Un examen détaillé des utilitaires du projet minishell – réponses chatgpt condensées

Intégré: Exécuté par le shell lui-même.
Externe: Nécessite que le shell lance un nouveau processus pour exécuter le programme.

Intégré: GETENV

Syntaxe:

```
#include <stdlib.h>

char *getenv(const char *nom);
```

`gettenv` est une fonction qui récupère la valeur d'une variable d'environnement. Les variables d'environnement sont des paires clé-valeur qui fournissent des informations sur l'environnement du système d'exploitation dans lequel un programme s'exécute. Par exemple, les variables d'environnement peuvent stocker des informations telles que le répertoire personnel de l'utilisateur actuel, le chemin du système ou les paramètres de configuration spécifiques d'une application.

Exemple de cas d'utilisation :

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

int main() {
	// Récupère la valeur de la variable d'environnement HOME
	char *home = getenv("HOME");

	if (home != NULL) {
	printf("Répertoire personnel : %s\n", home);
	} autre {
	printf("La variable d'environnement HOME n'est pas définie.\n");
	}

	renvoie 0 ;
}
```

Points clés :

- **Variables d'environnement**: Il s'agit de paramètres à l'échelle du système qui peuvent affecter le comportement des processus. Les exemples incluent :
    - `CHEMIN`: Les répertoires dans lesquels le shell recherche les programmes exécutables.
    - `MAISON`: Le répertoire personnel de l'utilisateur actuel.
    - `UTILISATEUR`: Le nom d'utilisateur de l'utilisateur actuellement connecté.
- **Vérification nulle**: Vérifiez toujours si `gettenv` retours `NUL`, car la variable d'environnement peut ne pas être définie.
- `gettenv` récupère la valeur d'une variable d'environnement par son nom.

Intégré: FSTAT

Syntaxe:

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

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

Cas d'utilisation :

- `état rapide` est utilisé lorsque vous avez déjà un fichier ouvert via un descripteur de fichier (à partir de fonctions comme `ouvrir()`, `tuyau()`, ou `douille()`), et vous souhaitez récupérer des métadonnées sur le fichier.
- Contrairement à `statistique()`, qui prend un chemin de fichier en entrée, `fstat()` fonctionne avec un descripteur de fichier ouvert, ce qui le rend utile pour travailler avec des fichiers dont les chemins pourraient ne plus être accessibles (par exemple, si le fichier n'est pas lié).

```
structure statistique {
      dev_t  st_dev;      // ID du périphérique contenant le fichier
      ino_t  st_ino;      // Numéro d'inode
      mode_t st_mode;     // Type et mode de fichier (autorisations)
      nlink_t st_nlink;   // Nombre de liens physiques
      uid_t  st_uid;      // ID utilisateur du propriétaire
      gid_t  st_gid;      // ID de groupe du propriétaire
      dev_t  st_rdev;     // ID de périphérique (si fichier spécial)
      off_t  st_size;     // Taille totale, en octets
      blksize_t st_blksize; // Taille du bloc pour les E/S du système de fichiers
      blkcnt_t st_blocks ;  // Nombre de blocs 512B alloués
      heure_t      st_atime;   // Heure du dernier accès
      heure_t      st_mtime;   // Heure de la dernière modification
      heure_t      st_ctime;   // Heure du dernier changement de statut
} ;
```

Intégré: GETCWD

Syntaxe:

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

getcwd représente **obtenir le répertoire de travail actuel** et est une fonction en C qui récupère le chemin absolu du répertoire de travail actuel.

Points clés :

- Si le tampon (buf) est trop petit pour contenir le chemin, getcwd() échouera avec l'erreur ERANGE.
- Si buf est NULL, getcwd() allouera dynamiquement un tampon suffisamment grand pour contenir le chemin. L'appelant est responsable de libérer la mémoire allouée dans ce cas.
- size spécifie la taille du tampon lorsqu'il est alloué manuellement.

Exemple de code :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
	char *cwd = getcwd(NULL, 0);  // Laisse getcwd allouer le tampon

	si (cwd != NULL) {
	printf("Répertoire de travail actuel : %s\n", cwd);
	gratuit (cwd);  // Libère le buffer alloué
	} autre {
	perror("getcwd");
	}

	renvoie 0 ;
}
```

Intégré: ISATTY

Syntaxe:

```
#include <unistd.h>

int isatty(int fd);
```

Cas d'utilisation :

- `isatty` est souvent utilisé dans les programmes pour détecter si leur entrée ou leur sortie est connectée à un terminal ou si elle est redirigée vers/depuis un fichier ou un canal. Par exemple, si la sortie est redirigée vers un fichier, certains programmes peuvent choisir de supprimer le formatage convivial comme les codes de couleur.

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

int main() {
	si (isatty(STDIN_FILENO)) {
	printf("L'entrée standard est un terminal.\n");
	} autre {
	printf("L'entrée standard n'est pas un terminal.\n");
	}

	si (isatty(STDOUT_FILENO)) {
	printf("La sortie standard est un terminal.\n");
	} autre {
	printf("La sortie standard n'est pas un terminal.\n");
	}

	renvoie 0 ;
}
```

Utilisation courante :

- **Détection du mode interactif**: Un programme peut se comporter différemment s'il détecte qu'il s'exécute de manière interactive (dans un terminal) ou dans un script ou dans le cadre d'un pipeline. Par exemple:
    - Si le programme est connecté à un terminal, il peut afficher des invites ou une sortie formatée.
    - S'il est redirigé vers un fichier, cela peut supprimer le formatage de sortie ou les invites inutiles.

Intégré: STAT + LSTAT

Syntaxe:

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

int stat(const char *chemin, struct stat *buf);
```

statistique de structure:

Le `statistique de structure` structure, qui est utilisée à la fois par `statistique` et `lstat`, contient divers champs pour décrire les propriétés du fichier. Certains champs clés incluent :

```
structure statistique {
       dev_t  st_dev;      // ID de périphérique contenant le fichier
       ino_t  st_ino;      // Numéro d'inode
       mode_t st_mode;     // Mode fichier (type et autorisations)
       nlink_t st_nlink;   // Nombre de liens physiques
       uid_t  st_uid;      // ID utilisateur du propriétaire
       gid_t  st_gid;      // ID de groupe du propriétaire
       dev_t  st_rdev;     // ID de périphérique (si fichier spécial)
       off_t  st_size;     // Taille totale, en octets
       blksize_t st_blksize; // Taille du bloc pour les E/S du système de fichiers
       blkcnt_t st_blocks ;  // Nombre de blocs 512B alloués
       heure_t      st_atime;   // Heure du dernier accès
       heure_t      st_mtime;   // Heure de la dernière modification
       heure_t      st_ctime;   // Heure du dernier changement de statut
} ;
```

Supposer `mon lien` est un lien symbolique qui pointe vers un fichier `monfichier.txt`:

```
struct stat fichier_stat ;
struct stat lien_stat ;

stat("monlien", &file_stat);  // Renvoie des informations sur "monfichier.txt"
lstat("monlien", &link_stat); // Renvoie des informations sur "mylink" lui-même
```

Avec `statistique()`, vous obtiendrez des informations sur myfile.txt (le fichier vers lequel pointe le lien). Avec `lstat()`, vous obtiendrez des informations sur `mon lien` (le lien symbolique lui-même).

Intégré: TCGETATTR

Syntaxe:

```
#include <termios.h>
#include <unistd.h>

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

`tcgetattr` est une fonction en C utilisée pour obtenir les attributs (ou paramètres) actuels du terminal associés à un descripteur de fichier de terminal. Ces attributs contrôlent la façon dont les entrées et les sorties sont gérées par le terminal, comme le mode d'entrée (canonique ou non canonique), l'écho de l'entrée et le traitement des caractères spéciaux.

Exemple de cas d'utilisation :

- **Mode canonique ou non canonique**: Les terminaux fonctionnent généralement dans **mode canonique**, où l'entrée est mise en mémoire tampon de ligne, ce qui signifie que l'entrée n'est envoyée au programme qu'après que l'utilisateur a appuyé sur "Entrée". tcgetattr peut être utilisé pour obtenir les paramètres actuels du terminal, les modifier et définir de nouveaux attributs à l'aide de tcsetattr (par exemple, pour passer en mode non canonique où l'entrée est traitée immédiatement sans attendre « Entrée »).

```
structure termios {
	tcflag_t c_iflag;	// Modes de saisie
	tcflag_t c_oflag;	// Modes de sortie
	tcflag_t c_cflag;	// Modes de contrôle
	tcflag_t c_lflag;	// Modes locaux (par exemple, mode canonique ou non
canonique)
	cc_t	c_cc[NCCS]; // Caractères spéciaux (par exemple, EOF, EOL, etc.)*
} ;
```

Points importants :

- **Cas d'utilisation courants**:
  - Désactivation de l'écho des caractères saisis (utile pour la saisie de mots de passe).
  - Basculement entre les modes de saisie canoniques et non canoniques.
  - Gestion des caractères de contrôle spéciaux (par exemple, Ctrl+C, Ctrl+Z).*
- **Descripteur de fichier**: Le descripteur de fichier est généralement destiné à un terminal (par exemple, STDIN_FILENO pour une entrée standard), mais il peut également s'agir d'un descripteur de fichier pour un autre terminal.
- **Fonctions associées**:
  - `tcsetattr`: Pour paramétrer les attributs du terminal (après les avoir modifiés).
  - `cfmakeraw`: Pour définir le terminal en mode brut, où l'entrée est traitée octet par octet sans mise en mémoire tampon de ligne.

Voir exemple ci-dessous ↓

Exemple : désactivation de l'écho

```
#include <stdio.h>
#include <unistd.h>
#include <termios.h>

int main() {
	struct termios term_settings ;

	// Récupère les attributs actuels du terminal
	tcgetattr(STDIN_FILENO, &term_settings);

	// Désactive le drapeau ECHO
	term_settings.c_lflag &= ~ECHO;

	// Définir les nouveaux attributs du terminal
	tcsetattr(STDIN_FILENO, TCSANOW, &term_settings);

	printf("Tapez quelque chose (il ne sera pas répercuté) : ");
	entrée de caractères [100] ;
	fgets(entrée, taillede(entrée), stdin);

	// Restaurer les paramètres d'origine du terminal
	term_settings.c_lflag |= ÉCHO ;
	tcsetattr(STDIN_FILENO, TCSANOW, &term_settings);

	printf("\nVous avez tapé : %s\n", input);
	renvoie 0 ;
}
```

Ce programme désactive temporairement l'écho, permet à l'utilisateur de saisir une entrée sans la voir à l'écran, puis restaure les paramètres d'origine du terminal.

Caractères de contrôle: SIGNAUX

`EOL :` La fin de la ligne.
`EOF :` La fin du dossier.

`VEOF`: Le signal de fin de fichier (généralement `Ctrl+D`).
`TRANSPORT`: Le signal vers la fin de la ligne (généralement une nouvelle ligne).
`HIVER`: Processus d'interruption (généralement `Ctrl+C`).
`VSUSP` : Suspendre le processus (généralement `Ctrl+Z`).
`SERA VU`: Effacer le dernier caractère (généralement `Retour arrière` ou `Ctrl+H`).
`VKILL`: Effacer toute la ligne (généralement `Ctrl+U`).

`EOF` (Fin du dossier):

- **But**: Le caractère EOF signale la fin de la saisie. En mode canonique (saisie en tampon de ligne), ce caractère permet d'indiquer au terminal que l'utilisateur a terminé la saisie des données.
- **Signification**: EOF signifie « Fin de fichier ». C'est un concept général qui indique la fin de l'entrée d'un flux. Lorsqu'un programme atteint EOF, il sait qu'il n'y a plus de données à lire.
- **Caractère par défaut**: Le caractère EOF par défaut est généralement `Ctrl+D` sur les systèmes de type Unix.
- **Comportement**: Lorsque l'utilisateur appuie sur `Ctrl+D`, le terminal envoie un signal EOF au programme et la saisie est traitée comme si l'utilisateur avait fini de taper.

Exemple: Lorsque vous tapez dans un terminal, appuyez sur `Ctrl+D` indique au terminal qu'aucune entrée n'arrive, provoquant des programmes comme `chat` ou `lire` pour terminer la saisie.

`EOL` (Fin de ligne):

- **But**: Le caractère EOL marque la fin d'une ligne de saisie. Ceci est utilisé en mode canonique où l'entrée est traitée ligne par ligne.
- **Caractère par défaut**: Le caractère de fin de ligne par défaut est généralement la nouvelle ligne (`\n`, ou `Entrer` clé), bien qu'il puisse être personnalisé dans `c_cc`.

Exemple: Lorsque vous appuyez sur le `Entrer` clé, le terminal traite l'entrée et l'envoie au programme pour un traitement ultérieur.

`HIVER` (Interrompre):

- **But**: Le caractère d'interruption envoie un signal (généralement `SIGINT`) au processus de premier plan, y mettant généralement fin.
- **Caractère par défaut**: Le caractère d'interruption par défaut est `Ctrl+C`.

Exemple: Quand vous appuyez `Ctrl+C` dans un terminal, il envoie le `SIGINT` signal au programme en cours d'exécution (comme `dormir` ou `chat`), l'interrompant et l'arrêtant.

`VSUSP` (Suspendre):

- **But**: envoie un signal (`SIGTSTP`) au processus pour suspendre son exécution.
- **Caractère par défaut**: Le caractère de suspension par défaut est `Ctrl+Z`.

Exemple: Quand vous appuyez `Ctrl+Z`, le processus de premier plan est suspendu, ce qui vous permet de le reprendre ultérieurement avec le `fg` commande.

`SERA VU` (Effacer):

- **But**: Ce caractère permet d'effacer le dernier caractère saisi.
- **Caractère par défaut**: Le caractère d'effacement par défaut est `Retour arrière` (`Ctrl+H` sur certains systèmes).

Exemple: Quand vous appuyez `Retour arrière` dans le terminal, le dernier caractère est supprimé de l'entrée.

`VKILL` (Tuer):

- **But**: Le caractère kill efface toute la ligne de saisie actuelle.
- **Caractère par défaut**: Le caractère de mise à mort par défaut est `Ctrl+U`.

Exemple: Pressage `Ctrl+U` efface toute la ligne de texte que vous tapez.

`VEOF`:

- **Signification**: Le `VEOF` indice dans le `c_cc[]` tableau des `termios` La structure représente le caractère de contrôle spécifique utilisé pour indiquer EOF (End of File) en mode canonique. Par défaut, ce caractère est souvent défini sur `Ctrl+D` dans les systèmes de type Unix, signalant la fin de la saisie lors de la saisie dans le terminal.
- **Utilisation en entrée de terminal**: Lorsqu'un utilisateur appuie sur le `VEOF` caractère lors de la saisie dans un terminal, il indique au terminal que l'utilisateur a fini de fournir une entrée. Ce caractère est reconnu par le terminal et utilisé pour signaler la fin du flux d'entrée au programme.

Intégré: TCSETATTR

Syntaxe:

```
#include <termios.h>
#include <unistd.h>

int tcsetattr(int fd, int options_actions, const struct termios *termios_p);
```

`tcsetattr` est utilisé en C pour définir les attributs du terminal associés à un descripteur de fichier, vous permettant de modifier le comportement du terminal.

Paramètres :

- **`fd`**: Le descripteur de fichier du terminal (par exemple, `STDIN_FILENO` pour entrée standard).
- **`actions_facultatif`**: Ceci détermine la manière dont les modifications sont appliquées. Les options courantes sont :
    - `TCSANOW`: Modifiez les attributs immédiatement.
    - `TCSADRAIN`: modifiez les attributs une fois que toutes les sorties ont été transmises.
    - `TCSAFLUSH`: Modifiez les attributs une fois que toutes les sorties ont été transmises et supprimez toutes les entrées.
- **`termios_p`**: Un pointeur vers un `structure termios` contenant les nouveaux attributs du terminal à définir.

jeRemarques importantes :

- Assurez-vous toujours de restaurer les paramètres d'origine du terminal après les avoir modifiés pour éviter tout comportement involontaire.
- Vérifiez les valeurs de retour de `tcgetattr` et `tcsetattr` pour la gestion des erreurs.
- Vous pouvez ajuster d'autres drapeaux dans le `termios` structure pour contrôler divers comportements du terminal, tels que l'activation/désactivation du mode canonique, la modification du contrôle de flux, etc.

`Voir exemple ci-dessous ↓`

Exemple : désactivation de l'écho

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>

int main() {
       struct termios term_settings ;

       // Récupère les attributs actuels du terminal
       if (tcgetattr(STDIN_FILENO, &term_settings) == -1) {
       perror("tcgetattr");
       renvoyer EXIT_FAILURE ;
       }

       // Désactive l'écho des caractères
       term_settings.c_lflag &= ~ECHO;

       // Définir les nouveaux attributs du terminal
       if (tcsetattr(STDIN_FILENO, TCSANOW, &term_settings) == -1) {
       perror("tcsetattr");
       renvoyer EXIT_FAILURE ;
       }

       printf("L'écho est maintenant désactivé. Tapez quelque chose : ");
       entrée de caractères [100] ;
       fgets(entrée, taillede(entrée), stdin);  // Lire l'entrée sans écho

       // Restaurer les attributs du terminal d'origine
       term_settings.c_lflag |= ÉCHO ;  // Réactive l'écho
       if (tcsetattr(STDIN_FILENO, TCSANOW, &term_settings) == -1) {
       perror("tcsetattr");
       renvoyer EXIT_FAILURE ;
       }

       printf("\nVous avez tapé : %s\n", input);
       renvoyer EXIT_SUCCESS ;
}
```

Résumé:

`tcsetattr` est essentiel pour contrôler le comportement du terminal dans les applications qui nécessitent une gestion spécifique des entrées ou des sorties, telles que les éditeurs de texte, les interfaces de ligne de commande ou les programmes interactifs.

Intégré : NOMTTY

Syntaxe:

```
#include <unistd.h>

char *ttyname(int fd);
```

Exemple de cas d'utilisation :

Vous pourriez utiliser `nom de téléphone` pour identifier le terminal avec lequel un programme interagit, ce qui peut être utile à des fins de journalisation ou de débogage.

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

int main() {
	char *term_name = ttyname(STDIN_FILENO);

	if (nom_terme != NULL) {
	printf("Le nom du terminal est : %s\n", term_name);
	} autre {
	perror("nomtty");
	}

	renvoie 0 ;
}
```

Points importants :

- **Référence des bornes**: `nom de téléphone` ne fonctionne qu'avec les descripteurs de fichiers qui font référence aux terminaux. Si vous passez un descripteur de fichier qui ne correspond pas à un terminal, il renverra `NUL`.
- **Cas d'utilisation**: Utile dans les applications où vous devez déterminer le périphérique terminal pour la sortie ou configurer les paramètres spécifiques à ce terminal.

Intégré: TTYSLOT

Syntaxe:

```
#include <unistd.h>

int ttyslot (vide);
```

`ttyslot` est une fonction en C qui renvoie le numéro d'emplacement du terminal associé à un descripteur de fichier donné. Ce numéro d'emplacement est souvent utilisé pour identifier le terminal dans un environnement multi-utilisateurs, en particulier dans les systèmes prenant en charge plusieurs terminaux.

Exemple de cas d'utilisation :

`ttyslot` est utile dans les environnements où vous souhaitez suivre ou gérer les sessions de terminal, en particulier dans les systèmes multi-utilisateurs. Il peut être utilisé conjointement avec d'autres fonctions liées au terminal pour déterminer les sessions utilisateur ou à des fins de journalisation.

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

int main() {
	int emplacement = ttyslot();

	si (emplacement != -1) {
	printf("Le numéro d'emplacement du terminal est : %d\n", slot);
	} autre {
	perror("ttyslot");
	}

	renvoie 0 ;
}
```

Intégré: DIRENT + OPENDIR + CLOSEDIR

Syntaxe:

```
#include <unistd.h>

int ttyslot (vide);
```

Résumé:

Le `dirent` La structure est un élément clé de la gestion des répertoires en C, permettant aux programmes d'interagir avec le contenu du répertoire et de le lire efficacement. Il est couramment utilisé dans les tâches de navigation et de manipulation du système de fichiers.

`répertoire ouvert` et `fermé` sont des fonctions essentielles pour la manipulation de répertoires en C, permettant aux programmes d'ouvrir, de lire et de fermer efficacement les flux de répertoires tout en gérant correctement les ressources.

Composants clés :

1. **`struct dirent`**:
    - Cette structure contient généralement des informations sur une entrée de fichier ou de répertoire. Les domaines les plus pertinents incluent généralement :
        - **`d_ino`**: Le numéro d'inode du fichier (un identifiant unique pour le fichier dans le système de fichiers).
        - **`type_d`**: Le type du fichier (par exemple, fichier normal, répertoire, lien symbolique).
        - **`nom_d`**: Le nom du fichier ou du répertoire (une chaîne terminée par un caractère nul).

    Exemple de définition :
    ```
    struct dirent {
            ino_t d_ino;              // Numéro d'inode
            off_t d_off;              // Décalage vers la direction suivante
            court d_reclen non signé; // Longueur de cet enregistrement
            caractère non signé d_type ;     // Type de fichier
            char d_name[256];         // Nom de fichier
    } ;
    ```

2. **`Opérations d'annuaire`**:
    - Fonctions qui utilisent le `dirent` structure comprend :
        - **`répertoire ouvert`**: Ouvre un flux de répertoire correspondant à un nom de répertoire.
        - **`répertoire de lecture`**: Lit une entrée de répertoire à partir du flux de répertoire.
        - **`fermé`**: ferme un flux de répertoire.

`Voir exemple ci-dessous ↓`

Exemple : Lire le contenu du répertoire

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main() {
	VOUS *vous;
	struct direct *entrée ;

	// Ouvre le répertoire courant
	dir = opendir(".");

	si (rép == NULL) {
	perror("opendir");
	renvoyer EXIT_FAILURE ;
	}

	// Lire les entrées du répertoire
	while ((entry = readdir(dir)) != NULL) {
	// Affiche le nom de chaque entrée
	printf("Nom : %s, Inode : %lu, Type : %d\n", entrée->nom_d, (long non
signé)entrée->d_ino, entrée->type_d);
	}

	// Ferme le flux du répertoire
	closeir(rép);
	renvoyer EXIT_SUCCESS ;
}
```

Points importants :

- **Types de fichiers**: Le `type_d` Ce champ peut aider à identifier le type d'entrée. Les valeurs communes incluent :
    - `DT_REG`: Fichier régulier
    - `DT_REP`: Annuaire
    - `DT_LNK`: Lien symbolique
    - `DT_CHR`: Appareil de caractère
    - `DT_BLK`: Bloquer l'appareil
    - `DT_SOCK`: Prise
    - `DT_FIFO`: Canal nommé (FIFO)
- Notez que la disponibilité de `type_d` peut dépendre du système de fichiers et de la plate-forme.
- **Portabilité**: Le `dirent` La structure et ses fonctions associées font partie de la norme POSIX, ce qui les rend largement disponibles sur les systèmes de type Unix, notamment Linux et macOS.

Intégré:SIGNAL

Syntaxe:

```
#include <signal.h>

void (*signal(int sig, void (*handler)(int)))(int);
```

Résumé:

Le `signal` La fonction spécifie comment un programme doit réagir à un signal spécifique lorsqu'il est reçu. Il définit comment gérer les signaux envoyés par le système d'exploitation. Les signaux sont des notifications asynchrones envoyées à un processus pour l'informer d'événements tels que des interruptions, des demandes de terminaison ou un accès mémoire non valide.

Paramètres :

- **`eux-mêmes`**: Le numéro du signal à attraper ou à gérer. Les signaux courants incluent :
  - `SIGINT`: Envoyé lorsque l'utilisateur interrompt le programme (par exemple, en appuyant sur `Ctrl+C`).
  - `DURÉE CIBLE`: Envoyé pour demander la fin du programme.
  - `SIGTUER`: termine immédiatement le processus (ne peut pas être intercepté ou ignoré).
  - `SIGSEGV`: Envoyé sur une référence mémoire invalide (défaut de segmentation).
  - `SIGALRM`: Envoyé lorsqu'une minuterie réglée par le `alarme()` la fonction expire.
- **`gestionnaire`**: Un pointeur vers une fonction qui spécifie comment gérer le signal. Cela peut être :
  - Une fonction de gestionnaire de signal personnalisée que vous définissez.
  - `SIG_DFL`: L'action par défaut pour le signal (telle que la fin du processus).
  - `SIG_IGN`: Ignorez le signal.

Cas d'utilisation courants:

Attraper des interruptions comme `SIGINT` pour des arrêts sécurisés, la gestion des défauts de segmentation (`SIGSEGV`), ou définir un comportement personnalisé pour divers signaux de terminaison.

Exemple d'ignorance `SIGINT`:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main() {
	// Ignore le signal SIGINT
	signal(SIGINT, SIG_IGN);

	// Boucle infinie, en attente de signaux
	tandis que (1) {
	printf("En cours d'exécution... Vous ne pouvez pas m'arrêter avec
Ctrl+C!\n");
	dormir(1);
	}

	renvoie 0 ;
```

```
}
```

Intégré: SIGEMPTYSET + SIGADDSET

Syntaxe: `sigemptyset`

```
#include <signal.h>

int sigemptyset(sigset_t *set);
```

1. `sigemptyset`

- **But**: Initialise un signal défini comme étant vide, ce qui signifie qu'il ne contient aucun signal.
- **Usage**: Avant d'ajouter des signaux à un ensemble, vous l'initialisez généralement avec `sigemptyset` pour effacer tout contenu existant.

Syntaxe: `la mouette`

```
#include <signal.h>

int sigaddset(sigset_t *set, int signum);
```

2. `la mouette`

- **But**: Ajoute un signal spécifique à un ensemble de signaux pour des opérations futures, comme le bloquer ou le gérer.

Exemple de code :

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

int main() {
	sigset_t défini ;

	// Initialise le signal défini pour qu'il soit vide
	sigemptyset(&set);

	// Ajoute SIGINT (Ctrl+C) à l'ensemble de signaux
	sigaddset(&set, SIGINT);

	// Vérifiez si SIGINT est dans l'ensemble
	if (sigismember(&set, SIGINT)) {
	printf("SIGINT est dans le jeu de signaux.\n");
	} autre {
	printf("SIGINT n'est pas dans le jeu de signaux.\n");
	}

	renvoie 0 ;
}
```

Intégré: EXEMPLE

Syntaxe:

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

Exemple de cas d'utilisation :

`ttyslot` est utile dans les environnements où vous souhaitez suivre ou gérer les sessions de terminal, en particulier dans les systèmes multi-utilisateurs. Il peut être utilisé conjointement avec d'autres fonctions liées au terminal pour déterminer les sessions utilisateur ou à des fins de journalisation.

```
#include <stdio.h>

Voir exemple ci-dessous ↓
```

Intégré: EXEMPLE

Syntaxe:

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

Exemple de cas d'utilisation :

`ttyslot` est utile dans les environnements où vous souhaitez suivre ou gérer les sessions de terminal, en particulier dans les systèmes multi-utilisateurs. Il peut être utilisé conjointement avec d'autres fonctions liées au terminal pour déterminer les sessions utilisateur ou à des fins de journalisation.

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

`Voir exemple ci-dessous ↓`

Intégré: EXEMPLE

Syntaxe:

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

Exemple de cas d'utilisation :

`ttyslot` est utile dans les environnements où vous souhaitez suivre ou gérer les sessions de terminal, en particulier dans les systèmes multi-utilisateurs. Il peut être utilisé conjointement avec d'autres fonctions liées au terminal pour déterminer les sessions utilisateur ou à des fins de journalisation.

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

`Voir exemple ci-dessous ↓`

Intégré: EXEMPLE

Syntaxe:

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

Exemple de cas d'utilisation :

`ttyslot` est utile dans les environnements où vous souhaitez suivre ou gérer les sessions de terminal, en particulier dans les systèmes multi-utilisateurs. Il peut être utilisé conjointement avec d'autres fonctions liées au terminal pour déterminer les sessions utilisateur ou à des fins de journalisation.

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

`Voir exemple ci-dessous ↓`

Intégré: EXEMPLE

Syntaxe:

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

Exemple de cas d'utilisation :

`ttyslot` est utile dans les environnements où vous souhaitez suivre ou gérer les sessions de terminal, en particulier dans les systèmes multi-utilisateurs. Il peut être utilisé conjointement avec d'autres fonctions liées au terminal pour déterminer les sessions utilisateur ou à des fins de journalisation.

```
#include <stdio.h>

Voir exemple ci-dessous ↓
```