Programmation Système (UNIX) POSIX

Philippe Ezequel

Modalité de contrôle de connaissance : Coding Party (mise en ligne d'un sujet à 8h, rendu le lendemain avant minuit)

Bibliographie :

- *Programmation Système en UNIX* (2003), J.M Rifflet
- */proc et /sys*, (Linux)
- manuel en ligne : `man`

Le manuel est divisé en **sections** :

1) Les commandes
2) Les appels système UNIX ➔ Ce qu'on va étudier
3) Bibliothèques (`glibc`)

`man kill` ➔ Commande `kill`
Mais `kill` est aussi un appel système. On fait :
`man 2 kill` ➔ Appel système `kill`

Plan du cours :

1) Processus : naissance, vie et mort
2) Communications entre processus :
   - IPC System V
   - sockets
3) Système de Gestion de Fichier (SGF)

Histoire :

UNIX a été développé dans les laboratoires de Berkeley et AT&T.
MULTICS → UNIXS → INUX (01/01/1970)
Nouveau langage qui a été développé : le langage C.
Dans les années 70, schisme :

- System V (AT&T)
- BSD (Berkeley)

Dans les années 80 : beaucoup d'UNIX (chaque fabricant développait sa propre version)
Fin 80 : **POSIX** : norme qui définit ce que doit être UNIX. La totalité des systèmes actuels sont POSIX.
Années 90 : LINUX

Programmation Système :

- en C
- on utilise les appels système : fonctionnalités du noyau.

Exemple d'appels système :

```
pid_t getpid();
```

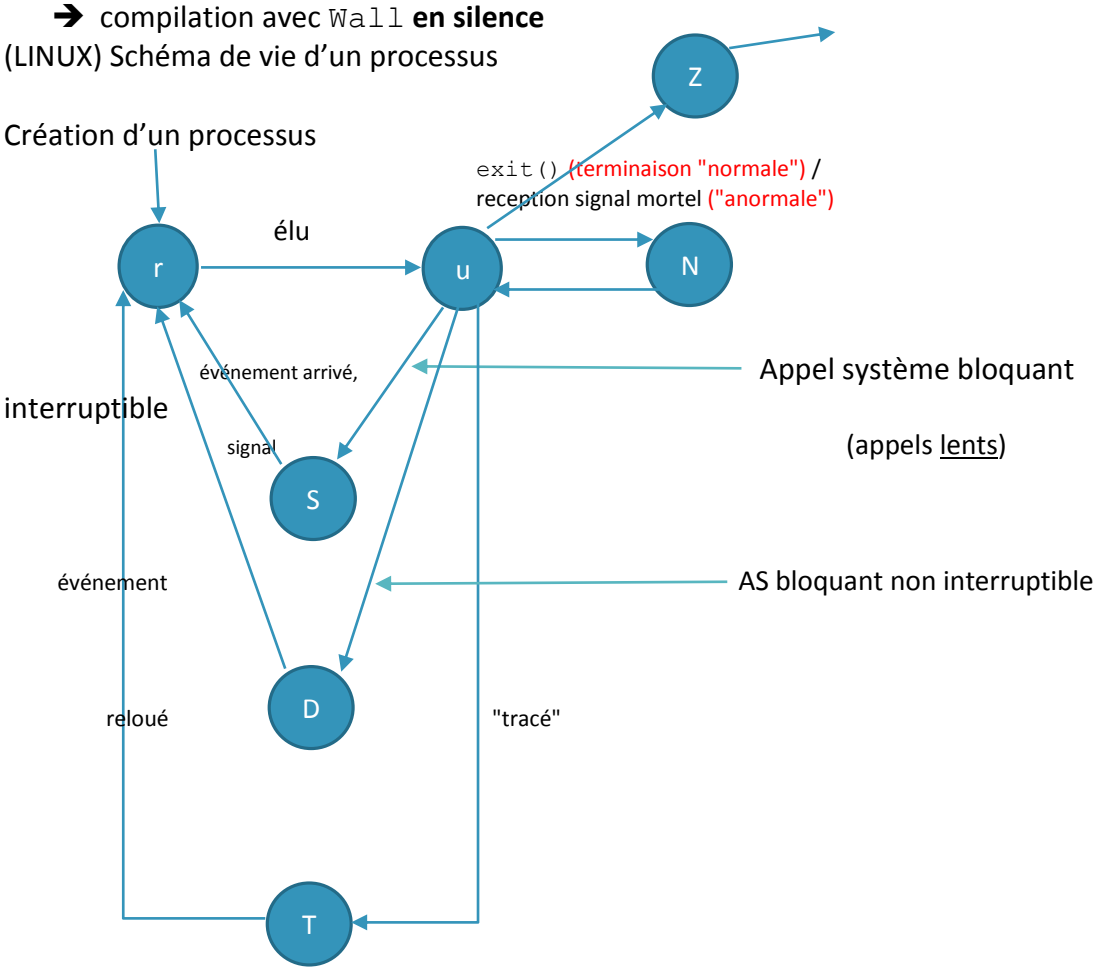
```
pid_t waitpid(pid_t proc, int *statut, int options);
int un_appel(…);
```
↳ renvoit `0` (pas de problème) ou `-1` (problème !)
Mécanisme "errno" `<errno.h>` variable globale `errno(int)`

GCC : GNU C Compiler (Coq –compliant)

- Option `Wall`

➔ compilation avec `Wall` **en silence**

(LINUX) Schéma de vie d'un processus



Processus : programme en cour d'éxectution. Il possède un **espace d'adressage** (EA) : partie de la mémoire virtuelle allouée au processus (LINUX 2 Go)

Tout processus peut créer d'autres processus (relation père/fils).
Donc tous les processus ont un père, (sauf INIT).
Pour visualiser l'arborescende des processus : `pstree`

ATTENTION : Dans le cas d'applications mutltiprocessus, on ne peut faire **aucune** hypothèse a priori sur le résultat de l'ordonnancement des processus impliqués. On peut imposer un ordonnancement (synchronisation, communication), cf partie 2

→informations système : Process Control Block (PCB)
(linux : pseudo SGF / proc)

Contenu du PCB :

1) Identité du processus : entier, le PID.

```
<unistd.h>
pid_t getpid() ;
ppid_t getppid() ;
```
} ces deux appels réussissent TOUJOURS (norme POSIX)

2) Propriétaire : Le propriétaire d'un processus c'est celui du père.

Un processus a 2 propriétaires :

- Le propriétaire réel : "vrai" propriétaire.
- Le propriétaire effectif : le déguisement du processus.
  ex : "`ls є root`" mais root fait en sorte qu'on puisse l'utiliser

L'appel système qui donne le propriétaire réel : `uid_t getuid() ;`
L'appel système qui donne le propriétaire effectif : `euid_t geteuid() ;`
} réussissent toujours

Modification des propriétaire :

- Propriétaire réel :
  ```
  int setuid(uid_p prop);
  ```
  ⇨ prop devient le propriétaire réel et effectif du processus
  Réservé à root
  Si on est pas root, on change que l'euid :
  ```
  int seteuid(uid_p prop);
  ```
- Groupe du propriétaire :
  ```
  gid_t getgid();
  gid_t getegid();
  int setgid(gid_t gid);
  int setegid(gid_t gid);
  ```

Répertoire de travail :

Modification :

```
int chdir(char * rep);
```

Consultation :

```
char* getcwd(char * buf,size_t taille)
```

- Ecrit à l'adresse `buf` le répertoire courant sur `taille` caractères maximums et renvoie cette adresse
- Si la longueur du repertoire courant est supérieur à `taille`, renvoie `NULL` et positionne `errno` à `ERANGE`
- (LINUX only) Si `buf == NULL && taille == 0` Alors get cwd se charge du malloc

Date de création :

En seconde depuis 01/01/1970 0h0m0s

Remarque

En `int`, la valeur max est 217843647 (Le mardi 19/01/2038 à 3h14m07s GMT)

Temps CPU consommé :

```
clock_t time(struct tms *buf);
```

`clock_t` : Top d'horloge (Linux : 100/sec) Indeterminé
`struct tms` : 4 champs

- `clock_t tms_utime` ; Temps passé en mode Utilisateur
- `clock_t tms_stime` ; Temps passé en mode Appel Système
- `clock_t tms_cutime` ; Temps total passé en mode Utilisateur
- `clock_t tms_cstime` ; Temps total passé en mode Appel Système

`gettimeofday( µs);` Heure en micro-secondes
`clock_gettime( ns);` Heure en nano-secondes

Signalisation

Signal : Interruption logicielle (NSIG disponibles)

Comportement :

- Par défaut :
  - Ignorer
  - Mourir (SIGTERM par ^C (Ctrl + C))
  - Stopper (SIGSTOP par ^Z)

Configuration :

- Masquage d'un ou plus de signaux
- Capture d'un ou + de signaux (Positionnement d'une "routine" de traitement) } Sauf pour SIGKILL

Masque de création de fichier

Entrer sur 9bits, XORé avec le mode de création
Exemple : 27 car *rwxrwxrwx*
1 111

- `setuid bit` | Pour les éxécutables, permet aux processus exécutant le fichier de prendre le propriétaire du fichier comme propriétaire effectif le temps de l'éxécution du fichier (et rwxrwxrwx est remplacé par rw**s**rwxrwx)

- `setgid bit` | Idem pour groupe du propriétaire
- `stickybit` | Pour répertoires, seuls les propriétaire des fichiers du répertoire peuvent modifier leurs fichiers

Table des descripteurs de fichiers ouverts

Table qui pointe vers les fichiers ouverts (généralement 1er = stdin, 2ème = stdout)
Taille maximal de `OPEN_MAX` fichiers ouverts (Linux : `OPEN_MAX=1024`)

Etat du processus

R, U, N, S, D, T, Z

Z (Zombie) : Attend que son père prenne en compte sa terminaison.

Priorité

Valeur de gentillesse/courtoisie (**nice**)
SystemV: valeur de 0 à 39, valeur 20 par défaut
BSD: valeur de -20 à 20, valeur 0 par défaut

Le processur est libre d'**augmenter** son nice (baisser en priorité) mais seul **root** peut le **réduire**

Partie I : Naissance, vie et mort d'un processus

Création

```
pid_t fork();
```

Quand un processus fait un `fork` son fils est identique à lui, sauf :

- PID
- PPID
- Temps CPU
- Signaux pendants
- nice

Plusieurs retour de `fork` :

- Si problème, renvoit -1
- Sinon :
  - 0 : fils
  - >0 : père, PID du fils

Méthodologie

Forme d'un programme utilisant fork :

```
pid_t p;
if((p=fork()) == -1)
     /*traitement problème*/
else if (p==0)
          /*code du fils :*/
          execve(…);
          exit(-1);
     else /*code du père*/
```

Remarque

Recopie paresseuse du père dans le fils (copy-on-write)

ATTENTION : Lapins
Exemple :

```
int main()
{
     while(1)
          fork();
```

```
}
```

Recouvrements

C'est le changement de programme en cours d'exécution :

```
int execve(char* prog, char* argv[], char* envp[]);
```

- `argv` et `envp` sont terminées par `NULL`
- Seul valeur de retour possible : `-1`

Effet :

- Remise à zéro de l'**espace d'adressage** (EA) donc tas, pile et code
- Remise à zéro de la **signalisation** (en particulier capture)
- Si fichier set vid/gid, on modifie evid/egid

Taille maximale des arguments : (hauteur de pile maximale pour empiler les arguments de `execve`)

- En POSIX : `ARG_MAX = 128 ko`
- En Linux : `ARG_MAX = 2 Mo` (par défaut)

(puisque la taille max de la pile = `8 Mo`, et il faut pouvoir executer le `main`, au moins.)
! Si on veut un programme portable, il faut s'assurer que la taille maximale des arguments ne dépasse pas `128 ko`.

Exemple :

```
char * cmdline[]={"/bin/emacs", "projet.c", NULL} ;
execve("/usr/games/minesweper", cmdline, envp) ;
exit(-1) ;
```

Frontaux de `execve` :

```
int execv (char *prog, char *argv[]) ;

int execvp (char *prog, char *argv[]) ; /*Le programme va être
chercher dans la variable d'environnement PATH*/

int execle (char *prog, char *arg1, …, char *argN, NULL, char
*envp[]) ; /*plus facile quand il y a peu d'arguments*/

int execl (char *prog, char *arg1, …, char *argN, NULL) ;

int execlp (char *prog, char *arg1, …, char *argN, NULL) ;
```

Attente de terminaison des fils

2 terminaisons possibles :

- "normale" : appel à `exit`
- "anormale" : réception d'un signal mortel

En principe, un processus qui crée des fils, atteint la terminaison des fils pour se terminer.
Si un père se termine avant ses fils, les processus orphelins sont "adoptés"par `INIT`.

Appel système d'attente :

```
<sys/types.h>
<sys/wait.h>

pid_t waitpid (pid_t fils, int *statut, int options)

fils :
     1) si fils > 0 : on attend la fin de ce fils là
     2) si fils =-1 : on attend la fin de n'importe quel fils
```

Sémantique de l'appel :

- Si pas ou plus de fils, retour immédiat de `-1`, `errno = ECHILD`
- Sinon, appel bloquant jusqu'à terminaison d'un fils (par défaut)
- renvoie le `pid` de **l'un** des fils terminés, `*statut` contient la cause de la terminaison du fils
- Si interrompu par réception d'un signal non mortel, `waitpid` retourne `-1` et `errno=EINTR`

Autre forme :
`pid_t wait (int *statut)`
Avec `wait (&cause_fin)` ⇔ `waitpid(-1, &cause_fin,0)`

Exemple :

```
while (wait(NULL)>0)
     ;
```

Permet d'attendre la terminaison de tous les fils

Exploitation de `*statut`

1) Si le fils est terminé "normalement", `WIFEXITED(*statut)≠0`
   et alors `WEXITSTATUS(*statut)` renvoie l'argument de l'invoction d'`exit` d'un fils.
2) Si le fils est terminé par réception d'un signal mortel,`WIFSIGNALED(*statut)≠0`
   et alors `WTERMSIG(*statut)` renvoie le signal mortel

Options :

Combinaison bit à bit des indicateurs suivants :

- `WNOHANG` : rend l'appel de `waitpid` non bloquant.
- `WUNTRACED` : `waitpid` renvoie en plus le `pid` des fils arrêtés par `SIGTSTP`

  (signal qu'un processus reçoit quand on tape Ctrl + Z) ou `SIGSTOP` (et les macros `WIFSTOPPED(*statut)` et `WSTOPSIG(*statut)`
- `WCONTINUED` : si l'un des fils stoppés (par la reception de `SIGTSTP` ou `SIGTSTOP`) a été relancé par par le signal `SIGCONT` (d'où la macro `WIFCONTINUED(*statut)`)

Si on ne veut pas d'options, on met 0. Si on veut une seule options, on l'écrit. Pour tracer les fils arrêté et continuées : `WUNTRACED | WCONTINED`. On combine donc les options avec le symbole `|`.

Partie II : Communication inter-processus

1) Signalisation

- Signal = interruption logicielle
- 4 réaction par défaut à la reception d'un signal :
    - Ignorer le signal
    - Stopper
    - Redémarrer
    - Mourir (avec ou sans fichier `CORE`) (un fichier CORE contient l'espace d'adressage "utile" du processus au moment de la reception du signal
- Signaux envoyés par
    - d'autres personnes
    - noyau

Quand un signal est-il reçu ? Quand il arrive dans la partie U (unité centrale)
Signal mortel reçu pedant un appel système bloquant interrruptible : retour AS -1, `errno=EINTR`
Attente d'un signal :

```
int pause() ; /*Endort le processus jusqu'à reception d'un
signal non mortel ; retour=-1, errno=EINTR*/
```

Primitives d'envoi de signaux :

```
int kill(pid_t pus, int signal)
```

```
0 ou -1

pus :
-  pus > 0 : pid du pus destinataire
-  pus = -1 : signal envoyé à tous les processus de même UID

signal :
-  signal > 0 : ce signal
-  signal = 0 : vérification de l'existence du (ou des)
   processus

kill -KILL -1  ⇔ kill(-1, SIGKILL)
```

Causes d'erreurs de `kill` :

- Le ou les processus spécifiés n'existent pas → `errno = ESRCH`
- Processus pas de même UID → `errno = EPERM`
- Signal inéxistant → `errno = EINVAL`

```
int raise(int signal) ;

raise(s) ⇔ kill(getpid(), s)
/*Permet au processus de s'envoyer lui-même un signal*/
```

A quoi ça sert ?
Le processus peut mettre un « point de sauvegarde », puis continuer. S'il rencontre une erreur dans la suite, le `raise(s)` permet de revenir en arrière comme si rien ne s'était produit

Modification du comportement par défaut

2 façons :
- masquage
- capture

1) Masquage de signaux

Type de données "ensmemble de signaux" : `sigset_t`

```
int sigemptyset (sigset_t *e); //e←Ø
int sigtillset (segset_t *e); //e←{1,…,NISG}
int sigaddset (sigset_t *e, int s); //e←eU{s}
int sigdelset (sigset_t *e, ints); //e←e-{s}S
```

Ces fonctions renvoient 0 ou -1

```
int sigisnumber(sigset_t *e, int s); //s∈e?
```

Masquage/Démasquage

`int sigprocmask(int op, sigset_t *new, sigset_t *old)` (0 ou -1)

Valeur possibles du paramètre `op` :

- `op=SIG_SETMASK` : `new` devient le nouveau masque et `old` (si non NULL) reçoit l'ancien masque
- `op=SIG_BLOCK` : ajoute `new` au masque actuel
- `op=SIG_UNBLOCK` : enlève `new` au masque actuel
- si `new=NULL`, `old` ne doit pas l'être, le masquage n'est pas modifié.

Masquage de `SIGKILL` et `SIGSTOP` est ignoré en silence.
Signal pendant : signal reçu mais masqué.

Consultation des signaux pendants :

```
int sigpending(sigset_t *e)
```

Remarque : `fork et execve` conserve le masquage .

Capture d'un signal

Spécifier une fonction qui sera appelée à la reception du signal.

Remarque : `fork` conserve la capture mais `execve` ne conserve pas la capture.

ATTENTION ! Dans les versions AT&T, le comportement par défaut associé au signal est remis en place après l'exécution de la routine. Il faut "réarmer" la routine après utilisation si besoin.
ATTENTION ! Dans les versions AT&T, le signal capturé n'est pas masqué dans la routine (en BSD oui).

Structure **sigaction**

```
<sigaction.h>
struct sigaction {
     void (*sa_handler)(int); //pointeur sur une fonction qui
prend un entier en paramètre et qui ne renvoie rien
     sigset_t sa_mask; //signaux supplémentaires à masquer
     int sa_flags; //options
```

Appel système qui permet de capturer des signaux :
`int sigaction(int sig, struct sigaction* nouveau, struct sigaction *ancien);`
→installe la capture de `sig` décrite dans `nouveau` si `ancien ≠ NULL`, envoit l'ancienne capture
Retourne `0` ou `-1` si capture de `SIGKILL` ou `SIGSTOP`

Options :
`SA_NOCLDSTOP` : (`sig = SIGCHLD`) pas de `SIGCHLD` à l'arrêt ou à la continuation d'un fils.
`SA_RESETHAND` : Pour avoir un comportement AT&T.
`SA_NODEFER` : Ne pas masquer le `sig` dans la routine.

Problème : écrire un programme prenant un entier `n` en paramètre (ligne de commande), créant un fils, attendant `n SIGUSR1` du fils. Affiche le nombre de `SIGUSR1` reçus.

Démasquage et attente atomiques de signaux

```
int sigsuspend (sigset_t *e)
```

`-1, errno=EINTR` — signaux à démasquer

2 IPC System V

IPC = Inter Process Communication

3 sortes d'IPC :
- Segments de mémoire partagés (SMP)
- Files de messages (FM)
- Ensemble de sémaphores (ES)

Fichiers d'inclusion :
`<sys/ipc.h>`
`<sys/shm.h>` →SMP
`<sys/msg.h>` →FM
`<sys/sem.h>` →ES

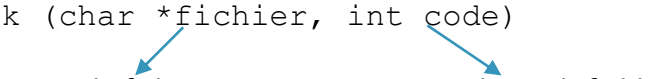
Pour chaque type d'IPC :
→Création / récupération :
`shmget`
`msgget`
`semget`

→Modification / destruction
`shmctl`
`msgctl`
`semctl`

Identification des IPC :
2 identifiants :

- externe, choisi par le créateur de l'IPC (**clé**).
- interne, interne, utilisé par le noyau (**identifiant**).

Gestion des clés :
`key_t ftok (char *fichier, int code)`



bijection `fichiers x char → key_t`
`N° Inode`

Attention !!

- Les IPC sont System-wide ➔ limites physiques sur le numéro et la taille des IPC. (`ipcs -l` pour connaitre les limites)
  - SHM : #max SHM : 4096

  (on peut créer au maximum 4096 segments de mémoire partagés)
  taille max : 32 Mo
  taille totale max : 8 Go
  taille minimal : 1 octet (Attention à la fragmentation interne)
  - FM : #max FM : 1711
  taille max message : 8 Ko (POSIX)
  taille totale max : 16 Ko
  - ES : #max ES : 128
  #max S/ES : 250
  #max S : 32000
  nb max d'opération simultanées : 32
  valeur max d'un S : 32767

Les IPC survivent à leur créateur. Destruction EXPLICITE

- (AS shmctl, msgctl, semctl)
- commande ipcrm

Tous les IPC ont des attributs :

- propriétaire : celui du processus créateur de l'IPC
- groupe : ……..

- droits : `rwx pour ugo`

- modifiés par `shmctl, msgctl, semstl`

2.1 Les SMP

Un processus P crée un SMP.
Les processus qui utilisent le SMP vont s'**attacher** au SMP.

```
int shmget (key_t clé, size_t taille, int droits)

key_t clé → obtenu par ftok (ou IPC_PRIVATE)
int droits → combinaison de  IPC_CREAT
                             IPC_EXCL
                             droits(rw, ugo)
shmget renvoie :
>0 : id du SMP
-1 : erreur
```

→Si `clé = IPC_PRIVATE`, création du SMP (ou récupération du SMP)
→Sinon Si le SMP de clé `clé` n'existe pas :
Si `IPC_CREAT` non positionné : erreur, `errno = ENOENT`
Sinon Création du SMP avec les droits spécifiés (LINUX : SMP initialisé à 0, **POSIX : contenu quelconque**).
Sinon : si `IPC_CREAT` et `IPC_EXCL` : erreur `errno = EEXIST`
sinon renvoie l'ID du SMP

Création typique de SMP :

```
cle_smp = ftok ("toto", 'a');
     if ((id_smp=shmget(clesmp, n*sizeof(truc), IPC_CREAT |
IPC_EXCL | 0660)==-1)
     {
          …
     }
```

Récupération typique :
```
cle_smp = ftok ("toto", 'a');
if((id_smp=shmget,0,0)==-1) {…}
```

Appel système d'attachement d'un SMP à l'espace d'adressage :
```
void *shmat (int ID, void *adresse, int options)

ID : obtenu par shmget
void *adresse : adresse d'attachement
options : 0 ou SHM_RDONLY (lecture seule)
```
la fonction retourne l'adresse d'attachement (ou -1)
Convention POSIX : si adresse = NULL, c'est le noyau qui choisit l'adresse d'attachement.

(SEUL moyen de rendre les applis portables)

exemple : `p=shmat(idsmp, NULL, 0)`

Détachement d'un SMP de l'espace d'adressage.

```
int shmdt (void *adresse)

*adresse = retour du shmat
```

Remarque : A la terminaison d'un processus, `exit` détache tous les SMP.
Un SMP n'est officiellement détruit que si :

- il est marqué à détruire
- aucun processus ne l'a attaché à son EA.

Contrôle du SMP :

```
int shmctl (int id, int op, struct shmid_ds *buffer)

int id : obtenu par shmget
int op :  IPC_RMID : marqué "A détruire"
          IPC_STAT : récupérer infos système dans buffer
          IPC_SET : modifier les infos système comme spécifié
dans buffer
retourne 0 ou -1.
```

2.2 Files de messages

Création :
`msgget`
Poster :
`msgsmd`
Relever :
`msgrcv`
Modif/destruction :
`msgctl`

Nature d'un message :

```
struct {
     long mtype; ←obligatoirement le premier
     …
     …
}
```

Remarque : dans `msg.h` :

```
struct msgbuf {
     long mtype;
     char mtext[1];
}
```

⇨ INUTILISABLE ‼

Création :

```
int msgget (key_t clé, int options)
```

clé : obtenu par ftok ou IPC_PRIVATE
options : cf shmget

Algo : cf shmget

msgget retourne l'id de la file de message ou -1.

Envoi de message :

```
int msgsnd (int ID, void *message, size_t taille, int options)
```

ID : identifiant de la file de message dans laquelle on poste
taille : taille du message qu'on envoit (sauf champ mtype)
options : IPC_NOWAIT

msgsnd retourne 0 ou -1.
→Normalement bloquant sur file de message pleine (si interrompu, retour -1, errno = EINTR)
→non bloquant si IPC_NOWAIT a été positionné (si FM pleine, retour -1, errno = EAGAIN)

Retrait de message :

```
int msgrcv (int ID, void* adresse, size_t taille, long type,
int options)
```

ID : obtenu par msgget
adresse : adreesse où le msg doit être écrit
type : type du message qu'on veut retirer
>0: le plus vieux msg de ce type
=0: le plus vieux msg
<0: le plus vieux msg de type le plus petit inférieur à typei
options: - IPC_NOWAIT
- MSG_NOERROR (message trop gros tronqué en silence)

msgrcv retourne la longueur du message reçu ou -1
→Normalement bloquant sur file de message vide … cf msgsnd

Destruction/Modification:

```
int msgctl (int ID, int op, struct msqid_ds * p)
```

```
op:  IPC_RMID  Destruction
     IPC_STAT  Consultation attributs (résultat dans *p)
     IPC_SET   Modification attributs selon *p
```

2.3 Ensembes de sémaphores

Sémphore d'exclusion mutuelle

P(s) Prendre le sémaphore
V(s) Libérer le sémaphore
Z(s) Attente du passage à 0 du sémaphore

Permet de protéger une ressource, la synchronisation et même l'ordonnencement

Si chaque processus attend l'autre: Dead Lock / Etreinte mutuelle

Problèmes de lecteurs / rédacteurs

Contraintes:

→Au plus un rédacteur, et alors 0 lecteurs
→Autant de lecteurs qu'ont veut

Lecteurs prioritaires:

→ Sémaphore mutex écriture
→le 1er lecteur prend écriture
→var globale, vue par tous les lecteurs: nombre_lecteurss
→mutex nombre protège l'accès à nombre_lecteurs
→le dernier lecteur rend écriture

Plus général: Réservation/libération atomique

Opération (sur 1 sémaphore d'un ES):

```
struct sembuf
{
     ushort sem_num; //num du semaphore sur lequel porte
l'opératoion
     short sem_op;
     int sem_fly;
}
```

sem_op:
<0: réservation de |sem_op| exemplaires de la ressource (P)
=0: attente de passage à 0 (Z)
>0: Libération de sem_op exemplaires de la resource (V)

Création/Récupération

```
int semget (key_t clé, int nombre, int options)

Retourne: id de l'escié (j'ai pas compris)sinon -1
clé: obtenu par ftsk (ou TPC_PRIVATE)
nombre: #de sem de l'ensemble
options: IPC_CREAT, IPC_EXCL, droits
```

Réservation/libération de sémaphores

```
int semop (int id, struct sembuf *tab_op, int nb_op)
```

IPC_NOWAIT: Rend l'opération non bloquante
SEM_UNDO: à la terminaision du processus, l'opération est annulée
tx: sémaphore mutex

Création

```
(id_mutex=semget(clé,1,IPC_CREAT|IPC_EXCL|0660)) == -1
```

Récupération

```
(id_mutex=semget(clé,0,0)) == -1;

short sembuf   P={0,-1,SEM_UNDO},
               V={0,1,SEM_UNDO};
```

Réservation

```
semop(id_mutex,&P,1)
```

Libération

```
semop(id_mutex,&V,1)
```

Consultation/modification d'un ES

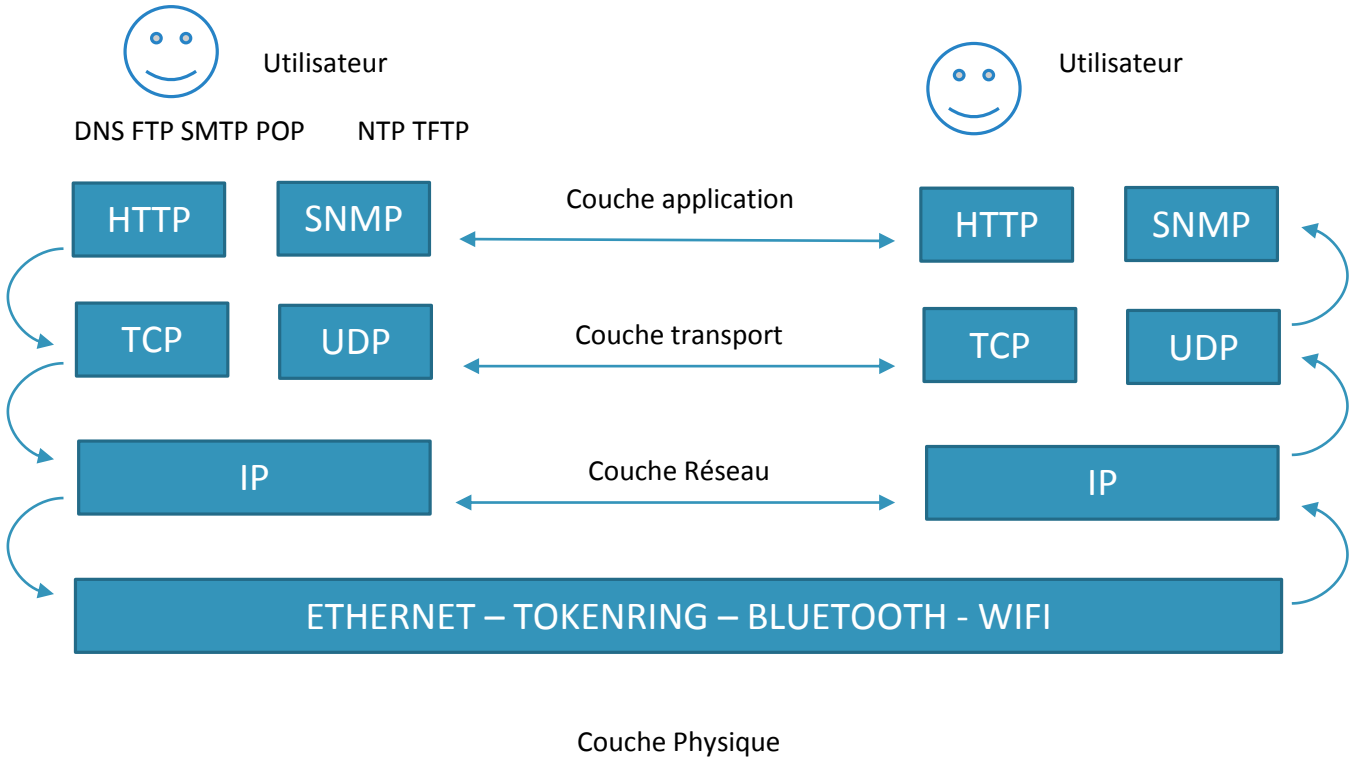
```
semctl(int id, int semmnum, int op {, struct semid_ds *p});
```

id: Obtenu par semget
op: Code opération

(Voir fiche tableau (op, semnum, arg, retour & effet))

3. Sockets BSD

- Utilisable en local
- Utilisable en réseau



Modes de comunication

- Mode connecté: Etablissement d'une "liaison" entre interlocuteurs via un **protocole**
- Mode non connecté: Envoi de message sans établissemenet de liaison

TCP: Protocole "Fiable" orienté liaison
Fiable: Acheminement garanti, ordre des messages respecté

UDP: protocole "Non fiable" orienté message
Message envoyés et reçus entiers

Caractéristique d'une socket

- **Domaine:**

Espace de communication utilisé, caractérise la frame d'une adresse

Ex:

AF_UNIX	Sur la même machine
AF_INET	Internet

- **Type:**

Modalités de communication

Pour AF_INET:

SOCK_RAW:	Couche IP (root)
SOCK_DGRAM	Messages, mode non connecté, non fiable (sockets UDP)
SOCK_STREAM	Mode connecté, fiable (sockets TCP)
SOCK_SEQPACKET	Messages, en mode connecté, fiable

Création de socket

```
int socket (int domaine, int type, int protocole)
```

```
Retour: Identifiant de la socket créer (ou -1 si pb)

domaine: AF_INET
type: SOCK_STREAM ou SOCK_DGRAM
protocole: 0 protocole par défaut
```

3.1 Sockets UDP

Socket = 2 buffers (lecture/écriture) (E/S normalement bloquants)

Il faut brancher la socket dans un port de communication

Branchement d'une socket

```
int bind (int ID, struct socketaddr *ptr, int lgr_adresse)
```

```
Retour: 0 ou -1
ID: Obtenu par socet
*ptr: Adresse du port de branchement
lgr_adresse: Taille de l'adresse de branchement
```

Rien n'est simple: "Le type struct sockaddr est inutilisable tel quel"
→ Dépend du domaine de la socket

```
<netinet/in.h>
→Adresse d'une machine:

struct in_addr
{
     in addr_t      s_addr;
}
→ Adresse Internet d'une socket

struct sockaddr_in
{
     short sin_family; //Nécessairement AF_INET
```

```
    ushort sin_port; //Si port de branchement
    /*htons(0) ou htons(port)
    struct in_addr sin_addr; //Adresse IP machine
    /*→INADDR_ANY */
    char sin_zero[sizeof(struct sockaddr)-sizeof(short)-
sizeof(ushort)-sizeof(struct in_addr)];
    (char sin_zero[8];) (TRU64)
}
```

Numéro de port:
Il y a 65536 ports/entité de transport
Les ports de n° ≤ IPPORT_RESERVED sont réservés au noyau
Il est recommandé d'utiliser un n° de port > IPPORT_USERRESERVED (5000)

Dans une architecture client/serveur

N° de port de la socket côté serveur, n° conventionnel, commun de tous.tili

N° de port de la socket côté client: 0 veut dire "n'importe quel port disponible (>IPPORT_USERRESERVED)

Tout se complique: `htons(0)` au lieu de 0

```
{hton, ntoh} × {s,l}
(h = host ; n = network ; s = short ; l = long)
```

Utilisation de la socket UDP (user datagram protocol)

Remarque : →Aucune erreur signalée sur la machine distante
→ Taille datagrammes < 9000 octets

Réception d'un message :

```
int recvfrom(int sock, void* buffer, int lgr, int option,
struct sock_addr *adresse, socklen_t *lgr_adr)

sock : id de la socket obtenu par l'appel
buffer : adresse à laquelle on va recevoir le message
lgr : nb d'octets réservés pour le message (si le message est plus
gros que ce nb d'octets, il est tronqué en silence)
options : 0 ou MSG_PEEK (lire le message sans l'extraire)
adresse : adresse de l'expéditeur
lgr_adr : longueur de l'adresse de l'expéditeur, initialisé à
l'appel
La fonction retourne le nombre d'octets lus.
```

→ Messages extraits entiers
→ AS bloquant syr buffer réception vide

Emission d'un datagramme

```
sendto (int id, void * msg, int lgr, int option, struct
sock_addr * adresse, socklen_t lgr_adresse)
```

Mais Jamy ? Comment on connait l'adresse à qui on veut envoyer le message ?!

Et bien Fred c'est très simple !
Obtention d'une adresse IP à partir d'un nom :

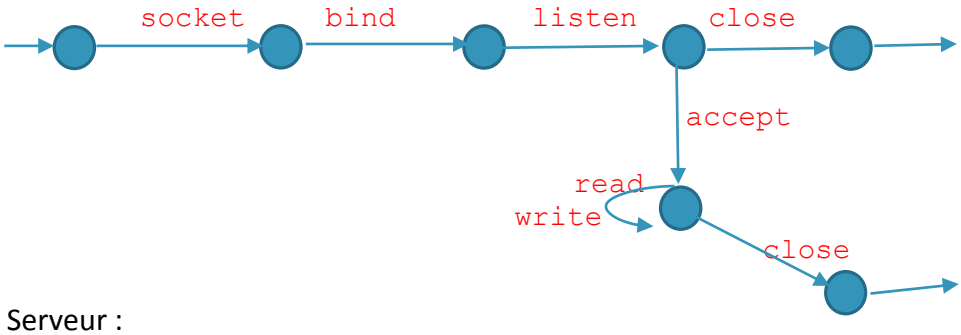
```
<netdb.h>
struct hostent *gethostbyname(char * nom)

struct hostent {
     char * h_name; //nom officiel
     char ** h_aliases; //liste des autres noms
     int h_addrtype; //toujours AF_INET
     int h_length; //longueur de l'@
     char ** h_addr_list; //liste des @IP
     #define h_addr h_addr_list[0]
}
```

Sockets TCP

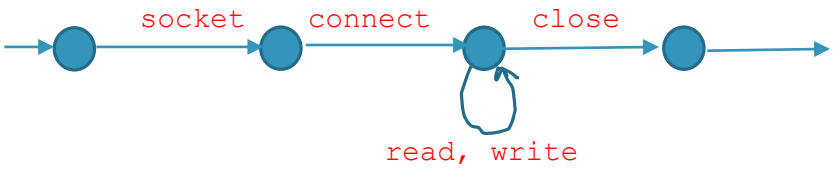
Différences avec UDP :

- Serveur : attend les connexions, duplication de socket à la connexion
- Client : Pas de branchement (bind) mais connexion direct avec le serveur distant
- Socket : ≡ fichier régulier

Serveur :



Serveur :




```
int listen (int sock, int maxattente)

sock : id obtenu par socket
maxattente : taille file d'attente (Linux ⩽128)
retourne 0 ou -1

int accept (int sock, struct sockaddr * addr, socklen_t * lgr)

retourne l'id de la copie de la socket ou -1
addr : @du client
```

lgr : longueur de l'adr

connect (int sock, struct sockaddr *adr, socklen_t lgr)
retourne 0 ou -1 (erreur fréquente: errno=ECONNREFUSED)

Entrées / Sorties : Système de fichiers

"Perhaps the messiest aspect of OS is I/O"

SGF :
→ ∀ fichiers/devices
→ Partition
→ Types et Algos

EXT4 :

`<linux/ext3_fs.h>`

Superbloc : descripteur du SF
`u32 s_inode_count ;` /*nombre d'inodes dans la partition*/
→ Au maximum 2^{32}-1 fichiers ≈ 4.10^9
`u32 s_blocks_count ;` /*nombre maximal de blocs de données*/
→ Au max 4.10^9 blocs
`EXT3_BLOCK_SIZE :` taille d'un bloc
si 1K : 4 To max
si 4K : 16 To max

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

On obtient l'inode d'un fichier avec l'appel :

```
int stat (char* nom, struct stat * inode)
```

(0 ou -1)

```
struct stat {
    dev_t st_dev ;
    ino_t st_ino ;
    node_t st_node; //droits sur le fichier et type du fichier
    nlink_t st_nlink; //nb de liens physiques sur le fichier
    uid_t st_uid; //identifiant du propriétaire
    gid_t st_gid; //groupe du propriétaire
    off_t st_size; //offset max du fichier
    time_t st_atime;//date de dernière consultation du fichier
    unsigned long int st_atimensec; //LINUX pas POSIX (en ns)
    time_t st_mtime; //date de dernière modif du contenu
    unsigned long int st_mtimeensec; //en nanoseconde
    time_t st_ctime; //dernière modif de l'inode
    unsigned long int st_ctimeensec; //en nanoseconde
    unit_t st_blksize; //taille "préférée" d'un bloc de donnée
    int st_blocks ; //nb de blocks qu'occupe le fichier
}
```

Types de fichiers :

1) Fichier régulier

```
S_ISREG(  )
```

champ `st_node` (renvoyer par `stat`)

2) Répertoire

→ Liste de couple (n° d'Inode, char *)

```
S_ISDIR(  )
```

3) Lien symbolique

→ "Pointeur" vers une entrée de répertoire.

```
S_ISLNK(  )
```

4) Fichiers spéciaux "caractère" (c)

```
S_ISCHR(  )
```

→ unités d'E/S orientées "caractère"
→E/S non tamponnés
→ granularité : octet

5) Fichiers spéciaux "blocs" (b)

```
S_ISBLK(  )
```

→ E/S orientés "blocs"
→ Granularité bloc de donnée
→ E/S Tamponnées

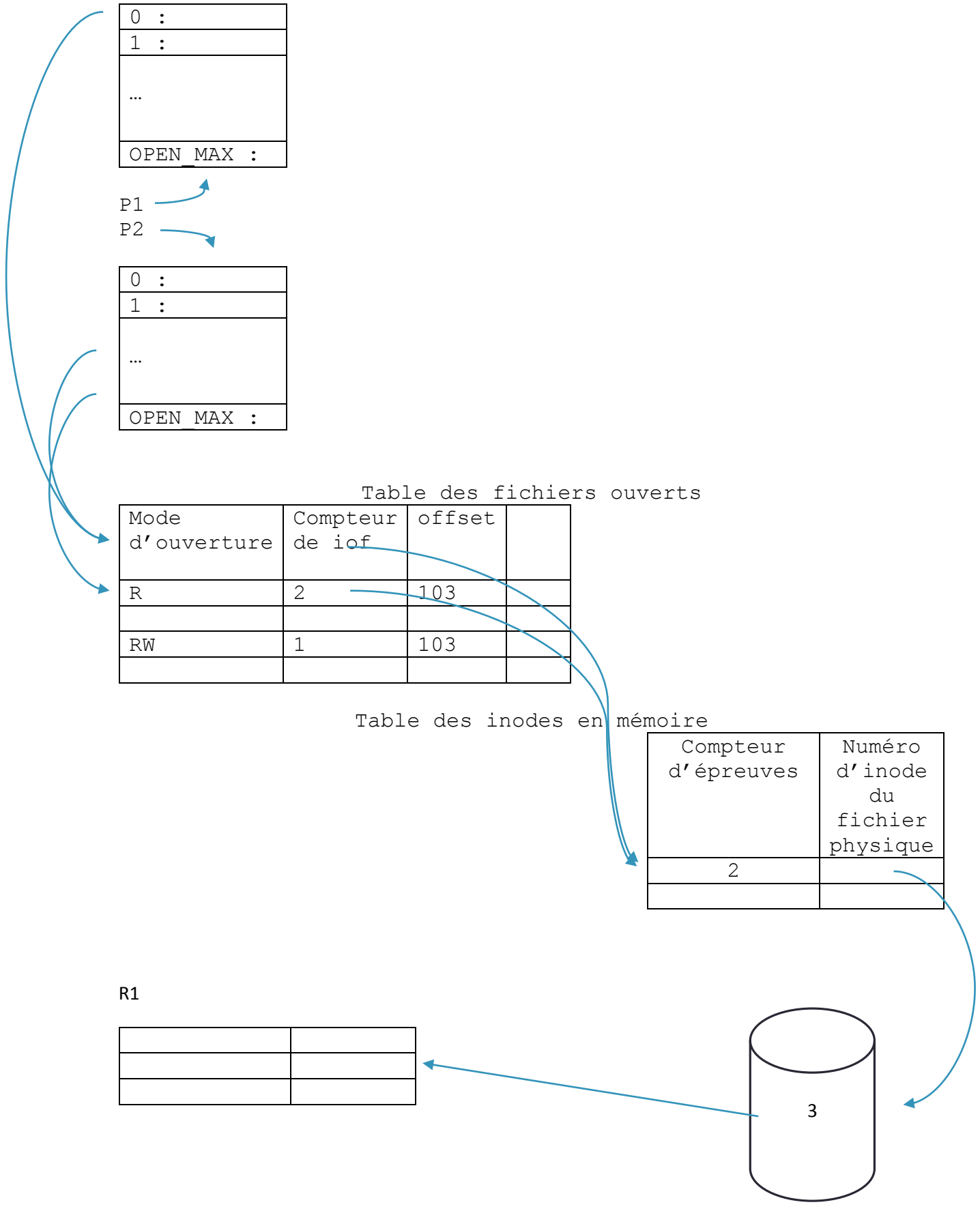
6) Sockets (s)

```
AF_UNIX
AF_INET

S_ISSOCK(  )
```

7) Tubes (p)

```
S_ISFIFO
```

Table du Système


0 :
1 :
…
OPEN_MAX :
P1
P2
0 :
1 :
…
OPEN_MAX :
Table des fichiers ouverts
Mode d'ouverture
Compteur de iof
offset
R
2
103
RW
1
103
Table des inodes en mémoire
Compteur d'épreuves
Numéro d'inode du fichier physique
2
R1
3

E/S bas niveau

Ouverture

```
int open (char *nom, int mode {, mode droits}) ;
```

Renvoi : 0 ou -1

mode: combinaison bit à bit de

1) Un param O_RDONLY, O_WRONLY, O_RDWR (/ !\ O_RDWR != O_RDONLY | O_WRONLY)
2) Au choix parmi
   O_TRUNC: fichier vide à l'ouverture (s'il existe)
   O_CREAT: création
   O_EXCL: (avec O_CREAT) creation exclusive
   O_APPEND: écriture en fin de fichier
   O_NONBLOCK: rend non bloquant les E/S sur le fichier

Fermeture

```
int close (int descr) ;
```

Renvoi : 0 ou -1

Lecture

```
size_t read (int desc, void *buf, size_t lgr) ;
```

Renvoi : -1 ou 0 (au début de la lecture)
Ou >0 taille des octets lus (<= lgr)
Desc : obtenu par open
***buf :** @ où ranger les octets lus
lgr: taille d'octets à lire

→ Non temponné dans l'EA du pus

Ecriture

```
size_t write (int desc, void *buf, size_t lgr) ;
```

Renvoi : -1 ou 0 (au début de la lecture)
Ou >0 taille des octets lus (<= lgr)
Desc : obtenu par open
***buf :** @ où trouver les octets lus
lgr: taille d'octets écrits

→ Non temponné dans l'EA du pus

Synchronisation MC/MD

```
void sync();
```

Ne renvoie rien car toujours réussit.

Contrôle de l'offset :

```
<vnistd.h>
off_t lseck (int dscr, off_t offset, int origine)

dscr est obtenu par open
offset : décalage (en octets)
origine : origine du décalage
Renvoit la nouvelle position de l'offset ou -1

origine :  - SEEK_SET : début du fichier
           - SEEK_CUR : position courante
           - SEEK_END : fin du fichier
```

/!\ Impossible de déplacer l'offset avant le début du fichier, mais possible de le déplacer après la fin du fichier (création de fichier à trou)

Répertoires :

→ Répertoire = suite de couple (nom de fichier, numéro d'Inode)

```
<dirent.h>

DIR * opendir (char *nep);
```

renvoit pointeur vers la structure réperoire ou NULL si il y a un problème. Le malloc est fait par le noyau.

Entrée de répertoire :

/*norme POSIX*/

```
struct dirent {
      int_t d_ino ;
      char * d_name;
}
```

/*LINUX*/

```
struct dirent {
      int_t d_ino;
      off_t d_off;
      ushort d_reclen;
      uchar d_type;
```

```
      char d_name[256];
}
```

/*TROU 64*/

```
struct dirent {
      int_t d_ino;
      ushort_t d_reclen;
      ushort_t d_namelen;
      char * d_name;
}
```

Lecture du répertoire :

```
readdir (DIR * rep)
```

```
rep : renvoyé par opendir
```

Entrée courante : Offset (ou curseur) d'un répertoire positionné sur l'entrée suivante.
Retour NULL si pb ou en fin de répertoire.

```
void rewinddir (DIR * rep)
```

positionne l'offset du répertoire sur la première entrée.

Fermeture :

```
int closedir (DIR * rep)
```