Les primitives du système de fichiers

Zied Bouyahia

Groupe II1-F

La gestion des erreurs : perror()

- Tous les appels système retournent une valeur différente de 0 si une erreur survient
- L'appel "**open**()" peut échouer pour plusieurs raisons
  - "**errno**", est une variable globale qui contient le code numérique de l'erreur du dernier appel système
  - "**perror**()" est une fonction qui décrit les erreurs des appels systèmes
- Chaque processus détient une variable globale "errno" initialisée à 0 lors de sa création .
- "`/usr/include/sys/errno.h`" contient une liste des codes d'erreurs prédéfinies
- ↘ Exemples:

```
#define   EPERM     1   /* Not  owner */
#define   ENOENT    2   /* No such file or directory */
#define   ESRCH     3   /* No such process */
#define   EINSR     4   /* Interrupted System call */
#define   EIO       5   /* I/O error */
```

La gestion des erreurs : perror()

- Un appel système réussi ne modifie pas la valeur courante de errno
- Un appel qui échoue modifie la valeur de errno
- Pour accéder au contenu de errno depuis votre programme inclure “errno.h”

void perror(char* str)

- “perror()” affiche la chaîne str, suivie par “:” et une descritpion de la dernière erreur rencontrée lors d’un A.S.
- S’il n’y a pas d’erreurs elle affiche la chaîne “Error 0”

Remarque

“perror()” n’est pas un A.S. mais plutôt une routine de la bibliothèque standard de C.

La gestion des erreurs : perror()

```
$ cat showErrno.c

    #include <stdio.h>
    #include <sys/file.h>
    #include <errno.h>
    main()
    {
       int fd;
       /* Open a nonexistent file to cause an error */
       fd = open("nonexist.txt", O_RDONLY);
       if ( fd==-1 ) /* fd == -1 , an error occurred */
         {
            printf( "errno = %d \n", errno );
            perror("main");
          }
       fd=open( "/", O_WRONLY );  /* Force a different error */
       if ( fd== -1 )
         {  printf("errno=%d\n", errno);
             perror("main"); }
```

La gestion des erreurs : perror()

```
/* Execute a successful system call */
        fd = open("nonexist.txt", O_RDONLY | O_CREAE, 0644 );
        printf("errno=%d\n", errno); /* Display after successful
       call */
        perror("main");
        errno=0;  /* Manually reset error variable */
        perror("main");
    }
```

```
$ showErrno
errno=2
main: No such file or directory
errno=21
main: Is a directory
errno=21
main: Is a directory
main: Error 0
```

Les primitives de gestion du S.F

- Permettent de manipuler tous les types de fichiers, répertoires et même les fichiers spéciaux:
  - disques
  - terminaux
  - périphériques
  - IPC (tubes, sockets)

- L'appel système open() est utilisé initialement pour accéder au fichier (ou le créer dans certains cas)

- Si l'appel est réussi le retour est un entier (positif est "petit") qui sera utilisé dans le reste des opérations sur le fichier sinon le retour est une erreur (à gérer)

Une séquence “typique” d’appels

```
1. int  fd;                    /*  descripteur du fichier */
2. …
3. fd = open( fileName, … );  /* Ouvrir un fichier, le retour
                                 est un descripteur de
   fichier */
4. if ( fd==-1 )  {  /* Gérer l’erreur en cas d’échec */ }
5. …
6. fcntl( fd, … ); /* Ajuster eacertains paramètres */
7. …
    1. read( fd, … );  /* Lecture */
    2. …
    3. write( fd, …); /* Ecriture */
    4. …
    5. lseek( fd, …); /* Recherche*/
    6. …
8. close(fd);  /* Fermer le fichier et libérer le
   descripteur*/
```

Descripteur de fichier

- Lorsqu'un processus n'a plus besoin de l'accès à un fichier, il faut le fermer en utilisant l'appel close()
- Un fichier est fermé automatiquement lorsque le processus l'ayant ouvert est terminé

Mais, ça n'excuse pas le fait de laisser ouvert un fichier don't on n'a pas besoin

- Les descripteur de fichier sont affectés séquentiellement à partir de 0

Valeur du d.f.	Fichier correspondant
0	standard input
1	standard output
2	standard error

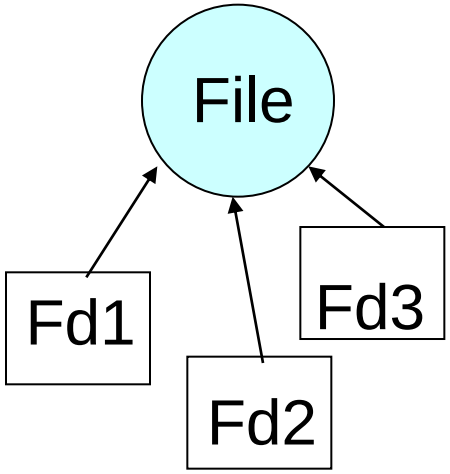
Descripteur de fichier

- La fonction printf() correspond à write(1,…)
- La fonction scanf() correspond à un read(0,…)

- Un même fichier peut être ouvert plusieurs fois (simultanément?) et donc il peut avoir plusieurs descripteurs
- Chaque descripteur a ses propres caractéristiques

  Exemple Un pointeur sur la position dans le fichier (à la lecture ou à l'écriture)

- Lorsqu'un descripteur est créé, son pointeur est positionné sur le premier caractère et il est mis à jour au fur et à mesure que le le processus lit ou écrit dans le fichier



Ouvrir un fichier : open()

int **open**(char* *fileName*, int *mode*[, int *permissions*])

"open()" ouvre ou crée un fichier pour lecture et/ou écriture

fileName	chemin absolu ou relatif du fichier,
mode	mode (read write) flags + flags divers
permissions	un nombre encodant les droits d'accès au fichier

read/write flags:

FLAG	Signification
O_RDONLY	Open for read only.
O_WRONLY	Open for write only.
O_RDWR	Open for both read and write.

Flags divers:

FLAG	Signification
O_APPEND	Positionne le pointeur à la fin du fichier avant chaque appel write()
O_CREAT	Si le fichier n'existe pas, créer le fichier et affecter à ownerID le PID
O_EXCL	Si O_CREAT est utilisé et le fichier existe open() échoue
O_TRUNC	Si le fichier existe, il est vidé de son contenu (tronqué)

Pour créer un fichier utiliser O_CREAT dans les mode flags et fournir les droits d'accès initiaux (en valeur octale)

↘ Exemple

```
sprintf( tmpName, ".fic.%d", getpid() );  /* nom aléatoire */

tmpfd = open( tmpName, O_CREAT | O_RDWR, 0600);
if ( tmpfd== -1 ) perror("création de fichier");
```

Pour ouvrir un fichier utiliser les *mode flags* uniquement

```
fd = open( fileName, O_RDONLY );
if ( fd== -1 ) fatalError();
```

Lire dans un fichier : read()

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

"read()" copie *count* octets depuis le fichier référencé par le descripteur de fichier fd dans le tampon (buffer) *buf*.

Si on lit un seul caractère (octet) à la fois
⇨On aura un très grand nombre d'appel système
⇨Ce qui ralentit les performances du système

Pour lire "BUFFER_SIZE" octet à la fois

```
charsRead = read( fd, buffer, BUFFER_SIZE );
if ( charsRead == 0 ) break;  /* EOF */
if ( charsRead == -1 ) perror();  /* Error */
```

Ecrire dans un fichier: write()

Pour écrire dans un fichier utiliser l'appel système "write()"

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

"write()" copie *count* octets depuis un tampon (buffer) *buf* dans le fichier référencé par le descripteur de fichier *fd*.

Si le flag O_APPEND est utilisé, le curseur est positionné à la fin du fichier avant chaque appel write()

"write()" retourne le nombre d'octets écrits en cas de succès et -1 en cas d'échec

Exemple

```
if ( standardInput )
   {
      charsWritten = write(tmpfd, buffer, charsRead );
      if ( charsWritten != charsRead ) fatalError();
   }
```

Se déplacer dans un fichier : lseek()

off_t lseek(int *fd*, off_t *offset*, int *mode*)

"lseek()" permet de changer la position courante dans le fichier référencé par le descripteur

fd : descripteur de fichier,
offset : long integer (déplacement),
mode : comment interpréter l'offset (le déplacement).

mode	interprétation
SEEK_SET	Début du fichier
SEEK_CUR	Position courante dans le fichier.
SEEK_END	Fin du fichier.

Se déplacer dans un fichier : lseek()

Le nombre d'octets à lire est calculé calculé en soustrayant la valeur de décalage du début de la ligne suivante de la valeur de décalage du début de la ligne actuelle:

```
lseek( fd, lineStart[i], SEEK_SET );  /* Trouver et lire la ligne */
charsRead = read ( fd, buffer, lineStart[i+1] - lineStart[i] );
```

Pour connaître votre position actuelle sans déplacer, utilisez une valeur de décalage de zéro par rapport à la situation actuelle:

```
currentOffset = lseek( fd, 0, SEEK_CUR );
```

Fermer un fichier: “close()”

Utiliser l’appel “close()” pour libérer le descripteur de fichier alloué au fichier ouvert

int close(int fd)

Si fd est le dernier descripteur de fichier associé à un fichier ouvert les ressources allouées à ce fichier sont libérées

- En cas de succès, “close()” retourne 0;
- En cas d’échec il retourne -1.
- Exemple

```
close(fd);   /* Close input file */
```

Supprimer un fichier: unlink()

int unlink(const char* *fileName*)

"unlink()" supprime les liens hard depuis le nom *fileName* au fichier
Si *fileName* est le dernier lien au fichier, les ressources allouées à ce fichier sont libérées

Un fichier exécutable peut supprimer ses propres liens et continuer son exécution (pourquoi d'ailleurs?)

- En cas de succès "unlink()" retourne 0;
- Sinon il retourne -1.

```
If ( standardInput ) unlink( tmpName );  /* Remove temp file */
```

Informations sur un fichier : stat(), lstat(), fstat()

On peut obtenir des informations sur un fichier en utilisant "stat()"

```
int  stat( const char* name, struct stat* buf )
int lstat( const char* name, struct stat* buf )
int fstat( int fd, struct stat* buf )
```

"stat()" remplit le tampon *buf* avec les informations concernant le fichier *name*.

La structure "stat" est définie dans "/usr/include/sys/stat.h".

"lstat()" retourne des informations concernant un lien symbolique plutôt que le fichier auquel il réfère

"fstat()" fait la même chose que "stat()", sauf qu'elle prend le descripteur de fichier comme premier paramètre de l'appel

```
result = stat( fileName, &statBuf );
```

Informations sur un fichier : stat(), lstat(), fstat()

La structure stat contient les champs suivant:

Champs	rôles
st_dev	device number
st_ino	n° inode
st_mode	permissions
st_nlink	#liens durs
st_uid	user ID
st_gid	group ID
st_size	taille
st_atime	date dernier accès
st_mtime	date dernière modification
st_ctime	date dernier changement d'état

On peut décoder les champs st_time, st_mtime, st_ctime en utilisant les routines asctime() et localtime()

Informations sur un fichier : stat(), lstat(), fstat()

Quelques macros prédéfinies sont dans “/usr/include/sys/stat.h”

➔Prend en argument st_mode et retourne true (1) pour les types suivant

MACRO	TRUE FOR FILE TYPE
S_IFDIR	répertoire
S_IFCHR	fichier spécial type caractère
S_IFBLK	fichier spécial type bloc
S_IFREG	fichier ordinaire
S_IFFIFO	pipe

Informations sur un répertoire: getdents()

int getdents(int fd, struct dirent* buf, int structSize)

Lit le répertoire spécifié par fd depuis sa position actuelle et remplit la structure pointée par buf

La structure “dirent” est definie dans “/usr/include/sys/dirent.h” et contient les champs suivants:

- d_ino c'est le numéro du i-noeud.
- d_off c'est la position relative de la prochaine entrée du répertoire.
- d_reclen c'est la longueur de la structure d'une entrée du répertoire.
- d_name c'est le nom du fichier.

Appels systèmes supplémentaires

Nom	Rôle
chown	changer le propriétaire et/ou le groupe d'un fichier
chmod	changer les droits d'un fichier
dup / dup2	dupliquer le descripteur de fichier
fchown	chown
fchmod	chmod
fcntl	Donner accès à diverses caractéristique de fichier
ftruncate	truncate
ioctl	Contrôler un périphérique
link	Créer un lien dur
mknod	Créer un fichier spécial
sync	Purger les tampons sur disque
truncate	tronquer un fichier
fflush	vider le contenu d'un fichier

Dupliquer un descripteur de fichier: dup() and dup2()

- “dup()”, “dup2()”
  sont utilisés pour dupliquer un descripteur de fichier:

```
int dup( int oldFd )
int dup2( int oldFd, int newFd )
```

“dup()” trouve le plus petit descripteur libre et le pointe vers le même fichier pointé par *oldFd*

“dup2()” ferme *newFd* s’il est actuellement utilisé et le pointe vers le même fichier pointé par *oldFd*

- les descripteur original et copié partagent le même pointeur sur le fichier et les mêmes droits d’accès

- ces appels retournent l’index du nouveau descripteur en cas de succès et -1 sinon

- **Exemple: dup() and dup2()**

```
$ cat exdump.c
#include <stdio.h>
#include <fcntl.h>
main()
{
    int fd1, fd2, fd3;
    fd1 = open( “test”, O_RDWR | O_TRUNC );
    printf(“fd1 = %d”\n”, fd1 );
    write( fd1, “quoi “, 6 );
    fd2 = dup(fd1);  /* copie de fd1 */
    printf( “fd2=%d\n”, fd2);
    write( fd2, “de neuf ”, 3 );
    close(0);  /* fermer stdin */
    fd3 = dup(fd1);  /* une autre copie de fd1*/
    printf(“fd3 = %d\n”, fd3);
    write(0, “ les gars“, 4);
    dup2(3,2);  /* Dupliquer 3 sur 2 */
    write(2, “?\n”, 2 );
 }
```

```
$ exdump
fd1 = 3
fd2 = 4
fd3 = 0
$ cat test
  quoi de neuf les gars
$ -
```

Création de fichiers spéciaux mknod()

```
int  mknod( const char* fileName, mode_t  type, dev_t device )
```

Remarque: on peut utiliser mkdir() pour la création d'un répertoire

```
int  mkdir(const char *path,mode_t permissions)
```

Valeur	Signification
S_IFDIR	directory
S_IFCHR	character-oriented file
S_IFBLK	block-oriented file
S_IFREG	regular file
S_IFIFO	named pipe

Quelques appels système supplémentaires

```
int chown( const char* fileName, uid_t ownerId, gid_t  groupId )

int lchown( const char* fileName, uid_t ownerId, git_t  groupId )
/*link*/

int fchown( int fd, uid_t ownerId, gid_t groupId )

/*la valeur -1 dans un champs ne le change pas */
```

```
int chmod( const char* fileName, int mode )

int fchmod( int fd, mode_t mode );
```

```
int link( const char* oldPath, const char* newPath )
```