Gestion des I/O fichiers

Document de cours
– 1ère Partie –

Prof. Hicham GIBET TANI
(h.gibettani@uae.ac.ma)

Linux

Licence – Filière Sciences Mathématiques Informatiques (SMI)

Module – Programmation Système

Table des matières

Introduction:

La programmation système est souvent séparée de la programmation d'application. La raison principale est que les programmeurs système doivent avoir une connaissance approfondie du matériel et du système d'exploitation sur lesquels ils travaillent.

Cependant, passer de la programmation d'application à la programmation système (ou vice versa) n'est pas difficile. Il n'y a que quelques différences entre les bibliothèques utilisées et les appels effectués.

Malgré l'évolution extraordinaire de la programmation d'applications, la majorité du code des systèmes d'exploitations sur lesquels ces applications s'exécutent est toujours écrit au niveau système. Dont une grande partie de ces systèmes est en C, et subsiste principalement sur les interfaces fournies par la bibliothèque C et le noyau.

Appels Système :

La programmation système commence par les appels système. Les appels système (souvent abrégés syscalls) sont des invocations de fonctions effectuées à partir de l'espace utilisateur vers le noyau afin de demander un service ou une ressource du système d'exploitation. Par exemple: read( ) et write( ).

Linux implémente moins d'appels système que la plupart des autres noyaux de système d'exploitation. Par exemple, le nombre d'appels système de l'architecture « i386 » est d'environ 300, par rapport aux milliers d'appels systèmes sur Microsoft Windows.

La bibliothèque C :

La bibliothèque C (libc) est au cœur des applications Unix/Linux. La bibliothèque **C** est en jeu même lorsque vous programmez dans un autre langage. Elle est encapsulée par les bibliothèques de niveau supérieur et fournit les services système de base.

Sous Linux, le compilateur C standard est fourni par « GNU Compiler Collection (gcc) ». « gcc » était la version GNU de « cc ».

« gcc » est également le binaire utilisé pour appeler le compilateur C.

I/O de fichier avec les fonctions C standard :

Avant qu'on puisse lire ou écrire un fichier, ce fichier doit être ouvert.

Le noyau gère une liste de processus pour les fichiers ouverts, appelée la table de fichiers. Ce tableau est indexé via des entiers non négatifs appelés « descripteurs de fichiers » (souvent abrégés fds).

Chaque entrée de la liste contient des informations d'un fichier ouvert, y compris un pointeur vers une copie en mémoire de l'i-node du fichier et les métadonnées associées, telles que la position du fichier et les modes d'accès.

Types de Fichier :

Lorsque vous traitez des fichiers, vous devez connaître deux types de fichiers:

- **Fichiers textes:** Les fichiers textes sont les fichiers normaux. Vous pouvez facilement créer des fichiers texte à l'aide des éditeurs simples de texte tels que « vi, nano… ». Lorsque vous ouvrez ces fichiers, vous verrez tout le contenu du fichier sous forme de texte brut. Vous pouvez facilement modifier ou supprimer le contenu.

- **Fichiers binaires:** Les fichiers binaires sont principalement les fichiers .bin de votre ordinateur. Au lieu de stocker des données en texte brut, ils les stockent sous forme binaire (0 et 1). Ils peuvent contenir une grande quantité de données, ils ne sont pas lisibles facilement et offrent une meilleure sécurité que les fichiers texte.

Les Opérations Fichiers :

En C, vous pouvez effectuer quatre opérations majeures sur les fichiers (texte ou binaire):

- Création d'un nouveau fichier,
- Ouverture d'un fichier existant,
- Fermeture d'un fichier,
- Lecture et écriture d'informations dans un fichier.

Gestion des Fichiers :

Lorsque vous travaillez sur des fichiers, vous devez déclarer un pointeur de type fichier. Cette déclaration est nécessaire pour la communication entre le fichier et le programme.

```
FILE    *fptr;
```

L'ouverture d'un fichier s'effectue à l'aide de la fonction **fopen()** définie dans l'en-tête du fichier **<stdio.h>**.

La syntaxe d'ouverture d'un fichier dans I/O standard est la suivante:

```
fptr = fopen ("fileopen", "mode");
```

Exemple:

La première fonction crée un nouveau fichier avec le nom « **fichier.txt** » et l'ouvre en écriture selon le mode ‘w’ (Le mode écriture vous permet de créer et de modifier (écraser) le contenu d’un fichier).

```
fopen("/home/user1/fichier.txt","w");
```

La deuxième fonction ouvre un fichier existant pour la lecture en mode binaire ‘rb’. Le mode de lecture vous permet uniquement de lire le fichier.

```
fopen("/home/user1/fichier.txt.bin","rb");
```

Mode	Signification du Mode	En cas d’absence du fichier
r	Ouvrir pour lecture	fopen() renvoie NULL.
rb	Ouvrir pour lecture en mode binaire	fopen() renvoie NULL.
w	Ouvrir en écriture	Le fichier sera créée. Si le fichier existe son contenu sera écrasé.
wb	Ouvrir en écriture en mode binaire	Le fichier sera créée. Si le fichier existe son contenu sera écrasé.
a	Ouvrir pour ajouter des données	Le fichier sera créée.
ab	Ouvrir pour ajouter des données en binaire	Le fichier sera créée.
r+	Ouvrir pour lecture et écriture	fopen() renvoie NULL.
rb+	Ouvrir pour lecture et écriture en binaire	fopen() renvoie NULL.
w+	Ouvrir pour lecture et écriture	Le fichier sera créée. Si le fichier existe son contenu sera écrasé.

wb+	Ouvrir pour lecture et écriture en binaire	Le fichier sera créée. Si le fichier existe son contenu sera écrasé.
a+	Ouvrir pour lecture et ajout	Le fichier sera créée.
ab+	Ouvrir pour lecture et ajout en binaire	Le fichier sera créée.

Fermeture d'un Fichier:

Le fichier (texte ou binaire) doit être fermé après lecture/écriture.

La fermeture d'un fichier s'effectue à l'aide de la fonction fclose().

```
fclose(fptr);
```

fptr: est le pointeur associé au fichier à fermer.

Lecture et écriture d'un Fichier Texte :

Pour lire et écrire dans un fichier texte, nous utilisons les fonctions fprintf( ) et fscanf( ).

Ce sont les versions printf( ) et scanf( ) pour les fichiers. La seule différence est que fprint( ) et fscanf( ) attendent un pointeur sur la structure FILE.

Exemple d'écriture :

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int num;
  FILE *fptr;
  fptr = fopen("/root/fileFPL","w");
  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }
  printf("Enter num: ");
  scanf("%d",&num);
  fprintf(fptr,"%d",num);
  fclose(fptr);
  return 0;
}
```

Exemple de lecture :

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int num;
  FILE *fptr;
  if ((fptr = fopen("/root/fileFPL","r")) == NULL){
     printf("Error! opening file");
exit(1);
  }
  fscanf(fptr,"%d", &num);
  printf("Value of n=%d", num);
  fclose(fptr);
  return 0;
}
```

Lecture et écriture d'un Fichier Binaire :

Les fonctions **fread( )** et **fwrite( )** sont utilisées pour lire et écrire respectivement dans le cas des fichiers binaires.

Pour écrire dans un fichier binaire, vous devez utiliser la fonction **fwrite( )**. La fonction prend quatre arguments:

- L'adresse des données à écrire sur le disque.
- La taille des données à écrire sur le disque.
- Un nombre pour le type de données.
- Un pointeur vers le fichier.

```
fwrite(addresse, taille, NombreDonnées, PointeurFichier);
```

Exemple d'écriture :

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("/root/ProgSys/test.bin","wb")) == NULL){
```

```
      printf("Error! opening file");
      exit(1);
  }
for(n = 1; n < 5; ++n)
  {
    num.n1 = n;
    num.n2 = 5*n;
    num.n3 = 5*n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, fptr);
  }
  fclose(fptr);
  return 0;
}
```

Exemple de lecture :

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("/root/ProgSys/test.bin","rb")) == NULL){
      printf("Error! opening file");
      // Program exits if the file pointer returns NULL.
      exit(1);
  }
for(n = 1; n < 5; ++n)
  {
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
  }
  fclose(fptr);

  return 0;
}
```

La gestion des I/O de fichier avec les appels système :

L'Appel Système open( ) :

Un fichier peut aussi être ouvert avec l'appel système open( ) qui permet l'obtention d'un descripteur de fichier:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
                    int open (const char *name, int flags);
                    int open (const char *name, int flags, mode_t mode);
```

Mode d'ouverture (flags) :

Le mode d'ouverture (flag) avec l'appel système « **open( )** » permet la spécification du mode d'accès: lecture **O_RDONLY**, écriture **O_WRONLY** ou bien les deux **O_RDWR**.

```
int fd;
fd = open ("/home/user1/file.txt", O_RDONLY);
if (fd == −1)
                    /* error */
```

Les options (flag) :

flag	Description
O_APPEND	La position du fichier sera mise à jour pour pointer vers la fin du fichier.
O_CREAT	Si le nom du fichier n'existe pas, le noyau va le créer.
O_DIRECTORY	Si le nom n'est pas un répertoire, l'appel open( ) échouera.
O_EXCL	Lorsqu'il est utilisé avec **O_CREAT**, il entraînera l'échec de l'appel open( ) si le nom fichier existe déjà.
O_LARGEFILE	Le fichier donné sera ouvert à l'aide du mode 64 bits, permettant la manipulation des fichiers plus larges que deux (2) giga-octets.
O_SYNC	Aucune opération d'écriture ne se termine tant que les données n'ont pas été physiquement écrites sur le disque.
O_TRUNC	Si le fichier existe, normal et le mode d'ouverture permet l'écriture, le fichier sera réduit à une longueur nulle.

Les permissions des nouveaux fichiers :

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

Lorsqu'un fichier est créé, l'argument mode fournit les permissions du fichier. L'argument mode peut contenir la notation octale des permissions, comme par exemple 0644 (le propriétaire peut lire et écrire, tout le monde ne peut que lire).

Permission	Description
S_IRWXU	Le propriétaire a le droit de lire, écrire et exécuter.
S_IRUSR	Le propriétaire a le droit de lire.
S_IWUSR	Le propriétaire a le droit d'écrire.
S_IXUSR	Le propriétaire a le droit d'exécuter.
S_IRWXG	Le groupe propriétaire a le droit de lire, écrire et exécuter.
S_IRGRP	Le groupe propriétaire a le droit de lire
S_IWGRP	Le groupe propriétaire a le droit d'écrire.
S_IXGRP	Le groupe propriétaire a le droit d'exécuter.
S_IRWXO	Les autres ont le droit de lire, écrire et exécuter.
S_IROTH	Les autres ont le droit de lire.
S_IWOTH	Les autres ont le droit d'écrire.
S_IXOTH	Les autres ont le droit d'exécuter.

Par exemple, le code suivant ouvre un fichier pour l'écriture. Si le fichier n'existe pas (umask de 022) il sera créé avec les permissions 0644 (même si l'argument mode spécifie 0664). Enfin, si le fichier existe, il sera réduit à une longueur nulle:

```
int fd;
fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, S_IWUSR | S_IRUSR | S_IWGRP |
S_IRGRP | S_IROTH);
if (fd == −1)
                    /* error */
```

NB: On peut utiliser:

```
fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, 0664)
```

L'Appel Système creat( ) :

Lors de l'ouverture d'un fichier, les indicateurs « **O_WRONLY | O_CREAT | O_TRUNC** » sont souvent utiliser. Pour cette raison, un appel système existe pour grouper les trois indicateurs:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
                    int creat (const char *name, mode_t mode);
```

L’appel Système read( ) :

Le mécanisme le plus simple pour la lecture des fichiers est l'appel système read( ).

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

Pour assurer la lecture total d’un fichier:

```
ssize_t ret;
while (len != 0 && (ret = read (fd, buf, len)) != 0) {
                        if (ret == −1) {
                           if (errno == EINTR)
                                   continue;
                           perror ("read");
                           break;
                        }
                        len -= ret;
                        buf += ret;
}
```

Écriture avec write( ) :

Afin d’écrire des données dans un fichier, on utilise l’appel système write( ):

```
#include <unistd.h>

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

Exemple:

```
const char *buf = “Mon code est solide !";

ssize_t nr;

        nr = write (fd, buf, strlen (buf));

        if (nr == −1)

        /* error */
```

Le mode O_APPEND :

Les écritures se produisent à la fin du fichier. Le mode O_APPEND garantit que la position du fichier est toujours définie à la fin du fichier, afin que toutes les écritures soient toujours ajoutées, même lorsqu'il existe plusieurs processus qui veulent écrire sur le même fichier.

Fermeture de fichier close( ) :

Une fois qu'un programme a fini le travail sur un descripteur de fichier, il peut détacher le descripteur de fichier du fichier via l'appel système close( ):

```
#include <unistd.h>

int close (int fd);
```

Les Appels Systèmes fsync( ) et fdatasync( ) :

La méthode la plus simple pour s'assurer que les données ont atteint le disque se fait via l'appel système fsync( ):

```
#include <unistd.h>

int fsync (int fd);
```

Pour utiliser l'appel système fsync( ) le descripteur de fichier « fd » doit être ouvert pour écriture.

Dans le cas où le disque utilise du cache pour l'écriture, l'appel système fdatasync( ) doit être utiliser:

```
#include <unistd.h>

int fdatasync (int fd);
```

Chercher en utilisant lseek( ) :

Normalement, les I/O se produisent de façon linéaire dans un fichier. Cependant, certaines applications souhaitent sauter dans un fichier, offrant un accès aléatoire plutôt que linéaire. L'appel système lseek( ) est fourni pour définir la position d'un descripteur de fichier sur une valeur donnée.

```
#include <sys/types.h>

#include <unistd.h>

off_t   lseek(int fd, off_t pos, int origin);
```

Le comportement de lseek( ) dépend de l'argument « origin », qui peut être l'un des suivants:

- **SEEK_CUR:** La position actuelle du fichier « fd » est définie sur sa valeur actuelle plus (+) la valeur du « pos ».
- **SEEK_END:** La position actuelle du fichier « fd » est définie sur la longueur actuelle du fichier plus (+) la valeur du « pos ».
- **SEEK_SET:** La position actuelle du fichier « fd » est définie sur pos.

```
off_t ret;

ret = lseek (fd, (off_t) 1825, SEEK_SET);

if (ret == (off_t) −1)

/* error */
```