Programmation des systèmes

Système de fichiers et entrées/sorties

Philippe MARQUET
Philippe.Marquet@lifl.fr
Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille

Licence d'informatique de Lille
décembre 2004
révision de janvier 2007

La dernière version de ce cours est accessible à
www.lifl.fr/~marquet/cnl/pds/

$Id: fs.tex,v 1.37 2012/11/27 23:17:54 marquet Exp $

Références

Unix, programmation et communication
Jean-Marie Rifflet et Jean-Baptiste Yunès
Dunod, 2003

The Single Unix Specification
The Open Group
www.unix.org/single_unix_specification/

man section 2

Table des matières

Mémoire persistante

Des besoins

- Processus manipule des données
  - conserve en mémoire
  - tout au long de son exécution
- Besoins de mémoriser de grandes quantités de données
  - taille supérieure à la mémoire (virtuelle)
- Besoins de conservation des données
  - au delà de la fin du processus
- Besoins de partage des données
  - données accessibles (simultanément) par plusieurs processus

Des fichiers

- Fichiers
  - mémoriser des données
  - sur disques (ou autres supports « externes »)
  - de manière persistante
- Fichier = entité gérée par le système d'exploitation
  - structuration, nommage, accès, protection, implantation...
  - système de fichiers = partie du système d'exploitation
- Fichier = mécanisme d'abstraction
  - présentation à l'utilisateur
  - implantation des systèmes de fichiers

Des répertoires

- Répertoire = fichier particulier
  - mémorise la structure du système de fichiers
  - opérations contrôlées par le système d'exploitation
- Fichier ordinaire
  - contient les données « utilisateur »

Pluralité des systèmes de fichiers

- Différents types de systèmes de fichiers
  - à l'origine fourni par un système d'exploitation
  - exemple : MS-DOS, ufs (Unix), ext2/ext3 (Linux), NTFS (Windows NT), HFS (Mac OS X)...
  - fournissent une même abstraction (à première vue...)
- Découplage système d'exploitation / système de fichiers
  - un système Linux peut « monter » un système de fichiers MS-DOS
- Système de fichiers = abstraction d'un disque
  - plusieurs disques
  - plusieurs systèmes de fichiers !
  - éventuellement de types différents !
  - vue unifiée des systèmes de fichiers présents = une unique hiérarchie

Organisation d'un système de fichiers

Hiérarchie

- Système de fichiers présente une hiérarchie
  - répertoire « *contient* » des fichiers
  - racine du système de fichiers
  - position courante dans la hiérarchie
- Système de fichiers n'est pas une hiérarchie
  - implantation sur la machine est un ensemble de nœuds
  - un nœud = un ensemble de blocs de données
  - détails d'implémentation cachés
- Le programmeur doit savoir que le système de fichiers n'est pas une hiérarchie
  - répertoire *contient une liste* de noms d'entrées
  - manipulation des liens symboliques
  - manipulation des liens physiques

Système de fichiers arborescent

- Le système de fichier est un arbre
  - vue simplificatrice (... sur laquelle on reviendra)
  - arbre = racine + nœuds à un parent unique + arcs
- Racine
  - notée `/`
  - est son propre parent
- Arcs ou entrées
  - nommés, tous caractères sauf `'\0'` et `'/'`
  - éviter les espaces, les non imprimables, et non ASCII
- Nœuds non terminaux
  - répertoires
  - toujours deux fils : `.` et `..`
  - `.` désigne le nœud lui-même, `..` désigne son père
- Nœud terminaux
  - fichiers standard
  - contiennent des données

Numérotation des nœuds

- Désignation d'un fichier sur le support matériel
  - numéro de périphérique (*device*)
  - numéro d'inœud (*inode*)
- Association d'une numérotation à un nœud
  - lien entre le nommage et le contenu
- Nommages multiples d'un nœud
  - de part les arcs `.` et `..`
  - (entre autres... à suivre)
  - accès au même contenu
  - partage des modifications du contenu

Chemins dans le système de fichiers

- Système de fichiers arborescent
  - nommage non ambigu d'un fichier depuis la racine
  - `/`(racine) nom de l'ascendant `/` ... `/` nom du parent `/` nom du fichier
  - caractère `/` de construction de nom de chemin
- Chemin absolu
  - interprété depuis la racine
  - précédé de `/` qui désigne la racine
  - `/home/licence/duchmol/pds/tp1/sc.c`
- Chemin relatif
  - position courante dans le système de fichier
  - interprété par rapport à cette position
  - `tp1/sc.c` ou `../pdc/projet/Makefile`

Liens multiples

- Entrées multiples pour un nœud
  - plusieurs entrées (arcs)
  - d'un même répértoire ou de répertoires différents
  - désignent le même nœud
- Lien physique
  - ensemble des liens désignant un même nœud
  - ensemble des chemins désignant un même nœud
- Non autorisé pour les répertoires
  - assurer la cohérence de la hiérarchie
- Nœud associé à un périphérique
  - couple (numéro de périphérique, numéro d'inœud)
  - pas de lien physique entre nœuds de périphériques différents

Différents types de fichiers

- Fichiers ordinaires
- Répertoires
- Liens symboliques
  - contient un nom de chemin
  - qui désigne un autre nœud
  - (à suivre...)
- Approche Unix = « tout » est fichier
  - tubes nommés, sockets : communications entre processus (à suivre...)
  - fichiers spéciaux associés aux périphériques
    - terminaux, cdrom, imprimantes...
    - une opération sur le fichier = une opération sur le périphérique
    - mode bloc ou caractère
  - utilisation des mêmes primitives

Liens symboliques

- Contient des données = chemin
  - chemin absolu, ou
  - chemin relatif
- Chemin désigné = chemin
  - chemin d'un répertoire, ou
  - chemin d'un fichier ordinaire
- Interprétation du nom
  - le lien symbolique lui-même, ou
  - le fichier qu'il désigne
  - peut dépendre du contexte d'utilisation

```
% rm symlink
% cat symlink
```


Liens symboliques *(cont'd)*

- Création par `ln -s`

```
% ls
foo.txt
% cat foo.txt
fOOOOOOoooo.....
% ln -s foo.txt bar.txt
% ls
bar.txt   foo.txt
% ls -l
total 16
lrwxr-xr-x  1 phm  phm   7 28 Dec 23:33 bar.txt -> foo.txt
-rw-r--r--  1 phm  phm  17 28 Dec 23:33 foo.txt
% cat bar.txt
fOOOOOOoooo.....
```

- Lien symbolique pas toujours valide

```
% rm foo.txt
% cat bar.txt
cat: bar.txt: No such file or directory
```


Cycles possibles dans la hiérarchie

- Structuration des répertoires
  - entrées `.` et `..`
- Liens symboliques
  - raccourcis
  - mais aussi des cycles

```
% cd /tmp
% ln -s /tmp foo
% cd foo/foo/foo
% pwd
/tmp

% ln -s bar bar
% cd bar
bar: Too many levels of symbolic links.
```


Parcours d'une hiérarchie

- Traitement récursif
  - commandes `find`, `du`, etc.
- Type de fichier ?
  - ordinaire ⇒ traitement standard
  - répertoire ⇒ itérer sur toutes les entrées
    - sauf `.` et `..`
  - lien symbolique ⇒ suivre ou pas ?
- Suivre les liens symboliques
  - choix de la commande, ou option
  - mémoriser les nœuds visités, et/ou
  - limiter le nombre maximal de liens symboliques traversés

Montage de systèmes de fichiers

- Partition
  - disque logique
  - division d'un disque physique
  - périphérique
  - disque distant (serveur via réseau)
- Point de montage
  - arborescence unique
  - ensemble des partitions positionnées dans l'arborescence
- Commandes `mount` et `df`

```
% mount -t ext3
/dev/hda1 on / type ext3 (rw,noatime)
/dev/hda8 on /tmp type ext3 (rw,noatime)
/dev/hda5 on /usr type ext3 (rw,noatime)
/dev/hda6 on /var type ext3 (rw,noatime)
```


Montage de systèmes de fichiers *(cont'd)*

- Commandes `mount` et `df` (suite)

```
% mount -t vfat
/dev/sda1 on /mnt/removable type vfat (rw)
%
% mount -t nfs
molson:/usr/export on /usr1 type nfs (rw,addr=172.16.12.1)
molson:/home/enseign on /home/enseign type nfs (rw,addr=172.16.12.1)
molson:/home/admin on /home/admin type nfs (rw,addr=172.16.12.1)
%
% df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda1             291M   97M  180M  35% /
/dev/hda8             7.1G   80M  6.7G   2% /tmp
/dev/hda5             8.1G  3.1G  4.6G  41% /usr
/dev/hda6             2.0G  146M  1.7G   8% /var
molson:/usr/export    4.5G  2.3G  2.0G  54% /usr1
molson:/home/enseign   51G   23G   26G  47% /home/enseign
molson:/home/admin     51G   23G   26G  47% /home/admin
/dev/sda1             499M  496M  3.2M  99% /mnt/removable
```


Opérations sur les fichiers

- Informations
  - numéro périphérique, numéro inœud
  - type du fichier, taille...
  - dates...
  - propriétaire et groupe propriétaire
  - droits
- Parcours de la hiérarchie
  - listage
  - déplacement dans la hiérarchie
- Modification de la hiérarchie
  - création, destruction de nœuds
  - liens physiques et symboliques
- Écriture et lecture des données des fichiers ordinaires

Interface POSIX de manipulation d'un système de fichiers

Informations d'un fichier

- Structure `struct stat`
  - retournée par

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

int stat(const char *path, struct stat *sb);
int lstat(const char *path, struct stat *sb);
```

  - `stat()` → fichier désigné par un lien symbolique
    `lstat()` → le lien lui-même
- Identification du nœud dans le système
  - partition et inœud

```
struct stat {
    dev_t     st_dev;
    ino_t     st_ino;
    ...
```

  - nombre de liens physiques sur le fichier

```
struct stat {
    ...
    nlink_t   st_nlink;
    ...
```


Informations d'un fichier *(cont'd)*

- Type et droits
  - propriétaire, groupe propriétaire, mode

```
struct stat {
    ...
    mode_t    st_mode;
    uid_t     st_uid;
    gid_t     st_gid;
    ...
```

  - informations propriétaire

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd {
    char    *pw_name;   /* User's login name. */
    uid_t    pw_uid;    /* Numerical user ID. */
    gid_t    pw_gid;    /* Numerical group ID. */
    char    *pw_dir;    /* Initial working directory. */
    char    *pw_shell;  /* Program to use as shell. */
}
```


Informations d'un fichier *(cont'd)*

- Type et droits (suite)
  - type du fichier = utilisation de macros sur la champ `st_mode`
    `S_ISREG(m)` Fichier ordinaire
    `S_ISDIR(m)` Répertoire
    `S_ISLNK(m)` Lien symbolique
    `S_ISFIFO(m)`, `S_ISBLK(m)`, `S_ISCHR(m)`, `S_ISSOCK(m)` etc.
  - droits du propriétaire = positions binaires `S_IRWXU`
    `S_IRUSR` lecture
    `S_IWUSR` écriture
    `S_IXUSR` exécution
  - droits du groupe et des autres

Informations d'un fichier *(cont'd)*

```
struct stat sb;
int status;

status = stat(pathname, &sb);
if (status) {
    perror("appel de stat");
    return;
}
if (S_ISREG(sb.st_mode)) {
    printf("Fichier ordinaire");
    if (sb.st_mode & S_IXUSR)
        printf(", executable par son proprietaire");
}
```

```
% ls -l foo.txt bar
drwxr-xr-x  2 root  wheel  4 29 Dec 00:04 bar
-rw-r--r--  1 root  wheel  4 29 Dec 00:04 foo.txt
```

Informations d'un fichier *(cont'd)*

- Taille
  - longueur en octets et place occupée sur le disque

```
struct stat {
    ...
    off_t     st_size;
    blkcnt_t  st_blocks;
    ...
```

  - peuvent ne pas correspondre
    - allocation unitaire de blocs
    - non allocation de blocs non utilisés (fichiers creux, à suivre...)
  - lien symbolique : longueur du chemin pointé par le lien

Informations d'un fichier *(cont'd)*

- Dates de modifications
  - dernier accès (a)
  - dernière modification du contenu (m)
  - dernière modification du contenu ou des méta-données (propriétaire...) (c)

```
struct stat {
    ...
    time_t    st_atime;
    time_t    st_mtime;
    time_t    st_ctime
    ...
```

  - en secondes depuis l'« Epoch », 1er janvier 1970
  - conversion en une chaîne de caractères pour affichage par

```
#include <time.h>

char *ctime(const time_t *time);
```


Droits d'accès

- Vérification du droit d'accès à un fichier
  - `#include <unistd.h>`

```
int access(const char *path, int amode);
```

  - droit défini par une combinaison « ou » des macros `R_OK`, `W_OK`, `X_OK`, et `F_OK`
  - exemple:

```
if (! access(pathname, X_OK))
    printf("Fichier exécutable\n");
```

- Positionner les droits d'un fichier
  - `#include <sys/stat.h>`

```
int chmod(const char *path, mode_t mode);
```

  - spécification de `mode` sous forme d'un masque binaire (`S_IRUSR`, `S_IWUSR`, `S_IXUSR`, etc.)

Lecture d'un fichier

- Lire un fichier ?
  - type du fichier
- Fichier ordinaire
  - accéder aux données
- Répertoire
  - accéder à la liste des entrées
- Lien symbolique
  - pointe sur un fichier ordinaire : accéder aux données
  - pointe sur un répertoire : accéder à la liste des entrées
  - accéder au nom du fichier pointé
- Fichier spécial
  - accéder aux données d'un périphérique
  - identique à l'accès à un fichier ordinaire
  - des limitations possibles
  - des opérations spécifiques possibles

Parcours des répertoires

- Contenu d'un répertoire = liste de liens
  - itération de la liste
- Descripteur de répertoire
  - ouverture et fermeture

```
#include <dirent.h>

DIR *opendir(const char *dirname);
int closedir(DIR *dirp);
```

- Itération sur les entrées du répertoire
  - obtenir les informations d'une entrée :

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

  - numéro d'inœud et nom :

```
struct dirent {
    ino_t d_ino;
    char  d_name[];
```


Parcours des répertoires *(cont'd)*

- Exemple : recherche dans le répertoire courant

```
static int
lookup(const char *name)
{
    DIR *dirp;
    struct dirent *dp;

    if ((dirp = opendir(".")) == NULL) {
        perror("couldn't open '.'");
        return 0;
    }

    while ((dp = readdir(dirp))) {
        if (! strcmp(dp->d_name, name)) {
            printf("found %s\n", name);
            closedir(dirp);
            return 1;
        }
    }

    if (errno != 0)
        perror("error reading directory");
    else
        printf("failed to find %s\n", name);
    closedir(dirp);
    return 0;
}
```


Fichier pointé par un lien symbolique

- Lire le contenu d'un lien
  - le chemin du fichier pointée

```
#include <unistd.h>

ssize_t readlink(const char *path, char *buf, size_t bufsize);
```

  - retourne le nombre de caractères du chemin
    -1 en cas d'erreur
- Utilisation typique

```
char buf[PATH_MAX+1];
ssize_t len;

if ((len = readlink(path, buf, PATH_MAX)) != -1)
    buf[len] = '\0';
else
    perror("error reading symlink");
```


Création

- Créer une entrée dans un répertoire
  - répertoire existant
  - sinon le créer préalablement → itération
  - nécessite le droit d'écriture sur le répertoire
- Masque des droits
  - variable système `umask` du processus
  - modifiée par

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

mode_t umask(mode_t cmask);
```

  - qui retourne l'ancien masque
  - paramètre `mode` d'une primitive de création
  - droits de l'entrée créée = `mode & ~umask`

Création (cont'd)

- Créer un répertoire
  - primitive

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

int mkdir(const char *path, mode_t mode);
```

  - exemple

```
int status;

status = mkdir("/tmp/dir", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

  - résultat

```
% mmkdir
% ls -ld /tmp/dir
drwxr-xr-x  1 phm  phm  4 29 Dec 00:32 /tmp/dir
% umask
022
% umask 002
% rmdir /tmp/dir
% mmkdir
% ls -ld /tmp/dir
drwxrwxr-x  1 phm  phm  4 29 Dec 00:34 /tmp/dir
```


Création (cont'd)

- Créer un lien physique (dur)
  - `#include <unistd.h>`

```
int link(const char *old, const char *new);
```

  - `old` doit exister, ne pas être un répertoire
  - `new` ne doit pas exister
- Créer un lien symbolique
  - `#include <unistd.h>`

```
int symlink(const char *name, const char *new);
```

  - `name` peut exister ou non
  - `new` ne doit pas exister
- Créer un fichier
  - à suivre...

Destruction

- Destruction d'un nœud
  - supprimer une entrée dans la hiérarchie
  - supprimer l'inœud si dernière entrée
  - implémentation : compteur de références sur un inœud
- Détruire un répertoire
  - `#include <unistd.h>`

```
int rmdir(const char *path);
```

  - doit être vide
- Détruire un fichier
  - `#include <unistd.h>`

```
int unlink(const char *path);
```

  - fichier ordinaire, ou lien symbolique...

Interface POSIX de lecture/écriture dans un fichier

Lire / écrire dans un fichier

- Fichiers ordinaires
  - suite d'octets (caractères)
  - taille quelconque
  - accès par

```
ouverture ;
while () {
    lectures/écritures ;
}
fermeture ;
```

  - curseur : position courante
- Descripteur de fichier
  - référence sur un fichier ouvert
- Bonne pratique
  - la fonction qui ouvre un fichier le referme
  - ou couple : ouverture/fermeture
  - toujours refermer

Ouverture

- Ouvrir un fichier
  - obtenir un descripteur de fichier

```
#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

  - position au début du fichier
  - différents mode d'accès et options précisés par `oflag`
- Mode d'accès
  - exactement un parmi les suivants
  - lecture seule : `O_RDONLY`
  - écriture seule : `O_WRONLY`, ou
  - lecture/écriture : `O_RDWR`

Ouverture (cont'd)

- Options d'accès
  - aucun, un ou plusieurs parmi les suivants
  - `O_APPEND` : mode ajout
  - `O_CREAT` création du fichier s'il n'existe pas
    - 3e paramètre de type `mode_t`
  - `O_EXCL` et `O_CREAT` : échec si le fichier existe déjà
  - `O_TRUNC` (et `O_WRONLY` ou `O_RDWR`) si le fichier existe, le tronquer à zéro
  - `O_NONBLOCK`, `O_DSYNC`, `O_RSYNC`, `O_SYNC` à suivre...

Ouverture (cont'd)

- Utilisations typiques
  - lecture d'un fichier (que l'on veut) existant

```
int fd = open(path, O_RDONLY);
if (fd == -1) {
    perror("ouverture du fichier");
```

  - création d'un fichier (et troncature si existant)

```
open(path, O_WRONLY | O_CREAT | O_TRUNC,
     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
```

ou

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

  - pour un fichier exécutable

```
open(path, O_WRONLY | O_CREAT | O_TRUNC,
     S_IRWXU | S_IRWXG | S_IRWXO);
```

ou

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, 0777);
```

  - ajout à un fichier existant

```
open(path, O_WRONLY | O_APPEND | O_CREAT,
     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
```

Ouverture *(cont'd)*

- Utilisations concurrentes
  - accès exclusif par un processus à un fichier
    → assurer cette exclusivité
  - accès partagés
    → assurer une cohérence des écritures
- Accès exclusif
  - accès exclusif en écriture

```
open(path, O_WRONLY | O_EXCL | O_CREAT,
     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
```

  - échec si le fichier existe déjà
  - succès pour unique processus
  - le fichier peut servir de verrou
    - le processus propriétaire du verrou réalise une commande de manière exclusive et atomique
    - il rend le verrou en détruisant le fichier par `unlink()`
  - exemple : accès quelconque à un autre fichier...

Ouverture *(cont'd)*

- Accès exclusif quelconque à un fichier
  - accès gardé par un fichier verrou

```
char *filename = "...";
char lockfile[PATH_MAX+1];
int lock, fd;

sprintf(lockfile, "%s.lock", filename);

/* acquisition du verrou */
lock = open(lockfile, O_WRONLY | O_EXCL | O_CREAT,
            S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
if (lock == -1) {
    printf("occupe, essayez plus tard\n");
    return;
}

/* ouverture non concurrente de filename */
fd = open(filename, O_RDWR | O_CREAT,
          S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

/* utilisation de fd */
...

/* fermeture du fichier et destruction du verrou */
close(fd);
unlink(lockfile);
```


Ouverture *(cont'd)*

- Accès multiples en écriture
  - ouverture simultanée, en ajout par plusieurs processus

```
open(path, O_WRONLY | O_APPEND | O_CREAT,
     S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
```

  - multiples écritures par de multiples processus
  - les écritures se font toujours en fin de fichier
  - pas de perte de données

Ouverture *(cont'd)*

- Ouverture à la création du processus
  - réalisée automatiquement par le système
  - entrées/sorties standard
  - descripteur `STDIN_FILENO` (0) : entrée standard
  - descripteur `STDOUT_FILENO` (1) : sortie standard
  - descripteur `STDERR_FILENO` (2) : sortie d'erreur

Fermeture

- Nécessité de fermer un descripteur après utilisation
  - libération de ressources
  - nombre maximal de fichiers ouverts
  - `#include <unistd.h>`

```
int close(int fd);
```


Opérations à partir d'un descripteur de fichier

- Lecture/écriture
  - à suivre...
- Quelques « f fonctions »
  - informations

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

int stat(const char *path, struct stat sb);
int lstat(const char *path, struct stat sb);
int fstat(int fd, struct stat sb);
```

  - changement des droits

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

int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

  - etc.

Lecture / écriture

- Transfert de/vers le fichier d'une zone mémoire
  - `#include <unistd.h>`

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

  - fichier : descripteur ouvert `fd`
  - taille transfert : `nbyte`
  - zone mémoire : `buf`, assez grande !
- Retourne nombre d'octets lus/écrits
  - normalement `nbyte`
  - inférieur à `nbyte` en lecture à l'approche de la fin de fichier
  - 0 en cas de lecture en fin de fichier
  - (-1 en cas d'erreur)
- Déplace la position courante
  - du nombre d'octets effectivement lus/écrits
- Fichier de taille arbitraire
  - fichier grandit si écriture au delà de la fin de fichier

Exemple : copie de fichier

mcp1.c

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

#define BUFSIZE 4096

static void
copy_file(const char *src, const char *dst)
{
    int fdsrc, fddst;
    char buffer[BUFSIZE];
    int nchar;

    fdsrc = open(src, O_RDONLY);
    fddst = open(dst, O_WRONLY | O_CREAT | O_TRUNC,
                 S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

    while ((nchar = read(fdsrc, buffer, BUFSIZE))) {
        write(fddst, buffer, nchar);
    }

    close(fdsrc);
    close(fddst);
}
```

- problème ?

Exemple : copie de fichier *(cont'd)*

```
% ./mcp1 foo gee
% diff foo gee
% ls -li foo gee
 807930 -rw-r--r--    2 phm     phm     38 29 Dec 01:09 foo
 807933 -rw-r--r--    2 phm     phm     38 29 Dec 01:10 gee

% ln foo bar
% ls -li foo bar
 807930 -rw-r--r--    2 phm     phm     38 29 Dec 01:09 bar
 807930 -rw-r--r--    2 phm     phm     38 29 Dec 01:09 foo

% ./mcp1 foo bar
% diff foo bar
% ls -li foo bar
 807930 -rw-r--r--    2 phm     phm      0 29 Dec 01:11 bar
 807930 -rw-r--r--    2 phm     phm      0 29 Dec 01:11 foo
```


Exemple : copie de fichier *(cont'd)*

mcp2.c

```
static void
copy_file(const char *src, const char *dst)
{
    struct stat stsrc, stdst;
    int fdsrc, fddst;
    char buffer[BUFSIZE];
    int nchar;

    lstat(src, &stsrc);
    lstat(dst, &stdst);

    if (stsrc.st_ino == stdst.st_ino && stsrc.st_dev == stdst.st_dev) {
        fprintf(stderr, "%s et %s sont le meme fichier\n", src, dst);
        return;
    }

    fdsrc = open(src, O_RDONLY);
    fddst = open(dst, O_WRONLY | O_CREAT | O_TRUNC,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    ...
```

- exercice : ajouter les traitements des erreurs...

Interface POSIX : opérations avancées

Lecture / écriture indirecte

- Lecture/écriture de zones mémoire non contiguës
  - notion de vecteur d'entrées/sorties (*I/O vector*)
  - peut éviter une copie mémoire
  - peut factoriser des appels systèmes
  - `#include <sys/uio.h>`

```
struct iovec {
    void   *iov_base; /* Base address of a memory region for
                         input or output. */
    size_t  iov_len;  /* The size of the memory pointed to
                         by iov_base. */
};

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Lecture / écriture indirecte (cont'd)

printenv-iov.c

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>

int
main (int argc, char *argv[], char **arge)
{
    struct iovec *iov;
    int argl, i;

    /* number of strings in arge */
    for (argl=0; arge[argl]; argl++)
        ;

    /* iovect creation */
    iov = malloc(argl * sizeof(struct iovec));
    assert(iov);

    for (i=0; i<argl; i++) {
        iov[i].iov_base = arge[i];
        iov[i].iov_len = strlen(arge[i]);
    }

    writev(STDOUT_FILENO, iov, argl);
    exit(EXIT_SUCCESS);
}
```


Positionnement

- Déplacer la position courante
  - écritures/lectures se font à la position courante
  - écritures/lectures modifient la position courante
  - modification « explicite » de cette position courante

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

  - déplacement de `offset` octets
  - `offset` peut être négatif
  - suivant `whence`, à partir de
    - la position courante `SEEK_CUR`
    - le début de fichier (a/c 0) `SEEK_SET`
    - la fin de fichier `SEEK_END`
  - retourne la nouvelle position absolue

Positionnement (cont'd)

- Positionnement parfois impossible / interdit
  - fichiers/périphériques sans capacité de positionnement
    - tubes, terminaux...
  - retourne erreur (-1) ; positionne `errno` (`ESPIPE`)

Positionnement (cont'd)

skable.c

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

    status = lseek(STDIN_FILENO, 0, SEEK_SET);

    if (status == -1) {
        fprintf(stderr, "Non seekable file\n");
        exit(EXIT_FAILURE);
    }

    fprintf(stderr, "Seekable file \n");
    exit(EXIT_SUCCESS);
}
```

```
% ./skable
Seekable file

% ./skable < /etc/passwd
Seekable file

% cat /etc/passwd | ./skable
Non seekable file
```

Positionnement *(cont'd)*

- Positionnement possible au delà de la fin de fichier
  - ne change pas la taille
  - écriture à cette position changera la taille
  - trou laissé entre l'ancienne fin de fichier et cette position : plein de zéro
  - fichier « creux »

Positionnement *(cont'd)*

hole.c

```
#define NZ    32768

static char bufa[] = "abcdef";
static char bufb[] = "xyz";

static void
create_file(const char *pathname)
{
    int fd, i;
    const int zero = 0;

    fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

    write(fd, bufa, sizeof(bufa));

    switch (option) {
        case OPT_SEEK:
            lseek(fd, NZ, SEEK_SET); break;
        case OPT_WRITE_ZEROES:
            for (i=sizeof(bufa); i<NZ; i++)
                write(fd, &zero, 1);
            break;
    }

    write(fd, bufb, sizeof(bufb));

    close(fd);
}
```


Positionnement *(cont'd)*

```
% ./hole -w foo-w
% ./hole -s foo-s
% od -c foo-s
0000000   a   b   c   d   e   f  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0100000   x   y   z  \0
0100004
% diff foo-w foo-s
% ls -l foo-*
-rw-r--r--  1 phm     phm     32772 Dec 30 02:36 foo-s
-rw-r--r--  1 phm     phm     32772 Dec 30 02:36 foo-w
% du -b foo-*
32772   foo-s
32772   foo-w
% du foo-*
8       foo-s
36      foo-w
```


Lecture / écriture indexée

- Pas d'utilisation de la notion de position courante
  - lecture/écriture à une position donnée
  - ne modifie pas la position courante
  - prototypes et sémantiques équivalents à `write()` et `read()`

```
ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
               off_t offset);
ssize_t pread(int fildes, void *buf, size_t nbyte,
              off_t offset);
```

  paramètre `offset` supplémentaire : position absolue / début du fichier

Lecture / écriture indexée *(cont'd)*

```
#define SIZEOF_PNGINT   4

static int
png_write_sizes(int fd, png_int x, png_int y)
{
    ssize_t lx, ly;

    lx = pwrite(fd, &x, SIZEOF_PNGINT, 16);
    ly = pwrite(fd, &y, SIZEOF_PNGINT, 20);

    return (lx == SIZEOF_PNGINT) && (ly == SIZEOF_PNGINT);
}
```

```
% cat /usr/share/file/magic
[...]
# PNG [Portable Network Graphics, or "PNG's Not GIF"] images
# 137 P N G \r \n ^Z \n [4-byte length] H E A D [HEAD data] [HEAD crc] ...
0       string          \x89PNG         PNG image data,
>4      belong          0x0d0a1a0a
>>16    belong          x               %ld x
>>20    belong          x               %ld,
>>24    byte            x               %d-bit
>>25    byte            0               grayscale,
>>25    byte            2               \b/color RGB,
>>25    byte            3               colormap,
>>25    byte            4               gray+alpha,
```


Lecture / écriture indexée *(cont'd)*

```
% file world*
world1.png: PNG image data, 20 x 22, 2-bit colormap, non-interlaced
world2.png: PNG image data, 20 x 22, 4-bit colormap, non-interlaced

% od -xa world1.png | head -4
0000000     8950    4e47    0d0a    1a0a    0000    000d    4948    4452
          89   P   N   G  cr  nl sub  nl nul nul nul  cr   I   H   D   R
0000020     0000    0014    0000    0016    0203    0000    00bd    2f54
        nul nul nul dc4 nul nul nul syn stx etx nul nul nul  bd   /   T

% od -xa world2.png | head -4
0000000     8950    4e47    0d0a    1a0a    0000    000d    4948    4452
          89   P   N   G  cr  nl sub  nl nul nul nul  cr   I   H   D   R
0000020     0000    0014    0000    0016    0403    0000    0032    6fa1
        nul nul nul dc4 nul nul nul syn eot etx nul nul nul   2   o  a1
```


Verrouillage

- Nécessité de contrôler les accès concurrents aux données
  - cohérence des données
  - écritures multiples en fin de fichier : cohérence assurée par le système si `O_APPEND`
  - écritures multiples quelconques ?
  - écriture et lecture simultanées ?
  - lectures multiples simultanées possibles
- Verrou (*lock*)
  - mécanisme général de contrôle d'accès
    - cas particulier des accès aux données d'un fichier
  - notion de propriétaire d'un verrou
    - opérations autorisées au seul propriétaire du verrou
  - prise et relâche d'un verrou
    - devient propriétaire
  - portée d'un verrou
    - ensemble des positions d'un fichier contrôlées par un verrou

Verrouillage *(cont'd)*

- Accès exclusif en écriture lors de la création d'un fichier
  - options `O_CREATE | O_EXCL` de `open()` (voir *supra*)
  - restrictif : création exclusive et non accès exclusif
  - granularité fixe : le fichier
- Portées des verrous POSIX
  - ensemble de positions d'un fichier
  - intervalle : $[\text{offset}_i..\text{offset}_f]$
  - intervalle infini : $[\text{offset}_i..\infty]$
- Deux types de verrous POSIX
  - verrous partagés ou verrous de lecture
  - verrous exclusifs ou verrous d'écriture

Verrouillage (cont'd)

- Deux comportements possibles vis-à-vis des verrous
  - mode consultatif (ou coopératif)
    - les opérations `read()`/`write()` sont toujours possibles
    - la pose d'un verrou empêche la pose de verrous incompatibles
    - le programme doit, de lui-même, consulter les verrous
  - mode impératif
    - le système contrôle les accès via `read()`/`write()` en fonction des verrous posés
    - verrou partagé : interdit les accès en écriture
    - verrou exclusif : interdit tout accès
  - pas de spécification POSIX du choix du mode (!)
    - habituellement mode coopératif

Verrouillage (cont'd)

- Type `struct flock` de description d'un verrou
  -
```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

struct flock {
    short l_type;   /* Type of lock; F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence; /* Flag for starting offset */
    off_t l_start;  /* Relative offset in bytes */
    off_t l_len;    /* Size; if 0 then until EOF */
    pid_t l_pid;    /* Process ID */
}
```
  - verrou partagé (`F_RDLCK`), verrou exclusif (`F_WRLCK`), ou déverrouillage (`F_UNLCK`)
  - `whence` parmi `SEEK_CUR`, `SEEK_SET`, et `SEEK_END`
  - portée [`l_start .. l_start+l_len`]
  - identification du processus détenant le verrou (consultation verrouillage)

Verrouillage (cont'd)

- Opérations de verrouillage
  - appel système
```
#include <fcntl.h>

int fcntl(int fildes, int cmd, struct flock *plock);
```
    (plus général que le verrouillage)
  - `cmd` parmi `F_SETLK`, `F_SETLKW`, `F_GETLK`
  - `F_SETLK` : demande non bloquante de prise du verrou
    - erreur -1 en cas de verrou incompatible : `EACCESS` ou `EAGAIN`
    - erreur -1 en cas d'interblocage (*deadlock*) : `EDEADLK`
  - `F_SETLKW` : demande bloquante de prise de verrou
    - attente que le verrou puisse être posé
    - erreur -1 en cas d'interblocage (*deadlock*) : `EDEADLK`
  - `F_GETLK` : test d'existence d'un verrou incompatible avec le verrou donné
    - verrou incompatible retourné dans `plock`

Options d'ouverture

- Différents modes de lecture/écriture
  - mode bloquant / non bloquant ; bloquant par défaut
  - mode synchronisé / non synchronisé ; non synchronisé par défaut
- Spécification du mode à l'ouverture du fichier (`open()`)
  - paramètre `oflag` : mode d'accès + options
- Spécification du mode postérieur à l'ouverture
  - fonction `fcntl()`, commandes `F_GETFL`/`F_SETFL` (*get*/*set flag*)
  - exemple : ajout d'un mode
```
fcntl(fd, F_SETFL,
          fcntl(fd, F_GETFL) | O_FLAG);
```

Options d'ouverture *(cont'd)*

- Mode bloquant / non bloquant
  - par défaut un appel `read()` est bloquant
    - lecture depuis un terminal, depuis un tube...
  - un appel `write` peut être bloquant
    - à cause d'un verrou impératif...
  - option `O_NONBLOCK` du mode d'ouverture : mode non bloquant
    - ```
      fcntl(fd, F_SETFL,
                fcntl(fd, F_GETFL) | O_NONBLOCK);
      ```
  - les appels à `read()`/`write()` ne sont jamais bloquants
  - situation de blocage : `read()`/`write()` retourne une erreur (-1) ; positionne `errno` (`EAGAIN`)

Options d'ouverture *(cont'd)*

- Mode synchronisé / non synchronisé
  - par défaut les écritures se font en mode non synchronisé
  - les données à écrire sont mémorisées dans des caches du système d'exploitation
  - de manière asynchrone, le système réalise les écritures des caches sur les disques
  - évite aux processus d'attendre les écritures disques
  - pas de garantie que l'écriture disque ait été réalisée...
  - option `O_SYNC` du mode d'ouverture : écritures en mode synchronisé
  - appel à `write` retourne quand les données sont écrites sur le disque

Projection mémoire

- Charger un fichier dans l'espace d'adressage du processus
  - ensemble ou partie du fichier
  - manipulation du fichier via l'adressage mémoire
- Évite de multiples copies vers/depuis les caches systèmes
  - lors de multiples écritures/lectures successives de mêmes positions
- Implantation
  - le fichier n'est pas lu dans son intégralité lors de la projection !
- Appel système `mmap()`

Projection mémoire *(cont'd)*

- Appel système `mmap()`
  - ```
    #include <sys/mman.h>

    void *mmap(void *addr, size_t len, int prot, int flags,
               int fildes, off_t off);
    ```
  - considère les données $[$`off`..`off`+`len`$[$ du fichier `fildes`
  - `addr` est une adresse de projection choisie ; la positionner à `NULL`
  - `prot` définit les accès possibles à la zone mémoire
    - `PROT_NONE`, aucun accès, ou
    - combinaison binaire de `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`
  - `flags` précise des options
    - répercutions des modifications sur toutes les projections du fichier ou non
  - retourne l'adresse de la projection
  - Appel
    ```
    int munmap(void *addr, size_t len);
    ```
    libère la zone mémoire de projection

Projection mémoire *(cont'd)*

mcp-mmap.c

```
static void
copy_file(const char *src, const char *dst)
{
    struct stat stsrc, stdst;
    int fdsrc, fddst;
    char *psrc;
    int size;

    lstat(src, &stsrc);
    lstat(dst, &stdst);

    if (stsrc.st_ino == stdst.st_ino && stsrc.st_dev == stdst.st_dev) {
        fprintf(stderr, "%s et %s sont le meme fichier\n", src, dst);
        return;
    }

    fdsrc = open(src, O_RDONLY);
    fddst = open(dst, O_WRONLY | O_CREAT | O_TRUNC,
                 S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

    size = stsrc.st_size;
    psrc = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fdsrc, 0);

    write(fddst, psrc, size);

    munmap(psrc, size);
    close(fdsrc);
    close(fddst);
}
```


Accès aux fichiers distants

- Accès aux fichiers distants
  - reposants sur un serveur
  - accessibles via le réseau
  - accès « identiques » aux accès locaux : `open()`, `read()`, etc.
  - montage NFS (*network file system*)
- Protocole NFS
  - émission de requêtes vers le serveur
    - recherche d'un fichier dans un répertoire, création, renommage de liens, accès/modification des attributs d'un fichier, lecture/écriture dans un fichier...
  - serveur sans état
- Conséquences
  - non respect de la sémantique POSIX
    - POSIX : droits à l'ouverture / NFS : ouverture à chaque lecture
  - serveurs sans état $\Rightarrow$ verrouillage impossible
  - gestion des droits/sécurité...

Bibliothèque C d'entrées/sorties

Pourquoi une bibliothèque

- Abstraction de plus haut niveau
  - entrées/sorties formatées
  - `fprintf()`, `fscanf()`, etc.
- Performances
  - entrées/sorties tamporisées (*bufferisées*)
  - réduire le nombre des coûteux appels système
  - coût d'un appel système $\approx$ 1000 instructions
  - écriture (/lecture) dans un tampon utilisateur
  - quand le tampon est plein (/vide), appel système `write()` (/`read()`)

Exemple : copie de fichier

mcp-bib.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

static void
copy_file(const char *src, const char *dst)
{
    struct stat stsrc, stdst;
    FILE *fsrc, *fdst;
    int c;

    lstat(src, &stsrc);
    lstat(dst, &stdst);

    if (stsrc.st_ino == stdst.st_ino && stsrc.st_dev == stdst.st_dev) {
        fprintf(stderr, "%s et %s sont le meme fichier\n", src, dst);
        return;
    }

    fsrc = fopen(src, "r");
    fdst = fopen(dst, "w");

    while ((c = fgetc(fsrc)) != EOF)
        fputc(c, fdst);

    fclose(fsrc);
    fclose(fdst);
}
```


Attention une bibliothèque

- Ne pas mixer les fonctions bibliothèques et appels système
  - la bibliothèque utilise des appels système !
- Bibliothèque tamporisée
  - attente que le tampon soit plein (/vide)
- Possible surcoût ?
  - copie supplémentaire
  - regagné par la factorisation des appels systèmes

Détails

- Voir un manuel / le manuel en ligne
- Voir votre/un cours de C
- Voir les exercices de TD/TP

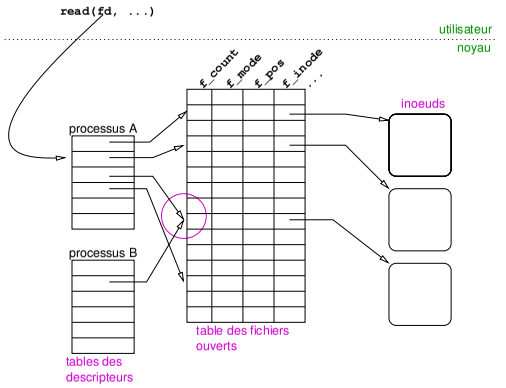
Quelques éléments d'implantation d'un système de fichiers

Éléments d'implantation d'un système de fichiers

- Implantation relève du cours master ASE
  - architecture et conception des systèmes d'exploitation
- Quelques éléments d'implantation « visibles » du monde utilisateur
  - partage des fichiers entre processus
  - messages d'erreur du système (commande `fsck`)
- Quelques éléments pour les « curieux »...

Tables des descripteurs / table des fichiers ouverts

- Descripteur de fichier
  - index dans la table des descripteurs *du processus*
  - une table des descripteurs par processus
- Table des fichiers ouverts
  - une table *unique* pour tous les processus
  - entrées *partagées* par tous les processus
  - `f_count` nombre de références
  - `f_mode` mode ouverture (lecture/écriture...)
  - `f_pos` position courante
  - `f_inode` inœud

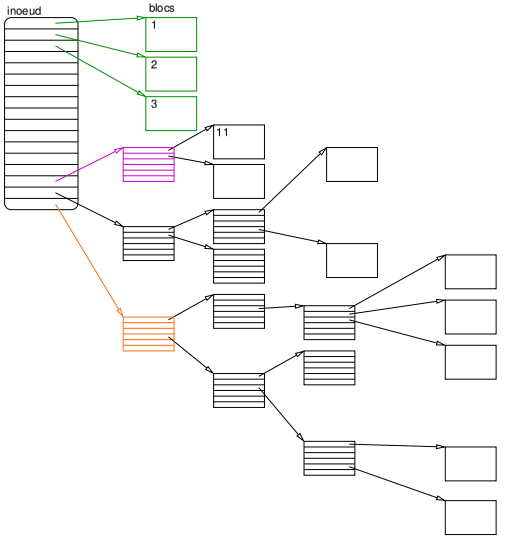


Structure de données inœud

- Inœud = données associées à un fichier
  - propriétaire, droits, date, taille fichier...
  - identification des *blocs* disques
- Inœud = informations pérennes
  - les inœuds sont stockés sur les disques !
  - copies en mémoire (cache)
- Organisation des données sur le disque
  - données regroupées en blocs
  - bloc $\pm \equiv$ secteur disque

Classique triple indirection

- Identification des blocs de données d'un fichier
  - inœud contient une liste de numéros de blocs
  - fichiers de taille variable
- Numéros de blocs
  - numéros des 10 premiers blocs de données
  - numéro d'un bloc contenant les numéros des blocs suivants...
- Tout est bloc !
  - blocs de données
  - blocs d'indirection, de double/triple indirection...
  - blocs contenant les inœuds



Performance = cache

- Accès disques lents
  - temps d'accès à la mémoire $\ll$ temps d'accès disque
  - débit & latence
- Mémoire cache (*buffer cache*)
  - garder en mémoire des données devant être écrites sur le disque, par exemple pour les blocs
  - dispositif logiciel ($\neq$ mémoire cache matérielle)
  - pas d'écriture systématique sur le disque à chaque modification
  - algorithme de recherche de la copie du « bloc » en mémoire (hachage)
  - algorithme d'éviction d'un bloc du cache plein (LRU, FIFO...)
  - vidage sur le disque $\Rightarrow$ données sauvegardées
  - non vidage sur le disque $\Rightarrow$ données non sauvegardées !
  - asynchronisme
  - vidage « prioritaire » de blocs sensibles : *master boot record*, inœud, blocs d'indirection...

Performance = cache *(cont'd)*

- Cache à tous les niveaux
  - bibliothèque d'entrées/sorties tamporisées
  - cache d'inœuds
  - cache de blocs (*buffer cache*)
- Vidage explicite du cache
  - appel système `sync()`
  - commande `sync`
  - démontage de périphériques amovibles (disquette, clé USB...)
- Anticipation des lectures
  - cache en lecture
  - lectures séquentielles de fichiers
  - heuristique : observer si les lectures d'un fichier se font bien séquentiellement

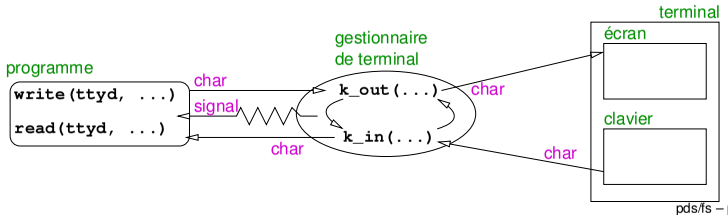
Gestion des terminaux

Terminaux

- Terminal
  - élément d'interaction entre l'utilisateur et une application
  - terminal physique : clavier + écran
  - pseudo-terminal : une application jouant le rôle de terminal
- Utilisations du terminal
  - « *fichier* » entrées et sorties
  - particulièrement les entrées et sorties standard
  - *contrôle* de la saisie (par exemple sans écho...) et de l'affichage

Interaction avec un terminal

- Un « programme » écrit...
- La chaîne de caractères est traitée par le « gestionnaire de terminal »
  - code du noyau
  - modification possible, p. e. fins de lignes...
- La chaîne de caractères est reçue par le terminal
  - matériel
  - interprétation comme une séquence de contrôle, ou
  - affichage
- Idem du clavier au programme
  - interruption / signal



Transmission des données & Écho

- Terminaux en « full duplex »
  - entrée et sortie simultanées
  - gestionnaire de terminal gère des tampons mémoires
    - un tampon d'entrée
    - un tampon de sortie
- Transmission des données, entrée
  - système fournit les caractères à la demande du programme
  - tampon plein ⇒ caractères perdus
- Transmission des données, sortie
  - tampon plein ⇒ écriture bloquante
- Gestion de l'écho
  - le gestionnaire de terminal est responsable de l'écho
  - tout caractère reçu dans le tampon d'entrée est placé dans le tampon de sortie
  - sauf mode sans écho...

Terminaux = fichiers

- Référence aux terminaux
  - fichiers spéciaux
  - `/dev/tty*`
  - terminal de contrôle du processus : `/dev/tty`
- Récupération du descripteur de fichier associé au terminal
  - ouvrir le fichier `/dev/tty` !

wrttty.c

```
int
main (int argc, char *argv[])
{
    int ttyd;
    int i, len;

    ttyd = open("/dev/tty", O_RDWR);
    assert(ttyd != 0);

    for (i=1; i<argc; i++) {
        len = strlen(argv[i]);
        argv[i][len] = '\n';
        write(ttyd, argv[i], len+1);
    }

    exit(EXIT_SUCCESS);
}
```

```
% ./wrttty foo bar gee
foo
bar
gee
% cat /dev/null | ./wrttty foo bar > /dev/null
foo
bar
```


Fichier = terminal ?

- Un (descripteur de) fichier donné est-il associé à un terminal ?
  - `#include <unistd.h>`

    `isatty(int fd);`
- Nom de ce terminal ?
  - `#include <unistd.h>`

    `char *ttyname(int fd);`

aretty.c

```
static void
tty_info(int fd, const char *name)
{
    if (isatty(fd))
        fprintf(stderr, "%s: %s\n", name, ttyname(fd));
    else
        fprintf(stderr, "%s: n'est pas un terminal\n", name);
}

int
main (void)
{
    tty_info(STDIN_FILENO, "entree std");
    tty_info(STDOUT_FILENO, "sortie std");

    exit(EXIT_SUCCESS);
}
```

```
% ./aretty
entree std: /dev/ttyp5
sortie std: /dev/ttyp5
% ./aretty > /dev/null
entree std: /dev/ttyp5
sortie std: n'est pas un terminal
% true | ./aretty
entree std: n'est pas un terminal
sortie std: /dev/ttyp5
```

Modes canonique et non-canonique

- Différentes configurations du gestionnaire de terminaux
  - mode canonique
  - mode non-canonique
- Mode canonique
  - caractère `<eol>` ou `<eof>` rend le tampon d'entrée disponible au programme
  - caractères de contrôle `<erase>`, `<kill>`... modife le tampon d'entrée
  - lecture par le programme à l'intérieur d'une seule ligne (taille bornée)
- Mode non-canonique
  - plus d'usage de la notion de ligne
  - caractères disponibles quand
    - `MIN` caractères tapés, ou
    - `TIME` dixièmes de secondes écoulés depuis la dernière frappe
  - `MIN` et `TIME` paramètres de configuration du gestionnaire de terminal

Exemple mode canonique

rdtty.c

```
#define BSIZE 10

int
main (int argc, char *argv[])
{
    char buf[BSIZE+1];
    ssize_t nread;

    while ((nread = read(STDIN_FILENO, buf, BSIZE)) > 0) {
        buf[nread] = '\0';
        printf("(%d) %s\n", nread, buf);
    }

    exit(EXIT_SUCCESS);
}
```

```
% ./rdtty
123
(4) 123

123456789_123456789_123
(10) 123456789_
(10) 123456789_
(4) 123

^D
%
```

- lecture par ligne, `read()` moins de caractères que demandés
- y compris le caractère de fin de ligne
- fin de fichier (`CTRL-D`, `^D`)
  ⇒ `read()` → 0
- saisie d'une très longue entrée
  ⇒ plus d'écho ; caractères perdus

Paramétrage d'un terminal

- Attributs d'un terminal
  - structure `struct termios`

```
#include <termios.h>

struct termios {
    tcflag_t  c_iflag;      /* Input modes. */
    tcflag_t  c_oflag;      /* Output modes. */
    tcflag_t  c_cflag;      /* Control modes. */
    tcflag_t  c_lflag;      /* Local modes. */
    cc_t      c_cc[NCCS];   /* Control characters. */
```

  - champs de type `tcflags_t` : champs de bits pour les quatre spécifications de modes
  - champ de type `cc_t` : caractères de contrôle du terminal
- Modes d'entrée, `c_iflag` (mineur)
  - traitements à appliquer aux caractères venant du terminal
  - exemple : `ISTRIP`, les caracères valides sont tronqués à 7 bits
  - exemple : `INLCR`, les caractères `<NL>` sont transformés en `<CR>`

Paramétrage d'un terminal *(cont'd)*

- Modes de sortie, `c_oflag` (mineur)
  - traitements à appliquer aux caractères envoyés au terminal
  - exemple : `OLCUC`, les lettres minuscules sont transformées en majuscules
  - exemple : `OCRNL`, les `<CR>` sont transformés en `<NL>`
- Modes de contrôle, `o_cflag` (mineur)
  - contrôle de la transmission au niveau matériel
  - exemple : `CS7`, ou `CS8` les caractères sont codés sur 7 ou 8 bits
  - exemple : `PARENB`, mécanisme de contrôle de parité activé (un bit de parité est ajouté à chaque caractère)

Paramétrage d'un terminal *(cont'd)*

- Modes locaux
  - les plus importants
  - définissent le traitement des caractères de contrôle
  - `ECHO` : les caractères en provenance du terminal sont remis sur le terminal
    désactivation pour la saisie des mots de passe...
  - `ICANON` : utilisation du terminal en mode canonique
  - `ISIG` : les caractères de contrôle provoquent l'envoi de signaux au processus

caractère symbolique	(habituellement)	signal	action
`<intr>`	(`CTRL-C`)	`SIGINT`	interruption
`<quit>`	(`CTRL-\`)	`SIGQUIT`	terminaison
`<susp>`	(`CTRL-Z`)	`SIGTSTP`	suspension

  - etc.

Paramétrage d'un terminal *(cont'd)*

- Caractères de contrôle, `c_cc`
  - association d'un caractère à un caractère symbolique
  - exemple associer le caractère `CTRL-H` au caractère symbolique `<erase>` d'effacement d'un caractère
  - caractères symboliques :
    - `<erase>` effacement un caractère
    - `<kill>` effacement de la ligne en cours
    - `<eof>` fin de fichier
    - `<eol>` fin de ligne
    - `<intr>` interruption
    - `<susp>` suspension, etc.
  - position dans le tableau `c_cc[]`
    `V<nom_du_caractere>`
  - exemple

```
struct termios tio;
tio.c_cc[VERASE] = 8; /* 8: code ascii de BS, backspace */
```


Paramétrage d'un terminal *(cont'd)*

- Caractères de contrôle, `c_cc`, (suite)
  - paramétrage du mode non canonique
  - caractères disponibles quand
    - `MIN` caractères tapés, ou
    - `TIME` dixièmes de secondes écoulés depuis la dernière frappe
  - exemple :

```
tio.c_cc[VMIN] = 10;     /* 10 caractères */
tio.c_cc[VTIME] = 10;    /* 1 seconde */
```


Gestion du paramétrage de terminaux

- Commande shell `stty`

```
% stty
speed 9600 baud;
lflags: echoe echoke echoctl pendin
oflags: -oxtabs
cflags: cs8 -parenb
```

- Fonctions POSIX
  - `#include <termios.h>`

```
int tcgetattr(int ttyd, struct termios *ptio);
int tcsetattr(int ttyd, int option, const struct termios *ptio);
```

  - valeurs possibles du paramètre `option`

`TCSANOW`	immédiatement
`TCSADRAIN`	après le vidage du tampon de sortie
`TCSAFLUSH`	`TCSADRAIN` + les caractères reçus sont oubliés

Gestion du paramétrage de terminaux (cont'd)

- Utilisation typique
  - récupérer les valeurs courantes
  - modifier quelques champs
  - installer ces nouvelles valeurs
  - exemple : positionner le mode sans écho, non canonique, rendre les caractères disponibles immédiatement

```
struct termios tio;

if (tcgetattr(STDIN_FILENO, &tio) == -1) {
    perror("tcgetattr");
    ...
}

tio.clflag &= ~(ICAN|ECHO);
tio.c_cc[VMIN] = 1;

if (tcsetattr(STDIN_FILENO, TCSANOW, &tio) == -1) {
    perror("tcsetattr");
    ...
}
```