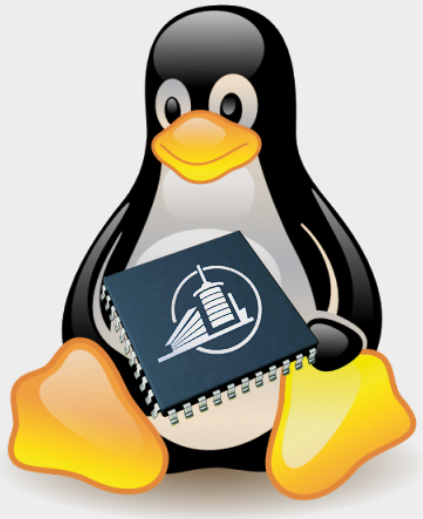


Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

CSEL1 – Construction Systèmes Embarqués sous Linux

# Programmation système: Système de fichiers

HES-SO//Master TIC/TIN 2020

Daniel Gachet – HEIA-FR – Télécommunications

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

# Contenu

- Introduction
- Traitement des erreurs
- Traitement des fichiers ordinaires
- Gestion des répertoires
- Surveillance de changements dans le système de fichiers
- Traitement des fichiers spéciaux
- Multiplexage des entrées/sorties

MASTER OF SCIENCE IN ENGINEERING

MSE

# Introduction

MASTER OF SCIENCE IN ENGINEERING

MSE

# Programmation système – Introduction

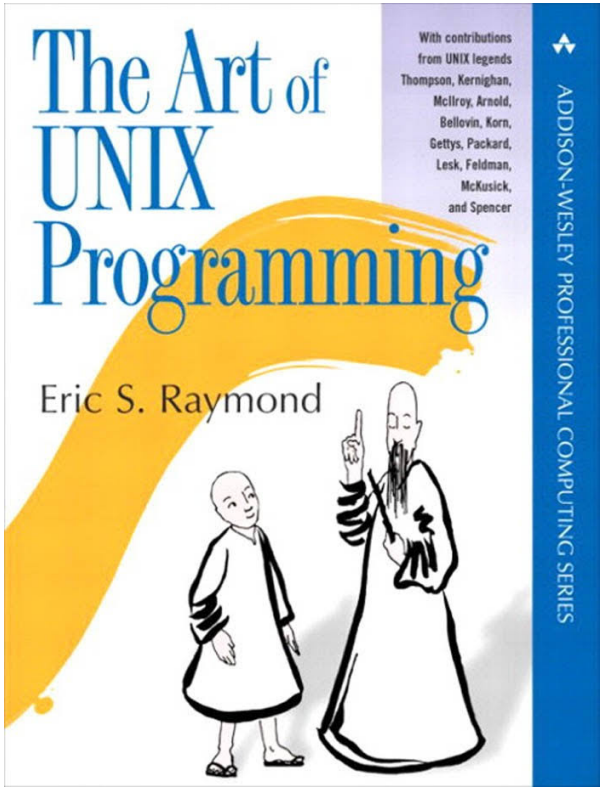
- **Qu'est-ce que la programmation système?**
  - La programmation système se fait en espace utilisateur, mais à un niveau très bas, très proche des interfaces proposées par le noyau
  - Elle offre des fonctionnalités à des applications plus évoluées afin de leur permettre d'effectuer leurs tâches
  - Les programmes système offrent des services que le système lui-même n'offre pas ou pas encore
- **Quelles compétences faut-il avoir pour concevoir des programmes système?**
  - Pour développer des programmes système, il faut maîtriser les interfaces offertes par le noyau (system programming API), les librairies (y compris la librairie standard) et les outils de développement (compilateur, debugger, …)
- **Qu'offre la connaissance de la programmation système?**
  - La connaissance de la programmation système offre une meilleure vue d'ensemble du système Linux, même si on ne l'utilise pas chaque jour

MASTER OF SCIENCE IN ENGINEERING

MSE

# La philosophie Unix

- **La philosophie Unix/Linux est fondée sur 4 paradigmes**
  - KISS (Keep It Simple and Stupid)
  - Faire une chose et le faire bien
  - Tout est fichier
  - Les données sont du texte




MASTER OF SCIENCE IN ENGINEERING
MSE

# KISS – Faire une chose et le faire bien

- **Les grands programmes monolithiques sont difficiles à comprendre et souvent très complexes à maintenir. Ils souffrent souvent d'effets de bord les rendant moins robustes et moins fiables.**
- **Il est plus judicieux de concevoir des applications basées sur de petits programmes simples, nettement plus facile a maintenir et plus évolutifs.**
- **Le système d'exploitation Linux offre divers outils et services pour autoriser de tels designs.**
- **Exemples**
  - File redirection

```
$ cat readfile > savefile
$ ls -l > savefile
$ grep "text" < filelist.txt
```

  - Pipes

```
$ cat long_file.txt | wc
$ find -name "*" | xargs grep hallo
```

MASTER OF SCIENCE IN ENGINEERING

MSE

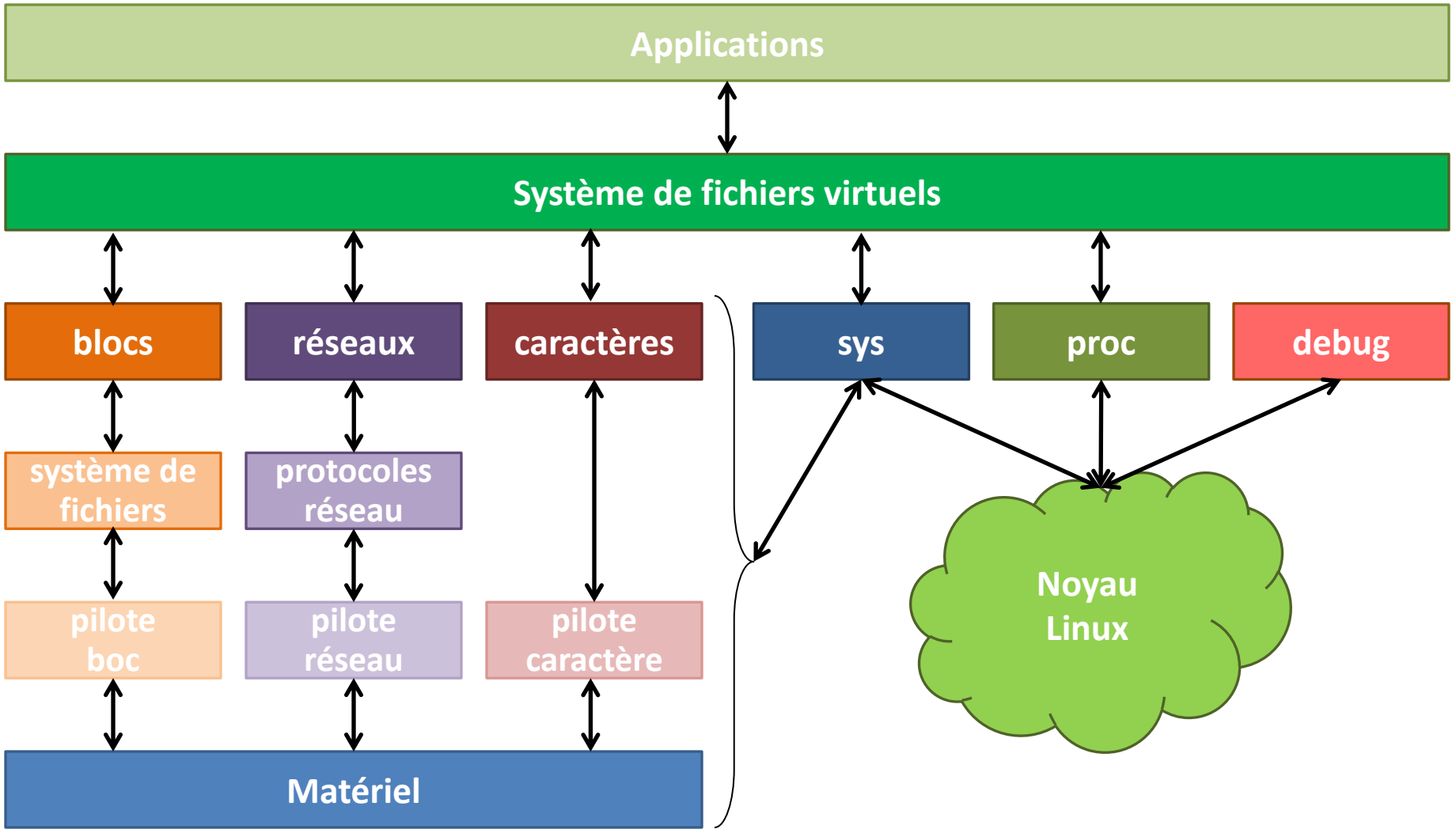
# Tout est fichier - Les données sont du texte

- **Under Linux everything is a file!**
  - Sous Linux, un fichier est beaucoup plus que de simples données stockées sur un disque (disque dur magnétique, flash disk, disque réseau, …)
  - Sur les systèmes embarqués, hormis l'accès aux données stockées dans des fichiers ordinaires, il est important de pouvoir accéder de manière simple et efficace aux autres ressources du système (périphériques, noyau, réseau, …)
- **Linux implémente un système de fichiers virtuels permettant de représenter**
  - Des fichiers ordinaires (stockage de données sur disque)
  - Des périphériques d'entrées/sorties (devfs: `/dev`)
  - Des données (status/configuration) du noyau Linux (procfs: `/proc`)
  - Des données (status/configuration) des pilotes (sysfs: `/sys`)
  - Des données de débogage du noyau (debugfs: `/sys/kernel/debug`)
  - Des périphériques réseau (network interfaces)
- **Les données sont du texte**
  - La plupart des données stockées dans les fichiers spéciaux (procfs, sysfs, debugfs,…) sont du texte
  - Ce format permet d'être manipulé très aisément à l'aide d'outils tels que `echo` et `cat`
  - Les fichiers de configuration sont également stockés sous forme de fichier texte

MASTER OF SCIENCE IN ENGINEERING

MSE

# Système de fichiers virtuels – Rappel




MASTER OF SCIENCE IN ENGINEERING
MSE

# Accès aux fichiers

- **Seuls quatre services/opérations sont nécessaires pour accéder en lecture et en écriture à des fichiers**

```
int fd = open (…);
```

```
read (fd, …);
```

```
write (fd, …);
```

```
close(fd);
```

- **Il existe naturellement bien d'autres opérations…**

# Interfaces avec le système de fichiers virtuels

- **Linux propose diverses interfaces pour accéder au système de fichiers virtuels**
  - File I/O
  - Standard I/O (ou Buffered I/O)
  - File and Directory Management
  - Surveillance de changements dans le système de fichiers
  - Multiplexage des entrées/sorties

- **Comportement des services**
  - Bien que les services proposés par ces interfaces sont disponibles pour tous les types de fichiers, ils peuvent se comporter légèrement différemment sur des fichiers ordinaires que sur des fichiers spéciaux.

MASTER OF SCIENCE IN ENGINEERING

MSE

# Traitement des erreurs

MASTER OF SCIENCE IN ENGINEERING

MSE

# Traitement des erreurs

- **Sous Linux, il est usuel que l'appel à des fonctions système retourne une valeur entière signée (`int syscall()`). Si la valeur retournée est `0` ou supérieur à `0`, elle indique le succès de l'opération. Par contre, en cas d'erreur, on obtient généralement `-1`.**

- **Le détail de l'erreur (sa cause / son type), peut être obtenu par l'intermédiaire de la variable globale «`extern int errno;`» déclarée dans le fichier `<errno.h>`, lequel définit aussi les codes d'erreurs du système, p.ex.:**

| *Code* | *Description* | *Code* | *Description* |
|---|---|---|---|
| EPERM | Operation not permitted | EXDEV | Cross-device link |
| ENOENT | No such file or directory | ENODEV | No such device |
| ESRCH | No such process | ENOTDIR | Not a directory |
| **EINTR** | **Interrupted system call** | **EISDIR** | **Is a directory** |
| EIO | Ierror | EINVAL | Invalid argument |
| ENXIO | No such device or address | ENFILE | File table overflow |
| E2BIG | Argument list too long | EMFILE | Too many open files * |
| ENOEXEC | Exec format error | ENOTTY | Not a typewriter |
| EBADF | Bad file number | ETXTBSY | Text file busy |
| ECHILD | No child processes | EFBIG | File too large |
| **EAGAIN** | **Try again** | **ENOSPC** | **No space left on device** |
| ENOMEM | Out of memory | ESPIPE | Illegal seek |
| EACCES | Permission denied | EROFS | Read-only file system |
| EFAULT | Bad address | EMLINK | Too many links |
| ENOTBLK | Block device required | EPIPE | Broken pipe |
| EBUSY | Device or resource busy | EDOM | Math argument out of domain of func |
| EEXIST | File exists | ERANGE | Math result not representable |

# Traitement des erreurs (II)

- **La bibliothèque standard C fournit plusieurs services pour convertir les codes d'erreurs en une représentation textuelle et les afficher sur la console.**

- **Exemple 1**

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

int ret = fonction();
if (ret == -1)
    perror("ERROR");
```

**Exemple 2**

```
#include <stdio.h>
#include <stdio.h>
#include <errno.h>

int ret = fonction();
if (ret == -1) {
   char estr[100] = {[0]=0,};
   strerror_r(errno, estr, sizeof(estr)-1);
   fprintf (stderr, "ERROR: %s", estr);
}
```

- **Remarque**
  - La méthode `strerror` n'est pas « thread-safe ». Il est préférable d'utiliser la méthode `strerror_r`
  - Bien que la variable `errno` soit globale, sous Linux, elle est stockée par thread et par conséquent sûre.

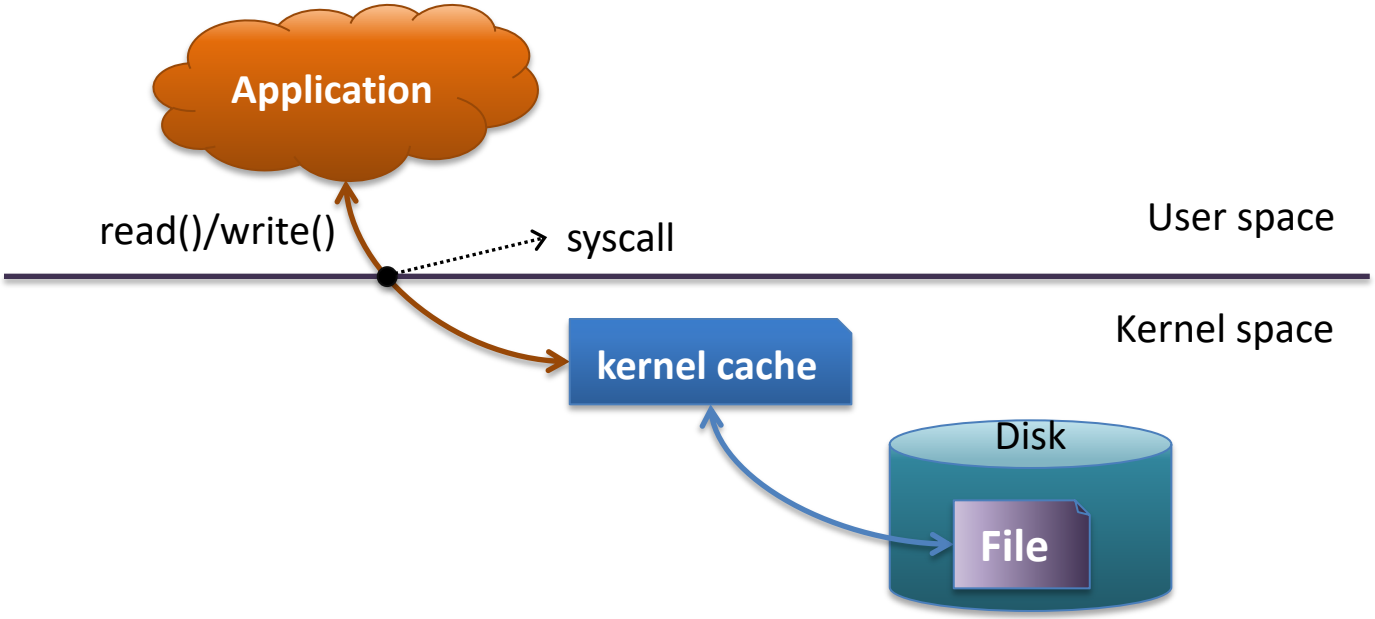
# Traitement des fichiers ordinaires

MASTER OF SCIENCE IN ENGINEERING

MSE

# File I/O – Fichiers ordinaires

- **Propriétés**
  - File I/O propose une interface au niveau du noyau Linux pour la gestion de fichiers ordinaires et spéciaux.
  - Pour les fichiers ordinaires, le noyau implémente un cache des données par blocs de 4KiB usuellement.
  - Le nombre de fichiers qu'un processus peut manipuler est limité à 1024 par défaut. Celui-ci peut cependant être augmenté à l'aide de la commande
    `$ ulimit –n`



# File I/O – Opérations

- **Linux propose une large palette de services pour traiter avec les fichiers ordinaires**
  - Ouverture d'un fichier (syscall: open)
  - Création d'un fichier (syscall: creat)
  - Lecture du contenu d'un fichier (syscall: read)
  - Ecriture de données dans un fichier (syscall: write)
  - Positionnement dans le fichier (syscall: lseek)
  - Troncation d'un fichier (syscall: ftruncate)
  - Mise en mémoire d'un fichier (syscall: mmap)
  - Lecture/écriture & positionnement combinés (syscall: pread/pwrite)
  - Lecture des métadonnées (status) d'un fichier (syscall: fstat)
  - Synchronisation des données avec le disque (syscall: fsync)
  - Fermuture d'un fichier (syscall: close)

# File I/O – Ouverture d'un fichier

- **L'ouverture d'un fichier est obtenue à l'aide de l'appel système `open()`**

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

- **Exemple**

```
int fd = open ("/home/lmi/myfile", O_RDONLY);
if (fd == -1)
     /* error*/
```

- **Comportement**
  - La fonction `open()` associe le fichier spécifié par l'argument `pathname` au descripteur `fd`, lequel est retourné en cas de succès.

# File I/O – Ouverture d'un fichier (II)

- **Arguments**
  - L'argument `pathname` spécifie le chemin et le nom du fichier à ouvrir. Le chemin peut être absolu ou relatif.
  - L'argument `flags` spécifie le mode d'accès au fichier, soit:
    - `O_RDONLY`: en mode lecture
    - `O_WRONLY`: en mode écriture
    - `O_RDWR`: en mode lecture et écriture
  - En mode écriture, l'utilisateur peut spécifier si les nouvelles données doivent être ajoutées au contenu existant ou si celles-ci doivent le remplacer
    - `O_APPEND`: ajouter les données au contenu existant
    - `O_TRUNC`: remplacer les données du fichier
  - Par exemple

```
int fd = open ("/home/lmi/myfile", O_WRONLY | O_APPEND);
```

# File I/O – Création d'un fichier ordinaire

- **La création d'un fichier ordinaire est obtenue à l'aide de l'appel système `creat()` ou la fonction `open()`**

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

- **Exemple**

```
int fd = creat ("/home/lmi/myfile", 0664);
```

**ou**

```
int fd = open ("/home/lmi/myfile", O_WRONLY | O_CREAT | O_TRUNC, 0664);
if (fd == -1)
     /* error*/
```

- **Comportement**
  - Ces deux appels sont équivalents. Ils permettent de créer un nouveau fichier. Si le fichier est déjà existant, le contenu de celui-ci sera simplement effacé.
  - Il faut noter, que si le fichier existe déjà, le mode du nouveau fichier ne sera pas adapté au mode spécifié lors de l'appel de la fonction.

# File I/O – Création d'un fichier ordinaire (II)

- **Arguments**
  - L'argument `pathname` spécifie le chemin et le nom du fichier à créer. Le chemin peut être absolu ou relatif.
  - L'argument `flags` spécifie le mode d'accès au fichier, en principe:
    - `O_WRONLY | O_CREAT | O_TRUNC`
  - L'argument `mode` spécifie les droits d'accès au fichier. Ce mode peut être passé sous forme octale ou sous forme symbolique en utilisant les constantes suivantes:

| | Owner | Group | Other |
|---|---|---|---|
| all | S_IRWXU | S_IRWXG | S_IRWXO |
| read | S_IRUSR | S_IRGRP | S_IROTH |
| write | S_IWUSR | S_IWGRP | S_IWOTH |
| execute | S_IXUSR | S_IXGRP | S_IXOTH |

Il est important de noter que les droits du fichier sera déterminé avec un ET logique entre le mode spécifié lors de la création du fichier et le complément à 1 du masque de création de fichiers attribué à l'utilisateur (valeur `umask`). Ce dernier peut être obtenu avec la commande `$ umask`

# File I/O – Lecture d'un fichier

- **La lecture d'un fichier est obtenue à l'aide de l'appel système `read()`**

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

- **Exemple**

```
char buf[100];
size_t len = sizeof(buf);
ssize_t nr = read (fd, buf, len);
if (nr == -1)
     /* error*/
```

- **Comportement**
  - La fonction `read()` permet de lire au maximum `len` octets du fichier et de les stocker dans `buf`. En cas d'erreur, la fonction retourne `-1`. La position du fichier est avancée du nombre d'octets lus, permettant ainsi de lire les octets restants.

# File I/O – Lecture d'un fichier (II)

- **Valeur de retour**
  - Si la valeur de retour est égale à `len` tout est en ordre
  - Si la valeur de retour est égale à `0`, cela indique la fin du fichier (`EOF`) et qu'il ne reste plus rien à lire
  - Si la valeur de retour est inférieure à `len`, mais supérieure à `0`, le nombre d'octets lus est stocké dans `buf`. Plusieurs causes peuvent être à l'origine de cette situation. Pour en connaître la raison, il suffit d'exécuter une lecture supplémentaire.
  - Si la valeur de retour est égale à `-1`, cela indique qu'une erreur est survenue. Dans ce cas, si `errno` est mis à `EINTR`, cela indique qu'un signal a été levé et qu'il faut réeffectuer une lecture, sinon une erreur sévère est survenue et il faut, par conséquent, stopper la lecture du fichier.

# File I/O – Lecture d'un fichier (III)

- **Lecture complète d'un fichier**

```
char buf[100];
while (1) {
     ssize_t nr = read (fd, buf, sizeof(buf));
     if (nr == 0)
              break; // --> all data have been read

     if (nr == -1) {
              if (errno == EINTR)
                       continue;  // --> continue reading

              char estr[100] = {[0]=0,};
              strerror_r (errno, estr, sizeof(estr)-1);
              fprintf (stderr, "ERROR: %s\n", estr);
              break; // --> error: stop reading
     }
     /* process read data */
}
```

# File I/O – Ecriture d'un fichier

- **L'écriture de données dans un fichier s'effectue à l'aide de l'appel système `write()`**

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

- **Exemple**

```
const char* buf = "data to be written";
size_t len = strlen(buf)
ssize_t count = write (fd, buf, len);
if (count == -1)
     /* error, check errno */
else if (count != len)
     /* severe error, but errno not set */
```

- **Comportement**
  - La fonction `write()` permet de stocker dans le fichier les `len` octets contenus dans `buf`. En cas d'erreur, la fonction retourne `-1`. La position du fichier est avancé du nombre d'octets écrits, permettant ainsi d'ajouter les octets supplémentaires.
  - Si la valeur retournée `count` est différente de `len`, cela signifie qu'une erreur sévère est survenue.

# File I/O – Synchronisation des données avec le disque

- **La synchronisation des données d'un fichier ordinaire contenues dans la mémoire cache s'effectue de façon asynchrone. Linux propose plusieurs services pour forcer cette synchronisation. Cette dernière peut s'effectuer avec l'appel système `fsync()`**

```
#include <unistd.h>
int fsync (int fd);
```

- **Exemple**

```
int ret = fsync(fd);
if (ret == -1)
     /* error, check errno */
```

- **Comportement**
  - La fonction `fsync()` demande au noyau de copier toutes les données restantes (contenue du fichier et métadonnées associées au fichier) sur le disque.

# File I/O – Positionnement dans un fichier

- **Usuellement, l'accès à un fichier s'effectue de manière linéaire. Cependant, certaines applications requièrent de pouvoir se déplacer librement dans le fichier pour accéder aux données souhaitées. Ce positionnement s'effectue avec l'appel système `lseek()`**

```
#include <unistd.h>
off_t lseek (int fd, off_t offset, int whence);
```

- **Exemple**

```
off_t ret = lseek (fd, 1150, SEEK_SET);
if (ret == -1)
     /* error, check errno */
```

- **Comportement**
  - La fonction `lseek()` déplace de `offset` octets la position du pointeur d'accès aux données dans le fichier. Ce déplacement s'effectue en relation avec l'origine `whence` (début, position courrante ou fin). La nouvelle position est retournée. En case d'erreur on obtient `-1`.

# File I/O – Positionnement dans un fichier (II)

- **Arguments**
  - L'argument `offset` spécifie le décalage du pointeur par rapport à une origine. Ce décalage peut être positif ou négatif.
  - L'argument `whence` spécifie l'origine du pointeur pour le calcul de la nouvelle position. Celui-ci peut prendre les valeurs suivantes:
    - `SEEK_SET`: le pointeur d'origine est placé au début du fichier
    - `SEEK_CUR`: le pointeur d'origine est placé à la position courante/actuelle
    - `SEEK_END`: le pointeur d'origine est placé à la fin du fichier
  - Exemples
    - pour obtenir la position courante

```
off_t pos = lseek(fd, 0, SEEK_CUR);
```

    - pour placer le pointeur au début du fichier

```
off_t pos = lseek(fd, 0, SEEK_SET);
```

    - pour placer le pointeur à la fin du fichier

```
off_t pos = lseek(fd, 0, SEEK_END);
```

# File I/O – Troncation d'un fichier

- **La troncation d'un fichier est obtenue à l'aide de l'appel système `ftruncate()`**

```
#include <sys/types.h>
#include <unistd.h>
int ftruncate (int fd, off_t length);
```

- **Exemple**

```
int ret = ftruncate (fd, 1150);
if (ret == -1)
     /* error, check errno */
```

- **Comportement**
  - La fonction `ftruncate()` tronque le fichier à `length` octets.
  - Si le fichier était plus long, les données supplémentaires sont perdues.
  - Si le fichier était plus court, il est étendu, et la portion supplémentaire est remplie d'octets nuls.

# File I/O – Lecture/écriture & positionnement combinés

- **Linux offre des services de lecture et d'écriture avec positionnement, en combinant les fonctions `read` et `write` avec `lseek`. Ces opérations s'effectuent avec les appels système `pread` et `pwrite`**

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
ssize_t pwrite (int fd, const void* buf, size_t count, off_t pos);
ssize_t pread  (int fd, void* buf, size_t count, off_t pos);
```

- **Exemple**

```
ssize_t count = pwrite (fd, buf, len, 0);
if (count == -1)
     /* error, check errno */
ssize_t nr = pread (fd, buf, len, 0);
if (nr == -1)
     /* error, check errno */
```

- **Comportement**
  - La 1ère fonction `pwrite()` permet de stocker dans au début du fichier (à la position 0) les `len` octets contenus dans `buf`.
  - La 2ème fonction `pread()` permet de lire `len` premiers octets du fichier (de la position 0) et de les stocker dans `buf`.

MASTER OF SCIENCE IN ENGINEERING | MSE

# File I/O – Lecture des métadonnées (statut) d'un fichier

- **Linux offre divers services pour obtenir les métadonnées d'un fichier. Ces opérations peuvent s'effectuer avec les appels système `stat`, `fstat` et `lstat`**

```
#include <unistd.h>

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

- **Exemple**

```
struct stat status;
int ret = fstat (fd, &status);
if (ret == -1)
     /* error, check errno */
```

- **Comportement**
  - L'appel de la fonction `fstat` retourne les informations liées au fichier dans la structure `stat`. En cas d'erreur, la fonction retourne -1.
  - La fonction `stat` retourne les mêmes informations que la fonction `fstat`, mais utile le nom du fichier et non pas le descripteur.
  - Si le fichier est un lien symbolique, la fonction `lstat` retourne les métadonnées du fichier symbolique et non pas celles du fichier cible.

# File I/O – Lecture des métadonnées (statut) d'un fichier (II)

- **Arguments**
  - La structure `struct stat` fournit les informations suivantes:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

- **Commande Linux**
  - Il est possible d'obtenir ces informations en utilisant la commande
    `$ stat <filename>`

# File I/O – Fermeture d'un fichier ordinaire

- **La fermeture d'un fichier ordinaire s'effectue à l'aide de l'appel système `close()`**

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

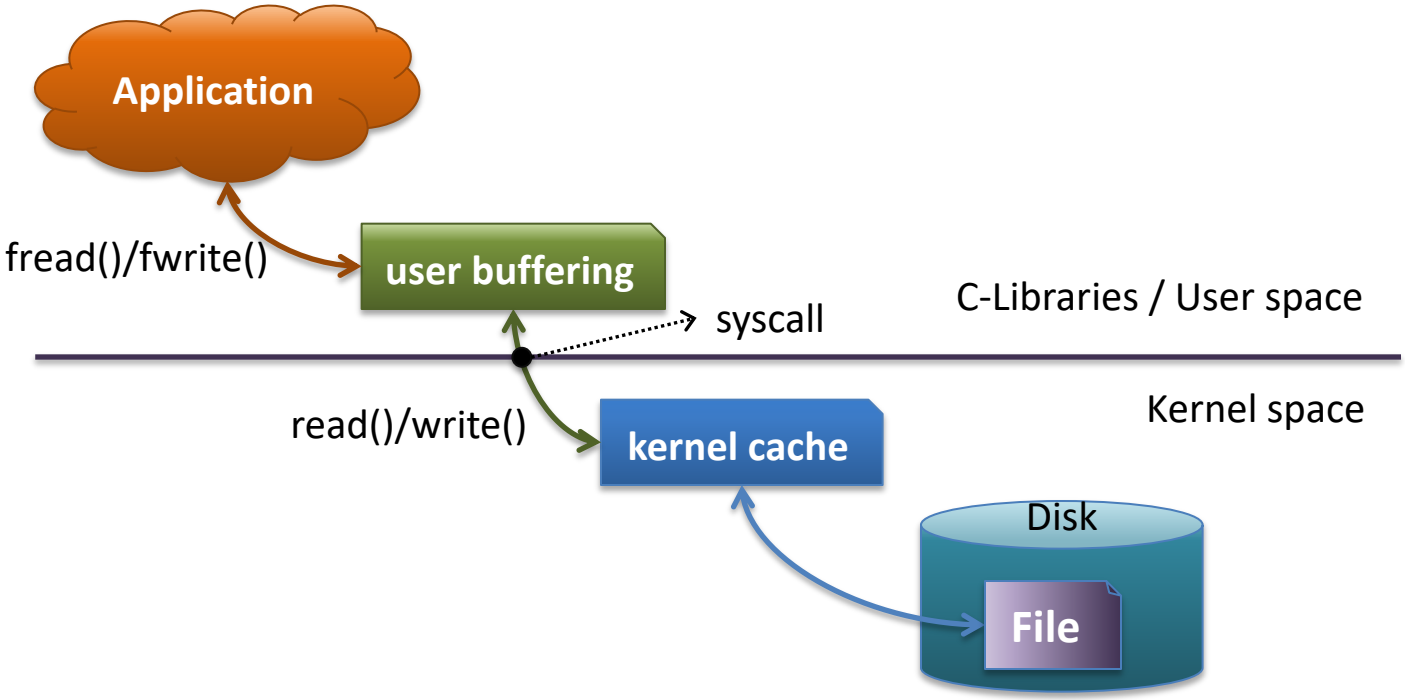
- **Exemple**

```
int ret = close (fd);
if (ret == -1)
     /* error, check errno */
```

- **Comportement**
  - La fonction `close()` permet de fermer le fichier et de libérer les ressources liées à son ouverture et de le dissocier du processus.
  - Il est important de noter que la fermeture du fichier ne garantit pas que toutes les données contenues dans la mémoire cache du noyau seront copiées sur le disque. Ce processus est totalement asynchrone sous Linux.

# Standard I/O – Fichiers ordinaires

- **Hormis l'accès direct aux fonctions du noyau Linux, C/C++ proposent d'autres services pour le traitement des fichiers. Ces services sont fournis par les bibliothèques standard (C: `stdio.h` / C++: `cstdio` ou `iostream`).**
- **Ces bibliothèques implémentent des caches supplémentaires au niveau des processus dans l'espace utilisateurs (user space), évitant ainsi de nombreux appels système (moins de changements de contexte entre espaces utilisateur (processus) et le noyau Linux).**



# Standard I/O – Fonctions principales

- **La bibliothèque standard C `<stdio.h>` propose toute une série de méthodes pour l'accès aux fichiers au format ascii ou binaire.**

```
/* functions to open, close, rename and remove files */
FILE* fopen (const char* path, const char* mode);
int fclose (FILE* stream);
int rename (const char* oldpath, const char* newpath);
int remove (const char* path);

/* functions to read/write binary files */
size_t fread  (void* buf, size_t size, size_t nr, FILE* stream);
size_t fwrite (void* buf, size_t size, size_t nr, FILE* stream);

/* functions to read/write text files */
int fgetc (FILE* stream);
int ungetc(int c, FILE* stream);
char* fgets (char* str, int size, FILE* stream);
int fputc (int c, FILE* stream);
int fputs (const char* str, FILE* stream);

/* auxiliary functions */
int fseek (FILE* stream, long offset, int whence);
int fflush (FILE* stream);
int ferror (FILE* stream);
int feof (FILE* stream);
```

MASTER OF SCIENCE IN ENGINEERING
MSE

# Standard I/O – Utilisation

- **L'utilisation des bibliothèques « Standard I/O » est recommandée lors de manipulation de fichiers avec de petites quantités de données à la fois (ascii ou binaire) ou accès aux données orienté caractère ou ligne.**
- **Elles offrent:**
  - Diminution des appels système
  - Amélioration des performances (accès aux entrées/sorties par blocs)

# Gestion des répertoires

MASTER OF SCIENCE IN ENGINEERING

MSE

# Gestion – Files and Directories Management

- **La gestion des fichiers et des répertoires est un aspect important pour un système d'exploitation. Linux propose divers services à cet effet.**
- **Horsmis les services pour créer et échanger des données avec des fichiers, Linux, via les bibliothèques `<unistd.h>` et `<dirent.h>`, nous offre des fonctions pour:**
  - Modifier les permissions

```
int chmod (const char* path, mode_t mode);
int chown (const char* path, uid_t owner, gid_t group);
```

  - Gérer les répertoires

```
char* getcwd (char* buf, size_t size);
int chdir (const char* path);
int mkdir (const char* path, mode_t mode);
int rmdir (const char* path);
```

  - **Lire le contenu de répertoires / traverser une arborescence**

```
DIR* opendir(const char* path);
struct dirent* readdir (DIR* dir);
int closedir (DIR* dir);
```

  - Gérer les liens (hardlinks et softLinks)

```
int link (const char* oldpath, const char* newpath);
int symlink (const char* oldpath, const char* newpath);
int unlink (const char* path);
```

MASTER OF SCIENCE IN ENGINEERING

MSE

# Gestion – Lecture du contenu de répertoires

- **La lecture du contenu de répertoires ou la traversée de l'arborescence d'un système de fichiers sont des opérations assez courantes. Elles sont utilisées pour obtenir l'annuaire des fichiers résidants dans un répertoire donné ou pour rechercher certaines informations, telles que la structure du système.**
- **La lecture s'effectue en trois opérations distinctes**
  - Ouverture du répertoire (méthode: opendir)
  - Lecture du répertoire (méthode: readdir)
  - Fermeture du répertoire (méthode: closedir)
- **Si l'on souhaite obtenir l'annuaire des répertoires d'une arborescence donnée, il suffit d'effectuer récursivement les trois opérations ci-dessus sur tous les fichiers représentant un répertoire.**

MASTER OF SCIENCE IN ENGINEERING

MSE

# Gestion – Ouverture de l'annuaire d'un répertoire

- **L'ouverture d'un répertoire est obtenue à l'aide de la méthode `opendir()`**

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char* dirname);
```

- **Exemple**

```
DIR* dirp = opendir ("/home/lmi/mydirectory");
if (dirp == 0)
     /* error*/
```

- **Comportement**
  - La fonction `opendir()` retourne un pointeur sur l'annuaire d'un répertoire (stream) spécifié par l'argument `dirname`. Si une erreur survient lors de l'ouverture, le pointeur sera `NULL` et la variable `errno` indiquera l'erreur, les plus probables:
    - `EACCES` → pas de permission
    - `ENOENT` → répertoire n'existe pas
    - `ENOTDIR` → `dirname` n'est pas un répertoire

MASTER OF SCIENCE IN ENGINEERING

MSE

# Gestion – Lecture de l'annuaire d'un répertoire

- **L'ouverture d'un répertoire est obtenue à l'aide de la méthode `readdir()`**

```
#include <dirent.h>

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

- **Exemple**

```
errno = 0;
while (true) {
     struct dirent* entry = readdir (dirp);
     if (entry == 0) break;
     if (strcmp(entry->d_name, filename) == 0)
               /* do something... */
}
if ((errno != 0) && (entry == 0))
     /* error*/
```

- **Comportement**
  - La fonction `readdir()` retourne un pointeur sur une entrée de l'annuaire du répertoire. Si toutes les entrées ont été découvertes, la fonction retourne un pointeur `NULL`. En cas d'erreur, le pointeur sera `NULL` et la variable `errno` indiquera l'erreur avec une valeur différente de `0`.
  - Il est important de noter que le contenu des données retournées par `readdir()` peut être modifié par d'autres appels à cette fonction pour le même annuaire. La méthode `readdir_r()` offre un service réentrant.

# Gestion – Lecture de l'annuaire (II)

## Données retournées

- La structure dirent contient les attributs suivants:

```
struct dirent {
   ino_t          d_ino;       // inode number
   off_t          d_off;       // offset to the next dirent
   unsigned short d_reclen;    // record length
   unsinged char  d_type;      // file type; not supported
                               // by all file system types
   char           d_name[256]; // filename
};
```

- L'attribut `d_type` peut prendre les valeurs suivantes:
  - `DT_BLK` → it's a block device
  - `DT_CHR` → it's a character device
  - `DT_DIR` → it's a directory
  - `DT_FIFO` → it's a named pipe (FIFO)
  - `DT_LNK` → it's a symbolic link
  - `DT_REG` → it's a regular file
  - `DT_SOCK` → it's a UNIX domain socket
  - `DT_UNKNOWN` → file type unknown

- Il est important de noter que seuls les attributs `d_ino` et `d_name` sont toujours disponibles (raisons de compatibilité). Pour obtenir les métadonnées du fichier, il suffira d'utiliser la fonction `stat()`.

# Gestion – Fermeture de l'annuaire d'un répertoire

- **La fermeture de l'annuaire d'un répertoire est obtenue à l'aide de l'appel système `closedir()`**

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR* dirp);
```

- **Exemple**

```
int err = closedir (dirp);
if (err == -1)
    /* error */
```

- **Comportement**
  - La fonction `closedir()` permet de fermer l'annuaire du répertoire et de libérer les ressources liées à son ouverture.
    - `EBADF` → pointeur sur l'annuaire de répertoire dirp invalide

MASTER OF SCIENCE IN ENGINEERING

MSE

# Surveillance de changements dans le système de fichiers

MASTER OF SCIENCE IN ENGINEERING

MSE

# inotify – Introduction

- **Pour certaines applications, il peut être très utile de surveiller les changements apportés à certains fichiers ou répertoires.**
- **Par exemple, un système de backup peut être intéressé à connaitres tous les fichiers qui ont été ajoutés, enlevés ou modifiés. Cette même surveillance peut s'avérer très pratique pour des processus daemon utilisant des fichiers de configuration. Elle lui permettrait d'adapter sa paramétrisation aux nouvelles modifications.**
- **La solution proposée par Linux s'appelle `inotify`.**
- **`inotify` permet de surveiller 12 événements distincts**

| Event | Description |
|---|---|
| IN_ACCESS | File was accessed (read()) |
| IN_ATTRIB | File metadata changed |
| IN_CLOSE_WRITE | File opened for writing was closed |
| IN_CLOSE_NOWRITE | File opened read-only was closed |
| IN_CREATE | File/directory created inside watched directory |
| IN_DELETE | File/directory deleted from inside watched directory |
| IN_DELETE_SELF | Watched file/directory was itself deleted |
| IN_MODIFY | File was modified |
| IN_MOVE_SELF | Watched file/directory was itself moved |
| IN_MOVED_FROM | File moved out of watched directory |
| IN_MOVED_TO | File moved into watched directory |
| IN_OPEN | File was opened |

# inotify – Opérations

- **Le mécanisme `inotify` de Linux propose divers services la surveillance de fichiers ou de répertoires. Il est intéressant de noter que ceux-ci peuvent être bloquants ou non bloquants. Dans le cas de services bloquants, les services de multiplexage (p.ex. `epoll`) peuvent être mis en œuvre pour attendre sur des événements.**

- **Opérations**
  - Créer une instance de surveillance (syscall: inotify_init1)
  - Ajouter un nouvel article à surveiller (syscall: inotify_add_watch)
  - Eliminer un article de la surveillance (syscall: inotify_rm_watch)
  - Lire les événements survenus (syscall: read)
  - Fermer une instance de surveillance (syscall: close)

# inotify – Créer une instance de surveillance

- **Pour créer une instance de surveillance, Linux propose l'appel système `inotify_init1()`.**

```
#include <sys/inotify.h>
int inotify_init1(int flags);
```

- **Exemple**

```
int ifd = inotify_init1(0);
if (ifd == -1)
   /* error */
```

- **Comportement**
  - La fonction `inotify_init1()` crée une nouvelle instance de surveillance. La méthode retourne un descripteur de fichier. En cas d'erreur, la valeur `-1` est retournée.
  - Si l'on souhaite un service non bloquant, il suffit de passer `IN_NONBLOCK` dans l'argument `flags`.

# inotify – Ajouter un article à surveiller

- **Pour ajouter un nouvel article dans l'instance de surveillance, Linux propose l'appel système `inotify_add_watch()`.**

```
#include <sys/inotify.h>
int inotify_add_watch (int fd, const char* pathname, uint32_t mask);
```

- **Exemple**

```
int wd = inotify_add_watch(ifd, "/path/to/file_dir", IN_ALL_EVENTS);
if (wd == -1)
   /* error */
```

- **Comportement**
  - La méthode `inotify_add_watch()` ajoute à l'instance de surveillance `fd` un nouvel article (fichier ou répertoire) spécifié par le 2e argument `pathname`. Les événements que l'on souhaite surveiller sont indiqués par le 3e argument `mask`.
  - La méthode retourne un descripteur correspondant à l'article. Ce dernier sera associé à chaque événement de ce même article. En cas d'erreur, la valeur `-1` est retournée.
  - Le processus doit naturellement disposer des droits pour surveiller un fichier ou un répertoire.

# inotify – Lire les événements

- **La lecture des événements se réalise simplement avec la méthode `read()`.**

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

- **Exemple**

```
char buff[sizeof(struct inotify_event)+NAME_MAX+1];
ssize_t len = read (ifd, buff, sizeof(buff));
if (len == -1)
   /* error */
char* p = buff;
while (len > 0) {
   struct inotify_event* event = (struct inotify_event*)p;
   /* process event... */
   len -= sizeof(struct inotify_event) + event->len;
   p += sizeof(struct inotify_event) + event->len;
}
```

- **Comportement**
  - La méthode `read()` retourne dans le `buff` une liste d'événements

# inotify – Lire les événements (II)

- **Arguments**
  - La `struct inotify_event` fournit les informations suivantes

```
struct inotfy_event {
   int      wd;     // watch descriptor
   uint32_t mask;   // maks of events
   uint32_t cookie; // unique cookie associating related events → rename(2)
   uint32_t len;    // size of name field
   char     name[]; // optional null-terminated name
};
```

  - `wd` identifie l'article surveillé pour lequel l'événement a été levé.
  - `mask` contient les bits décrivant l'événement
  - `cookie` est une valeur entière unique permettant de mettre en relation deux événements. Ceci arrive si l'on renomme un fichier ou un répertoire. Dans les autres cas, `cookie` vaut 0.
  - `name` identifie le nom relatif du fichier ou du répertoire pour lequel l'événement a été levé. `len` décrit le nombre de charactères contenus dans `name` incluant le caractère `0` terminant le string.

# Traitement des fichiers spéciaux

MASTER OF SCIENCE IN ENGINEERING

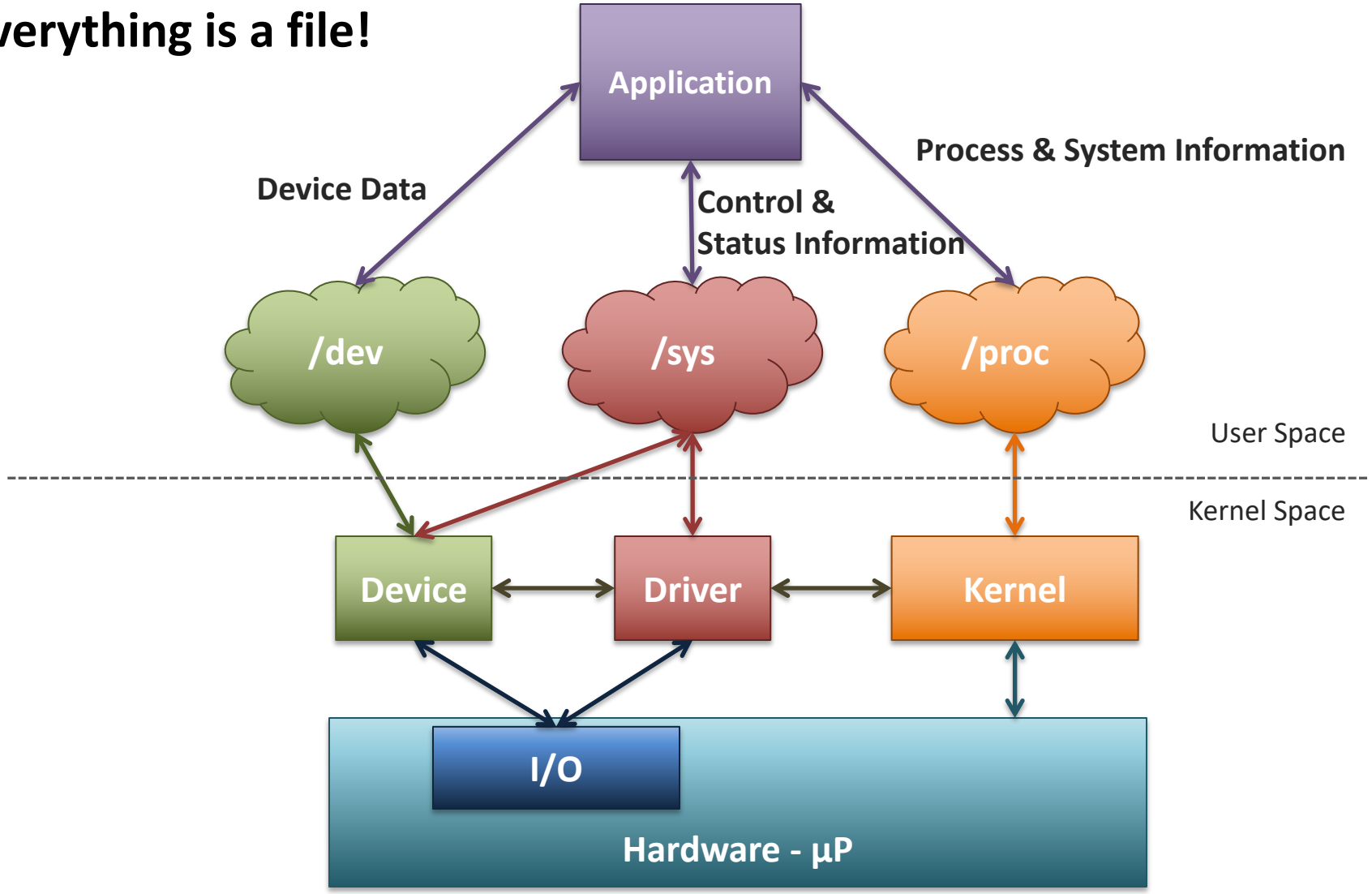
MSE

# Introduction aux fichiers spéciaux

- **Les fichiers jouent un grand rôle sous Linux (Everything is a file!)**
- **Les fichiers spéciaux offrent l'accès aux**
  - Données des périphériques d'entrées/sorties
    - Accès séquentiel aux données des périphériques (binaire ou ascii)
    - Fichiers d'accès placés sous `/dev (devfs)`
    - Configuration du pilote et du périphérique
  - Informations des périphériques, de leur pilote et des modules noyau
    - Accès séquentiel aux données (ascii)
    - Fichiers d'accès placés sous l'arborescence `/sys (sysfs)`
    - Configuration et monitoring des pilotes de périphériques et des modules noyau
  - Informations des processus et du noyau Linux
    - Accès séquentiel aux données (ascii)
    - Fichiers d'accès placés sous l'arborescence `/proc (procfs)`
    - Configuration et monitoring des processus et du noyau Linux
- **La majorité des opérations utilisées sur les fichiers ordinaires peuvent s'appliquer aux fichiers spéciaux**

# Fichies spéciaux – Architecture générale – Rappel

## Everything is a file!




MASTER OF SCIENCE IN ENGINEERING

MSE

# devfs – Opérations sur les fichiers `/dev`

- **devfs est un système de fichiers virtuels offrant l'accès aux données des périphériques. Cet accès est géré par des pilotes de périphériques.**
- **Ces pilotes offrent normalement qu'un accès séquentiel aux données, c.à.d., ces dernières ne peuvent être lues ou écrites que dans un flux continu sans saut aléatoire.**
- **Opérations principales supportées par File-I/O**
  - Ouverture du fichier (accès au périphérique) → open
  - Lecture continue du contenu du fichier (périphérique) → read
  - Ecriture continue dans le fichier (périphérique) → write
  - Lecture des métadonnées du fichier (périphérique) → stat
  - Fermeture du fichier (libération du périphérique) → close
- **Les méthodes citées ci-dessus sont à celles appliquées sur les fichiers ordinaux et peuvent être utilisées de façon identique.**
- **Il est cependant à remarquer que la méthode `open` n'autorise pas la création de fichiers d'accès aux périphériques. Cette opération est à exécuter avec d'autres services du noyau Linux lors de l'installation du pilote de périphérique.**

# devfs – Lecture/écriture non bloquante

- **Contrairement aux fichiers ordinaires, il est très courant qu'un périphérique ne puisse accéder immédiatement à la requête souhaitée (lecture ou écriture). L'exécution de l'application est alors suspendue pendant certain temps; p. ex. sur une interface série en mode lecture, cette situation peut arriver si aucune donnée n'a été reçue.**
- **Linux offre deux possibilités d'accès:**
  - Accès bloquant
  - Accès non bloquant
- **A l'ouverture du fichier d'accès (`open`), il est possible de choisir le type d'accès. Si l'on souhaite un accès non bloquant, il suffit d'ajouter avec un `OU` logique le fanion `O_NONBLOCK` au flags usuels.**
- **Si l'accès non bloquant a été choisi, les méthodes read et write retourneront le statut `-1` avec `errno` à `EAGAIN` si le périphérique n'est pas accessible. Il suffira alors simplement de resoumettre un peu plus tard la requête.**
- **En cas d'accès bloquant, le processus sera tout simplement suspendu jusqu'à ce que le périphérique soit disponible.**

# devfs – Lecture/écriture non bloquante – exemple

```
char buf[100];
ssize_t nr;
while (1) {
     nr = read (fd, buf, sizeof(buf));
     if (nr >= 0) break;
     if (errno == EINTR)
              continue;  // --> read again

     if (errno == EAGAIN)
              break; // --> resubmit later

     perror ("ERROR");
     break; // --> error: stop reading
}

if (nr > 0)
     // → process read data
```

## sysfs – Opérations sur les fichiers `/sys`

- **sysfs est un système de fichiers virtuels créé pour faciliter la configuration et le monitoring de pilotes de périphériques. Des outils, tels que `cat` ou `echo`, permettent d'accéder simplement aux informations stockées dans le sysfs.**
- **L'échange d'information avec les fichiers sous sysfs se fait sous forme ascii.**
- **Opérations principales supportées par File-I/O**
  - Ouverture du fichier (accès à l'attribut) → open
  - Lecture continue du contenu du fichier (attribut) → read / pread
  - Ecriture continue dans le fichier (attribut) → write / pwrite
  - Positionnement dans le fichier (attribut) → lseek
  - Lecture des métadonnées du fichier (attribut) → stat
  - Fermeture du fichier (libération de l'attribut) → close
- **Les méthodes citées ci-dessus sont à celles appliquées sur les fichiers ordinaires et peuvent être utilisées de façon identique. Il est cependant à remarquer que la méthode `open` n'autorise pas la création de nouveaux fichiers.**
- **La taille des données pouvant être échangées ne peut pas dépasser la taille maximale d'un page mémoire soit `PAGE_SIZE` bytes (généralement 4KiB).**

# procfs – Interface utilisateurs

- **procfs est un système de fichiers virtuels créé pour faciliter l'accès aux données et information du noyau Linux ainsi que sur l'état des processus. La majorité des fichiers se trouvant dans son arborescence sont au format texte et peuvent ainsi être accédés à l'aide d'outils tels que `echo` et `cat`.**
- **Pour chaque processus Linux, procfs crée un répertoire avec le numéro du processus correspondant au PID (Process ID). Les différentes informations relatives à l'état du processus sont contenues dans des fichiers séparés, p.ex. :**

```
/proc/<PID>/
    |-- cwd       → path/symlink to current working directory
    |-- exe       → path/symlink to original executable file
    |-- root      → path/symlink to the root path as seen by the process
    |-- cmdline   → command that launched the process
    |-- status    → basic information about the process
    |-- task      → directory with all threads in the process
    |-- fd        → directory with all open file descriptors
    |-- maps      → information about mapped files and blocks (heap, stack…)
    |-- limits    → information process' resource limits
    |-- ...
```

- **`/proc/self` permet d'accéder aux informations de son propre processus**

# procfs – Interface utilisateurs (II)

- **Les informations liées au noyau Linux seulement sont fournies par procfs directement à la racine du système de fichiers virtuels, p.ex.**

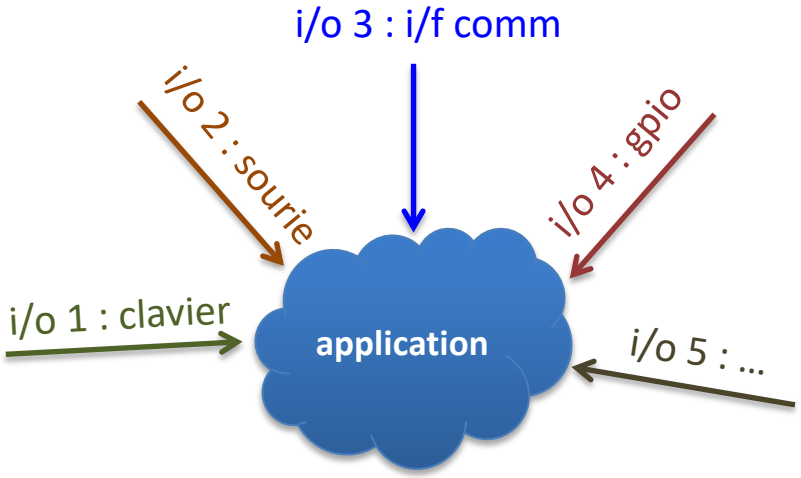
```
/proc/
   |-- cmdline      → command that launched the Linux kernel (boot options)
   |-- cpuinfo      → information about CPU and vendor
   |-- devices      → list of device drivers (character or block)
   |-- modules      → list of loaded modules
   |-- meminfo      → information about memory usage
   |-- version      → information about Linux kernel version
   |-- interrupts   → information about number of interrupts per CPU and I/O
   |-- iomem        → information about I/O memory map
   |-- net          → directory with information about network
```

- **Les méthodes que celles appliquées sur les fichiers ordinaires peuvent être utilisées sur ces fichiers. Cependant, pour faciliter l'accès à ces informations, il existe naturellement toute une série d'utilitaires, p.e.x. :** ps, uname, dmesg, …
- **Plus de détails sous [http://man7.org/linux/man-pages/man5/proc.5.html](http://man7.org/linux/man-pages/man5/proc.5.html)**

# Multiplexage des entrées/sorties

# Introduction

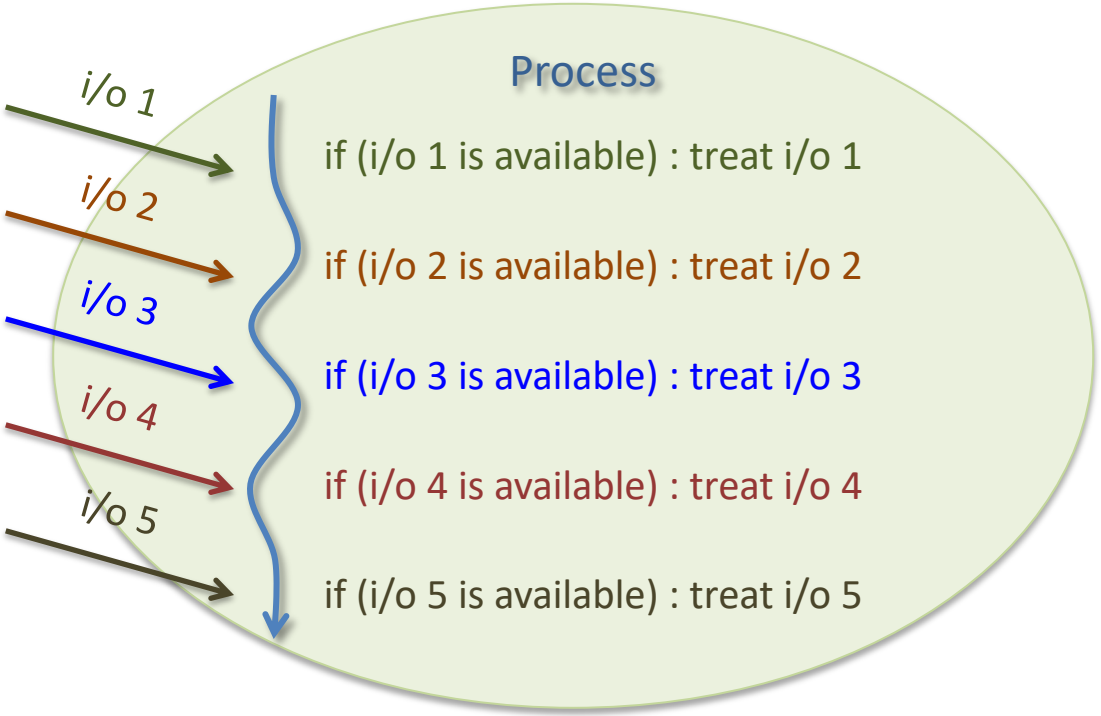
- **Il est usuel que des applications doivent attendre sur des périphériques d'entrées/sorties, tels que clavier, souris, interface de communication, etc.**
- **Ces périphériques sont représentés dans l'application par un descripteur de fichier, lequel offre généralement un accès au périphérique par un des appels système `read` ou `write`.**




- **Mécanismes à disposition**
  - Utilisation des services non bloquants
  - Implémentation d'un thread par entrée/sortie
  - Utilisation de services offrant un multiplexage des entrées/sorties

# Mécanismes – Services non bloquants

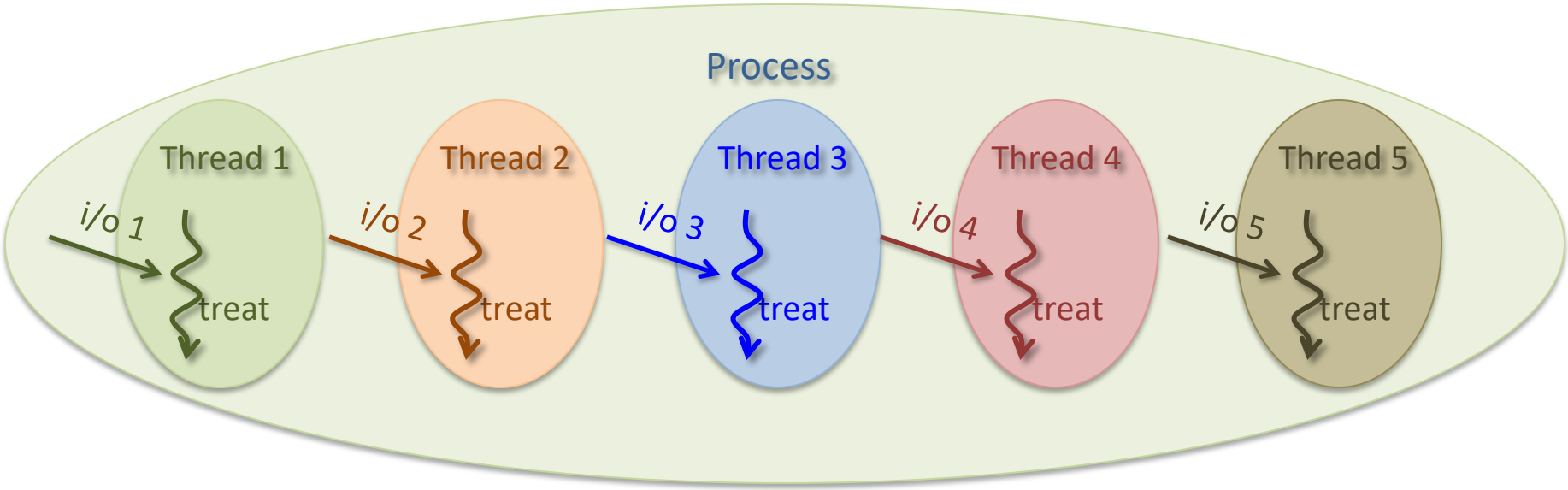
- **Un chemin possible pour solutionner cette problématique est l'utilisation de services non bloquants pour scruter chaque entrée/sortie séquentiellement.**
- **Ces services permettent de tester le périphérique si son accès est possible ou pas. Si l'accès est disponible, alors l'entrée/sortie est traitée.**




- **Carences**
  - Usage excessif du processeur pour la scrutation
  - Complexité du logiciel

# Mécanismes – Multi-threading

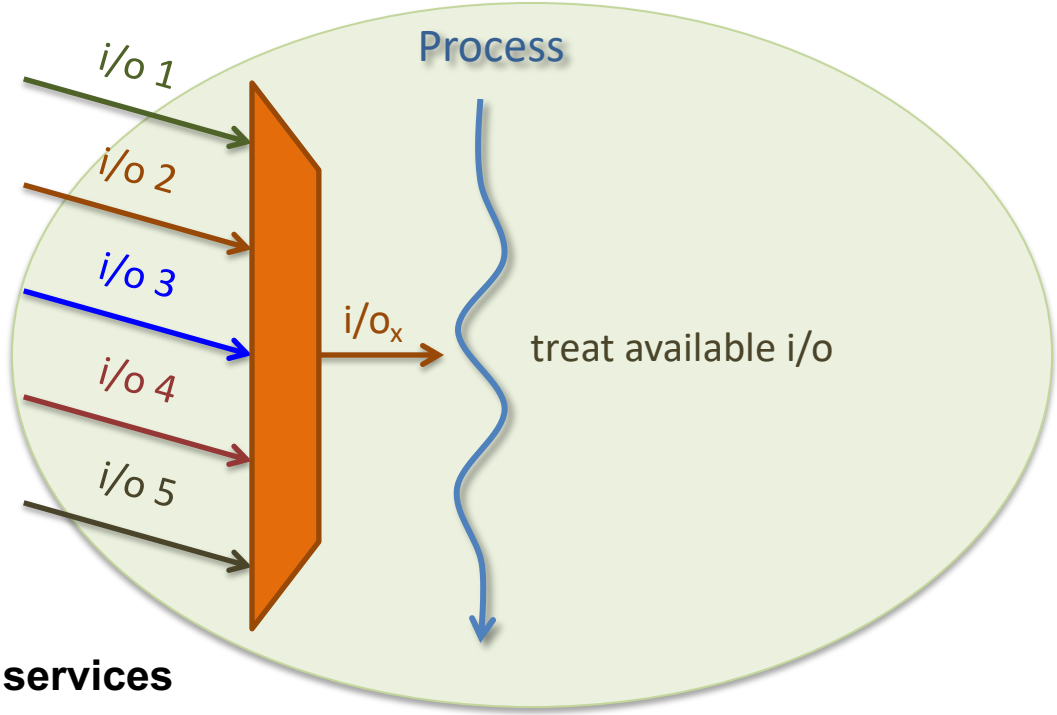
- **Une deuxième voie possible consiste à utiliser les services bloquants et à créer un thread par entrée/sortie**
- **Cette technique permet d'éviter que l'indisponibilité d'une entrée/sortie bloque le traitement des autres**




- **Carences**
  - Nombre de threads nécessaires pour réaliser l'application
  - Complexité du logiciel pour synchroniser les différentes données
  - Complexité lors du debugging du logiciel

# Mécanismes – Multiplexage des entrées/sorties

- **Les systèmes Unix, et plus particulièrement Linux, proposent des services autorisant le multiplexage des entrées/sorties. Ces services permettent de sélectionner un catalogue d'entrées/sorties à traiter et de le passer au noyau. Ce dernier informera le processus sur ceux disponibles pour traitement.**




- **Linux propose 3 services**
  - select() (compatible avec les systèmes Unix)
  - poll() (pas traité ici)
  - epoll() (Linux spécifique)

# select – Service

- **Linux propose avec l'appel système `select()` une interface simple pour le multiplexage d'entrées/sorties.**

```
#include <sys/select.h>
#include <sys/time.h>

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set)
```

- **Comportement**
  - La fonction `select()` permet d'attendre jusqu'à ce qu'un des descripteurs de fichiers soit prêt pour effectuer l'opération de lecture ou d'écriture souhaitée. En cas de succès, la fonction retourne le nombre de descripteurs de fichiers disponibles.
  - Linux man page: http://linux.die.net/man/2/select

# select – Service (II)

- **Arguments**
  - L'argument `n` spécifie le numéro du plus grand descripteur de fichiers plus un.
  - Les arguments `readfds`, `writefds` et `exceptfds` contiennent la liste des descripteurs de fichiers en lecture, écriture respectivement exception sur lesquels la méthode `select()` doit attendre.
    En retour, ces arguments indiquent les descripteurs disponibles pour le service demandé.
  - L'argument `timeout` permet d'attendre sur un événement avec un temps limite. Si `timeout` est non `NULL`, la `struct timeval` permet de spécifier le temps en microsecondes, p.ex. pour attendre 10 seconds au maximum:

```
struct timeval timeout;
timeout.tv_sec = 10;
timeout.tv_usec = 0;
```

- **Macros**
  - `FD_ZERO` initialise la liste des descripteurs de fichiers
  - `FD_CLR` enlève un descripteur de fichiers de la liste
  - `FD_SET` ajoute un descripteur de fichiers dans la liste
  - `FD_ISSET` teste si un descripteur de fichiers est contenu dans liste

# select – Exemple

```
fd_set fd_in, fd_out;
FD_ZERO(&fd_in);
FD_ZERO(&fd_out);

// monitor fd1 for input events and fd2 for output events
FD_SET(fd1, &fd_in);
FD_SET(fd2, &fd_out);

// find out which fd has the largest numeric value
int largest_fd = (fd1 > fd2) ? fd1 : fd2;

// wait up to 5 seconds
struct timeval tv = {.tv_sec = 5, .tv_usec = 0, };

// wait for events
int ret = select (largest_fd+1, &fd_in, &fd_out, NULL, &tv);

// check if select actually succeed
if (ret == -1) {
    // report error and abort
} else if (ret == 0) {
    // timeout; no event detected
} else {
    if (FD_ISSET(fd1, &fd_in))  { // input event on sock1  }
    if (FD_ISSET(fd2, &fd_out)) { // output event on sock2 }
}
```

# epoll – Service

- **Le service `epoll` a été introduit sur Linux pour pallier aux carences du service `select()` et pour offrir une interface performante lorsqu'un grand nombre de descripteurs de fichiers doit être traité.**
- **Opérations**
  - Creation d'un contexte epoll (syscall: epoll_create1)
  - Contrôle du contexte epoll (syscall: epoll_ctl)
  - Attente sur des événements (syscall: epoll_wait)

MASTER OF SCIENCE IN ENGINEERING

MSE

# epoll – Création d'un contexte epoll

- **La création d'un contexte epoll est obtenue à l'aide de l'appel système `epoll_creat1()`**

```
#include <sys/epoll.h>
int epoll_create1(int flags);
```

- **Exemple**

```
int epfd = epoll_create1(0);
if (epfd == -1)
    /* error*/
```

- **Comportement**
  - La fonction `epoll_create1()` permet de créer une nouveau contexte epool pour le multiplexage des entrées/sorties. Les erreurs pouvant survenir lors de la création d'un contexte epoll sont `EINVAL`, `EMFILE`, `ENFILE` ou `ENOMEM`.

# epoll – Contrôle d'un contexte epoll

- **Le contrôle d'un contexte epoll est obtenu à l'aide de l'appel système `epoll_ctl()`**

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

- **Exemple**

```
struct epoll_event event = {
     .events = EPOLLIN | EPOLLOUT,
     .data.fd = fd,
};

int ret = epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event);
if (ret == -1)
     /* error*/
```

- **Comportement**
  - La fonction `epoll_ctl()` permet d'ajouter ou de retirer un descripteur de fichiers au contexte contexte epool. Il est également possible de modifier les événements que l'on souhaite surveiller pour un descripteur de fichiers donné.

# epoll – Contrôle d'un contexte epoll (II)

- **Arguments**
  - L'argument `epfd` spécifie le contexte epoll à contrôler
  - L'argument `op` indique l'operation que l'on souhaite exécuter avec le contexte epoll pour le descripteur de fichiers `fd`.
    - `EPOLL_CTL_ADD` ajouter un nouveau descripteur de fichiers
    - `EPOLL_CTL_DEL` retirer un descripteur de fichiers
    - `EPOLL_CTL_MOD` modifier les événements à surveiller
  - L'argument `event`, à l'aide de la `struct epoll_event`, permet de spécifier les événements (attribut `events`) que l'on souhaite surveiller pour le descripteur de fichiers `fd`. Cette même structure permet également d'attacher un paramètre supplémentaire (attribut `data`) pour identifier le descripteur de fichiers ayant levé un événement.

```
struct epoll_event {
        __u32 events;
        union {
                void *ptr;
                int fd;
                __u32 u32;
                __u64 u64;
        } data;
};
```

# epoll – Contrôle d'un contexte epoll (III)

- **Arguments** (suite…)
  - Les événements principaux que l'on peut surveiller sont
    - `EPOLLERR` une condition d'erreur a été levée
    - `EPOLLIN` un fichier virtuel est disponible lecture
    - `EPOLLOUT` un ifchier virtuel est disponible en écriture
    - `EPOLLPRI` des données prioritaires « out-of-band » sont diponible

MASTER OF SCIENCE IN ENGINEERING

MSE

# epoll – Attente sur la levée d'événements

- **L'attente sur des événements pour un contexte epoll est réalisée à l'aide de l'appel système** `epoll_wait()`

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

- **Exemple**

```
struct epoll_event events[2];
int nr = epoll_wait(epfd, events, 2, -1);
if (nr == -1)
     /* error*/
for (int i=0; i<nr; i++) {
   printf ("event=%ld on fd=%d\n", events[i].events, events[i].data.fd);
   // operation on events[i].data.fd can be performed without blocking...
}
```

- **Comportement**
  - La fonction `epoll_wait()` permet d'attendre jusqu'à ce que un ou plusieurs événements soient levés pour les descripteurs de fichiers attachés au contexte epoll.

# epoll – Attente sur la levée d'événements (II)

- **Arguments**
  - L'argument `epfd` spécifie le contexte epoll à surveiller
  - L'argument `events` spécifie l'adresse d'un tableau pour stocker la liste d'événements ayant été levés et pouvant être traités. Si le nombre d'événements levés dépasse la taille du tableau, des appels successifs au service `epoll_wait` fourniront les événements restants.
  - L'argument `maxevents` indique la taille maximale du tableau
  - L'argument `timeout` spécifie l'intervalle de temps maximal en millisecondes à attendre sur un événement. Si la valeur de `timeout` est `-1`, la fonction ne retournera que si un événement a été levé.