3 – Appels système, entrées/sorties et opérations sur les fichiers

mickael.hoerdt@hesge.ch

HEPIA 2016/2017

Contenu

- Rappels
- introduction aux appels système
- Appels système de modification du système de fichiers
- Entrée/Sorties standard
- Programmation système d'entrées/sorties
- Fichiers stockés

Objectifs d'un système d'exploitation

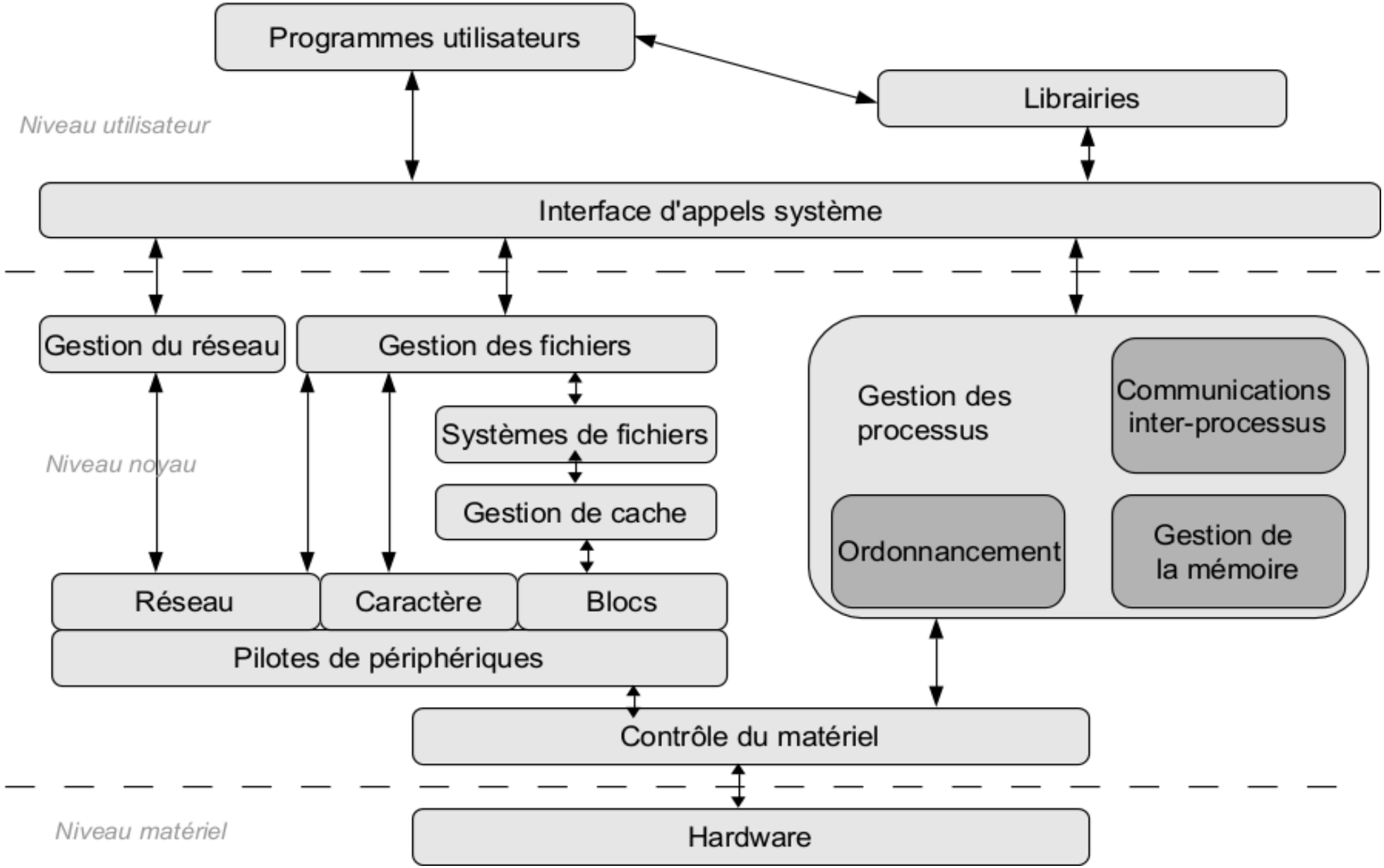
- Gérer efficacement les ressources matérielles par la définition de ressources logiques.
  - Ex :
    - Fichiers stocké = unité d'accès aux périphériques de stockage
    - Processus = unité d'accès aux CPUs.

- Offrir une API permettant d'unifier l'accès aux différentes fonctions du système
  - Ex :
    - Un seule fonction pour écrire dans un fichier, pas forcément stocké

      `write()`
    - Une seule fonction pour démarrer un processus.

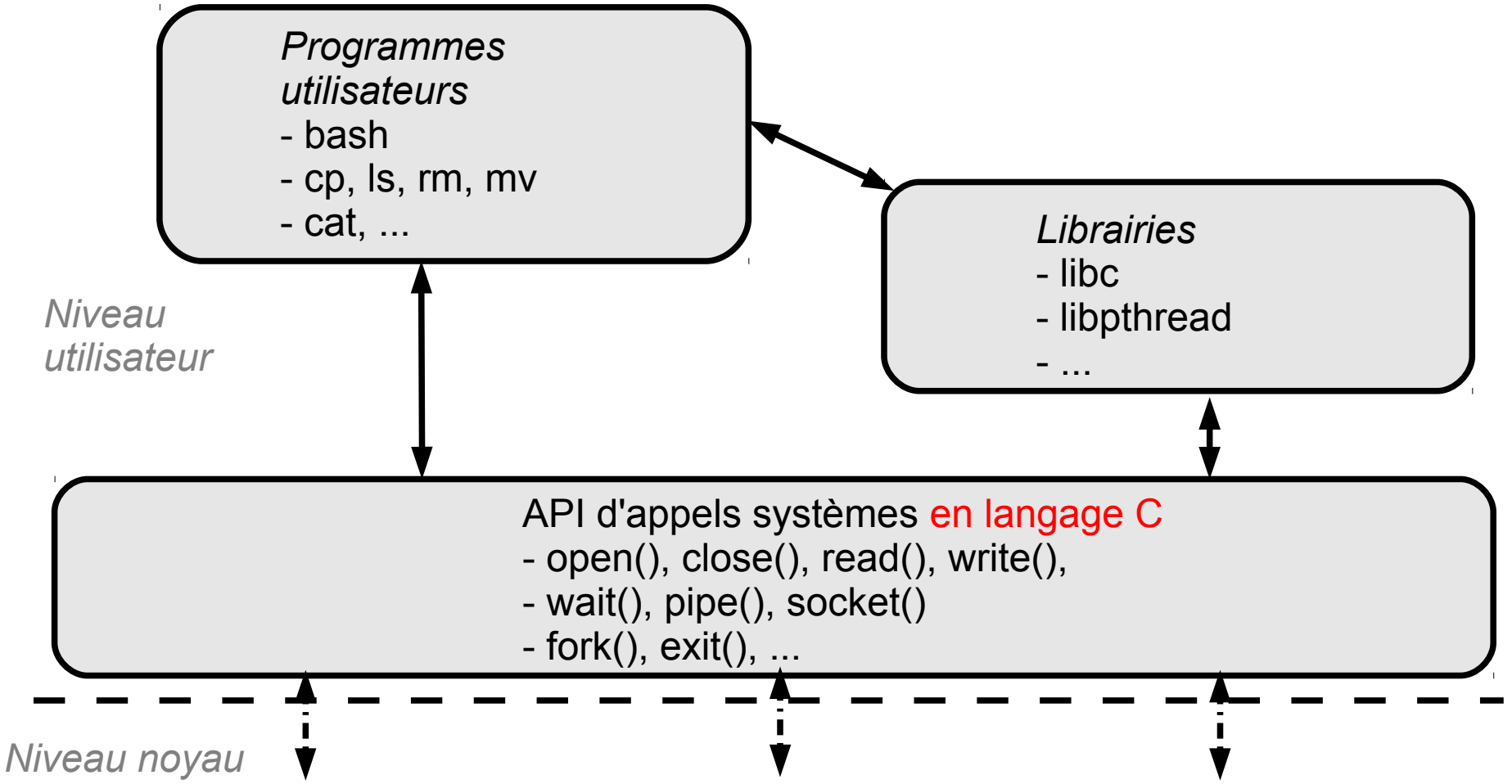
      `fork()`

Rappels sur la vue d'ensemble



Zoom sur la vue d'ensemble

- Ultimement, tous les processus dépendent d'un ou plusieurs appels système pour fonctionner.



- Tracer les appels systèmes d'une commande sous Linux : strace

Appels système versus librairies

Appels systèmes

- Exécuté par le système
- Opérations simples à but unique
- Ne peuvent pas faire appel à une librairie
- Pas visible sous la forme de fichier, interne au système.
- Documentés en Section 2 des pages man

Exemple d'appels systèmes :

- int open(const char *pathname, int flags)
- ssize_t read(int fd, void *buf, size_t count);

Librairies

- Exécuté par le programme utilisateur
- Opérations pouvant effectuer plusieurs tâches
- Peuvent faire des appels système
- Visible et exploités sous la forme de fichiers stockés
- Documentés en section 3 des pages man

Exemple de libraires :

- libpthread : /lib/x86_64-linux-gnu/libpthread-2.13.so
- libc /lib/x86_64-linux-gnu/libc-2.13.so

Note : Les appels systèmes passent en fait par des fonctions *wrappers* implémentées dans la libc en assembleur. Si il n'existe pas de wrapper on peut utiliser syscall pour exécuter l'appel système de manière indirecte (syscall est un appel système qui exécute un autre appel système).

Appels système versus librairies

- Exemple d'implémentation d'appels systèmes sous Linux

Equivalent de write() et exit() sur x86

```
.data
msg: .ascii "Hello World\n"

.text
.global _start

_start:
    movl $4, %eax
    movl $1, %ebx
    movl $msg, %ecx
    movl $12, %edx
    int $0x80

    movl $1, %eax
    movl $1, %ebx
    int $0x80
```

Equivalent de write() et exit() sur x86_64

```
.data
msg: .ascii "Hello World\n"

.text
.global _start

_start:
    movq $1, %rax
    movq $1, %rdi
    movq $msg, %rsi
    movq $12, %rdx
    syscall

    movq $60, %rax
    movq $0, %rdi
    syscall
```


[1] https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Contenu

- Rappels
- introduction aux appels systèmes
- Entrée/Sorties standards
- Programmation système d'entrées/sorties
- Fichiers stockés
- Appels système de modification du système de fichiers

Mon premier appel système

- Quitter un processus avec le code de sortie indiqué en paramètre

```
void exit(int status)
```

- Ex : programme test-exit.c

```
void main()
{
        exit(3);
}
```

Comment tester le programme avec bash ?

Mon premier appel système

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
mickael@computer:~/work/current/hepia$ echo "void main(){ exit(4); }" | gcc -xc - -o /tmp/test-exit 2> /
dev/null; /tmp/test-exit; echo $?
4
mickael@computer:~/work/current/hepia$ vi test-exit.c
mickael@computer:~/work/current/hepia$ cat test-exit.c
void main()
{
        exit(4);
}
mickael@computer:~/work/current/hepia$ gcc -o /tmp/test-exit test-exit.c
test-exit.c: In function ‘main’:
test-exit.c:3:2: attention : incompatible implicit declaration of built-in function ‘exit’ [enabled by d
efault]
mickael@computer:~/work/current/hepia$ /tmp/test-exit
mickael@computer:~/work/current/hepia$ echo $?
4
mickael@computer:~/work/current/hepia$
```

Convention des appels système

- La valeur de retour est généralement un entier

```
int res=appel_systeme(arg1, arg2, ...) ;
```

- Une valeur de retour `< 0` indique qu'une erreur a eu lieu pendant l'appel système.

- Une valeur de retour `>= 0` indique que l'appel système s'est déroulé sans erreur.

- Attention : La valeur de retour ne peut être utilisée comme un booléen indiquant la réussite de l'appel
  - En C : 0 est égal à false, toute valeur différente de 0 est égale à true.
  - Ceci est faux et ambigu : `if ( appel_systeme(arg) ) {...}`

Détails sur l'erreur d'un appel système

- Après chaque appel système, une variable globale de la libc appelée `errno` indique le numéro de la dernière erreur d'un appel système.
- On affiche l'erreur correspondante en utilisant la fonction correspondante de la libc :
  - `void perror(const char *s)`
- Ex :

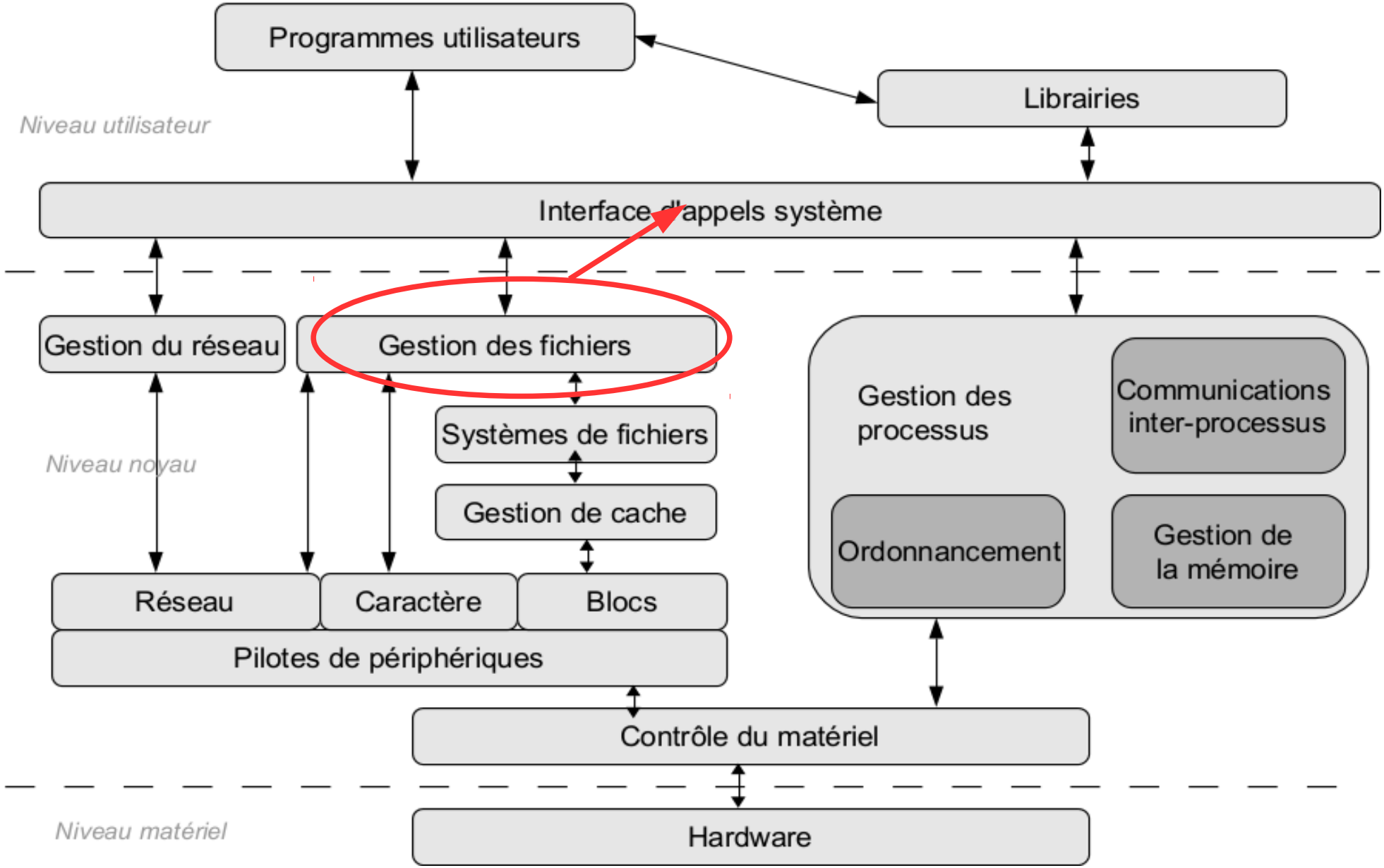
```
void main()
{
        if (appel_syst() < 0 )
                perror("echec de appel_syst") ;

}
```

Contenu

- Rappels
- introduction aux appels systèmes
- Appels système de modification du système de fichiers
- Entrée/Sorties standards
- Programmation système d'entrées/sorties
- Fichiers stockés

Rappels sur la vue d'ensemble



Création de fichier

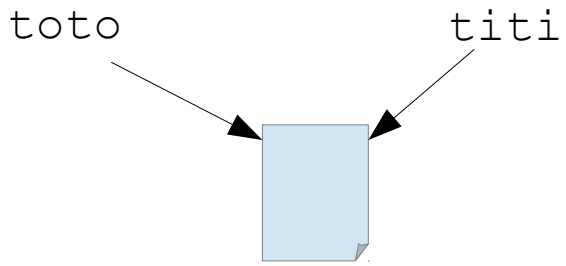
- `int fd = creat(nom, mode)`
  - `const char *nom` = chemin d'accès du fichier en ascii.
  - `int mode` = **masque binaire** des droits d'accès du nouveau fichier ou les droits des trois types d'utilisateurs sont exprimés par un chiffre en octal (`0644 == 110 100 100 == rw- r-- r--`). Attention : le droit d'accès effectivement créé dépends de `umask()` : cf man umask ou help umask dans le shell
  - `int fd =` Un entier positif qui est le descripteur de fichier retourné par la fonction si elle réussit.
  - Le fichier est créé si il n'existe pas.
  - Si le fichier existe, l'appel est équivalent à `open()` en écriture (voir diapo. correspondante)
  - Des constantes sont disponibles pour décrire le mode (cf man 2 creat).
  - Lorsque crée, le fichier est ouvert à l'écriture, même si les droits d'accès ne le permettent pas

"effacer" un fichier

- La commande `rm "efface"` des fichiers.
- En réalité elle utilise l'appel système `unlink()`
- `int res=unlink(nom)`
  - `const char *nom` : nom du fichier
  - `int res` : négatif si échoue, égal à 0 si réussit.
- N'efface pas forcément le contenu du fichier, mais supprime le lien vers le contenu.
- Les noms de fichier UNIX sont en fait des pointeurs vers le contenu du fichier.
- Il peut y avoir plus d'un pointeur sur un même fichier.

Liens directs vers un fichier

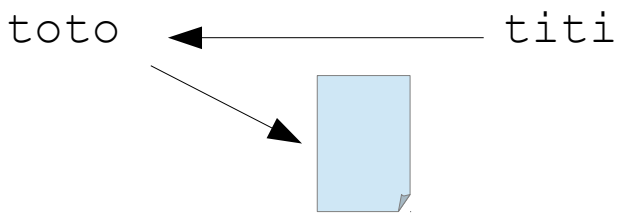
- Lier un fichier existant
  - Dans le shell : `ln toto titi`
  - Appel système : `link("toto", "titi")`
  - `toto` doit déjà exister
  - `toto` et `titi` pointent vers le même fichier !



  - `unlink()et rm` enlèvent simplement un pointeur vers le fichier.
  - Le fichier devient très difficilement accessible ou effacé seulement si le dernier lien est enlevé
  - Très pratique pour faire des sauvegardes sans prendre de place en plus sur le périphérique de stockage !

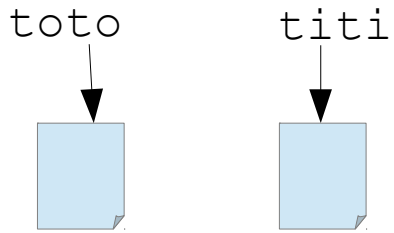
Liens symboliques

- Les liens directs sont limités.
  - Pas possible de créer un lien sur un autre périphérique de stockage
  - Un seul lien par répertoire, si on omets .. et .
- Les liens symboliques sont plus flexibles
  - Shell : `ln -s toto titi`
  - Appel système : `symlink("toto", "titi")`
  - `toto` n'as pas besoin d'exister
  - `titi` pointe vers `toto` comme un alias : il encode simplement le chemin d'accès de toto

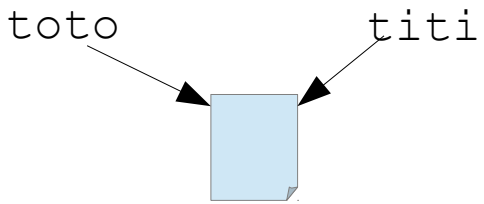


Liens et mise à jour de fichiers

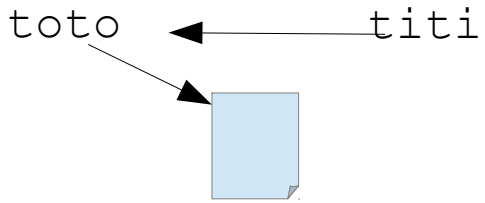
`% Cp toto titi`



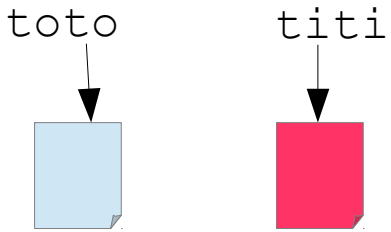
`% ln toto titi`

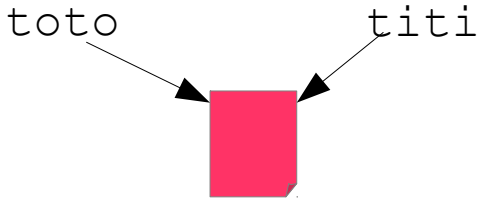


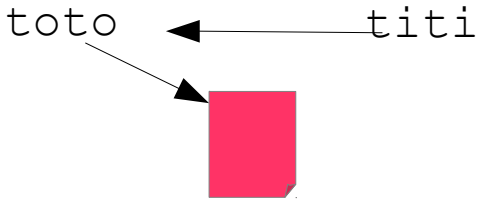
`% ln -s toto titi`



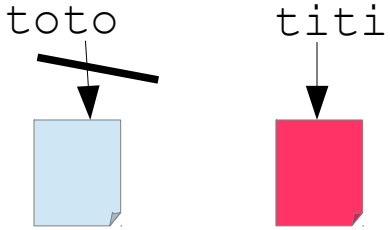
Fichier `titi` mis à jour

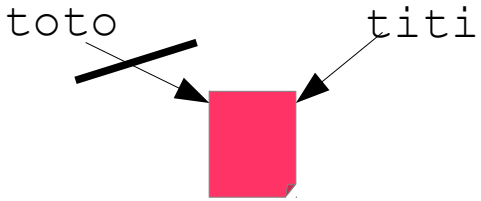


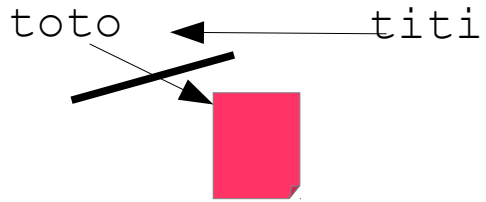




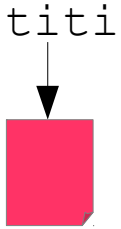
Fichier `toto effacé - unlink ("toto")`

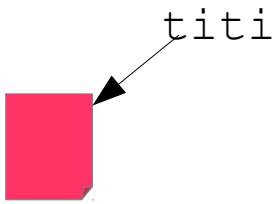






Contenu du fichier `titi ?`

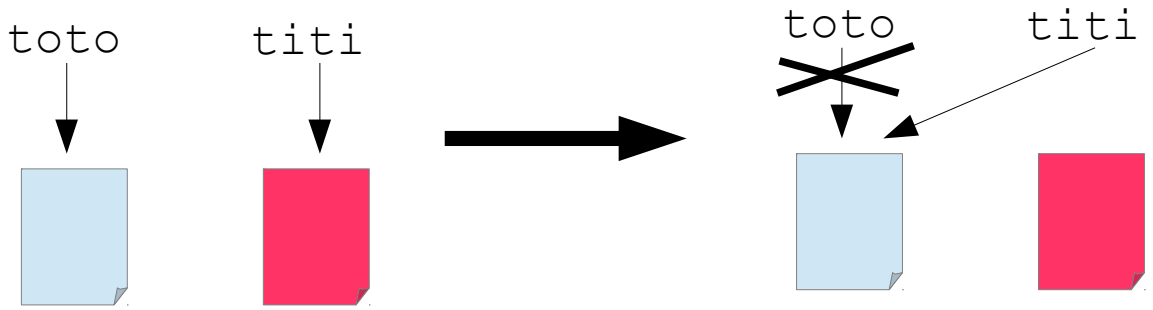




Renommer un fichier

- `int res=rename(old_nom, new_nom)`
  - `const char *old_nom` : Ancien chemin d'accès du fichier
  - `const char *new_nom` : Nouveau chemin d'accès du fichier
  - `int res` : égal à 0 si réussit, négatif sinon.
- Appel système utilisé par la commande `mv`
- Étapes :
  - `unlink(new_nom)`
  - `link (old_nom, new_nom)`
  - `unlink (old_nom)`
- Résultat : new_nom pointe sur le fichier qui était pointé par old_nom

  Ex : `rename ("toto", "titi")`



Répertoires

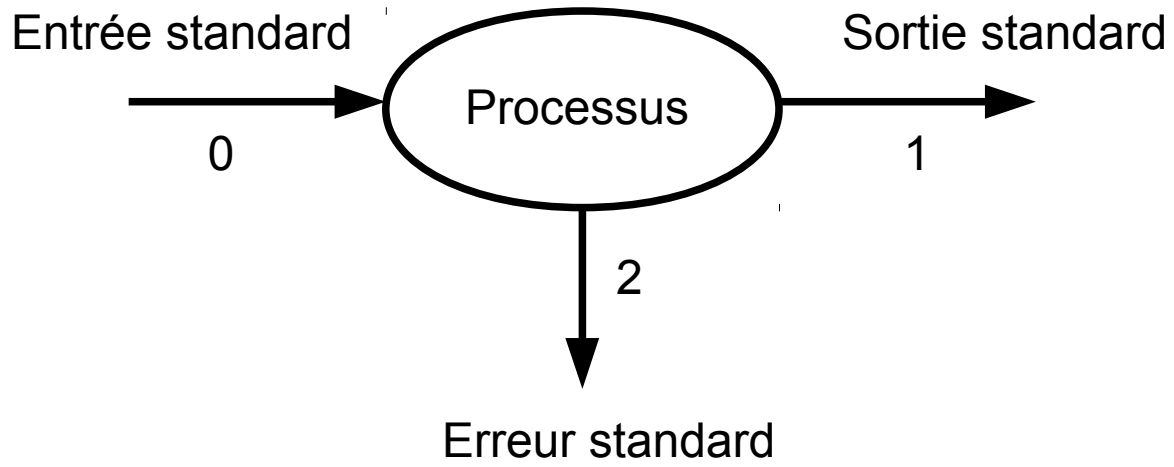
- `int res=mkdir(nom, mode)`
  - `const char *nom` : chemin d'accès du répertoire
  - `mod_t mod` : droits d'accès du répertoire, en notation octale (`ex : 0700 == rwx --- ---`)
  - `int res` : zéro si réussit, négatif sinon .
- `int res= rmdir(nom)`
  - `const char *nom` : chemin d'accès du répertoire
  - `int res` : zéro si réussit, négatif sinon
  - Ne peut effacer que les répertoire déjà vides.

Contenu

- Rappels
- introduction aux appels systèmes
- Appels système de modification du système de fichiers
- Entrée/Sorties standard
- Programmation système d'entrées/sorties
- Fichiers stockés

Entrée/Sorties

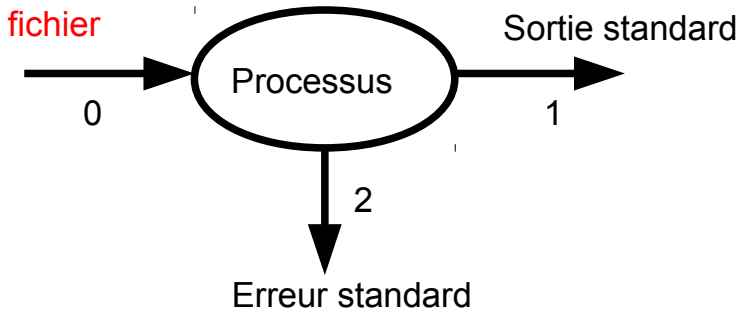
- A son démarrage tout processus a accès à 3 canaux d'entrée/sorties standard.
- Ces canaux sont représentés par un entier de 0 à 2 , qui est un descripteur de fichier :



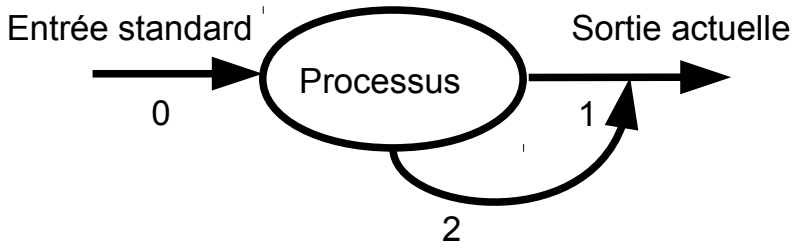
- Entrée standard (0)
  - Utilisée souvent par le processus si aucun fichier donné en paramètre
    - Ex : `cat, grep`.
  - Entrée par défaut depuis le terminal où est exécuté le programme (corresponds généralement au clavier).
- Sortie standard (1)
  - Les sorties simples d'un programme s'affichent ici
    - Ex : `printf()`
  - Sortie par défaut sur le terminal d'exécution du processus.
- Erreur standard (2)
  - Utilisée pour afficher des erreurs depuis un processus
    - Ex : `perror()`
  - Sortie par défaut sur le terminal d'exécution du processus.

Entrée/Sorties : redirections shells

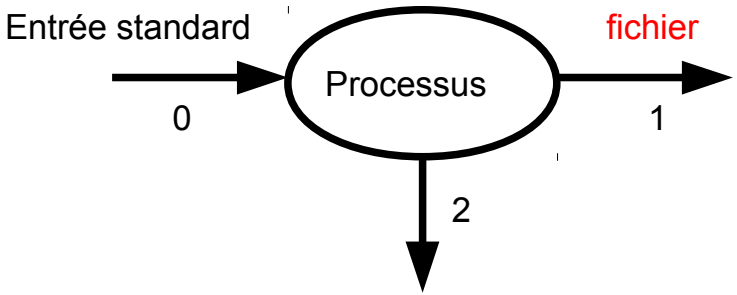
- Commande < fichier



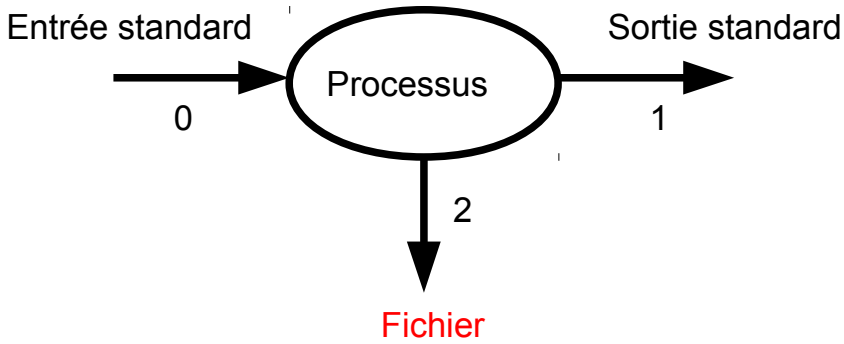
Commande 2>&1



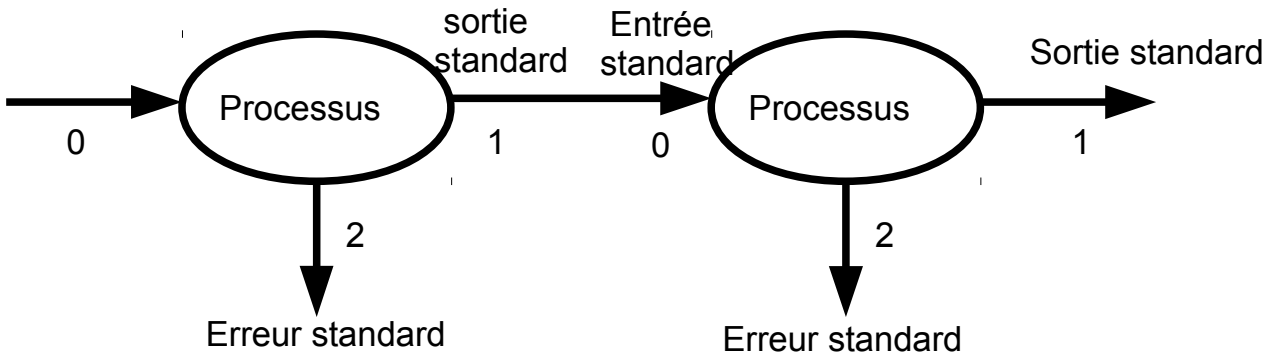
- Commande > fichier



Commande 2>fichier



- Commande1 | commande2



Entrées/Sorties : filtres

- Certaines commandes demandent obligatoirement un/des nom de fichiers en paramètre :
  - Ex : `cp fichier1 fichier2`
- Beaucoup de commandes utilisent l'entrée standard et la sortie standard par défaut
  - `cat, cut, sort, head, tail, wc, uniq, grep ...`
- Ces dernières commandes sont appelées filtres
  - Très facile à utiliser via un |
- Très facile à programmer : juste un processus qui lit et écrit
  - Pas besoin de se gérer l'ouverture/fermeture de fichiers
  - fonctionne sans argument de ligne de commande.

Contenu

- Rappels
- introduction aux appels systèmes
- Appels système de modification du système de fichiers
- Entrée/Sorties standard
- Programmation système d'entrées/sorties
- Fichiers stockés

Ecrire dans un fichier en C : appel système

- `res = write (fd, buffer, longueur)`
  - `int fd =` descripteur de fichier ouvert en lecture.
  - `void *buffer` = pointeur d'octet vers buffer source des données à écrire.
  - `size_t longueur` = longueur des données à écrire en octets (entier).
  - `ssize_t res` = le nombre d'octets écrits au retour de la fonction
    - Res est >=0 si pas d'erreur < 0 si il y a erreur
    - Res peut-être < longueur (Si par ex, les buffers de l'OS sont plein).

      ⟶ Toujours vérifier que res est égal à longueur.
    - Buffer est une suite d'octets quelconques, pas forcément une chaîne C terminant par \0

Lire depuis un fichier en C : appel système

- `res = read(fd, buffer, longueur)`
  - `int fd` = descripteur de fichier ouvert en lecture.
  - `void *buffer` = pointeur d'octets vers buffer de destination des données à lire.
  - `size_t longueur` = longueur des données à lire en octets (entier).
  - `ssize_t res` = le nombre d'octets lus au retour de la fonction.
    - Res est à `0` si c'est la fin de fichier et `<0` si il y a une erreur.
    - Buffer ne contient que le résultat de la lecture, si on lit une chaîne de caractère, elle ne se terminera pas par \0, il faut le rajouter et prévoir la place nécessaire !
    - Res peut être < longueur (Si on lit le dernier bout d'un fichier par ex)
    - longueur peut-être plus petit que la taille des données à lire, au prochain appel de read, read lira la suite par bloc de longueur octets

Ouvrir d'autres fichiers

```
#include <fcntl.h>
```

- `int fd = open(nom, flags)`
  - `const char *nom` = chemin d'accès du fichier en ascii.
  - `int flags =` un champs binaire indiquant l'accès au fichier : en lecture, en écriture, en création, etc..
    - L'un de ces flags est obligatoire :
      - `O_RDONLY` lecture uniquement (0)
      - `O_WRONLY` ecriture uniquement (1)
      - `O_RDWR` lecture et ecriture (2)
    - Qu'on peut combiner avec d'autre avec l'opérateur logique binaire "OU" du C : O_WRONLY | O_CREAT
  - `open(nom,O_WRONLY|O_CREAT|O_TRUNC, mode)` équivalent à `creat(nom, mode)`
  - `int fd` = Un entier positif qui est le descripteur de fichier retourné par la fonction si elle réussit.
  - Si `fd < 0` il y a eu erreur, l'afficher avec perror.
  - Le fichier n'est pas forcément un fichier stocké, il peut s'agir d'un périphérique !

Fermer un fichier

- `int res = close(fd)`

  `int fd =` Un descripteur de fichier déjà ouvert.

  `int res =` resultat = 0 si succès, négatif sinon.

- Le système gère une table de descripteurs disponibles/utilisés.
- Un processus peut très bien fermer l'entrée standard, le descripteur 0 sera alors disponible.

Dupliquer les descripteurs

- `int fd1 = dup(oldfd)`
- `int fdnew = dup2 (oldfd,newfd)`
  - Les appels système `dup` et `dup2` dupliquent un descripteur de fichier et renvoient le nouveau descripteur dupliqué
  - Ancien et nouveau descripteur peuvent être utilisés de manière interchangeable
  - Tous deux référent au même fichier ouvert et partagent les mêmes flags.
  - `Dup` utilise le plus petit descripteur de fichier disponible comme nouveau descripteur de fichier : la clef des redirections du shell !
  - `Dup2` permet à l'appelant de choisir le numéro du nouveau descripteur de fichier ; ce dernier est automatiquement fermé si nécessaire.
  - Si `oldfd` n'est pas valide, l'appel échoue et newfd n'est pas fermé
  - Si `oldfd` est valide et `newfd` == `oldfd` alors `dup2` ne fait rien et

Contenu

- Rappels
- introduction aux appels systèmes
- Appels système de modification du système de fichiers
- Entrée/Sorties standard
- Programmation système d'entrées/sorties
- Fichiers stockés

Type de fichier fichier particulier : les fichiers stockés

- Définition : une suite linéaire de bytes, faisant abstraction d'une région mémoire non-volatile.
  - Le contenu du fichier persiste même lorsque le programme se termine
  - Aucune information sur l'organisation de l'espace du support n'est présente à ce niveau d'abstraction
- On manipule un fichier par des attributs indépendants du support :
  - Nom, type, taille
  - Date de création, modification, dernier accès
  - Propriétaire et droits d'accès.

Fichiers stockés : accès aléatoire

- `read()` et `write()` accèdent à un descripteur de fichier de manière séquentielle.
- La fonction `lseek()` permet un accès à un fichier à une position définie arbitrairement.
- `off_t pos = lseek(fd, offset, wherefrom)`
  - `int fd` : un descripteur de fichier ouvert.
  - `off_t offset` : entier indiquant nombre d'octets à se déplacer dans le fichier. Si négatif se déplace en arrière.
  - `int wherefrom` : constante indiquant depuis où s'effectue le déplacement.
    - `SEEK_SET` : depuis le début du fichier.
    - `SEEK_CUR` : depuis la position courante.
    - `SEEK_END` : depuis la fin du fichier.
- Pour trouver où est la position courante : se déplacer de `0` depuis `SEEK_CUR`.

Récapitulatif des appels système vus

Quitte proprement un processus et indiquer un code de sortie :

- `void exit(int status)`

Opérations sur le système de fichier :

- `int creat(const char *name, mode_t mode)`
- `int link(const char *oldpath, const char *newpath)`
- `int symlink(const char *oldpath, const char *newpath)`
- `int unlink(const char *pathname)`
- `int rename(const char *oldpath, const char *newpath)`
- `int mkdir(const char *name)`
- `int rmdir(const char *name)`

Opérations sur les fichiers :

- `int read(int fd, void *buffer, size_t longueur)`
- `int write(int fd, void *buffer, size_t longueur)`
- `int open(const char *nom, int flags, mode_t mode)`
- `int open(const char *nom, int flags)`
- `int close(int fd)`
- `int dup(int oldfd)`
- `int dup2(int oldfd, int newfd)`
- `int lseek(int fd, off_t offset, int whence)`