FONCTION AUTORISÉE MINISHELL :

Pour les structures présentes dans certaines fonctions, vous trouverez des informations en fin de fichiers.

➔ **ACCESS:** `int access(const char *pathname, int mode);`

- Librairies à inclure:
  - `#include <unistd.h>`
- La fonction vérifie si le le path en fonction du/des mode(s) sélectionnés renseigne en paramètres est valides.
- Valeur de **mode**:
  - `R_OK` : Lecture.
  - `W_OK` : Ecriture.
  - `X_OK` : Exécution.
  - `F_OK` : Existence.
- **Retour:** La fonction renvoie 0 si les permissions demandées sont disponibles ou -1 si l'accès est refusé ou si une erreur survient.

➔ **ADD_HISTORY:** `void add_history(const char *line);`

- Librairies à inclure:
  - `#include <readline/history.h`
- La fonction prend en paramètre une chaîne de caractère qu'elle va ajouter à l'historique actuel des commandes.
- **Retour:** La fonction ne renvoie rien.

➔ **CHDIR:** `int chdir(const char *path);`

- Librairies à inclure:
  - `#include <unistd.h>`
- La fonction prend en paramètre une chaîne de caractère représentant le chemin vers le répertoire où l'on souhaite se déplacer. Ce chemin peut être absolu (commençant par `/`) ou relatif par rapport au répertoire courant.
- **Retour:** La fonction renvoie 0 en cas de succès et -1 en cas d'échecs. En cas d'erreurs, errno et défini:
  - `EACCES (13)`: Permission refusée pour accéder au répertoire spécifié.
  - `ENOENT (2)`: Le répertoire spécifié n'existe pas.
  - `ENOTDIR (20)`: Une partie du chemin spécifié n'est pas un répertoire.

➔ **CLOSE:** `int close(int fd);`

- Librairies à inclure:
  - `#include <unistd.h>`
- La fonction prend en paramètre un descripteur de fichiers que l'on veut fermer.

- **Retour:** La fonction renvoie 0 en cas de succès et -1 en cas d'échecs. En cas d'erreurs, errno et défini:
  - `EBADF (9)` : Le descripteur de fichier `fd` est invalide ou n'est pas ouvert.
  - `EINTR (4)` : La fermeture a été interrompue par un signal avant d'être complétée.
  - `EIO (5)` : Une erreur d'entrée/sortie a été détectée lors de la fermeture du fichier.

➔ **CLOSEDIR:** `int closedir(DIR *dirp);`

- `Librairies à inclure:`
  - `#include <dirent.h>`
- La fonction ferme le flux de répertoire pointé par `dirp`, un pointeur de type `*DIR` (structure représentant un répertoire ouvert), qui doit avoir été préalablement ouvert avec la fonction `opendir()`.
- **Retour:** La fonction renvoie 0 en cas de succès et -1 en cas d'échecs. En cas d'erreurs, errno et défini:
  - `EBADF (9)` : Le descripteur de fichier `dirp` est invalide ou n'est pas ouvert.

➔ **DUP:** `int dup(int oldfd);`

- `Librairies à inclure:`
  - `#include <unistd.h>`
- La fonction crée une copie du descripteur de fichier `oldfd` et retourne le plus petit descripteur de fichier disponible. Ce nouveau descripteur de fichier fait référence à la même ressource que `oldfd` (fichier ou autre) et partage le même offset de lecture/écriture et les mêmes droits d'accès.
- **Retour:** La fonction renvoie le nouveau fd duplique en cas de succès et -1 en cas d'échecs. En cas d'erreurs, errno et défini:
  - `EBADF (9)` : Le descripteur de fichier `oldfd` est invalide ou n'est pas ouvert.
  - `EINTR (4)` : La fermeture a été interrompue par un signal avant d'être complétée.
  - `EMFILE (24)` : Le processus a atteint la limite de descripteurs de fichiers ouverts.

➔ **DUP2:** `int dup2(int oldfd, int newfd);`

- `Librairies à inclure:`
  - `#include <unistd.h>`
- La fonction est similaire à `dup`, mais il permet de spécifier explicitement le descripteur de fichier dupliqué dans `newfd`. Si `newfd` est déjà ouvert, `dup2` le ferme d'abord avant de le rendre une copie de `oldfd`, sauf si `newfd` est identique à `oldfd` (dans ce cas, aucun changement n'est effectué).

- **Retour:** La fonction renvoie `newfd` en cas de succès et -1 en cas d'échecs. Si oldfd = newfd, alors la fonction renvoie newfd. En cas d'erreurs, errno et défini:
  - `EBADF (9)` : Le descripteur de fichier oldfd est invalide ou n'est pas ouvert.
  - `EINTR (4)` : La fermeture a été interrompue par un signal avant d'être complétée.
  - `EMFILE (24)` : Le processus a atteint la limite de descripteurs de fichiers ouverts.

➔ **EXECVE:** **`int execve(const char *pathname, char *const argv[], char *const envp[]);`**

- `Librairies à inclure:`
  - **`#include <unistd.h>`**
- **pathname** : C'est le chemin absolu ou relatif vers l'exécutable que vous voulez exécuter. Il doit pointer vers un fichier exécutable.
- **argv** : C'est un tableau de chaînes de caractères (tableau de `char *`) qui représente les arguments passés au programme. Par convention, le premier élément de `argv` (c'est-à-dire `argv[0]`) est le nom du programme lui-même. Ce tableau doit se terminer par un pointeur `NULL`.
- **envp** : C'est un tableau de chaînes de caractères qui représente l'environnement à passer au nouveau programme. Chaque chaîne doit être au format `clé=valeur`, et ce tableau doit également se terminer par un pointeur `NULL`.
- **Retour:** La fonction ne renvoie rien en cas de succès (le processus est remplacé par le nouveau programme) et -1 en cas d'échecs. En cas d'erreurs, errno et défini:
  - `EACCES (13)`: Le fichier spécifié n'est pas exécutable ou les permissions d'accès sont insuffisantes
  - `ENOENT (2)`: Le fichier spécifié par `pathname` n'existe pas.
  - `EFAULT (14)`: Le chemin d'accès ou l'un des tableaux `argv` ou `envp` pointe en dehors de l'espace d'adressage accessible.
  - `ENOMEM (12)`: Il n'y a pas assez de mémoire pour charger le nouveau programme.
  - `EINVAL (22):` L'un des arguments est invalide, par exemple si `argv` ou `envp` ne se termine pas par `NULL`.

➔ **EXIT:** **`void exit(int status);`**

- `Librairies à inclure:`
  - **`#include <stdlib.h>`**
- La fonction prend un entier qui représente le code de retour du programme. Il est transmis au système d'exploitation pour indiquer l'état final du programme.

➔ **FORK:** `pid fork(void);`

- Librairies à inclure:
  - `#include <unistd.h>`
- La fonction duplique le processus actuel (le parent) et crée un nouveau processus (le fils/enfant). Le processus enfant est une copie presque identique du processus parent, mais avec un identifiant de processus (PID) distinct.
- **Retour:**
  - **Dans le parent:** La fonction renvoie le PID du processus enfant en cas de succès et -1 en cas d'échecs.
  - **Dans l'enfant:** La fonction renvoie 0.
  - **Errno:**
    - `EAGAIN (11)`: Il n'y a pas assez de ressources, l'utilisateur a atteint le nombre maximal de processus.
    - `ENOMEM (12)`: Il n'y a pas assez de mémoire pour créer un nouveau processus.

➔ **FSTAT:** `int fstat(int fd, struct stat *buf);`

- Librairies à inclure:
  - `#include <unistd.h>`
  - `#include <sys/types.h>`
  - `#include <sys/stat.h>`
- La fonction est utilisée pour obtenir des informations sur un fichier déjà ouvert via un descripteur de fichier. Elle est similaire à la fonction *`stat`, mais elle s'applique à un fichier déjà référencé par un descripteur (c'est-à-dire après l'ouverture du fichier avec un appel à `open`). Ces fonctions appartiennent à la bibliothèque standard C, notamment définies dans `<sys/stat.h>` (Voir le man pour la structure).
- **Retour:** La fonction renvoie 0 en cas de succès et -1 en cas d'échecs et errno est défini:
  - `EFAULT (14)`: Le pointeur `buf` invalide.
  - `EBADF (9)`: Le descripteur de fichier `fd` est invalide ou n'est pas ouvert.
  - `EOVERFLOW (75)`: Certaines valeurs dans la structure *`stat` ne peuvent pas être représentées dans les types de données utilisés.

➔ **GETCWD:** `char *getcwd(char *buf, size_t size);`

- Librairies à inclure:
  - `#include <unistd.h>`
- La fonction est utilisée pour obtenir le chemin absolu du répertoire de travail actuel dans un programme. Elle prend en paramètres Un pointeur vers un buffer où le chemin absolu du répertoire de travail actuel sera stocké et La taille (en octets) du buffer. Cette taille doit être suffisante pour contenir le chemin complet, y compris le caractère nul `\0` de fin de chaîne..

- **Retour:** La fonction renvoie retourne un pointeur vers `buf` cas de succès et NULL en cas d'échecs et errno est défini:
  - `EACCES (13)`: Permission refusée pour lire le répertoire courant ou un des répertoires parents.
  - `ERANGE (34)`: La taille du buffer est insuffisante pour contenir le chemin absolu.
  - `EFAULT (14)`: Le pointeur `buf` pointe en dehors de l'espace d'adressage accessible.
  - `ENOMEM (12)`: Il n'y a pas assez de mémoire.

➔ **GETENV:** **`char *getenv(const char *name);`**

- `Librairies à inclure:`
  - **`#include <stdlib.h>`**
- La fonction récupère la valeur d'une variable d'environnement qui est spécifié par `name`.
- **Retour:** la fonction retourne un pointeur vers une chaîne de caractères contenant sa valeur en cas de succès (cette chaîne ne doit pas être modifiée). Elle renvoie NULL en cas d'échecs.

➔ **IOCTL:** **`int ioctl(int fd, unsigned long request, ...);`**

- `Librairies à inclure:`
  - **`#include <unistd.h>`**
  - **`#include <sys/ioctl.h>`**
- La fonction est utilisée pour configurer des périphériques matériels, contrôler des canaux de communication et effectuer des opérations spécifiques qui ne peuvent pas être réalisées avec des fonctions standard comme `read` ou `write.`
- Elle prend en paramètres:
- **fd** : Un descripteur de fichier, généralement obtenu après l'ouverture d'un périphérique ou d'un fichier avec `open()`.

- **request** : Un code de commande (ou un numéro de requête) qui spécifie l'opération à effectuer. Ce code est généralement défini par des macros spécifiques à chaque type de périphérique ou opération. Par exemple, pour les périphériques de type terminal, des codes comme `TIOCGWINSZ` peuvent être utilisés pour obtenir la taille du terminal.

- **...** : Ce sont des arguments supplémentaires, qui dépendent de la commande demandée. Par exemple, si l'on obtient la taille du terminal, un `struct` sera passé pour y stocker les informations retournées par la commande.
- **Retour:** Elle renvoie 0 en cas de succès et -1 en cas d'échecs errno est défini:
  - `ENOMEM (12)`: Il n'y a pas assez de mémoire.

- `EINVAL (22):` Cette erreur indique que la commande `request` passée à `ioctl` n'est pas reconnue ou est invalide pour le périphérique ou l'opération cible.
- `EBADF (9)` : Le descripteur de fichier est invalide ou n'est pas ouvert.
- `EIO (5)` : Si une opération d'entrées/sorties échoue.
- `EPERM (1)` : En cas d'opération non permises.
- `ENOTTY (25)` : Cette erreur indique que l'opération demandée n'est pas supportée par le périphérique auquel le descripteur de fichier fait référence.

➔ **ISATTY:** `int isatty(int fd);`

- `Librairies à inclure:`
  - `#include <unistd.h>`
- La fonction vérifie si un fichier ou `fd` correspond à un terminal. Elle est principalement utilisée pour déterminer si l'entrée/sortie standard est connectée à un terminal (tty) ou à un autre type de fichier (comme un fichier ou un pipe).
- **Retour:** Elle renvoie 1 si le `fd` fait référence à un terminal, 0 si le `fd` ne fait pas référence à un terminal. Si une erreur est survenue, -1 est renvoyé et errno est défini:
  - `EBADF (9)` : Le descripteur de fichier est invalide ou n'est pas ouvert.
  - `ENOTTY (25)` : Bien que `isatty` retourne 0 lorsque le descripteur de fichier ne correspond pas à un terminal, en cas d'erreur plus générale, `errno` peut être défini sur `ENOTTY`, indiquant que le fichier n'est pas un terminal. Toutefois, ce cas est assez rare, car la fonction retourne typiquement `0` dans ce cas, sans avoir à ajuster `errno`.

➔ **KILL:** `int kill(pid_t pid, int sig);`

- `Librairies à inclure:`
  - `#include <signal.h>`
- La fonction envoie un signal à un processus. Elle prend en paramètre le signal à envoyer. C'est un nombre qui représente un signal spécifique, comme `SIGKILL`, `SIGTERM`, `SIGSTOP`, etc… La constante `SIGKILL` permet d'arrêter un processus de manière brutale, tandis que `SIGTERM` est un signal de terminaison demandé plus "propre". Mais également le PID du processus à lequel vous voulez envoyer un signal. Ce paramètre peut être :
  - Un **PID positif** : cela désigne un processus spécifique à qui envoyer le signal.
  - **0** : cela signifie envoyer le signal à tous les processus dans le même groupe de processus que celui appelant `kill()`.
  - **-1** : le signal est envoyé à tous les processus pour lesquels l'appelant a la permission, sauf certains processus système.
  - Un **PID négatif** (autre que -1) : un groupe de processus dont le PID absolu est égal à cette valeur.

- **Retour:** Elle renvoie 0 en cas de succès et -1 en cas d'échecs errno est défini:
  - `ESRCH (9)` : le processus spécifié par `pid` n'existe pas.
  - `EPERM (1)` : l'utilisateur n'a pas les droits nécessaires pour envoyer le signal.
  - `EINVAL (22):` le signal spécifié n'est pas valide.

➔ **LSTAT:** **`int lstat(const char *pathname, struct stat *statbuf);`**

- `Librairies à inclure:`
  - **`#include <sys/stat.h>`**
- La fonction permet d'obtenir des informations sur un fichier ou un lien symbolique, mais contrairement à `stat()`, elle permet de récupérer les informations sur le lien symbolique lui-même, et non sur le fichier auquel il pointe. Elle prend en paramètre le chemin du fichier ou du lien symbolique pour lequel on souhaite obtenir les informations (pathname). Mais aussi la structure `*stat` qui recevra les informations sur le fichier ou le lien symbolique.
- **Retour:** Elle renvoie 0 en cas de succès et -1 en cas d'échecs errno est défini:
  - `ENOENT (2)` : Le fichier ou le lien symbolique n'existe pas.
  - `EACCESS (13)` : Permission refusée pour accéder au fichier ou au lien.
  - `EINVAL (22):` Argument invalide, généralement en cas de mauvais chemin de file.
  - `ENOMEM (12)`: Mémoire insuffisante pour allouer la structure `*stat`.
  - `ENAMETOOLONG (36)`: Le nom du fichier ou le chemin est trop long.

➔ **OPENDIR:** **`DIR *opendir(const char *name);`**

- `Librairies à inclure:`
  - **`#include <dirent.h>`**
- La fonction ouvre le répertoire dont le chemin est donné par l'argument `name`, qui est une chaîne de caractères représentant le chemin d'accès au répertoire (absolu ou relatif).
- **Retour:** Elle renvoie un pointeur sur une structure de type `*DIR` qui représente un répertoire ouvert. Cette structure est ensuite utilisée par des fonctions comme `readdir()` pour lire le contenu du répertoire. En cas d'échec, elle retourne `NULL`, errno est défini:
  - `EACCES (13)`: Le processus n'a pas les permissions nécessaires pour ouvrir le répertoire.
  - `ENOENT (2)`: Le répertoire spécifié n'existe pas.
  - `ENOTDIR (20)`: Le chemin spécifié ne correspond pas à un répertoire.
  - `ENOMEM (12)`: Mémoire insuffisante pour allouer une structure `*DIR`.

➔ **PERROR:** `void perror(const char *s);`

- Librairies à inclure:
  - `#include <errno.h>`
- La fonction est utilisée pour afficher un message d'erreur lié à une fonction système qui a échoué. Elle affiche une chaîne de caractères spécifiée par l'utilisateur suivie du message d'erreur correspondant à la valeur actuelle de la variable `errno`.

➔ **PIPE:** `int pipe(int pipefd[2]);`

- Librairies à inclure:
  - `#include <unistd.h>`
- La fonction est utilisée pour créer un canal de communication unidirectionnel entre deux processus. Cela permet à un processus d'écrire des données dans le tuyau, et à un autre de les lire.
  - **pipefd[0]** : le descripteur de fichier pour la lecture à partir du tuyau.
  - **pipefd[1]** : le descripteur de fichier pour l'écriture dans le tuyau.
- **Retour:** Elle renvoie 0 en cas de succès et en cas d'échec, elle retourne -1, errno est défini:
  - `ENFILE (23)`: Le système a atteint la limite du nombre total de descripteurs de fichiers ouverts.
  - `EMFILE (24)`: Le processus a atteint la limite maximale du nombre de descripteurs de fichiers ouverts.
  - `ENOMEM (12)`: Mémoire insuffisante pour allouer le pipe.

➔ **READDIR:** `struct dirent *readdir(DIR *dirp);`

- Librairies à inclure:
  - `#include <dirent.h>`
- La fonction est utilisée pour lire les entrées d'un répertoire, une à une. Elle permet de parcourir les fichiers et sous-répertoires dans un répertoire spécifié.
- Elle prend en paramètre:
  - Un pointeur vers un objet `DIR`, qui représente un répertoire ouvert. Ce pointeur est retourné par la fonction `opendir()`, qui ouvre le répertoire.
- **Retour:** Elle renvoie un pointeur vers une structure *`dirent` représentant l'entrée du répertoire en cas de succès et en cas d'échec, elle retourne `NULL`, errno est défini:
  - `EBADF (9)`: Si `dirp` n'est pas un descripteur de répertoire valide (par exemple, si le répertoire a été fermé).
  - `EINVAL (22):` Si `dirp` est un pointeur `NULL` ou non valide.

➔ **READLINE:** `char *readline(const char *prompt);`

- Attention, lors de la compilation il faut inclure le flag “**-lreadline**”.
- Librairies à inclure:
  - **#include <readline/readline.h>**
  - **#include <stdio.h>**
- Affiche l'invite de commande avec le prompt renseigne en paramètres.
- Elle attend que l'utilisateur saisisse une ligne de texte, puis appuie sur "Entrée". Pendant cette saisie, l'utilisateur peut utiliser les touches fléchées pour éditer la ligne, supprimer du texte, etc., car `readline` gère l'édition de ligne.
- La fonction retourne un pointeur vers une chaîne de caractères allouée dynamiquement. Il est important de libérer cette mémoire manuellement lorsque la chaîne n'est plus nécessaire.
- Les lignes saisies sont souvent stockées dans un historique grâce à la fonction `add_history()`, permettant à l'utilisateur d'accéder à des commandes précédentes en utilisant les flèches haut/bas.
- **Retour:** La fonction renvoie la ligne lue en cas de succès ou `NULL` en cas d’erreur ou de fin de fichiers.

➔ **RL_CLEAR_HISTORY:** **clear_history(void);**

- Attention, lors de la compilation il faut inclure le flag “**-lreadline**”.
- Librairies à inclure:
  - **#include <readline/readline.h>**
  - **#include <readline/history.h>**
- La fonction supprime l’historique de readline.
- **Retour:** La fonction renvoie 0 en cas de succès ou -1 en cas d’erreur.

➔ **RL_ON_NEW_LINE:** **void rl_on_new_line(void);**

- Attention, lors de la compilation il faut inclure le flag “**-lreadline**”.
- Librairies à inclure:
  - **#include <readline/readline.h>**
- La fonction est utilisée pour gérer les entrées de l'utilisateur en ligne de commande. Cette fonction est utilisée pour signaler à Readline qu'une nouvelle ligne de texte a été lue.
- **Retour:** La fonction ne renvoie rien.

➔ **RL_REDISPLAY:** **void rl_redisplay(void);**

- Attention, lors de la compilation il faut inclure le flag “**-lreadline**”.
- Librairies à inclure:
  - **#include <readline/readline.h>**

- La fonction permet à l'utilisateur d'éditer la ligne de commande tout en affichant correctement l'historique et les caractères déjà saisis.
- Elle se base sur l'état actuel du buffer de ligne (la ligne que l'utilisateur est en train de taper).

- Si le curseur est déplacé ou si le contenu de la ligne change, la fonction peut être appelée pour réactualiser l'affichage dans le terminal.

- Elle peut aussi être utilisée pour réinitialiser l'affichage en cas de problème graphique, par exemple, après que l'affichage ait été corrompu par un autre processus ou par un traitement de terminal.
- **Retour:** La fonction ne renvoie rien.

➔ **RL_REPLACE_LINE:** `int rl_replace_line(const char *text, int clear_undo);`

- `Attention, lors de la compilation il faut inclure le flag "-lreadline".`
- `Librairies à inclure:`
  - `#include <readline/readline.h>`
- La fonction remplace la ligne actuelle en mémoire par la chaîne `text`. Elle en paramètre:
  - **text** : une chaîne de caractères qui va remplacer la ligne actuelle dans le buffer.

  - **clear_undo** : un entier qui détermine si l'undo doit être effacé (`1` pour effacer, `0` pour garder les données d'undo).
  -
- **Retour:** La fonction renvoie 0 en cas de succès et -1 en cas d'échecs, errno est défini:
  - `EINVAL (22)`: L'argument est invalide.
  - `ENOMEM (12)`: Erreur mémoire.

➔ **SIGACTION:** `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`

- `Librairies à inclure:`
  - `#include <signal.h>`
- La fonction est utilisée pour définir un gestionnaire de signal pour un signal donné. Elle en paramètre:
  - **signum** : Le numéro du signal que vous voulez intercepter (par exemple, `SIGINT`, `SIGTERM`).

  - **act** : Un pointeur vers une structure *`sigaction` qui définit le nouveau gestionnaire de signal et d'autres options de gestion du signal.

- **oldact** : Un pointeur vers une structure *`sigaction` où l'ancien gestionnaire sera stocké. Ce paramètre peut être `NULL` si vous ne souhaitez pas sauvegarder l'ancien gestionnaire.

◆ **Retour:** La fonction sigaction renvoie 0 si elle réussit et -1 en cas d'erreur. Si une erreur se produit, errno est défini

- `EINVAL (22)`: Le signal spécifié est invalide.
- `ENOMEM (12)`: Erreur mémoire.
- `EFAULT (14)`: Un problème d'accès mémoire (par exemple, des pointeurs invalides dans `act` ou `oldact`).

➔ **SIGADDSET:** **`int sigaddset(sigset_t *set, int signum);`**

◆ `Librairies à inclure:`

- **`#include <signal.h>`**

◆ La fonction permet d'ajouter un signal à un ensemble de signaux. Elle en paramètre:

- **sigset_t *set** : Un pointeur vers un ensemble de signaux. Un ensemble de signaux est une structure de données utilisée pour manipuler plusieurs signaux de manière atomique.

- **signum** : Le signal que vous souhaitez ajouter à l'ensemble. Cela peut être un signal comme `SIGINT` (interruption) ou `SIGTERM` (arrêt), etc…

◆ **Retour:** La fonction sigaddset renvoie 0 si elle réussit et -1 en cas d'erreur. Si une erreur se produit, errno est défini

- `EINVAL (22)`: L'argument `signum` n'est pas un signal valide ou le signal n'est pas pris en charge par le système.
- `EFAULT (14)`: Si l'argument `set` n'est pas un pointeur valide (pointeur vers un emplacement mémoire inaccessible).

➔ **SIGEMPTYSET:** **`int sigemptyset(sigset_t *set);`**

◆ `Librairies à inclure:`

- **`#include <signal.h>`**

◆ La fonction initialise l'ensemble de signaux de la structure pointés par `set` et le vide de tous les signaux.

◆ Elle est souvent utilisée avant d'ajouter des signaux à un ensemble avec `sigaddset`.

◆ **Retour:** La fonction sigemptyset renvoie 0 si elle réussit et -1 en cas d'erreur. Si une erreur se produit, errno est défini (cas très rare).

- `EFAULT (14)`: Si l'argument `set` est NULL.

➔ **SIGNAL:** **`void (*signal(int signum, void (*handler)(int)))(int);`**

◆ `Librairies à inclure:`

- **`#include <signal.h>`**

- La fonction permet de définir un gestionnaire de signal pour un signal spécifique.
- Elle prend en paramètres.
  - `signum` : Le numéro du signal à capturer (ex. `SIGINT`, `SIGTERM`).
  - `handler` : Un pointeur vers la fonction de gestion du signal ou une valeur spéciale :
    - `SIG_IGN` : Ignore le signal.
    - `SIG_DFL` : Rétablit le comportement par défaut du signal.
    - Une fonction utilisateur prenant un `int` en paramètre (le numéro du signal).
- **Retour:** La fonction signal renvoie l'ancienne fonction de gestion du signal, en cas d'échec, elle retourne `SIG_ERR`, errno est défini:
  - `EINVAL (22)`: Si le signal donné est vide ou si l'on tente de redéfinir un signal qui ne peut pas être capturé comme (`SIGKILL ou SIGSTOP`).

➔ **STAT:** `int  stat(const char *pathname, struct stat *statbuf);`

- `Librairies à inclure:`
  - `#include <sys/types.h>`
  - `#include <sys/stat.h>`
  - `#include <unistd.h>`
- La fonction récupère les informations d'un fichier désigné par `pathname` et les stocke dans la structure *`stat`.
  - `signum` : Le numéro du signal à capturer (ex. `SIGINT`, `SIGTERM`).
  - `handler` : Un pointeur vers la fonction de gestion du signal ou une valeur spéciale :
    - `SIG_IGN` : Ignore le signal.
    - `SIG_DFL` : Rétablit le comportement par défaut du signal.
    - Une fonction utilisateur prenant un `int` en paramètre (le numéro du signal).
- **Retour:** La fonction stat renvoie 0 et -1 en cas d'échecs, errno est défini:
  - `ENOENT (2)`: Fichier inexistant.
  - `EACCESS (13)` **:** Permission refusée pour accéder au fichier.
  - `ENOTDIR (20)`: Le chemin spécifié ne correspond pas à un répertoire.
  - `ENAMETOOLONG (36)`: Le nom du fichier ou le chemin est trop long.

➔ **STRERROR:** `char *strerror(int errnum);`

- `Librairies à inclure:`
  - `#include <errno.h>`
- La fonction permet de retourner une chaîne de caractères décrivant le message d'erreur correspondant à une valeur donnée d'`errno`.

- **Retour:** La fonction renvoie la chaîne de caractères descriptive du code `errnum` renseigne en paramètres.

➔ **TCGETATTR:** **`int  tcgetattr(int fd, struct termios *termios_p);`**

- `Librairies à inclure:`
  - **`#include <termios.h>`**
- La fonction permet d'obtenir les attributs d'un terminal, c'est-à-dire ses paramètres de contrôle comme la vitesse de transmission, le mode d'affichage, ou encore les caractères spéciaux utilisés par le terminal.
- Elle prend en paramètres:
  - `fd` : C'est un descripteur de fichier associé au terminal. Habituellement, cela peut être un descripteur comme `STDIN_FILENO` (0), `STDOUT_FILENO` (1), ou `STDERR_FILENO` (2).
  - `termios_p` : Pointeur vers une structure *`termios` dans laquelle les attributs actuels du terminal seront stockés.
- **Retour:** La fonction strerror renvoie 0 et -1 en cas d'échecs, errno est défini:
  - `EBADF (9)`**:** Le descripteur de fichier `fd` est invalide ou n'est pas ouvert.
  - `ENOTTY (25)`**:** Le `fd` renseigne ne correspond pas à un terminal.

➔ **TCSETATTR:** **`int  tcsetattr(int fd, int optional_actions, const struct termios *termios_p);`**

- `Librairies à inclure:`
  - **`#include <termios.h>`**
- La fonction est utilisée pour configurer les paramètres d'un terminal sous Linux ou un système Unix. Elle modifie les attributs de la session de terminal.
- Elle prend en paramètres:
  - `fd` : C'est un descripteur de fichier du terminal à configurer. Habituellement, cela peut être un descripteur comme `STDIN_FILENO` (0), `STDOUT_FILENO` (1), ou `STDERR_FILENO` (2) ou d'un `fd` ouvert avec `open()`.
  - `optional_actions` : Ce paramètre spécifie quand les changements doivent être appliqués. Il peut avoir trois valeurs possibles :
    - `TCSANOW` : Applique immédiatement les changements.
    - `TCSADRAIN` : Applique les changements après que toutes les sorties de données aient été envoyées.
    - `TCSAFLUSH` : Applique les changements après avoir vidé les tampons de lecture et d'écriture.
  - `termios_p` : Pointeur vers une structure *`termios` qui contient les nouveaux paramètres du terminal.
- **Retour:** La fonction tcsetattr renvoie 0 et -1 en cas d'échecs, errno est défini:
  - `EBADF (9)`**:** Le descripteur de fichier `fd` est invalide ou n'est pas ouvert.
  - `ENOTTY (25)`**:** Le `fd` renseigne ne correspond pas à un terminal.

- EINVAL (12): L'argument optional_actions est invalide.

➔ **TGETENT:** **int tgetent(char *bp, const char *name);**

- Attention, lors de la compilation il faut inclure le flag "**-lncurses**" après le nom de l'exécutable (Ex: cc $(FLAG) $(OBJET) -o $(NAME) -lncurses).
- Librairies à inclure:
  - **#include <termcap.h>**
- La fonction permet d'accéder aux informations sur les capacités des terminaux.
- Elle prend en paramètres:
  - bp : Pointeur vers une zone mémoire où seront stockées les informations sur le terminal. Si vous passez NULL pour bp, la fonction utilise une zone de mémoire statique interne.
  - name : Le nom du terminal (souvent obtenu via la variable d'environnement TERM).
- **Retour:** La fonction tgetent renvoie 1 si elle a réussi a charger la description du terminal, 0 si aucune entrée n'a été trouvé et -1 en cas d'échecs, errno est défini:
  - ENOENT (2) : Le fichier termcap n'existe pas.
  - EACCES (13) : Le fichier termcap existe mais vous n'avez pas les permissions pour y accéder.
  - ENOMEM (22): Une allocation mémoire échoue.

➔ **TGETFLAG:** **int tgetflag(char *id);**

- Attention, lors de la compilation il faut inclure le flag "**-lncurses**" après le nom de l'exécutable (Ex: cc $(FLAG) $(OBJET) -o $(NAME) -lncurses).
- Librairies à inclure:
  - **#include <termcap.h>**
- La fonction est utilisée pour interroger une capacité booléenne d'un terminal. Elle prend en argument une chaîne de caractères id, représentant l'identifiant de la capacité, et vérifie si cette capacité est présente dans la base de données des terminaux chargée avec tgetent.
- **Retour:** La fonction tgetflag renvoie 1 si la capacité est présente et 0 si elle ne l'est pas.

➔ **TGETNUM:** **int tgetnum(char *id);**

- Attention, lors de la compilation il faut inclure le flag "**-lncurses**" après le nom de l'exécutable (Ex: cc $(FLAG) $(OBJET) -o $(NAME) -lncurses).
- Librairies à inclure:
  - **#include <termcap.h>**

- La fonction interroge une **capacité numérique** du terminal spécifiée par `id`, une chaîne de deux caractères représentant l'identifiant de la capacité. Ces capacités numériques sont des propriétés comme le nombre de lignes (`li`), le nombre de colonnes (`co`)...
- **Retour:** La fonction tgetnum renvoie la valeur numérique de la capacité trouvée. Si la capacité n'est pas définie elle renvoie -1.

➔ **TGETSTR:** `char *tgetstr(char *id, char **area);`

- `Attention, lors de la compilation il faut inclure le flag "-lncurses" après le nom de l'exécutable (Ex: cc $(FLAG) $(OBJET) -o $(NAME) -lncurses).`
- `Librairies à inclure:`
  - `#include <termcap.h>`
- La fonction est utilisée pour récupérer une **séquence de contrôle** ou une **capacité de chaîne** d'un terminal. Elle permet d'obtenir des commandes spécifiques au terminal, telles que les séquences d'échappement qui déclenchent des actions comme le déplacement du curseur, l'effacement de l'écran, ou l'activation de modes particuliers.
- Elle prend en paramètres:
  - `id :` Il s'agit de la chaîne de deux caractères représentant l'identifiant de la capacité. Par exemple, `"cl"` pour la capacité "clear screen" (effacer l'écran).
  - `area :` C'est un pointeur qui pointe vers une zone mémoire où la séquence de contrôle sera stockée. Cette zone doit être suffisamment grande pour contenir la chaîne retournée. La variable `area` est mise à jour après chaque appel pour pointer vers la fin de la chaîne copiée, afin d'éviter d'écraser les données existantes.
- **Retour:** La fonction tgetstr renvoie un pointeur sur cette chaîne si la capacité de la chaîne est trouvée. Si elle n'est pas définie, elle renvoie `NULL`.

➔ **TGOTO:** `char *tgoto(const char *cap, int col, int row);`

- `Attention, lors de la compilation il faut inclure le flag "-lncurses" après le nom de l'exécutable (Ex: cc $(FLAG) $(OBJET) -o $(NAME) -lncurses).`
- `Librairies à inclure:`
  - `#include <termcap.h>`
- La fonction génère une séquence de contrôle qui déplace le curseur à une position donnée sur l'écran du terminal, selon les capacités du terminal. Elle utilise un modèle de chaîne de caractères fourni par une capacité (comme `"cm"` pour le déplacement du curseur) et remplace les paramètres variables par les valeurs spécifiées (la colonne et la ligne).
- Elle prend en paramètres:
  - `cap :` Une chaîne de caractères représentant une capacité du terminal (par exemple, `"cm"` pour le déplacement du curseur).

- `col :` La colonne où placer le curseur.
- `row :` La ligne ou placer le curseur.

◆ **Retour:** La fonction tgoto renvoie un pointeur vers la séquence de contrôle générée. Si une erreur survient, elle renvoie `NULL`.

➔ TPUTS: **`int tputs(const char *str, int affcnt, int (*putc)(int));`**

◆ `Attention, lors de la compilation il faut inclure le flag “-lncurses” après le nom de l'exécutable (Ex: cc $(FLAG) $(OBJET) -o $(NAME) -lncurses).`

◆ `Librairies à inclure:`

- **`#include <term.h>`**

◆ La fonction sert à envoyer une séquence de contrôle au terminal. Elle prend en charge les séquences qui dépendent du nombre de lignes affectées par l'opération (comme le défilement ou le nettoyage d'une partie de l'écran). Elle gère également les pauses nécessaires entre certains caractères pour éviter d'envoyer des séquences trop rapidement au terminal, ce qui peut être utile sur des terminaux lents.

◆ Elle prend en paramètres:

- `str :` Chaîne de caractères contenant la séquence de contrôle à envoyer au terminal (souvent obtenue via `tgetstr` ou `tgoto`).
- `affcnt :` Nombre de lignes affectées par l'opération. Pour des opérations comme le défilement, ce paramètre indique combien de lignes sont déplacées. Pour d'autres séquences qui n'affectent pas directement plusieurs lignes, il peut être mis à 1.
- `row :` Un pointeur vers une fonction qui sera utilisée pour envoyer chaque caractère au terminal. Cette fonction doit accepter un caractère sous forme d'entier et le transmettre à la sortie.

◆ **Retour:** La fonction tputs renvoie 0 en cas de succès et -1 en cas d'échecs.

➔ **TTYNAME:** **`char *ttyname(int fd);`**

◆ `Librairies à inclure:`

- **`#include <unistd.h>`**

◆ La fonction retourne le chemin d'accès au fichier du terminal associé au descripteur de fichier `fd`.

◆ **Retour:** La fonction ttyname retourne un **pointeur vers une chaîne de caractères** contenant le chemin absolu du terminal associé au descripteur de fichier donné. En cas d'échecs, elle renvoie NULL, errno est défini:

- `EBADF (9)` : Le `fd` donné n'est pas valide, il n’est pas ouvert ou ne correspond pas à un terminal.
- `ENOTTY (25)` : Mémoire insuffisante pour allouer les ressources nécessaires.
- `ENOTTY (25)` : Le `fd` est valide mais ne correspond pas à un terminal (par exemple, un fichier ou un socket).

➔ **TTYSLOT:** `int ttyslot(void);`

- Librairies à inclure:
  - `#include <unistd.h>`
- **Retour:** La fonction ttyslot retourne l'index du terminal courant dans le fichier `/var/run/utmp ou /etc/utmp` (qui contient les informations sur les utilisateurs connectés). Elle est souvent utilisée pour identifier le terminal utilisé par un processus. La fonction renvoie 0 si le terminal n'a pas pu être trouvé dans le fichier `utmp` et -1 si une erreur est survenue, errno est défini:
  - `ENOENT (2)` : Le fichier `utmp` n'a pas pu être trouvé. Le fichier `/var/run/utmp` n'existe pas sur le système.
  - `EBADF (9)` : Le `fd` donné n'est pas valide, il n'est pas ouvert ou ne correspond pas à un terminal.
  - `ENOTTY (25)` : Le terminal courant n'a pas pu être trouvé dans le fichier `utmp`.
  - `ENOMEM (12)` : Il n'y a pas assez de mémoire pour traiter la requête

➔ **UNLINK:** `int unlink(const char *pathname);`

- Librairies à inclure:
  - `#include <unistd.h>`
- La fonction supprime un **lien (link)** vers un fichier, ce qui peut être un fichier régulier, un fichier spécial ou un lien symbolique. Lorsque le dernier lien vers un fichier est supprimé, et qu'aucun processus ne l'a ouvert, le fichier est effectivement supprimé du système de fichiers.
- **Retour:** La fonction unlink renvoie 0 en cas de succès et -1 en cas d'échecs, errno est défini:
  - `EPERM (1)` : Opération non permise.
  - `ENOENT (2)` : Aucun fichier ou répertoire correspondant trouvé. Le chemin donné dans `pathname` n'existe pas.
  - `EACCES (13)` : Permission refusée. L'utilisateur n'a pas les droits d'écriture sur le répertoire contenant le fichier ou les droits de suppression du fichier.
  - `EBUSY (16)` : Le fichier est actuellement utilisé ou monté dans un autre contexte, donc il ne peut pas être supprimé.
  - `ENOTDIR (20)` : Une composante du chemin est censée être un répertoire mais ne l'est pas.
  - `EISDIR (21)` : Tentative de supprimer un répertoire. `unlink` ne fonctionne que sur des fichiers réguliers et des liens symboliques, pas sur des répertoires.
  - `EINVAL (22)` : Argument invalide, par exemple si un lien non valide est passé à `unlink`.
  - `EROFS (30)` : Le fichier se trouve sur un système de fichiers monté en lecture seule, donc il est impossible de le modifier ou de le supprimer.
  - `ENAMETOOLONG (36)` : Le nom de fichier ou chemin est trop long.

- `ELOOP (40)` : Trop de redirections de liens symboliques lors de la résolution du chemin.

➔ **WAIT:** **`pid_t wait(int *wstatus);`**

- `Librairies à inclure:`
  - **`#include <sys/wait.h>`**
  - **`#include <unistd.h>`**
- La fonction suspend l'exécution du processus appelant jusqu'à ce qu'un de ses processus enfants se termine ou change d'état (arrêté, terminé, …).
- Elle prend en paramètre:
  - `wstatus` : Un pointeur vers un entier où sera stocké le statut de terminaison du processus enfant. Si ce pointeur est **`NULL`**, l'information sur le statut est ignorée.
    - Elle permet de récupérer le **statut** de terminaison d'un processus enfant grâce à des macros comme `WIFEXITED(wstatus), WEXITSTATUS(wstatus)`, ...
- **Retour:** La fonction wait renvoie le **PID** (process ID) du processus enfant qui s'est terminé ou a changé d'état en cas de succès et -1 en cas d'échecs, errno est défini:
  - `EINTR (4)` : L'appel système a été interrompu par un signal avant que l'état d'un processus enfant ne soit modifié.
  - `ECHILD (10)` : Aucun processus enfant n'existe pour le processus appelant. Cela se produit si le processus appelant n'a pas de processus enfant.
  - `EINVAL (22)` : Le statut fourni n'est pas valide.

➔ **WAIT3:** **`pid_t wait3(int *wstatus, int options, struct rusage *rusage);`**

- `Librairies à inclure:`
  - **`#include <sys/wait.h>`**
  - **`#include <sys/resource.h>`**
  - **`#include <unistd.h>`**
- La fonction attend la terminaison d'un processus enfant et renvoie des informations sur son statut de terminaison (comme la fonction `wait`), mais elle fournit également des informations sur l'utilisation des ressources du processus enfant grâce à une structure *`rusage`.
- Elle prend en paramètre:
  - `wstatus` : Un pointeur vers un entier où sera stocké le statut de terminaison du processus enfant. Si ce pointeur est `NULL`, l'information sur le statut est ignorée.
    - Elle permet de récupérer le **statut** de terminaison d'un processus enfant grâce à des macros comme `WIFEXITED(wstatus), WEXITSTATUS(wstatus)`, ...

- `options` : Une combinaison d'options qui modifient le comportement de la fonction. Par exemple :
  - `WNOHANG` : Ne pas attendre si aucun enfant n'est encore terminé, retourne immédiatement.
  - `WUNTRACED` : Retourne si un enfant a été arrêté mais pas encore terminé.
- `*rusage` : Un pointeur vers une structure `*rusage` dans laquelle seront stockées les informations sur l'utilisation des ressources du processus enfant (temps processeur, mémoire, etc.). Si ce pointeur est `NULL`, ces informations ne sont pas collectées.

◆ **Retour:** La fonction wait3 renvoie le **PID** (process ID) du processus enfant qui s'est terminé ou a changé d'état en cas de succès et -1 en cas d'échecs, errno est défini:

- `EINTR (4)` **:** L'appel système a été interrompu par un signal avant que l'état d'un processus enfant ne soit modifié.
- `ECHILD (10)` **:** Aucun processus enfant n'existe pour le processus appelant.
- `EINVAL (22)` **:** Argument(s) invalide(s). Par exemple, si une mauvaise combinaison d'options ou un mauvais pointeur.

➔ **WAIT4:** **`pid_t wait4(pid_t pid, int *wstatus, int options, struct rusage *rusage);`**

◆ `Librairies à inclure:`

- **`#include <sys/wait.h>`**
- **`#include <sys/resource.h>`**
- **`#include <unistd.h>`**

◆ La fonction attend qu'un processus enfant se termine ou change d'état, comme `waitpid`, tout en offrant la possibilité de récupérer des informations sur l'utilisation des ressources du processus enfant via une structure `*rusage`. Elle est similaire à `wait3`, mais elle permet de spécifier un **PID** précis à surveiller.

◆ Elle prend en paramètre:

- `pid` :
  - Si `pid > 0` : La fonction attend le processus avec le **PID** spécifié.
  - Si `pid == 0` : La fonction attend un processus enfant du même groupe de processus que le processus appelant.
  - Si `pid == -1` : La fonction attend n'importe quel processus enfant.
  - Si `pid < -1` : La fonction attend n'importe quel processus enfant du groupe de processus avec un **PID** égal à la valeur absolue de `pid`.

- `wstatus` : Un pointeur vers un entier où sera stocké le statut de terminaison du processus enfant. Si ce pointeur est `NULL`, l'information sur le statut est ignorée.
    - Elle permet de récupérer le **statut** de terminaison d'un processus enfant grâce à des macros comme `WIFEXITED(wstatus), WEXITSTATUS(wstatus)`, ...
- `options` : Une combinaison d'options qui modifient le comportement de la fonction. Par exemple :
    - `WNOHANG` : Ne pas attendre si aucun enfant n'est encore terminé, retourne immédiatement.
    - `WUNTRACED` : Retourne si un enfant a été arrêté mais pas encore terminé.
- *`rusage` : Un pointeur vers une structure *`rusage` dans laquelle seront stockées les informations sur l'utilisation des ressources du processus enfant (temps processeur, mémoire, etc…). Si ce pointeur est `NULL`, ces informations ne sont pas collectées.

◆ **Retour:** Si un processus enfant se termine ou change d'état, la fonction renvoie le **PID** de ce processus. Elle renvoie -1 en cas d'échecs, errno est défini:

- `EINTR (4)` **:** L'appel système a été interrompu par un signal avant que l'état d'un processus enfant ne soit modifié.
- `ECHILD (10)` **:** Aucun processus enfant n'existe pour le processus appelant ou aucun processus enfant correspondant au critère de `pid`.
- `EINVAL (22)` **:** Argument(s) invalide(s). Par exemple, si une mauvaise combinaison d'options ou un mauvais pointeur.

➔ **WAITPID:** **`pid_t waitpid(pid_t pid, int *wstatus, int options);`**

◆ `Librairies à inclure:`

- **`#include <sys/wait.h>`**
- **`#include <sys/types.h>`**
- **`#include <unistd.h>`**

◆ La fonction permet à un processus parent d'attendre la terminaison ou un changement d'état d'un processus enfant. Contrairement à **`wait`**, elle permet de spécifier un processus enfant particulier en utilisant son `pid` et offre des options pour contrôler son comportement.

◆ Elle prend en paramètre:

- `pid` :
    - Si `pid > 0` : La fonction attend le processus avec le `pid` spécifié.
    - Si `pid == 0` : La fonction attend un processus enfant du même groupe de processus que le processus appelant.
    - Si `pid == -1` : La fonction attend n'importe quel processus enfant.

        - Si `pid < -1` : Attendre tout processus enfant appartenant au groupe de processus dont l'ID est égal à `-pid`.
    - `wstatus` : Un pointeur vers un entier où sera stocké le statut de terminaison du processus enfant. Si ce pointeur est `NULL`, l'information sur le statut est ignorée.
        - Elle permet de récupérer le **statut** de terminaison d'un processus enfant grâce à des macros comme `WIFEXITED(wstatus), WEXITSTATUS(wstatus)`, ...
    - `options` : Une combinaison d'options qui modifient le comportement de la fonction. Par exemple :
        - `WNOHANG` : Ne pas attendre si aucun processus enfant n'est encore terminé, retourne immédiatement.
        - `WUNTRACED` : Retourne également si un processus enfant a été arrêté, mais pas encore terminé.
- **Retour:** La fonction renvoie le **PID** du processus enfant qui a changé d'état. Elle renvoie -1 en cas d'échecs, errno est défini:
    - `EINTR (4)` : L'appel système a été interrompu par un signal avant que l'état d'un processus enfant ne soit modifié.
    - `ECHILD (10)` : Aucun processus enfant correspondant au `pid` spécifié n'existe.
    - `EINVAL (22)` : `options` spécifie une valeur invalide.
    - `ENOMEM (12)` : L'implémentation ne dispose pas des ressources suffisantes pour effectuer l'appel.

➔ **WRITE:** **`ssize_t write(int, fd, const void *buf, size_t count);`**

- `Librairies à inclure:`
    - **`#include <unistd.h>`**
- La fonction est utilisée pour écrire des données depuis un buffer en mémoire vers un fichier ou un autre descripteur de fichier
- **Retour:** La fonction renvoie le nombre d'octets écrit dans le `fd` renseigne.Elle renvoie -1 en cas d'échecs, errno est défini:
    - `EINTR (4)` : L'appel a été interrompu par un signal avant que des données ne soient écrites.
    - `EIO (5)` : Une erreur d'entrée/sortie s'est produite.
    - `EBADF (9)` : Le descripteur de fichier `fd` est invalide ou n'est pas ouvert en écriture.
    - `EAGAIN (11)` : Le descripteur de fichier a été ouvert en mode non-bloquant, et l'écriture bloquerait.
    - `ENOMEM (12)` : Mémoire insuffisante.
    - `EFAULT (14)` : Le pointeur `buf` pointe vers un espace mémoire invalide.
    - `EINVAL (22)` : Paramètre invalide, comme une taille négative de `count`.

- `EFBIG (27)` : La taille du fichier a dépassé la limite maximale autorisée.
- `ENOSPC (28)` : Il n'y a plus d'espace sur le périphérique ou dans le système de fichiers pour écrire les données.
- `EPIPE (32)` : Tentative d'écrire dans un pipe ou un socket pour lequel le lecteur a été fermé.

LEXIQUE :

Signal	Valeur décimale	Description
SIGHUP	1	Terminal fermé
SIGINT	2	Interruption (Ctrl + C)
SIGQUIT	3	Fin de processus + core dump (Ctrl + \)
SIGILL	4	Instruction illégale
SIGABRT	6	Abandon (abort)
SIGFPE	8	Erreur arithmétique (division par zéro, etc…)
SIGKILL	9	Fin de processus forcée (non interceptable)
SIGSEGV	11	Erreur de segmentation
SIGPIPE	13	Ecriture sur un pipe sans lecteur
SIGALRM	14	Expiration de la fonction alarm* ?
SIGTERM	15	Terminaison normale
SIGUSR1	10	Signal utilisateur 1
SIGUSR2	12	Signal utilisateur 2

***alarm :** C'est une fonction de la librairie `unistd.h` prototype comme suit :

```
unsigned int     alarm(unsigned int seconds);
```

- Cette fonction programme un signal `SIGALRM` après `seconds` secondes.
- Il ne peut y avoir **qu'une seule alarme active** par processus.
- **Retour:** La fonction `alarm()` renvoie **nombre de secondes restantes** si un `alarm()` était déjà programmée, sinon `0`. Si un `alarm()` est appelé avec `seconds = 0`, cela annule toute `alarm()` en attente.

Structure *DIR :

```
struct DIR
{
    int         fd;         // Descripteur de fichier pour le répertoire
    long        loc;        // Position actuelle dans le répertoire
    long        size;       // Taille du buffer
    char        *buf;       // Buffer pour les entrées du répertoire
    struct dirent *next;    // Prochaine entrée dans le répertoire
};
```

Structure *dirent :

```
struct dirent
{
    ino_t           d_ino;      // Numéro d'inode (identifiant de fichier)
    off_t           d_off;      // Décalage jusqu'à la prochaine entrée
    unsigned short  d_reclen;   // Longueur de l'entrée
    unsigned char   d_type;     // Type de fichier (fichier, répertoire, lien symbolique, ...)
    char            d_name[256]; // Nom de fichier
};
```

Structure *dirent :

```
struct sigaction
{
    void        (*sa_handler)(int);                 // Pointeur vers la fonction de gestion du signal
    void        (*sa_sigaction)(int, siginfo_t *, void *);  // Handler pour signaux avec informations supplémentaires
    sigset_t    sa_mask;                            // Ensemble de signaux à bloquer pendant le handler
    int         sa_flags;                           // Options de comportement pour la gestion du signal
    void        (*sa_restorer)(void);               // Utilisé uniquement pour compatibilité dans certaines architectures
};
```

Structure *stat :

```
struct stat
{
    dev_t       st_dev;     // ID du périphérique contenant le fichier
    ino_t       st_ino;     // Numéro d'inœud
    mode_t      st_mode;    // Permissions et type de fichier
    nlink_t     st_nlink;   // Nombre de liens physiques
    uid_t       st_uid;     // Identifiant de l'utilisateur propriétaire
    gid_t       st_gid;     // Identifiant du groupe propriétaire
    dev_t       st_rdev;    // ID du périphérique (si fichier spécial)
```

```
    off_t       st_size;    // Taille en octets
    blksize_t   st_blksize; // Taille de bloc d'E/S préférée
    blkcnt_t    st_blocks;  // Nombre de blocs alloués
    time_t      st_atime;   // Dernier accès
    time_t      st_mtime;   // Dernière modification
    time_t      st_ctime;   // Dernier changement d'état
};
```

Structure *termios :

```
struct termios
{
    tcflag_t    c_iflag;    // Modes d'entrée
    tcflag_t    c_oflag;    // Modes de sortie
    tcflag_t    c_cflag;    // Modes de contrôle
    tcflag_t    c_lflag;    // Modes locaux
    cc_t        c_cc[NCCS]; // Caractères de contrôle
    speed_t     c_ispeed;   // Vitesse d'entrée
    speed_t     c_ospeed;   // Vitesse de sortie
};
```

Structure *rusage :

```
struct rusage
{
    struct timeval ru_utime;    // Temps d'utilisation CPU en mode utilisateur
    struct timeval ru_stime;    // Temps d'utilisation CPU en mode noyau
    long        ru_maxrss;      // Mémoire maximale résidente
    long        ru_ixrss;       // Mémoire partagée
    long        ru_idrss;       // Mémoire privée
    long        ru_isrss;       // Mémoire de la pile
    long        ru_minflt;      // Nombre de défauts de pages mineurs
    long        ru_majflt;      // Nombre de défauts de pages majeurs
    long        ru_nswap;       // Nombre d'opérations de swap
    long        ru_inblock;     // Nombre de lectures du système de fichiers
    long        ru_oublock;     // Nombre d'écritures sur le système de fichiers
    long        ru_msgsnd;      // Nombre de messages envoyés
    long        ru_msgrcv;      // Nombre de messages reçus
    long        ru_nsignals;    // Nombre de signaux reçus
    long        ru_nvcsw;       // Nombre de changements de contexte volontaires
    long        ru_nivcsw;      // Nombre de changements de contexte involontaires
};
```