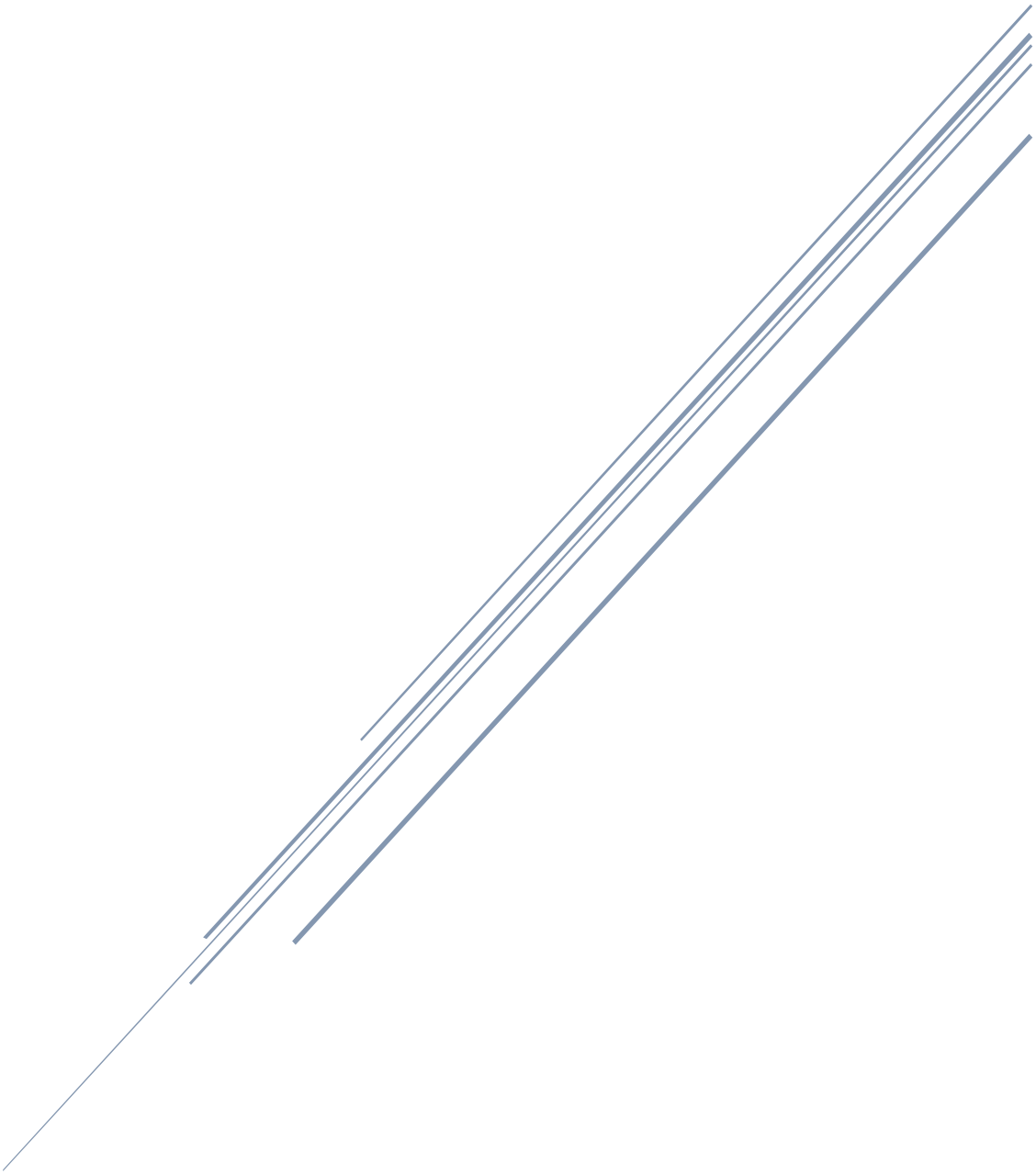
MINISHELL

Práctica obligatoria 2



Grado en Ingeniería del Software
Sistemas Operativos

Índice

AUTORAS

Arshia Ambar Saleem

Sandra Cañadas Gómez

MYSHELL.C

Archivo con el código C correspondiente al minishell. Queremos implementar un programa que actúe como intérprete de mandatos, debe implementar y ejecutar mandatos leyéndolos de la entrada estándar.

Incluimos en nuestro código los siguientes archivos:

- *stdio.h* para poder utilizar las funciones *fgets, fprintf, fileno* y *printf* junto con las variables *stderr, stdout* y *stdin*.
- *stdlib.h* para poder utilizar las funciones *getenv* y *exit*.
- *unistd.h* para poder utilizar las funciones *chdir, getcwd, fork, execvp, pipe, dup, dup2* y *close*.
- *string.h* para utilizar las funciones *strcmp* y *strerror*.
- *errno.h* biblioteca para controlar los errores y convertirlos en algo legible y para poder usar la variable *errno.*
- *parser.h* para poder utilizar la función *tokenize* y dos tipos de datos *tline* y *tcommand.*
- *sys/types.h* para poder utilizar el tipo de dato *pid_t*
- *sys/wait.h* para poder utilizar la función *wait.*
- *fcntl.h* para poder utilizar las funciones *open* y *creat* y para las variables *O_RDONLY, S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH y S_IWOTH*
- *signal.h* para poder utilizar las constantes *SIGINT* y *SIGQUIT,* para usar *SIG_IN* y *SIG_DFL* y para usar la función *signal.*

Las funciones nos devolverán un código de error en una variable global, por ello hemos declarado la variable externa *errno*.

Para controlar todos los errores utilizamos *stderr* que es un puntero a FILE que referencia la salida de error estándar, normalmente el monitor. También utilizamos la función *strerror* a la cual pasamos la variable *errno*, esta función convierte el código de error a una cadena de texto más explicativa.

Otros variables globales que hemos utilizados son *back_process* que es un array para guardar el pid de los procesos en background y otro *num_back* que es un contador de procesos de background.

Función nozombie():

Hemos implementado esta función para evitar la creación de procesos zombies cuando se terminan los procesos en background.

Función mycd()

Hemos implementado una función para ejecutar el mandato cd. Permite el acceso a través de rutas absolutas como relativas, además de la posibilidad de acceder al directorio especificado

en la variable HOME si no recibe ningún argumento, escribiendo la ruta absoluta del nuevo directorio actual del trabajo.

Lo primero que hacemos es declarar las variables que vamos a utilizar: *dir* que es un puntero que contiene la dirección de memoria de un *char* y *buf* que será nuestro array de caracteres.

Comprobamos si han introducido el comando cd sin argumentos, si es así *dir* apuntará a HOME y si no apuntará al directorio introducido, después comprobamos si el directorio introducido es un directorio o no, en caso de que no lo sea mostrará un mensaje de error, y si lo es nos lleva al directorio elegido.

Esta función ha sido la primera que hemos realizado, no nos ha resultado complicada y no hemos tardado mucho en hacerlo debido a que ya la habíamos realizado con anterioridad en uno de los ejercicios de clase.

Función myothercommands()

Hemos implementado una función para ejecutar el resto de los comandos (uno, dos o más).

Lo primero es para un solo comando, comprobamos que así sea y si el comando es válido. Si el comando no es válido imprimirá un mensaje de error. Si el comando es válido:

> Declaramos las variables que vamos a utilizar: *pid* que es el identificador de proceso y *status* en la cual almacenaremos el valor de retorno de la función *wait*.
>
> Creamos un nuevo proceso utilizando la función *fork*, en caso de error la variable *pid* será menor que cero y se mostrará en pantalla un mensaje de error.
>
> Si *pid* es igual a cero tendremos al proceso hijo en el cual se ejecutará el comando, en caso de error a la hora de ejecutarlo se imprimirá un mensaje.
>
> Si *pid* es mayor que cero tendremos al proceso padre que está esperando por el proceso hijo.

Después tenemos para dos comandos, comprobamos que así sea.

> Declaramos las variables que vamos a utilizar: *p[2]* que vamos a utilizar para escritura y lectura en pipe, *pid* que es el identificador de proceso.
>
> Creamos los pipes utilizando la función *pipe(p)*.
>
> Creamos un nuevo proceso que será el primer hijo utilizando la función *fork()*, en caso de error la variable *pid* será menor que cero y se mostrara en pantalla un mensaje de error.
>
> Si *pid* es igual a cero tendremos al primer hijo (proceso), cerramos su *pipe* de lectura y redirigimos su salida al *pipe* de escritura del siguiente proceso usando *dup2*.
>
> Comprobamos que el mandato sea válido, en caso de error imprimirá un mensaje, si todo va bien lo ejecuta, si hay algún problema al ejecutar imprimirá un mensaje de error.
>
> Si *pid* es mayor que cero, crearemos un nuevo proceso (segundo hijo), cerramos su *pipe* de escritura ya que solo redirige su entrada hacia su *pipe* de lectura.

Comprobamos que el mandato sea válido, en caso de error imprimirá un mensaje, si todo va bien lo ejecuta, si hay algún problema al ejecutar imprimirá un mensaje de error.

Por último, creamos el proceso padre que cierra todos los pipes y espera hasta que termine su proceso hijo.

Por últimos tenemos para más de dos comandos, comprobamos que sea así:

Declaramos las variables que vamos a utilizar que son*: p[line->ncommands][2]* que será descriptor de archivo (file descriptors), hasta n comandos para acceder a pipe, uno para escribir y otro para leer y *pid* que es el identificador de proceso

Hacemos un bucle *for*, creamos pipes hasta n comandos con *pipe(p),* luego creamos hijo usando *fork().*

En ambos procesos verificamos si es un hijo o un padre, si es un padre continuamos con la ejecución, en caso de un hijo cerramos todos los pipes excepto su propio pipe de lectura y pipe de escritura del siguiente proceso.

Para el primer hijo cerramos su pipe de lectura *close(p[i][0])* y redirigimos su salida al pipe de escritura del siguiente proceso usando *dup2(p[i+1][1],1),* en caso de último hijo, hemos cerrado su pipe de escritura, ya que como solo va a redirigir su entrada hacia su pipe de lectura *dup2(p[i][0],0).* Para cualquier otro hijo, redirigimos el *stdout* al pipe del siguiente proceso y el *stdin* a su pipe de lectura *dup2(p[i][0],0)*, *dup2(p[i+1][1],1)*.

Cada hijo va a ejecutar su comando correspondiente, utilizando *execvp* y va a esperar para que se termine el hijo anterior para poder leer su salida.

Por último, creamos el proceso padre que cierra todos los pipes hasta n comandos y se espera a los hijos.

Luego se comprueba si se ha introducido la *&* (background), en caso de haberse introducido, se mostrará el pid del proceso y se guardará el pid en un array *back_process*.

Una vez que sale del bucle *for*, comprueba que se haya introducido la *&* (background), en caso de ser así imprime entre corchetes el *pid*.

Esta función nos ha costado mucho más, sobre todo para mas de dos comandos, en un principio la hicimos de otra manera, pero se nos salía de nuestra Shell, no éramos capaces de ver el por qué, probamos cambiando, quitando y añadiendo, distintas cosas y ni por esas, le preguntamos a Alberto y nos explicó el por qué, nos dijo que probáramos quitando una cosa pero aun así nada, nos quedábamos sin tiempo, teníamos aun que hacer lo de backgroud aun y más prácticas, así que al final decidimos hacerlo de otra manera (como está ahora). Quitando eso, todo lo demás bien.

Main()

Esta función lo que hace es llamar a las funciones implementadas más arriba.

Lo primero es declarar las variables que vamos a utilizar: *buf* que será nuestro array de caracteres, *line* para poder acceder al resto de datos de tipo *tline* y *fd* es el descriptor de fichero del fichero que se va a tratar.

Lo siguiente que hacemos es declarar tres variables e inicializarlas: *guardar_redirect_input*, *guardar_redirect_output* y *guardar_redirect_error*, lo hacemos para guardar la entrada, salida y error estándar para las redirecciones.

Imprimimos el *prompt* y a continuación utilizamos un *while* para leer todo lo que se escriba por pantalla hasta que se pulse CTRL+C.

Utilizamos la función *tokenize* que recibe como argumento un puntero a una cadena de caracteres, y devuelve un puntero a una variable de tipo *tline* que contiene la información sobre la cadena analizada (*buf*) que contendrá los comandos.

> Si es igual a NULL continuamos.
>
> Si la línea tiene redirección de entrada se abrirá el fichero para lectura, si el descriptor de fichero es menos uno quiere decir que ha habido un fallo al abrir el fichero e imprimirá un mensaje de error, si todo ha salido bien redirigimos la entrada estándar desde archivo.
>
> Si la línea tiene redirección de salida se crea el fichero, si el descriptor de fichero es menos uno quiere decir que ha habido un fallo al crear el fichero e imprimirá un mensaje de error, si todo ha salido bien redirigimos la salida estándar al archivo.
>
> Si la línea tiene redirección de error ocurre lo mismo que con la redirección de salida solo que en vez de redirigir la salida estándar redirige el error estándar.
>
> Si la línea contiene *&* (background), utilizaremos la función *signal* para que cuando recibamos por teclado la señal *SIGINT (Ctrl-C)* o *SIGQUIT (Ctrl-\)* sean ignoradas (*SIG_IGN*), porque no queremos que la minishell ni los procesos finalicen al recibir dichas señales.
>
> Si no contiene & utilizaremos la función *signal* para que cuando recibamos por teclado la señal *SIGINT (Ctrl-C)* o *SIGQUIT (Ctrl-\)* realice la acción por defecto (*SIG_DFL*).
>
> Si el comando introducido es *cd* llamamos a la función *mycd* a la cual pasamos la variable *line*.
>
> Si no es comando *cd* ni *jobs* llamamos a la función *myothercommands* a la cual pasamos las variables *line*, *redirect_input, redirect_output* y *redirect_error*.
>
> Si el comando introducido es *jobs* se mostrará array de pid de los procesos que están ejecutando en background.
>
> Después tenemos tres *if* en los cuales se restablecen las redirecciones, en cada uno una redirección.

Por último, volvemos a imprimir el *prompt.*

Nos ha resultado bastante sencillo debido a que teníamos un main de ejemplo de los profesores, lo único que nos ha dado un poco de problemas han sido las redirecciones, pero al final lo hemos logrado.

Esta práctica, en general, nos ha llevado mucho más tiempo que la primera, nos ha costado bastante más, sobre todo la parte de más de dos comandos