Manipulation des fichiers en C

Cyril Rabat
cyril.rabat@univ-reims.fr

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

2021-2022

Cours n°3

Appels système

Fonctions bas-niveau et haut-niveau

Version 14 décembre 2021

Table des matières

Qu'est-ce qu'un appel système ?

Définition : appel système

Un appel système désigne le moment où un programme s'interrompt pour laisser le système d'exploitation réaliser une certaine tâche. Cela désigne également une fonction fournie par le noyau du système d'exploitation.

- Tant qu'un appel système n'est pas traité, le processus est bloqué
- Exemples d'appels systèmes 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 systèmes :
  ↪ Généralement plus simples à utiliser . . .
  ↪ . . . mais limitent les possibilités
- Nommées également **appels de haut niveau**
- Attention aux effets :
  - Erreurs possibles correspondant à l'appel système réalisé
  - Certaines fonctions correspondent à plusieurs appels systèmes
- Exemple avec `fclose` :
  ↪ Appels "cachés" de `close`, `write` et `fflush`

Surveiller les appels système : `strace`

- Outil permettant de tracer les appels systèmes d'un processus
- Utilisation classique :
  ↪ `strace ./toto` (où `toto` est le programme à exécuter)
- Possible d'effectuer sur un processus en cours d'exécution :
  ↪ `strace -p PID` où PID est le PID du processus
- Création d'un rapport (bilan des appels, des erreurs) :
  ↪ `strace -c ./toto`
- Redirection de la sortie de `strace` (utile si le programme nécessite des saisies clavier) :
  ↪ `strace -o output.txt ./toto`
- Surveillance d'un ou de plusieurs appels système :
  ↪ `strace -e write,close ./toto`
- Pour l'horodatage : options `-tt`, `-T`
  ↪ `-T` pour la durée des appels systèmes
  ↪ `-tt` pour l'horodatage (`-t` pour afficher uniquement les secondes)

Exemple d'utilisation de strace (1/2)

```c
#include <stdio.h>      // Pour fprintf, perror
#include <stdlib.h>     // Pour exit, EXIT_SUCCESS

int main(int argc, char **argv) {
  FILE *f;

  if((f = fopen("toto.txt", "w")) == NULL) {
    perror("Erreur ");
    exit(EXIT_FAILURE);
  }
  fprintf(f, "Coucou tout le monde !!!");

  if(fclose(f) != 0) {
      perror("Erreur ");
      exit(EXIT_FAILURE);
  }

  return EXIT_SUCCESS;
}
```

Exemple d'utilisation de `strace` (2/2)

```
mmap(0x7f36cd20d000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_
mmap(0x7f36cd213000, 13528, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f36cd218540) = 0
mprotect(0x7f36cd20d000, 12288, PROT_READ) = 0
mprotect(0x556a4a9db000, 4096, PROT_READ) = 0
mprotect(0x7f36cd252000, 4096, PROT_READ) = 0
munmap(0x7f36cd219000, 48648)           = 0
brk(NULL)                               = 0x556a4bf82000
brk(0x556a4bfa3000)                     = 0x556a4bfa3000
openat(AT_FDCWD, "toto.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
fstat(3, {st_mode=S_IFREG|0777, st_size=0, ...}) = 0
write(3, "Coucou tout le monde !!!", 24) = 24
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

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 systèmes
  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 systèmes (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)

```
ERRORS
       EBADF  fd isn't a valid open file descriptor.

       EINTR  The close() call was interrupted by a signal; see signal(7).

       EIO    An I/O error occurred.

       ENOSPC, EDQUOT
              On  NFS,  these  errors  are  not normally reported against the first write which exceeds the available
              storage space, but instead against a subsequent write(2), fsync(2), or close().

       See NOTES for a discussion of why close() should not be retried after an error.
```

extrait de `man -s2 close`

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 **obligatoire** de vérifier chaque retour d'appel système

Erreurs associées à un appel système (2/2)

Dealing with error returns from close()

A careful programmer will check the return value of **close()**, since it is quite possible that errors on a previous **write**(2) operation are reported only on the final **close()** that releases the open file description. Failing to check the return value when closing a file may lead to silent loss of data. This can especially be observed with NFS and with disk quota.

Note, however, that a failure return should be used only for diagnostic purposes (i.e., a warning to the application that there may still be I/O pending or there may have been failed I/O) or remedial purposes (e.g., writing the file once more or creating a backup).

Retrying the **close()** after a failure return is the wrong thing to do, since this may cause a reused file descriptor from another thread to be closed. This can occur because the Linux kernel always releases the file descriptor early in the close operation, freeing it for reuse; the steps that may return an error, such as flushing data to the filesystem or device, occur only later in the close operation.

Many other implementations similarly always close the file descriptor (except in the case of **EBADF**, meaning that the file descriptor was invalid) even if they subsequently report an error on return from **close()**. POSIX.1 is currently silent on this point, but there are plans to mandate this behavior in the next major release of the standard.

A careful programmer who wants to know about I/O errors may precede **close()** with a call to **fsync**(2).

extrait de `man -s2 close`

Remarque

- Les étudiants d'INFO0601 sont des programmeurs prudents...

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

```
CONFORMING TO
       open(), creat() SVr4, 4.3BSD, POSIX.1-2001, POSIX.1-2008.

       openat(): POSIX.1-2008.

       The  O_DIRECT,  O_NOATIME, O_PATH, and O_TMPFILE flags are Linux-specific.  One must define _GNU_SOURCE to ob-
       tain their definitions.

       The O_CLOEXEC, O_DIRECTORY, and O_NOFOLLOW flags are not specified  in  POSIX.1-2001,  but  are  specified  in
       POSIX.1-2008.   Since  glibc  2.12, one can obtain their definitions by defining either _POSIX_C_SOURCE with a
       value greater than or equal to 200809L or _XOPEN_SOURCE with a value greater than or equal to 700.   In  glibc
       2.11 and earlier, one obtains the definitions by defining _GNU_SOURCE.

       As  noted  in  feature_test_macros(7),  feature  test  macros  such  as  _POSIX_C_SOURCE,  _XOPEN_SOURCE,  and
       _GNU_SOURCE must be defined before including any header files.
```

Extrait de `man -s2 open`

À proscrire en INFO0601

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

Utilisation d'options de compilation de *gcc*

- `-Werror` : tous les avertissements deviennent des erreurs
- `-Wall` : affiche tous les avertissements

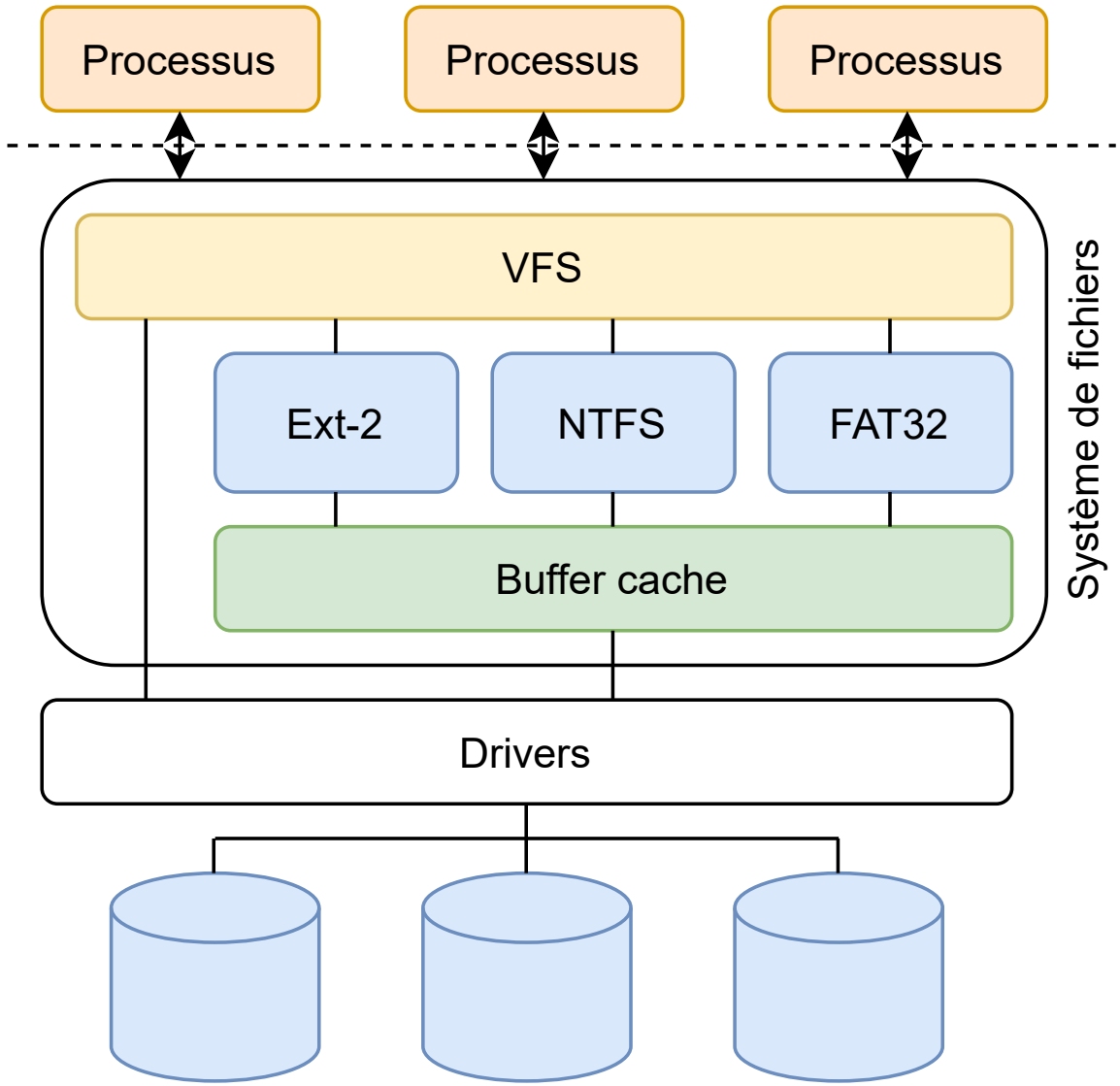
Obligatoire pour tous les codes fournis en INFO0601

N'oubliez pas de tester vos codes sur la VM, les options sont parfois différentes

Le VFS (1/2)

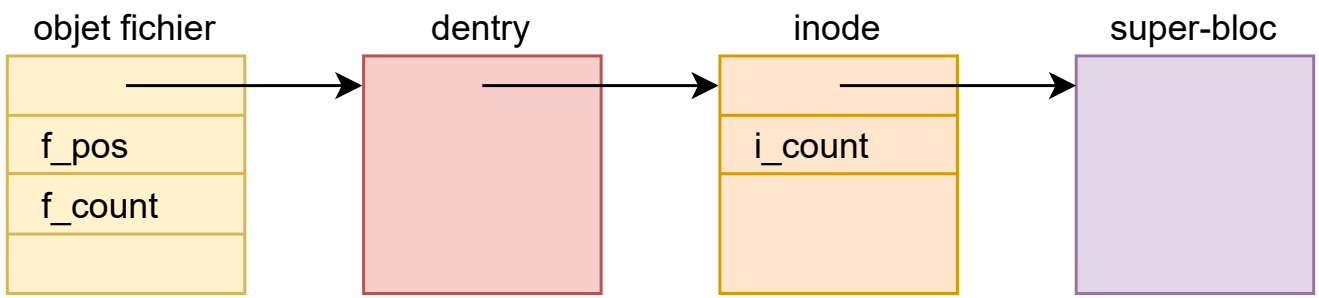
- Nécessité de supporter de multiples systèmes de fichiers
- *Linux* propose le VFS pour *Virtual File System* :
  ↪ Couche logicielle dans le noyau
- Permet de masquer à l'utilisateur les différences des systèmes de fichiers
- Mise en cache de différentes informations :
  ↪ *Buffer Cache* (blocs du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

Le VFS (2/2)


Processus
Processus
Processus
VFS
Ext-2
NTFS
FAT32
Buffer cache
Système de fichiers
Drivers

Les différentes structures (1/4)

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



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 :
  $\hookrightarrow$ 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

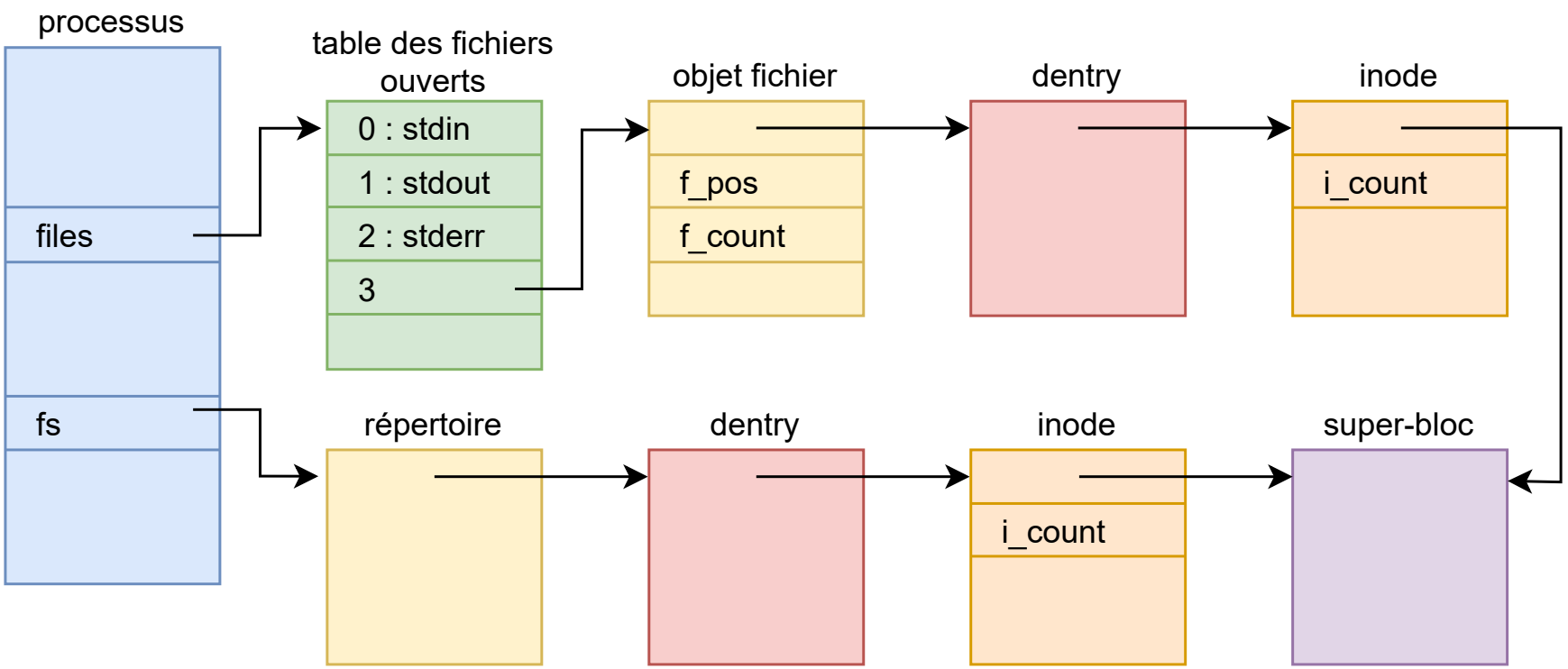
Les différentes structures (4/4) : fichier et dentry

- 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* :
  $\hookrightarrow$ 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 :
  $\hookrightarrow$ 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 :
  $\Longrightarrow$ Si c'est différent de `taille`, on est à la fin du fichier

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

Visualisation du contenu du fichier

- Données stockées dans le fichier au format binaire
- Édition de `toto.bin` avec un éditeur :
  ↪ Caractères illisibles (en fonction de la valeur saisie), sauf pour les chaines de caractères
- Éditeur hexadécimal graphique
  ↪ Exemple : `https://hexed.it/` (éditeur en ligne)
- Commande *Linux* `od`
  ↪ Exemple : `od -t x1 toto.bin`

Exemple avec une saisie de la valeur 12

```
0000000 0c 00 00 00
0000004
```

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 "`0`"

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'

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)

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 ?