QL : Cheatsheet

Next Revision	@November 1, 2023
Status	In progress
Exam	Qualité Logiciel : Exam #1

La Manipulation des inodes 🟢

La structure stat

On ne peut pas changer le type de fichier

▼ Que caractèrise **un fichier** ?

- Inode
- Partition : se trouve dans la structure **stat**

▼ Qu'est ce qu'un inode ?

L'inode : identité unique **par partition** de fichier , une **structure** , un ensemble des champs (parmi ces champs , **l'inode**)

▼ Qu'est ce qu'une table des inodes ?

Une table des inodes : une table à taille fixe qui contient les inodes de l'ensemble des fichiers , meme si on a de l'espace physiquement , une fois cette table est remplie on ne peut plus créer des fichiers sur le système .

▼ Que signifie **la structure stat** ?

La structure stat : permet l'accès au **caratèristiques** d'un fichier (l'état de fichier : la taille , les permissions ..)

▼ Comment trouver des informations sur **la structure stat**?

```
# pour trouver la fonction stat
# chercher tout les commandes et fonctions qui contient stat
man -k stat | grep '^stat'
```

```
# s'il se trouve dans la section 2
man 2 stat
```

```
  The stat structure
      All of these system calls return a stat structure,  which  contains
      the following fields:

          struct stat {
              dev_t     st_dev;         /* ID of device containing file */
              ino_t     st_ino;         /* Inode number */
              mode_t    st_mode;        /* File type and mode */
              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;     /* Block size for filesystem I/O */
              blkcnt_t  st_blocks;      /* Number of 512B blocks allocated
*/

              /* Since Linux 2.6, the kernel supports nanosecond
                 precision for the following timestamp fields.
                 For the details before Linux 2.6, see NOTES. */

              struct timespec st_atim;  /* Time of last access */
              struct timespec st_mtim;  /* Time of last modification */
              struct timespec st_ctim;  /* Time of last status change */

          #define st_atime st_atim.tv_sec      /* Backward compatibility */
          #define st_mtime st_mtim.tv_sec
          #define st_ctime st_ctim.tv_sec
          };
```

▼ Que signifie le champ **st_mode ?**

st_mode : Définit les **droits d’accès** et le **type des fichiers .**

▼ Quels sont les macros de **stat** pour récuperer le type ?

Pour le type les macros ci-dessous prennent en argument le champs **st_mode** de la structure **stat** , ces macros sont définis par POSIX et prennent une valeur **vrai** (**TRUE**) si le fichier correspondant au type indiquié .

- **S_ISREG : renvoie TRUE s’il s’agit d’un fichier régulier**
- **S_ISDIR : renvoie TRUE s’il s’agit d’un fichier répertoire**
- **S_ISLINK** : **renvoie TRUE s’il s’agit d’un lien symbolique**
- **S_ISBLK : renvoie TRUE s’il s’agit d’un bloc spécial**
- **S_ISCHR : renvoie TRUE s’il s’agit d’un spécial caractère**
- **S_ISFIFO : renvoie TRUE s’il s’agit d’un tube nomé**

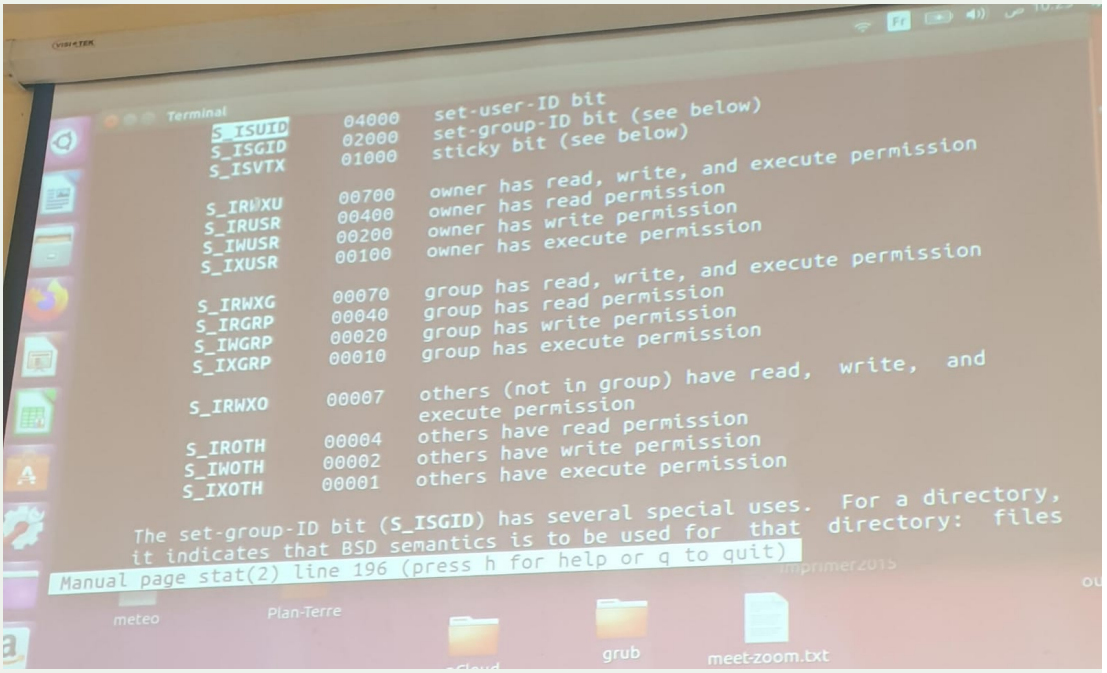
▼ Comment récuperer **les droits d’accès** à partir de **stat** ?

OU binaire vs OU logique

ET binaire vs ET logique

> ## Les droits sont sur 12 bits

Pour les droits d’accès on cummule par l’intermédiare d’un **‘ | ‘ (le OU binaire)** les constants suivantes :




```
man 2 stat
# pour chercher une valeur dans le manual
/keyword
#exemple
/S_IS
```

La fonction stat

▼ Qu’est ce qu’une **fonction stat** ?

Elle permet d’obtenir dans un objet de structure **stat** les informations relatives à un fichier donné .

```
struct stat info;
// remplir la structure stat par les information de fichier file1 ( chemin )
stat("file1",&info);
```

▼ Comment connaitre **les droits d’accès** des fichiers ?

Pour connaitre les droits d’accès des fichiers , on prends les constants symbolique (IS_IRUSR ..) et en utilisant un **ET Binaire (&)** on compare au champs **ST_MODE** de la structure .

Pour plusieurs droits , on compare un par un .

La manipulation des liens physiques

▼ Comment créer un **lien physique** sur un fichier existant ?

- **Un lien physique :** une réference sur un fichier (meme fichier et non pas une copie)

Permet de créer un nouveau lien physique **file2** sur le fichier **file1 .**

- **retourne -1** : ça se passe mal
- **retourne 0** : ça se passe bien

```
int link(const char *oldpath,const char *newpath);
//exemple
link("file1","file2");
```

▼ Comment supprimer un **lien physique** ?

> Pour supprimer un fichier physiquement Il faut **supprimer tous ses liens physiques .**

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

▼ Comment renommer un fichier ?

```
int rename(const char *oldpath,const char *newpath);
```

▼ Comment changer les droits d’un fichier ?

Attribue au fichier **file** les droits d'accès définies par le deuxieme arg **mode** .

- Le mode est définie par desjonction des constantes (OU binaire)

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

▼ EXCERCICE

Ecrir un programme en language C qui permet d'afficher le premier champs de la commande **ls -l** d'un fichier passé en argument

```
void affiche(const char* file){
struct stat info;
lstat(file,$info);
//type de fichier
if(S_ISREG(info.st_mode))
printf("-");
else if(S_ISDIR(info.st_mode))
printf("d");
else if(S_ISLINK(info.st_mode))
printf("l");
else if(S_ISBLK(info.st_mode))
printf("b");
else if(S_ISCHR(info.st_mode))
printf("c");
else if(S_ISFIFO(info.st_mode))
printf("p");

//Droits du propriétaire
(info.st_mode & S_IRUSR ) ? printf("r") : printf("-");
(info.st_mode & S_IWUSR ) ? printf("w") : printf("-");
(info.st_mode & S_IXUSR ) ? printf("x") : printf("-");
//Droits du groupe
(info.st_mode & S_IRGRP ) ? printf("r") : printf("-");
(info.st_mode & S_IWGRP ) ? printf("w") : printf("-");
(info.st_mode & S_IXGRP ) ? printf("x") : printf("-");
//Droits des autres
(info.st_mode & S_IROTH) ?printf("r"):printf("-");
printf('\n');

}

int main(int argc,char *argv[]){
int i;
  affiche(argv[1]);
return EXIT_SUCCESS;
}
```

▼ Comment compiler un programme en Linux ?

`gcc -o EXO program.c`

▼ Comment executer un programme en Linux ?

`./EXO`

▼ Traitement des fichiers 🟡

▼ Comment **ouvrir** un fichier ?

> Il faut qu'on a les droits pour écrir ou lire
> Il faut toujours récuperer le numéro de descripteur à partir de **open**

Pour écrire dans un fichier , il faut l'ouvrire en utilisant **open .**

La primitive **open** permet à un processus de réaliser une ouverture de fichier .

```
// fichier déja existant
open(char* ref,int mode_ouverture)
// nouveau fichier
open(char* ref,int mode_ouverture,/*droits/*)
```

- **ref** : une référence au fichier à ouvrir (chemin absolu ou relative)
- **mode_ouverture :** permet de demander la réalisation d'opération au cours de l'ouverture , il est construit par **disjonction** bit à bit (OU binare |) de constantes macro défini dans le fichier standard **fctnl.h .**

  Cette **disjonction** comporte **exactement** une des ces trois constants suivantes :

  - **O_RDONLY** → Lecture seule
  - **O_WRONLY** → Ecriture seule
  - **O_RDWR** → Lecture/Ecriture

  OU avec :

  - **O_TRUNC** → Si le fichier existe et est regulier , le fichier tranque à l'ouverture → sa taille remise au zero .
  - **O_CREAT** → Si le fichier n'existe pas , on va créer un fichier régulier , dans ce cas il faut préciser les droits d'accès de fichier dans un troisième argument , les droits sont déduis après masquage .

Le propriètaire et le groupe propriètaire sont ceux effectives de processus

- **O_EXCL** → Si **O_CREAT** est positioné et si le fichier existe ⇒ echec de l'instruction **open** et retour -1 (vérifier si le fichier existe pour refuser l'ouverture)
- **O_APPEND :** Dans le cas d'un fichier régulier , avant chaque écriture , la position currente (**offset**) prends pour valeur la taille de fichier ⇒ toute écriture a lieu au fin de fichier

▼ Comment **créer** un fichier ?

Si **le fichier existe** et est régulier sur lequel il a le droit d'écrir → son contenu sera écrasé , sa taille devient nulle .

```
int creat(const char* ,mode_t mode);
```

▼ Comment **fermer** un fichier ?

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

▼ Comment **lire** un fichier ?

Cette fonction lis au plus **count** caractères via le descripteur **fd** , les caractères lus sont écrits dans l'espace d'adressage de processus à l'adresse **buf** .

Pour définir **count** :

#define taille=30;

char buf[taille];

Il retourne -1 dans le cas d'échec sinon il retourne le nombre des caractères lus ou zéro lorsqu'on a dans la fin de fichier

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

▼ Comment **écrire** dans un fichier ?

Ecriture dans le fichier de descripteur **fd** , de **count** caractères trouvé à l'adresse **buf** , dans l'espace d'adressage de processus .

L’écriture a lieu à partir de la position pointée par le curseur de fichier (position currente) ou à partir de la fin de fichier si **O_APPEND** est positioné .

Il retourne le nombre des caractères écrits , -1 dans le cas d’échec .

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

▼ Comment **manipuler** l’offset ?

Il permet de déplacer le **curseur** (la position currente) dans le descripteur **fd** à la nouvelle valeur **offset** .

Le point de départ fourni au troisième argument peut prendre les valeurs suivantes :

- **SEEK_SET** : début de fichier
- **SEEK_CUR** : position currente
- **SEEK_END** : fin fichier

Il renvoie la nouvelle position mesurée en octets à partir de début de fichier

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

▼ Les verrous 🟡

Les verrous des fichiers

▼ Quels sont les **verrous** des fichiers ?

Des méchanismes de control d’accès **concurrent** à un fichier , les verrous sont **rattachées** aux inodes (inode == fichier , pour le système) , on veut protéger le fichier lui meme et non pas simplement aux liens de fichier .

Toutes les ouvertures d’un meme fichier ⇒ tous les descripteurs sur un fichier voient le verrou (ils doivent etre vus par toute les verrous)

La protection réalisée par le verrou **a donc lieu sur le fichier physique** .

▼ A qu’il appartient un **verrou** ?

> Un verrou est la **proprieté d’un seul processus** , et seul le processus propiétaire du verrou peut le modifier ou

> l’enlever , le verrou ne protège pas contre les accèss de processus propriètaire .

Caratèristiques d’un verrou

▼ Quels sont les **deux caratèristiques** d’un verrou ?

- **La portée :** ensemble des caractères et positions de fichier auxquels s’applique le verrou , c’est un **interval** ou une **portion** de fichier ou **jusqu’à la fin** de fichier (si le fichier augmente , les nouvelles positions sont verouillées)
- **Le type :**
  - **Partagé : (F_RDLCK)**

    Plusieurs verrous de ce type peuvent avoir des portées non disjointes (intersection ≠ ensemble vide)

    Exemple : [20, 60] , [50 , 80]
  - **Exclusif : (F_WRLCK)**

    Pas de cohabitation possible avec un autre verrou qlq soit son type

▼ Que signifie le **mode opératoire** ?

Le mode opératoire joue sur le comportement des fonctions **read** et **write** .

Les verrous des fichiers sont soit :

- **Consultatif** : dans ce mode , un verrou n’agit pas sur les **entrées-sorties (read/write : on ne va pas empecher la lecture ou écriture)** , la présence d’un verrou n’est testé qu’à la pose **d’un autre verrou** , la pose sera réfusé s’il existe déja un verrou de **portée non disjointe** et l’un des deux **soit exclusive .**

  L’objectif : empecher les autres verrous d’étre placées .

  > C’est le mode utilisé par défaut

- **Impératif :** La présence de verrou est testée pour la pose d’un autre verrou mais aussi pour les appels systèmes **read/write ⇒ Les verrous ont un impact sur les écritures et les lectures de tout les autres processus .**

Sur les verrous de type **partagé ,** toute tentative **d'écriture** par un autre processus est bloqué sur la **portée** .

Sur les verrous de type **exclusif ,** toute tentative **d'écriture** ou **lecture** par un autre processus est bloqué sur la **portée** .

▼ Comment activer le mode opératoire **impératif** ?

> Ce mode doit etre activé sur la partition contenant le fichier à verouiller et sur le fichier lui meme .

Activer sur la partition : `mount -o remount , mand  /partition`

Activer sur un fichier :

- On désactive le droit x de groupe
- On active le s gid bit

  `chmod g-x,g+s file`

Manipulation des verrous

▼ Qu'est ce qu'**une table de verrou** ?

Le noyau gére une table de verrou dont chaque entrée a une structure **flock** qui peut etre consulté ou modifié avec la primitive **fcntl**

▼ Quels sont les champs de la structure **flock** ?

```
struct flock {
short l_type; //partagé ou exclusif
short l_whence; // meme constants que l_seek
l_start; // la position de début d'interval par
//rapport à l_whence
l_len; // la longeur
...
}
```

▼ Comment utiliser la fonction **fcntl** ?

```
int fctnl(int fd,int operation,struct flock *verrou);
```

1. Créer une variable de type **flock**
2. Remplir les champs de **flock**

3. Utiliser la fonction **fctnl**

On a trois opération possibles pour **operation** :

- **F_SETLKW** : pose bloquante (un **wait**) , s'il existe un verrou **incompatible (l'un des deux est exclusif)** ou s'il on a pas les droits d'accès sur le fichier pour le type de verrou demandé (on attend jusqu'à la raison pour laquelle on ne peut pas poser le verrou est enlevé) .
- **F_SETLK** : pose non bloquante , soit on pose le verrou (succès imédiat) soit la pose échoue et **on retourne -1** en continuant l'execution de programme (verrou incompatible) .
- **F_GETLK :** test l'existence d'un verrou **incompatible** avec le verrou passé en paramètre , si un tel verrou existe alors la structure **flock** passé en paramètre est rempli avec les valeurs de ce verrou incompatible et **L_PID** contient l'identité de processus **propriètaire de verrou incompatible .**

▼ Processus 🟡

FORK()

▼ Comment créer un **nouveau processus** à partir d'un autre ?

Le premier processus système **init** est crée directement par le noyau au démarrage , ensuite la seule manière de créer un nouveau processus est d'appler l'appel système **fork()** qui va dupliquer le processus appelant .

Au retour de cet appel système , deux processus identiques (pére et fils) continuront d'executer le code (meme instruction) , la différence essentielle est le PID (et le PPID) .

▼ Quelle est la valeur de retour de **fork()** ?

La valeur de retour est :

- **Dans le processus père** : PID de processus fils
- **Dans le processus fils** : La valeur est 0 (récupéré à partir de processus parent)
- **fork()** retourne -1 en cas d'erreur : mémoire insuffisante par exemple

```
afficher(PID); // 1000 par exemple
fork(); // executer le meme code restant dans un autre processus
```

```
afficher(PID); // afficher 1000 pour pere - afficher 1001 pour fils
```

▼ Quelle est la différence entre **getpid()** et **getppid()** ?

getpid() : retourne le pid de processus appelant

getppid() : retourne le pid de processus père de processus appelant

▼ Quels sont les identifiants d'utilisateur pour chaque processus ?

Pour chaque processus il existe trois identifiants des utilisateurs :

- **UID réel** : l'utilisateur de celui qui a lancé le programme **réellement**
- **UID effectif** : celui qui correspond aux priviléges (droits d'access) accordé au processus (celui dont les droits sont utilisés lors de l'execution)
- **UID sauvé :** est une copie de l'ancien UID **effectif** lorsque celui si est modifé par le processus

▼ **Exemple**

- file appartient au **user1**

  ls -l file

  -rwxr-xr-x ——— file

  user1 fait ./file

  ⇒ *UID réel = UID effectif = USER2*

- file appartient au **user1**

  chmod u+s file

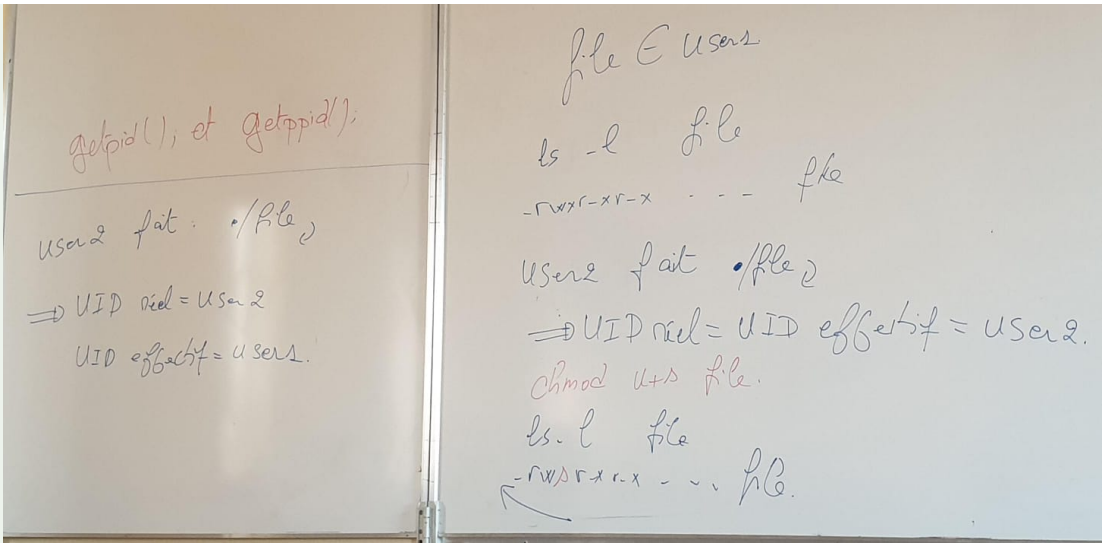
  ls -l file

  -rw**s**r-xr-x ——— file

  user2 fait ./file

  ⇒ *UID réel = USER2 , UID effectif = USER1*

> **Un processus s'execute sous l'identité effective .**



▼ Comment récuperer l’UID réel et effectif d’un processus ?

- **getuid**() : uid réel
- **geteuid**() : uid effectif

▼ Comment changer l’uid effectif ?

- **setuid()**
- **seteuid()**

▼ Qu’est ce qu’un **primitive de recouvrement** ?

Il s’agit d’un ensemble des primitives permettant à un processus de charger en mémoire , en vue de son execution , un nouveau programme binaire pour l’executer :

```
int execlp(char *ref,char *arg,…,NULL)
// chemin absolu dans ref
```

ref : le nom de la commande à executer

arg : spécifie les arguments de cette commande , c’est une liste des variables terminé par un pointeur **NULL**

Le premier argument doit contenir le nom de la commande

```
execlp("ls","ls","-l","/us",NULL);
```

> Cette fonction retourne -1 en cas d'erreur , Si l'opération se passe normalement ne retourne jamais puisque **il détruit (remplace) le code de programme appelant** !!!

▼ Comment se fonctionne **execlp** ?

execlp : cherche la **commande** à executer dans le path **.**

▼ Quelle est la différence ente **execl** et **execlp** ?

Dans **execl** on doit spécifier le path absolu .

```
execl("bin/ls","ls","l","/usr",NULL);
```

▼ Que fait la fonction **exit** ?

C'est une fonction qui ne retourne jamais puisqu'il termine le processus qui l'appelle .

`void exit(int stat);`

stat : permet de savoir ou le programme a terminé ⇒ 0 : fin normal (par convention) , il indique au père l'endroit de cet exit

▼ Comment récuperer l'argument de **exit** ou **return (main)** ?

L'argument de l'instruction **return** de la fonction **main** ou l'argument d'un **exit** est renvoyé au père de processus

Tant que le processus père n'a pas traiter cette valeur , le processus fils reste dans un état appelé **zombie (il libére toutes les resources , mais il est encore dans la table des processus avec status : zombie)**

> **Dans le shell , on n'aura pas des processus zombie .**

Lorsque le processus père a termine à son tour , le fils devient orephelain est **adopté** par le processus **init** qui lit sa valeur de retour permettant à ce processus fils de terminer .

▼ Que fait **wait** ?

Permet à un processus d'attendre la fin de l'un de ses fils .

- Si le processus n’a pas de fils ou si une erreur se produit , **wait** retourne -1
- Si le processus appelant possède au moins un processus fils **zombie** , **wait** retourne le PID d’un fils zombie sans savoir lequel .
- Si le processus appelant possède des fils mais aucun fils zombie , le processu est **bloqué jusqu’à ce que l’un de ces fils devient zombie** .

> Il y a la fonction waitpid , il permet d’attendre un fils bien précis

L’adresse **stat** contient des informations sur la terminaison de ce processus zombie .

```
pid_t wait(int *stat);
```

▼ Quelle le role des **signaux** ?

Ils permettent d’avertit un processus qu’un événement est arrivé .

Certains signaux comme le signal 9 entraire la terminaison de processus , pour d’autres , le processus peut spécifier une fonction **traitante** que sera automatiquement appelée par le système lors de la réception d’un signal particulier

> Il y a certains signaux dont lequels le comportement peut etre modifié (pas le signal 9)

▼ mortel vs non mortel

> Tous les signaux sont mortels par défaut

▼ Comment envoyer un **signal** à un processus ?

On envoie le signal de numéro **sig_num** au processus de pid **pid**

```
int kill(int pid,int sig_num);
```

▼ Comment spécifier une fonction à appeler lors de la réception d'un **signal non mortel** ?

Pour spécifier qu'une fonction c doit etre appelée lors de la réception d'un signal non mortel , on utilise la fonction **signal**

```
mafonction(int ..){
//
}
signal(SIGUSR1,mafonction);
// il faut mettre un sleep pour attendre le processus
sleep(num_sec);
```

▼ Que fait la fonction **pause** ?

La fonction **pause :** permet d'endormir le processus appelant jusqu'à qu'il recoive un signal (il ne doit pas etre mortel sinon le processus termine immédiatement)

```
int pause(void);
```

▼ Que fait la fonction **alarm** ?

La fonction **alarm** permet d'envoyer à soi meme (le meme processus appelant) le signal `SIGALARM` **(signal numéro 14) .**

Ce signal permet de mettre fin à une **temporisation**

```
int alarm(int nb_sec);
```

> **alarm** programme une temporisation pour qu'il envoie un signal **SIGALARAM** au processus en cours dont **nb_sec** second

- Sa réception terminera le processus

▼ Comment utiliser des **alarm** successives ?

Les programmation successives d'**alarm** ne sont pas empilées , chaque appel d'alarm annule la programmation précedente , si **nb_sec** vaut 0 ,

aucune **alarm** n’est planifié (`alarm(0)`)

alarm renvoie le nombre de seconds qui restait de la programmation précedente (annulée) ou 0 sinon

```
alarm(20);
sleep(5);
alarm(30); // renvoie 15 sec
```

```
ssize_t read(int fd,void* buf,size_t count);
off_t lseek(int fd,off_t offset,int whence);
structure flock {
int l_pid;
int l_type;
int l_whence;
int l_start;
int l_len;


}
int fctnl(int fd,struct flock,int operation)
```