Module 10
Développement C++ avec Linux
Appels systèmes sous Linux

Table des matières

Compilateur GCC et G++

Il existe plusieurs compilateurs qui peuvent compiler des programmes écrit en plusieurs langages de programmation. Il existe, entre autres :

le compilateur gcc : pour les fichiers sources en langage C
le compilateur g++ : pour les fichiers sources en langage C++

Les deux compilateurs ont des syntaxes très similaire.

Compilation d'un fichier source en langage C avec gcc

Supposons un fichier source qui se nomme « Exemple.c » et qui contient le code suivant :

```
#include <stdio.h>

int main()
{
  printf("Salut à toi");  // Équivalent en langage C du cout en C++
  return 0;
}
```

La syntaxe pour compiler avec gcc est la suivante : gcc -o nom_executable Nom_Fichier_Source.c

Ainsi, avec le fichier précédent Exemple.c, la commande pour compiler devient :

```
gcc -o exemple exemple.c
```

Si tout est bien écrit, le compilateur procède à créer un fichier objet « exemple.o » et utilise ce fichier pour faire l'édition des liens et créer le fichier exécutable.

Compilation d'un fichier source en langage C++ avec g++

Supposons un fichier source en langage C++ qui se nomme « Exemple.cpp » et qui contient le code suivant :

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Salut à toi' << endl;
  return 0;
}
```

La syntaxe pour compiler avec le compilateur g++ est la suivante :

```
g++  -o  nom_executable  Nom_Fichier_Source.cpp
```

Ainsi, avec le fichier précédent Exemple.cpp, la commande g++ devient :

```
g++  -o  exemple  exemple.cpp
```

Exécuter un programme

Pour exécuter un programme qui a été compilé avec gcc ou g++, il faut utiliser la même syntaxe que celle que nous avions utilisée pour exécuter un script. Revoici cette syntaxe :

```
./Nom_Executable
```

Ainsi, dans l'exemple précédent, le fichier exécutable se nomme « exemple ». On peut donc lancer l'exécution en tapant : « ./exemple ».

Compiler un programme contenant plusieurs fichiers sources en langage C++

Le code source se retrouvant dans plusieurs fichiers sources peut être compilé en passant chaque fichier source (.cpp) au compilateur g++. Voici un exemple :

Supposons les fichiers sources « ficher1.cpp » et « fichier2.cpp » alors ces fichiers peuvent être compilés en utilisant la commande suivante :

```
g++  -o  Nom_Executable  fichier1.cpp  fichier2.cpp
```

Appels systèmes

Dans le cours, nous avons utilisé des commandes consoles pour effectuer certaines opérations sur les fichiers. De même, nous avons réutilisé ces commandes en les intégrants dans des scripts bash pour pouvoir automatiser certaines tâches. Les commandes consoles que vous utilisez dans ces situations font appels à des fonctions de bibliothèque particulière. L'ensemble de ces fonctions présente dans cette bibliothèque représente ce qu'on nomme les appels systèmes.

Un appel système est implanté au sein du noyau Linux. Lorsqu'un programme effectue un appel système, les options sont mis en forme et transférés au noyau qui prend en charge l'exécution jusqu'à la fin de l'appel. Un appel système n'est pas identique à un appel de fonction classique et une procédure spécifique est nécessaire pour transférer le contrôle au noyau.

Les appels système Linux constituent l'interface de base entre les programmes et le noyau Linux. À chaque appel correspond une opération ou une fonctionnalité de base. Certains appels sont très puissants et influent au niveau du système. Par exemple, il est possible d'éteindre le système ou d'utiliser des ressources système tout en interdisant leur accès aux autres utilisateurs. De tels appels ne sont utilisables que par des programmes s'exécutant avec les privilèges super-utilisateur.

En gros, les appels systèmes sont des fonctions écrites en langage C qui permettent d'utiliser le système de fichier et d'autres ensemble du système d'exploitation. En fait, ce sont les mêmes fonctions que le système d'exploitation utilise.

Quelques appels systèmes importants pour la gestion des fichiers

Renommer un fichier ou un répertoire – Fonction « rename »

rename - Changer le nom ou l'emplacement d'un fichier.

Entête : stdio.h

Syntaxe : int rename(const char ****oldpath*, const char ****newpath***);**

DESCRIPTION

rename renomme un fichier, en le déplacant vers un autre répertoire si besoin est.
Tous les autres liens matériels (créés avec **link**) restent inchangés.
Si *newpath* existe deja, il sera écrasé (avec quelques restrictions, voir le paragraphe ERREURS), de manière à ce qu'à aucun moment, un autre processus tentant d'accéder à *newpath* ne le voit absent. Si l'opération échoue pour une raison quelconque, **rename** garantit la présence d'une instance de *newpath* en place.
Néanmoins, pendant un écrasement, il se trouve un court instant pendant lequel à la fois *oldpath* et *newpath* font référence au fichier.
Si *oldpath* correspond à un lien symbolique, le lien est renommé; si *newpath* correspond à un lien symbolique, le lien est écrasé.

VALEUR RENVOYÉE

rename renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas *errno* contient le code d'erreur.

ERREURS

EISDIR
newpath est un répertoire existant mais *oldpath* n'est pas un répertoire
EXDEV
oldpath et *newpath* ne sont pas sur le même système de fichiers.
ENOTEMPTY ou **EEXIST**

newpath est un répertoire non vide (contient autre chose que "." et "..").

EBUSY

Le renommage a échoué car *oldpath* ou *newpath* est un répertoire utilisé par un processus (peut-être comme répertoire de travail, ou comme répertoire racine, ou ouvert en lecture), ou il est utilisé par le système (comme point de montage par exemple). Le système a donc considéré qu'il y avait une erreur. (Notez qu'il n'est pas indispensable de renvoyer EBUSY dans un tel cas - rien n'empêche d'effectuer le renommage malgré tout - mais il est permis de retourner EBUSY si le système n'arrive pas à gérer une telle situation).

EINVAL

Une partie du nouveau chemin contient en préfixe l'ancien chemin, ou plus généralement, un répertoire ne peut pas être déplacé dans ses propres sous-répertoires.

EMLINK

oldpath a déjà un nombre maximal de liens, ou bien c'est un répertoire, et le répertoire contenant *newpath* a le nombre maximal de liens.

ENOTDIR

Un élément utilisé dans *oldpath* ou *newpath* n'est pas un répertoire, ou *oldpath* est un répertoire et *newpath* existe mais n'est pas un répertoire.

EFAULT

oldpath ou *newpath* pointent en dehors de l'espace d'adressage accessible.

EACCES

Les droits d'écriture dans le répertoire contenant *oldpath* ou *newpath* ne sont pas autorisés pour l'UID effectif du processus, ou bien un des répertoires de *oldpath* ou *newpath* ne permet pas le parcours, ou encore *oldpath* était un répertoire et ne permet pas l'écriture (nécessaire pour mettre à jour les entrées **`.'** et **`..'**).

Exemple :

```
#include <string>
#include <iostream>
#include <stdio.h>  // Pour la fonction rename

using namespace std;

int main(int argc, char *argv[])
{

        if (argc != 3)
        {
                cout << "Mauvais nombre de parametre" << endl;
                cout << "chnom ancien_nom  nouveau_nom" << endl;
        }
        else
        {
                string NouveauNom = argv[2];
                cout << NouveauNom.c_str();

                if (rename(argv[1], NouveauNom.c_str()) == 0)
                {
                        cout << "Changement de nom réussi" << endl;
                }
                else
                {
                        cout << "Erreur changement de nom" << endl;
                }
        }
}
```

Capter les erreurs :

Exemple :

```
#include <errno.h> // Pour capter les erreurs
#include <stdio.h>  //Pour le rename

#include <string>


using namespace std;

int main(int argc, char *argv[])
{

   if (argc != 3)
        {
                cout << "Mauvais nombre de parametre" << endl;
                cout << "chnom ancien_nom  nouveau_nom" << endl;
        }
        else
        {
                string NouveauNom = argv[2];

                if (rename(argv[1], NouveauNom.c_str()) == 0)
                {
                        cout << "Changement de nom réussi" << endl;
                }
                else
                {
                        switch(errno)
                        {
                                case ENOENT:
                                        cout << "Fichier ou répertoire source non existant!";
                                break;
                        }
                }
        }
}
```

Modifier les permissions d'accès à un fichier – Fonction « chmod »

chmod - Modifier les permissions d'accès à un fichier.

Entête **<sys/types.h>**
<sys/stat.h>

Syntaxe : **int chmod(const char** ******pathname***, mode_t** *mode***);**

DESCRIPTION
chmod change le mode d'accès du fichier *pathname passé en chaîne de caractère.*

Le mode est spécifié par un *OU* binaire (|) entre les éléments suivants (les nombres sont en octal) :

Valeur	Description
S_ISUID 04000	modification du numéro d'utilisateur (UID) à l'exécution
S_ISGID 02000	modification du numéro de groupe (GID) à l'exécution.
S_ISVTX 01000	positionner le sticky bit pour conserver le code du programme en mémoire après exécution.
S_IRUSR (S_IREAD) 00400	accès en lecture pour le propriétaire
S_IWUSR (S_IWRITE) 00200	accès en écriture pour le propriétaire
S_IXUSR (S_IEXEC) 00100	accès en exécution/parcours par le propriétaire
S_IRGRP 00040	accès en lecture pour le groupe
S_IWGRP 00020	accès en écriture pour le groupe
S_IXGRP 00010	accès en exécution/parcours pour le groupe
S_IROTH 00004	accès en lecture pour les autres
S_IWOTH 00002	accès en écriture pour les autres
S_IXOTH 00001	accès en exécution/parcours pour les autres

L'UID effectif du processus doit être nul (root) ou doit correspondre à celui du propriétaire du fichier.
Si l'UID effectif du processus n'est pas nul, et si le groupe du fichier ne correspond ni au GID effectif du processus, ni à l'un de ses éventuels groupes supplémentaires, le bit S_ISGID sera désactivé, mais cela ne créera pas d'erreur.
Suivant le type de système de fichiers, les bits Set-UID et Set-GID peuvent être effacés si un fichier est écrit. Sur certains système de fichiers, seul le Super-User peut positionner le Sticky-Bit, lequel peut avoir une signification spécifique (par ex: pour un répertoire, un fichier ne pourra y être effacé que par le propriétaire ou le Super-User).
Sur les systèmes de fichiers NFS, une restriction des autorisations d'accès aura un effet immédiat y compris sur les fichiers déjà ouverts, car les contrôles d'accès sont effectués sur le serveur, mais les fichiers sont maintenus ouverts sur le client. Par contre, un élargissement des autorisations peut ne pas être immédiat, si le client dispose d'un cache.

VALEUR RENVOYÉE
chmod et **fchmod** renvoient 0 s'ils réussissent, ou -1 en cas d'échec, auquel cas *errno* contient le code d'erreur.

ERREURS
Suivant le type de système de fichiers, différentes erreurs peuvent être renvoyées. Les plus courantes pour **chmod** sont :

EPERM
L'UID effectif ne correspond pas au propriétaire du fichier, et n'est pas nul.
EROFS
Le fichier se trouve sur un systeme de fichiers en lecture seule.
EFAULT
pathname pointe en dehors de l'espace d'adressage accessible.
ENAMETOOLONG
pathname est trop long.
ENOENT
Le fichier n'existe pas.
ENOMEM
Pas assez de mémoire pour le noyau.
ENOTDIR
Un élément du chemin d'accès n'est pas un répertoire.
EACCES
Le parcours d'un élément du chemin de recherche est interdit.
ELOOP
pathname contient une référence circulaire (à travers un lien symbolique)
EIO
Une erreur d'entrée / sortie bas-niveau s'est produite durant la modification de l'i-noeud.

Exemple :

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

#include <errno.h>>
#include <string>


using namespace std;

int main(int argc, char *argv[])
{


   if (chmod(argv[1], 0777) == 0)
   {
      cout << "Changement de droits réussi" << endl;
   }
   else
   {
          switch(errno)
          {
                case ENOENT:
                      cout << "Fichier ou répertoire source non existant!";
                break;
          }
   }
}
```

Créer un répertoire – Fonction « mkdir »

Entête : **<sys/types.h>**

Syntaxe : **int mkdir(const char ****pathname***, mode_t** *mode***);**

DESCRIPTION
mkdir crée un nouveau répertoire nommé *pathname*.

Mode :
mode spécifie les permissions à appliquer au répertoire.

Par exemple , pour créer un répertoire avec les droits de lecture seulement : 0400
Les valeurs sont les mêmes que la fonction chmod.

VALEUR RENVOYÉE
mkdir renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas *errno* contient le code d'erreur.

ERREURS

EPERM
Le système de fichiers contenant *pathname* ne permet pas la création de répertoires.
EEXIST
pathname existe déjà (pas nécessairement un répertoire). Ceci inclut le cas où *pathname* est un lien symbolique, pointant quelque part ou pas.
EFAULT
pathname pointe en-dehors de l'espace d'adressage accessible.
EACCES
Le répertoire parent n'autorise pas l'écriture au processus, ou l'un des répertoires de *pathname* n'autorise pas la consultation de son contenu.
ENAMETOOLONG
pathname trop long.
ENOENT
Un répertoire du chemin d'accès *pathname* n'existe pas ou est un lien symbolique pointant nulle part.
ENOTDIR
Un élément utilisé dans le chemin *pathname* n'est pas un répertoire.
ENOMEM
Pas assez de mémoire pour le noyau.
EROFS
pathname serait sur un système de fichier en lecture seule.
ELOOP
pathname contient une référence circulaire (à travers un lien symbolique).
ENOSPC
Le périphérique contenant *pathname* n'a pas assez de place pour le nouveau répertoire. Cette erreur se produit également si le quota de disque de l'utilisateur est dépassé.

Fonction access

```
#include <unistd.h>
int access(const char *pathname, int mode);
```

description : Tester les permissions d'un fichier

Paramètres :

const char *pathname : le fichier et son chemin d'accès.
Int mode : Le ou les droits à tester.

Mode : R_OK Teste si le fichier a le droit de lecture.
W_OK Teste si le fichier a le droit d'écriture.
X_OK Teste si le fichier a le droit d'exécution.
F_OK Teste si le fichier existe.

L'appel système access détermine si le processus appelant a le droit d'accéder à un fichier. Il peut vérifier toute combinaison des permissions de lecture, écriture ou exécution ainsi que tester l'existence d'un fichier.

La valeur de retour est zéro si le processus dispose de toutes les permissions passées en paramètre. Si le fichier existe mais que le processus n'a pas les droits, access renvoie -1 et positionne errno à EACCES (ou EROFS si l'on a testé les droits en écriture d'un fichier situé sur un système de fichiers en lecture seule).

Si le second argument est F_OK, access vérifie simplement l'existence du fichier. Si le fichier existe, la valeur de retour est 0; sinon, elle vaut -1 et errno est positionné à ENOENT.

Exemple 1 : Vérifier l'existence d'un fichier
on peut faire le bout de code suivant :
supposons le fichier qui se nomme « fichier.cpp » dans le répertoire « /home/louis/travail »

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

#include <unistd.h>
#include <errno.h>>
#include <string>


using namespace std;

int main(int argc, char *argv[])
{

      if (access("/home/louis/travail/fichier.cpp", F_OK) == 0)
      {
         cout << "fichier existe" << endl;
      }
        else
      {
         switch(errno)
         {
            case ENOENT:

               cout << "le fichier n'existe pas " << endl;
               break;

            case EACCES:

              cout << "le fichier n'est pas accessible" << endl;
              break;
         }
      }
```

Ouverture ou création d'un fichier - Fonction open

Entête **<sys/types.h>**
<sys/stat.h>
<fcntl.h>

int open(const char ******pathname***, int** *flags***);**

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

DESCRIPTION

L'appel-système **open()** sert à convertir un chemin d'accès en descripteur de fichier (un petit entier non négatif utilisable pour les opérations d'entrées/sorties ultérieures telles **read**, **write**, etc.).

flags est l'un des éléments **O_RDONLY**, **O_WRONLY** ou **O_RDWR** qui permet d'ouvrir le fichier en lecture seule, écriture seule, ou lecture/écriture respectivement. À cette valeur peut être ajouté un ou plusieurs attributs avec un *OU binaire* (|) :

O_CREAT

Créer le fichier s'il n'existe pas. Le possesseur (UID) du fichier est renseigné avec l'UID effectif du processus.

O_EXCL

En conjonction avec **O_CREAT**, déclenchera une erreur si le fichier existe, et **open** échouera. On considère qu'un lien symbolique, quelque soit l'endroit où il pointe. **O_EXCL** ne fonctionne pas sur les systèmes de fichiers NFS. Les programmes qui ont besoin de cette fonctionnalité pour verrouiller des tâches risquent de rencontrer une concurrence critique (race condition). La solution consiste à créer un fichier unique sur le même système de fichiers (par exemple avec le PID et le nom de l'hôte), utiliser **link**(2) pour créer un lien sur un fichier de verrouillage et d'utiliser **stat**(2) sur ce fichier unique pour vérifier si le nombre de liens a augmenté jusqu'à 2. Ne pas utiliser la valeur de retour de **link()**.

O_NOCTTY

Si *pathname* correspond à un périphérique de terminal --- voir **tty**(4) ---, il ne deviendra pas le terminal contrôlant le processus même si celui-ci n'est attaché à aucun autre terminal.

O_TRUNC

Si le fichier existe, est un fichier régulier, et est ouvert en écriture (O_RDWR ou O_WRONLY), il sera tronqué à une longueur nulle. Si le fichier est une FIFO ou un périphérique temrinal, l'attribut O_TRUNC est ignoré. Sinon, le comportement de O_TRUNC n'est pas précisé. Sur de nombreuses versions de Linux, il sera ignoré ; sur d'autres versions il déclenchera une erreur).

O_APPEND

Le fichier est ouvert en mode "ajout". Initialement, et avant chaque **write**, la tête de lecture/écriture est placée à la fin du fichier comme avec **lseek**. Il y a un risque d'endommager le fichier lorsque **O_APPEND** est utilisé, sur un système de fichiers NFS, si plusieurs processus tentent d'ajouter des données simultanément au même fichier. Ceci est dû au fait que NFS ne supporte pas l'opération d'ajout de données dans un fichier, aussi le noyau client est obligé de la simuler, avec un risque de concurrence des tâches.

Les constantes symboliques suivantes sont disponibles pour *mode. Ce sont les mêmes que pour la fonction « chmod » :*

Valeur	Description
S_ISUID 04000	modification du numéro d'utilisateur (UID) à l'exécution
S_ISGID 02000	modification du numéro de groupe (GID) à l'exécution.
S_ISVTX 01000	positionner le sticky bit pour conserver le code du programme en mémoire après exécution.
S_IRUSR (S_IREAD) 00400	accès en lecture pour le propriétaire
S_IWUSR (S_IWRITE) 00200	accès en écriture pour le propriétaire
S_IXUSR (S_IEXEC) 00100	accès en exécution/parcours par le propriétaire
S_IRGRP 00040	accès en lecture pour le groupe
S_IWGRP 00020	accès en écriture pour le groupe
S_IXGRP 00010	accès en exécution/parcours pour le groupe
S_IROTH 00004	accès en lecture pour les autres
S_IWOTH 00002	accès en écriture pour les autres
S_IXOTH 00001	accès en exécution/parcours pour les autres

Le *mode* devrait toujours être indiqué quand **O_CREAT** est dans les attributs *flags*, (il est ignoré dans les autres cas).

Valeur de retour

open et **creat** renvoient le nouveau descripteur de fichier s'ils réussissent, ou -1 s'ils échouent, auquel cas *errno* contient le code d'erreur.

ERREURS

EEXIST
pathname existe déjà et **O_CREAT** et **O_EXCL** ont été indiqués.
EISDIR
On a demandé une écriture alors que *pathname* correspond à un répertoire.
EACCES
L'accès demandé au fichier est interdit, ou l'un des répertoires du chemin *pathname* ne permet pas de consultation, ou le fichier n'existe pas mais le répertoire parent ne permet pas l'écriture.
ENAMETOOLONG
pathname est trop long.
ENOENT
Un répertoire du chemin d'accès *pathname* n'existe pas où est un lien symbolique pointant nulle part.
ENOTDIR
Un élément du chemin d'acces *pathname* n'est pas un répertoire, ou l'attribut **O_DIRECTORY** est utilisé et *pathname* n'est pas un répertoire.
ENXIO
O_NONBLOCK | O_WRONLY est indiqué, le fichier est une FIFO et le processus n'a pas de fichier ouvert en lecture. Ou le fichier est un noeud spécial et il n'y a pas de périphérique correspondant.

ENODEV
pathname correspond à un fichier spécial et il n'y a pas de périphérique correspondant.
EROFS
Un accès en écriture est demandé alors que *pathname* réside sur un système de fichiers en lecture seule.
ETXTBSY
On a demandé une écriture alors que *pathname* correspond à un fichier exécutable actuellement utilisé.
EFAULT
pathname pointe en dehors de l'espace d'adressage accessible
ELOOP
pathname contient une référence circulaire (à travers un lien symbolique), ou l'attribut **O_NOFOLLOW** est indiqué et *pathname* est un lien symbolique.
ENOSPC
pathname devrait être créé mais le périphérique concerné n'a plus assez de place pour un nouveau fichier.
ENOMEM
Pas assez de mémoire pour le noyau
EMFILE
Le processus a déjà ouvert le nombre maximal de fichiers.

ENFILE
La limite du nombre total de fichiers ouverts sur le système est atteinte.

Exemple :

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

#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <string>


using namespace std;

int main(int argc, char *argv[])
{
   int idFichier = 0;

      if (access(argv[1], F_OK) == 0)
      {
         cout << "fichier existe" << endl;
         idFichier = open(argv[1], O_RDWR);
      }
        else
      {
         switch(errno)
         {
            case ENOENT:

               cout << "le fichier n'existe pas, on le crée " << endl;
               idFichier = open(argv[1], O_CREAT | O_RDWR, 0700);
               break;

            case EACCES:

              cout << "le fichier n'est pas accessible\n" << endl;
              break;
         }
       }

          // On fait quelque chose avec le fichier ...

          // On ferme le fichier
          close(idFichier);
```

Fonction read

Description : Permet de lire un certain nombre d'octet d'un fichier

```
#include <unistd.h>

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

Paramètres :

- int fd : Le descripteur de fichier précédemment ouvert par la fonction open
- void *buf : La chaîne de caractère qui recevra le ou les caractères lus.
- size_t count : Le nombre de caractère à lire dans le fichier.

Exemple :

Ouverture du fichier et lecture d'un caractère à la fois jusqu'à la fin du fichier.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
	int DescFichier;
	char *Tampon = new char[1];
	cout << argv[1] << endl;

	DescFichier = open(argv[1], O_CREAT|O_RDONLY);  // Ici O_CREAT permet de créerle fichier
				// O_RDONLY : lecture seulement

	if (DescFichier == -1)
	{
		cout << "Fichier" << argv[1] << "n'a pas été créé correctement" << endl;
		exit(0); // fin du programme
	}
	else
	{
		// Le fichier est correctement ouvert
		 // faire le traitement avec le fichier
		while ( read(DescFichier, Tampon, 1) != 0 )  // read retourne 0 si c'est la fin du fichier
		{
			// Afficher le caractère lu
			cout << "caractere :  " << Tampon << endl;
		 }
		close(DescFichier);
	}

    return 0;
}
```

Fonction write

Description : Permet d'écrire un octet ou une série d'octet dans un fichier.

```
#include <unistd.h>

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

Exemple :
On ouvre un fichier et on écrit dans un autre.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
  int DescFichierSource;
  int DescFichierDest;

  char *Tampon = new char[1];
      cout << argv[1] << endl;

     if (open(argv[1], O_CREAT|O_RDONLY) == -1) // Ici O_CREAT permet de créer le fichier
     {
          cout << "Fichier" << argv[1] << "n'a pas été ouvert correctement" << endl;
          exit(0); // fin du programme
     }

     if (open(argv[2], O_CREAT | O_RDWR) == -1 )
     {
          cout << "Fichier" << argv[1] << "n'a pas été ouvert correctement" << endl;
          exit(0); // fin du programme
      }
          // Le fichier est correctement ouvert
           // faire le traitement avec le fichier
           while ( read(DescFichierSource, Tampon, 1) != 0 )  // read retourne 0 si c'est la fin du fichier
           {
                // Écrire le caractère lu dans l'autre fichier.
                write(DescFichierDest, Tampon, 1);
                cout << "caractere :  " << Tampon << endl;  // à titre de déverminage
           }
           close(DescFichierSource);
           close(DescFichierDest);
      return 0;
}
```

Fonction close

Description : Ferme un descripteur de fichier précédemment ouvert avec la fonction open

#include <unistd.h>

int close(int fd);

Voir les exemples précédent.

Opération sur les répertoires

Changer de répertoire – fonction chdir

Entête <unistd.h>

Syntaxe : int chdir(const char *_path_);

DESCRIPTION

chdir remplace le répertoire courant par celui indiqué dans le chemin *path*.

VALEUR RENVOYÉE

chdir et **fchdir** renvoient 0 s'ils réussissent, ou -1 s'ils échouent, auquel cas *errno* contient le code d'erreur.

ERREURS

Suivant le type de système de fichiers, plusieurs erreurs peuvent être renvoyées, les plus courantes pour **chdir** sont les suivantes :

EFAULT

path pointe en dehors de l'espace d'adressage accessible.

ENAMETOOLONG

path est trop long.

ENOENT

Le fichier n'existe pas.

ENOMEM

Pas assez de mémoire pour le noyau.

ENOTDIR

Un élément du chemin d'accès n'est pas un répertoire.

EACCES

L'accès n'est pas autorisé sur un élément du chemin.

ELOOP

path contient des références circulaires (à travers un lien symbolique).

EIO

Erreur générique d'entrée/sortie.

Les erreurs courantes pour **fchdir** sont :

EBADF

fd n'est pas un descripteur de fichier valide.

EACCES

Le répertoire ouvert sur *fd* n'autorise pas le parcours.

Exemple :

Changer le répertoire courant pour « /home/schasse ».

```
#include <iostream>

#include <errno.h>
#include <unistd.h>
#include <string>


using namespace std;

int main(int argc, char *argv[])
{

      if (chdir(argv[1]) == 0)
      {
         cout << "Répertoire courant: " << argv[1] << endl;

      }
        else
      {
         switch(errno)
         {
            case ENOENT:

               cout << "le répertoire n'existe pas" << endl;
               break;
          }
      }
```

Lecture du contenu du répertoire

Linux dispose de fonctions pour lire le contenu des répertoires.

Pour lire le contenu d'un répertoire, les étapes suivantes sont nécessaires :

1. Ouvir le répertoire que l'on veut consulter avec la fonction « opendir » :

   Appelez opendir en lui passant le chemin du répertoire que vous souhaitez explorer. Opendir renvoie un descripteur DIR*, dont vous aurez besoin pour accéder au contenu du répertoire. Si une erreur survient, l'appel renvoie NULL ;

   Syntaxe : DIR *opendir(const char *nom);
   Entête : <sys/types.h >
   <dirent.h>

   La fonction opendir() ouvre un répertoire correspondant au répertoire nom, et renvoie un pointeur sur ce flux. Le flux est positionné sur la première entrée du répertoire.

   VALEUR RENVOYÉE
   Les fonctions opendir() et fdopendir() renvoient un pointeur sur le flux répertoire. Si une erreur se produit, NULL est renvoyé, et errno contient le code d'erreur.

   ERREURS
   EACCES Accès interdit.

   EBADF fd n'est pas un descripteur de fichier valable ouvert en lecture.

   EMFILE Trop de descripteurs de fichier pour le processus en cours.

   ENFILE Trop de fichiers ouverts simultanément sur le système.

   ENOENT Le répertoire n'existe pas, ou nom est une chaîne vide.

   ENOMEM Pas assez de mémoire pour terminer l'opération.

   ENOTDIR nom n'est pas un répertoire

Exemple :

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

DIR[1] *leRep = opendir("."); // On ouvre le répertoire courant
```

1Le type DIR est en fait une redéfinition de type avec un typedef sur une structure de type «__dirstream ».

2. Dans une boucle, utilisez la fonction « readdir » en lui passant le descripteur DIR* que vous a renvoyé opendir. À chaque appel, readdir renvoie un pointeur vers une structure de type « dirent » correspondant à l'entrée suivante dans le répertoire. Lorsque vous atteignez la fin du contenu du répertoire, readdir renvoie NULL.

   La structure dirent que vous obtenez via « readdir » dispose d'un champ d_name qui contient le nom de l'entrée.

   Vous aurez besoin d'un pointeur sur une structure de type dirent
   struct dirent *Element_du_Rep;

   TANT QUE (Il y a des éléments dans le répertoire)
   Allez chercher les informations de cet élément.

   On lit une entrée du répertoire avec la fonction « readdir » en passant à cette fonction, le pointeur de la structure obtenu par l'appel à « opendir ». Si la lecture retourne NULL (la valeur 0) c'est qu'il n'y a plus d'autres éléments à lire dans le répertoire en question.

   On utilise donc « leRep » tel qu'obtenu par l'appel à « opendir » de l'étape précédente et on passe cette variable comme paramètre à « readdir ». Il ne reste plus qu'à mettre le tout dans une boucle comme ci-dessous :

```
while ( (Element_du_Rep = readdir(leRep)) != 0 )
{
        // Aller chercher les informations de cette entrée
}
```

3. Appelez closedir en lui passant le descripteur DIR* après la sortie de la boucle.

```
closedir(leRep);
```

   Incluez <sys/types.h> et <dirent.h> si vous utilisez ces fonctions dans votre programme.

Information d'une entrée de répertoire.

Comme nous l'avons vu précédemment la boucle suivante permet d'aller chercher chaque entrée du répertoire :

```
while ( (Element_du_Rep = readdir(leRep)) != 0 )
{
        // Aller chercher les informations de cette entrée
}
```

Il nous reste maintenant à aller chercher les informations d'une entrée de répertoire tel que « readdir » nous la retourne.

Pour aller chercher le nom de l'entrée, il suffit de faire afficher l'élément « d_name », qui est une chaîne de caractère, de la structure dirent.

```
while ( (Element_du_Rep = readdir(leRep)) != 0 )
{
        // Faire afficher le nom
        cout << "Nom: " << Entree_du_Rep->d_name << endl;
}
```

Voici la structure « dirent » :

```
struct dirent
{
        #ifndef __USE_FILE_OFFSET64
          __ino_t d_ino;
          __off_t d_off;
        #else
          __ino64_t d_ino;
          __off64_t d_off;
        #endif
          unsigned short int d_reclen;
          unsigned char d_type;
          char d_name[256];
};
```

L'élément de cette structure qui nous intéresse est la chaîne de caractère « d_name » qui représente le nom de l'entrée. Cette entrée peut évidemment être le nom d'un fichier ou d'un répertoire.

Mais il n'y a pas que le nom d'un fichier ou d'un répertoire qui est normalement affiché. On aimerait peut-être également faire afficher les droits du fichier, sa grosseur en octet, la date de dernier accès, etc...

Ces informations supplémentaires sont disponibles en appelant la fonction « stat » et en passant le nom de l'entrée en paramètre à cette fonction.

Syntaxe : int stat(const char *path, struct stat *buf);

Entête : <sys/types.h>
<sys/stat.h>
<unistd.h>

DESCRIPTION
Cette fonction renvoie des informations à propos du fichier indiqué. Vous n'avez besoin d'aucun droit d'accès au fichier pour obtenir les informations, mais vous devez avoir le droit de parcours de tous les répertoires mentionnés dans le chemin menant au fichier.

Le premier paramètre « const char *path » représente le chemin du fichier pour lequel on désire obtenir les informations.

La fonction retourne une structure stat contenant les champs suivants :

```
struct stat {
      dev_t     st_dev;      /* Périphérique                 */
      ino_t     st_ino;      /* Numéro inœud                 */
      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 de 512B alloués */
      time_t    st_atime;    /* Heure dernier accès          */
      time_t    st_mtime;    /* Heure dernière modification  */
      time_t    st_ctime;    /* Heure dernier changement état */
  };
```

Il faut donc déclarer une variable de type « stuct stat » et passer l'adresse de cette variable en deuxième paramètre à la fonction stat pour récupérer les informations.

Exemple :

```
struct stat  Info;  // Pour recevoir les informations du fichier ou du répertoire

...

Pour chaque Entrée du répertoire
        | //Appeler la fonction stat avec le nom du répertoire ou du fichier comme 1er paramètre et
        |//  l'adresse de la variable « Info » en 2ème paramètre
        |// Faire afficher les informations voulues (nom, grosseur, heure, droits, etc...)
        | // Les éléments à faire afficher concernant le fichier sont dans la structure « stat »
        | // nommée « Info ».
        |
```

Affichage des temps d'accès

Les informations concernant les temps d'accès du fichier (temps de dernière modification, temps de dernier accès, etc...) sont accessibles par la structure « stat ». Il existe trois éléments dans cette structure qui représente les différents temps d'accès.

```
st_atime;   /* Heure dernier accès           */
st_mtime;   /* Heure dernière modification   */
st_ctime;   /* Heure dernier changement état */
```

Cependant, toutes ces valeurs sont exprimées en nanoecondes depuis la version du noyau 2.5.48. Il faut donc utiliser une fonction pour transformer les valeurs en nanosecondes en chaînes de caractères affichables.

Heureusement, il existe la fonction « ctime » qui permet cette transformation.

Par exemple pour transformer la valeur contenu dans le champ « st_atim » utilisons la fonction ctime de la façon suivante :

```
struct stat  Info;

stat(Entree_du_Rep->d_name, &Info)

cout << "Heure de dernier accès : " << ctime(&Info.st_atim.tv_sec) << endl;
```

Exemple complet pour faire afficher le nom du fichier et le numéro du propriétaire :

```
DIR *leRep = opendir("."); // Ouvre le répertoire courant dans ce cas

  struct dirent *Entree_du_Rep;

  struct stat  Info; // POur les infos du fichier ou du répertoire


while ( (Entree_du_Rep = readdir(leRep)) != 0 )
  {
     string Nom = Entree_du_Rep->d_name;

        // Récupérer les informations du fichier
        // et faire afficher le nom et le numéro du propriétaire du fichier.

     if (stat(Entree_du_Rep->d_name, &Info) == 0)
     {
        if (Nom.length() <= 7)       // Cette vérification permet seulement de faire
                                     // en sorte que l'affichage se fasse sur les mêmes
                                     // colonnes
           cout << Entree_du_Rep->d_name << "\t\t" << Info.st_uid << endl;
        else
           cout << Entree_du_Rep->d_name << "\t" <<  Info.st_uid << endl;

     }
  }
```