📚 Funciones Autorizadas para Minishell

📖 Índice

🔤 Lectura/Escritura

readline

Prototipo:

```c
char *readline(const char *prompt);
```

Descripción	**Lee una línea completa desde stdin con edición y gestión de historial**
Entrada	`prompt`: String mostrado antes de leer (ej: "minishell> ")
Salida	Puntero a string con la línea leída, o NULL en EOF/error

💡 **Uso en Minishell:**
Leer los comandos del usuario en el bucle principal del shell. Muestra el prompt "minishell> " y permite al usuario escribir comandos con edición de línea. Detecta Ctrl+D para salir del shell.

---

write

Prototipo:

```c
```

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

Descripción	Escribe datos en un file descriptor
Entrada	`fd`: Descriptor (1=stdout, 2=stderr), `buf`: datos, `count`: bytes
Salida	Número de bytes escritos (≥0), o -1 si error

💡 **Uso en Minishell:**
Escribir mensajes de error personalizados en stderr cuando un comando falla. También se usa para escribir en archivos durante redirecciones de salida. Útil para escribir en pipes entre comandos.

`read`

Prototipo:

```
c

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

Descripción	Lee datos desde un file descriptor
Entrada	`fd`: Descriptor origen, `buf`: buffer destino, `count`: bytes máx
Salida	Bytes leídos (≥0), 0 si EOF, -1 si error

💡 **Uso en Minishell:**
Leer contenido de archivos durante redirecciones de entrada. Procesar el contenido de heredocs almacenados temporalmente. Leer desde pipes en comandos encadenados.

📁 Gestión de Archivos

`open`

Prototipo:

```
c

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

Descripción	Abre un archivo y retorna su file descriptor
Entrada	`pathname`: ruta, `flags`: modo apertura, `mode`: permisos si crea (ej: 0644)
Salida	File descriptor (≥0) o -1 si error

💡 **Uso en Minishell:**
Abrir archivos durante redirecciones de salida (>) con permisos 0644. Abrir archivos de entrada (<) en modo solo lectura. Crear archivos temporales para heredocs. Gestionar redirecciones de append (>>).

`close`

Prototipo:

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

Descripción	**Cierra un file descriptor**
Entrada	`fd`: File descriptor a cerrar
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Cerrar descriptores después de hacer dup2 en redirecciones. Cerrar ambos extremos de pipes tras configurar comunicación entre procesos. Cerrar archivos temporales de heredocs. Limpiar descriptores al finalizar comandos.

`access`

Prototipo:

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

Descripción	**Verifica permisos de acceso a un archivo**
Entrada	`pathname`: ruta, `mode`: tipo acceso (F_OK, R_OK, W_OK, X_OK)
Salida	0 si acceso permitido, -1 si denegado o error

💡 **Uso en Minishell:**
Verificar si un comando existe en las rutas del PATH antes de intentar ejecutarlo. Comprobar permisos de ejecución con X_OK. Validar que archivos de entrada existen antes de redirigir.

`unlink`

Prototipo:

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

Descripción	**Elimina un archivo del sistema**
Entrada	`pathname`: Ruta del archivo a eliminar
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Eliminar archivos temporales creados para heredocs después de su uso. Limpiar archivos auxiliares que el shell crea durante su ejecución.

`dup` / **`dup2`**

Prototipo:

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

Descripción	**Duplican file descriptors (dup2 permite especificar el nuevo)**
Entrada	`oldfd`: fd a duplicar, `newfd`: fd destino (solo dup2)
Salida	Nuevo file descriptor o -1 si error

💡 **Uso en Minishell:**
Guardar copias de stdin/stdout originales con dup para restaurar después de redirecciones. Redirigir stdout a un archivo con dup2. Conectar procesos mediante pipes duplicando descriptores. Restaurar descriptores estándar tras ejecutar comandos.

`pipe`

Prototipo:

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

Descripción	Crea una tubería para comunicación entre procesos
Entrada	`pipefd`: Array donde guardar fds (pipefd[0]=lectura, pipefd[1]=escritura)
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Implementar pipelines entre comandos (ls | grep txt). El proceso que escribe usa pipefd[1] como stdout, el que lee usa pipefd[0] como stdin. Cada proceso cierra el extremo que no necesita.

⚙️ Procesos

`fork`

Prototipo:

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

Descripción	Crea un proceso hijo duplicando el actual
Entrada	Ninguna
Salida	0 en hijo, PID del hijo en padre, -1 si error

💡 **Uso en Minishell:**
Crear procesos hijo para ejecutar comandos externos sin afectar el proceso principal del shell. El hijo ejecuta el comando con execve, el padre espera con wait. Necesario para cada comando en un pipeline.

`wait` / **`waitpid`** / **`wait3`** / **`wait4`**

Prototipo:

```c
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

Descripción	Esperan finalización de procesos hijo y obtienen su estado
Entrada	`pid`: proceso específico o -1, `status`: estado salida, `options`: WNOHANG, etc.
Salida	PID del hijo terminado o -1 si error

💡 **Uso en Minishell:**
Esperar a que comandos externos terminen antes de mostrar nuevo prompt. Obtener código de salida del comando para actualizar la variable $?. Usar waitpid con WNOHANG para no bloquear en pipelines complejos.

`execve`

Prototipo:

```c
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Descripción	**Ejecuta un programa reemplazando la imagen del proceso**
Entrada	`pathname`: ruta ejecutable, `argv`: argumentos, `envp`: entorno
Salida	No retorna si éxito, -1 si error

💡 **Uso en Minishell:**
Ejecutar comandos externos en el proceso hijo creado con fork. Se llama después de configurar redirecciones y pipes. Si execve falla (comando no encontrado), el hijo debe terminar con exit(127).

`exit`

Prototipo:

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

Descripción	**Termina el proceso con código de salida**
Entrada	`status`: Código de salida (0=éxito)
Salida	No retorna (termina proceso)

💡 **Uso en Minishell:**
Implementar el builtin "exit" que termina el shell con un código específico. Terminar procesos hijo cuando execve falla. Salir del shell cuando se detecta EOF (Ctrl+D).

📂 Directorios

`getcwd`

Prototipo:

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

Descripción	Obtiene el directorio de trabajo actual
Entrada	`buf`: buffer destino (NULL para malloc automático), `size`: tamaño
Salida	Puntero a la ruta o NULL si error

💡 **Uso en Minishell:**
Implementar el builtin "pwd" que muestra el directorio actual. Actualizar la variable de entorno PWD después de cambiar de directorio con cd. Guardar el directorio actual antes de ejecutar cd para OLDPWD.

`chdir`

Prototipo:

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

Descripción	Cambia el directorio de trabajo actual
Entrada	`path`: Ruta del nuevo directorio
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Implementar el builtin "cd" para cambiar de directorio. Manejar casos especiales como "cd" sin argumentos (ir a HOME), "cd ~" (expandir a HOME), "cd -" (ir a OLDPWD). Actualizar variables PWD y OLDPWD tras el cambio.

`opendir` / **`readdir`** / **`closedir`**

Prototipo:

```c
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

Descripción	Abren, leen y cierran directorios para listar contenido
Entrada	`name`: ruta directorio, `dirp`: directorio abierto
Salida	opendir: DIR* o NULL; readdir: dirent* o NULL; closedir: 0 o -1

💡 **Uso en Minishell:**
Expandir wildcards/globbing como "*.c" listando archivos del directorio actual. Implementar funcionalidad similar a "ls" si se requiere. Filtrar archivos ocultos (que empiezan con .) según sea necesario.

🚦 Señales

`signal`

Prototipo:

```c
sighandler_t signal(int signum, sighandler_t handler);
```

Descripción	Establece manejador para una señal
Entrada	`signum`: señal (SIGINT, SIGQUIT, etc.), `handler`: función o SIG_IGN/SIG_DFL
Salida	Manejador anterior o SIG_ERR si error

💡 **Uso en Minishell:**
Configurar comportamiento de Ctrl+C (SIGINT) para mostrar nuevo prompt sin terminar el shell. Ignorar Ctrl+\ (SIGQUIT) en modo interactivo. Restaurar señales a comportamiento predeterminado en procesos hijo.

`sigaction`

Prototipo:

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

Descripción	Alternativa robusta a signal para manejar señales
Entrada	`signum`: señal, `act`: nueva acción, `oldact`: acción anterior
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Configurar manejo de señales de forma más confiable que signal. Usar flag SA_RESTART para reiniciar

syscalls interrumpidas como readline. Preferible a signal en implementaciones modernas.

`sigemptyset` / **`sigaddset`**

Prototipo:

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

Descripción	**Manipulan conjuntos de señales para sigaction**
Entrada	`set`: conjunto de señales, `signum`: señal a añadir
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Inicializar máscaras de señales vacías antes de configurar sigaction. Bloquear temporalmente señales durante operaciones críticas que no deben ser interrumpidas. Restaurar máscaras después de completar operaciones.

`kill`

Prototipo:

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

Descripción	**Envía una señal a un proceso**
Entrada	`pid`: proceso objetivo, `sig`: señal a enviar
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Terminar procesos hijo cuando el shell recibe Ctrl+C. Enviar SIGTERM o SIGKILL a procesos que deben terminar. Útil para implementar timeout de comandos si se requiere.

💻 Terminal

`isatty`

Prototipo:

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

Descripción	Verifica si un descriptor está asociado a terminal
Entrada	`fd`: File descriptor a verificar
Salida	1 si es terminal, 0 si no lo es

💡 **Uso en Minishell:**
Detectar si el shell se ejecuta interactivamente o recibe entrada de archivo/pipe. Mostrar prompt solo cuando stdin es terminal interactiva. Decidir si usar colores en la salida basándose en si stdout es terminal.

`ttyname` / **`ttyslot`**

Prototipo:

```c
char *ttyname(int fd);
int ttyslot(void);
```

Descripción	ttyname: obtiene nombre del terminal; ttyslot: obtiene slot en utmp
Entrada	`fd`: descriptor del terminal (solo ttyname)
Salida	ttyname: nombre (ej: "/dev/pts/0") o NULL; ttyslot: número o 0

💡 **Uso en Minishell:**
Implementar un comando "tty" que muestra el terminal actual. Útil para debugging y mostrar información del entorno. Generalmente no crítico para funcionalidad básica del shell.

`ioctl`

Prototipo:

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

Descripción	Control de dispositivos I/O, usado para operaciones de terminal
Entrada	`fd`: descriptor, `request`: operación (ej: TIOCGWINSZ), `...`: args
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Obtener dimensiones de la ventana del terminal con TIOCGWINSZ. Útil para formatear salida de comandos que necesitan ajustarse al ancho de terminal. Detectar cambios de tamaño de ventana.

`tcsetattr` / `tcgetattr`

Prototipo:

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

Descripción	**Obtienen/establecen atributos del terminal**
Entrada	`fd`: terminal, `optional_actions`: cuándo aplicar, `termios_p`: config
Salida	0 si éxito, -1 si error

💡 **Uso en Minishell:**
Modificar modo del terminal para desactivar echo al leer contraseñas. Configurar modo raw vs canonical según necesidades. Guardar y restaurar configuración original del terminal al salir.

`tgetent` / `tgetflag` / `tgetnum` / `tgetstr` / `tgoto` / `tputs`

Prototipo:

```c
int tgetent(char *bp, const char *name);
int tgetflag(char *id);
int tgetnum(char *id);
char *tgetstr(char *id, char **area);
char *tgoto(const char *cap, int col, int row);
int tputs(const char *str, int affcnt, int (*putc)(int));
```

Descripción	**Funciones termcap para control avanzado de terminal**
Entrada	Varios según función (term name, capability id, coordenadas, etc.)
Salida	Varía: flags, números, strings, códigos de resultado

💡 **Uso en Minishell:**
Limpiar pantalla de forma portable usando capacidades termcap. Mover cursor a posiciones específicas. Implementar características avanzadas de visualización si se requieren (colores, formato).

🌍 Variables de Entorno

`getenv`

Prototipo:

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

Descripción	**Obtiene valor de variable de entorno**
Entrada	`name`: Nombre de la variable (ej: "PATH", "HOME")
Salida	Valor de la variable o NULL si no existe

💡 Uso en Minishell:

Buscar comandos recorriendo las rutas en PATH. Expandir ~ a directorio HOME. Obtener valor de variables para expansión en comandos. Inicializar variables del shell al arrancar.

🧠 Memoria

`malloc` / **`free`**

Prototipo:

```c
void *malloc(size_t size);
void free(void *ptr);
```

Descripción	**Asignan y liberan memoria dinámica**
Entrada	`size`: bytes a asignar (malloc), `ptr`: puntero a liberar (free)
Salida	malloc: puntero o NULL; free: void

💡 Uso en Minishell:

Reservar memoria para tokens del parser. Crear arrays dinámicos de argumentos. Almacenar strings de comandos y paths. Liberar memoria de readline. Gestionar listas de variables de entorno. Fundamental para evitar memory leaks.

🛠️ Utilidades

`perror`

Prototipo:

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

Descripción	**Imprime mensaje de error del sistema en stderr**
Entrada	`s`: Prefijo para el mensaje de error
Salida	void (imprime en stderr)

💡 **Uso en Minishell:**
Mostrar mensajes de error descriptivos cuando fallan syscalls (open, fork, etc.). Preceder el error del sistema con contexto del comando que falló. Ayudar al usuario a entender qué salió mal.

`strerror`

Prototipo:

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

Descripción	**Retorna string describiendo código de error**
Entrada	`errnum`: Número de error (generalmente errno)
Salida	String con descripción del error

💡 **Uso en Minishell:**
Construir mensajes de error personalizados combinando contexto del shell con descripción del error del sistema. Alternativa a perror cuando se necesita más control sobre el formato del mensaje.

Macros `WIFEXITED` / `WEXITSTATUS` / `WIFSIGNALED` / `WTERMSIG`

Prototipo:

```c
```

```
int WIFEXITED(int status);
int WEXITSTATUS(int status);
int WIFSIGNALED(int status);
int WTERMSIG(int status);
```

Descripción	Macros para interpretar el status de wait/waitpid
Entrada	`status`: Valor obtenido de wait/waitpid
Salida	WIFEXITED/WIFSIGNALED: bool; WEXITSTATUS/WTERMSIG: int

💡 **Uso en Minishell:**
Determinar si comando terminó normalmente o por señal. Extraer código de salida del comando para actualizar $?. Si terminó por señal, calcular exit status como 128 + número de señal. Distinguir entre errores del comando y terminación forzada.

`rl_clear_history` / **`rl_on_new_line`** / **`rl_replace_line`** / **`rl_redisplay`** / **`add_history`**

Prototipo:

```
c

void rl_clear_history(void);
void rl_on_new_line(void);
void rl_replace_line(const char *text, int clear_undo);
void rl_redisplay(void);
void add_history(const char *line);
```

Descripción	Funciones de readline para gestión de historial y display
Entrada	Varía: líneas de texto, flags
Salida	void

💡 **Uso en Minishell:**
Añadir comandos ejecutados al historial con add_history. Limpiar línea actual cuando llega Ctrl+C con rl_replace_line. Redibujar prompt con rl_redisplay. Gestionar visualización correcta del prompt tras señales.

📋 Resumen de Uso por Componente

Componente	Funciones Clave
Lexer/Parser	readline, malloc, free
Redirecciones	open, close, dup2, access

Componente	Funciones Clave
Pipes	pipe, dup2, close, fork
Ejecución	fork, execve, wait/waitpid, access, getenv
Builtins	chdir, getcwd, getenv, exit, write
Señales	signal/sigaction, sigemptyset, sigaddset
Variables	getenv, expansión con malloc
Heredoc	open, write, read, unlink
Terminal	isatty, tcgetattr, tcsetattr