Minishell

As beautiful as a shell

Thobenel et Dpascal

Appréhender le projet :

Objectif ?

Objectif : recréé un Shell minimal qui imite le comportement de bash avec des fonctionnalités spécifique il sera lancer depuis le Shell classique et devra lancer un exec qui aura un PATH / un USER / un input et un output / ...

2. Plan d'attaque (plus de détail un peu plus bas)

1. entree user et prompt

lire une ligne avec readline et Gerer historique (add_history, rl_replace_line)

2. Commande interne (builtins) = Cd, echo, pwd, export, unset, env, exit
Connecter les commandes avec des pipes (|) et gérer les redirections (>, < , <<, >>).

3. Reactions à Ctrl-C, Ctrl-D, Ctrl-

4. Trouver et exécuter les programmes externes avec exècre (avec $PATH)

(Ma partie) Le parsing :

C'est la partie qui "lit" ce que l'utilisateur tape et prépare les commandes pour qu'elles soient compréhensibles par le programme. Elle sert à : Découper la commande en petits morceaux, appelés tokens (par exemple, séparer ls -l | grep txt en ls, -l, |, grep, et txt). Identifier les éléments spéciaux comme les redirection (>, <) ou les pipes (|), et les traiter correctement. Gérer les protections comme les guillemets (", ') ou les échappements (\) pour que les caractères spéciaux soient bien interprétés. Préparer les arguments et les options pour que la partie exécution sache quoi faire avec.

Pre ambule au man bash :

Le bash ou Bourne-Again SHell est un interpréteur de commande Shell compatible sh qui exec les commande lues depuis l'entrée standard (ton terminal quoi). Bash a été conçu pour être a la Norme POSIX par défaut

La norme POSIX l'objectif est de fournir un ensemble de normes pour les logiciels de système d'exploitation de type Unix. Elle vise à garantir la portabilité et la compatibilité entre les différents systèmes d'exploitation qui la suivent.

1. Les commandes

A. Type commande :
Une commande c'est une séquence de mot séparer par des espaces, en gros quand on fait une commande dans notre terminal, la commande peut différer comme
Simple = une seule commande suivis d'arguments
Compose = inclut des structures comme { liste; } ou (liste) (voir pt.2 GRAMMER)
Pipieline = plusieurs commandes reliées par |

B. Méta-caracteres :
C'est un caractères ayant une significations spécial :
| = pipe.
<, >, <<, >> = redirections
`, « , \ = protections des caractères spéciaux
C. Les opérateurs :
; = exécute les commandes successivement
& = exécute les commande en arrière plan
&& = exécute la commande suivant seulement si la précédente est réussit
|| = exécute la commande suivnte seulement si la pressente échoue

2. Grammaire de l'interpréteur

A. Commande simple :

Une commande simple est composée Dun nom suivi de ses arguments, séparer par des espaces.

B. Commande composée :
1 = (liste) = execute liste dans un sous shell
2 = { liste; } = execute liste dans l'environnement actuel du shell
3 = ((expression)) = évalue une expression arithmétique
4 = [[expression]] = teste une conditions (utiles pour les chaînes, fichier, etc …)
D. Pipeline
Syntax = command1 | command2
La sortie standard de command1 devient lente standard de command2

E. Redirections

1 = < = redirige l'entrée standard depuis un fichier
2 = > = redirige la sortie standard vers un fichier (écrasé le fichier)
3 = << = redirige la sortie standard vers un fichier (ajoute a la fin du fichier)
4 = >> = heredoc, lit jusqu'à une ligne contenant un délimiter spécifié

3. BUILTIN COMMANDS:

Commande obligatoire pour MiniShell :

1. Cd

Syntaxe = cd [chemin]

Sans arguments, retourne au répertoire défini par HOME

Variable associées = PWD (repo acutel), OLDPWD (precedent)

2. Echo

Syntaxe = écho [-n] [text...]

Options -n n'ajoute pas de saut a la ligne

3. Pwd

Affiche le repo actuel.

4. Export

Syntaxe = export VAR=valeur

Ajoute pou modifie une variable environnement

5. Unset

Syntaxe = unset VAR

Supprime une variable environnement

6. Env

Affiche les variables environnements

7. Exit

Syntaxe = exit [code]

Quitte le shell avec le code de retour

4. Redirections

Types de redirections :

1. Entree

< fichier = lit depuis le fichier spécifié au lieu de l'entrée standard

2. Sortie

> fichier = envoie la sortie standard vers un fichier (ecrase le contenue)

3. Heredoc

<< délimiter = lit l'entrée jusqu'à ce que le délimiter soit rencontrer

4. Files descriptor

>> = redirige la sortie d'erreur standard.

5. Expansion

Variable environnement :

$VAR = développe la valeur de la variables VAR

${VAR} = forme étendue pour eviterles ambiguïtés (utile si VAR est suivi d'un texte colle)

parametre spéciaux :

$? = code de retour de la dernier commande exécutée

$$ = PID du shell en cours

$# = nombre total d'arguments positionnels

$@ = liste tous les arguments positionnels (séparer individuellement)

$* = liste tous les arguments positionnels (dans une seul chaine)

Substituions de commandes :

'Command' ou $(commande) : exécuté une commande et remplace son résultat,

6. Quoting

Proteger des caractères spéciaux :

1 = apostrophe simple (') = tout ce que entre ' est littoral (aucune expression, meme $ n'est pas interprété)

2 = double Guillet («) = potefent la pluspart des caractère sanf $, \ et '

3 = echappement (\) = échappe une seul caravelle pour qu'ils soit interprété littéralement

7.Signal

CTRL-C = interrompt la commande en cours, doit afficher un nouveau promt

CTRL-D = termine le shell si aucune commande nest en cours

Ctrl+ = ne doit rien faire dans un minishell

o

8. Environnement

Variable clés :

PATH = répertoire pour rechercher les exécutable

PWD = repertoire de travail actuel

OLDPWD = dernier répertoire de travail

HOME = répertoire actuel

9. Heredoc

Todo ...

3. Fonction autorisée :

Fonction liée a readline

char *readline(const char *prompt);

Lit une ligne de commande depuis le terminal avec un prompt.

Retourne : La ligne lue, allouée dynamiquement (à libérer avec free()).

Si prompt == NULL ou la chaîne est vide, aucun prompt n'est affiché.

En cas de EOF, retourne NULL.

void rl_clear_history(void);

Efface l'historique des commandes et libère la mémoire associée.

int rl_on_new_line(void);

Indique qu'une nouvelle ligne est en cours dans l'interface readline.

void rl_replace_line(const char *text, int clear_undo);

Remplace le contenu de la ligne actuelle dans le buffer de readline.

int rl_redisplay(void);

Réaffiche la ligne actuelle dans le terminal, utile après modification du texte.

void add_history(const char *line);

Ajoute une ligne à l'historique de readline.

Fonctions gestion mémoire entrée\sortie.

Ces fonctions gèrent la mémoire dynamique et les opérations de base sur les fichiers.

```
void *malloc(size_t size);
```

Alloue dynamiquement une mémoire de taille spécifiée et retourne un pointeur vers cette mémoire.

```
void free(void *ptr);
```

Libère la mémoire précédemment allouée par malloc.

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

Écrit des données dans un descripteur de fichier (par exemple, le terminal ou un fichier).

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

Vérifie si un fichier ou un répertoire existe et ses permissions.

```
int open(const char *pathname, int flags, mode_t mode);
```

Ouvre un fichier et retourne un descripteur de fichier.

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

Lit des données à partir d'un descripteur de fichier.

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

Ferme un descripteur de fichier.

Gestion des processus et des signaux :

Ces fonctions permettent de créer des processus, d'attendre leur exécution, et de gérer les signaux.

```
pid_t fork(void);
```

Crée un nouveau processus en dupliquant le processus appelant. Retourne le PID du processus enfant.

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

Attend la fin d'un processus enfant.

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

Attend un processus enfant spécifique.

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

Attend la fin d'un ou plusieurs processus enfants et recueille des informations d'utilisation des ressources.

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

Similaire à wait3 mais pour un processus spécifique.

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

Définit un gestionnaire pour un signal particulier.

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

Configure un gestionnaire de signaux avec des options avancées.

```
int sigemptyset(sigset_t *set);
```

Initialise un ensemble de signaux vide.

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

Ajoute un signal à un ensemble de signaux.

```
int kill(pid_t pid, int sig);
```

Envoie un signal à un processus.

```
void exit(int status);
```

Termine l'exécution du programme.

Gestion des fichiers et des répertoires

Ces fonctions fournissent des informations sur les fichiers et gèrent les répertoires.

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

Retourne le chemin absolu du répertoire de travail actuel.

```
int chdir(const char *path);
```

Change le répertoire de travail.

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

Renvoie des informations sur un fichier.

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

Similaire à stat, mais traite les liens symboliques.

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

Renvoie des informations sur un fichier ouvert.

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

Supprime un fichier.

Fonctionnalités avancées

Ces fonctions gèrent des aspects spécifiques comme les terminaux et les environnements.

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

Récupère la valeur d'une variable d'environnement.

```
int dup(int oldfd);
```

Duplique un descripteur de fichier.

```
int dup2(int oldfd, int newfd);
```

Duplique un descripteur de fichier vers un autre, avec remplacement.

```
int pipe(int pipefd[2]);
```

Crée un canal de communication entre processus (pipe).

```
DIR *opendir(const char *name);
```

Ouvre un répertoire.

```
struct dirent *readdir(DIR *dirp);
```

Lit une entrée (fichier ou dossier) dans un répertoire.

```
int closedir(DIR *dirp);
```

Ferme un répertoire.

Gestion des terminaux

Ces fonctions manipulent les terminaux pour configurer les entrées/sorties.

```
int isatty(int fd);
```

Vérifie si un descripteur correspond à un terminal interactif.

```
char *ttyname(int fd);
```

Renvoie le nom du terminal associé à un descripteur.

```
int ioctl(int fd, unsigned long request, ...);
```

Permet des manipulations bas niveau des périphériques comme les terminaux.

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

Récupère les paramètres d'un terminal.

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

Configure les paramètres d'un terminal.

Fonctions de gestions des erreurs

```
char *strerror(int errnum);
```

Retourne une chaîne décrivant une erreur à partir d'un code errno.

```
void perror(const char *s);
```

Affiche un message d'erreur avec le contenu de errno.

4. Token :

Les tokens sont les composants de base d'une commande, comme les mots, les opérateurs (|, < , >, etc ...), ou les espaces. Lors du parsing, je vais devoir analyser et organiser ces tokens pour construire une structure logique (par exemple, un arbre ou une liste chaînées) représentant la commande.
Pour les analyser on peut commencer par :

1) lire la ligne entrée avec readline()
2) Diviser la ligne en tokens

Les token peuvent être extrait en utilisant des séparateurs comme les espaces ou (<, >, |, etc ...). Une fonction comme strtok ou une analyse manuel avec un pointeur peut être efficace.

3) identifier les types de tokens

Mot simple = (e.g., ls, echo)
Operateur = (|, <, >, <<, >>)
Variables = ($VAR)
Quotes = (', ").
Caractere spéciaux a gerer = (;, &, etc ...)

4) idee ??

Cree une structure pour représenter la valeur et le type (type de token = mot ...) et un ptr vers le next pour avancer

3. Commande quitter :

Check les CTRL-D CTRL-C CTRL-\ (return specific signaux)
CTRL-D = Ne gener pas de signal mais indique une fin de fichier (EOF). Dans un shell cela doit fermer le shell interactif si aucune commande n'est saisie.
CTRL-C = Envoie le signal SIGINT. Ca interrompt un processus en cours.
CTRL- = envoie le signal SIGQUIT. Cela interrompt un processus mais affiche en plus un *core-dump* (non nécessaire pour minishell)
Idee ?? Intercepter les signaux avec signal ou sigaction:
Signal est simple mais moins flexible|| Sigaction permet une meilleur gestions

Usuelles fonction = Strace

.... todo