Institut Galilée
Systèmes d'exploitation
ISPG 1
2007-2008

11 mars 2008

Système de fichier
Gestion des erreurs
Arguments du main
variables d'environnement

Première partie

TD 2

1 Représentation interne des fichiers

Exercice 1. *Afin d'économiser de la place en mémoire, il est possible de regrouper différents objets dans un même octet ou mot mémoire. en C, on utilise les champs de bits qui consistent en un ensemble de bit consécutifs dans un mot qui se caractérise par le nombre de bits qui le composent.*

Définir une structure de champs de bits qui permet de représenter le mode d'un fichier UNIX.

Exercice 2. *Écrivez un programme qui prend en argument un fichier et renvoie :*

1. *son numéro de inode ;*
2. *le nombre de liens qu'il possède ;*
3. *son propriétaire ;*
4. *son groupe ;*
5. *sa taille ;*
6. *son type.*

Vous utiliserez la fonction `stat` *puis modifierez le programme afin d'utiliser la fonction* `fstat`*.*

Exercice 3. *Écrivez un programme qui interprète ses deux arguments comme des références de fichiers et renvoie la valeur :*

-1 *sil le nombre de paramètres n'est pas correct ;*

0 *si les deux références correspondent à deux nœuds différents du système de gestion de fichiers ;*

1 *si elles correspondent au même nœud ;*

2 *si une erreur est rencontrée dans des appels à la primitive* `stat`*.*

Exercice 4. *Écrivez un programme qui recopie caractère par caractère le fichier donné en argument sur la sortie standard, uniquement si le fichier donné en argument est un fichier régulier (en utilisant la fonction* `fstat`*).*

Exercice 5. *Écrivez un programme qui effectue le travail de la commande cp en E-S de bas niveau.*

Deuxième partie

TP 2

2 La gestion des erreurs

Les appels système apparaissent, traditionnellement, comme des fonctions à valeur entière et l'échec d'un appel est matérialisé par une valeur de retour égale à -1. En cas d'échec d'un appel à l'une de ces fonctions, il est possible d'obtenir une information plus précise sur la nature de l'erreur rencontrée par l'intermédiaire de la variable entière externe `errno`. La valeur de la variable `errno` n'est significative qu'au retour d'un appel à une fonction ayant échoué (la variable n'est pas affectée au cours des appels réussis). La variable `errno` est définie dans le fichier `errno.h` qui contient la liste des erreurs susceptibles de provoquer l'Échec d'un appel système. Dans ce même fichier, pour chacune des valeurs possibles de `errno`, on trouve la définition d'une constante symbolique et un texte en commentaire indiquant la nature de l'erreur. Il est possible, lorsqu'une fonction échoue (valeur de retour -1 pour les fonctions à valeur numérique ou NULL pour celles renvoyant un pointeur), de visualiser le message correspondant à l'erreur rencontrée au moyen de la fonction `void perror (const char *p_chaine)` qui affiche le message associé à l'erreur, précédé de la chaîne pointée par `p_chaine` (définie par l'utilisateur) suivie du séparateur `:<space>`.

Par ailleurs, deux variables `sys_nerr` et `sys_errlist` sont accessibles si elles ont été déclarées externes dans le programme, sous la forme `extern int sys_nerr; extern char *sys_errlist[];` `sys_errlist` est un tableau de `sys_nerr` pointeurs et pour chaque entier `i`, `sys_errlist[i]` est un pointeur sur le message correspondant à l'erreur `i`.

```
void main (int nb_args, char *args[])
{
...
  if ( open (args[1], O_RDONLY) == -1) {
    perror("Erreur d'Overture");
    exit(1);
  }
...
}
```

Utiliser la gestion des erreurs dans les programmes à écrire dans la suite.

3 Les variables d'environnement

La forme la plus générale de la fonction principale d'un programme (en langage C) correspond au prototype suivant :

```
int main (int argc, char *argv[], char *arge[]);
```

Le paramètre `argc` est le nombre total de paramètres. C'est à dire, le nombre de composantes de la commande shell correspondante. Le paramètre `argv` est un tableau contenant les différents paramètres de la commande. Il contient `argc` pointeurs (plus un pointeur NULL pour marquer la fin). Le i-ème élément du tableau est le i-ème argument de la commande (le premier élément du tableau, `argv[0]`, pointe sur le nom de la commande). Le paramètre `arge` est une liste de pointeurs permettant l'accès à l'environnement dans lequel le processus s'exécute. Chacun des pointeurs permet d'accéder à une chaîne de caractères de la forme `nom_variable=chaine_valeur` Ceci permet à un processus correspondant à l'exécution d'une commande tapée sous un shell de s'exécuter dans l'environnement défini par l'utilisateur (par exemple, les variables TERM ou PATH).

3.1 Accès à l'environnement

L'accès aux variables de l'environnement d'un processus peut se faire, dans un programme C, de différentes façons.

Accès par le paramètre `arge` de la fonction main.

```
#include<stdio.h>
void main (int argc, char *argv[], char **arge) {
  int i;
  for ( i=0; arge[i] != NULL; i++)
    puts(arge[i]);
}
```

Tester l'accès aux variables de l'environnement selon la méthode précédente.

Accès par la variable externe `environ`. L'environnement est également accessible au travers de la variable externe environ. Le programme doit contenir la déclaration de cette variable sous la forme `extern char **environ;`

```
#include<stdio.h>
  void main (int argc, char *argv[], char **arge) {
  extern **environ; char **env;
  env = environ; /* préserve environ */
  while ( *env != NULL )
    puts( *env++);
}
```

Tester l'accès aux variables de l'environnement selon la méthode précédente.

Accès par la fonction standard `char * getenv(const char *nom_variable);`

La fonction standard `getenv()` recherche, dans la liste des variables d'environnement, une chaîne de la forme "VAR=valeur" et retourne un pointeur sur la chaîne "valeur". Si la variable n'est pas définie, la veleur retournée est le pointeur NULL.

```
#include<stdio.h>
#include<stdlib.h>
void main (int argc, char *argv[], char **arge) {
  char *VAR;
  if ( (VAR = getenv("PATH")) != NULL )
    fprintf(stdout, "Valeur PATH = %s", VAR);
  else
    fprintf(stderr, "VARIABLE PATH NON DEFINIE !");
}
```

Tester l'accès aux variables de l'environnement selon la méthode précédente.

3.2 Création, ou modification de la valeur, d'une variable de l'environnement

Un processus peut modifier la valeur d'une variable de son environnement ou y ajouter une nouvelle variable par l'appel à la fonction standard `int putenv(const char *chaine);`.

La chaîne de caractères donnée en paramètre a la forme VAR=valeur. La variable de nom VAR est soit créée (si elle n'existe pas déjà) soit modifiée dans l'environnement du processus. La valeur de retour de la fonction est nulle si la création ou la modification de la variable a été possible et non nulle sinon.

```
#include<stdio.h>
#include<stdlib.h>
void main (int argc, char *argv[], char **arge) {
  putenv("MYVAR=titi");
  fprintf(stdout, "Vefir MYVAR = %s\n", getenv("MYVAR"));
}
```

Tester la création et la modification des variables de l'environnement selon la méthode précédente. Habituellement, un processus hérite de l'environnement du processus père. Initialement, la variable environ, positionnée avant d'entrer dans la fonction principale `main()` d'un programme, est identique au troisième argument `**arge` de cette même fonction `main()`. La fonction `putenv()` manipule l'environnement pointé par la variable externe environ. La fonction `putenv()` ne modifie pas l'environnement accessible par `**arge`. La fonction `putenv()` permet uniquement de créer ou de modifier un environnement qui sera passé à un processus fils.

L'effet d'une action effectuée sur une variable d'environnement par un processus n'est jamais répercuté sur l'environnement du processus père. Vérifier les affirmations précédentes.

4 Appels système gestion des fichiers

4.1 Création d'un fichier

Écrire un programme qui crée, au travers de l'appel

```
int creat(const char *pathname, mod_t mode);
```

un fichier en lecture/écriture. Si le fichier existe déjà , une erreur doit Étre retournée. Quel est le code d'erreur retourné lorsque le fichier existe déjà ? Si l'argument mode spécifié est 755 (`rwxr-xr-x`), est-ce que le fichier est créé avec exactement ces droits ? Expliquer !

4.2 Obtention des caractéristiques d'un fichier

Écrire un programme qui récupère les caractéristiques de fichiers donnés, au travers des appels des fonctions `int stat(); int fstat();` et `int lstat();`. Pour un fichier donné, afficher les caractéristiques suivantes :

- le numéro d'inode,
- la taille du fichier,
- la protection,
- la taille de bloc,
- le nombre de liens physiques,
- le nombre de blocs,
- l'ID du propriétaire,
- l'heure du dernier accès,
- l'ID du groupe.

L'affichage d'une heure dans un format lisible peut être accompli en utilisant la fonction `ctime()`.

4.3 Ouverture d'un fichier, lecture et écriture dans un fichier.

Cet exercice a pour but d'utiliser les fonctions `int open()`, `ssize_t read()` et `ssize_t write()`.

Écrire un programme qui recopie un fichier source, `fichier_source`, dans un fichier destinataire, `fichier_destinataire`. Le programme doit vérifier que le fichier source est un fichier régulier (utiliser la macro `S_ISREG(m)`, cf. `int stat()`). Le programme doit également vérifier qu'il n'existe pas déjà de fichier de même nom que le fichier destinataire. Quels sont les temps d'exécution (utiliser `/bin/time`) respectifs si la taille du buffer utilisée dans la fonction read() est de 1024 octets puis un octet ? Expliquer !

4.4 Duplication de descripteurs

Cet exercice a pour but d'utiliser la fonction `int dup()`.

Écrire un programme qui redirige la sortie d'erreur standard vers un fichier, `fichier_erreur`, préalablement créé. C'est à dire, toute écriture de la forme `write(2, ...)` doit se faire dans le fichier `fichier_erreur`. Le descripteur de valeur 2 étant au départ celui de la sortie d'erreur standard. Quelle est la propriété de la fonction `dup()` qui est exploitée pour ainsi rediriger les E/S standards ?

4.5 Positionnement de la tête de lecture/écriture dans un fichier

Cet exercice a pour but d'utiliser la fonction `off_t lseek()`.

Écrire un programme qui crée un fichier vide. Positionner la tête de lecture/écriture sur le 10 000e octet à partir du début du fichier. Ecrire un caractère à cette position. Quelle doit être la taille du fichier ? Est-ce cette taille qui est retournée par la commande `ls -l` ? Est-ce que les blocs correspondant au trou de 9 999 caractères ont été alloués (utiliser `df`) ? La primitive `lseek()` permet de déplacer l'offset courant d'un fichier et retourne le nouvel offset. Ecrire un programme qui, après un certain nombre de lecture/écriture sur un fichier, retourne la valeur de l'offset courant. L'offset est associé à un fichier et non pas à un descripteur. Si deux descripteurs référencent un même fichier, la modification (par lecture/écriture ou `lseek()`) de l'offset du fichier via un descripteur est "visible" via l'autre descripteur. Vérifier l'affirmation précédente.

```
STAT(2)            Manuel du programmeur Linux

NOM
 stat,  fstat, lstat - Obtenir le statut d'un fichier (file
       status).

SYNOPSIS
 #include &ltsys/stat.h>
 #include &ltunistd.h>

 int stat(const char *file_name, struct stat *buf);
 int fstat(int filedes, struct stat *buf);
 int lstat(const char *file_name, struct stat *buf);

DESCRIPTION
 Ces fonctions  renvoient  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 men-
 tionnés dans le chemin menant au fichier.

 stat récupère le statut du fichier pointé par file_name et
 remplit le buffer buf.

 lstat  est  identique  à  stat, sauf qu'il donne le statut
 d'un lien lui-même et non pas du  fichier  pointé  par  ce
 lien.

 fstat est identique à stat, sauf que le fichier ouvert est
 pointé par le descripteur filedes, obtenu avec open(3).

 Les trois fonctions retournent  une  structure  stat  con-
 tenant les champs suivants :

  struct stat
  {
      dev_t   st_dev;      /* Périphérique    */
      ino_t   st_ino;      /* Numéro i-noeud  */
      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     */
      unsigned long st_blksize;  /* Taille de bloc pour E/S     */
      unsigned long 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    */
  };

 La valeur st_blocks donne la taille du fichier en blocs de
 512 octets.  La valeur st_blksize  indique  la  taille  de
 bloc   "préférée"  pour  les  entrées/sorties  du  système
 (l'écriture dans  un  fichier  par  petits  morceaux  peut
 induire de nombreuses étapes lecture-modification-écriture
 peu efficaces).

 Les systèmes de fichiers de Linux n'implémentent pas  tous
 les  champs  "time".  Traditionnellement st_atime est mod-
 ifié par mknod(2), utime(2), read(2), write(2),  et  trun-
 cate(2).

 Généralement  st_mtime est modifié par mknod(2), utime(2),
 et write(2).  mais pas en  cas  de  modification  de  pro-
 priétaire,  de  groupe, de compte de liens physiques ou de
 mode.

 Traditionnellement st_ctime est modifié par une  écriture,
 une  lecture,  ou  une  modification de données concernant
 l'i-noeud (propriétaire, groupe, mode, etc...)

 Les macros POSIX suivantes sont  fournies  pour  manipuler
 les statuts d'un fichier

  S_ISLNK(m)  est-ce un lien symbolique ?

  S_ISREG(m)  un fichier régulier ?

  S_ISDIR(m)  un répertoire ?
```

```
 S_ISCHR(m)  un peripherique en mode caractère ?

 S_ISBLK(m)  un périphérique en mode blocs ?

 S_ISFIFO(m) une FIFO ?

 S_ISSOCK(m) une socket ?

Les attributs suivants correspondent au champ st_mode.

 S_IFMT   00170000  (non POSIX) masque de l'ensemble
    des bits du type de fichier

 S_IFSOCK 0140000 (non POSIX) socket

 S_IFLNK  0120000 (non POSIX) lien symbolique

 S_IFREG  0100000 (non POSIX) fichier régulier

 S_IFBLK  0060000 (non POSIX) périphérique blocs

 S_IFDIR  0040000 (non POSIX) répertoire

 S_IFCHR  0020000  (non  POSIX)  périphérique   car
    actères

 S_IFIFO  0010000 (non POSIX) fifo

 S_ISUID  0004000 bit Set-UID

 S_ISGID  0002000 bit Set-Gid

 S_ISVTX  0001000 (non POSIX) bit "sticky"

 S_IRWXU  00700 droits de lecture/écriture/exécution
    du propriétaire

 S_IRUSR  00400 le propriétaire a le droit  de  lec
    ture (comme S_IREAD qui n'est pas POSIX)

 S_IWUSR  00200  le propriétaire a le droit d'écrit
    ure (comme S_IWRITE qui n'est pas POSIX)

 S_IXUSR  00100   le   propriétaire   a   le   droit
    d'exécution  (comme  S_IEXEC qui n'est pas
    POSIX)

 S_IRWXG  00070 droits de lecture/écriture/exécution
    du groupe

 S_IRGRP  00040 le groupe a le droit de lecture

 S_IWGRP  00020 le groupe a le droit d'écriture

 S_IXGRP  00010 le groupe a le droit d'exécution

 S_IRWXO  00007 droits de lecture/écriture/exécution
    des autres

 S_IROTH  00004 les autres ont le droit de lecture

 S_IWOTH  00002 les autres ont le droit d'écriture

 S_IXOTH  00001 les autres ont le droit d'exécution

VALEUR RENVOYÉE
 Ces fonctions retournent zero si  elles  réussissent.  En
 cas  d'echec  -1  est  renvoyé,  et errno contient le code
 d'erreur.

ERREURS
 EBADF   filedes est un mauvais descripteur.

 ENOENT  Un composant de  file_name  n'existe  pas,  ou  il
   s'agit d'une chaine vide.

 ENOTDIR Un composant du chemin d'accès n'est pas un réper-
   toire.

 ELOOP   Trop  de  liens  symboliques  rencontrés  dans  le
   chemin d'accès.

 EFAULT  Un  pointeur  se  trouve  en  dehors  de  l'espace
   d'adressage.

 EACCES  Autorisation refusée.

 ENOMEM  Pas assez de mémoire pour le noyau.

 ENAMETOOLONG
   Nom de fichier trop long

EXEMPLE
 #include &ltstdio.h>
 #include &ltstdlib.h>
 #include &ltsys/stat.h>
 #include &ltunistd.h>

 int
 main (int nb_args, char * args [])
 {
      struct stat    sts;

      if (nb_args != 2) {
     fprintf (stderr, "syntaxe : %s &ltfichier>\n", args [0]);
     exit (1);
      }

      if ( stat (args [1], & sts) != 0) {
     fprintf (stderr, "%s : erreur %X\n", args [0], errno);
     exit (1);
      }

      fprintf (stdout,
     "Périphérique : %d\n",
     sts . st_dev);
      fprintf (stdout,
     "Noeud : %ld\n",
     sts . st_ino);
      fprintf (stdout,
     "Protection : %o\n",
     sts . st_mode);
      fprintf (stdout,
     "nb liens matériels: %d\n",
     sts . st_nlink);
      fprintf (stdout,
     "ID propriétaire : %d\n",
     sts . st_uid);
      fprintf (stdout,
     "ID groupe: %d\n",
     sts . st_gid);
      fprintf (stdout,
     "Taille : %lu octets\n",
     sts . st_size);
      fprintf (stdout,
     "Taille de bloc : %lu\n",
     sts . st_blksize);
      fprintf (stdout,
     "Nombre de blocs : %lu\n",
     sts . st_blocks);
 }

CONFORMITÉ
 Les appels stat, etfstat sont conformes aux versions SVr4,
 PPOSIX,  X/OPEN,  BSD 4.3.  L'appel lstat est conforme aux
 versions BSD 4.3 et SVr4.  SVr4 mentionne  des  conditions
 d'erreurs  supplémentaires  pour  lstat EINTR, ENOLINK, et
 EOVERFLOW.  Pour stat, etlstat SVr4 indique les conditions
 supplémentaires  EACCES,  EINTR,  EMULTIHOP,  ENOLINK,  et
 EOVERLOW.  L'utilisation des champs st_blocks  et  st_blk-

 size  risque  d'être moins portable. Ils ont été introduit
 dans BSD, et ne  sont  pas  mentionnée  dans  POSIX.  Leur
 interprétation  change  suivant les systèmes, voire sur un
 même système s'il y a des montages NFS.

VOIR AUSSI
 chmod(2), chown(2), readlink(2), utime(2)
```

Christophe Blaess, 1997.

TRADUCTION