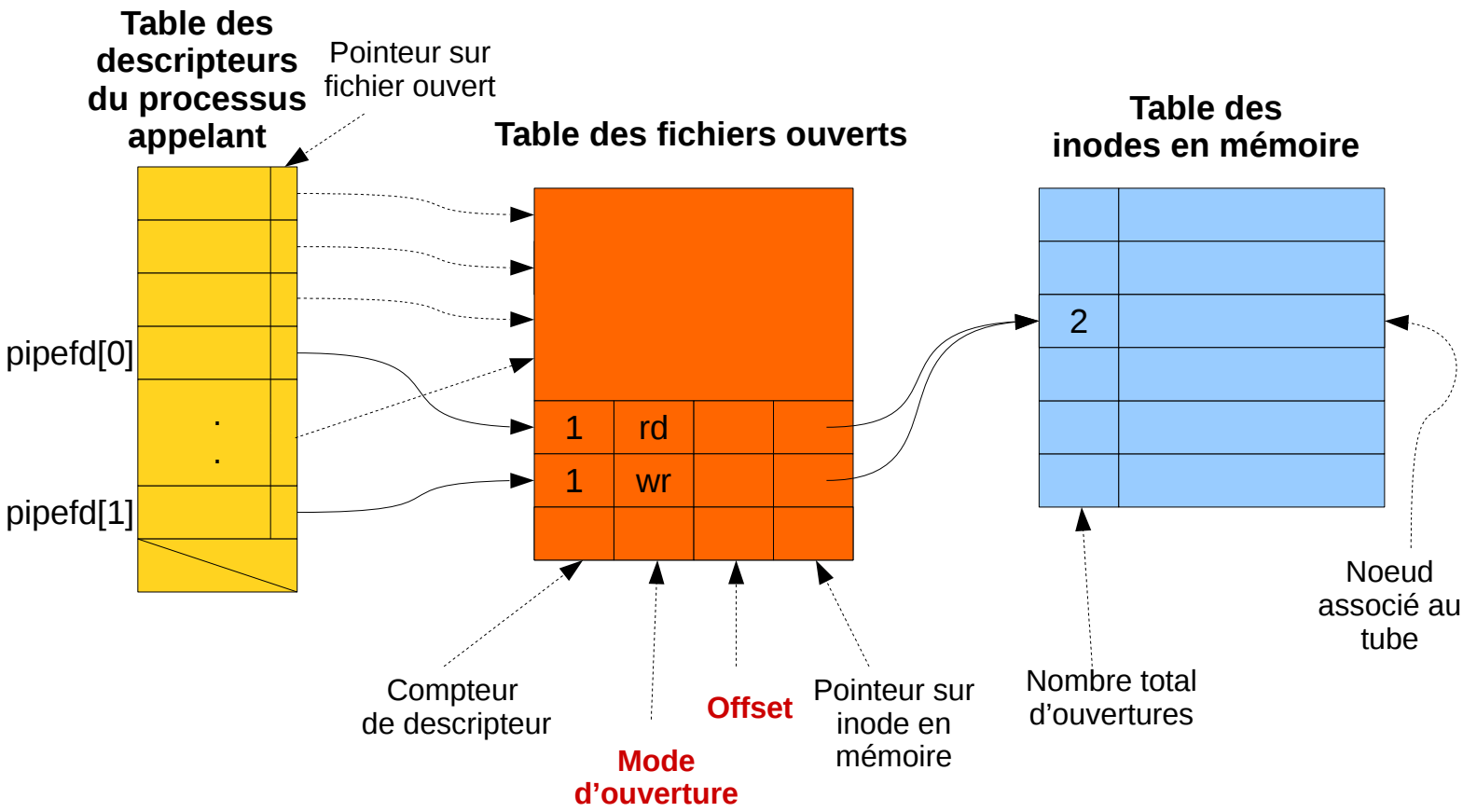
```
int pipe(int pipefd[2]);
```

Demande la création d'un tube. En cas de réussite, il y a allocation d'un noeud associé au tube, de 2 entrées dans la table des fichiers ouverts (une en lecture et une en écriture) et pour, chacune de ces entrées, d'un descripteur dans la table des descripteurs du processus appelant.
Le tableau *pipefd* reçoit les deux descripteurs de fichier faisant référence aux extrémités du tube :

- `pipefd[0]` fait référence à l'extrêmité en lecture (sortie du tube)
- `pipefd[1]` fait référence à l'extrêmité en écriture (entrée du tube)

Créer un tube anomyme avec `pipe(2)` ⋆⋆

Effet sur les tables du système



Nombre de lecteurs et d'écrivains dans un tube anomyme⋆⋆

Nombre de lecteurs

C'est le nombre de descripteurs de fichiers pointant sur l'entrée en lecture sur le tube dans la table des fichiers ouverts. Si ce nombre devient nul, l'entrée en lecture sur le tube est supprimée de la table des fichiers ouverts, et une tentative d'écriture provoquera l'envoi du signal SIGPIPE au processus appelant.

Nombre d'écrivains

C'est le nombre de descripteurs de fichiers pointant sur l'entrée en écriture sur le tube dans la table des fichiers ouverts. Si ce nombre devient nul, l'entrée en écriture sur le tube est supprimée de la table des fichiers ouverts. La nullité de ce nombre détermine le comportement de la fonction `read(2)` lorsque le tube est vide et permet de définir la notion de **fin de fichier** sur un tube.

Rq : un même processus peut disposer de plusieurs descripteurs de fichiers en lecture et/ou écriture sur un même tube

Lecture dans un tube avec `read(2)` ⋆⋆

Lecture dans un tube avec `read(2)`

Quand un processus lit dans un tube, l'appel de
`read(pipefd[0], buf, BUF_SIZE);`
correspond à une demande de lecture d'au plus `BUF_SIZE` octets :

- si le tube n'est pas vide $\Leftrightarrow$ il contient NB octets (avec NB $>$ 0)
  $\Rightarrow$ lecture de `min(NB,BUF_SIZE)` octets
- sinon
  - si il n'y a plus de processus écrivain, alors la **fin de fichier** est atteinte (le tube est vide et le restera)
    $\Rightarrow$ aucun octet n'est lu et `read(2)` **renvoie 0**.
  - sinon, par défaut la lecture est bloquante $\Rightarrow$ le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide, ou qu'il n'y ait plus d'écrivain

Lecture / écriture dans un tube

★★

Remarques

Situations de blocage

Le fait qu'une demande de lecture dans un tube vide soit par défaut bloquante, s'il reste des écrivains, peut conduire à des situations d'autoblocage d'un processus ou d'interblocage de plusieurs processus.
⇒ pour éviter de telles situations, **ne conserver que les descripteurs effectivement utilisés, fermer systématiquement tous les autres**

La constante `PIPE_BUF`

POSIX indique que les écritures de `n` octets, avec `n` $\leq$ `PIPE_BUF`, doivent être atomiques : les `n` octets sont alors tous écrits dans le tube de façon contiguë, ou aucun ne l'est (en cas de tube plein). Les écritures de plus que `PIPE_BUF` peuvent ne pas être atomiques : le noyau peut entrelacer les données avec des données écrites par d'autres processus. Sous Linux, `PIPE_BUF` vaut 4096 octets.
⇒ Conseil : ne jamais chercher à écrire dans un tube plus de `PIPE_BUF` octets par appel de `write(2)`

Ecriture dans un tube avec `write(2)` ⋆⋆

Ecriture dans un tube avec `write(2)`

Par défaut, l'écriture peut être bloquante.
Quand un processus écrit dans un tube, l'appel de
`write(pipefd[1], buf, n);`
correspond à une demande d'écriture de `n` octets.
On suppose : `n` $\leq$ `PIPE_BUF`

- si le nombre de lecteurs dans le tube est nul, le signal SIGPIPE est envoyé au processus appelant, ce qui par défaut le termine. Dans le cas où le signal SIGPIPE est capté, si il y a retour du gestionnaire de signal, l'appel de `write(2)` renvoie `-1` et errno prend comme valeur `EPIPE`.
- sinon, comme l'écriture est atomique, le retour de `write(2)` n'a lieu que lorsque les `n` octets ont été écrits. Le processus est donc susceptible de passer dans l'état endormi en attendant qu'il y ait suffisamment de place dans le tube.

Exemple d'utilisation d'un tube ⋆⋆

Exemple

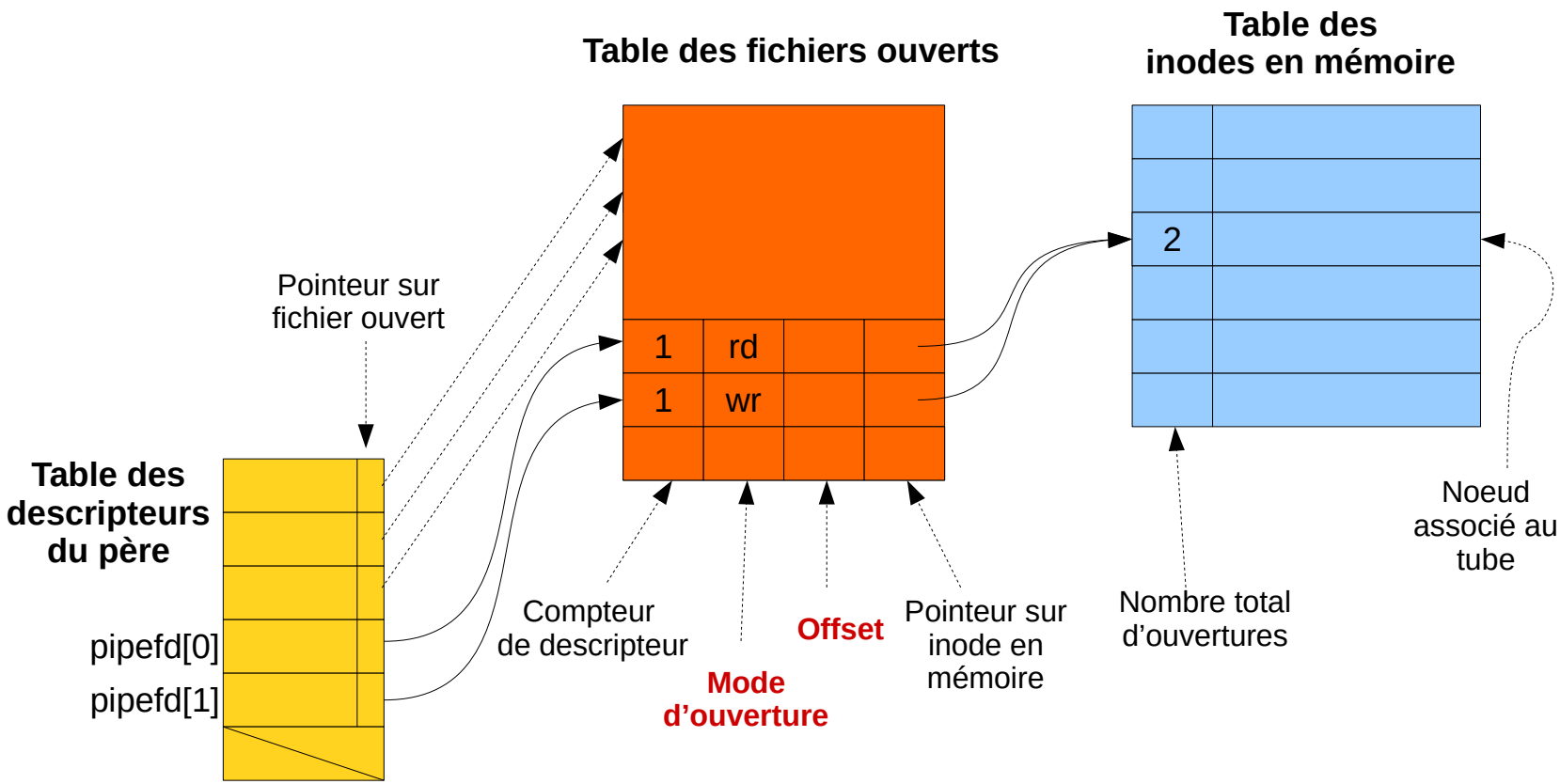
```
int pipefd[2];
char buf;

pipe(pipefd);
pid_t pid = fork();
if (pid == 0) {
  close(pipefd[1]);
  while (read(pipefd[0], &buf, 1) > 0) {
    write(STDOUT_FILENO, &buf, 1);
  }
  close(pipefd[0]);
  exit(0);
}
const char *str = "Hello World!\n";
close(pipefd[0]);
write(pipefd[1], str, strlen(str));
close(pipefd[1]);
wait(NULL);
```

Exemple d'utilisation d'un tube (1/3)

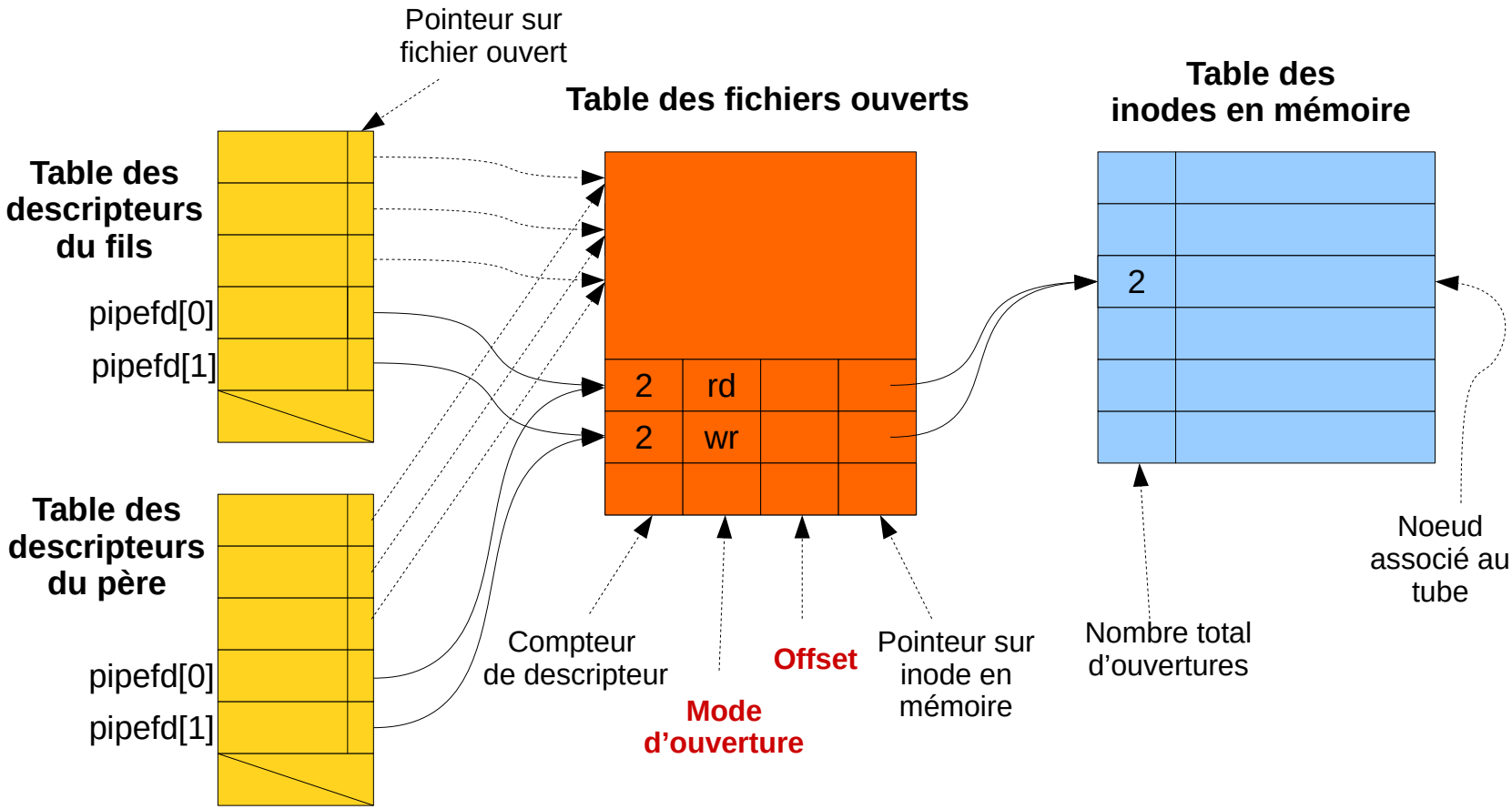
★★

Tables du système avant l'appel de `fork(2)`



Exemple d'utilisation d'un tube (2/3)

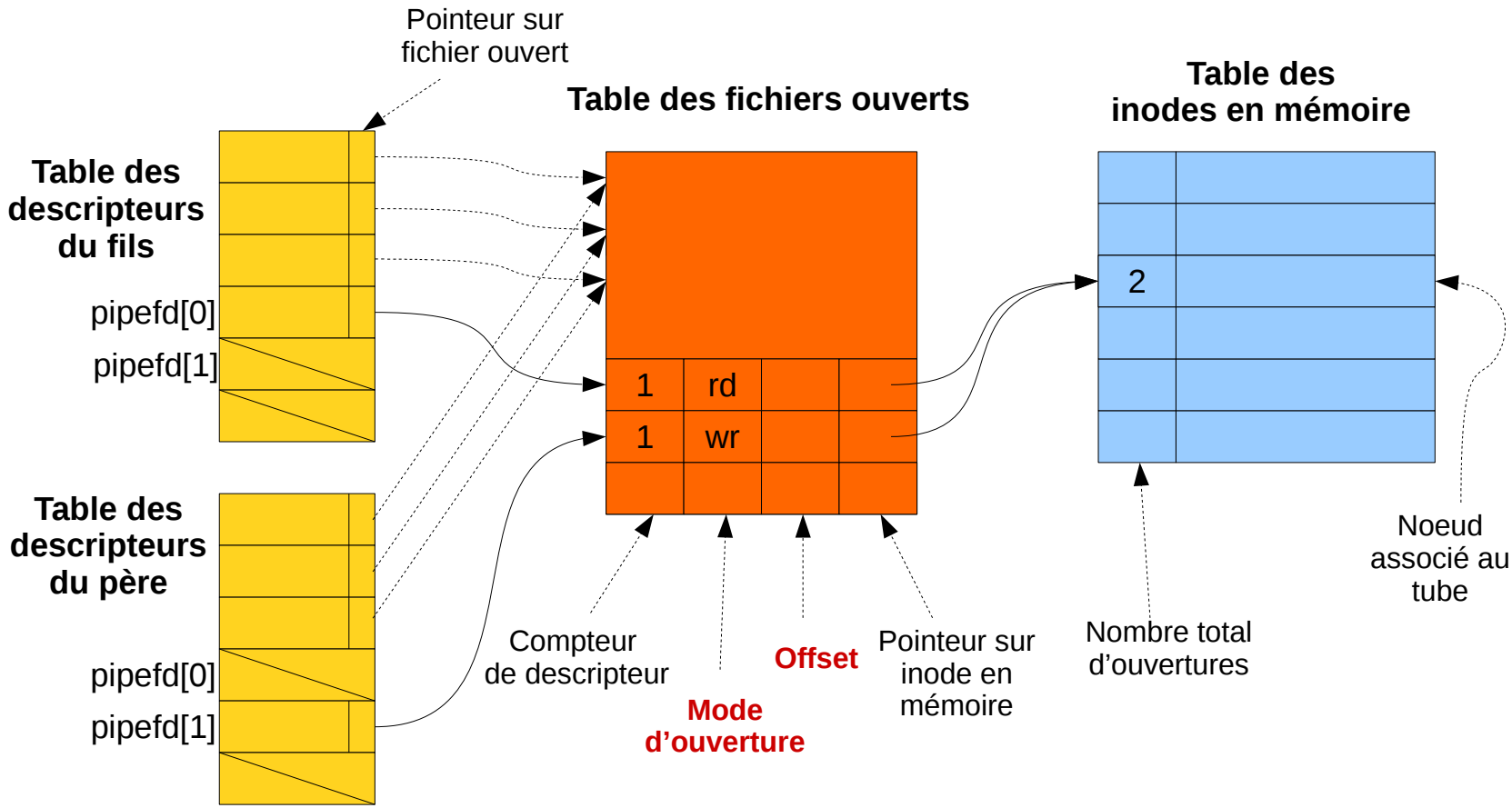
Tables du système après l'appel de `fork(2)`



Exemple d'utilisation d'un tube (3/3)

★★

Tables du système lors de la transmission des données



Plan

Généralités

★★★

Généralités

Linux prend en charge à la fois les signaux POSIX standards et les signaux POSIX temps-réel. Seule la prise en charge des signaux standards est présentée.

- Chaque signal standard (sauf SIGUSR1 et SIGUSR2) est associé à un événement. Exemple : le signal SIGSEGV est associé à l'événement "Référence mémoire invalide"
- A chaque signal correspond une **disposition** (ou **action**) courante qui détermine le comportement du processus lorsqu'il reçoit ce signal. Chaque signal a une disposition par défaut (`Term`, `Ign`, `Core`, `Stop` ou `Cont`)(cf. diapo 338). Un processus peut changer la disposition d'un signal avec `sigaction(2)` (sauf pour SIGKILL et SIGSTOP).
- `sigaction(2)` permet la mise en place, pour un signal, d'un **gestionnaire de signal** (ou **handler**), c'est à dire d'une fonction définie par le programme, qui est invoquée automatiquement lorsque le signal est délivré (ou distribué). On dit alors que le signal est **capté**, **capturé**, **intercepté** ou **attrapé**.
- Les gestionnaires de signaux sont exécutés de façon **asynchone** : ils peuvent interrompre l'exécution du programme à n'importe quel moment.

RTFM : `signal(7)`

Envoyer un signal

★★

`kill(2)` et `raise(3)`

`kill(2)`

```
int kill(pid_t pid, int sig);
```

Envoie le signal *sig* au(x) processus indiqués par *pid*.
Destinataire(s) du signal *sig* en fonction de *pid* :

- *pid* > 0 : le processus d'identifiant *pid*
- *pid* $= 0$: tous les processus appartenant au même groupe que le processus appelant
- *pid* $=$ *-1* : tous les processus du système sauf le processus `init`
- *pid* $<$ *-1* : tous les processus du groupe *-pid*

Remarques sur le paramètre *sig* :

- il faut systématiquement utiliser les **noms symboliques** des signaux
- si *sig* $= 0$, pas de signal envoyé mais test d'existence d'un processus

`raise(3)`

```
int raise(int sig);
```

Envoie le signal *sig* au processus appelant

Dissociation envoi / délivrance d'un signal ★★★

Définition (Délivrance)

La **délivrance** (ou **distribution**) d'un signal correspond à sa prise en compte effective par le processus destinataire (c'est à dire à l'exécution de l'action courante associée)

Définition (Signal pendant)

Un signal **pendant** (ou en attente) est un signal qui a été envoyé à un processus mais qui n'a pas été encore délivré. Un indicateur de signal pendant occupant **un seul bit** est associé à chaque signal standard.

Dissociation envoi / délivrance d'un signal

1. envoi d'un signal $\Rightarrow$ l'indicateur de signal pendant correspondant passe à 1.
2. délivrance du signal $\Rightarrow$ l'indicateur de signal pendant est basculé à 0, puis l'action courante du signal (handler dans le cas d'un signal capté) est exécutée

$\Rightarrow$ Quand l'indicateur de signal pendant d'un signal S est basculé à 1, les exemplaires suivants de ce même signal S, qui sont émis **AVANT** la délivrance du signal, ne sont pas pris en compte.

Masquage d'un signal ⋆

Définition (Masquage d'un signal)

Un signal peut être **masqué** (ou bloqué), ce qui signifie qu'il ne sera pas délivré au processus destinataire avant d'être débloqué.

⇒ Si plusieurs exemplaires d'un signal standard arrivent alors qu'il est masqué, un seul exemplaire sera mémorisé.
→ Le masquage d'un signal permet de **différer** sa délivrance.
→ Les signaux SIGKILL et SIGSTOP ne peuvent pas être masqués.

Définition (Masque de signaux d'un processus)

Le masque de signaux d'un processus est égal à l'ensemble des signaux qui sont masqués dans ce processus.

Informations associées à un signal standard ⋆⋆⋆

Informations associées à un signal standard

- **une action courante**
- un indicateur de signal pendant (**mémorisé sur un seul bit**)
- un indicateur de signal masqué (mémorisé lui aussi sur un seul bit)

→ en tant que programmeur, vous pouvez modifier :

- l'action courante associée à un signal (sauf pour SIGKILL et SIGSTOP),
- et/ou le masque de signaux d'un processus.

Manipulation des ensembles de signaux ⋆

La constante macrodéfinie SIGRTMAX

Elle correspond au plus grand numéro de signal valide du système. Un signal valide est identifié par un entier > 0 et ≤ SIGRTMAX.

Le type sigset_t

Il correspond à un **ensemble** de signaux. Les fonctions suivantes permettent la manipulation des variables de ce type :

Fonction	Effet
`int sigemptyset(sigset_t *set);`	`*set = ∅`
`int sigfillset(sigset_t *set);`	`*set = {1, ..., SIGRTMAX}`
`int sigaddset(sigset_t *set, int sig);`	`*set = *set ∪ {sig}`
`int sigdelset(sigset_t *set, int sig);`	`*set = *set - {sig}`
`int sigismember(sigset_t *set, int sig);`	`sig ∈ *set ?`

Toutes les fonctions renvoient -1 en cas d'erreur. La fonction `sigismember` renvoie 1 ou 0 selon que `sig` appartient ou non à l'ensemble `*set`.

RTFM : `SIGSETOPS(3)`

Manipulation des ensembles de signaux ⋆

Exemple (Contruction d'un ensemble de signaux contenant les signaux SIGINT et SIGQUIT)

```
sigset_t ens;
sigemptyset(&ens);
sigaddset(&ens, SIGINT);
sigaddset(&ens, SIGQUIT);
```

→ Pour des raisons de portabilité évidentes, pour désigner un signal, il faut toujours utiliser la constante entière macrodéfinie correspondante (par exemple SIGINT, SIGQUIT, SIGKILL, etc.)
→ après l'exécution de ces instructions, le comportement du processus vis à vis des signaux SIGINT et SIGQUIT n'est pas modifié : on a simplement créé un ensemble de signaux.

Masquage de signaux

`sigprocmask(2)`

★

`sigprocmask(2)`

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Récupère ou change le **masque de signaux** du processus.
Son comportement est dépendant de la valeur de `how` :

Valeur de `how`	Nouveau masque
`SIG_SETMASK`	`*set`
`SIG_BLOCK`	UNION de l'ensemble actuel et de `*set`
`SIG_UNBLOCK`	Ensemble actuel MOINS `*set`

Si `oldset` n'est pas `NULL`, la valeur précédente du masque de signaux est stockée dans `*oldset`. Si `set` est NULL, le masque de signaux n'est pas modifié (`how` est donc ignoré), mais la valeur actuelle du masque de signaux est tout de même renvoyée à l'adresse `oldset` (si elle n'est pas `NULL`).

Liste des signaux pendants masqués

sigpending(2)

sigpending(2)

```
int sigpending(sigset_t *set);
```

Écrit à l'adresse set la liste des signaux pendants qui sont masqués.

sigpending(2) renvoie 0 s'il réussit, et -1 en cas d'erreur. En cas d'erreur, errno est mise à jour.

Signaux masqués et signaux pendants masqués ⋆⋆

Exemple

```c
int main(int argc,char *argv[]) {
  sigset_t ens1;
  /* Construction de l'ensemble ens1 = {SIGINT, SIGQUIT, SIGUSR1} */
  sigemptyset(&ens1);
  sigaddset(&ens1, SIGINT);
  sigaddset(&ens1, SIGQUIT);
  sigaddset(&ens1, SIGUSR1);
  /* Installation du masque ens1 */
  sigprocmask(SIG_SETMASK, &ens1, NULL);

  /* Le processus s'envoie 2 fois le signal SIGINT et une fois SIGUSR1 */
  raise(SIGINT);
  kill(getpid(), SIGINT);
  kill(getpid(), SIGUSR1);
  /* Extraction des signaux pendants masqués */
  sigset_t ens2;
  sigpending(&ens2);
  /* Impression de la liste des signaux pendants masqués */
  fprintf(stderr,"Signaux pendants masqués : ");
  for (int sig = 1; sig <= SIGRTMAX; ++sig){
    if (sigismember(&ens2, sig))
      fprintf(stderr,"%d ", sig);
  }
  fprintf(stderr, "\n");
  sleep(5);
  sigemptyset(&ens1);
  printf("Deblocage de tous les signaux\n");
  sigprocmask(SIG_SETMASK, &ens1, NULL);
  printf("Fin du processus\n"); //ne s'exécute pas car un signal qui termine le processus est délivré
  return EXIT_SUCCESS;
}
```

Examiner et modifier l'action associée à un signal ⋆⋆⋆

La structure sigaction et l'appel système sigaction(2) (1/2)

La structure sigaction

Le comportement général que doit avoir un processus lors de la délivrance d'un *signal* est décrit en utilisant la structure `sigaction` :

```
struct sigaction {
void (*sa_handler)(int); → SIG_DFL, SIG_IGN, ou pointeur sur handler spécifique
sigset_t sa_mask; → signaux supplémentaires à bloquer (temporairement) lors de
                    la délivrance
int sa_flags; → options
};
```

- `sa_handler` indique l'action affectée au signal considéré, et peut être :
  - SIG_DFL pour l'action par défaut (diffère d'un signal à l'autre)
  - SIG_IGN pour ignorer le signal
  - ou un pointeur sur un handler (i.e. une fonction définie par le programme). On dit alors que le signal est capté, capturé, intercepté ou attrapé.
- `sa_mask` correspond à une liste de signaux qui seront temporairement ajoutés au masque de signaux du processus pendant l'exécution du handler (N.B. : lors de l'exécution d'un handler, le signal en cours de délivrance est par défaut temporairement ajouté au masque de signaux)

Examiner et modifier l'action associée à un signal ⋆⋆⋆

La structure sigaction et l'appel système sigaction(2) (2/2)

La structure sigaction (suite)

- sa_flags spécifie un ensemble d'attributs qui modifient le comportement qu'aura le processus lors de la délivrance d'un signal. Il est formé par un OU binaire « | » entre diverses options :
  - SA_RESTART : certains appels systèmes interrompus par un signal capté sont repris au lieu de renvoyer -1 avec errno = EINTR

L'appel système sigaction(2)

```
int sigaction(int signum,
const struct sigaction *act,struct sigaction *oldact);
```

Permet de modifier l'action qui sera effectuée par un processus à la réception du signal signum. Si act n'est pas NULL, la nouvelle action pour le signal signum est définie par *act. Si oldact n'est pas NULL, l'ancienne action est sauvegardée à l'adresse oldact.

Mise en place d'un gestionnaire de signal ⋆⋆

Exemple

Exemple (sig1.c)

```
void handSIGINT (int sig){
  write(1,"\t appel de handSIGINT\n",22);
}

int main(int argc, char *argv[]){
  // Installation d'un handler pour SIGINT
  struct sigaction action;
  sigemptyset(&action.sa_mask);
  action.sa_handler = handSIGINT;
  action.sa_flags = 0;
  sigaction(SIGINT, &action, NULL);
  while(1){
    sleep(1);
    printf("Je travaille\n");
  }
  exit(0);
}
```

Exemple (Exécution)

```
$ ./sig1
Je travaille
^C  appel de handSIGINT
Je travaille
Je travaille
^C  appel de handSIGINT
Je travaille
Je travaille
^\Quit (core dumped)
$
```

→ il n'y a pas d'appel direct du handler, il est appelé **automatiquement** lorsque le signal est délivré

→ le paramètre `sig` reçoit le numéro du signal délivré

Mémorisation d'informations dans un handler ⋆⋆

Dans le programme précédent, comment compter le nombre de signaux SIGINT reçus ?

Mémorisation d'informations dans un handler

Pour mémoriser des informations dans un handler, il est possible d'utiliser des variables globales. Mais quand il n'est pas nécessaire d'accéder à ces informations dans le programme principal, il est préférable d'utiliser des variables locales statiques.

Classe d'allocation statique ⋆⋆

Classe d'allocation statique

Les variables globales (variables déclarées en dehors de toute fonction) ou locales statiques (variables locales d'une fonction déclarées avec le mot réservé static) sont dites de classe d'allocation statique. Elles sont placées dans le segment des données et occupent un emplacement parfaitement définie lors de la compilation. Elles sont créées une fois pour toutes au début de l'exécution du programme et existent pendant toute sa durée d'exécution.

Visibilité (ou portée) des variables de classe statique

- Les variables globales sont visibles (c'est à dire accessibles) dans toute la partie du fichier source qui suit leur déclaration.
- Les variables locales statiques ne sont visibles qu'à l'intérieur de la fonction où elles sont déclarées.

Variables locales statiques dans un handler

★★

Exemple

Exemple (sig2.c)

```
void handSIGINT (int sig){
  static int nb = 0; char tab[80];
  sprintf(tab, "\t nb = %d\n", ++nb);
  write(1, tab, strlen(tab));
}
int main(int argc, char *argv[]){
  struct sigaction action;
  sigemptyset(&action.sa_mask);
  action.sa_handler = handSIGINT;
  action.sa_flags = 0;
  sigaction(SIGINT, &action, NULL);
  while(1){
    sleep(1);
    printf("Je travaille\n");
  }
  exit(0);
}
```

Exemple (Exécution)

```
$ ./sig2
Je travaille
^C  nb = 1
Je travaille
Je travaille
^C  nb = 2
Je travaille
Je travaille
^\Quit (core dumped)
$
```

Dissociation envoi / délivrance d'un signal ⋆⋆

Exemple (1/2)

Exemple (handler)

```
void handler(int sig) {
  static int nb_sigusr1 = 0;
  char s[80];
  switch(sig) {
  case SIGUSR1:
    ++nb_sigusr1;
    return;
  case SIGINT:
    sprintf(s,"Nb reçus : %d\n",
            nb_sigusr1);
    write(1, s, strlen(s));
    _exit(0);} }
```

```
$ ./pere_fils
fin du fils
^CNb reçus : 6724
```

Exemple (Fonction main())

```
int main(int argc, char *argv[]) {
  struct sigaction action;
  sigemptyset(&action.sa_mask);
  action.sa_handler = handler;
  action.sa_flags = 0;
  sigaction(SIGUSR1, &action, NULL);
  sigaction(SIGINT, &action, NULL);
  if (fork() == 0) {
    for (size_t i = 0; i < 10000; ++i){
      kill(getppid(), SIGUSR1);
    }
    printf("fin du fils\n");
    exit(0);
  }
  while(1){ sleep(1);}
  return 0;
}
```

Dissociation envoi / délivrance d'un signal ⋆⋆

Exemple (2/2)

Exemple (Explications)

Un fils envoie une rafale d'exemplaires (10000) du signal `SIGUSR1` à son père.
Le père est dans une boucle infinie et il compte dans le handler du signal le nombre d'exemplaires du signal SIGUSR1 qui lui sont délivrés. Lorsqu'il est interrompu (à la délivrance du signal `SIGINT`), ce nombre est affiché et le processus se termine.
Tous les exemplaires du signal `SIGUSR1` envoyés ne sont pas délivrés : certains signaux envoyés sont "perdus". Ces signaux perdus correspondent à des envois subséquents de signal alors que l'indicateur de signal pendant était déjà à 1, **avant délivrance du signal**. Cela peut se produire dans 2 situations :

- le processus fils envoie plusieurs signaux `SIGUSR1` pendant un intervalle de temps où le père n'est pas actif (i.e. ne dispose pas du processeur pour s'exécuter)
- le processus fils envoie plusieurs signaux `SIGUSR1` pendant que le père est en train d'exécuter le handler (`SIGUSR1` est masqué lors de cette exécution)

Remarque : il est important d'installer le handler du signal `SIGUSR1` avant l'appel de `fork(2)` car l'action par défaut associée au signal `SIGUSR1` est égale à `Term` : il ne faut en aucun cas que le fils puisse envoyer `SIGUSR1` au père avant que ce dernier ait installé le handler pour `SIGUSR1`.

Attendre un signal

pause(2)

★★

pause(2)

```
int pause(void);
```

- force le processus appelant à s'endormir jusqu'à ce qu'un signal soit délivré, qui soit termine le processus, soit entraîne l'appel d'un gestionnaire de signal.
- pause(2) retourne seulement si un signal a été intercepté et si il y a un retour du handler. Dans ce cas, pause(2) renvoie -1 et errno est positionné à la valeur EINTR (qui signifie "Appel système interrompu").

Comportement d'ignorance

★★

(Installation) et délivrance

- Le comportement d'ignorance peut être l'action par défaut associée à un signal, ou peut être installé avec `sigaction(2)` en affectant `SIG_IGN` dans le champ `sa_handler`.
- La délivrance d'un signal ignoré a pour seul effet de mettre à 0 l'indicateur de signal pendant. Elle ne peut pas interrompre un appel système (contrairement à un handler vide).

Programmer une alarme

`alarm(2)`

★★

`alarm(2)`

```
unsigned int alarm(unsigned int nb_sec);
```

- programme une alarme, c'est à dire demande l'envoi d'un signal `SIGALRM` au processus appelant dans `nb_sec` secondes.
- Si `nb_sec` vaut zéro, toute alarme en attente est annulée.
- Dans tous les cas, l'appel `alarm(2)` annule l'éventuelle programmation précédente, et retourne immédiatement.

Cas typique d'utilisation

Le programme doit rendre une réponse en un temps donné :

- on programme une alarme et on lance le calcul.
- et au moment où le signal est reçu, on rend la meilleure réponse obtenue jusqu'à présent.

Exemple : jeu de dames, jeu d'échecs, etc.

Programmation d'une temporisation ⋆⋆

Exemple

Exemple

```
void handSigalrm(int sig) {
  write(2, "\nTrop tard !!!\n", 15);
  _exit(1);
}
int main(int argc,char *argv[]) {
  char reponse[80];
  struct sigaction action;
  sigemptyset(&action.sa_mask);  //vide l'ensemble sa_mask
  action.sa_flags = 0;       // pas d'option
  action.sa_handler = handSigalrm;
  sigaction(SIGALRM, &action, NULL); //installation du handler
  printf("Question ? ");
  alarm(5);   /* demande d'envoi de SIGALRM dans 5 secondes */
  fgets(reponse, 80, stdin);
  alarm(0);   /* si la réponse a été lue, désarmer le timer */
  printf("Réponse lue : %s\n", reponse);
  return EXIT_SUCCESS;
}
```

Gestion des signaux

Comportement en cas de duplication / recouvrement

Comportement en cas de duplication / recouvrement

- Un fils créé par fork(2) hérite d'une copie des dispositions de signaux de son père. Lors d'un execve(2), les dispositions des signaux captés sont remises aux valeurs par défaut ; les dispositions des signaux ignorés ne sont pas modifiées.
- Un processus fils créé avec fork(2) hérite d'une copie du masque de signaux de son père ; le masque de signaux est conservé au travers d'un execve(2).
- Un fils créé avec fork(2) démarre avec un ensemble de signaux pendants vide ; l'ensemble de signaux pendants est conservé au travers d'un execve(2).

Divers problèmes liés aux gestionnaires de signaux ⋆⋆

Partage de données entre handler et programme principal ?

Pour partager des données entre un gestionnaire de signal et le programme principal, il faut utiliser des variables globales. Comme les gestionnaires sont exécutés de façon **asynchrone**, si les données partagées sont des **strutures de données** comme des listes : attention **danger** !
En effet, si le programmeur ne prend pas de précaution, le handler risque d'être appelé à un instant où la structure de données est "incohérente" (la mise à jour de ses champs n'étant pas terminée), et donc, si ce handler utilise cette même structure de données, c'est le plantage assuré.
Le même problème se pose pour les fonctions système ou de la bibliothèque standard.

Exemple (Fonctions non sûres)

Les fonctions d'entrée/sortie, de la bibliothèque standard, utilisant les flux, ne sont "pas sûres" car elles utilisent un buffer de données de classe d'allocation mémoire statique associé à des compteurs et des indexes mémorisant la quantité de données et la position courante dans le buffer.
Si le programme principal est interrompu au milieu d'un appel de `printf(3)`, et que le handler appelle lui aussi `printf(3)`, le comportement du programme est imprévisible ($\Rightarrow$ risque de plantage).

Divers problèmes liés aux gestionnaires de signaux ⋆⋆

Fonctions sûres pour signaux asynchrones

- POSIX possède la notion de fonctions «**async-signal-safe**».
- Si un signal interrompt l'exécution d'une fonction non sûre, et que le gestionnaire appelle une fonction non sûre, alors le comportement du programme n'est pas défini.
  ⇒ une solution : n'appeler, dans les handlers, que des fonctions sûres (i.e. `async-signal-safe`)
- Liste des fonctions «**async-signal-safe**» : `man 7 signal-safety`

Partage de données entre handler et programme principal ?

Solutions :

- utiliser le masquage de signaux pour faire en sorte que le programme principal et les handlers de signaux ne puissent pas être interrompus quand ils manipulent les données partagées
- déclarer les structures de données "complexes" en tant que variables locales de la fonction main(), utiliser de simples variables globales "drapeaux" (booléens ou entiers) pour mémoriser la réception des signaux, et surveiller les modifications de ces drapeaux dans la fonction main() afin d'y réaliser les traitements nécessaires

Divers problèmes liés aux gestionnaires de signaux ⋆⋆

Problème de consistance mémoire ?

- Problème : quand le compilateur fait des optimisations (-O[n]), les valeurs lues pour une même variable globale peuvent être différentes suivant que l'on se trouve dans la fonction main() ou dans un handler de signal
- Solution possible : utiliser le mot clé `volatile` qui demande au compilateur de ne pas optimiser les accès à cette variable

Exemple (sans problème de consistance mémoire)

```
volatile int usr1_receive = 0;
void handSIGUSR1(int sig) {
  usr1_receive = 1;
}
int main(){
  ...
  if (usr1_receive == 1){...}
}
```

Plan

Qu'est-ce qu'un daemon ?

Définition (daemon)

Un daemon (*Disk And Execution MONitor*) est un programme qui tourne en tâche de fond (quasiment) en permanence. Le nom d'un daemon finit généralement par la lettre `d`.

Quelques exemples

- `httpd` : le serveur Web Apache
- `sshd` : le serveur SSH OpenSSH
- `crond` : le planificateur de tâche Cron

Comment devenir un daemon ?

Daemon

Un daemon a généralement comme PPID le processus `init`, soit directement au démarrage de la machine, soit parce qu'il est devenu orphelin. De plus, il n'est pas attaché à un terminal. Un daemon effectue également les actions suivantes :

- Changer le répertoire courant à `/` via `chdir(2)`
- Changer le umask à 0 via `umask(2)`
- Fermer tous les descripteurs de fichiers des entrée/sorties standard

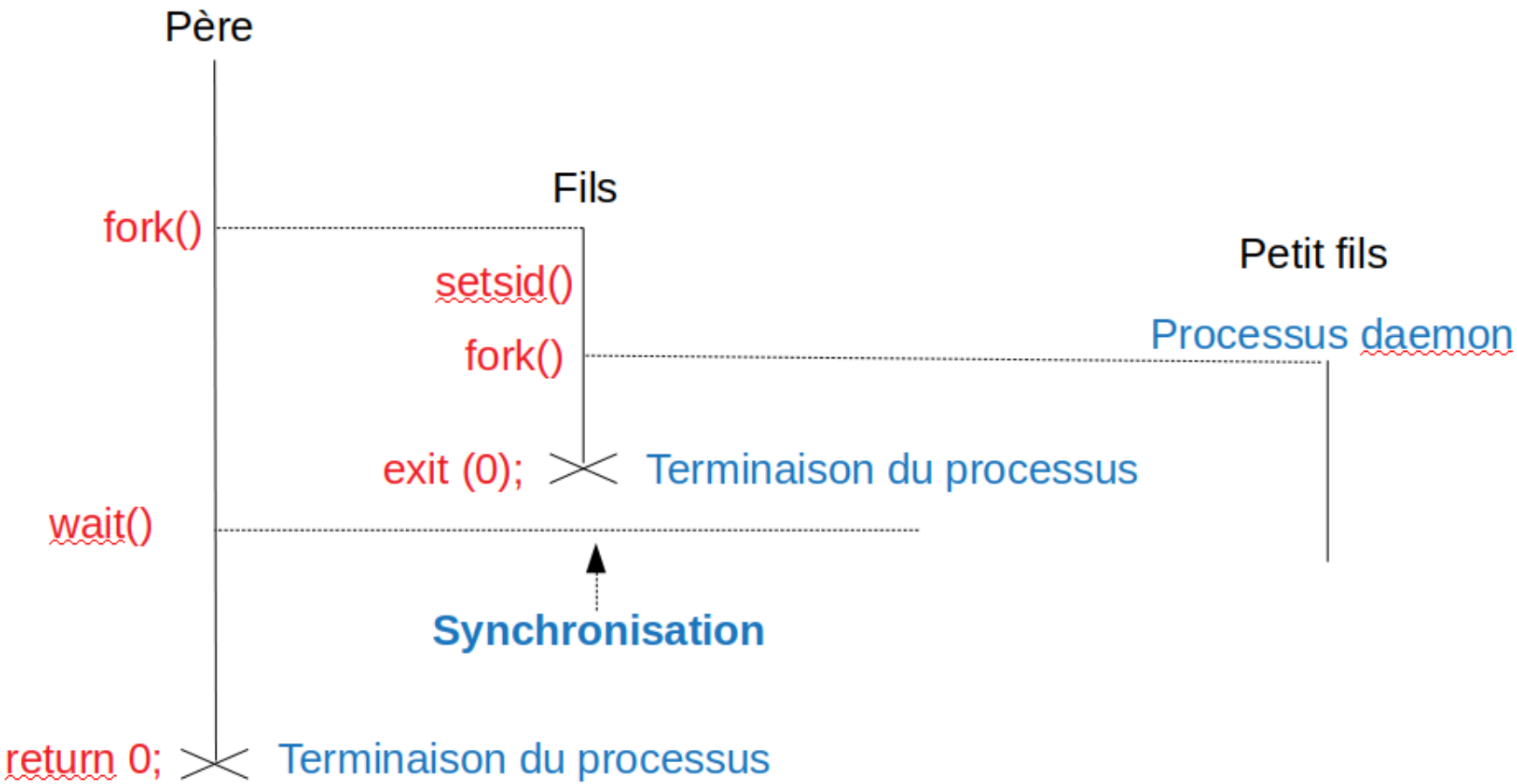
Lancement d'un daemon (1/3) ⋆

Lancement d'un daemon

- Le processus principal crée un nouveau processus et attend son fils
- Le fils devient leader de session via `setsid(2)`
- Le fils crée un nouveau processus et se termine via `exit(3)`, rendant le petit-fils orphelin
- Le petit-fils n'est pas leader de session, ce qui l'empêche d'être rattaché à un terminal
- Le processus principal se termine, laissant le petit-fils être le daemon

RTFM : credentials(7)

Lancement d'un daemon (2/3)



Lancement d'un daemon (3/3) ⋆

Explications

- Le shell qui lance le processus principal (ou processus père) crée un nouveau groupe de processus, dont le leader est le processus principal. La création d'une nouvelle session passe par la création d'un nouveau groupe de processus ; la nouvelle session et le nouveau groupe prennent l'identifiant du processus appelant (i.e. le processus appelant en est le leader). Or pour créer un nouveau groupe (et donc une nouvelle session), un processus ne doit pas être déjà leader de son groupe : il est indispensable que cet identifiant ne soit pas encore attribué à un groupe, ou à une session, qui pourraient éventuellement contenir d'autres processus.
- à partir du moment où le fils crée une nouvelle session, tous ses descendants (qui ne créent pas eux mêmes une nouvelle session) appartiennent à cette session.
- Tous les processus d'une session partagent (éventuellement) un même terminal de contrôle. Une session sans terminal de contrôle en acquiert un, lorsque le leader de session ouvre un terminal pour la première fois. Ici, le leader de session se termine sans ouvrir de terminal. Donc la session, à laquelle appartient le petit fils, ne pourra plus jamais acquérir un terminal de contrôle.

Questions de la fin

À propos des processus zombie

Comment créer un processus zombie en C ?

Neuvième partie

Programmation système avancée

Plan de ce cours

15 Communication inter-processus
- Introduction à la communication inter-processus
- Tube nommé
- File de messages

Plan

Communication inter-processus ★★★

Définition (Communication inter-processus)

La communication inter-processus (IPC, *Inter-Process Communication*) est l'ensemble des mécanismes qui permettent à des processus concurrents de communiquer. On peut séparer ces mécanismes en deux familles :

- l'échange des données
- la synchronisation

Pourquoi ?

- Vitesse d'exécution
- Modularité
- Séparation des privilèges
- Partage d'information

Mécanismes de communication inter-processus ⋆

Exemples de mécanismes connus

- Fichiers réguliers
- Signaux
- Tubes anonymes

Autres exemples de mécanismes

- Tubes nommés
- Mémoire partagée
- Sémaphores
- Gestion des accès concurrents à un même fichier régulier : verrouillage
- Files de messages
- Sockets
- RPC (*Remote Procedure Call*)

Mécanismes de communication inter-processus ⋆

Exemples de mécanismes connus

- Fichiers réguliers
- Signaux
- Tubes anonymes

Autres exemples de mécanismes

- Tubes nommés
- Mémoire partagée
- Sémaphores
- Gestion des accès concurrents à un même fichier régulier : verrouillage
- Files de messages
- Sockets
- RPC (*Remote Procedure Call*)

API pour les mécanismes IPC

API pour les mécanismes IPC

Il existe deux API pour les IPC :

- L'API SystemV
- L'API POSIX

Cela concerne :

- Mémoire partagée
- Sémaphores
- Files de messages

Nous nous intéresserons à l'API POSIX.

Plan

Qu'est-ce qu'un tube nommé ? ⋆⋆

Définition (Tube)

Un tube est un mécanisme de type FIFO (*First In First Out*) qui permet à deux processus d'échanger des informations de manière unidirectionnelle en mode flot.

Définition (Tube anonyme)

Un tube anonyme est un tube qui n'existe que pendant la durée des processus qui l'utilisent et disparaît ensuite. Il ne permet la communication qu'entre processus ayant un lien de parenté. On le crée en shell avec un `|`.

Définition (Tube nommé)

Un tube nommé est un tube qui a une existence dans le système de fichiers, et qui persiste tant qu'il n'est pas explicitement détruit. Il permet la communication entre processus sans lien de parenté. On le crée en shell avec la commande `mkfifo(1)`.

RTFM : `fifo(7)`

Création et utilisation de tubes nommés ⋆

mkfifo(1)

mkfifo(1)

`mkfifo name`

Crée un tube nommé dont le nom est *name*.

Exemple

```
$ mkfifo foo
$ ls -l foo
prw-r--r-- 1 eric eric 0 18 mars  10:51 foo
$ rm foo
```

Exemple

```
$ mkfifo bar
$ gzip -9 -c < bar > out.gz
--
$ cat file > bar
```

Création et ouverture d'un tube nommé en C ⋆⋆

`mkfifo(3)`

`mkfifo(3)`

```
int mkfifo(const char *pathname, mode_t mode);
```

Crée le tube nommé dont le nom est `pathname` avec les permissions `mode`.

Ouverture d'un tube nommé

- Une fois créé, le tube nommé peut être ouvert avec `open(2)` comme n'importe quel fichier.
- Pour être utilisé, un tube nommé doit être ouvert par deux processus, un en lecture et l'autre en écriture.
- L'ouverture d'un tube nommé **peut être bloquante** :
  - une demande d'ouverture en lecture est bloquante en l'absence d'écrivain sur le tube et de processus bloqué sur une ouverture en écriture
  - une demande d'ouverture en écriture est bloquante si il n'y a aucun lecteur sur le tube et aucun processus bloqué sur une ouverture en lecture

Création et utilisation d'un tube nommé en C ⋆⋆

Exemple

Exemple (mk_read_fifo.c)

```
int main (void)
{
  if (mkfifo("baz", 0644)) {
    perror("mkfifo");
    exit(EXIT_FAILURE);
  }
  ...

  int fd = open("baz", O_RDONLY);
  char buf;
  while (read(fd, &buf, 1) > 0){
      ...
  }
  close(fd);
  unlink("baz");
  return 0;
}
```

Exemple (write_fifo.c)

```
int main (void)
{
  ...
  int fd = open("baz", O_WRONLY);
  ...

  const char *str = "hello world!";
  size_t lo = strlen(str);
  ssize_t nb = write(fd, str, lo);
  close(fd);
  return 0;
}
```

Plan

Qu'est-ce qu'une file de messages ? ⋆⋆

Définition

Une file de messages est un mécanisme qui permet à deux processus d'échanger des informations. Les messages sont délivrés par ordre de priorité dans l'intervalle [0, `sysconf(_SC_MQ_PRIO_MAX)-1`], soit [0, 32767] avec Linux.

Comparaison file de message / tube

tube	**file de message**
unidirectionnel	-
non-structuré	semi-structuré
pas de priorité	priorité

→ Sous Linux, les files de messages sont créées dans un système de fichiers virtuel, elles sont placées dans `/dev/mqueue/`

Les files de messages POSIX ⋆⋆

Les files de messages POSIX

Les files de messages POSIX sont une implémentation des files de messages pour les systèmes POSIX. Les files POSIX possèdent :

- un nom de la forme `/nom`
- une capacité en nombre de message
- une taille maximum de message

RTFM : `mq_overview(7)`

Ouverture

mq_open(3)

mq_open(3)

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
struct mq_attr *attr);
```

Ouvre (ou crée) une file de message appelée *name* dans le mode *oflag*. Si *oflag* contient `O_CREAT` alors, on utilise la deuxième version et on doit indiquer les permissions *mode* et les attributs de la file **attr*.

- *oflags* est un parmi `O_RDONLY`, `O_WRONLY`, `O_RDWR` associé à d'autres drapeaux éventuels
- **attr* permet de fixer la capacité de la file et la taille maximum des messages

Cette fonction renvoie un descripteur de file de message.

Fermeture et suppression

mq_close(3) et mq_unlink(3)

mq_close(3)

```
int mq_close(mqd_t mqdes);
```

Ferme la file de message dont le descripteur est *mqdes*.

→ Même après fermeture, une file de message reste **persistante**, à la disposition des autres processus.

mq_unlink(3)

```
int mq_unlink(const char *name);
```

Supprime la file de message dont le nom est *name*.

Envoi de message

mq_send(3)

mq_send(3)

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
size_t msg_len, unsigned int msg_prio);
```

- Envoie le message pointé par *msg_ptr* de longueur *msg_len* avec la priorité *msg_prio* dans la file dont le descripteur est *mqdes*.
- Si la file est pleine, alors l'appel bloque tant que la file n'est pas vidée par un autre processus.
- En cas de succès, cette fonction renvoie 0. Sinon, elle renvoie -1 tout en positionnant `errno`.

Réception de message

`mq_receive(3)`

`mq_receive(3)`

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
size_t msg_len, unsigned int *msg_prio);
```

- Reçoit le message, dans le buffer pointé par *msg_ptr*, depuis la file dont le descripteur est *mqdes*.
- *msg_len* indique la taille du buffer pointé par *msg_ptr* ; *msg_len* doit être $\geq$ à la taille maximum d'un message.
- La priorité du message est stockée à l'adresse *msg_prio*.
- Si la file est vide, alors l'appel bloque tant qu'un autre processus n'envoie pas un message.
- En cas de succès, cette fonction renvoie le nombre d'octets du message reçu. Sinon, elle renvoie -1 tout en positionnant `errno`.

C'est tout pour le moment...

Des questions ?