

# Manipulation des fichiers en C

Cyril Rabat

`cyril.rabat@univ-reims.fr`

Licence 3 Informatique - Info0601 - Systèmes d'exploitation - concepts avancés

2019-2020



**Cours n°3**

*Appels système*

*Fonctions bas-niveau et haut-niveau*

Version 20 décembre 2019

# Table des matières

## 1 Généralités sur les appels systèmes

- Appels système
- Gestion des erreurs
- Compatibilité

## 2 Manipulation des fichiers en C

- Le VFS
- Manipulation bas-niveau : `open`, `close`, `read` et `write`
- Les répertoires

# Appels systèmes

- *Définition* : **appel système**

↪ Un appel système est une fonction fournie par le noyau du système d'exploitation et utilisée par les programmes s'exécutant en mode utilisateur.

- Exemples d'appels système en C :

- Fichiers : `open`, `close`, `read`, `write` ...
- Processus : `exec`, `fork`, `wait`, `exit` ...
- Communications inter-processus : `pipe`, `msgget`, `semget` ...

# Appels systèmes “masqués”

- Certaines fonctions de la bibliothèque sont des frontaux d'appels système :
  - ↪ Généralement plus simples à utiliser
  - ↪ Mais limitent les possibilités
- On parle aussi d'appels de haut niveau
- Attention aux effets :
  - ↪ Erreurs possibles correspondant à l'appel système réalisé
  - ↪ Certaines fonctions correspondent à plusieurs appels système
- Exemple avec `fclose` :
  - ↪ Appels "cachés" de `close`, `write` et `fflush`

# Le man

- Commande *Unix* qui permet d'afficher les pages du *Unix Programmer's Manual*
- Utilisation :

```
man [-s <section>] <commande>
```

- Les sections :
  - 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.

# Gestion des erreurs

- La variable `errno` est renseignée par les appels système (et quelques fonctions de la bibliothèque)
- Si un appel système échoue, la valeur retournée est généralement -1 et `errno` est mise à jour
- Fonctions/variables relatives aux erreurs :
  - `extern int errno` : valeur de la dernière erreur rencontrée (exemples : `EEXIST`, `EIO`, `EACCES`...)
  - `void perror (const char *s)` : affiche un message sur la sortie d'erreur standard concernant la dernière erreur rencontrée
  - `char *strerror (int errnum)` : affiche le message correspondant à un numéro d'erreur
  - `const char *sys_errlist[]` : tableau contenant les messages d'erreur
  - `int sys_nerr` : nombre de cases dans le tableau `sys_errlist`

# Erreurs associées à un appel système (1/2)

## ERREURS

**EBADF** Le descripteur de fichier `fd` est invalide.

**EINTR** L'appel système `close()` a été interrompu par un signal ; voir [signal\(7\)](#).

**EIO** Une erreur d'entrée-sortie s'est produite.

*extrait de man -s2 close (sur le Web)*

## Remarques

- Il n'est pas nécessaire de proposer des traitements personnalisés pour toutes les erreurs
- Certaines ne sont pas forcément à traiter
- Mais il est conseillé de vérifier chaque retour d'appel système !

## Erreurs associées à un appel système (2/2)

### NOTES

Ne pas vérifier la valeur de retour de `close()` est une pratique courante mais également une grave erreur de programmation. Il est possible qu'une erreur correspondant à un appel `write(2)` antérieur ne soit rapportée que lors du `close()` final. Ne pas vérifier la valeur de retour lorsque l'on ferme un fichier peut conduire à une perte silencieuse de données. Ceci est principalement vrai dans le cas de systèmes de fichiers NFS, ou avec l'utilisation des quotas de disques.

Une fermeture sans erreur ne garantit pas que les données ont été vraiment écrites sur le disque, car le noyau repousse les écritures le plus tard possible. Il n'est pas fréquent qu'un système de fichiers vide les tampons dès la fermeture d'un flux. Si vous désirez vous assurer que les informations sont en sûreté sur le disque, utilisez `fsync(2)` (mais des considérations matérielles entrent en jeu à ce moment).

Il est probablement imprudent de fermer des descripteurs de fichier alors qu'ils peuvent peut-être être utilisés par des appels système dans d'autres threads du même processus. Puisqu'un descripteur de fichier peut être réutilisé, il y a des conditions de concurrence obscures qui peuvent provoquer des effets de bord non désirés.

*extrait de man close (sur le Web)*



# Différents standards pour le C

- Dans chaque page des sections 2 et 3, section *Conformité*  
↪ En anglais : *Conforming to*
- Quelques standards :
  - C89 : premier standard ratifié par l'ANSI (en 1989), noté aussi ANSI C (mais ambigu)  
↪ Normalisé en 1990 par l'ISO : ISO/IEC 9899 :1990 (aussi appelé ISO C90)
  - C99 : révision du langage C ratifié en 1999 (ISO/IEC 9899 :1999)
  - POSIX.1-1990 : *Portable Operating System Interface for Computing Environments*  
↪ IEEE 1003.1-1990 partie 1 (ISO/IEC 9945-1 :1990)
  - POSIX.1X, POSIX.2, etc. : différentes consolidations
- Pour plus d'informations : `man standards` (section 7)

# System V

- Version d'origine de l'OS *Unix*, développé par AT&T
- Une des principales branches des systèmes *Unix*
- À la base de l'élaboration de la norme POSIX
- Standard `System V release 4 (SVr4)` :
  - ↪ Publiée en 1989
  - ↪ Considérée comme la *release* définitive
- Nombreuses références :
  - ↪ Les IPC (pour *Inter Process Communication*) System V

# Exemple de compatibilité

## CONFORMITÉ

SVr4, BSD 4.3, POSIX.1-2001. Les attributs `O_DIRECTORY`, `O_NOATIME` et `O_NOFOLLOW` sont spécifiques à Linux; il faut définir la macro `_GNU_SOURCE` pour avoir leurs définitions.

L'attribut `O_CLOEXEC` n'est pas spécifié dans POSIX.1-2001, mais l'est dans POSIX.1-2008.

`O_DIRECT` n'est pas spécifié par POSIX; il faut définir la macro `_GNU_SOURCE` pour obtenir sa définition.

*Extrait de `man -s2 open` (sur le Web)*

## À éviter

- Utilisation de fonctions non conformes C89/C99, POSIX ou SVr4
- Ne pas utiliser d'options spécifiques au système (cas de *Linux*)

## Utilisation d'options de compilation de *gcc*

- `-ansi` : équivalent à `-std=c90` (dépend de la version installée de *gcc*)
- `-pedantic` : suit le standard spécifié par `-std`
- `-Werror` : tous les avertissements deviennent des erreurs
- `-Wall` : affiche tous les avertissements

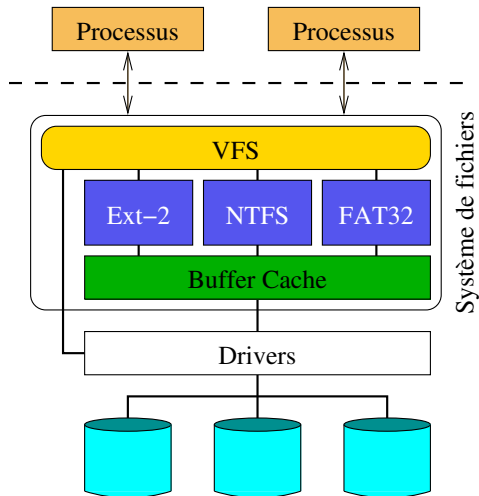
### Obligation pour tous les codes fournis en Info0601

- Les codes fournis devront être compilés avec les options :
  - ↪ `-Wall` et `-Werror` : pour éviter les erreurs
  - ↪ `-c90` et `-pedantic` : pour le respect du langage C

# Le VFS (1/2)

- Nécessité de supporter de multiples systèmes de fichiers
- *Linux* propose le VFS pour *Virtual File System* :
  - ↪ Couche logicielle dans le noyau
- Permet de masquer à l'utilisateur les différences des systèmes de fichiers
- Mise en cache de différentes informations :
  - ↪ *Buffer Cache* (blocs du disque), conversions noms/*i-node*
- Accès unifié : primitives `read`, `write`, etc.

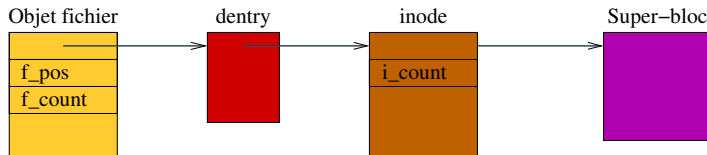
## Le VFS (2/2)



# Les différentes structures (1/4)

- Les structures C utilisées :
  - Super-bloc
  - *inode*
  - fichier
  - *dentry*

## Illustration



## Les différentes structures (2/4) : super-bloc

- Représente un système de gestion de fichiers
- Décrit :
  - Le type du système
  - Ses caractéristiques, etc.
- Un système de fichiers est accessible lorsqu'il est monté :  
↪ Enregistrement + montage (commande `mount`)



## Les différentes structures (3/4) : inode

- Représente chaque fichier dans le VFS
- Informations diverses sur le fichier (comme Ext2)
- Informations sur le système de fichiers + opérations possibles
- Organisés dans une liste double-chainée circulaire
- Table de hachage pour accélérer les recherches

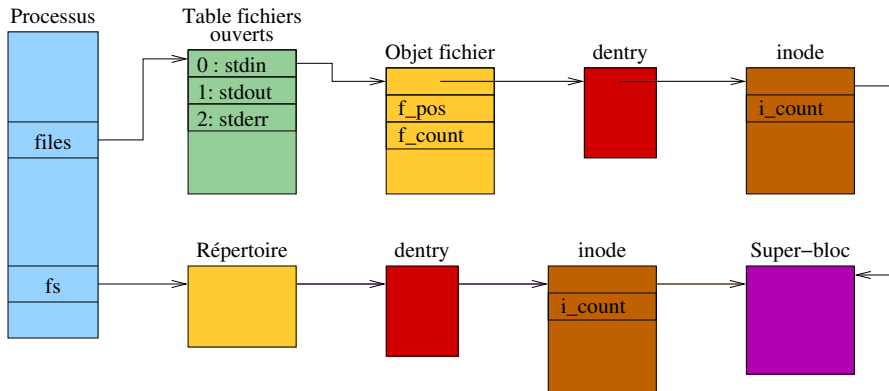
## Les différentes structures (4/4) : fichier et dentry

- Objet fichier :
  - Représente un fichier ouvert
  - Informations diverses pour la lecture, l'écriture, etc.
  - Position dans le fichier (`f_pos`)
  - Nombre de références vers cet objet (`f_count`)
- Structure `dentry` :
  - Assure la correspondance entre nom et numéro d'*inode*
  - Mise en cache pour accélérer les accès

## Processus et VFS (1/2)

- Pour chaque processus (entre autres) :
  - ↪ Une table des fichiers ouverts
  - ↪ Un champ `fs` qui correspond au répertoire d'exécution du processus
- Chaque entrée de la table des fichiers :
  - ↪ Identifiée par un numéro (descripteur de fichier)
  - ↪ Pointe vers un objet fichier
- Fichier pointe vers un objet *dentry* :
  - ↪ Correspondance nom/*inode*
- *dentry* pointe vers l'*inode* du fichier
- L'*inode* du fichier pointe vers un super-bloc

## Processus et VFS (2/2)



# Descripteur de fichier

- Manipulation des fichiers via des descripteurs de fichiers :  
↪ Sous *Windows*, on parle plutôt de *filehandle*
- Sous *Linux*, un descripteur peut être :
  - Un fichier ou un répertoire
  - Un tube nommé ou anonyme
  - Une socket ...

## Remarques sur les méthodes de manipulation des fichiers

- Les méthodes bas-niveau travaillent directement sur les descripteurs de fichier
- Les méthodes haut-niveau travaillent sur des flux qui exploitent les descripteurs de fichier

# Ouverture/création d'un fichier (1/3)

## En-têtes des fonctions (S3)

- `int open(const char* chemin, int flags)`
- `int open(const char* chemin, int flags, mode_t mode)`
- `int creat(const char *chemin, mode_t mode)`
- *Inclusions* :
  - `fcntl.h` pour les fonctions
  - `sys/stat.h` et `sys/types.h` pour les constantes/types

## Explications

- `open` permet de créer ou d'ouvrir un fichier en lecture/écriture
- `creat` est un équivalent de `open` : le fichier est créé et ouvert en écriture

## Ouverture/création d'un fichier (2/3)

### Paramètres

- `chemin` : nom du fichier (chemin relatif ou absolu)
- `flags` définit les options :
  - Obligatoirement :
    - `O_RDONLY` : ouverture en lecture seule
    - `O_WRONLY` : ouverture en écriture seule
    - `O_RDWR` : lecture et écriture
  - Optionnels :
    - `O_CREAT` : crée le fichier s'il n'existe pas
    - `O_EXCL` : génère une erreur avec "`O_CREAT`" si le fichier existe déjà
    - `O_TRUNC` : la taille est tronquée à 0 (uniquement avec "`O_WRONLY`" ou "`O_RDWR`")
    - `O_APPEND` : ouverture en mode ajout (incompatible avec certains systèmes de fichiers)

## Ouverture/création d'un fichier (3/3)

### Paramètres

- mode définit les droits sur le fichier (R, W, X ou RWX) avec l'option "O\_CREAT" uniquement :
  - S\_IRUSR, S\_IWUSR, S\_IXUSR, S\_IRWXU : pour l'utilisateur
  - S\_IRGRP, S\_IWGRP, S\_IXGRP, S\_IRWXG : pour le groupe
  - S\_IROTH, S\_IWOTH, S\_IXOTH, S\_IRWXO : pour les "autres"

### Valeur retournée et erreurs générées

- Retourne un descripteur de fichier ou -1 en cas d'erreur
- Quelques erreurs possibles :
  - EEXIST : le fichier existe déjà (et flags O\_CREAT et O\_EXCL)
  - EACCES : accès interdit
  - ENSPC : pas assez de place pour créer le fichier



## Fermeture d'un fichier

### En-tête de la fonction (S2)

- `int close(int fd)`
- *Inclusion* : `unistd.h`

### Explications

- Ferme le fichier correspondant au descripteur de fichier “fd”

### Paramètres

- `fd` : le descripteur de fichier à fermer

### Valeur retournée et erreur générée

- Retourne 0 en cas de réussite ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EBADF` : le descripteur “fd” est invalide
  - `EINTR` : interrompu par un signal

# Lecture/écriture depuis un fichier (1/2)

## En-têtes des fonctions (S2)

- `ssize_t read(int fd, void *tampon, size_t taille)`
- `ssize_t write(int fd, void *tampon, size_t taille)`
- *Inclusion* : `unistd.h`

## Explications

- `read` lit un tampon depuis un fichier ouvert en lecture
- `write` écrit un tampon dans un fichier ouvert en écriture
- Le type de retour est utilisé pour vérifier le nombre d'octets lus/écrits :  
⇒ Si c'est différent de `taille`, on est à la fin du fichier

## Attention

La lecture/écriture peut aussi être interrompue par un signal

## Lecture/écriture depuis un fichier (2/2)

### Paramètres

- `fd` : le descripteur de fichier
- `tampon` : un tampon où stocker les données lues ou contenant les données à écrire
- `taille` : taille du tampon

### Valeur retournée et erreurs générées

- Retourne le nombre d'octets lus/écrits ou -1 en cas d'échec
- Quelques erreurs possibles :
  - `EIO` : erreur d'entrée/sortie
  - `EBADF` : `fd` n'est pas un descripteur valide
  - `ENOSPC` : le périphérique correspondant à `fd` n'a plus de place disponible
  - `EINTR` : interruption par un signal

# Déplacement dans un fichier (1/2)

## En-têtes des fonctions (S2)

- `off_t lseek(int fd, off_t offset, int pointDepart)`
- *Inclusions* :
  - `unistd.h` pour la fonction, les constantes
  - `sys/types.h` pour `off_t`

## Explications

- Déplace la tête de lecture dans le fichier d'un nombre d'octets "`offset`" à partir de la position spécifiée :
  - Début du fichier
  - Position courante
  - Fin de fichier
- Peut être utilisée pour récupérer la position courante :  
↪ Avec "`offset`" égal à "`0L`"

## Déplacement dans un fichier (2/2)

### Paramètres

- `fd` : le descripteur de fichier
- `offset` : le nombre d'octets correspondant au déplacement
- `pointDepart` est le point de départ du déplacement :
  - `SEEK_SET` : à partir du début du fichier
  - `SEEK_CUR` : à partir de la position courante
  - `SEEK_END` : à partir de la fin du fichier

### Valeur retournée et erreurs générées

- En cas de réussite, retourne la position courante dans le fichier (à partir du début). En cas d'échec, retourne -1
- Quelques erreurs possibles :
  - `EBADF` : "`fd`" n'est pas valide
  - `EINVAL` : "`pointDepart`" n'est pas une valeur valide

# Tronquer un fichier (1/2)

## En-têtes des fonctions (S2)

- `int ftruncate(int fd, off_t longueur)`
- *Inclusions* :
  - `unistd.h` pour la fonction, les constantes
  - `sys/types.h` pour `off_t`

## Explications

- Tronque le fichier après la longueur spécifiée
- Si le fichier est plus court, ajout de `'\0'`

## Exigence de macro

```
_BSD_SOURCE >= 1 || _XOPEN_SOURCE >= 500 ||  
_POSIX_C_SOURCE > 200112L
```

## Tronquer un fichier (2/2)

### Paramètres

- `fd` : le descripteur de fichier (ouvert en écriture)
- `longueur` : la nouvelle longueur du fichier

### Valeur retournée et erreurs générées

- En cas de réussite, retourne 0. En cas d'échec, retourne -1
- Quelques erreurs possibles :
  - `EBADF` : "`fd`" n'est pas valable
  - `EINVAL` : "`fd`" n'est pas un descripteur de fichier ordinaire
  - `EBADF` ou `EINVAL` : le fichier n'est pas ouvert en écriture

# Ouverture d'un répertoire (1/2)

## En-têtes des fonctions (S3)

- `DIR *opendir(const char *chemin)`
- `int closedir(DIR *rep)`
- *Inclusions* :
  - `dirent.h` pour les fonctions
  - `sys/types.h` pour les types et constantes

## Explications

- `opendir` ouvre un répertoire en lecture
- `closedir` ferme le répertoire



## Ouverture d'un répertoire (2/2)

### Paramètres

- `chemin` : le nom du répertoire
- `rep` : le répertoire à fermer

### Valeurs retournées et erreurs générées

- `NULL` en cas d'erreur
- Quelques erreurs possibles :
  - `EACCESS` : accès interdit
  - `ENOENT` : répertoire inexistant
  - `ENOTDIR` : ce n'est pas un répertoire
  - `EBADF` : descripteur de flux invalide

# Lecture d'une entrée

## En-têtes des fonctions (S3)

- `struct dirent *readdir(DIR *rep)`
- `void rewinddir(DIR *rep)`
- La structure `direct` associée :

```
struct dirent {
    /* etc. */           /* Spécifique à l'implémentation */
    char d_name[256];    /* Nom de l'entrée */
}
```

- *Inclusions* :
  - `dirent.h` pour les fonctions
  - `sys/types.h` pour les types et constantes

## Explications

- `readdir` lit la prochaine entrée du répertoire
- `rewinddir` retourne à la première entrée du répertoire

# Statistiques sur un fichier (1/2)

## En-têtes des fonctions (S2)

- `int stat(const char *chemin, struct stat *buf)`
- `int fstat(int fd, struct stat *buf)`
- `int lstat(const char *chemin, struct stat *buf)`
- *Inclusions :*
  - `sys/stat.h` pour les fonctions
  - `unistd.h` et `sys/types.h` pour les types et constantes

## Explications

- `stat` donne des indications sur le fichier spécifié
- `fstat` idem à partir d'un descripteur ouvert
- `lstat` donne des indications sur le lien (et non du fichier lié)

## Exigence de macro (pour `lstat`)

```
_BSD_SOURCE || _XOPEN_SOURCE >= 500
```

## Statistiques sur un fichier (2/2)

### Paramètres

- `chemin` : le nom du fichier
- `buf` : une structure de type `stat` **allouée**
- `fd` : le descripteur de fichier (ouvert)

### Valeurs retournées et erreurs générées

- 0 en cas de réussite, -1 en cas d'échec
- Quelques erreurs possibles :
  - `EBADF` : mauvais descripteur
  - `ENOENT` : fichier n'existe pas
  - `EACCESS` : accès interdit

## Structure stat

```

struct stat {
    dev_t      st_dev;      /* Périphérique */
    ino_t      st_ino;      /* Numéro i-node */
    mode_t     st_mode;     /* Protection */
    nlink_t    st_nlink;    /* Nb liens matériels */
    uid_t      st_uid;      /* UID propriétaire */
    gid_t      st_gid;      /* GID propriétaire */
    dev_t      st_rdev;     /* Type périphérique */
    off_t      st_size;     /* Taille totale en octets */
    blksize_t  st_blksize;  /* Taille de bloc pour E/S */
    blkcnt_t   st_blocks;   /* Nombre de blocs alloués */
    time_t     st_atime;    /* Heure dernier accès */
    time_t     st_mtime;    /* Heure dernière modification */
    time_t     st_ctime;    /* Heure dernier changement */
};

```

## Quelques macros

- `S_ISREG(st_mode)` : est-ce un fichier régulier ?
- `S_ISDIR(st_mode)` : est-ce un répertoire ?
- `S_ISLINK(st_mode)` : est-ce un lien symbolique ?