fonctions-minishell

Raisonnement terminé pendant 55 secondes ›

Voici un tour d'horizon détaillé de chacune des fonctions autorisées dans le cadre du projet minishell de l'école 42. Pour chaque fonction, vous trouverez une brève description de son rôle, un exemple d'utilisation, la ou les bibliothèques à inclure et une suggestion d'exercice pour vous entraîner.

---

1. Fonctions de la bibliothèque Readline

Ces fonctions proviennent de la bibliothèque GNU Readline (et History) et permettent de gérer la saisie de lignes de commande avec édition, historique et rafraîchissement de l'affichage.

- **readline**
  - **Description** : Lit une ligne de commande depuis l'entrée standard, en offrant des fonctionnalités d'édition (navigation, suppression, etc.).
  - **Utilisation** :

```c
#include <readline/readline.h>
char *line = readline("minishell> ");
```

  - **Bibliothèque** : `<readline/readline.h>`
  - **Exercice** : Créez un petit programme qui affiche un prompt, lit la commande de l'utilisateur avec `readline` et affiche ensuite la chaîne récupérée.
- **rl_clear_history**
  - **Description** : Efface l'historique des commandes enregistrées.
  - **Utilisation** :

```c
#include <readline/history.h>
rl_clear_history();
```

  - **Bibliothèque** : `<readline/history.h>`
  - **Exercice** : Ajoutez plusieurs commandes à l'historique (avec `add_history`), puis effacez-le et vérifiez que l'historique est bien vidé.
- **rl_on_new_line**
  - **Description** : Indique à Readline qu'une nouvelle ligne est commencée. Utile notamment pour réafficher le prompt correctement après une interruption ou un changement d'état.
  - **Utilisation** :

```c
#include <readline/readline.h>
rl_on_new_line();
```

  - **Bibliothèque** : `<readline/readline.h>`

- **Exercice** : Simulez une interruption (par exemple via un signal) et utilisez `rl_on_new_line` pour forcer l'affichage du prompt sur une nouvelle ligne.

- **rl_replace_line**
  - **Description** : Remplace le contenu actuel de la ligne éditée par une nouvelle chaîne.
  - **Utilisation** :

```c
#include <readline/readline.h>
rl_replace_line("nouvelle commande", 0);  // 0 pour ne pas effacer l'historique de la mod
```

  - **Bibliothèque** : `<readline/readline.h>`
  - **Exercice** : Implémentez une fonctionnalité de correction automatique dans votre minishell en remplaçant la commande saisie par une version modifiée.
- **rl_redisplay**
  - **Description** : Force la mise à jour de l'affichage de la ligne en cours d'édition.
  - **Utilisation** :

```c
#include <readline/readline.h>
rl_redisplay();
```

  - **Bibliothèque** : `<readline/readline.h>`
  - **Exercice** : Après avoir modifié la ligne avec `rl_replace_line`, utilisez `rl_redisplay` pour que l'utilisateur voit immédiatement le changement.
- **add_history**
  - **Description** : Ajoute la ligne passée en paramètre à l'historique des commandes.
  - **Utilisation** :

```c
#include <readline/history.h>
add_history(line);
```

  - **Bibliothèque** : `<readline/history.h>`
  - **Exercice** : Enregistrez chaque commande saisie par l'utilisateur et proposez une commande interne (par exemple `history`) qui affiche l'historique.

---

2. Fonction d'affichage formaté

- **printf**
  - **Description** : Affiche sur la sortie standard une chaîne formatée (similaire à la commande shell `echo` mais avec un formatage puissant).
  - **Utilisation** :

```c
#include <stdio.h>
printf("Nombre: %d, chaîne: %s\n", 42, "exemple");
```

- **Bibliothèque** : `<stdio.h>`
- **Exercice** : Écrivez un programme qui affiche divers types de données (entiers, flottants, chaînes) en utilisant différents formats pour vous familiariser avec les spécificateurs de format.

---

3. Fonctions de gestion des fichiers et des processus

a) Vérification d'accès et gestion de fichiers

- **access**
  - **Description** : Vérifie si un fichier existe et/ou si l'utilisateur dispose des droits (lecture, écriture, exécution) sur ce fichier.
  - **Utilisation** :

```c
#include <unistd.h>
if (access("monfichier.txt", R_OK) == 0)
  printf("Le fichier est lisible.\n");
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Créez une fonction qui prend un chemin de fichier et vérifie si le fichier est accessible en lecture et exécution.

b) Création et gestion des processus

- **fork**
  - **Description** : Crée un nouveau processus (processus enfant) qui est une copie du processus appelant.
  - **Utilisation** :

```c
#include <unistd.h>
pid_t pid = fork();
if (pid == 0) {
    // Code du processus enfant
} else if (pid > 0) {
    // Code du processus parent
} else {
    perror("fork");
}
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Programmez un shell qui crée un processus enfant pour exécuter une commande et affiche des messages différents dans le parent et l'enfant.
- **wait**
  - **Description** : Attend la fin d'un processus enfant et récupère son statut de sortie.
  - **Utilisation** :

```c
#include <sys/wait.h>
int status;
```

```
wait(&status);
```

- **Bibliothèque** : `<sys/wait.h>`
- **Exercice** : Utilisez `fork` pour lancer un processus enfant, puis `wait` pour attendre sa terminaison et afficher le code de sortie.

- **waitpid**
  - **Description** : Attend la terminaison d'un processus enfant spécifique, permettant de contrôler plus précisément quel processus attendre.
  - **Utilisation** :

```c
#include <sys/wait.h>
int status;
waitpid(pid, &status, 0);
```

  - **Bibliothèque** : `<sys/wait.h>`
  - **Exercice** : Dans un shell multi-processus, utilisez `waitpid` pour attendre la fin d'un processus particulier pendant que d'autres continuent de s'exécuter.
- **wait3**
  - **Description** : Similaire à `wait`, mais permet également de récupérer des informations sur l'utilisation des ressources du processus terminé. (À noter : cette fonction peut être obsolète ou non portable sur certains systèmes.)
  - **Utilisation** :

```c
#include <sys/wait.h>
#include <sys/resource.h>
int status;
struct rusage usage;
wait3(&status, 0, &usage);
```

  - **Bibliothèque** : `<sys/wait.h>` et `<sys/resource.h>`
  - **Exercice** : Lancez un processus enfant et utilisez `wait3` pour afficher l'utilisation CPU ou mémoire du processus une fois terminé.
- **wait4**
  - **Description** : Permet d'attendre un processus enfant spécifique et de récupérer des informations sur son utilisation des ressources.
  - **Utilisation** :

```c
#include <sys/wait.h>
#include <sys/resource.h>
int status;
struct rusage usage;
wait4(pid, &status, 0, &usage);
```

  - **Bibliothèque** : `<sys/wait.h>` et `<sys/resource.h>`
  - **Exercice** : Créez un programme qui lance plusieurs processus enfants et récupère pour chacun des informations sur l'utilisation des ressources via `wait4`.

c) Exécution d'un nouveau programme

- **execve**
  - **Description** : Remplace l'image du processus courant par celle d'un nouveau programme.
  - **Utilisation** :

```c
#include <unistd.h>
char *argv[] = { "ls", "-l", NULL };
extern char **environ;
execve("/bin/ls", argv, environ);
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Dans votre shell, après avoir créé un processus enfant via `fork`, utilisez `execve` pour lancer une commande externe (comme `ls`).

d) Redirection des descripteurs de fichiers

- **dup**
  - **Description** : Duplique un descripteur de fichier, créant ainsi un nouveau descripteur qui référence le même fichier ouvert.
  - **Utilisation** :

```c
#include <unistd.h>
int new_fd = dup(old_fd);
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Créez un programme qui redirige la sortie standard vers un fichier en dupliquant et redirigeant le descripteur.
- **dup2**
  - **Description** : Duplique un descripteur de fichier sur un numéro spécifique (permet de rediriger un flux vers un descripteur connu).
  - **Utilisation** :

```c
#include <unistd.h>
dup2(old_fd, new_fd);
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Implémentez la redirection d'entrée/sortie dans votre minishell pour réaliser des opérations de type `>` ou `<`.
- **pipe**
  - **Description** : Crée un canal de communication (pipe) qui permet à deux processus de communiquer en écriture/lecture.
  - **Utilisation** :

```c
#include <unistd.h>
int fds[2];
if (pipe(fds) == -1) {
  perror("pipe");
```

```
}
// fds[0] pour la lecture, fds[1] pour l'écriture
```

- **Bibliothèque** : `<unistd.h>`
- **Exercice** : Créez un pipeline simple entre deux processus, par exemple pour émuler le comportement de `ls | grep txt`.

---

4. Fonctions de manipulation des répertoires et du système de fichiers

a) Gestion du répertoire de travail et informations sur les fichiers

- **getcwd**
  - **Description** : Renvoie le chemin absolu du répertoire de travail courant.
  - **Utilisation** :

```c
#include <unistd.h>
#include <limits.h>
char buffer[PATH_MAX];
if (getcwd(buffer, sizeof(buffer)) != NULL)
  printf("Répertoire courant : %s\n", buffer);
```

  - **Bibliothèque** : `<unistd.h>` (et éventuellement `<limits.h>` pour `PATH_MAX`)
  - **Exercice** : Écrivez un programme qui affiche le répertoire de travail courant.
- **chdir**
  - **Description** : Change le répertoire de travail courant du processus.
  - **Utilisation** :

```c
#include <unistd.h>
if (chdir("/nouveau/chemin") == -1)
  perror("chdir");
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Implémentez la commande interne `cd` dans votre minishell.
- **stat**
  - **Description** : Récupère les informations d'un fichier (taille, permissions, type, etc.) et les stocke dans une structure `struct stat`.
  - **Utilisation** :

```c
#include <sys/stat.h>
struct stat file_stat;
if (stat("monfichier.txt", &file_stat) == 0)
  printf("Taille: %ld\n", file_stat.st_size);
```

  - **Bibliothèque** : `<sys/stat.h>`

- **Exercice** : Écrivez un programme qui affiche, à la manière de `ls -l`, les permissions et la taille d'un fichier.

- **lstat**
  - **Description** : Comme `stat`, mais ne suit pas les liens symboliques (permet de connaître les informations sur le lien lui-même).
  - **Utilisation** :

```c
#include <sys/stat.h>
struct stat file_stat;
if (lstat("monlien", &file_stat) == 0)
  // Analysez file_stat pour déterminer s'il s'agit d'un lien symbolique
```

  - **Bibliothèque** : `<sys/stat.h>`
  - **Exercice** : Comparez les résultats de `stat` et `lstat` sur un fichier qui est un lien symbolique pour comprendre la différence.

- **fstat**
  - **Description** : Récupère les informations d'un fichier à partir de son descripteur de fichier.
  - **Utilisation** :

```c
#include <sys/stat.h>
int fd = open("monfichier.txt", O_RDONLY);
struct stat file_stat;
fstat(fd, &file_stat);
close(fd);
```

  - **Bibliothèque** : `<sys/stat.h>`
  - **Exercice** : Ouvrez un fichier avec `open`, puis utilisez `fstat` pour en afficher la taille et les permissions.

b) Suppression de fichiers

- **unlink**
  - **Description** : Supprime (efface) un fichier du système de fichiers.
  - **Utilisation** :

```c
#include <unistd.h>
if (unlink("fichier_a_supprimer.txt") == -1)
  perror("unlink");
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Implémentez une commande interne `rm` dans votre minishell pour supprimer un fichier.

---

5. Fonctions de gestion du terminal et de l'environnement

a) Gestion des erreurs

- **strerror**
  - **Description** : Renvoie une chaîne de caractères décrivant l'erreur correspondant au code passé (souvent `errno`).
  - **Utilisation** :

```c
#include <string.h>
#include <errno.h>
printf("Erreur : %s\n", strerror(errno));
```

  - **Bibliothèque** : `<string.h>` et `<errno.h>`
  - **Exercice** : Forcez une erreur (par exemple, en ouvrant un fichier inexistant) et affichez le message d'erreur à l'aide de `strerror`.
- **perror**
  - **Description** : Affiche un message d'erreur sur la sortie d'erreur standard en se basant sur la variable globale `errno`.
  - **Utilisation** :

```c
#include <stdio.h>
perror("Message d'erreur");
```

  - **Bibliothèque** : `<stdio.h>`
  - **Exercice** : Après un appel système qui échoue, utilisez `perror` pour afficher un message d'erreur explicatif.

b) Informations sur le terminal

- **isatty**
  - **Description** : Vérifie si un descripteur de fichier est associé à un terminal interactif.
  - **Utilisation** :

```c
#include <unistd.h>
if (isatty(STDIN_FILENO))
  printf("Entrée standard est un terminal.\n");
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Dans votre minishell, vérifiez si l'entrée standard est un terminal pour adapter le comportement (mode interactif vs. mode script).
- **ttyname**
  - **Description** : Renvoie le nom du terminal associé à un descripteur de fichier.
  - **Utilisation** :

```c
#include <unistd.h>
char *term = ttyname(STDIN_FILENO);
```

```
if (term)
  printf("Terminal: %s\n", term);
```

- **Bibliothèque** : `<unistd.h>`
- **Exercice** : Affichez le nom du terminal à partir duquel le shell a été lancé.

- **ttyslot**
  - **Description** : Retourne le numéro de « slot » du terminal à partir duquel l'entrée est fournie (fonction moins utilisée et parfois spécifique à certains systèmes).
  - **Utilisation** :

```c
#include <unistd.h>
int slot = ttyslot();
printf("Slot du terminal: %d\n", slot);
```

  - **Bibliothèque** : `<unistd.h>`
  - **Exercice** : Essayez d'afficher le slot du terminal dans un programme interactif pour voir si votre système le supporte.
- **ioctl**
  - **Description** : Permet d'effectuer des opérations de contrôle sur des périphériques (souvent utilisé pour configurer ou interroger des paramètres du terminal, comme la taille de la fenêtre).
  - **Utilisation** :

```c
#include <sys/ioctl.h>
// Exemple : récupération de la taille du terminal
struct winsize ws;
ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws);
printf("Colonnes: %d, Lignes: %d\n", ws.ws_col, ws.ws_row);
```

  - **Bibliothèque** : `<sys/ioctl.h>`
  - **Exercice** : Utilisez `ioctl` pour récupérer la taille de la fenêtre du terminal et adapter l'affichage de votre shell en conséquence.
- **getenv**
  - **Description** : Récupère la valeur d'une variable d'environnement.
  - **Utilisation** :

```c
#include <stdlib.h>
char *home = getenv("HOME");
if (home)
  printf("HOME: %s\n", home);
```

  - **Bibliothèque** : `<stdlib.h>`
  - **Exercice** : Implémentez une commande interne `env` qui affiche toutes les variables d'environnement ou recherche la valeur d'une variable donnée.

---

6. Fonctions de gestion des attributs du terminal (Termios / Termcap)

Ces fonctions permettent de configurer finement le comportement du terminal (mode canonique, non-canonique, gestion des capacités, etc.) :

- **tcsetattr**
  - **Description** : Définit ou modifie les attributs du terminal (ex. passage en mode non-canonique).
  - **Utilisation** :

```c
#include <termios.h>
struct termios term;
tcgetattr(STDIN_FILENO, &term);
term.c_lflag &= ~(ICANON | ECHO);
tcsetattr(STDIN_FILENO, TCSANOW, &term);
```

  - **Bibliothèque** : `<termios.h>`
  - **Exercice** : Créez un programme qui passe le terminal en mode non-canonique pour lire des caractères immédiatement sans attendre un retour à la ligne, puis restaurez les attributs.
- **tcgetattr**
  - **Description** : Récupère les attributs actuels du terminal.
  - **Utilisation** :

```c
#include <termios.h>
struct termios term;
tcgetattr(STDIN_FILENO, &term);
```

  - **Bibliothèque** : `<termios.h>`
  - **Exercice** : Sauvegardez la configuration du terminal au démarrage de votre shell, puis restaurez-la à la sortie du programme.
- **tgetent**
  - **Description** : Initialise la base de données Termcap en chargeant les informations de capacités du terminal.
  - **Utilisation** :

```c
#include <term.h>
char buffer[2048];
int success = tgetent(buffer, getenv("TERM"));
if (success < 1)
  fprintf(stderr, "Terminal non supporté.\n");
```

  - **Bibliothèque** : `<term.h>` (parfois via `<curses.h>` selon l'implémentation)
  - **Exercice** : Utilisez `tgetent` pour vérifier et afficher si certaines capacités du terminal (comme le support de la couleur) sont disponibles.
- **tgetflag**

- **Description** : Récupère la valeur booléenne d'une capacité du terminal dans la base Termcap (par exemple, support de l'automargin).
- **Utilisation** :

```c
int am = tgetflag("am");
```

- **Bibliothèque** : `<term.h>`
- **Exercice** : Vérifiez si le terminal possède certaines capacités (comme le défilement automatique) et affichez le résultat.

- **tgetnum**
  - **Description** : Récupère la valeur numérique d'une capacité du terminal (par exemple, le nombre de colonnes).
  - **Utilisation** :

```c
int cols = tgetnum("co");
```

  - **Bibliothèque** : `<term.h>`
  - **Exercice** : Utilisez `tgetnum` pour obtenir la largeur du terminal et adapter dynamiquement l'affichage de votre shell.
- **tgetstr**
  - **Description** : Récupère une chaîne de contrôle associée à une capacité du terminal (exemple : le code pour effacer l'écran).
  - **Utilisation** :

```c
char *clear_str = tgetstr("cl", NULL);
```

  - **Bibliothèque** : `<term.h>`
  - **Exercice** : Testez l'effacement de l'écran dans votre shell en récupérant et en envoyant la chaîne de contrôle via `tputs`.
- **tgoto**
  - **Description** : Construit une séquence de contrôle pour déplacer le curseur à une position donnée, à partir d'une chaîne obtenue avec `tgetstr`.
  - **Utilisation** :

```c
char *move_cursor = tgoto(clear_str, col, row);
```

  - **Bibliothèque** : `<term.h>`
  - **Exercice** : Créez une fonction qui déplace le curseur à une position (ligne, colonne) spécifique dans le terminal, puis affichez du texte à cet endroit.
- **tputs**
  - **Description** : Envoie au terminal une séquence de contrôle (avec gestion du padding, si nécessaire) pour appliquer une action, comme le repositionnement du curseur ou

l'effacement de l'écran.
- **Utilisation** :

```
c

tputs(move_cursor, 1, putchar);
```

- **Bibliothèque** : `<term.h>`
- **Exercice** : Utilisez `tputs` pour réaliser diverses actions sur le terminal, par exemple pour créer une animation simple en déplaçant du texte sur l'écran.

Conclusion

Chaque fonction présentée joue un rôle spécifique dans la gestion d'un minishell, que ce soit pour la saisie de commandes, la gestion des processus et des fichiers, ou encore la configuration et le contrôle du terminal. Pour bien assimiler ces notions, il est recommandé de réaliser de petits programmes ou exercices ciblés (comme ceux suggérés ci-dessus) avant d'intégrer l'ensemble dans votre projet minishell.

N'hésitez pas à tester chaque fonction individuellement et à consulter la documentation associée pour approfondir leur utilisation et leurs subtilités.