Minishell

Liens utiles

Man fr de bash : https://fr.manpages.org/bash
Subjet fr : https://github.com/iciamyplant/Minishell/blob/master/subject/fr.subject.pdf
Subject correction : https://github.com/Hqndler/42-minishell?tab=readme-ov-file#aper%C3%A7u-de-la-correction
Testeur : https://github.com/LucasKuhn/minishell_tester
All tests : https://docs.google.com/spreadsheets/d/1uJHQu0VPsjjBkR4hxOeCMEt3AOM1Hp_SmUzPFhAH-nA/edit?gid=0#gid=0
Apprendre GitHub : https://learngitbranching.js.org/?locale=fr_FR
Explications : https://github.com/iciamyplant/Minishell

Les Fonctions

Prompt

readline
char *readline(const char *prompt);
Description : Permet de lire une ligne de texte à partir de l'entrée standard (généralement le terminal) avec des fonctionnalités améliorées, telles que l'historique des commandes et l'édition de texte.

rl_clear_history
void rl_clear_history(void);
Description : Efface l'historique des commandes dans l'interface de ligne de commande gérée par la bibliothèque readline.

rl_on_new_line
int rl_on_new_line(void);
Description : Indique que l'on commence une nouvelle ligne dans le readline, souvent utilisée pour le contrôle du curseur dans un terminal.

rl_replace_line
int rl_replace_line(const char *text, int clear_undo);
Description : Remplace la ligne courante dans l'éditeur readline sans en ajouter une nouvelle dans l'historique.

rl_redisplay
int rl_redisplay(void);
Description : Réaffiche la ligne actuelle dans l'interface readline, utile après modification du texte.

add_history
int add_history(const char *line);
Description : Ajoute une ligne au fichier d'historique dans readline.

Print

#include <stdio.h>

printf

int printf(const char *format, ...);

Description : Permet d'afficher une sortie formatée sur la sortie standard (généralement le terminal).

#include <unistd.h>

write

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

Description : Écrit des données à un descripteur de fichier (par exemple, un fichier ou le terminal). C'est une version bas-niveau de l'écriture.

Allocation

#include <stdlib.h>

malloc

void *malloc(size_t size);

Description : Alloue dynamiquement de la mémoire pour un nombre donné d'octets et renvoie un pointeur vers cette mémoire allouée.

free

void free(void *ptr);

Description : Libère la mémoire précédemment allouée par malloc ou une autre fonction d'allocation dynamique de mémoire.

Process child/parent

#include <unistd.h>

fork

pid_t fork(void);

Description : Crée un nouveau processus en dupliquant le processus appelant. Ce processus fils a une copie de l'espace mémoire du parent.

#include <sys/wait.h>

wait

pid_t wait(int *wstatus);

Description : Attends que l'un de ses enfants termine son exécution, et récupère son code de sortie.

waitpid

pid_t waitpid(pid_t pid, int *wstatus, int options);

Description : Attend la fin d'un processus spécifique identifié par son PID. Offre un contrôle plus précis que wait.

wait3

pid_t wait3(int *wstatus, int options, struct rusage *rusage);

Description : Attends la fin d'un ou plusieurs processus fils et recueille des informations supplémentaires sur les ressources utilisées par les processus.

wait4

pid_t wait4(pid_t pid, int *wstatus, int options, struct rusage *rusage);

Description : Semblable à wait3, mais permet de spécifier un processus spécifique à attendre. Récupère également des informations sur les ressources utilisées.

Pipe et exec

#include <unistd.h>

pipe
int pipe(int pipefd[2]);
Description : Crée une paire de descripteurs de fichier (un tube) pour la communication entre processus.

execve
int execve(const char *pathname, char *const argv[], char *const envp[]);
Description : Remplace le processus actuel par un nouveau programme. Cela permet de lancer un autre programme à partir du programme en cours.

open/read/close File

#include <fcntl.h>

open
int open(const char *pathname, int flags, ...);
Description : Ouvre un fichier (ou crée un fichier s'il n'existe pas) et renvoie un descripteur de fichier pour effectuer des opérations d'entrée/sortie.

#include <unistd.h>

read
ssize_t read(int fd, void *buf, size_t count);
Description : Lit des données à partir d'un descripteur de fichier. C'est une opération d'entrée de bas niveau.

close
int close(int fd);
Description : Ferme un descripteur de fichier précédemment ouvert avec open..

open/read/close Dir

#include <dirent.h>

opendir
DIR *opendir(const char *name);
Description : Ouvre un répertoire pour permettre de lire son contenu avec readdir.

readdir
struct dirent *readdir(DIR *dirp);
Description : Lit un élément (fichier ou sous-répertoire) d'un répertoire ouvert.

closedir
int closedir(DIR *dirp);
Description : Ferme un répertoire ouvert par opendir

Duplique/supp

#include <unistd.h>

dup
int dup(int oldfd);
Description : Duplique un descripteur de fichier existant. Le nouveau descripteur renverra les mêmes données que l'ancien.

dup2
int dup2(int oldfd, int newfd);
Description : Duplique un descripteur de fichier en le remplaçant par un autre descripteur spécifié.

unlink
int unlink(const char *pathname);
Description : Supprime un fichier du système de fichiers. C'est l'équivalent de "delete" en C.

Infos file

#include <sys/stat.h>

stat
int stat(const char *pathname, struct stat *statbuf);
Description : Renvoie des informations détaillées sur un fichier, telles que sa taille, ses permissions, et ses dates de création/dernière modification.

lstat
int lstat(const char *pathname, struct stat *statbuf);
Description : Semblable à stat, mais renvoie des informations sur un lien symbolique plutôt que sur le fichier cible.

fstat
int fstat(int fd, struct stat *statbuf);
Description : Renvoie des informations sur un fichier ouvert (via un descripteur de fichier).

Chemin

#include <unistd.h>

getcwd
char *getcwd(char *buf, size_t size);
Description : Renvoie le chemin absolu du répertoire de travail actuel.

Variable d'environnement

#include <stdlib.h>

getenv
char *getenv(const char *name);
Description : Renvoie la valeur d'une variable d'environnement spécifiée.

Check

#include <unistd.h>

access

int access(const char *pathname, int mode);

Description : Vérifie l'existence et les permissions d'un fichier ou d'un répertoire.

Move

#include <unistd.h>

chdir

int chdir(const char *path);

Description : Change le répertoire de travail courant du programme.

Error

#include <string.h>

strerror

char *strerror(int errnum);

Description : Renvoie une chaîne de caractères décrivant l'erreur correspondant à un code d'erreur (par exemple, errno).

#include <stdio.h>

perror

void perror(const char *s);

Description : Affiche un message d'erreur avec la chaîne de caractères associée à errno et un message personnalisé.

Sortie

#include <stdlib.h>

exit

void exit(int status);

Description : Termine l'exécution du programme avec un code de sortie spécifique.

Signal

#include <signal.h>

signal

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

Description : Permet de définir un gestionnaire de signaux pour un signal spécifique (comme SIGINT pour une interruption de clavier).

sigaction

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

Description : Permet de définir un gestionnaire de signaux avec des options plus fines et robustes que signal.

sigemptyset

int sigemptyset(sigset_t *set);

Description : Initialise un ensemble de signaux vide.

sigaddset

int sigaddset(sigset_t *set, int signum);

Description : Ajoute un signal à un ensemble de signaux.

kill

int kill(pid_t pid, int sig);

Description : Envoie un signal à un processus (souvent utilisé pour envoyer des signaux comme SIGTERM ou SIGKILL).

Gestion des terminaux

#include <unistd.h>

isatty
int isatty(int fd);
Description : Vérifie si un descripteur de fichier correspond à un terminal interactif.

ttyname
char *ttyname(int fd);
Description : Renvoie le nom du terminal associé à un descripteur de fichier.

ttyslot
int ttyslot(void);
Description : Renvoie l'index du terminal actuel.

#include <sys/ioctl.h>

ioctl
int ioctl(int fd, unsigned long request, ...);
Description : Permet de manipuler des périphériques (disques, terminaux, etc.) à bas niveau. Très utilisé pour configurer des paramètres spécifiques.

Attributs des terminaux

#include <termios.h>

tcsetattr
int tcsetattr(int fd, int optional_actions, const struct termios *termios_p);
Description : Modifie les paramètres d'un terminal (comme la mise en forme des entrées/sorties).

tcgetattr
int tcgetattr(int fd, struct termios *termios_p);
Description : Récupère les paramètres actuels d'un terminal.

Capacités du terminal

#include <term.h>

tgetent

int tgetent(char *bp, const char *name);

Description : Charge les informations de terminal à partir de la base de données des terminaux, utilisée pour des fonctionnalités comme les couleurs ou le positionnement du curseur.

tgetflag

int tgetflag(const char *id);

Description : Vérifie si une capacité de terminal est présente.

tgetnum

int tgetnum(const char *id);

Description : Récupère une valeur numérique associée à une capacité de terminal.

tgetstr

char *tgetstr(const char *id, char **area);

Description : Récupère une chaîne de caractères correspondant à une capacité de terminal.

tgoto

char *tgoto(const char *cap, int col, int row);

Description : Crée une séquence d'échappement pour positionner le curseur dans un terminal.

tputs

int tputs(const char *str, int affcnt, int (*putc)(int));

Description : Envoie une séquence d'échappement au terminal pour effectuer une action (comme déplacer le curseur).

Built-in

Echo

Affiche un texte ou une variable
echo "Hello" = affiche Hello dans le terminal
Différence entre avec ou sans double cote (“ ”) :
echo "Hello World" = affiche Hello World
echo Hello World = affiche Hello World
L'utilisation de > redirige le contenu de echo vers un fichier au lieu du terminal
echo Hello > fichier.txt = Créez le fichier.txt si il n'existe pas et inscrit Hello à l'intérieur, si le fichier existe et qu il a les permissions efface son contenu pour le remplacer par Hello
l'utilisation de >> permet de rediriger le contenu de echo vers la fin d'un fichier, sans écraser son contenu
Avec les | pour rediriger la sortie de echo vers une autre commande
echo test | cat -e = test$
echo error >&2 = redirige echo vers la sortie standard stderr
Options :
-n = permet de ne pas afficher de nouvelle ligne

Ne pas gérer :
Options :
-e = permet de prendre en compte les caractères d'échappement (\t, \n, \\, \a, \b, \r)
-E = permet de désactiver les caractères d'échappement(la commande echo le fait de base)

Cd

Fonctions utiles :
chdir = permet de changer de répertoire courant
getcwd = permet d'obtenir le chemin du répertoire courant

cd = retourne dans le répertoire Home/USER
cd . = permet de rester dans le répertoire courant
cd .. = - se déplace dans le répertoire supérieur
- une fois dans le répertoire racine '/' ne fait rien
cd ../.. = renvoie dans le répertoire supérieur du répertoire supérieur
cd + tab = 1 - montre tous les dossiers/fichiers du répertoire courant actuel
2 - permet la complétion du nom d'un dossier si il y a correspondance

A prendre en compte :

Quand cd est lancé et change de répertoire, actualisé aussi pwd
Vérifier que : - le répertoire existe
- le répertoire que l'on souhaite accéder, a les permissions nécessaires

Pwd

Fonction : getcwd pour récupérer le chemin
Affiche le chemin absolu du répertoire courant
pwd = /home/user/test

A prendre en compte :
le chemin peut être supprimé avec unset

Ne pas gérer :
Options
-L = Renvoie le chemin absolu en tenant compte des liens symbolique
-P = Renvoie le chemin absolu sans prendre en compte les liens symbolique

Export

export [VARIABLE] = [VALEUR]
export = renvoie toutes les variables d'environnements dans l'ordre alphabétique
permet de créer une variable ou d'en changer la valeur
La commande export permet de rendre une variable locale dans un shell disponible dans tous les sous-processus de ce shell.
Une variable exportée est disponible pour tous les sous-processus du shell dans lequel elle a été définie. Cela inclut :

- Des programmes exécutés directement dans le terminal.
- Des scripts exécutés à partir du shell.
- Des processus fils créés par le shell.

par contre elle ne sera pas disponible dans d'autres sessions

Unset

Permet de supprimer une variable ou une fonction
unset [VARIABLE] = supprime la variable de l'environnement

Env

env = renvoie toutes les variables d'environnement dans l'ordre de création
Exemples de variables d'environnement :

PATH : Contient une liste de répertoires dans lesquels le système cherche des programmes exécutables.
HOME : Contient le répertoire personnel de l'utilisateur courant.
USER : Contient le nom de l'utilisateur courant.
SHELL : Contient le chemin vers le shell en cours d'exécution.

A prendre en compte :
Une valeur ajoutée sans “=” dans l’environnement avec export ne sera pas visible dans env

Exit

exit = ferme le shell en cours
exit [CODE DE SORTIE] = ferme le shell avec CODE DE SORTIE
exit prend seulement un code erreur compris entre 0 et 255

Objectif de l'exécution

- copier l'environnement
- pouvoir lancer des commandes (echo, cd, pwd, export, unset, exit, env)
- gérer les chemins d'accès pour pouvoir trouver et exécuter les autres commandes
- utiliser des processus enfant avec fork et exec pour l'exécution des commandes
- prendre en charge les redirections (>, <, >>)
- les pipes (|)
- gestion des erreurs des commandes et des mauvaises exécutions.

Exec des commandes

Processus enfant/parent :
- fork, wait, waitpid, wait3, wait4
- dup, dup2, pipe
- Cmd : execve
- utiliser la fonction pipe() pour créer un tuyau d'écoute et d'écriture entre chaque entrée/sortie d'une commande et/ou d'un fichier
- créer une copie(enfant) du processus parent avec fork pour gérer l'exécution de chaque commande individuellement
- pour chaque commande, chercher le chemin d'accès, tester le chemin avec acces(), si acces = 0, alors le chemin est valide
- si il n'est pas valide ou que la commande n'existe pas utilisez exit() pour quitter le processus enfant tout en envoyant un message d'erreur
- si la commande existe et que le chemin est valide, envoyer le tout à execve() pour exécuter la commande
- répéter l'opération autant de fois que de commande

A prendre en compte :
Le programme doit tester toutes les commandes, par exemple si ma première commande est fausse, il devra quand même tester la deuxième,
le programme s'arrête seulement après avoir tester toutes les commandes et avoir atteint la sortie(STDOUT)

Heredoc

Le heredoc est un fichier temporaire permettant de noter des informations via le terminal, il prend comme argument un délimiter qui comme son nom l'indique
délimite son champ d'action, en effet une fois le délimiter atteint la ligne de commande s'exécutera

Les signaux

Signal : signal, sigaction, sigemptyset, sigaddset, kill, exit
ctrl -c = renvoie à une nouvelle ligne
ctrl -d = quitte le shell en cours
ctrl -\ = ne fais rien

Les redirections

> : redirige la sortie standard (stdout) vers un fichier.
< : redirige l'entrée standard (stdin) depuis un fichier.
>> : ajoute à un fichier existant.

Gestions des erreurs

Error : strerror, perror
Il y aura un message d'erreur si :
- la commande n'existe pas ou le chemin n'a pas d'accès = [NOM_DE_LA_CMD]: command not found
- l'option de la commande n'existe pas (ne pas gérer)
- le fichier d'entrée n'existe pas = bash: [NOM_DU_FICHIER]: No such file or directory
- le fichier d'entrée ou de sortie n'a pas les permissions = bash: [NOM_DU_FICHIER]: Permission denied

A prendre en compte :
- si le fichier d'entrée n'existe pas ou n'a pas les permissions, même si la première commande n'est pas valide, ne pas mettre de message d'erreur, uniquement la première commande
puisque la première commande sera "mal connecté" vu que l'entrée de sa pipe n'existe pas ou n'est pas valide, donc elle échoue silencieusement
- pareil si le fichier de sortie n'a pas les permissions, la dernière commande échouera silencieusement

Token

```
# define INPUT      1     //"<"
# define HEREDOC    2     //"<<"
# define TRUNC      3     //">"
# define APPEND     4     //">>"
# define PIPE       5     //"|"
# define CMD        6     //"|"
# define ARG        7     //"|"
```

Les chemins

Il y a 2 formes de chemins :
- le chemin relatif qui spécifie le chemin d'un fichier ou répertoire par rapport au répertoire courant
- le chemin absolue qui spécifie le chemin par rapport à la racine '/' du système

Environnement

Structure pour faire une copie de l'environnement :

```
typedef struct s_env {
  char  *current;
  struct s_env  *next;
}    t_env;
```

FD

entrée standard (fd = 0)
sortie standard (fd = 1)
sortie erreur (fd = 2)
& + un chiffre = descripteur de fichier si il existe

$

$? = renvoie le dernier code erreur exécuter
$variable = permet d'afficher le contenu d'une variable d'environnement

Infos en vrac

Ne pas gérer :
Lien symbolique
lien symbolique = fichier spécial qui agit comme un raccourci vers un fichier/dossier/répertoire
ln -s [chemin cible] [nom du lien symbolique] = créé un raccourci permettant d'accéder au [chemin cible]
ls -l [nom du lien symbolique] = permet de voir le raccourci et la ou il pointe
Attention, les liens symboliques sont sensibles à la casse, si le chemin n'existe plus ou a été modifié, le lien ne fonctionnera plus.